Object-Relational Mapping (ELEN4010)

Wisani Salani (366128)

School of Electrical & Information Engineering, University of the Witwatersrand, Private Bag 3, 2050, Johannesburg, South Africa

Abstract: This paper presents the analysis of data-persistent storage offered by object relational mapping (ORM) in a software application. In this paper the advantages of using ORM and the disadvantages ORM poses to the application are discussed. The differences between the object-oriented and relational models are discussed and the need for a mapping is outlined respectively. A Django ORM based application is developed to demonstrate the discussed ORM benefits.

Key words: data-persistent storage, object relational mapping (ORM), object-oriented, relational models

1. INTRODUCTION

Object oriented programming has become the most preferred methodology, and has proven to be effective over the past decade [23]. This has benefitted programming, in delivering large project with minimal time, and better maintenance, greatly attributed to the OO principles.

Software programmes are mainly developed to deal with data, handling of the presented data, interpretation and storing thus becomes are factor to the overall efficiency of the programme [24]. A database is employed to store the data, storing data into and reading the data from the database are the other main activities determining the efficiency of the software application.

2. PROBLEM UNDERSTANDING

ORM is the software application layer concerned with converting object models into target data in a database and vice-versa [24]. Data in an object-oriented language is abstracted as object models, with attributes being the actual static data [25].

But a relational database organizes data in related tables and not in objects, bringing about a mismatch within the applications data flow. The use of relational databases is still prevalent because they are a proven technology and enjoy a huge maintenance support from big corporations over object oriented database which store object instead [26]. At the introduction of OO databases, relational databases had already been embraced greatly by the market, that moving to OO databases posed to be time and resources consuming than to develop a layer to match object models to relational databases.

The main difference between the Object oriented models and the relational models is the paradigm underlying the architecture [22].

2.1. Review of existing ORM frameworks

They all map object instances into relational database tables. Relational links are inherited from the models. Most java ORMs follow particular java standards for building data objects, maintaining data persistence, and abstracting the normal structured-query-language (SQL). Python ORMs, tend to handle all these within their libraries.

2.2. Primary Features

Database abstraction to interface the application with different relational databases available with minimal code change. The ORM should support multiple databases while offering one generic interface to the code base for interacting with all supported databases. Switching databases should at most only include changing settings within the ORM, or the data model objects, and never in the applications code base.

Harmonization of datatypes between the OO language and the database datatypes, since language datatypes and databases can be inconsistent. This also applies since the relational database uses datatypes for each field, to enforce data integrity [26]. The ORM should be able to perform these on-the-fly in the process of transferring data to-and-from the database itself.

The abstraction of the database language from the codebase, in a way to allow the programmer to develop a complete application with no SQL programming knowledge [26]. Different relational databases have customized SQL queries for some complex activities, which can be cumbersome for the programmer developing the application to be compatible with various relational databases. The queries are then implemented in the programming language used instead.

2.3. Secondary Features

Cache synchronization, to avoid data corruption or loss of data integrity when multiple applications are accessing the same relational database. This is when one application reads an entry, cached by the ORM for use in the application as an object, and the second application is also accessing and updating the entry at the same time. A mismatch in records then results.

Allowing SQL query handling is mainly of great use when dealing with very complex applications, wherein the use of ORM is discouraged, since ORM poses efficiency hick-ups when operating of SQL intensive methods.

Bulk data insertion into the database to increase the applications efficiency. Inserting bulk data does not usually happen during the normal running of the application, if not only at the very start when loading initial data. For faster and efficient bulk insertion with minimal time, one would opt to employ SQL queries instead.

3. IMPLEMENTATION

An application to store, manage and retrieve, contact details of friends, family and relative in a contact book format. The individual contacts are to be saved with basic pernal information, group of relation and physical address information.

The python-Django framework was selected to implement the application, using the default Django ORM for object model mapping.

To allow for extensibility, reuse, and scalability, the models are decoupled into simple basic models, with relational links to each other. The application has a contact model, address model, and group model, with the contact related to one address model instance, and multiple group model instance. Multiple address instances can be saved in the database, but one contact can only point or relate to one. Multiple group instances can be saved, and one contact instance can relate to multiple group instances. The contact model is the main instance of the application. Listing 1is the code implementing the models mentioned above.

The application is to allow saving of contact information, and retrieving respectively, the querying of all available contacts, contacts per address, contacts per group, groups available and addresses available.

3.1. How the framework was used

The django ORM was used map the applications models to a SQlite database, creating a database of tables with models names, the models were mapped to the database tables. Model relations were achieved using foreign keys relations, one-to-many, many-to-many relations.

To create a new instance of a contact, contact details were parsed to the instance constructor, and the other related instances were assigned to the placeholders within the contact instance. This completed the creation of a new contact entry. This was done using the django ORM syntax, which is in pure python programming. Queries were also done in the python programming language, using the django ORM syntax.

4. CRITICAL ANALYSIS

ORM is both a process and a mechanism, determining how objects and their relationships are made persistent in a relational database. The mechanism being the correspondence to pattern implementation as one or more mappings.

4.1. Advantages

ORM helps the development of software applications within a shorter period of time. The development process need not to involve SQL programming, which if not programmed correctly could lead to poor application efficiency. Particularly with python ORM frameworks and a few java frameworks, the final code base can be significantly reduced, since the ORM provides most of the scaffolding to data communications.

Like before, this allows a smooth switching of databases, which the ORM supports, with minimal code changes within the source code.

4.2. Disadvantages

For larger-complex applications, ORM poses reduced performance risk. Mainly because, the OO language queries need to be translated to SQL first, Object mappings are stored within another table, which needs to be referred to every time object data is requested. This will greatly be visible in complex applications. Also, the OO language queries are never as flexible as the SQL language queries, hence some will not be efficiently translated.

Impedance mismatch is easily introduced when the way in which the application is using objects is different and inconsistent with how the data is stored and joined in the relational tables. This can be attributed to the poor translation of some OO language queries to the actual SQL queries.

Development with weak field datatype enforcement in the OO language can easily lead to data integrity issues. The data types within the application models need to be matched to their closest types within the database to avoid these issues.

SQL queries are mostly consistent with all relational databases, which can be worthwhile mastering, but OO languages have multiple ORMs with specific advantages over the other. Learning and mastering them might not always be worthwhile, since another different one might be suitable for the particular application. SQL is always a fall-back-to when the ORM cannot achieve the desired query.

Using an ORM, implies shifting complexity from the database into the actual application code, hence for applications with complexity can slow down the development process.

