

Setting up the Database

**Introduction**:

In this section we will learn about:

1. Creating migrations
2. Running migrations
3. Reversing migrations
4. Populating the database

Purpose is to create a production grade database that we can query in next section.

**Supported Database Engines**:

Popular Database engines supported by Django are:

1. SQLite(comes default with Django, basic and lightweight)

Production grade Database Engines officially supported by Django:

1. PostgresSQL
2. MySQL
3. MariaDB
4. Oracle

3rd party libraries for other Database engines like Microsoft SQL server:

1. MS SQL Server

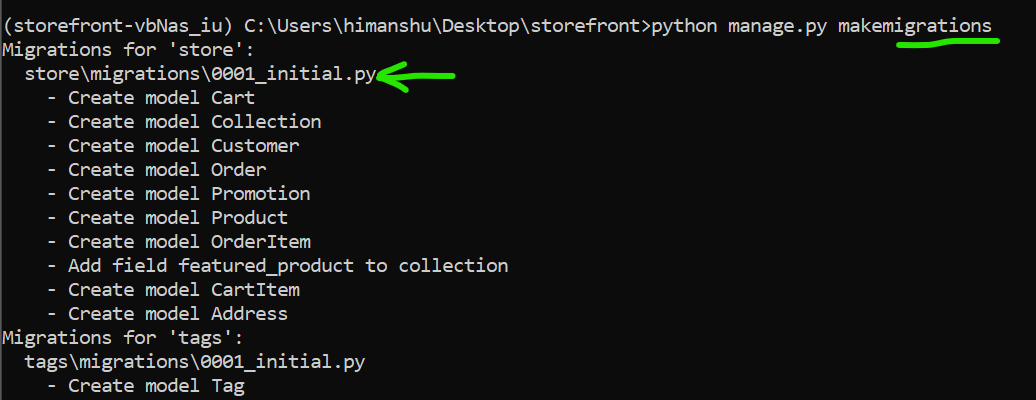
In the Django community two most supported Database engines are MySQL and PostgresSQL.

**Creating Migrations**:

In Django we use migrations to create or update our database tables based on models in our project. So *in Django projects we do not need to manually create or modify our database tables*.

Run this command

python manage.py makemigrations



When we run this command, *Django looked at all the installed apps in our project and for each app it created a new migration file*.

Notice a new file 0001\_initial.py is created inside store\migrations directory which is a migration file for our store app.

Note: Every migration file has a sequence number like 1234, and a descriptive name.

Let us see this migration file,

class Migration(migrations.Migration):

    initial = True

    dependencies = [

    ]

    operations = [

        migrations.CreateModel(

            name='Cart',

            fields=[

                ('id', models.BigAutoField(auto\_created=True, primary\_key=True, serialize=False, verbose\_name='ID')),

                ('created\_at', models.DateTimeField(auto\_now\_add=True)),

                ('title', models.CharField(max\_length=255)),

            ],

        ),

        migrations.CreateModel(

            name='Collection',

            fields=[

                ('id', models.BigAutoField(auto\_created=True, primary\_key=True, serialize=False, verbose\_name='ID')),

                ('title', models.CharField(max\_length=255)),

            ],

        ),

As we can see that this *migration file is just python module*, Here we have a class called Migration with a bunch of operations.

First operation is creating a model called Cart with three fields

migrations.CreateModel(

            name='Cart',

            fields=[

                ('id', models.BigAutoField(auto\_created=True, primary\_key=True, serialize=False, verbose\_name='ID')),

                ('created\_at', models.DateTimeField(auto\_now\_add=True)),

                ('title', models.CharField(max\_length=255)),

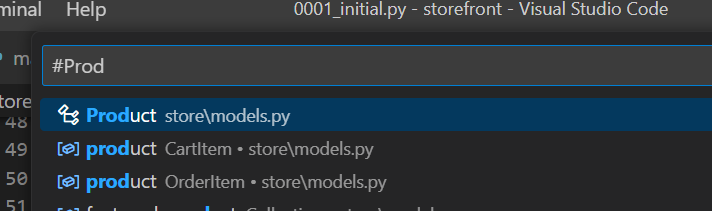
            ],

Note: Here id field is created by Django automatically which is a BigAutoField (roughly translates to big auto incremented column in the database).

At some point we need to run these migrations against a database. At that point, Django will translate this python code into SQL code and run it in top of our database.

Summary: When you create a new model or update existing ones, we need to run makemigrations command to generate a new migration.

Trick: Press ctrl+ T and you will see a search window with # symbol in VS code. Here you can type the name of the model you want and go to module having that model.

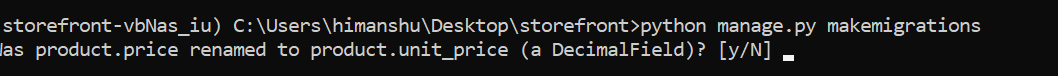


**Exercise 1**:

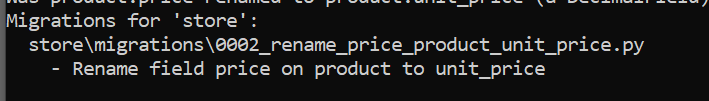
Go to the product model, modify price field name to unit\_price and run makemigrations command again.

