Creating a Django Project

```
cd some_folder/folder_you_want_to_store_your_project_in
django-admin startproject your_project
cd your_project
your_project> python manage.py runserver
mkdir apps
cd apps
apps> python ../manage.py startapp your_app_name
cd app_name
touch urls.py
```

In a text editor find settings.py and then find the variable INSTALLED_APPS

```
INSTALLED_APPS = [
  'apps.your_app_name_here', # added this line
  'django.contrib.admin',
  'django.contrib.auth',
  'django.contrib.contenttypes',
  'django.contrib.sessions',
  'django.contrib.messages',
  'django.contrib.staticfiles'
]
```

In the project level urls file add the root route to the urlpatterns file

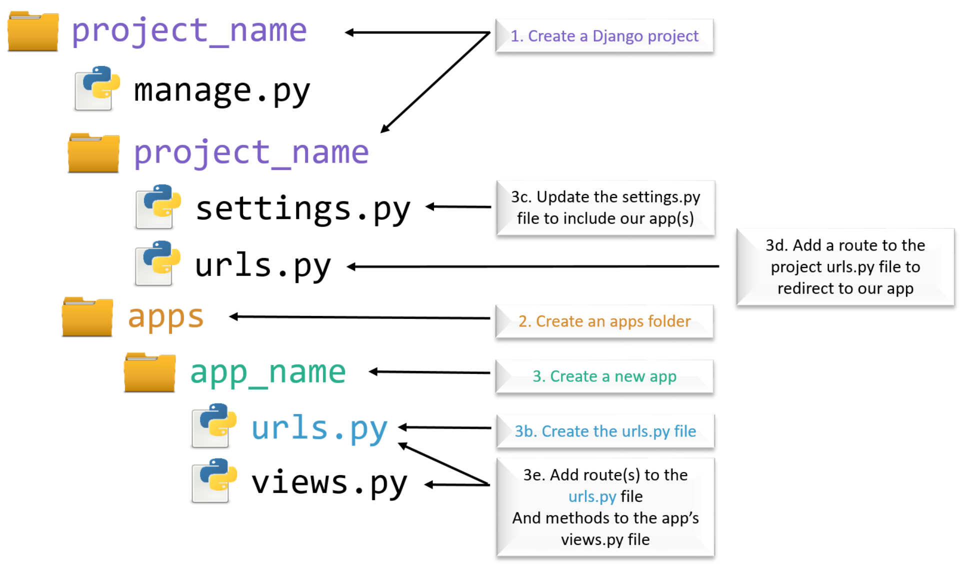
```
from django.urls import path, include
urlpatterns = [
    path('', include('apps.app_one.urls'))
]
```

In our app for every route that we want to add
1. in the app level urls.py file add a url pattern

```
from django.urls import path
from . import views

urlpatterns = [
    path("", views.index)
]
```

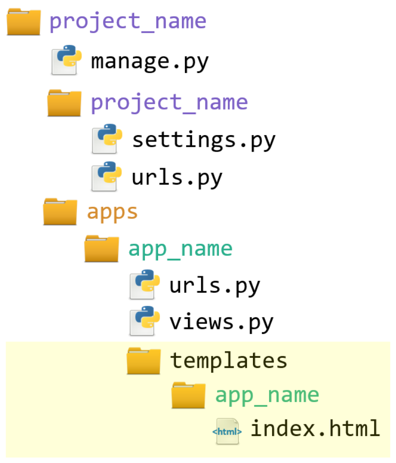
2. In your app's views.py file

```
from django.shortcuts import render, HttpResponse
def index(request):
    return HttpResponse("this is the equivalent of @app.route('/')!")
```


project_name
manage.py
project_name
settings.py
urls.py
apps
app_name
urls.py
views.py
1. Create a Django project
3c. Update the settings.py file to include our app(s)
3d. Add a route to the project urls.py file to redirect to our app
2. Create an apps folder
3. Create a new app
3b. Create the urls.py file
3e. Add route(s) to the urls.py file And methods to the app's views.py file

Templates

- Django expects a templates folder just like Flask
- Each app will have its own templates folder
- Within that folder add another folder with the same name as the app
- Finally within that folder store all of your HTML templates



```
project_name
    manage.py
    project_name
        settings.py
        urls.py
    apps
        app_name
            urls.py
            views.py
            templates
                app_name
                    index.html
```

- Seems tedious but reason for it is that when we run a Django project, even though there may be multiple apps, it runs as one project.
- Django looks through all the apps and collects all the contents of any templates folder together.
- If there are 2 apps with index.html but no subdirectories then there is no way to distinguish between these 2 files and the first one will always be rendered.

assuming that we have the folder structure setup properly we can render templates in out view.py folder

- when we call the render function the 1st argument will always be request
- 2nd argument will be a string indicating which HTML file to render

```python
from django.shortcuts import render
def index(request):
  return render(request, "app_name/index.html")
```

Pass Data to the Template

- In, Django we are also able to pass data to the template via the render method, but rather than being able to pass up any number of arguments, we can only pass a single dictionary whose keys will be the variable names available on the template.

