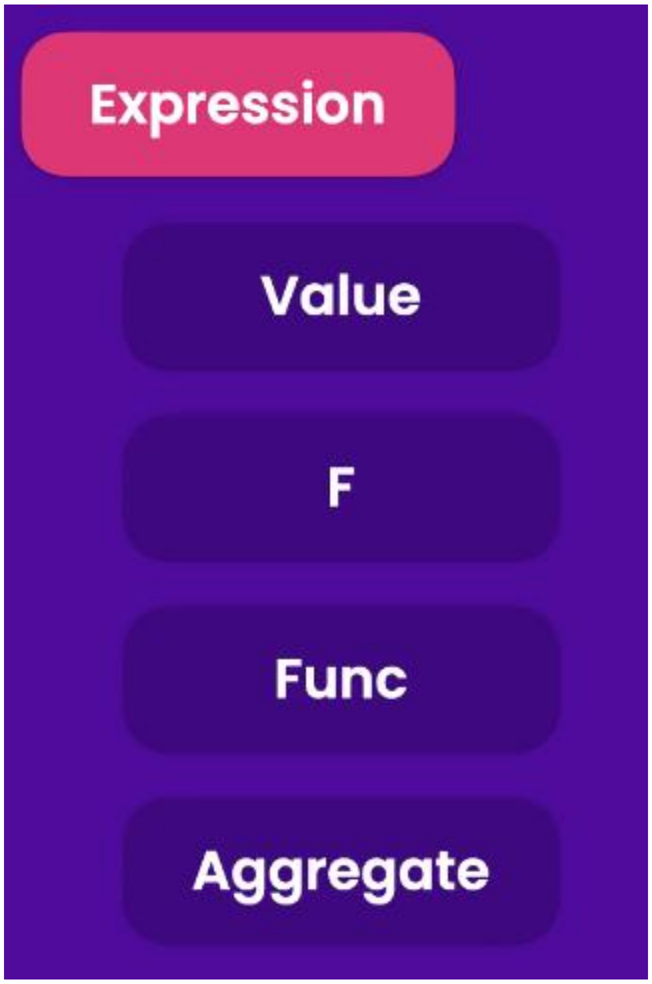
To Make Aggregation Queries, That Enable Us To Make Statistics, We Can use:

```
from django.db.models.aggregates import Count, Sum, Max, Min, Avg
from store.models import Product, OrderItem, Order


def say_hello_26(request):
    products_count = Product.objects\
        .aggregate(
            Count('id'),
            count=Count('id'),
            min_price=Min('unit_price'),
            max_price=Max('unit_price'),
        )

    return render(request, 'hello.html', {
        'products_count': products_count,
        'name': 'Jafar-Loka',
    })

def say_hello_27(request):
    result_1 = Order.objects.aggregate(count=Count('id'))

    result_2 = OrderItem.objects.filter(product__id =
1).aggregate(units_sold=Sum('quantity'))

    result_3 = Order.objects.filter(customer__id=1).aggregate(count=Count('id'))

    result_4 = Product.objects.filter(collection__id = 3)\
    .aggregate(
        min_price   =Min('unit_price'),
        avg_price   = Avg('unit_price'),
        max_price   =Max('unit_price'))

    return render(request, 'hello.html', {
        'name': 'Jafar Loka',
        'result_1': result_1,
        'result_2': result_2,
        'result_3': result_3,
        'result_4': result_4
    })
```
****

If We Want To Add New Fields To The Model Object, We Can Use annotate-Method, That Return Also queryset-object:

Note 1: We Must Pass Expression Object To The New Field.

Note 2: We Can Pass Mixed Of Expressions To The New Field.



****

The Code Like This:

```
from django.db.models import Q, F, Value
def say_hello_28(request):
    queryset = Customer.objects.annotate(is_new=Value(True), new_id=F('id') + 1)

    return render(request, 'hello.html', { 'name': 'Jafar Loka', 'customers':
queryset })
```