5. CONCLUSION

Some languages have standards on how the ORM is to be implemented, interfaced with the database, while allowing the extensibility of the interface to the application by the use, for extra lines of code. Python has a generic ORM library compatible with most python frameworks, which delivers consistence to the frameworks and the programming language.

ORMs are efficient for minimal to medium complexity projects, and are mostly useful in the early development of the projects, and usually needs the raw SQL implementation for the complex queries, and to achieve maximum efficiency. ORM alone cannot achieve the best application efficiency possible.

REFERENCES

[1] http://hibernate.org/orm/what-is-an-orm/

[2] W. Yaqing, X. Yangyang, Research Solution of Object-relational Mapping in JAVA Platform,

[3] Koong Wah Yan ; Nagendran M. Perumal ; Tharam Dillon, **Data migration ecosystem for big data invited paper**

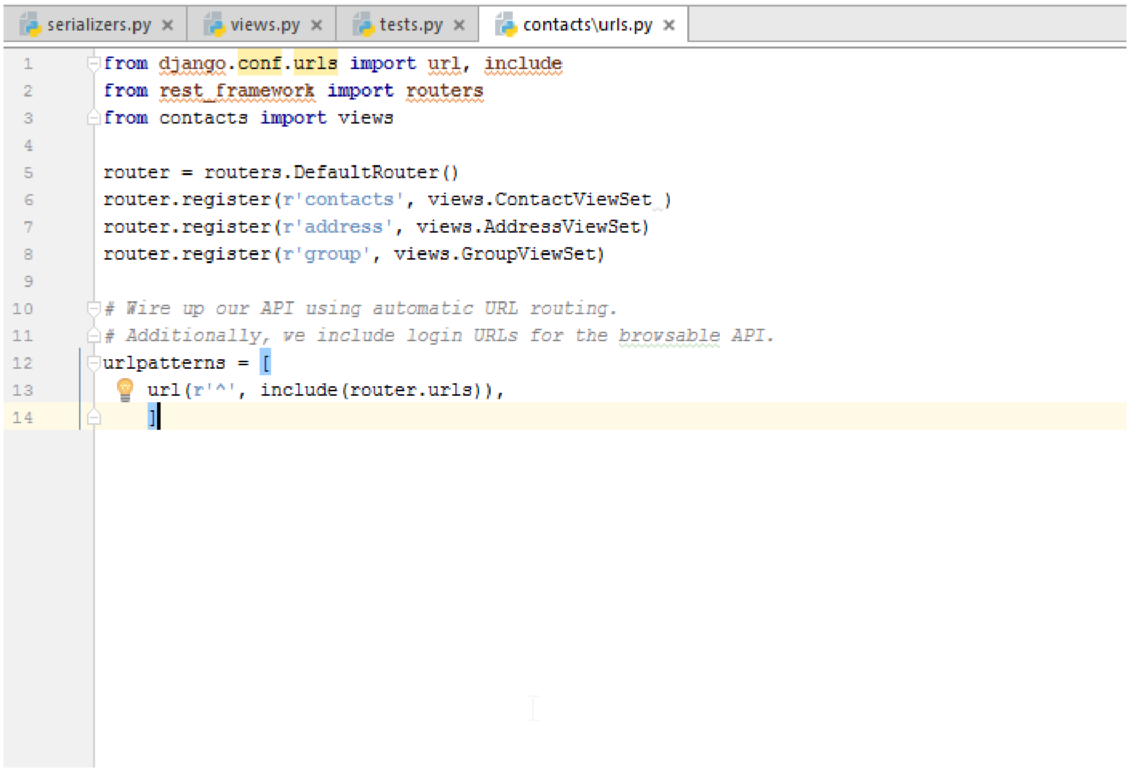
[4] K. Nwosu, B. Thurrisingham, Extending an object oriented data nodel for representing multimedia database application,

[5]

[6] http://www.intersystems.com/library/library-item/the-failure-of-relational-database-the-rise-of-object-technology-and-the-need-for-the-hybrid-database/

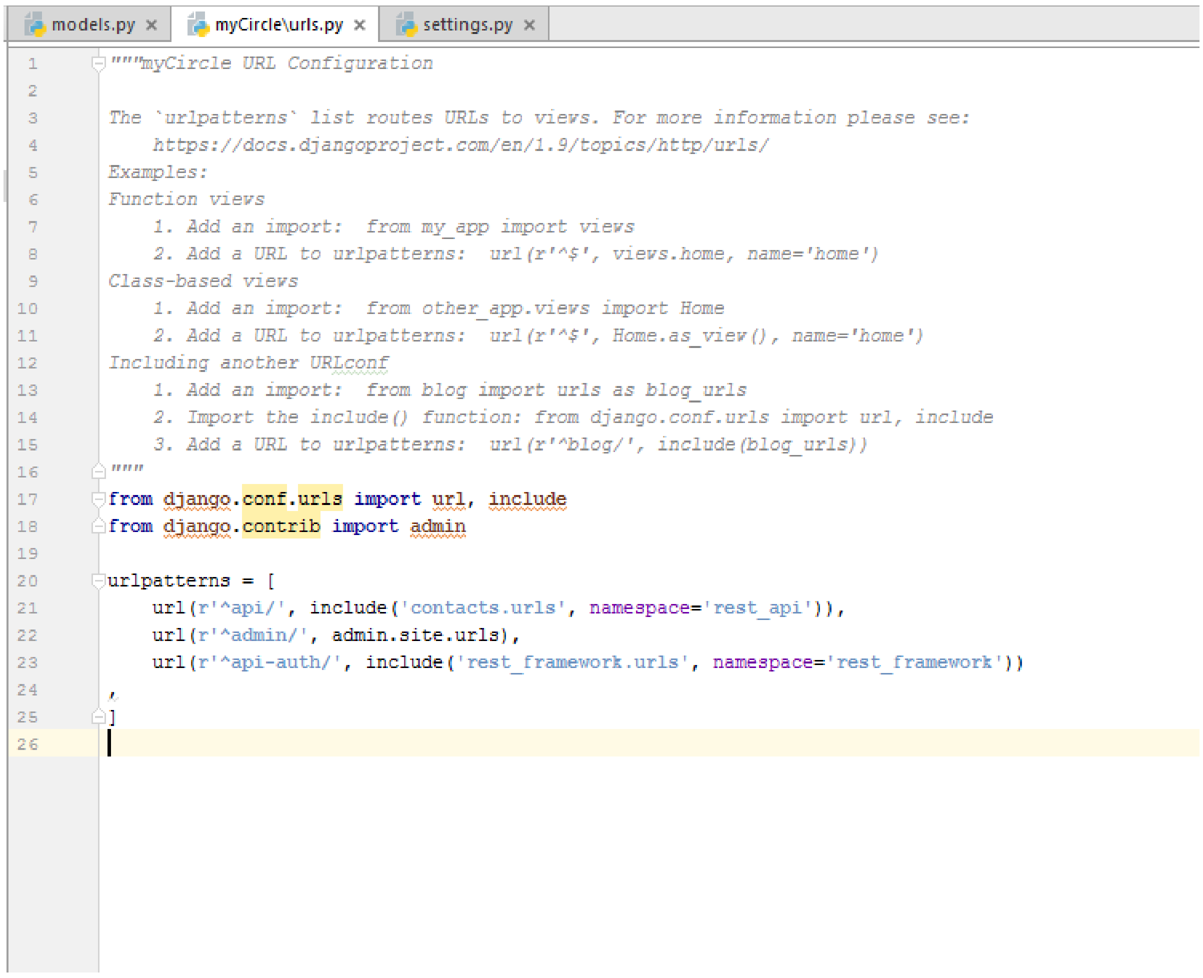
[7] https://www.fullstackpython.com/object-relational-mappers-orms.html