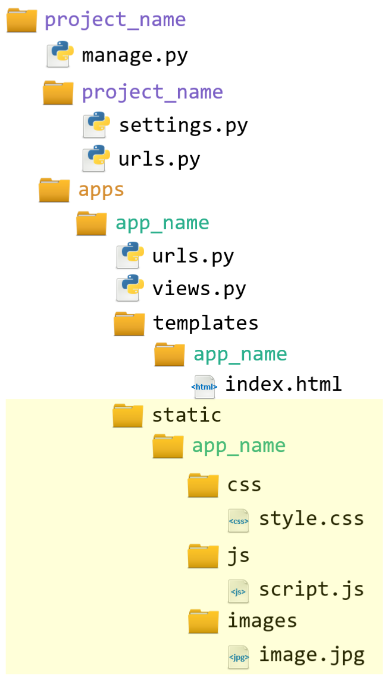
```python
from django.shortcuts import render

def index(request):
  context = {
    'name': 'Noelle',
    'favorite_color': 'turquoise',
    'pets': ['Bruce', 'Fitz', 'Georgie']
  }
  return render(request, 'app_name/index.html', context)
```

```html
<h1copy>Info From Server:</h1>
<p>Name: {{name}}</p>
<p>Color: {{favorite_color}}</p>
<p>Pets</p>
<ul>
{% for pet in pets %}
   <li>{{pet}}</li>
{% endfor %}
</ul>
```

- You cannot user square brackets with Django's template engine!
- You cannot comment out template engine syntax with regular HTML comments (check docs if you want to be able to comment out properly)

Static Files

- Organization and behavior of static files is similar to templates
- Within each app, at the same level as our templates folder, we also need a folder called static.
- Within the folder should be another folder with the same name as the app.
- Then within that folder we save all of our static content (further subdivide into js, css, images folders as desired)



```
project_name
    manage.py
    project_name
        settings.py
        urls.py
    apps
        app_name
            urls.py
            views.py
            templates
                app_name
                    index.html
            static
                app_name
                    css
                        style.css
                    js
                        script.js
                    images
                        image.jpg
```

```
<!DOCTYPE html>
  <html>
    <head>
      <meta charset="utf-8">
      <title>Index</title>
      {% load static %}     <!-- added this line -->
```

We can reference any static files relative to their location with the folder called static

```
<!DOCTYPE html>
  <html>
    <head>
      <meta charset="utf-8">
      <title>Index</title>
      {% load static %}
      <link rel="stylesheet" href="{% static 'app_name/css/style.css' %}">····
    <script src="{% static 'app_name/js/script.js' %}"></script>
  </head>
  <body>
      <img src="{% static 'app_name/images/image.jpg' %}" />
  </body>
```

GET vs. POST Requests

- In Django, we do not specify what types of requests a given route will receive when defining the url patterns
- Within the associated function we can specify a certain action depending on the type of request being received

```python
from django.shortcuts import render, redirect
def some_function(request):
    if request.method == "GET":
        print("a GET request is being made to this route")
        return render(request, "some_app/some_template.html")
    if request.method == "POST":
        print("a POST request is being made to this route")
        return redirect("/")
```

Submitting Form Data

- In Django we will user request.POST to access form data that is submitted
- If the form is a GET request then we can use request.GET

```python
from django.shortcuts import render, redirect
def some_function(request):
    if request.method == "GET":
        print(request.GET)
    if request.method == "POST":
        print(request.POST)
```

- Any forms that are being submitted as POST requests must include a CSRF token
- This token is used to prevent cross-site request forgery, which is a malicious kind of attack where a hacker can pretend to be another user and submit data to a site that recognizes that user
- CSRF will allow Django to add a hidden input field and value that helps our server recognize genuine requests

```html
<form action="/some_route" method="post">
  {% csrf_token %}
  <p>Field One: <input name="one" type="text"></p>
  <p>Field Two: <input name="two" type="text"></p>
  <button type="submit">Submit</button>
```

```
</form>
```

- Remember that the names of the input fields from out form will be the keys we use to access the data in out server

```
from django.shortcuts import render, redirect
def some_function(request):
    if request.method == "POST":
        val_from_field_one = request.POST["one"]
          val_from_field_two = request.POST["two"]
```