****

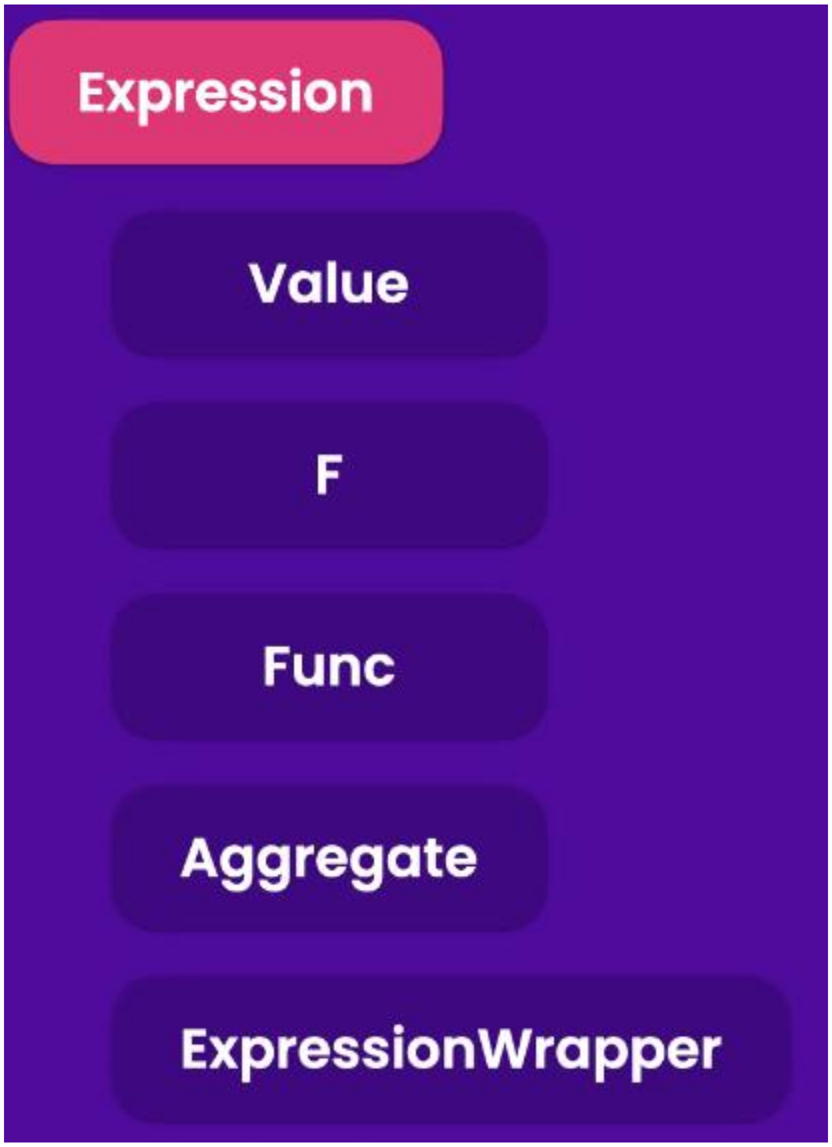
We Can Use annotate-Method, For Creating New Mixed Value Of Fields:

Note: We Can Also, Use Django Database Functions To Make The Same Result.

```
from django.db.models import Q, F, Value, Func
from django.db.models.functions import Concat
def say_hello_28(request):
    queryset = Customer.objects.annotate(
        is_new=Value(True),
        new_id=F('id') + 1,
        full_name= Func(F('first_name'), Value(' '), F('last_name'),
function='CONCAT'),
        full_name_02 = Concat('first_name', Value(' '), 'last_name')
    )

    return render(request, 'hello.html', { 'name': 'Jafar Loka', 'customers':
queryset })
```
****

If We Set Aggregation Function Like: Count, Inside The annotate-Method, Then Django Will Group The Results By The Fields That We Use.

```
def say_hello_29(request):
    queryset = Customer.objects.annotate(
        is_new=Value(True),
        new_id=F('id') + 1,
        full_name= Func(F('first_name'), Value(' '), F('last_name'),
function='CONCAT'),
        full_name_02 = Concat('first_name', Value(' '), 'last_name'),
        order_count = Count('order')
    )
    return render(request, 'hello.html', { 'name': 'Jafar Loka', 'customers':
queryset })
```
****


Expression
Value
F
Func
Aggregate
ExpressionWrapper

****

To Make Calculation Between Different Types Of Models:

```
from django.db.models import Q, F, Value, Func, ExpressionWrapper, DecimalField

def say_hello_30(request):
    queryset = Product.objects.annotate(
        discount_price = ExpressionWrapper(
            F('unit_price') * 0.8,
            output_field=DecimalField())
    )

    return render(request, 'hello.html', { 'name': 'Jafar Loka', 'products_02': 
queryset })
```

****

To Get The Data From ContentType Fields:

```
from django.contrib.contenttypes.models import ContentType
from tags.models import TaggedItem


def say_hello_31(request):
    content_type = ContentType.objects.get_for_model(Product)

    # Here We Use select_related To Avoid The Problems Of
    # Loading Many Tags
    tagItems = TaggedItem.objects\
        .select_related('tag')\
        .filter(
            content_type = content_type,
            object_id=1
    )

    return render(request, 'hello.html', {'name': 'Jafar Loka', 'tagsItem': 
tagItems})
```

****

We Must Understand The Query Set Cache.

