**Model in django**

* A model is a class that represents table or collection in our DB, and where every attribute of the class is a field of the table or collection. Models are defined in the app/models.py
* Each model is a Python class that subclasses [django.db.models.Model](https://docs.djangoproject.com/en/3.1/ref/models/instances/" \l "django.db.models.Model" \o "django.db.models.Model).
* Each attribute of the model represents a database field

With all of this, Django gives you an automatically-generated database-access

Lets take a n exapme



The above **Person** model would create a database table like this:



**Model Meta options**

We will see some possible [metadata options](https://docs.djangoproject.com/en/dev/topics/db/models/#meta-options) that you can give your model in its internal **class Meta**.

Meta class is subclass in mode name/method of model./py file

1. **db\_table=<table\_name>**

The name of the database table to use for the model. By default django creates table by name <appsName\_modelname>

1. **app\_label=<myapp>**

If a model is defined outside of an application in [**INSTALLED\_APPS**](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-INSTALLED_APPS), it must declare which app it belongs to

1. **options.permissions/models.permissions=[permissions]**

Extra permissions to enter into the permissions table when creating this object. Add, change, delete, and view permissions are automatically created for each model

permissions = [('can\_deliver\_pizzas', 'Can deliver pizzas')]

This is a list or tuple of 2-tuples in the format **(permission\_code, human\_readable\_permission\_name)**.

1. **Options.default\_permissions<permissions>**
2. **Options.constraints=<list of constrainsts>**

A list of constraints that you want to define on the model

**Adding data to DB**



**QuerySets**

A django queryset is like its name says, basically a collection of (sql) queries, in your example above print(b.query) will show you the sql query generated from your django filter calls

Since querysets are lazy, the database operation isn't done immediately but operations happens only when needed or when the queryset is evaluated

**QuerySets are lazy**

QuerySets are lazy – the act of creating a QuerySet doesn’t involve any database activity. You can stack filters together all day long, and Django won’t actually run the query until the QuerySet is evaluated.

**>>>** q = Entry.objects.filter(headline\_\_startswith="What")

**>>>** q = q.filter(pub\_date\_\_lte=datetime.date.today())

**>>>** q = q.exclude(body\_text\_\_icontains="food")

**>>>** print(q)----At this line DB will be hitted,there are many easy to evalue querysets

**Evaluating QuerySets**

* Internally, a QuerySet can be constructed, filtered, sliced, and generally passed around without actually hitting the database. No database activity actually occurs until you do something to evaluate the queryset.
* We can evaluate query in following ways

1. **Iteration**

A QuerySet is iterable, and it executes its database query the first time you iterate over it. For example, this will print the headline of all entries in the database



1. **repr()**

A QuerySet is evaluated when you call repr() on it. This is for convenience in the Python interactive interpreter, so you can immediately see your results when using the API interactively.

1. **len()**

A QuerySet is evaluated when you call len() on it.

*It returns the length of the result list ,*For this purpose django provides a **count()** method for precisely this reason.

1. **list()**

Force evaluation of a QuerySet by calling list() on it. For example:



1. **bool()**

Testing a QuerySet in a boolean context, such as using **bool(), or, and or an if** statement, will cause the query to be executed

If there is at least one result, the QuerySet is True, otherwise False.

1. **pickling/caching**
2. **slicing**

* If you use index to pick one/more element from a not evaluated QuerySet, it causes database hits
* you pick from an already evaluated QuerySet it uses cache(from stored variable or result)
* Slicing a not-evaluated QuerySet returns a new QuerySet. The returned QuerySet does not allow further modifications (e.g., adding more filters, or modifying ordering) but it does allow more slicing. Queryset (either sliced or not) saves results to its cache if you iterate over it



Note:

We can do chaining on queryset.

Not all methods from queryset API return a new query set when applied on a queryset

**Methods that return new QuerySets**

<https://docs.djangoproject.com/en/3.1/ref/models/querysets/#when-querysets-are-evaluated>

-------------- Methods which return a new query set

There are many methods for this but here we will see few of them

1. **filter(\*\*kwargs)/** **filter()**

Returns a new QuerySet containing objects that match the given lookup parameters.

If you need to execute more complex queries (for example, queries with OR statements), you can use Q objects.