Links

https://docs.djangoproject.com/en/2.1/ref/csrf/
https://en.wikipedia.org/wiki/Cross-site_request_forgery

Session

- Migrations help manage the state of our database including creating and updating any tables
- Django utilizes the database to manage sessions, we will need to update out database to allow for it to start maintaining session data for us.

```
project_name> python manage.py migrate
```

- To use session we can refer to it in our view.py file

```python
# some_project/apps/some_app/views.py
def some_function(request):
    request.session['name'] = request.POST['name']
    request.session['counter'] = 100
```

- We can access session directly in our Django templates
- Django does not process square brakcets so use dot notation

```html
<p>Name in session is: {{request.session.name}}</p>
```

Userful Session Methods

```python
# will retrieve (get) the value associated with 'key'
request.session['key']

# Set the value that will be stored by 'key' to 'value'
request.session['key'] = 'value'

# Returns a boolean of whether a key is in session or not
'key' in request.session

# Use dot notation (.) to access request.session keys from templates since square brackets ([]) aren't allowed there
{{ request.session.name }}

# Deletes a session key if it exists, throws a KeyError if it doesn't. Use along with try and except since it's better to ask for forgiveness than permission
del request.session['key']
```

Named Routes

- Makes referencing our Django routes pretty easy
- All we need to do is pass a keyword variable (name) to the path method we use inside our app’s urls.py file

ORMs

- Django’s ORM allows us to write pure python code without having to manage logn SQL query strings in our logic
- ORM helps our code become much clearer and easier to read
- ORM can translate each row in a table to an instance of a Python class
- Each of the database fields are an attribute of the class
- ORM comes with methods that actually perform SQL queries
- Help address security concerns by sanitizing user-provided data and preventing SQL injection attacks

Models

The Why

- Models are the M of the MTV architecture
- Goal of modularizing is to separate our code so tha teah part has a specific purpose
- Purpose of model is to do all the work of interfacing with the database, whether retrieving information from or putting information into it
- Skinny controllers and fat models

The How

- When we created the app Django automatically setup models.py
- Models are simply classes that map to our database tables

Why models.Model?

```python
from django.db import models

class Movie(models.Model):
    title = models.CharField(max_length=45)
    description = models.TextField()
    release_date = models.DateTimeField()
    duration = models.IntegerField()
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

- We are inheriting from the model.Model base class
- Do not need to type an id field into any of our classes, Django automatically adds a field called “id” to every class inheriting from models.Model and sets it to be an auto-incremented field
- We dont have to write a separate ___init___ method for each class

Migrations

```
> python manage.py makemigrations
> python manage.py migrate
```

- makemigrations is a kind of staging, when the command runs Django looks through all our code finds any changes we made to our models that will affect the database and then formulates the correct python code to move on to the next step
- Migrate actually applies the changes made above, this step is where the SQL queries are actually built and executed

Notes:

- Never delete migration files and always make migrations and migrate anytime you change something in your models.py files - thats what updates the actual database so it reflects what's in your models
- For now we are using SQLite, it is not recommended for use once our application is ready for production, in deployment section we will learn how to switch to a MySQL database
- Django ORM models and queries will always be the same no matter which database you are using

Django Shell

To use the shell we will run the command below in our terminal from our project's root directory

```
> python manage.py shell
> python manage.py shell -i ipython
```

Since we are interested specifically in working with our models lets import them

```
from apps.your_app_name_here.models import *
```

Caution

https://stackoverflow.com/questions/2360724/what-exactly-does-import-import

ORM CRUD Commands

```python
class Movie(models.Model):
    title = models.CharField(max_length=45)
    description = models.TextField()
    release_date =copy models.DateField()
    duration = models.IntegerField()
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

Creating

To add a new record to a table:

```python
ClassName.objects.create(field1="value for field1", field2="value for field2")
```

- The create method returns an instance of the model with the values that were just added. This means that if we wanted to do something with the instance after creating our database, we could set a variable and use it like so:

```python
newly_created_movie = Movie.objects.create(title="The Princess Bride",description="the best movie ever",release_date="1987-09-25", duration=98)
print(newly_created_movie.id)   # view the new movie's id
```

- Another way to add a row to our databse is by creating an instance of the class and saving it, like so:

```python
newly_created_movie = Movie(title="The Princess Bride",description="the best movie ever",release_date="1987-09-25",duration=98)
newly_created_movie.save()
```

- By default all fields in our models are non-nullable, meaning all fields are required upon creation.
- If you want to change this default behavior check out Django's documentation on the null property.