**Solution**:

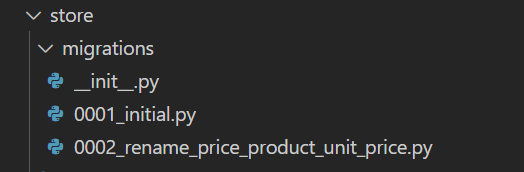
We change the name of the field and run the migrations command on terminal, we see this prompt asking if we did given changes in Product class



After entering y we get a successful migration



In migrations folder of our Store app we get a new migration file, we can change its name to 0002\_rename\_price\_to\_unit\_price.py



Note: If we change the migration file name, make sure that we update all the reference of that file in other migration files. For example in dependencies list.

In the new migration file, we can see a different type of operation

class Migration(migrations.Migration):

    dependencies = [

        ('store', '0001\_initial'),

    ]

    operations = [

        migrations.RenameField(

            model\_name='product',

            old\_name='price',

            new\_name='unit\_price',

        ),

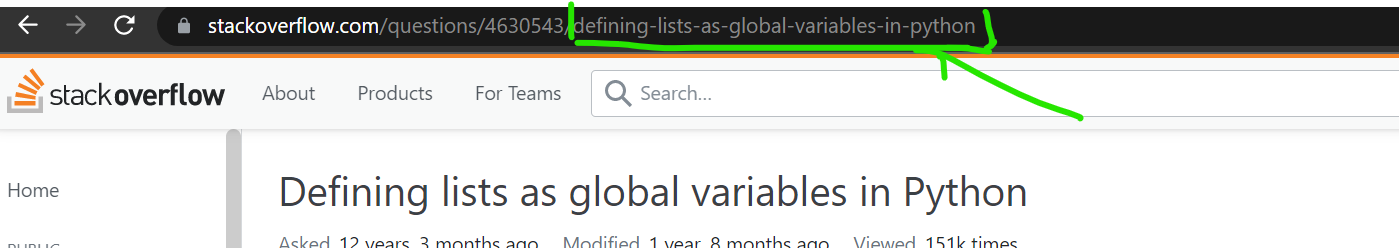
    ]

Instead of CreateModel operation we have a RenameField operation which means when we run this migration Django will go to the product table and rename the price column to unit price.

If we run the makemigrations command again we get



Note: What is a slug?



A slug can be found in some page URL, for example above screenshot

Here notice after question ID, we have *defining-lists-as-global-variables-in-python*

This type of arrangement of letters, numbers, underscores and hyphens in URL is called a slug.

*The whole reason of adding a slug in URL is to make easier for search engines to find our content(a SEO technique)*.

**Exercise 2**:

Create a new field in the product model called slug so that search engine can easily find our product.

Solution:

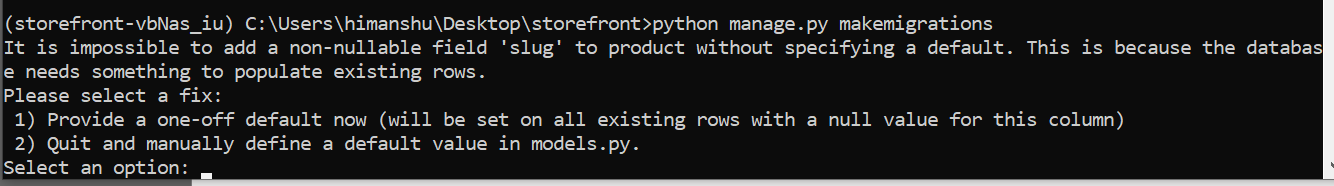
We add a slug field in our Product class and run makemigrations command

class Product(models.Model):

    title = models.CharField(max\_length=255)

    slug = models.SlugField()

We get this…



We have two options;

🡪provide one-off default value right now.

🡪quit and add a default in our model file.

*Second option*:

We quit by pressing 2 and in our models.py we can change slug field to

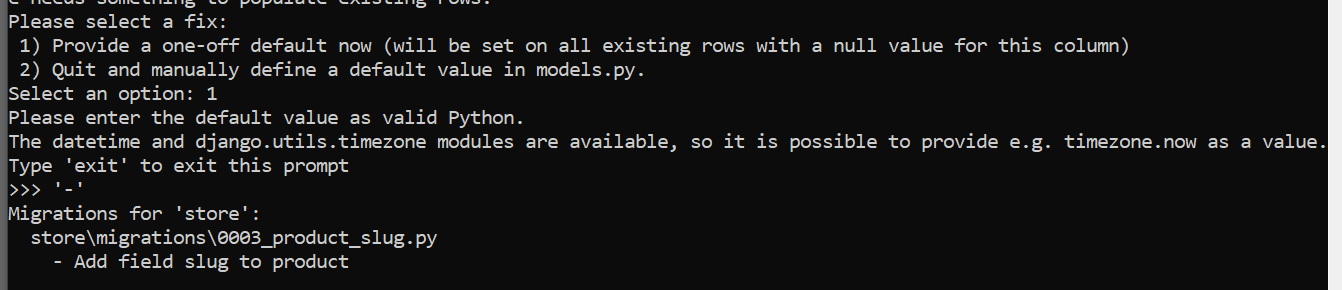
slug = models.SlugField(null=True)

or

slug = models.SlugField(default='-')

*First option*:

We press 1 and Django is asking us to give it default value



So we provided ‘**-**’ hyphen as a string and Django automatically created a new Django file with sequence number. We open this file and see

class Migration(migrations.Migration):

    dependencies = [

        ('store', '0002\_rename\_price\_to\_unit\_price'),

    ]

    operations = [

        migrations.AddField(

            model\_name='product',

            name='slug',

            field=models.SlugField(default='-'),

            preserve\_default=False,

        ),

    ]

Here we have a new operation for adding the field for an existing model/table. So when we run this migration, Django will go to product table and add a new field called *slug* and populate with ‘**-**’ a hyphen.