APPENDIX



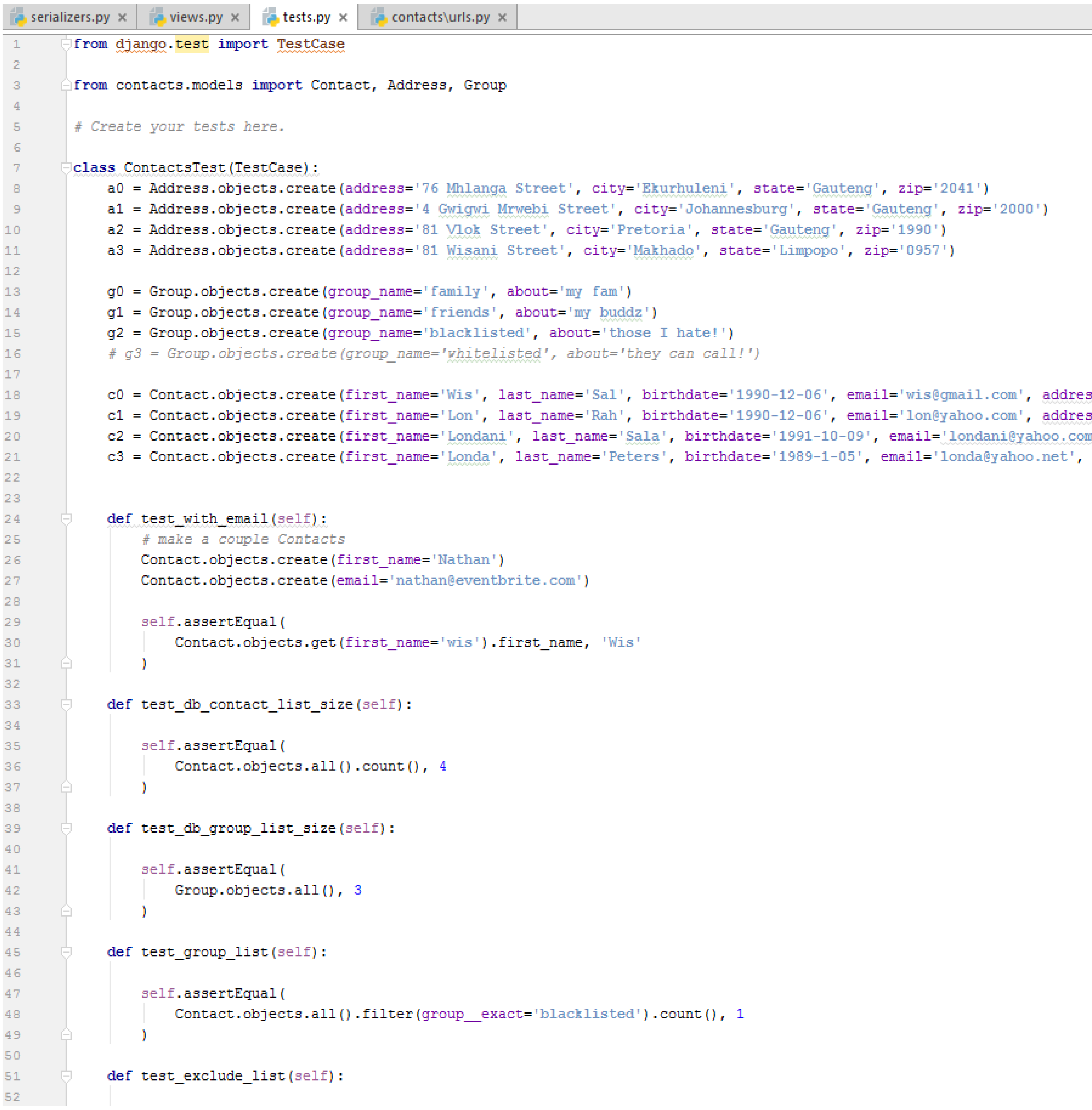
```python
from django.conf.urls import url, include
from rest_framework import routers
from contacts import views

router = routers.DefaultRouter()
router.register(r'contacts', views.ContactViewSet )
router.register(r'address', views.AddressViewSet)
router.register(r'group', views.GroupViewSet)

# Wire up our API using automatic URL routing.
# Additionally, we include login URLs for the browsable API.
urlpatterns = [
    url(r'^', include(router.urls)),
    ]
```