1. **exclude()/exclude(\*\*kwargs)**

Returns a new QuerySet containing objects that do not match the given lookup parameters/conditions

1. **order\_by()/order\_by(\*field)**

By default, results returned by a QuerySet are ordered by the ordering tuple given by the ordering option in the model’s Meta.

You can override this on a per-QuerySet basis by using the order\_by method



1. **reverse()**

Use the reverse() method to reverse the order in which a queryset’s elements are returned.

Calling reverse() a second time restores the ordering back to the normal direction.



To retrieve last 5 items in querysets

**Operations which return a new queryset**

**AND(&)** and **OR** operation returns a new queryset

1. **AND**

Combines two QuerySets using the SQL AND operator.



Q(xyz)----- this is Q lass we will see it later in this notes

This queryset evaluates to



1. **OR**

Combines two QuerySets using the SQL OR operator.



Above queryset evaluates to



**Methods that do not return QuerySet**

<https://docs.djangoproject.com/en/3.1/ref/models/querysets/#methods-that-do-not-return-querysets>

------------ Methods which don’t return a new queryset

There are masny methods which don’t return queryset , we will see some of them

1. **get()/get(\*\*kwargs)**

Returns the object matching the given lookup parameters, which should be in the format described in Field lookups.



**NOTE**:

1. If get() finds more than one object, it raises a Model.MultipleObjectsReturned exception.
2. If get() doesn’t find any object, it raises a Model.DoesNotExist except

We can handle those exceptions using "**django.core.exceptions**"

1. **2. create()/create(\*\*kwargs)**

A convenience method for creating an object and saving it all in one step.

1. **get\_or\_create()/get\_or\_create(\*kwargs)**

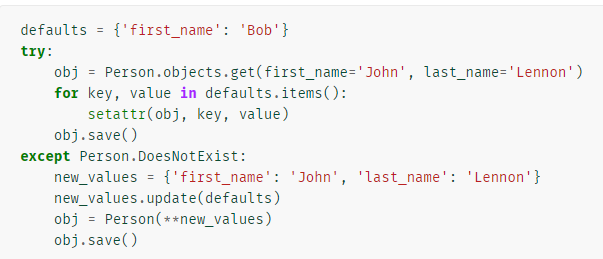
A convenience method for looking up an object with the given kwargs (may be empty if your model has defaults for all fields), creating one if necessary.

1. **update\_or\_create(defaults=None, \*\*kwargs)/update\_or\_create()**

A convenience method for updating an object with the given kwargs, creating a new one if necessary. The defaults is a dictionary of (field, value) pairs used to update the object. The values in defaults can be callables.

Returns a tuple of (object, created), where object is the created or updated object and created is a boolean specifying whether a new object was created.

Let us take example of create or update manually.



Above can be written easily using create\_or\_update() as below



1. **bulk\_update()/bulk\_update(objs, fields, batch\_size=None)**
2. **count()**
3. **in\_bulk()/in\_bulk(id\_list=None, field\_name='pk')**
4. **latest()**
5. **latest(\*fields)**
6. **first()**
7. **bulk\_create()**
8. **bulk\_create(objs, batch\_size=None, ignore\_conflicts=False)**

**Field lookups**

Field lookups are how you specify the meat of an SQL WHERE clause. They’re specified as keyword arguments to the QuerySet methods filter(), exclude() and get().

Django’s built-in lookups are listed below.:-

1. **exact**

Exact match. If the value provided for comparison is None, it will be interpreted as an SQL NULL (see [isnull](https://docs.djangoproject.com/en/3.1/ref/models/querysets/#std:fieldlookup-isnull) for more details).

Example:-



1. **iexact**

Case-insensitive exact match. If the value provided for comparison is None, it will be interpreted as an SQL NULL (see isnull for more details).

1. **Contains**

Case-sensitive containment test.



1. **Icontains**

Case-insensitive containment test.

1. **In**
2. **gt**
3. **gte**
4. **lt**
5. **lte**
6. **startswith**
7. **istartswith**
8. **endswith**
9. **iendswith**
10. **range**
11. **date**
12. **year**

**and so on**

**Caching and QuerySets**

Each [QuerySet](https://docs.djangoproject.com/en/3.1/ref/models/querysets/#django.db.models.query.QuerySet) contains a cache to minimize database access.

