djangochannelsrestframework

Release 1.0.0

hishnash

Feb 23, 2022

CONTENTS:

CHAPTER

ONE

INTRODUCTION

1.1 Django Channels Rest Framework

Django Channels Rest Framework provides a DRF like interface for building channels-v3 websocket consumers.

This project can be used alongside HyperMediaChannels and ChannelsMultiplexer to create a Hyper Media Style api over websockets. However Django Channels Rest Framework is also a free standing framework with the goal of providing an api that is familiar to DRF users.

theY4Kman has developed a useful Javascript client library dcrf-client to use with DCRF.

1.2 Installation

```
pip install djangochannelsrestframework
```

1.3 Thanks to

DCRF is based of a fork of Channels Api and of course inspired by Django Rest Framework.

1.3.1 Introduction

1.3.1.1 Django Channels Rest Framework

Django Channels Rest Framework provides a DRF like interface for building channels-v3 websocket consumers.

This project can be used alongside HyperMediaChannels and ChannelsMultiplexer to create a Hyper Media Style api over websockets. However Django Channels Rest Framework is also a free standing framework with the goal of providing an api that is familiar to DRF users.

theY4Kman has developed a useful Javascript client library dcrf-client to use with DCRF.

1.3.1.2 Installation

```
pip install djangochannelsrestframework
```

1.3.1.3 Thanks to

DCRF is based of a fork of Channels Api and of course inspired by Django Rest Framework.

1.3.2 Examples

This is a collection of examples using the *djangochannelsrestframework* library.

1.3.2.1 Generic Api Consumer

In DCRF you can create a GenericAsyncAPIConsumer that works much like a GenericAPIView in DRF.

There are set of mixins for the consumer, that add different actions based on the CRUD operations.

- `ListModelMixin` this mixin adds the action `list`, allows to retrieve all instances of a model class.
- `RetrieveModelMixin` this mixin adds the action `retrieve` allows to retrieve an object based on the pk sent.
- `PatchModelMixin` this mixin adds the action `patch`, allows to patch an instance of a model.
- `UpdateModelMixin` this mixin adds the action `update`, allows to update a model instance.
- `CreateModelMixin` this mixin adds the action `create`, allows to create an instance based on the data sent.
- `DeleteModelMixin` this mixin adds the action `delete`, allows to delete an instance based on the pk sent.

Example

This example shows how to create a basic consumer for the django's auth user model. We are going to create a serializer class for it, and mixin with the `GenericAsyncAPIConsumer` the action mixins.

```
# serializers.py
from rest_framework import serializers
from django.contrib.auth.models import User


class UserSerializer(serializers.ModelSerializer):

    class Meta:
        model = User
        fields = ["id", "username", "email", "password"]
        extra_kwargs = {'password': {'write_only': True}}

    def create(self, validated_data):
        user = User(
            email=validated_data['email'],
            username=validated_data['username']
        )
        user.set_password(validated_data['password'])
```

(continues on next page)

(continued from previous page)

```
        user.save()
        return user
```

```
# consumers.py
from django.contrib.auth.models import User
from .serializers import UserSerializer
from djangochannelsrestframework.generics import GenericAsyncAPIConsumer
from djangochannelsrestframework.mixins import (
    ListModelMixin,
    RetrieveModelMixin,
    PatchModelMixin,
    UpdateModelMixin,
    CreateModelMixin,
    DeleteModelMixin,
)

class UserConsumer(
        ListModelMixin,
        RetrieveModelMixin,
        PatchModelMixin,
        UpdateModelMixin,
        CreateModelMixin,
        DeleteModelMixin,
        GenericAsyncAPIConsumer,
):

    queryset = User.objects.all()
    serializer_class = UserSerializer
```

```
# routing.py
from django.urls import re_path
from . import consumers

websocket_urlpatterns = [
    re_path(r"^ws/$", consumers.UserConsumer.as_asgi()),
]
```

How to use it

First we will create the web socket instance in `javascript`.

```
const ws = new WebSocket("ws://localhost:8000/ws/")

ws.onmessage = function(e){
    console.log(e)
}
```

Note: We must have a few users in our database for testing, if not, create them.

1. *List action.*

```
ws.send(JSON.stringify({
    action: "list",
    request_id: new Date().getTime()
}))
/* The return response will be something like this.
{
    "action": "list",
    "errors": [],
    "response_status": 200,
    "request_id": 1550050,
    "data": [
        {'email': '1@example.com', 'id': 1, 'username': 'test 1'},
        {'email': '2@example.com', 'id': 2, 'username': 'test 2'},
        {'email': '3@example.com', 'id': 3, 'username': 'test 3'},
    ]
}
*/
```

2. *Retrieve action.*

```
ws.send(JSON.stringify({
    action: "retrieve",
    request_id: new Date().getTime(),
    pk: 2
}))
/* The return response will be something like this.
{
    "action": "retrieve",
    "errors": [],
    "response_status": 200,
    "request_id": 1550050,
    "data": {'email': '2@example.com', 'id': 2, 'username': 'test 2'},
    }
*/
```

3. *Patch action.*

```
ws.send(JSON.stringify({
    action: "patch",
    request_id: new Date().getTime(),
    pk: 2,
    data: {
        email: "edited@example.com"
    }
}))
/* The return response will be something like this.
{
    "action": "patch",
    "errors": [],
    "response_status": 200,
    "request_id": 1550050,
    "data": {'email': 'edited@example.com', 'id': 2, 'username': 'test 2'},
```