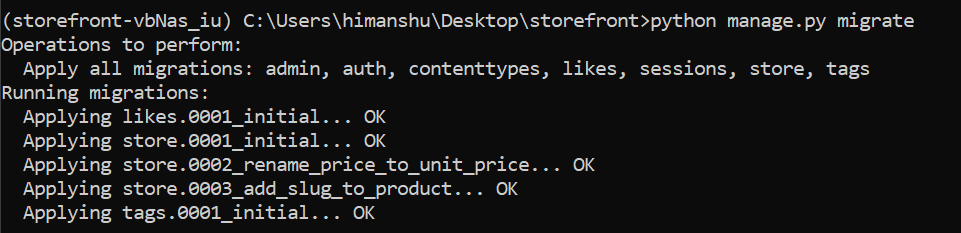
Note: Important thing to note here is that default value is not in our model but only in our migration file. It will be used only once.

**Running Migrations**:

Now we have a bunch of migrations where each migration describes a set of changes. So *each migration is like a commit in a version control system like GIT*.

We need to run these migrations in order to generate our database schema. So in the terminal run this command

python manage.py migrate



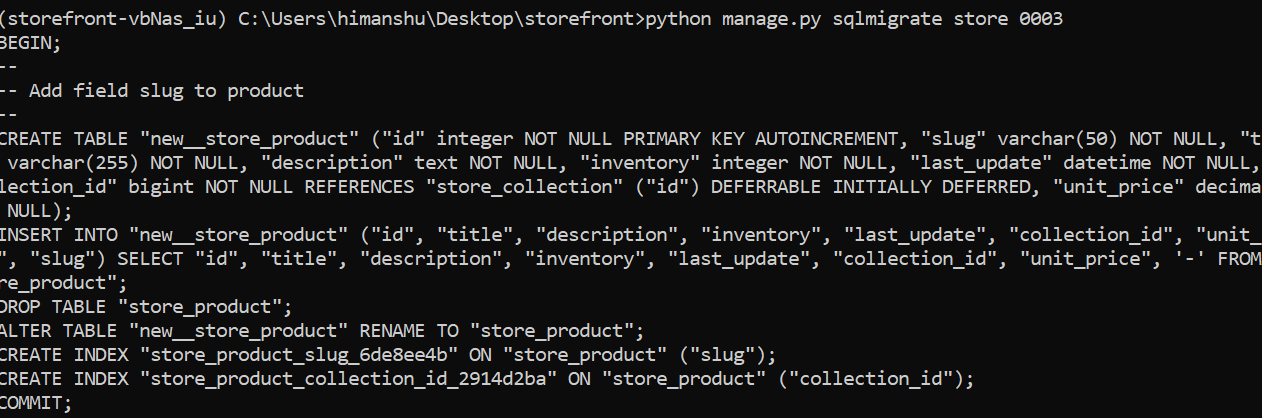
Django went through all the installed apps and executed all the pending migrations.

Note: We have a command called sqlmigrate and using this we can see the actual SQL code that is sent to our database at run time. For example , if you want to see the SQL code for the third migration in the store app

python manage.py sqlmigrate store 0003

store🡪name of app

0003🡪sequence number of migration



We see the actual SQL statement that Django will send to our database depending on database backend we use. Currently we are using SQLlite but *If we switch to MySQL, SQL code that will be generated will be different*.

**Exercise**:

Add zip to address

Create a migration

Run it and inspect migrations table

**Customizing Database Schema**:

Sometimes we need more control over the database schema, for example you may want to overwrite name of a table or you may want to add an index to a couple of columns.

Go to Customer model and inside we are going to create a Meta class where we will *define meta data about this model,* *using some options inside meta class*.

Note: If you search *Django model metadata*, you will find all the available meta options like db\_table for specifying table name, we can use ordering that will be used by default when querying objects, we can use indexes and so on.

class Customer(models.Model):

    MEMBERSHIP\_BRONZE = "B"

    MEMBERSHIP\_SILVER = "S"

    MEMBERSHIP\_GOLD = "G"

    MEMBERSHIP\_CHOICES = [

        (MEMBERSHIP\_BRONZE, "Bronze"),

        (MEMBERSHIP\_SILVER, "Silver"),

        (MEMBERSHIP\_GOLD, "Gold"),

    ]

    first\_name = models.CharField(max\_length=255)

    last\_name = models.CharField(max\_length=255)

    email = models.EmailField(unique=True)

    phone = models.CharField(max\_length=255)

    birth\_date = models.DateField(null=True)

    membership = models.CharField(

        max\_length=1, choices=MEMBERSHIP\_CHOICES, default=MEMBERSHIP\_BRONZE

    )

    class Meta:

        db\_table = "store\_customers"

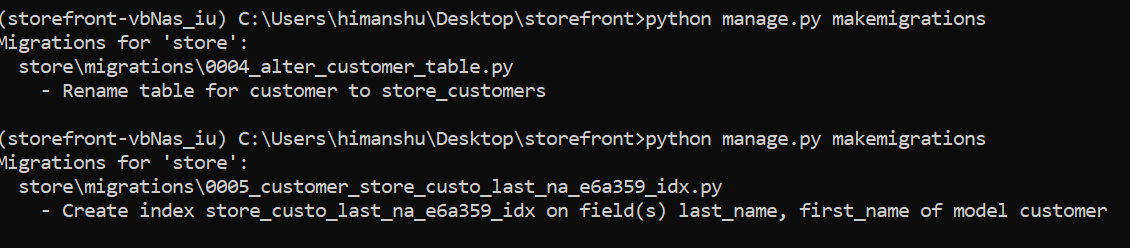
        indexes=[

            models.Index(fields=['last\_name', 'first\_name'])

        ]

Note: An index is an object in database that allows you to find a specific data in a table faster much like an Index section of a book that we refer to when we are trying to search a specific topic.

