Here's a working example of how to build up tables with relationships in Django. Use django's ORM capabilities, which relieves the developer of the effort of creating SQL scripts. ORM is portable across a broad range of RDMS, eliminating the need for the developer to redo their code. ORM is a development approach that may be used in a number of development environments.

This Demo relates to page 126 of the learner’s guide.

I purposefully left out the DJANGO setup because you should know how to accomplish that by now.

I've also provided a functional example, which will save you a lot of setup time, but here's some important information about this project.

Project’s supersuser is davidhunt Password1!

Because I wrote this on an iMac, some of the commands have a 3 suffix after them, such as python3; if you're on a Windows machine, simply use python or py.

So the first thing I did was make the models. Okay, so here's a snapshot of three tables configured in the models file.

from django.db import models

# Create your models here.

class Venue(models.Model):

name = models.CharField('Venue Name', max\_length=120)

address = models.CharField(max\_length=300)

post\_code = models.CharField('Post Code',max\_length=4)

phone = models.CharField('Contact Phone', max\_length=10)

website = models.URLField()

email\_address = models.EmailField('Email Address')

def \_\_str\_\_(self) :

return self.name

class ClubMember(models.Model):

first\_name = models.CharField(max\_length=30)

last\_name = models.CharField(max\_length=30)

def \_\_str\_\_(self) :

return self.first\_name + ' ' + self.last\_name

class Event(models.Model):

name = models.CharField('Event Name', max\_length=120)

event\_date = models.DateTimeField('Event Date')

venue = models.ForeignKey(Venue, blank=True, null=True, on\_delete=models.CASCADE)

manager = models.CharField(max\_length= 50)

description = models.TextField(blank=True)

attendees = models.ManyToManyField(ClubMember, blank =True)

def \_\_str\_\_(self) :

return self.name

Please keep in mind that there are two relationships in the above code in the class event table : **venue** is a foreign key based on the venue table (one to many) and **attendees** is based on the club member table as a many to many relationship.

Also note that you do not need to describe linking tables for the many to many relationships because Django will take care of this in the background for you.

Once the model has been established in code, it is time to register it with the admin so that the admin Backpage allows you to enter a link with the database enabling you to conduct typical database operations like as adding information, removing information, and formatting information in a lovely gui as below.

Graphical user interface, application

Description automatically generated

**Admin.py. - register**

So we shift our attention to the 'admin. py file,' where we will write the following code to allow us to interact with our database via the gui seen in the preceding image..

Graphical user interface, text, application, chat or text message

Description automatically generated

And, of course, we provide a reference to our app (db project) in the project's settings file.A screenshot of a computer

Description automatically generated with medium confidence

Okay, I have to perform all of the work establishing the models and attaching the models to the admin back in, as well as linking our app to the project settings, running the migrations, and then building the database for us. If we were the swap databases, we'd have to do this. We would perform the migrations after the swap, and it would create the database for a new RDMS.

python3 manage.py makemigrations

python3 manage.py migrate

python3 manage.py runserver

Now we're going to utilise the Django shell as a scratch area for testing our SQL, or should I say ORM statements to check if we get the proper result. At this point, we haven't billed any views on the sale, which saves us a lot of labour.

Run this command in the terminal to launch the Django shell (>>>)environment.

python3 manage.py shell

Below is some examples of OMR commands that you could initialise against the database.

**1 . A simple select statement retrieves all of the records.**

from db\_project.models import \*

>>> member = ClubMember.objects.all()

>>> print(member)

*QuerySet [<ClubMember: David Hunt>, <ClubMember: Jeff Frig>, <ClubMember: Alice Take>]>*

**2. Restrict the number of records returned to two records**

>>> member = ClubMember.objects.all()[:2]

>>> print(member)

*<QuerySet [<ClubMember: David Hunt>, <ClubMember: Jeff Frig>]>*

**3. Order the records to be returned according to a field.**

>>> members= ClubMember.objects.order\_by("first\_name")

>>> print(members)

*<QuerySet [<ClubMember: Alice Take>, <ClubMember: David Hunt>, <ClubMember: Jeff Frig>]>*

**4. How to do a select statement with a Where (equals… )**

VenueInPost = Venue.objects.filter(post\_code = '4215')

>>> print(VenueInPost)

*<QuerySet [<Venue: Southport Rsl>, <Venue: Southport Workers Club>]>*

**5. How to do a select statement with a Where Not equals to**

>>>VenueNotInPost = Venue.objects.exclude(post\_code = '4215')

>>> print(VenueNotInPost)

*<QuerySet [<Venue: Nerang RSl>]>*

**6. How to do a select statement with a Where id = 2**

>>> VenueNumber2 = Venue.objects.get(id=2)

>>> print(VenueNumber2)

Southport Rsl

>>>

**7.A case sensitive select statement, the target must match**

>>> VenueCaseSen = Venue.objects.get(name\_\_exact='Southport Rsl')

>>> print(VenueCaseSen)

Southport Rsl

**Note:** This Line would throw an error:

VenueCaseSen = Venue.objects.get(name\_\_exact='Southport **RSL**')

**8 . A non- case sensitive select statement**

VenueCaseSen = Venue.objects.get(name\_\_iexact='Southport RSL')

>>> print(VenueCaseSen)

Southport Rsl

**9. Contains a value – similar to a Select *like* statement**

(SQL join)

>>> Venuelike = Venue.objects.get(name\_\_icontains='Ner')

>>> print(Venuelike)

Nerang RSl

**10. Lookup that spans relationships this is similar to a joint statement.**

>>>eventAtEvent = Event.objects.filter(venue\_\_name='Southport Workers Club')

>>> print(eventAtEvent)

*<QuerySet [<Event: back to the 60's>, <Event: Disco lives again !!>]>*

**11. A Many to Many Lookup**

>>> peepsAtEvent = Event.objects.filter(attendees\_\_first\_name ='David')

>>> print(peepsAtEvent)

*<QuerySet [<Event: back to the 60's>, <Event: Disco lives again !!>]>*

>>>peepsAtEventVenue = Event.objects.filter(attendees\_\_first\_name ='David', venue\_\_name='Nerang Rsl')

>>> print(peepsAtEventVenue)

*<QuerySet []>.* **Note: returns empty QuerySet**

**11b . Below you may find information on the events that Alice attended at the Southport Rsl.**

>>>peepsAtEventVenue = Event.objects.filter(attendees\_\_first\_name ='Alice', venue\_\_name='Southport Rsl')

>>> print(peepsAtEventVenue)

<QuerySet [<Event: 90's pop>]>

**12. Delete a record, find the record and then delete it**

>>>record = Venue.objects.get(name\_\_exact='Southport Rsl')

>>> print(record)

Southport Rsl

>>>record.delete()

13. **Update – the use of the name\_\_exact is use for locating the exact file for the update** *‘Venue.objects.get(name\_\_exact='Southport Rsl’).update(name =’Southport RSL’)’*

>>>record = Venue.objects.filter(name\_\_exact='Southport Rsl').update(name='Southport RSL')

>>> print(record)

1

To prove that I didn’t effect any other records in the Venue table:

>>>allEvents = Venue.objects.all()

>>> print(allEvents)

*<QuerySet [<Venue: Nerang RSl>, <Venue: Southport RSL>, <Venue: Southport Workers Club>]>*

More reading

https://docs.djangoproject.com/en/4.0/topics/db/queries/

https://docs.djangoproject.com/en/4.0/topics/db/sql/