Cache Will Be Happened Only When The Entire QuerySet Evaluated, **Ex**:

list(queryset)

queryset[0]

****

To Create Custom Manger For Specific Model:

Note: get_tags_for Is Custom Method For Getting The Data That We Want.

```
class TaggedItemManager(models.Manager):
    def get_tags_for(self, obj_type, obj_value):
        content_type = ContentType.objects.get_for_model(obj_type)

        # Here We Use select_related To Avoid The Problems Of
        # Loading Many Tags
        return TaggedItem.objects\
            .select_related('tag')\
            .filter(
                content_type = content_type,
                object_id=obj_value
        )
```
****

Then We Override The Objects-Attribute Of The Model:

```
class TaggedItem(models.Model):
    objects = TaggedItemManager()
```
****

To Save The Collection Data, That Maybe Come With Request OR From Coder:

Note 1: Like Laravel; Collection-Class Has Also create-Method, But It Doesn't Have Auto-Complete For Attributes

Note 2: Also; For Saving Featured Product, We Can use collection_featured_product_id=1

```
def save_collection_example_1(request):
    collection = Collection()

    collection.title = "Video Games"
    collection.featured_product = Product(pk=1)
    collection.save()

    return render(request, 'hello.html', { 'name': 'Jafar Loka' })
```
****

Note: This Will Raise An Error: Manager isn't accessible via Collection instances

```
# collection = collection.objects.select_related('featured_product')
```
****

To Update The Collection-Class With Reading Data, To Populate It:

```
    collection = Collection.objects\
        .select_related('featured_product')\
        .get(pk=16) # In This Way We Can Populate All Fields
    collection.title = "New Video Games-07"
    collection.save()
```

****

To Update The Object Using Managers, Then We Need To Filter First, Else Every Thing Will Be Updated:

Note: Here Update-Method Will Return The Number Of Rows That Updated.

```
def update_collection_example_2(request):
    Collection.objects.filter(pk=18).update(
        title="New Video Games-18",
        featured_product=None
    )

    return render(request, 'hello.html', { 'name':'Jafar Loka' })
```

****

For Deleting Object/Objects From Table, We Can use *delete-Method* For Objects OR QuerySets.

If We Use QuerySets, Then We Should Use *Filter()-Method*.

****

To Create Transactions, That Can Be Used For Committing OR Rollbacking:

```
from django.db import transaction
# This Can Be Used AS Decorator OR Using With-Keyword
@transaction.atomic()
def test_transaction_example_1(request):
    order = Order()
    order.customer_id = 1
    order.save()

    orderItem = OrderItem()
    orderItem.order = order
    orderItem.product_id = 1
    orderItem.quantity = 1
    orderItem.unit_price = 1
    orderItem.save()

    return render(request, 'hello.html',{ 'name': 'Jafar Loka' })
```

****

For More Control On The Transaction Operator, We Use Context Manger Using with-Keyword:

```
# This Can Be Used AS Decorator OR Using With-Keyword
def test_transaction_example_1(request):

    with transaction.atomic():
        order = Order()
        order.customer_id = 1
        order.save()

        orderItem = OrderItem()
        orderItem.order = order
        orderItem.product_id = 1
        orderItem.quantity = 1
        orderItem.unit_price = 1
        orderItem.save()

    return render(request, 'hello.html',{ 'name': 'Jafar Loka' })
```

****

To Execute RAW SQL, We Can Use raw-Method, That Return Raw QuerySet Object, Not Query Set Object.

```
def test_raw_sql(request):
    products = Product.objects.raw('SELECT * FROM store_order')

    return render(request, 'hello.html', {'name': 'Jafar Loka', 'raw_result':
products})
```

****

To Use Stored Procedure We Can Use Cursor:

```
from django.db import connection


def test_raw_sql_2(request):
    # cursor = connection.cursor()
    # products = cursor.execute('SELECT id, title FROM store_product')
    # row = cursor.fetchone() # In This Way We Return The Result
    # cursor.close() # In Production We Use try-except-finally Block,
    # And Close The Cursor In Finally-Block, OR We Can Use
    # with-Keyword For Context Manager.

    # The Best Way To Use Cursor, By Using with-Keyword
    with connection.cursor() as cursor:
        cursor.callproc('Procedure Name Here', params=[1, 2, 'a'])
    return render(request, 'hello.html', {'name': 'Jafar Loka'})
```