After adding these changes we run makemigrations command

Note: If we do multiple changes and run makemigrations command, Django will create weird names because it cannot comprehend right name based on both changes.

**Reverting Migrations**:

Let us assume that migrations that we applied before was a mistake. We can undo it in a couple of ways.

*Option 1*:

Remove the changes, save and create a new migration file.

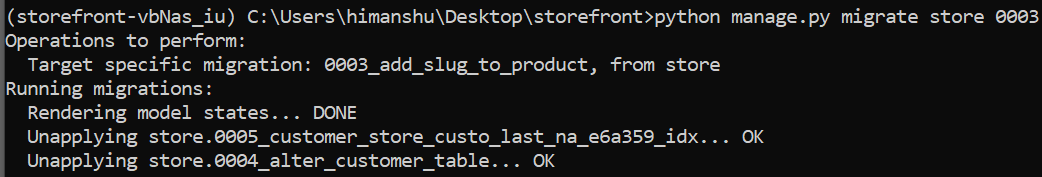
*Option 2*:

Revert the last migration.

Suppose our last migration was 0005 and now I want it to be migration number 0003

Run this command,

python manage.py migrate store 0003



Now our database is back to previous state and if you look at the *migration table*, the last migrations number 0005 and 0004 are gone as well.

But the *changes are still in the code*. The migration files are still in the migration folder and *if we run migrate command again, those files will apply again to our database*.

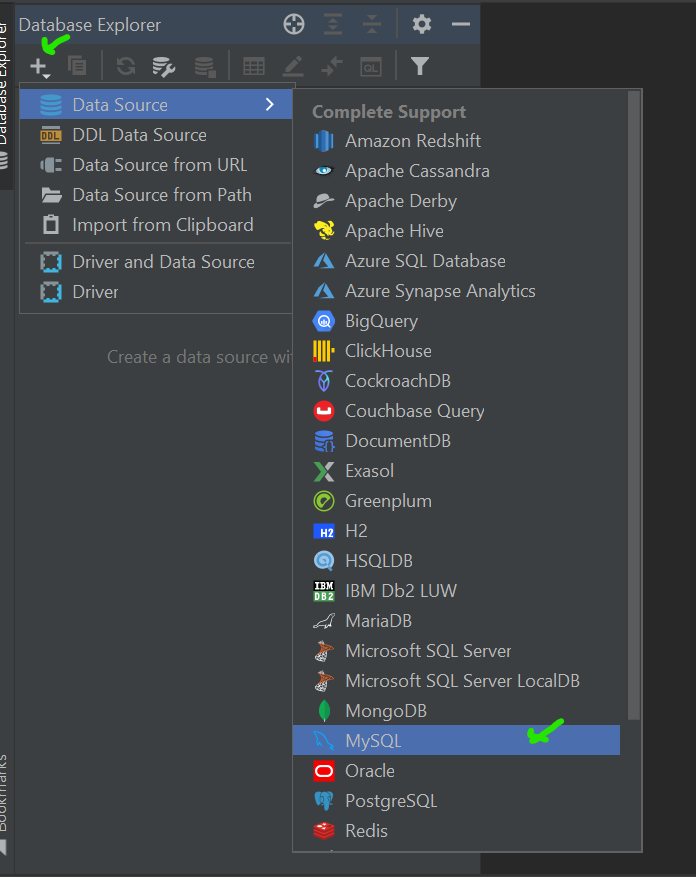
So to properly revert last migration, we should delete the migration file as well as all the changes made in our code.