Listing 1 app url configuration



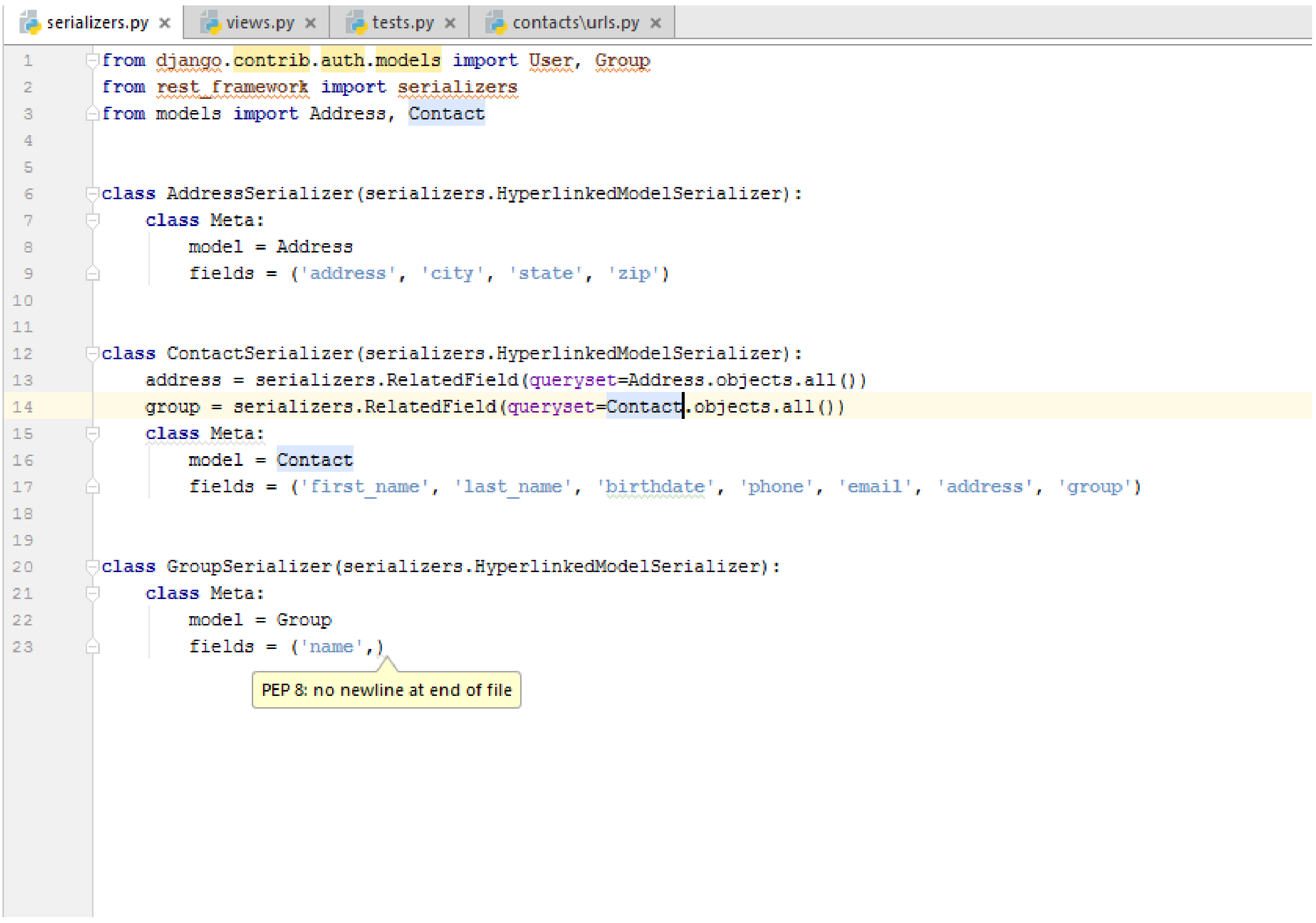
```
"""myCircle URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/1.9/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  url(r'^$', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  url(r'^$', Home.as_view(), name='home')
Including another URLconf
    1. Add an import:  from blog import urls as blog_urls
    2. Import the include() function: from django.conf.urls import url, include
    3. Add a URL to urlpatterns:  url(r'^blog/', include(blog_urls))
"""
from django.conf.urls import url, include
from django.contrib import admin

urlpatterns = [
    url(r'^api/', include('contacts.urls', namespace='rest_api')),
    url(r'^admin/', admin.site.urls),
    url(r'^api-auth/', include('rest_framework.urls', namespace='rest_framework'))
,
]
```

Listing 2Project url config



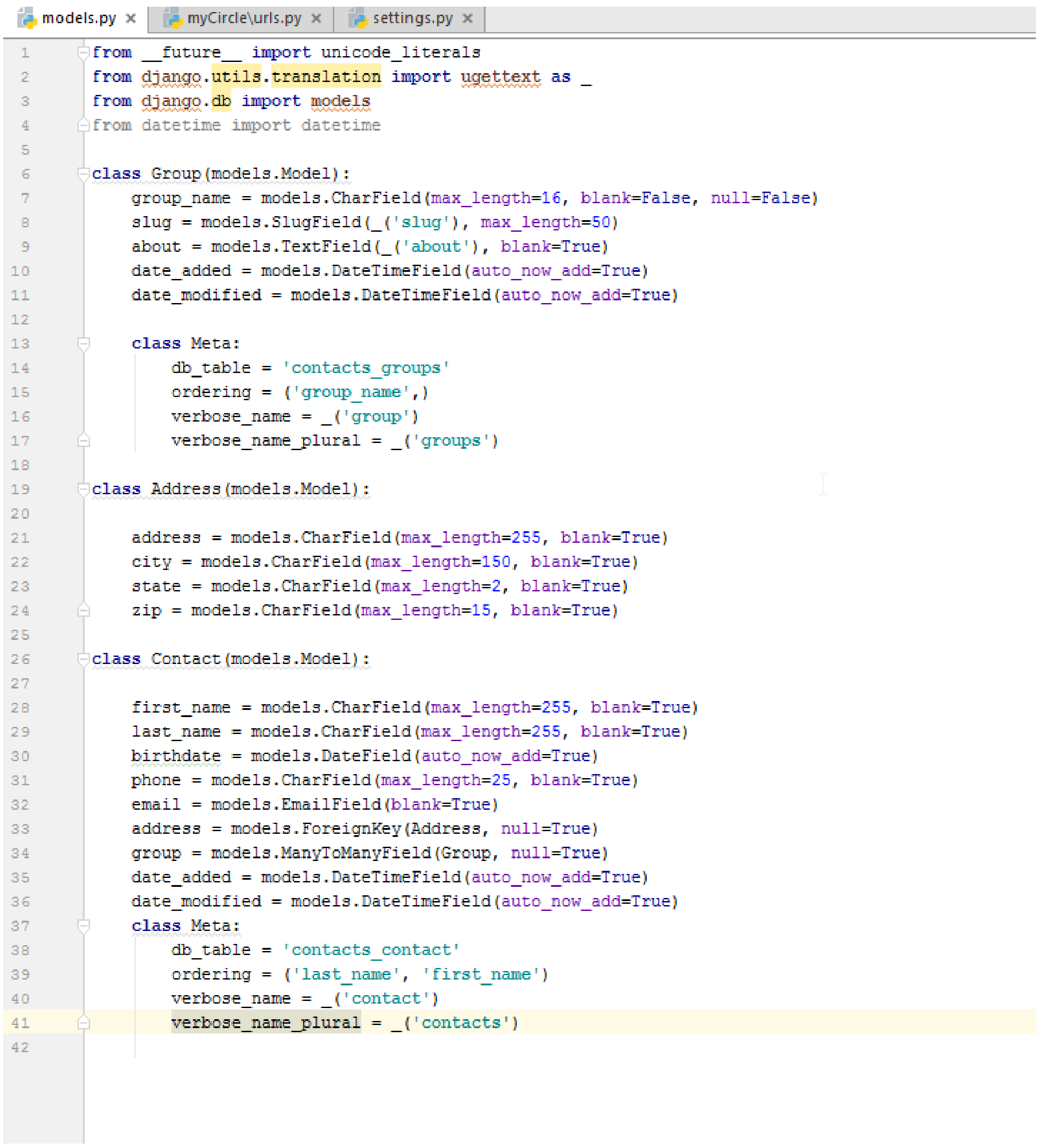
```python
from django.test import TestCase

from contacts.models import Contact, Address, Group

# Create your tests here.

class ContactsTest(TestCase):
    a0 = Address.objects.create(address='76 Mhlanga Street', city='Ekurhuleni', state='Gauteng', zip='2041')
    a1 = Address.objects.create(address='4 Gwigwi Mrwebi Street', city='Johannesburg', state='Gauteng', zip='2000')
    a2 = Address.objects.create(address='81 Vlok Street', city='Pretoria', state='Gauteng', zip='1990')
    a3 = Address.objects.create(address='81 Wisani Street', city='Makhado', state='Limpopo', zip='0957')

    g0 = Group.objects.create(group_name='family', about='my fam')
    g1 = Group.objects.create(group_name='friends', about='my buddz')
    g2 = Group.objects.create(group_name='blacklisted', about='those I hate!')
    # g3 = Group.objects.create(group_name='whitelisted', about='they can call!')

    c0 = Contact.objects.create(first_name='Wis', last_name='Sal', birthdate='1990-12-06', email='wis@gmail.com', addres
    c1 = Contact.objects.create(first_name='Lon', last_name='Rah', birthdate='1990-12-06', email='lon@yahoo.com', addres
    c2 = Contact.objects.create(first_name='Londani', last_name='Sala', birthdate='1991-10-09', email='londani@yahoo.com
    c3 = Contact.objects.create(first_name='Londa', last_name='Peters', birthdate='1989-1-05', email='londa@yahoo.net',


    def test_with_email(self):
        # make a couple Contacts
        Contact.objects.create(first_name='Nathan')
        Contact.objects.create(email='nathan@eventbrite.com')

        self.assertEqual(
            Contact.objects.get(first_name='wis').first_name, 'Wis'
        )

    def test_db_contact_list_size(self):

        self.assertEqual(
            Contact.objects.all().count(), 4
        )

    def test_db_group_list_size(self):

        self.assertEqual(
            Group.objects.all(), 3
        )

    def test_group_list(self):

        self.assertEqual(
            Contact.objects.all().filter(group__exact='blacklisted').count(), 1
        )

    def test_exclude_list(self):