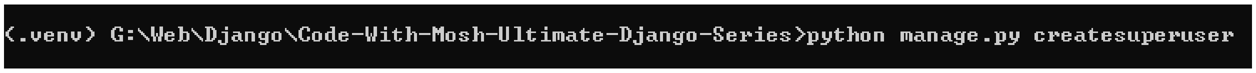
****

To Access The Admin Interface Of Our Project, Go To URL: http://project.url.com:port_if_exists/admin

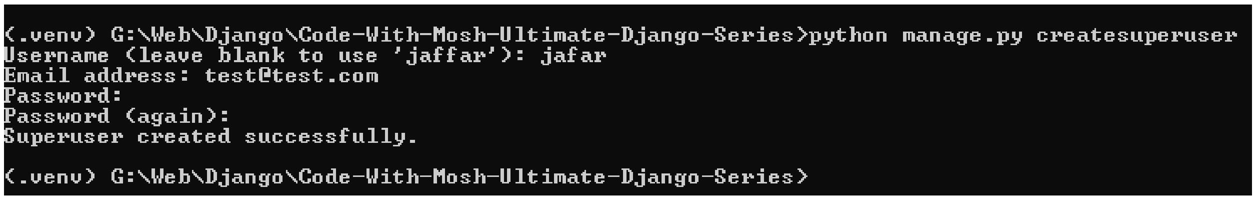
- Ex: http://localhost:8000/admin

****

To Create User For Admin Interface, Run Command: *python manage.py createsuperuser*

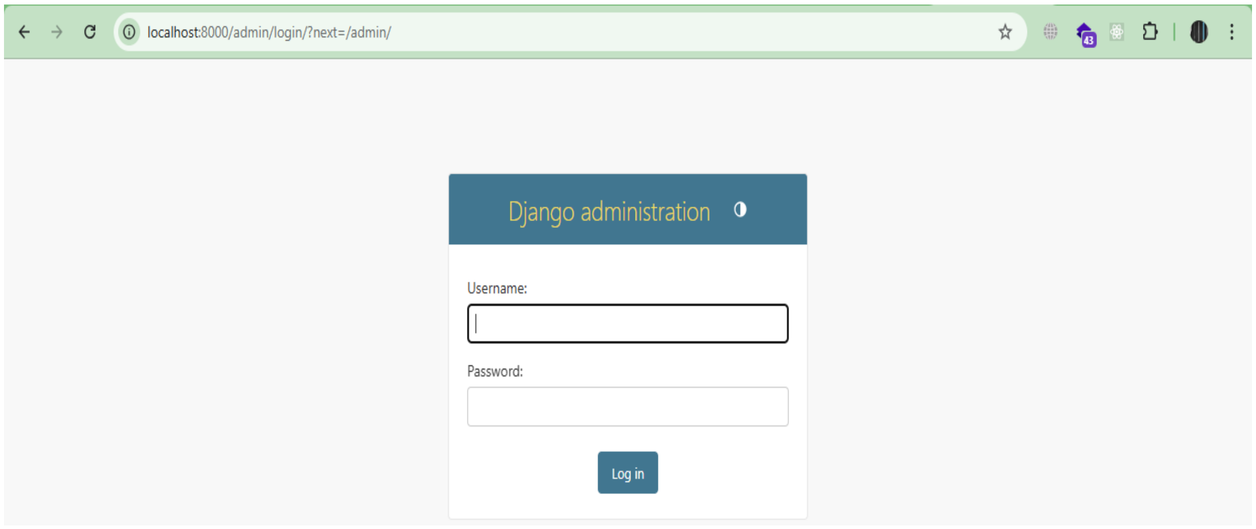


```
(.venv) G:\Web\Django\Code-With-Mosh-Ultimate-Django-Series>python manage.py createsuperuser
```