**Connecting to MySQL**:

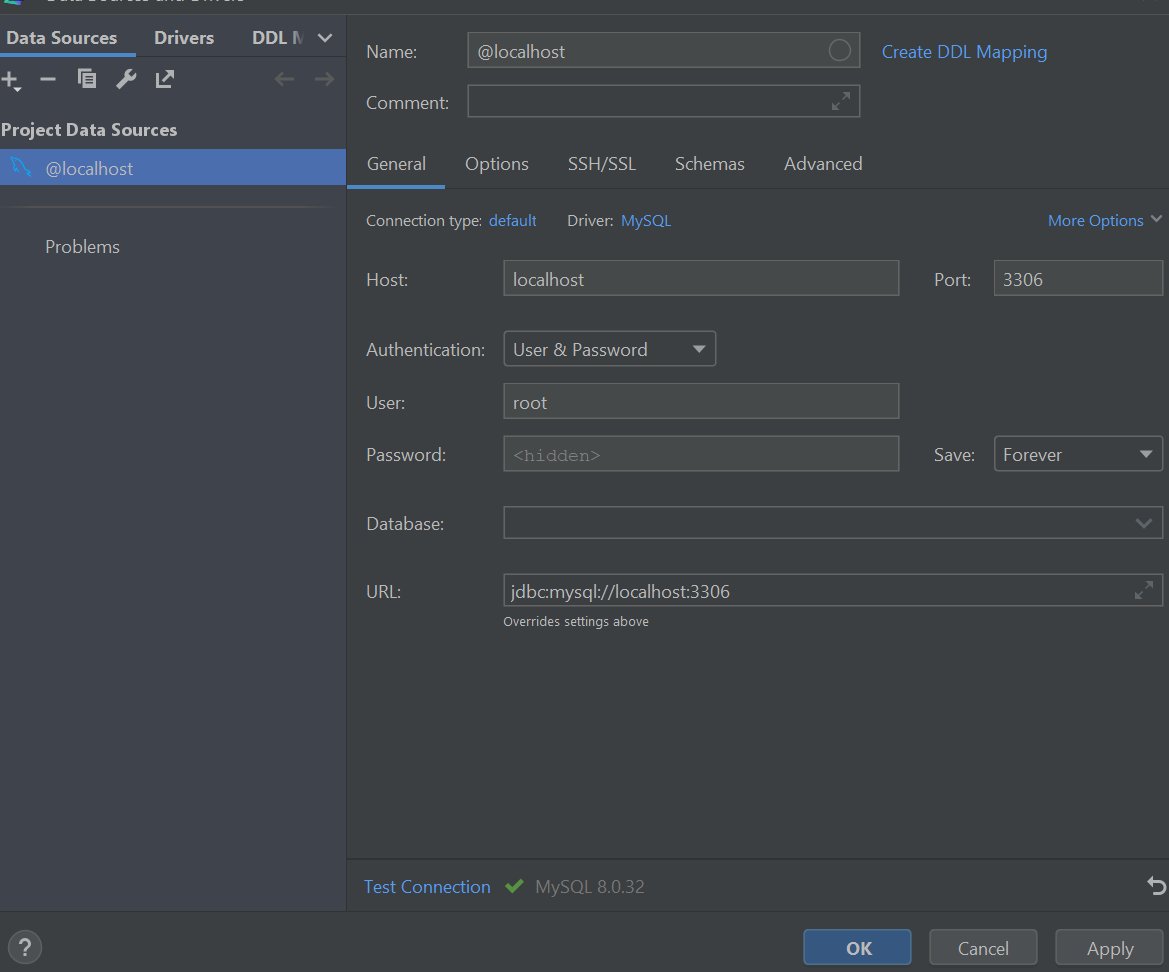
🡪Install DataGrip (it is a paid software)trial version from *JetBrains*.

🡪After installing create a new project called *storefront*.

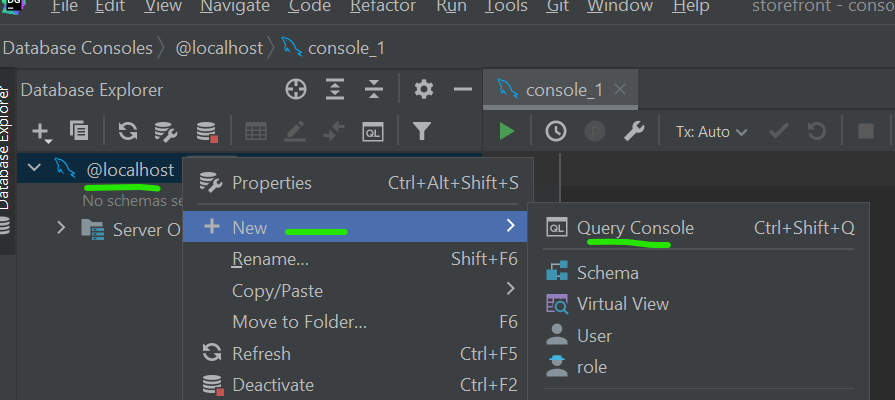
🡪Select a MySQL as Data source



Fill fields and Test connection

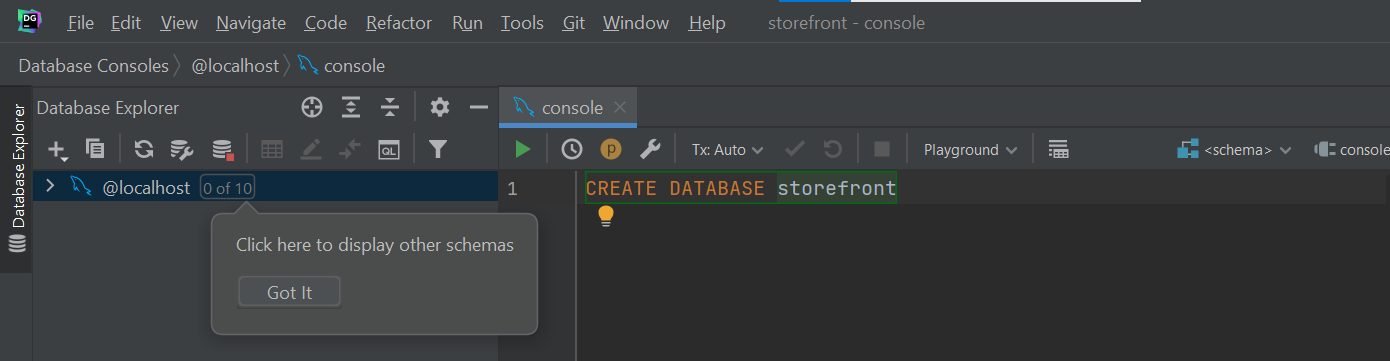


Go to @localhost --🡪New --🡪Query Console



We see the console window where we can write SQL commands.