```

Listing 3test suite



```python
from django.contrib.auth.models import User, Group
from rest_framework import serializers
from models import Address, Contact


class AddressSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Address
        fields = ('address', 'city', 'state', 'zip')


class ContactSerializer(serializers.HyperlinkedModelSerializer):
    address = serializers.RelatedField(queryset=Address.objects.all())
    group = serializers.RelatedField(queryset=Contact.objects.all())
    class Meta:
        model = Contact
        fields = ('first_name', 'last_name', 'birthdate', 'phone', 'email', 'address', 'group')


class GroupSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Group
        fields = ('name',)
```

Listing 4 api serializers



```
from __future__ import unicode_literals
from django.utils.translation import ugettext as _
from django.db import models
from datetime import datetime

class Group(models.Model):
    group_name = models.CharField(max_length=16, blank=False, null=False)
    slug = models.SlugField(_('slug'), max_length=50)
    about = models.TextField(_('about'), blank=True)
    date_added = models.DateTimeField(auto_now_add=True)
    date_modified = models.DateTimeField(auto_now_add=True)

    class Meta:
        db_table = 'contacts_groups'
        ordering = ('group_name',)
        verbose_name = _('group')
        verbose_name_plural = _('groups')

class Address(models.Model):

    address = models.CharField(max_length=255, blank=True)
    city = models.CharField(max_length=150, blank=True)
    state = models.CharField(max_length=2, blank=True)
    zip = models.CharField(max_length=15, blank=True)

class Contact(models.Model):

    first_name = models.CharField(max_length=255, blank=True)
    last_name = models.CharField(max_length=255, blank=True)
    birthdate = models.DateField(auto_now_add=True)
    phone = models.CharField(max_length=25, blank=True)
    email = models.EmailField(blank=True)
    address = models.ForeignKey(Address, null=True)
    group = models.ManyToManyField(Group, null=True)
    date_added = models.DateTimeField(auto_now_add=True)
    date_modified = models.DateTimeField(auto_now_add=True)
    class Meta:
        db_table = 'contacts_contact'
        ordering = ('last_name', 'first_name')
        verbose_name = _('contact')
        verbose_name_plural = _('contacts')