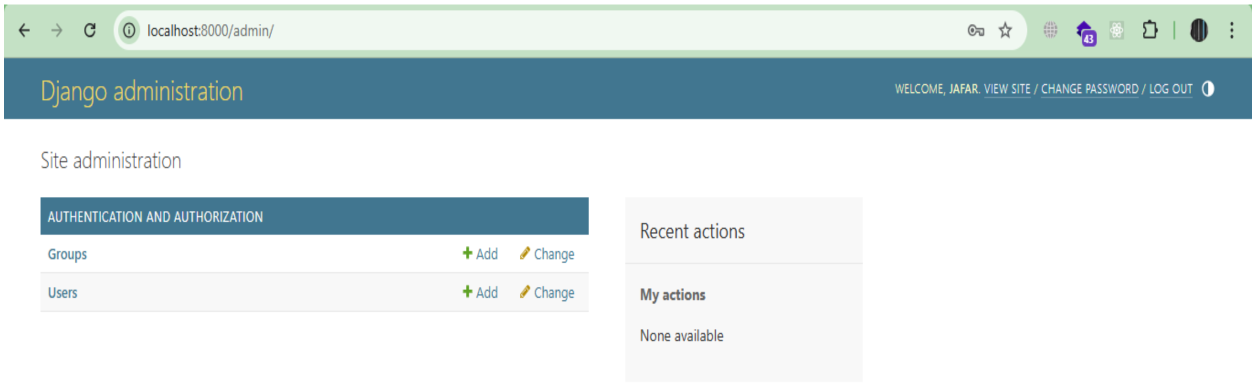


```
(.venv) G:\Web\Django\Code-With-Mosh-Ultimate-Django-Series>python manage.py createsuperuser
Username (leave blank to use 'jaffar'): jafar
Email address: test@test.com
Password:
Password (again):
Superuser created successfully.

(.venv) G:\Web\Django\Code-With-Mosh-Ultimate-Django-Series>
```

****



****



Django administration

WELCOME, JAFAR. VIEW SITE / CHANGE PASSWORD / LOG OUT

Site administration

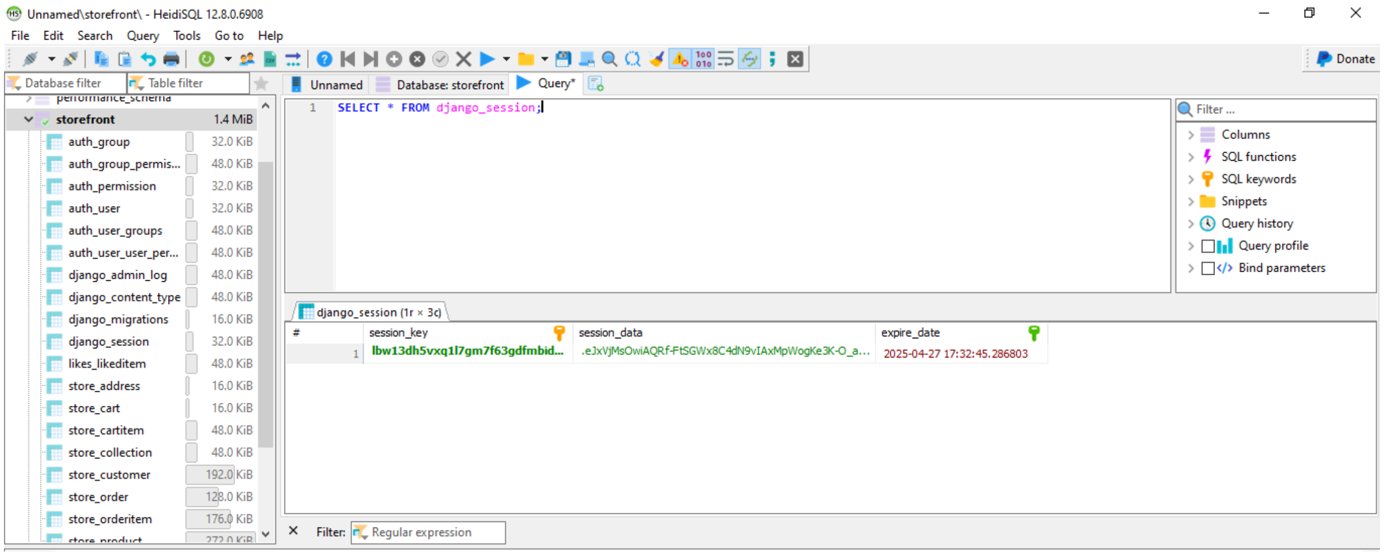
AUTHENTICATION AND AUTHORIZATION		
Groups	+ Add	Change
Users	+ Add	Change