🡪First command will be to create Database called storefront

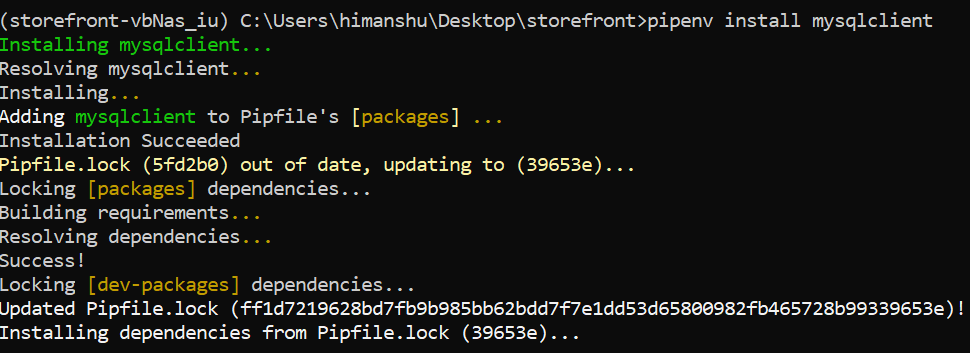


We need to create database first before connecting it to Django.

**Using MySQL in Django**:

To connect our Django project with MySQL we need to install a package called MySQL client.

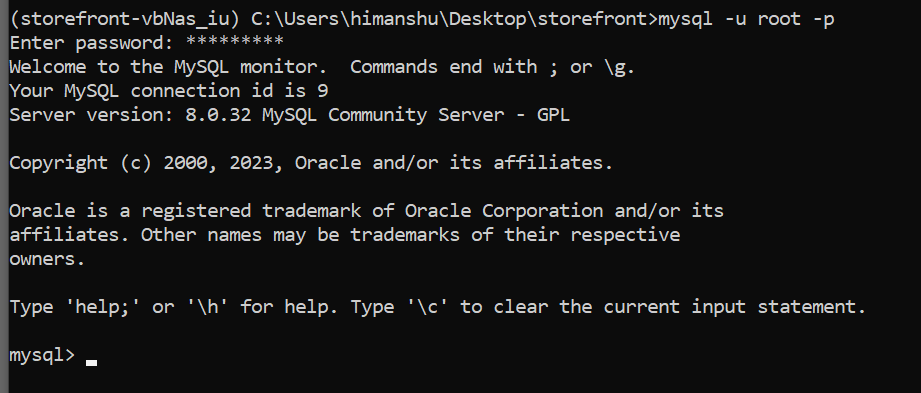
pipenv install mysqlclient



Once you have installed it, you can check if mysql is installed properly or not by using this command on our project terminal window.

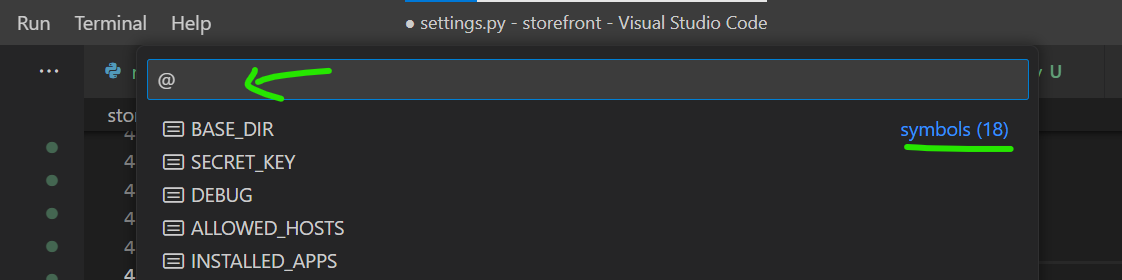
mysql -u root –p

You will be prompt to enter your password. If mysql is installed properly.

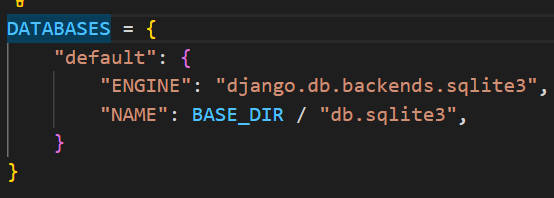


Note: If you see ‘mysql command not found’ on the terminal then *add the path of your mysql server to the environment variable and restart the system*.

Trick: Use ctrl+shift+O command to open all the symbols(variables/classes/functions etc.) in a file. E.g. we can jump to Databases directly from here.



🡪Next thing we need to do is to configure Databases setting in our settings.py module.



We have our default database engine SQLite and name of our SQLite file with its location inside BASE\_DIR or current project directory.