Reading

There are several ways that we might want to try to retrieve records from the database

Multiple Records

There are several different methods that will return multiple records from the database

All

To get all the rows from a given table:

```
all_movies = Movie.objects.all()
```

- The all method returns a list (technically a querySet) of instances of the model

Filter (WHERE)

To specify criteria for retrieving rows from a given table:

```
some_movies = Movie.objects.filter(release_date='2018-11-16')
```

- Filter method also returns a list (technically a QuerySet) of instances of the model.

Exclude (WHERE NOT)

To specify criteria for filtering out records to retrieve:

```
other_movies = Movie.objects.exclude(release_date='2018-11-16')
```

- The exclude method also returns a list (technically a QuerySet) of instances of the model.

Single Records

There are also several different methods that will return a single instance of a class.

Get

To get a specific row from the table, specify a field and value

```
one_movie = Movie.objects.get(id=7)
```

- The get method returns a single instance of the model
- If our specified value find no matching results or more than one matching result from the database, we will get an error
- This is why we should really only use fields that we know will be unique, with values that we are certain are

in the database

First

To get the first row from the table:

```
first_movie = Move.objects.first()
```

- The first methods returns a single instance of the model
- If no order is specified before calling the first method, the data is ordered by the primary key

Last

To get the last row from the table:

```
first_movie = Movie.objects.last()
```

- Returns a single instance of the model and if no order is specified thne it uses the primary key

Updating

In order to update an existing record, we first need to obtain the instance of the record we want to modify and then use the save method to commit those changes to the database

```
movie_to_update = Movie.objects.get(id=42)  # let's retrieve a single movie,
movie_to_update.description = "the answer to the universe"  # update one/some of its field values
movie_to_update.title = "The Hitchhiker's Guide to the Galaxy"
movie_to_update.save()  # then make sure all changes to the existing record get saved to the database
```

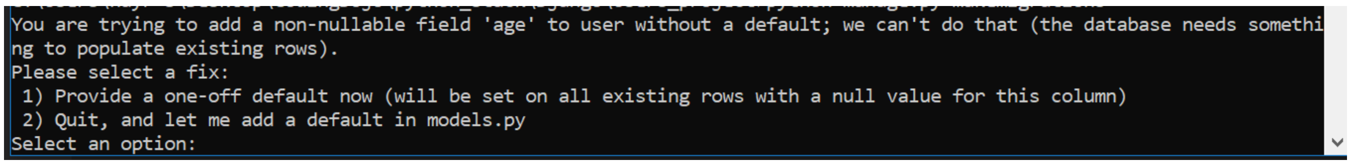
Deleting

In order to delete an existing record, we again need to obtain the instance of the record and then use the delete method

```
movie_to_delete = Movie.objects.get(id=2)   # let's retrieve a single movie,
movie_to_delete.delete()    # and then delete it
```

Alerting Models After Creation

- Whenever when need to make changes to our models after the initial migration, we will need to re-run the `makemigrations` and `migrate` commands

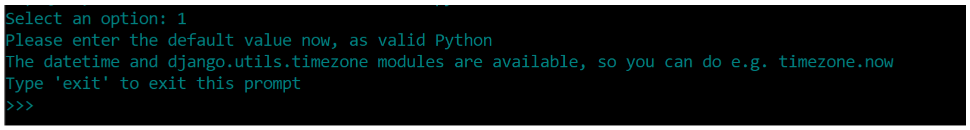


```
You are trying to add a non-nullable field 'age' to user without a default; we can't do that (the database needs somethi
ng to populate existing rows).
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a null value for this column)
 2) Quit, and let me add a default in models.py
Select an option:
```

- This error is telling us that we are trying to add a field that isnt allowed to be null, but any existing data will obviously not have values for that field.

Option 1

Quickest and simplest option. Type 1 and hit enter. You'll then see a prompt that asks what value you would like:



```
Select an option: 1
Please enter the default value now, as valid Python
The datetime and django.utils.timezone modules are available, so you can do e.g. timezone.now
Type 'exit' to exit this prompt
>>>
```

- The value you provide should be compatible with the field type: type the value next to the >>> and hit enter.
- The only exception is for a ForeignKeyField — the command line tool does not really allow for complex imports and retrievals, so you can actually specify a value for the field's primary key
- Once done, don't forget to run migrate