Recent actions

My actions

None available

****



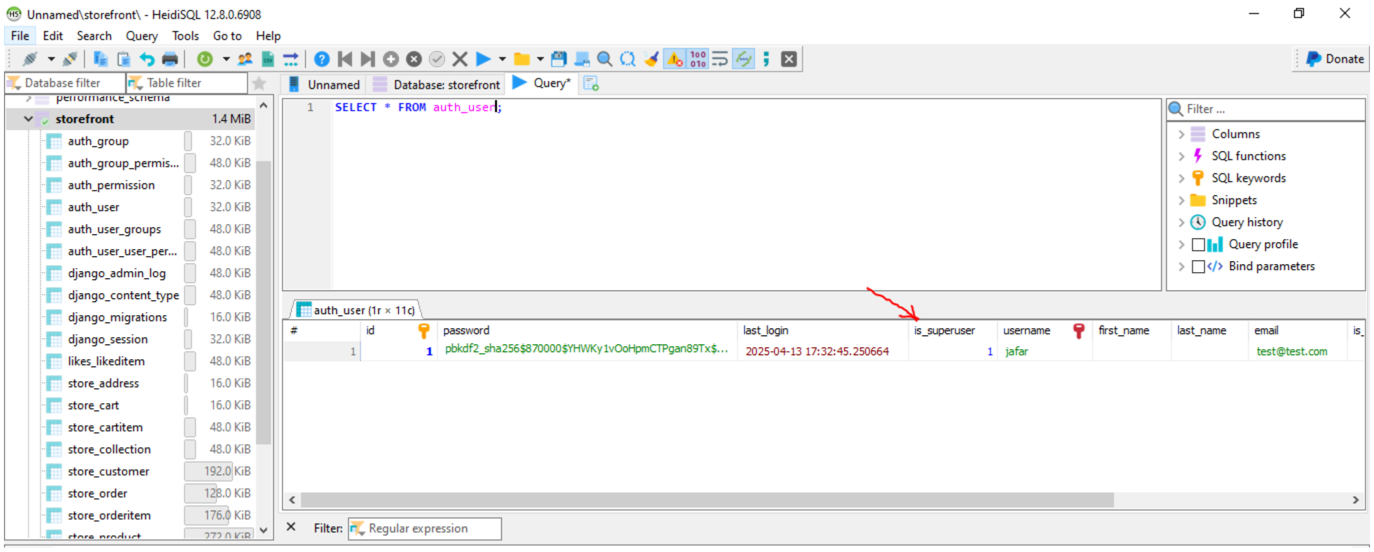
storefront	1.4 MiB
auth_group	32.0 KiB
auth_group_permis...	48.0 KiB
auth_permission	32.0 KiB
auth_user	32.0 KiB
auth_user_groups	48.0 KiB
auth_user_user_per...	48.0 KiB
django_admin_log	48.0 KiB
django_content_type	48.0 KiB
django_migrations	16.0 KiB
django_session	32.0 KiB
likes_likeditem	48.0 KiB
store_address	16.0 KiB
store_cart	16.0 KiB
store_cartitem	48.0 KiB
store_collection	48.0 KiB
store_customer	192.0 KiB
store_order	128.0 KiB
store_orderitem	176.0 KiB

```sql
SELECT * FROM django_session;
```

django_session (1r × 3c)

#	session_key	session_data	expire_date
1	lbw13dh5vxq1l7gm7f63gdfmbid...	.eJxVjMsOwiAQRf-FtSGWx8C4dN9vIAxMpWogKe3K-O_a...	2025-04-27 17:32:45.286803

****



storefront	1.4 MiB
auth_group	32.0 KiB
auth_group_permis...	48.0 KiB
auth_permission	32.0 KiB
auth_user	32.0 KiB
auth_user_groups	48.0 KiB
auth_user_user_per...	48.0 KiB
django_admin_log	48.0 KiB
django_content_type	48.0 KiB
django_migrations	16.0 KiB
django_session	32.0 KiB
likes_likeditem	48.0 KiB
store_address	16.0 KiB
store_cart	16.0 KiB
store_cartitem	48.0 KiB
store_collection	48.0 KiB
store_customer	192.0 KiB
store_order	128.0 KiB
store_orderitem	176.0 KiB

```sql
SELECT * FROM auth_user;
```

auth_user (1r × 11c)

#	id	password	last_login	is_superuser	username	first_name	last_name	email	is_
1	1	pbkdf2_sha256$870000$YHWKy1vOoHpmCTPgan89Tx$...	2025-04-13 17:32:45.250664	1	jafar			test@test.com	

****