🡪Change the default configuration with these values

DATABASES = {

    "default": {

        "ENGINE": "django.db.backends.mysql",

        "NAME": "storefront",

        "HOST": "localhost",

        "USER": "root",

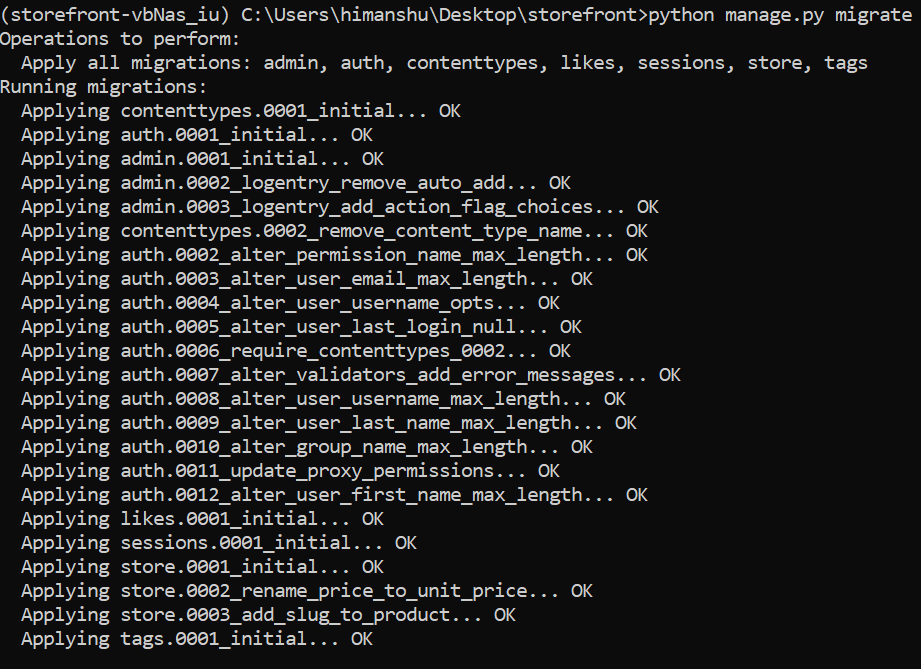
        "PASSWORD": "-------", //password is something else

    }

}

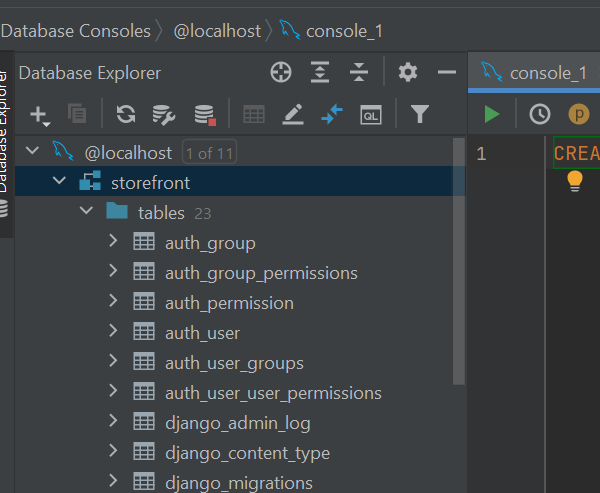
Note: Since this is just for development, we are adding our password as plain text but in production we need to use different strategy to configure our Database setting.

Now run the migrate command



All the migrations are applied to our database.

To confirm go to database and refresh



Our storefront database has these tables inside now.

**Running Customer SQL**:

Sometimes we need full control over generating or updating our Database schema. We can do this easily from Django by using an empty migration.

Suppose, we create an empty migration in our store app using

python manage.py makemigrations store --empty

As a result, we get a migration file with no operations.

class Migration(migrations.Migration):

    dependencies = [

        ('store', '0003\_add\_slug\_to\_product'),

    ]

    operations = [

    ]

Here in operations, we can use migrations.RunSQL which is a class defined in migrations module.

We can create an instance of this class where we provide SQL statements.

Inside the triple quotes, to break down SQL statements into multiple lines, I am writing a SQL command to insert a record in the collection table(*targeting title column and inserting a value called ‘collection1*’)

        migrations.RunSQL("""

            INSERT INTO store\_collection (title)

            VALUES ('collection1')

        """)

Note: Using the same procedure we can also create a stored procedure, a function, view and so on.

The first argument to RunSQL is to upgrade the database. As a best practice we can give an optional second argument which will downgrade the database to previous position. *Because if we fail to do this, we will not be able to revert the migration*.

In second statement just undo previous statement.

from django.db import migrations

class Migration(migrations.Migration):

    dependencies = [

        ("store", "0003\_add\_slug\_to\_product"),

    ]

    operations = [

        migrations.RunSQL(

            """

            INSERT INTO store\_collection (title)

            VALUES ('collection1')

        """,

            """

        DELETE FROM store\_collection WHERE

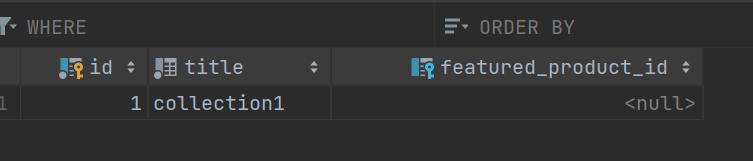
        title='collection1'

        """,

        )

    ]

Now run migrate command and refreshing database we see



Now, let us un-apply the last migration to make sure our implementation is right.