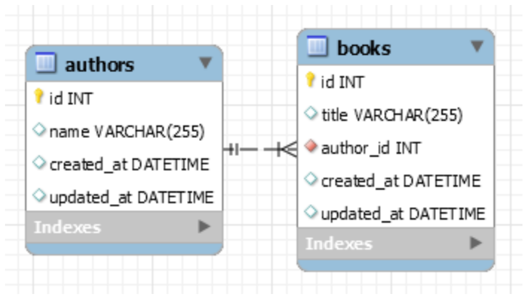
Option 2

If you know you want to provide a default value for any existing and potentially future entries for this new field in your table, type 2 and hit enter. In your model, revise the new field by adding a default argument and value. You might also consider setting the field to be nullable, if that field is optional. For example:

```python
age = models.IntegerField(default=200)  # if no age is entered for a new/existing, age will be set to 200
age = models.IntegerField(null=True)    # if no age is provided, the field will remain empty
```

One to Many Relationships

To indicate a one-to-many relationship between models, Django uses a special field, `ForeignKey`.
Consider these models, where one author can write many books and one book can be written by only oen author:



```python
class Author(models.Model):
    name = models.CharField(max_length=255)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
class Book(models.Model):
    title = models.CharField(max_length=255)
    author = models.ForeignKey(Author, related_name="books")
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

- Notice that rather than having a simple integer field as our foreign key, the Book model has this line: models.ForeignKey(Author, related_name="books")

Creating

The author field is expecting an instance of the Author class. So to create a record that has this foreign key relationship, we first need to have an instance of an Author, and then we can pass it like we have any other field:

```python
this_author = Author.objects.get(id=2)  # get an instance of an Author
my_book = Book.objects.create(title="Little Women", author=this_author) # set the retrieved author as the author of a new book

# or in one line...
my_book = Book.objects.create(title="Little Women", author=Author.objects.get(id=2))
```

Reading

Joins in Django happen automatically.
If you have a book object, you don't need to run an additional query to get information about the author.
If we retrieve an instance of a book:

```
some_book = Book.objects.get(id=5)
some_book.title     # returns a string that is the title of the book
some_book.author    # returns the Author instance associated with this book
```

Just as we are able to filter by other fields we can also search based off of a ForeignKey relationship.
This code will find all of the books by the author with ID 2:

```
this_author = Author.objects.get(id=2)
books = Book.objects.filter(author=this_author)

# one-line version:
books = Book.objects.filter(author=Author.objects.get(id=2))
```

Reverse Lookup

- The related_name field is used to do reverse look-ups.
- When we create a relationship in a Django model, Django places a field in the corresponding table, which we can access by the related_name we provide.
- This means that when we added the author field in our Book class, Django has also placed a field in the Author class that holds information about all of a given author's books.
- Because the Author class has a books field, we can access the books of a given author, `some_author.books.all()`

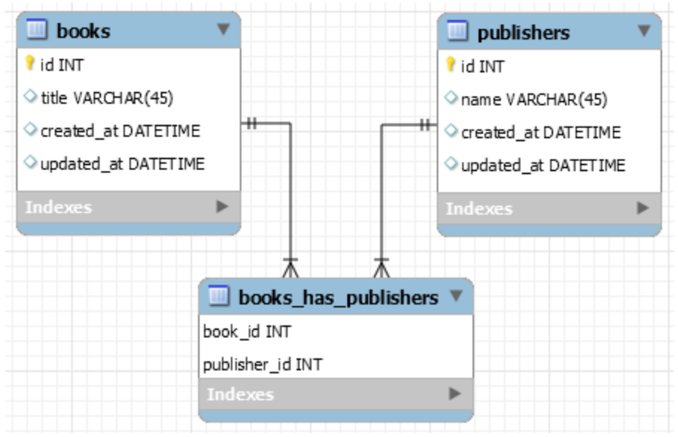
```
def index(request):
    context = {"authors": Author.objects.all()}     # we're only sending up all the
authorscopy
    return render(request, "books/index.html", context)
```

```
<h1>Author List</h1>
<ul>
  {% for author in authors %}
    <li>{{author.name}}
      <ul>
        <!-- looping through each author's books! -->
        {% for book in author.books.all %}
          <li><em>{{book.title}}</em></li>
        {% endfor %}
      </ul>
    </li>
  {% endfor %}
</ul>
```

Many to Many Relationships

Django will construct many to many relationships for you automatically if you model includes a `ManyToManyField`.

Each publisher can publish many books, and each book can be published by many publishers:



```
class Book(models.Model):
    title = models.CharField(max_length=255)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

class Publisher(models.Model):
    name = models.CharField(max_length=255)
    books = models.ManyToManyField(Book, related_name="publishers")
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

- Unlike with a `ForeignKey`, it doesnt matter which model has the `ManyToManyField`.
- The model would still work if the Book model has a field named publishers instead (though we would need to put the publisher class first)