```

Listing 5 app models

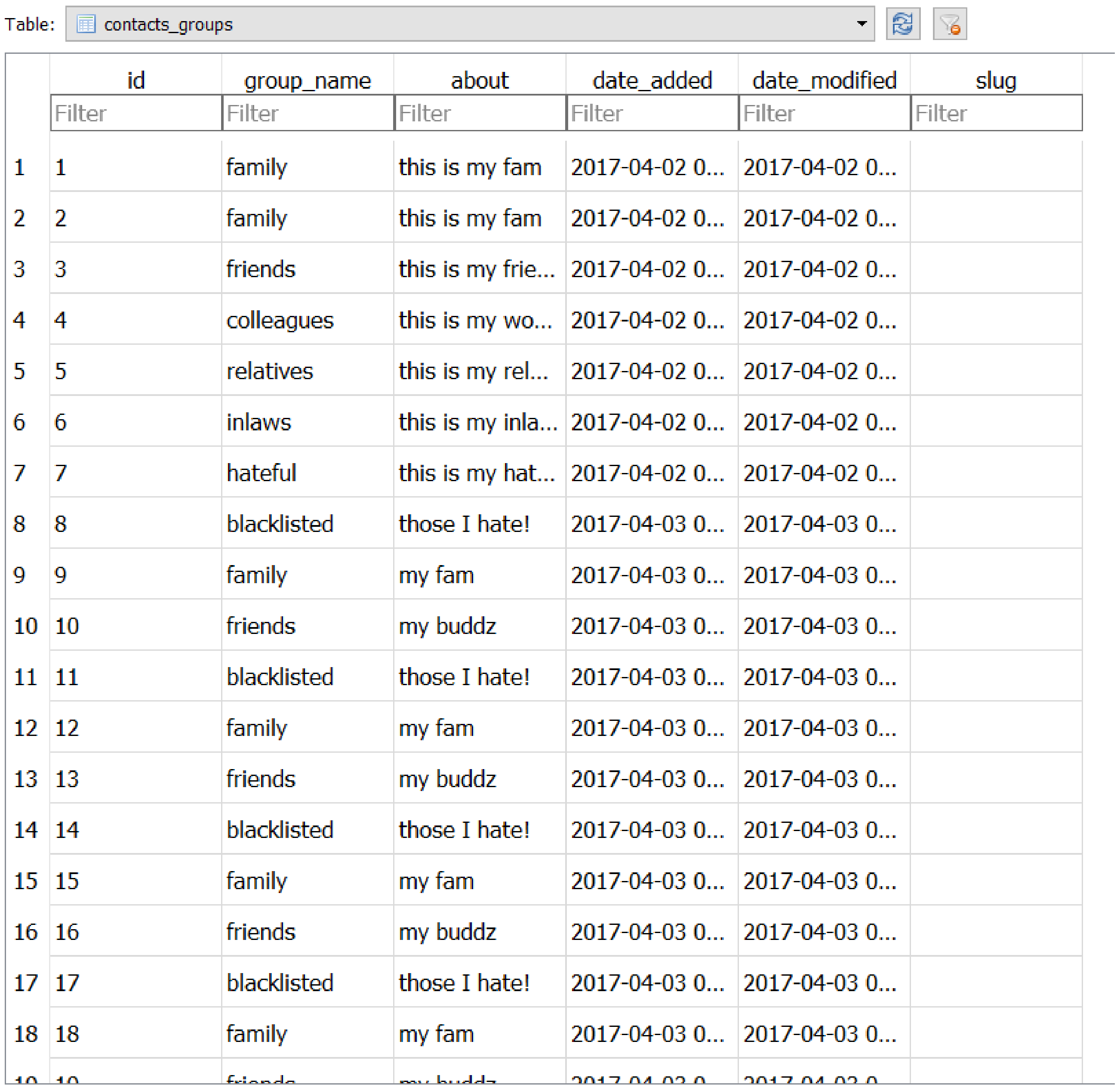


Table: contacts_groups

	id	group_name	about	date_added	date_modified	slug
	Filter	Filter	Filter	Filter	Filter	Filter
1	1	family	this is my fam	2017-04-02 0...	2017-04-02 0...	
2	2	family	this is my fam	2017-04-02 0...	2017-04-02 0...	
3	3	friends	this is my frie...	2017-04-02 0...	2017-04-02 0...	
4	4	colleagues	this is my wo...	2017-04-02 0...	2017-04-02 0...	
5	5	relatives	this is my rel...	2017-04-02 0...	2017-04-02 0...	
6	6	inlaws	this is my inla...	2017-04-02 0...	2017-04-02 0...	
7	7	hateful	this is my hat...	2017-04-02 0...	2017-04-02 0...	
8	8	blacklisted	those I hate!	2017-04-03 0...	2017-04-03 0...	
9	9	family	my fam	2017-04-03 0...	2017-04-03 0...	
10	10	friends	my buddz	2017-04-03 0...	2017-04-03 0...	
11	11	blacklisted	those I hate!	2017-04-03 0...	2017-04-03 0...	
12	12	family	my fam	2017-04-03 0...	2017-04-03 0...	
13	13	friends	my buddz	2017-04-03 0...	2017-04-03 0...	
14	14	blacklisted	those I hate!	2017-04-03 0...	2017-04-03 0...	
15	15	family	my fam	2017-04-03 0...	2017-04-03 0...	
16	16	friends	my buddz	2017-04-03 0...	2017-04-03 0...	
17	17	blacklisted	those I hate!	2017-04-03 0...	2017-04-03 0...	
18	18	family	my fam	2017-04-03 0...	2017-04-03 0...	
19	19	friends	my buddz	2017-04-03 0	2017-04-03 0	

Listing 6 group table

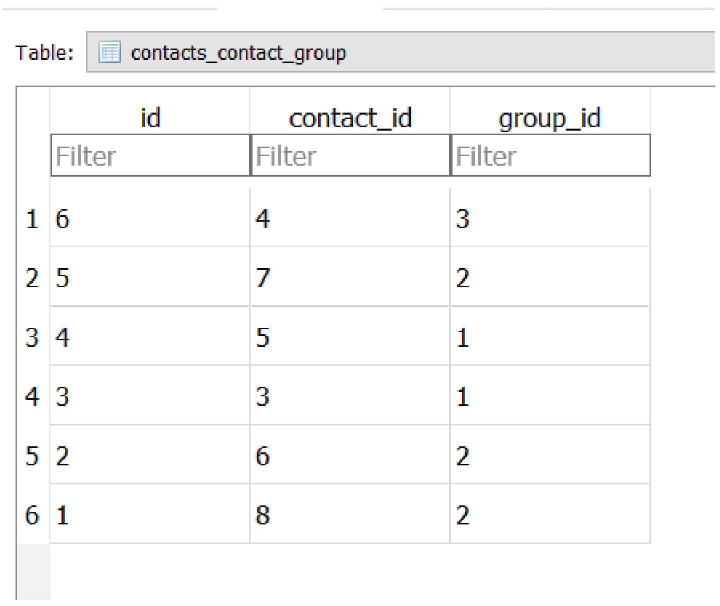


Table: contacts_contact_group

	id	contact_id	group_id
	Filter	Filter	Filter
1	6	4	3
2	5	7	2
3	4	5	1
4	3	3	1
5	2	6	2
6	1	8	2