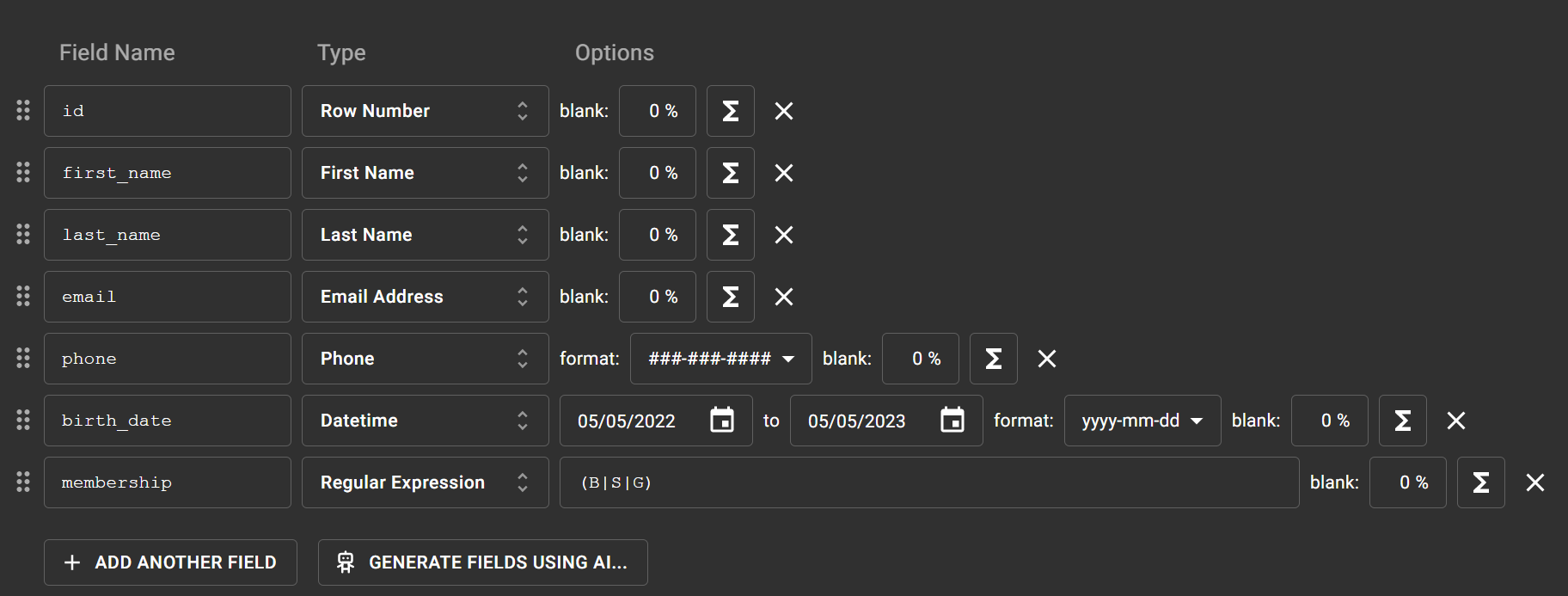
python manage.py migrate store 0003

and we can see we are back to original state of database.

**Generating Dummy Data**:

To generate some dummy data go to this site <https://mockaroo.com/>

Let us populate our customer table using mockaroo

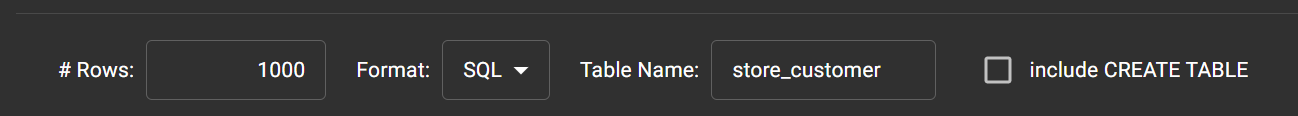


Here we added our field names and their type.

We can specify the format of each field.

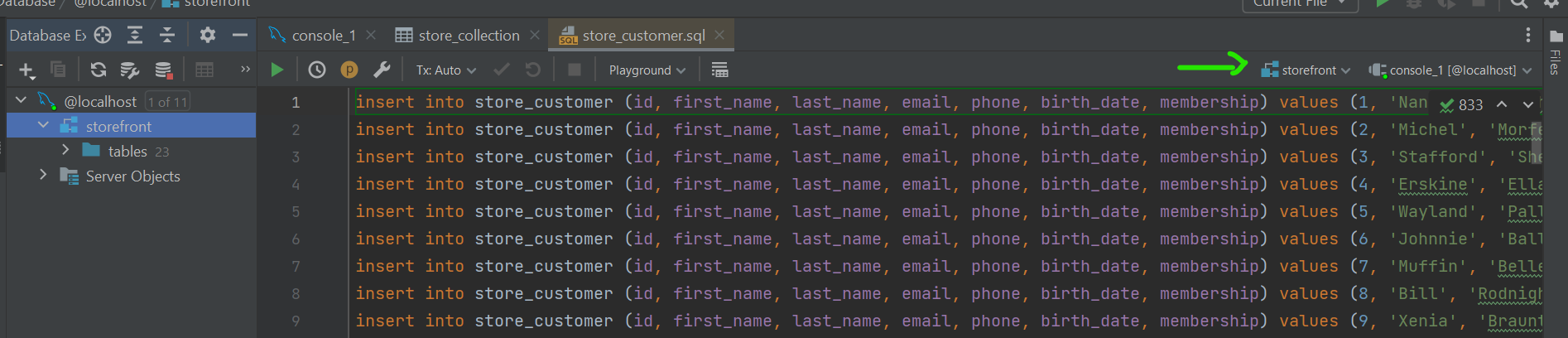
Note: In Membership field we use Regular Expression as the field type and give it values inside parentheses (B|S|G).

After entering all the fields, we can specify number of rows and Format of generated data.



After setting it, generate data and you will see a file name *store\_customer.sql* in your downloads.

Drag and drop this file on our DBMS, make sure our current schema *storefront* is selected.



Then select all Insert into statements and execute them.

Now, let us find number of records in customer table.

SELECT *count*(\*) FROM store\_customer

And we get 1000