To enable cache in QuerySet, simply save the QuerySet in a variable and reuse it

Example:-

print([e.headline for e in Entry.objects.all()])

print([e.pub\_date for e in Entry.objects.all()])

Above code will hit DB twice (each time query is executed)

Below code will execute only one and then result will be used from cached query



1. **Field types**

It is datatypes defined in SQL or datatype of each column in DB table.

In djnago we specify the field type/data type of each column of table/mode name in the model name

More we can check on:-

<https://docs.djangoproject.com/en/3.1/ref/models/fields/#field-types>

* 1. **CharField(max\_length=None, \*\*options)**
  2. **DateField(auto\_now=False, auto\_now\_add=False, \*\*options)**
  3. **DateTimeField(auto\_now=False, auto\_now\_add=False, \*\*options)**
  4. **DecimalField(max\_digits=None, decimal\_places=None, \*\*options)**
  5. **mailField(max\_length=254, \*\*options)**
  6. **FileField(upload\_to=None, max\_length=100, \*\*options)**
  7. **ImageField(upload\_to=None, height\_field=None, width\_field=None, max\_length=100, \*\*options)**
  8. **IntegerField(\*\*options)**
  9. **GenericIPAddressField(protocol='both', unpack\_ipv4=False, \*\*options)**

1. **Field Options**

Field option is something like constrains in SQL

Few are listed below, All available field options we can check it on below link

<https://docs.djangoproject.com/en/3.1/ref/models/fields/#field-options>

* 1. Field.null---If True, Django will store empty values as NULL in the database. Default is False.
  2. Field.blank---If True, the field is allowed to be blank. Default is False.
  3. Field.db\_column---The name of the database column to use for this field. If this isn’t given, Django will use the field’s name
  4. Field.default---default value for rhat field
  5. primary\_key---If True, this field is the primary key for the model.

Field.unique---If True, this field must be unique throughout the table

**Relationships in Django**

We learn relationship in Django. It has two type

1. Many-to-One
2. Many-to-Many
3. **Many-to-One relationship**

For managing many to one relationship we use **models.ForeighKey(modelname, on\_delete)** to acheive it.

This is used in the model/table which hold the foreign key

**models.ForeighKey(modelname, on\_delete)/models.ForeignKey(to, on\_delete, \*\*options)**

this is class in models API

A many-to-one relationship. Requires two positional arguments: the **class/object of class** to which the model is related and the on\_delete option.

*Arguments of ForeignKey()*

* **on\_delete argument/on\_delete=?**

When an object referenced by a ForeignKey is deleted, Django will emulate the behavior of the SQL constraint specified by the on\_delete argument. Available options:-

* + - CASCADE
    - PROTECT
    - RESTRICT
    - SET\_NULL-- Set the ForeignKey null; this is only possible if null is True
    - SET\_DEFAULT ---- Set the ForeignKey to its default value; a default for the ForeignKey must be set.
    - DO\_NOTHING --- Take no action

1. **Many-to-many relationships/ManyToManyField(to,\*\*options)**

* To define a many-to-many relationship, use ManyToManyField
* To establish many to many relationship python creates intermediate table name by tablele1\_table2(by default), we can change the intermediate table name also.
* A many-to-many relationship. Requires a positional argument: the class to which the model is related, which works exactly the same as it does for ForeignKey, including recursive and lazy relationships.
* Related objects can be added, removed, or created with the field’s RelatedManager.

**Arguments of ManyToMany field**

* **ManyToManyField.through**
  + - * If manually specify the intermediary table
      * The most common use for this option is when you want to associate extra data with a many-to-many relationship

Example:-

Below snip of code will have intermediate table by name "Membership" not Person\_Group(by default)





**ManyToManyField.db\_table**

The name of the table to create for storing the many-to-many data

If not used then Django will use table1\_table2 name as intermediatery table

**Generalize way of declaring model**

From all above discussion we can generalize declarations of model name in below way.