Listing 7 group to contact mapping

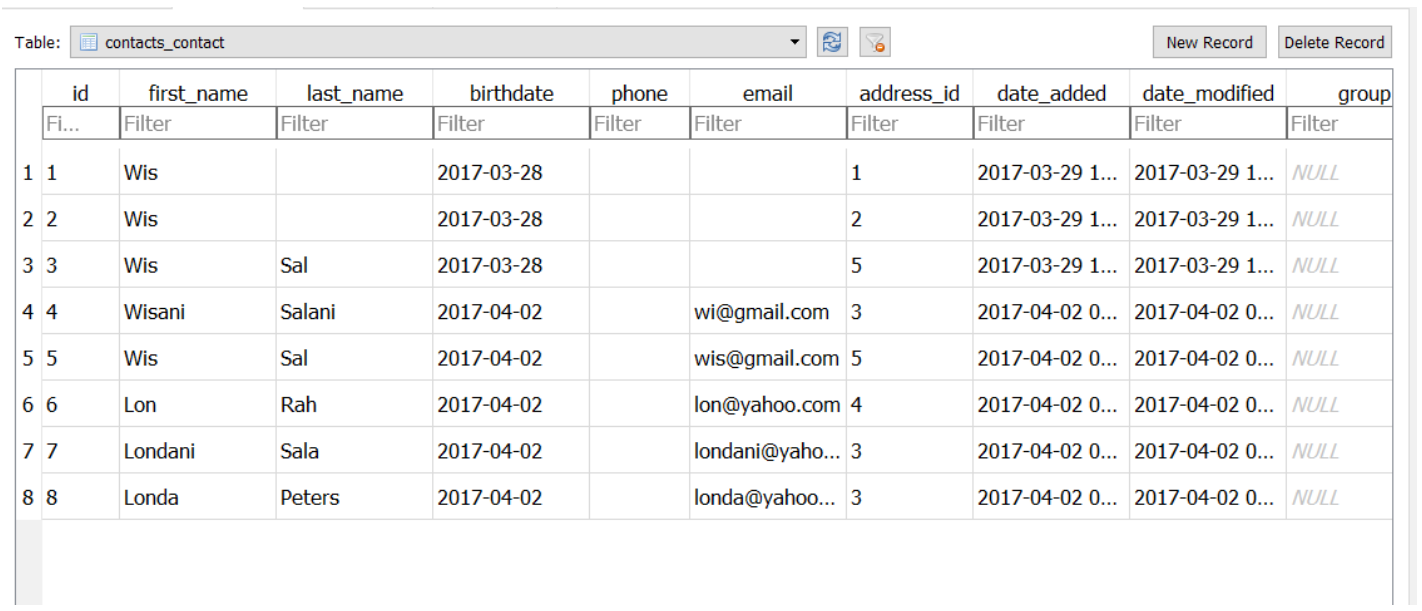


Table: contacts_contact

	id	first_name	last_name	birthdate	phone	email	address_id	date_added	date_modified	group
	Fi...	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	1	Wis		2017-03-28			1	2017-03-29 1...	2017-03-29 1...	NULL
2	2	Wis		2017-03-28			2	2017-03-29 1...	2017-03-29 1...	NULL
3	3	Wis	Sal	2017-03-28			5	2017-03-29 1...	2017-03-29 1...	NULL
4	4	Wisani	Salani	2017-04-02		wi@gmail.com	3	2017-04-02 0...	2017-04-02 0...	NULL
5	5	Wis	Sal	2017-04-02		wis@gmail.com	5	2017-04-02 0...	2017-04-02 0...	NULL
6	6	Lon	Rah	2017-04-02		lon@yahoo.com	4	2017-04-02 0...	2017-04-02 0...	NULL
7	7	Londani	Sala	2017-04-02		londani@yaho...	3	2017-04-02 0...	2017-04-02 0...	NULL
8	8	Londa	Peters	2017-04-02		londa@yahoo...	3	2017-04-02 0...	2017-04-02 0...	NULL