Adding a relationship between two records is simple:

```
this_book = Book.objects.get(id=4)  # retrieve an instance of a book
this_publisher = Publisher.objects.get(id=2)    # retrieve an instance of a publisher

# 2 options that do the same thing:
this_publisher.books.add(this_book)     # add the book to this publisher's list of books
# OR
this_book.publishers.add(this_publisher)    # add the publisher to this book's list of
```

```
publishers
```

To remove a relationship between two existing records:

```
this_book = Book.objects.get(id=4)  # retrieve an instance of a book
this_publisher = Publisher.objects.get(id=2)    # retrieve an instance of a publisher

# 2 options that do the same thing:
this_publisher.books.remove(this_book)      # remove the book from this publisher's list of books
# OR
this_book.publishers.remove(this_publisher) # remove the publisher from this book's list of publishers
```

- The two methods for adding shown above are equivalent because a ManyToManyField is automatically symmetrical
- By adding a book to a publisher, Django will automatically add the publisher to the book
- This means that we can add or look up from the other end without issue

The syntax to see all books from a given publisher is the same as when doing a reverse look-up on a ForeignKey relationship:

```
this_publisher.books.all()  # get all the books this publisher is publishing
this_book.publishers.all()  # get all the publishers for this book
```

Remember to leave off the parentheses when referring to this collection in your template `this_publisher.book.all`

Advanced Queries

Field Lookups

- How you specify the meat of an SQL WHERE clause. They're specified as keyword arguments to the QuerySet methods filter(), exclude(), amd get()
- Basic lookups keyword arguments take the form `field__lookuptype=value`

exact

exact match. if value for comparison is None, it is interpreted as an SQL NULL

```
Entry.objects.get(id__exact=14)
Entry.objects.get(id__exact=None)
```

iexact

case insensitive exact match. if value provided is None it will be interpreted as SQL NULL

```
Blog.objects.get(name__iexact='beatles blog')
Blog.objects.get(name__iexact=None)
```

contains

case sensitive containment test

```
Entry.objects.get(headline__contains='Lennon')
```

in

in a given iterable, often a list, tuple, or query set. It is not a common use case but strings are accepted

```
Entry.objects.filter(id__in=[1, 3, 4])
Entry.objects.filter(headline__in='abc')
```

you can also use a query set to dynamically evaluate the list of value instead of providing a list of literal values

```
inner_qs = Blog.objects.filter(name__contains='Cheddar')
entries = Entry.objects.filter(blog__in=inner_qs)
```

gt

greater than

```python
Entry.objects.filter(id__gt=4)
```

gte

greater than or equal to

lt

less than

lte

less than or equal to

startswith

case sensitive starts with

```python
Entry.objects.filter(headline__startswith='Lennon')
```

istartswith

case sensitive starts with

```python
Entry.objects.filter(headline__istartswith='Lennon')
```

endswith

case insensitive ends with

```python
Entry.objects.filter(headline__endswith='Lennon')
```

iendswith

case insensitive ends with

```python
Entry.objects.filter(headline__iendswith='Lennon')
```

range

Range test (inclusive). you can use range anywhere you can use BETWEEN in SQL — for dates, numbers and even characters

```python
import datetime
start_date = datetime.date(2005, 1, 1)
end_date = datetime.date(2005, 3, 31)
```

```
Entry.objects.filter(pub_date__range=(start_date, end_date))
```

date

for datetime fields, casts the value as date. allows chaining additional field lookups. takes a date value

```
Entry.objects.filter(pub_date__date=datetime.date(2005, 1, 1))
Entry.objects.filter(pub_date__date__gt=datetime.date(2005, 1, 1))
```

year

for date and datetime fields, an exact year match. allows chaining additional field lookups. takes an integer year

```
Entry.objects.filter(pub_date__year=2005)
Entry.objects.filter(pub_date__year__gte=2005)
```

month

For date and datetime fields, and exact month match. allows chaining additional field lookups. takes an integer 1 through 12

```
Entry.objects.filter(pub_date__month=12)
Entry.objects.filter(pub_date__month__gte=6)
```

day

for date and datetime fields, an exact day match. allows chaining additional field lookups. Takes an integer day

```
Entry.objects.filter(pub_date__day=3)
Entry.objects.filter(pub_date__day__gte=3)
```

week

for date and datetime fields, return the week number (1-52 or 53). weeks start on a monday and the first week contains the year's first Thursday.

```
Entry.objects.filter(pub_date__week=52)
Entry.objects.filter(pub_date__week__gte=32, pub_date__week__lte=38)
```