*class Modelname(model.Model):*

*table\_column\_name = models.Filed\_Type(Filed\_Options)*

*table\_column\_name = models.Filed\_Type(Relationship\_fields, Fiel\_Options)*

**Looping in Django templetes**

We can use for loop in Django templates in below way

*{% for i in list %}*

*#statements*

*{% endfor %}*

**Getting value from Django context**

It is accessed using cintext variable

{{ }} -----------this is called context variable

1. **Using dot operator**

context\_name.context\_variable\_name

e.g:-

{{ item.item\_name }}

{{ item.item\_desc }}

1. **Directory from variable**

*{{ context\_variable\_nmae))*

Let say I have created context as below



Now we can access value for each context variable in below ways



**Django templates tag**

All available templates can be found on below link

<https://docs.djangoproject.com/en/3.1/ref/templates/builtins/>

1. **for**

Loops over each item in an array, making the item available in a context variable.

For example, to display a list of athletes provided in athlete\_list:

<ul>

{% for athlete in athlete\_list %}

<li>{{ athlete.name }}</li>

{% endfor %}

</ul>

1. **if ({% if %})**

The {% if %} tag evaluates a variable, and if that variable is “true” (i.e. exists, is not empty, and is not a false boolean value) the contents of the block are output

{% if athlete\_list %}

Number of athletes: {{ athlete\_list|length }}

{% elif athlete\_in\_locker\_room\_list %}

Athletes should be out of the locker room soon!

{% else %}

No athletes.

{% endif %}

1. **url**

Returns an absolute path reference (a URL without the domain name) matching a given view and optional parameters. Any special characters in the resulting path will be encoded using iri\_to\_uri()

*{% url 'some-url-name' v1 v2 %}*

*{% url 'app\_name:view-name' %}*

*{% url 'some-url-name' arg1=v1 arg2=v2 %}*

**NOTE**:

1. The first argument is a [URL pattern name](https://docs.djangoproject.com/en/3.1/topics/http/urls/#naming-url-patterns). It can be a quoted literal or any other context variable. Additional arguments are optional and should be space-separated values that will be used as arguments in the URL
2. View-name is the name given in the url.py file for setting url , it’s not model’s view name

Let say we have below date in urls.py file for app “myapp”

path('myapp/', views.detail, name='myapp-detail')

then url should be like:

{% url 'myapp:myapp-detail' %}

1. **extends({% entends %}**

This used for reusing/extending some sort of templated/templates object

Normally the template name is relative to the template loader’s root directory. A string argument may also be a relative path starting with ./ or ../

* **{% extends "base.html" %}**

Uses the literal value "base.html" as the name of the parent template to extend.

* **{% extends variable %}**

If the variable evaluates to a string, Django will use that string as the name of the parent template

If the variable evaluates to a Template object, Django will use that object as the parent template



**Removing hardcoded/Dynamic Url**

The problem with this hardcoded, tightly-coupled approach is that it becomes challenging to change URLs on projects with a lot of templates

Let’s take example of one hardcoded url”

*<li><a href="/polls/{{ question.id }}/">{{ question.question\_text }}</a></li>*

---- this is hardcoded url as we are pasting link in html page it doesn’t required any modification in urls.py

--- this will work for few question ids url which are hardcoded here

We can remove the hardcoded url iin below way

*<li><a href="{% url 'detail' question.id %}">{{ question.question\_text }}</a></li>*

---- No w this will work for each question id

For above urls.py file should have configured like-

*path('<int:question\_id>/', views.detail, name='detail')*

**Namespacing**

It is something like providing the view name given in url.py file(not view name from models.spy file).

Generally used in dynamic url

e.g:-

<li><a href="{% url 'polls:detail' question.id %}">{{ question.question\_text }}</a></li>

**Forms in Django**

1. Bound form
2. Unbound form



Before learning more on Django form we will learn about csrf\_toekn

**csrf\_token**

The CSRF middleware and template tag provides easy-to-use protection against Cross Site Request Forgeries.

The best explanation we can get it from below link

<https://www.youtube.com/watch?v=hW2ONyxAySY>

let’s take an example how one can do it.

Suppose you are logged in to your bank account at same time you open a tab on your browser, that browser can trick with you back account(change password, fund transfer)

**CURD operation in django**

**C - Create**

**U –** Update

**R –** Retrive

**D –** Delete