Django Service Objects Documentation

Release 0.7.0

mixxorz, c17r

Apr 23, 2020

User Documentation

CHAPTER 1

What?

This is a small library providing a `Service` base class to derive your service objects from. What are service objects? You can read more about the whys and hows in this blog post, but for the most part, it encapsulates your business logic, decoupling it from your views and model methods. Service objects are where your business logic should go.

CHAPTER 2

Installation

Like every other Python Package

```
$ pip install django-service-objects
```

2.1 Philosophy

2.1.1 Decoupled from presentation logic

Now that your business logic is in its own class, you can call it from anywhere. From regular Django views, JSON endpoints, management commands, RQ tasks, Django admin, etc. Your business logic is no longer bound to your views or models.

2.1.2 Easily testable and mockable

Since service objects are just objects, you can easily call them from your tests to verify their behavior. Similarly, if you're testing your view or endpoint, you can easily mock out the service objects.

2.1.3 Input validation

Your code will become a lot more concise now that you don't have to manually check whether or not a parameter is valid. If you need to know why your inputs failed validation, just catch `InvalidInputsError` and access the `errors` or `non_field_errors` dictionary. Better yet, raise a custom exception from your service object.

2.1.4 But fat models...

Models just ended up doing too many things. Plus it wasn't very clear which model a method belongs to if it operated on two different models. (Should `Booking` have `create_booking()` or should `Customer`)?

2.2 Usage

2.2.1 Service

Super easy. Just:

1. Create a new class that inherits from `Service`
2. Add some fields, exactly like how you would with Django Forms
3. Define a `process()` method that contains your business logic
4. Optionally include a `post_process()` method to perform extra tasks. If using *db_transaction = True*, this will run after the *process* using a Django transaction on commit hook which will only run if the transaction is successfully committed. If using *db_transaction = False*, this will run after the *process* as long as there is no unhandled exceptions. The *post_process* is useful to running a task that should only be run if the process is successful (e.g. send an email, invoke a celery task, etc.).

A code sample is worth a thousand words.

Listing 1: your_app/services.py

```
class CreateBookingService(Service):
    name = forms.CharField()
    email = forms.EmailField()
    checkin_date = forms.DateField()
    checkout_date = forms.DateField()

    def process(self):
        name = self.cleaned_data['name']
        email = self.cleaned_data['email']
        checkin_date = self.cleaned_data['checkin_date']
        checkout_date = self.cleaned_data['checkout_date']

        # Update or create a customer
        customer = Customer.objects.update_or_create(
            email=email,
            defaults={
                'name': name
            }
        )

        # Create booking
        self.booking = Booking.objects.create(
            customer=customer,
            checkin_date=checkin_date,
            checkout_date=checkout_date,
            status=Booking.PENDING_VERIFICATION,
        )

        return self.booking

    def post_process(self):
        # Send verification email (check out django-herald)
        VerifyEmailNotification(self.booking).send()
```

Database transactions

By default, the process method on services runs inside a transaction. This is so that if an exception is raised while executing your service, the database gets rolled back to a clean state. If you don't want this behavior, you can set `db_transaction = False` on the service class.

Listing 2: your_app/services.py

```
class NoDbTransactionService(Service):
    db_transaction = False
```

2.2.2 Function Based View

Listing 3: your_app/views.py

```
from django.shortcuts import redirect, render

from .forms import BookingForm
from .services import CreateBookingService


def create_booking_view(request):
    form = BookingForm()

    if request.method == 'POST':
        form = BookingForm(request.POST)
        if form.is_valid():
            try:
              # Services raise InvalidInputsError if you pass
              # invalid values into it.
              CreateBookingService.execute({
                  'name': form.cleaned_data['name'],
                  'email': form.cleaned_data['email'],
                  'checkin_date': form.cleaned_data['checkin_date'],
                  'checkout_date': form.cleaned_data['checkout_date'],
              })
              return redirect('booking:success')
            except Exception as e:
                form.add_error(None, f'Sorry. Something went wrong: {e}')

    return render(request, 'booking/create_booking.html', {'form': form})
```

2.2.3 Class Based View

Listing 4: your_app/views.py

```
from django.core.urlresolvers import reverse_lazy

from service_objects.views import ServiceView

from .forms import BookingForm
from .services import CreateBookingService


```

(continues on next page)

(continued from previous page)

```
class CreateBookingView(ServiceView):
    form_class = BookingForm
    service_class = CreateBookingService
    template_name = 'booking/create_booking.html'
    success_url = reverse_lazy('booking:success')
```