week_day

For date and datetime fields, a 'day of the week' match. allows chaining field lookups. takes an integer value representing the day of week from 1 (Sunday) to 7 (Saturday).

```
Entry.objects.filter(pub_date__week_day=2)
Entry.objects.filter(pub_date__week_day__gte=2)
```

quarter

for date and datetime fields, a ‘quarter of the year’ match. allows additional field lookups. takes an integer value between 1 and 4 representing the quarter of the year.

```
Entry.objects.filter(pub_date__quarter=2)
```

time

for datetime fields, casts the value as time. allows chaining additional field lookups. takes a datetime.time value

```
Entry.objects.filter(pub_date__time=datetime.time(14, 30))
Entry.objects.filter(pub_date__time__range=(datetime.time(8), datetime.time(17)))
```

hour

for datetime and time fields, an exact hour match. allows chaining additional field lookups. takes an integer between 0 and 23

```
Event.objects.filter(timestamp__hour=23)
Event.objects.filter(time__hour=5)
Event.objects.filter(timestamp__hour__gte=12)
```

minute

for datetime and time fields, an exact minute match. allows chaining additional field lookups. takes an integer between 0 and 59.

```
Event.objects.filter(timestamp__minute=29)
Event.objects.filter(time__minute=46)
Event.objects.filter(timestamp__minute__gte=29)
```

second

for datetime and time fields, an exact second match. allows chaining additional field lookups. takes an integer between 0 and 59

```
Event.objects.filter(timestamp__second=31)
Event.objects.filter(time__second=2)
Event.objects.filter(timestamp__second__gte=31)
```

isnull

takes either True or False, which correspond to SQL queries of IS NULL and IS NOT NULL

```
Entry.objects.filter(pub_date__isnull=True)
```

regex

case sensitive regular expression match. regex syntax is that of the database backend in use. SQLite which has no built in regular expression support, this feature is provided by a use definied REGEXP function, and the regex syntax is therefore that of Python’s re module

```
Entry.objects.get(title__regex=r'^(An?|The) +')
```

Adding Validation

Since we are modularizing now this code will be part of models since models should be doing everything database related.

```python
# Inside your app's models.py file
# imports
class Blog(models.Model):
    name = models.CharField(max_length=255)
    desc = models.TextField()
    created_at = models.DateTimeField(auto_now_add = True)
    updated_at = models.DateTimeField(auto_now = True)
"""
models come with a hidden property:
      objects = models.Manager()
we are going to override this!
"""
```

- Without any additional work on our end, this objects property, or Manager in Django terms, has been making available to us the ORM queries like .all(), .get(), etc

Notice that Blog and BlogManager are inheriting from entirely different models.

- By inheriting from models.Model, Blog is made into a database table
- By inheriting from models.Manager, BlogManager will inherit from the ORM-building class

Manager class is another builtin Django class used to extend our models' functionality. This means BlogManager still contains all the methods it did before, but we are now able to add our own methods. Here we are making one called basic_validator, that expects a dictionary of data from our controller:

```python
# Inside your app's models.py file
from __future__ import unicode_literals
from django.db import models
# Our custom manager!
# No methods in our new manager should ever receive the whole request object as an argument!
# (just parts, like request.POST)
class BlogManager(models.Manager):
    def basic_validator(self, postData):
        errors = {}
        # add keys and values to errors dictionary for each invalid field
        if len(postData['name']) < 5:
            errors["name"] = "Blog name should be at least 5 characters"
        if len(postData['desc']) < 10:
            errors["desc"] = "Blog description should be at least 10 characters"
        return errors
```

Now to link our BlogManager to our Blog class, we are going to override the objects property and have it erference our newly created manager:

```
class Blog(models.Model):
    name = models.CharField(max_length=255)
    desc = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    objects = BlogManager() # add this line!
```

Now in our views.py file, we can use Blog.objects.basic_validator(postData)

```
# Inside your app's views.py file
from django.shortcuts import render, HttpResponse, redirect
from django.contrib import messages

from .models import Blog
def update(request, id):
    # pass the post data to the method we wrote and save the response in a variable called errors
    errors = Blog.objects.basic_validator(request.POST)
        # check if the errors dictionary has anything in it
        if len(errors) > 0:
            # if the errors dictionary contains anything, loop through each key-value pair and make a flash message
            for key, value in errors.items():
                messages.error(request, value)
            # redirect the user back to the form to fix the errors
            return redirect('/blog/edit/'+id)
        else:
            # if the errors object is empty, that means there were no errors!
            # retrieve the blog to be updated, make the changes, and save
            blog = Blog.objects.get(id = id)
            blog.name = request.POST['name']
            blog.desc = request.POST['desc']
            blog.save()
            messages.success(request, "Blog successfully updated")
            # redirect to a success route
            return redirect('/blogs')
```

- Notice that we used Django’s messages framework

Model Managers

- Interface through which database query operatoins are provided to Django models
- At least one Manager exists for every model in a Django application
- Managers are accessible only via model classes, rather than from model instances, to enforce separation between “table-level” operations and “record-level” operations.