Listing 8 contact list before group map

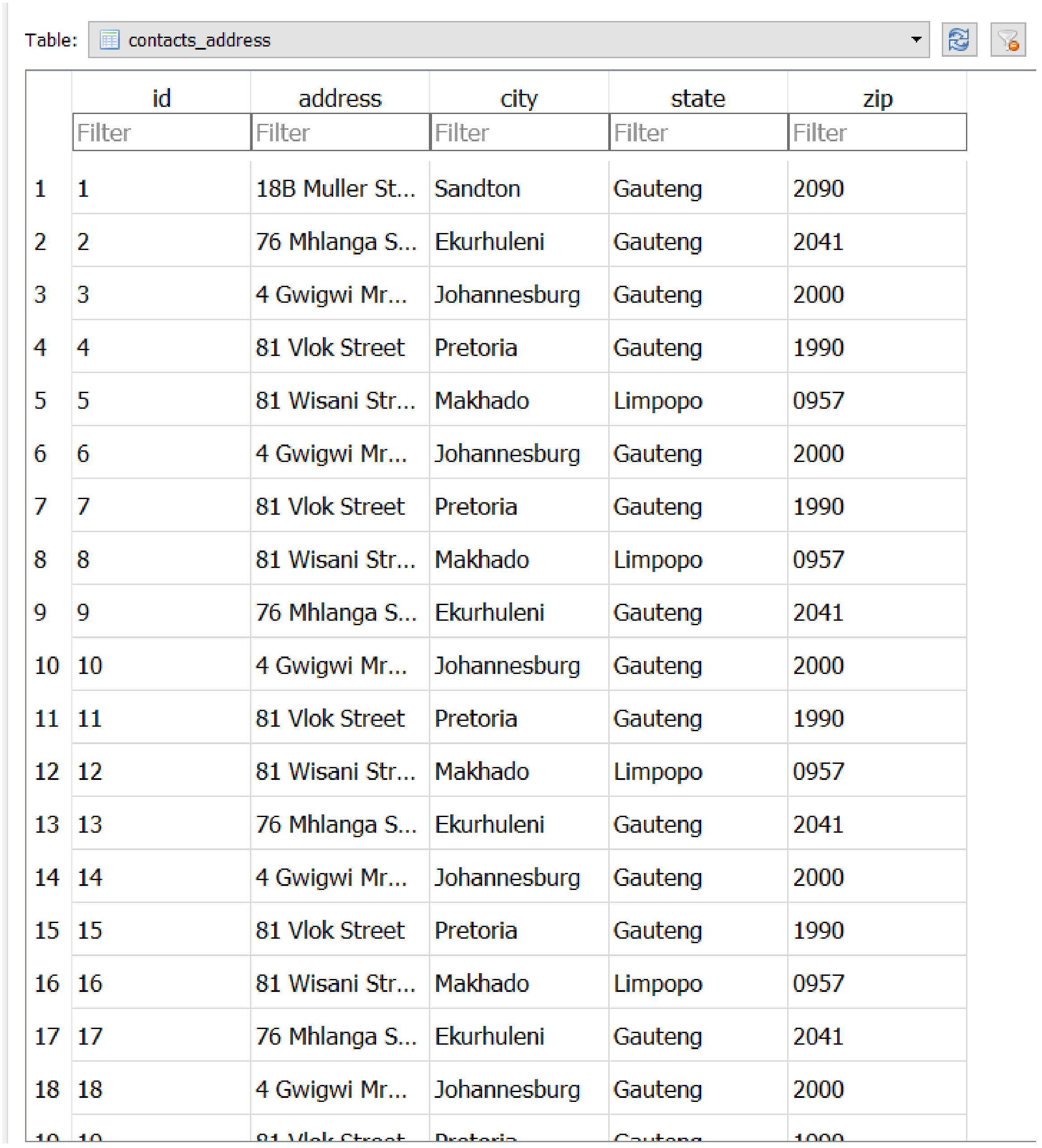
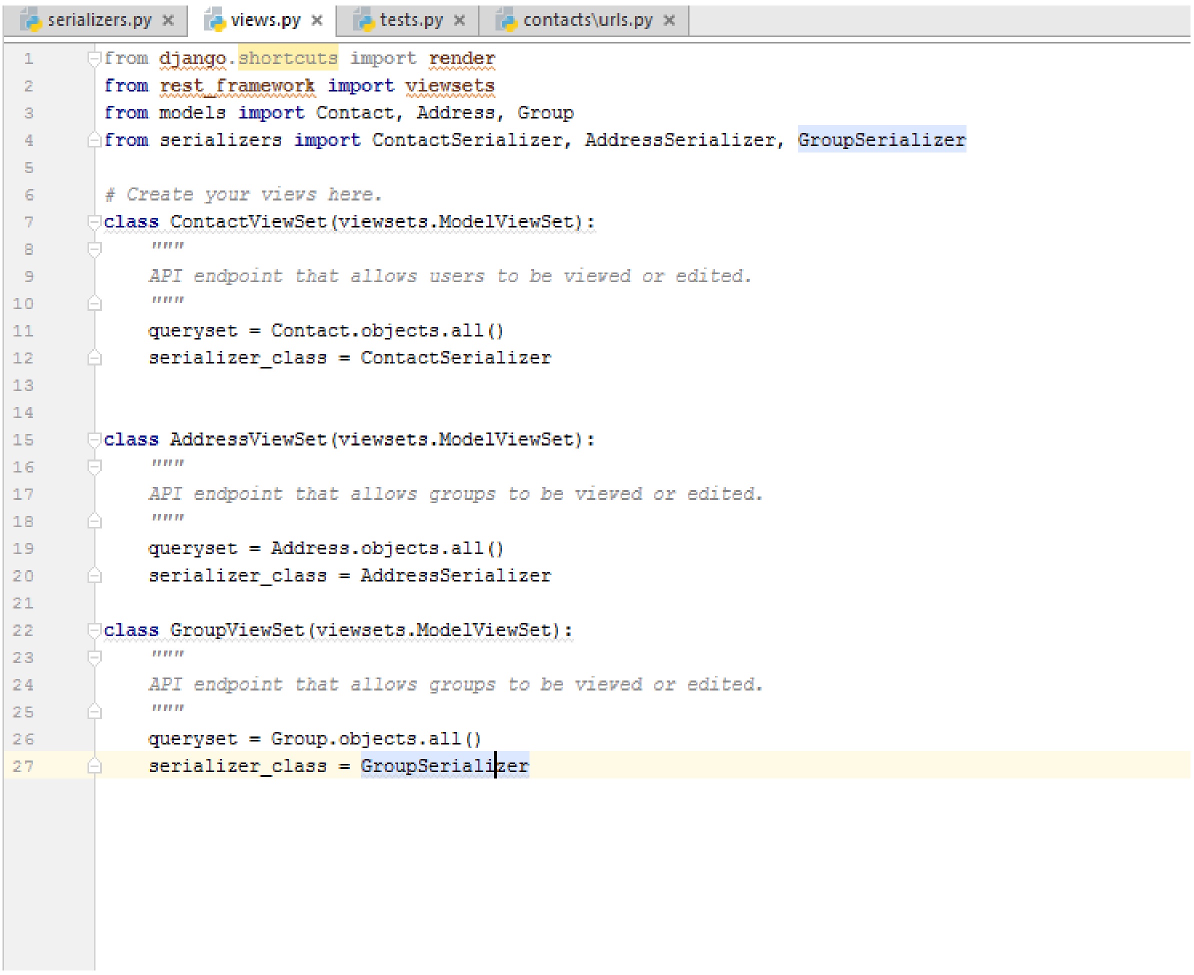


Table: contacts_address

	id	address	city	state	zip
	Filter	Filter	Filter	Filter	Filter
1	1	18B Muller St...	Sandton	Gauteng	2090
2	2	76 Mhlanga S...	Ekurhuleni	Gauteng	2041
3	3	4 Gwigwi Mr...	Johannesburg	Gauteng	2000
4	4	81 Vlok Street	Pretoria	Gauteng	1990
5	5	81 Wisani Str...	Makhado	Limpopo	0957
6	6	4 Gwigwi Mr...	Johannesburg	Gauteng	2000
7	7	81 Vlok Street	Pretoria	Gauteng	1990
8	8	81 Wisani Str...	Makhado	Limpopo	0957
9	9	76 Mhlanga S...	Ekurhuleni	Gauteng	2041
10	10	4 Gwigwi Mr...	Johannesburg	Gauteng	2000
11	11	81 Vlok Street	Pretoria	Gauteng	1990
12	12	81 Wisani Str...	Makhado	Limpopo	0957
13	13	76 Mhlanga S...	Ekurhuleni	Gauteng	2041
14	14	4 Gwigwi Mr...	Johannesburg	Gauteng	2000
15	15	81 Vlok Street	Pretoria	Gauteng	1990
16	16	81 Wisani Str...	Makhado	Limpopo	0957
17	17	76 Mhlanga S...	Ekurhuleni	Gauteng	2041
18	18	4 Gwigwi Mr...	Johannesburg	Gauteng	2000
19	19	81 Vlok Street	Pretoria	Gauteng	1990

Listing 9 address table



serializers.py × views.py × tests.py × contacts\urls.py ×

```
from django.shortcuts import render
from rest_framework import viewsets
from models import Contact, Address, Group
from serializers import ContactSerializer, AddressSerializer, GroupSerializer

# Create your views here.
class ContactViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows users to be viewed or edited.
    """
    queryset = Contact.objects.all()
    serializer_class = ContactSerializer


class AddressViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows groups to be viewed or edited.
    """
    queryset = Address.objects.all()
    serializer_class = AddressSerializer

class GroupViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows groups to be viewed or edited.
    """
    queryset = Group.objects.all()
    serializer_class = GroupSerializer
```

Listing 10 api views