2.2.4 Testing

An example of testing `CreateBookingService`

Listing 5: your_app/tests.py

```
from datetime import date

from django.core import mail
from django.test import TestCase

from .models import Booking, Customer
from .services import CreateBookingService


class CreateBookingServiceTest(TestCase):

    def test_create_booking(self):
        inputs = {
            'name': 'John Doe',
            'email': 'john@doe.com',
            'checkin_date': date(2017, 8, 13),
            'checkout_date': date(2017, 8, 15),
        }

        booking = CreateBookingService.execute(inputs)

        # Test that a Customer gets created
        customer = Customer.objects.get()
        self.assertEqual(customer.name, inputs['name'])
        self.assertEqual(customer.email, inputs['email'])

        # Test that a Booking gets created
        booking = Booking.objects.get()

        self.assertEqual(customer, booking.customer)
        self.assertEqual(booking.checkin_date, inputs['checkin_date'])
        self.assertEqual(booking.checkout_date, inputs['checkout_date'])

        # Test that the verification email gets sent
        self.assertEqual(1, len(mail.outbox))

        email = mail.outbox[0]
        self.assertIn('verify email address', email.body)
```

2.3 Object Reference

2.3.1 Errors module

exception service_objects.errors.**InvalidInputsError**(*errors*, *non_field_errors*)

Raised during `Service`'s `service_clean()` method. Encapsulates both field_errors and non_field_errors into a single entity.

Parameters

- **errors** (*dictionary*) – `Services`'s `errors` dictionary
- **non_field_errors** (*dictionary*) – `Service`'s `non_field_errors` dictionary

2.3.2 Fields module

class service_objects.fields.**DictField**(*, *required=True*, *widget=None*, *label=None*, *initial=None*, *help_text=''*, *error_messages=None*, *show_hidden_initial=False*, *validators=()*, *localize=False*, *disabled=False*, *label_suffix=None*)

A field for `Service` that accepts a dictionary:

class PDFGenerate(Service): context = DictField()

process(self): context = self.cleaned_data['context']

PDFGenerate.execute({ 'context': {'a': 1, 'b': 2}

})

clean(*value*)

Validate the given value and return its "cleaned" value as an appropriate Python object. Raise ValidationError for any errors.

class service_objects.fields.**ListField**(*, *required=True*, *widget=None*, *label=None*, *initial=None*, *help_text=''*, *error_messages=None*, *show_hidden_initial=False*, *validators=()*, *localize=False*, *disabled=False*, *label_suffix=None*)

A field for `Service` that accepts a list:

class EmailWelcomeMessage(Service): emails = ListField()

process(self): emails = self.cleaned_data['emails']

EmailWelcomeMessage.execute({ 'emails': ['blue@test.com', 'red@test.com']

})

clean(*value*)

Validate the given value and return its "cleaned" value as an appropriate Python object. Raise ValidationError for any errors.

class service_objects.fields.**ModelField**(*model_class*, *allow_unsaved=False*, **args*, ***kwargs*)

A field for `Service` that accepts an object of the specified `Model`:

```
class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    last_updated = models.DateTimeField()
```

(continues on next page)

(continued from previous page)

```
class UpdatePerson(Service):
    person = ModelField(Person)

    process(self):
        person = self.cleaned_data['person']
        person.last_updated = now()
        person.save()


user = Person(first_name='John', last_name='Smith')
user.save()

UpdatePerson.execute({
    'person': user
})
```

Parameters

- **model_class** – Django `Model` or dotted string of : class:*Model* name
- **allow_unsaved** – Whether the object is required to be saved to the database

clean(*value*)

Validate the given value and return its "cleaned" value as an appropriate Python object. Raise ValidationError for any errors.

class service_objects.fields.**MultipleFormField**(*form_class*, *min_count=1*, *max_count=None*, **args*, ***kwargs*)

A field for `Service` that accepts a list of objects which is translated into multiple `Form` objects:

```
class PersonForm(forms.Form):
    name = forms.CharField()


class UpdateOrganizationService(Service):
    people = MultipleFormField(PersonForm)

    def process(self):
        people = self.cleaned_data['people']
        for person in people:
            print(person.cleaned_data['name'])

UpdateOrganizationService.execute({
    'people': [
        { 'name': 'John Smith' },
        { 'name': 'Adam Davis' },
    ]
})
```

clean(*values*)

Validate the given value and return its "cleaned" value as an appropriate Python object. Raise ValidationError for any errors.

class service_objects.fields.**MultipleModelField**(*model_class*, *allow_unsaved=False*, **args*, ***kwargs*)

A multiple model version of *ModelField*, will check each passed in object to match the specified `Model`:

```
class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)


class AssociatePeople(Service):
    people = MultipleModelField(Person, allow_unsaved=True)


users = [
    Person(first_name='John', last_name='Smith'),
    Person(first_name='Jane', last_name='Smith')
]

AssociatePeople.execute({
    'people': users
})

for user in users:
    user.save()
```

Parameters

- **model_class** – Django `Model` or dotted string of : class:*Model* name
- **allow_unsaved** – Whether the object is required to be saved to the database

clean(*values*)
Validate the given value and return its "cleaned" value as an appropriate Python object. Raise ValidationError for any errors.