Model.objects

- Each non-abstract `Model` class must have a `Manager` instance added to it
- Django ensures that in your model class you have at least a default `Manager` specified
- If you do not add your own `Manager`, Django will add an attribute `objects` containing default `Manager` instance

Manager Names

- If you want to user `objects` as a field name, or if you want to use a name other than `objects` for the `Manager`, you can rename it on a per-model basis.
- To rename the `Manager` for a given class, define a class attribute of type `models.Manager()` on that model.

```python
from django.db import models

class Person(models.Model):
    #...
    people = models.Manager()
```

Custom Managers

- You can use a custom `Manager` in a particular model by extending the base `Manager` class and instantiating you custom `Manager` in your model.
- There are 2 reasons you might want to customize a `Manager`:
  - to add extra `Manager` methods
  - modify the initial `QuerySet` the `Manager` returns

Adding Extra Manager Methods

- Preferred way to add "table-level" functionality to your models.
- Custom Manager method can return anything you want, it does not have to return a QuerySet

This custom `Manager` offers a method `with_counts()`, which returns a list of all `OpinionPoll` objects, each with an extra `num_responses` attribute that is the result of an aggregate query:

```
from django.db import models

class PollManager(models.Manager):
    def with_counts(self):
        from django.db import connection
        with connection.cursor() as cursor:
            cursor.execute("""
                SELECT p.id, p.question, p.poll_date, COUNT(*)
                FROM polls_opinionpoll p, polls_response r
                WHERE p.id = r.poll_id
                GROUP BY p.id, p.question, p.poll_date
                ORDER BY p.poll_date DESC""")
            result_list = []
            for row in cursor.fetchall():
                p = self.model(id=row[0], question=row[1], poll_date=row[2])
                p.num_responses = row[3]
                result_list.append(p)
        return result_list

class OpinionPoll(models.Model):
    question = models.CharField(max_length=200)
    poll_date = models.DateField()
    objects = PollManager()

class Response(models.Model):
    poll = models.ForeignKey(OpinionPoll, on_delete=models.CASCADE)
    person_name = models.CharField(max_length=50)
    response = models.TextField()
```

- You use `OpinonPoll.objects.with_counts()` to return that list of `OpinionPoll` objects with `num_responses` attributes.
- `Manager` methods can access `self.model` to get the model class to which they're attached.

Modifying A Manager's Initial QuerySet

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

The following model has two Managers - one that returns all objects, and one that returns only the books by Roald Dahl:

```python
# First, define the Manager subclass.
class DahlBookManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().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.
```

Messages Framework

- Django provides full support for cookie and session-based messaging, for both anonymous and authenticated users.
- Allows you to temporarily store messages in one request and retrieve them for display in a subsequent request.

Message Levels

- DEBUG
    - Development related messages that will be ignored (or removed) in a production deployment
- INFO
    - Informational messages for the user
- SUCCESS
    - An action was successful ("Your profile was updated successfully")
- WARNING
    - A failure did not occur but may be imminent
- ERROR
    - An action was not successful or some other failure occurred

Using Messages In Views And Templates

Adding A Message

To add a message, call:

```python
from django.contrib import messages
messages.add_message(request, messages.INFO, 'Hello world.')
```

Some shortcut methods provide a standard way to add messages with commonly used tags

```python
messages.debug(request, '%s SQL statements were executed.' % count)
messages.info(request, 'Three credits remain in your account.')
messages.success(request, 'Profile details updated.')
messages.warning(request, 'Your account expires in three days.')
messages.error(request, 'Document deleted.')
```

Displaying Messages

In your template, use something like this:

```
{% if messages %}
  <ul class="messages">
    {% for message in messages %}
      <li {% if message.tag %} class="{{ message.tags }}"{% endif %}>
        {{ message }}
      </li>
    {% endfor %}
  </ul>
{% endif %}
```

- Even if you know there is only one message, you should still iterate over the messages sequence, because otherwise the message storage will not be cleared for the next request.

Outside of templates, you can use get_messages():

```
from django.contrib.messages import get_messages

storage = get_messages(request)
for message in storage:
    do_something_with_the_message(message)
```