2.3.3 Services module

class service_objects.services.**Service**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*, *renderer=None*)
Based on Django's `Form`, designed to encapsulate Business Rules functionality. Input values are validated against the Service's defined fields before calling main functionality:

```
class UpdateUserEmail(Service):
    user = ModelField(User)
    new_email = forms.EmailField()

    def process(self):
        old_email = user.email
        user.email = self.cleaned_data['new_email']
        user.save()

        send_email(
            'Email Update',
            'Your email was changed',
            'system',
            [old_email]
```

(continues on next page)

(continued from previous page)

```
        )


user = User.objects.get(id=20)

UpdateUserEmail.execute({
    'user': user,
    'new_email': 'John.Smith@example.com'
})
```

Variables

- **db_transaction** (*boolean*) – controls if *execute()* is performed inside a Django database transaction. Default is True.
- **using** (*string*) – In a multiple database setup, controls which database connection is used from the transaction. Defaults to DEFAULT_DB_ALIAS which works in a single database setup.

classmethod execute(*inputs*, *files=None*, ***kwargs*)
Function to be called from the outside to kick off the Service functionality.

Parameters

- **inputs** (*dictionary*) – data parameters for Service, checked against the fields defined on the Service class.
- **files** (*dictionary*) – usually request's FILES dictionary or None.
- ****kwargs** (*dictionary*) – any additional parameters Service may need, can be an empty dictionary

post_process()
Post process method to be perform extra actions once *process()* successfully executes.

process()
Main method to be overridden; contains the Business Rules functionality.

service_clean()
Calls base Form's `is_valid()` to verify `inputs` against Service's fields and raises `InvalidInputsError` if necessary.

class service_objects.services.**ModelService**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*, *renderer=None*)
Same as *Service* but auto-creates fields based on the provided `Model`. Additionally, You can manually create fields to override or extend the auto-created fields:

```
class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    email = models.EmailField()
```

(continues on next page)

(continued from previous page)

```
class CreatePersonService(Service):
    class Meta:
        model = Person
        fields = '_all_'

    notify = forms.BooleanField()

    def process(self):
        person = Person(
            first_name = self.cleaned_data['first_name'],
            last_name = self.cleaned_data['last_name'],
            email = self.cleaned_data['email']
        )
        person.save()

        if self.cleaned_data['notify']:
            django.send_mail(
                'Account Created',
                'An account has been created for you'
                'System',
                [person.email]
            )


CreatePersonService.execute({
    'first_name': 'John',
    'last_name': 'Smith',
    'notify': True
})
```

2.3.4 Views module

class service_objects.views.**CreateServiceView**(***kwargs*)
Based on Django's `CreateView`, designed to call the `Service` class if the form is valid.

class service_objects.views.**ServiceView**(***kwargs*)
Based on Django's `FormView`, designed to call a `Service` class if the Form is valid. If `form_class` is `None`, ServiceView will use `service_class` for the Form to present the UI to the User:

```
from django.core.urlresolvers import reverse_lazy

from service_objects.views import ServiceView

from .forms import BookingForm
from .services import CreateBookingService


class CreateBookingView(ServiceView):
    form_class = BookingForm
    service_class = CreateBookingService
    template_name = 'booking/create_booking.html'
    success_url = reverse_lazy('booking:success')
```

class service_objects.views.**UpdateServiceView**(***kwargs*)
Based on Django's `UpdateView`, designed to call the `Service` class if the form is valid.

2.4 Support

2.4.1 Supported Version

Tests are run on officially supported versions of `Django`

- Python 2.7, 3.5, 3.6, 3.7
- Django 1.11, 2.0, 2.1, 2.2, 3.0

2.4.2 Contributing

We welcome new functionality and bugfixes! Everything should be submitted as a GitHub Pull Request. If you want to be sure that the change will be accepted before starting then open a GitHub Issue to have a discussion.

Pull Request Process

1. Ensure any install or build dependencies are removed before the end of the layer when doing a build.
2. Include/update relevant documenation changes.
3. Include/update relevants test changes.
4. Make sure all tests are passing.
5. Make sure code is passing `flake8`

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index