(continues on next page)

(continued from previous page)

```
    }
*/
```

4. *Update action.*

```
ws.send(JSON.stringify({
    action: "update",
    request_id: new Date().getTime(),
    pk: 2,
    data: {
        username: "user 2"
    }
}))
/* The return response will be something like this.
{
    "action": "update",
    "errors": [],
    "response_status": 200,
    "request_id": 1550050,
    "data": {'email': 'edited@example.com', 'id': 2, 'username': 'user 2'},
    }
*/
```

5. *Create action.*

```
ws.send(JSON.stringify({
    action: "create",
    request_id: new Date().getTime(),
    data: {
        username: "new user 4",
        password: "testpassword123",
        email: "4@example.com"
    }
}))
/* The return response will be something like this.
{
    "action": "create",
    "errors": [],
    "response_status": 201,
    "request_id": 1550050,
    "data": {'email': '4@example.com', 'id': 4, 'username': 'new user 4'},
    }
*/
```

6. *Delete action.*

```
ws.send(JSON.stringify({
    action: "delete",
    request_id: new Date().getTime(),
    pk: 4
}))
/* The return response will be something like this.
{
```

(continues on next page)

(continued from previous page)

```
    "action": "delete",
    "errors": [],
    "response_status": 204,
    "request_id": 1550050,
    "data": null,
    }
*/
```

Full example

```
mysite/
    manage.py
    mysite/
        __init__.py
        asgi.py
        settings.py
        urls.py
        wsgi.py
    example/
        __init__.py
        consumers.py
        models.py
        serializers.py
        routing.py
        templates/
            example/
                index.html
        tests.py
        urls.py
        views.py
```

```
# serializers.py
from rest_framework import serializers
from django.contrib.auth.models import User


class UserSerializer(serializers.ModelSerializer):

    class Meta:
        model = User
        fields = ["id", "username", "email", "password"]
        extra_kwargs = {'password': {'write_only': True}}

    def create(self, validated_data):
        user = User(
            email=validated_data['email'],
            username=validated_data['username']
        )
        user.set_password(validated_data['password'])
        user.save()
```

(continues on next page)

(continued from previous page)

```
        return user
```

```
# consumers.py
from django.contrib.auth.models import User
from .serializers import UserSerializer
from djangochannelsrestframework.generics import GenericAsyncAPIConsumer
from djangochannelsrestframework.mixins import (
    ListModelMixin,
    RetrieveModelMixin,
    PatchModelMixin,
    UpdateModelMixin,
    CreateModelMixin,
    DeleteModelMixin,
)


class UserConsumer(
        ListModelMixin,
        RetrieveModelMixin,
        PatchModelMixin,
        UpdateModelMixin,
        CreateModelMixin,
        DeleteModelMixin,
        GenericAsyncAPIConsumer,
):

    queryset = User.objects.all()
    serializer_class = UserSerializer
```

```
# routing.py
from django.urls import re_path
from . import consumers

websocket_urlpatterns = [
    re_path(r"^ws/$", consumers.UserConsumer.as_asgi()),
]
```

```
from django.shortcuts import render, reverse


def index(request):
    return render(request, 'example/index.html')
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Generic Api Consumer</title>
</head>
<body>
```

(continues on next page)

(continued from previous page)

```
<button id="list">List</button>
<button id="retrieve">Retrieve</button>
<button id="create">Create</button>
<button id="patch">Patch</button>
<button id="update">Update</button>
<button id="delete">Delete</button>

<pre id="response"></pre>

<script>
    const ws = new WebSocket("ws://localhost:8000/ws/")

    ws.onmessage = function (e) {
        document.getElementById("response").textContent = JSON.stringify(JSON.parse(e.
↪data), undefined, 2);
        console.log(e.data)
    }

    document.querySelector('#list').onclick = function (e) {
        ws.send(JSON.stringify({
            action: "list",
            request_id: new Date().getTime()
        }))
    };

    document.querySelector('#retrieve').onclick = function (e) {
        ws.send(JSON.stringify({
            action: "retrieve",
            request_id: new Date().getTime(),
            pk: 2
        }))
    }

    document.querySelector('#create').onclick = function (e) {
        ws.send(JSON.stringify({
            action: "create",
            request_id: new Date().getTime(),
            data: {
                username: "newuser4",
                password: "testpassword123",
                email: "4@example.com"
            }
        }))
    }

    document.querySelector('#patch').onclick = function (e) {
        ws.send(JSON.stringify({
            action: "patch",
            request_id: new Date().getTime(),
            pk: 2,
            data: {
                email: "edited@example.com"
```

(continues on next page)

(continued from previous page)

```
            }
        }))
    }

    document.querySelector('#update').onclick = function (e) {
        ws.send(JSON.stringify({
            action: "update",
            request_id: new Date().getTime(),
            pk: 2,
            data: {
                username: "user 2"
            }
        }))
    }

    document.querySelector('#delete').onclick = function (e) {
        ws.send(JSON.stringify({
            action: "delete",
            request_id: new Date().getTime(),
            pk: 2
        }))
    }
</script>
</body>
</html>
```

1.3.2.2 Custom actions

Consumer that aren't bound to a Model.

We may want a consumer for handling certain actions that are not referred to any Django model. Maybe for fetching data from an external api service, using `requests` library or another async request lib.

```
# consumers.py
from djangochannelsrestframework.decorators import action
from djangochannelsrestframework.consumers import AsyncAPIConsumer
from rest_framework import status

class MyConsumer(AsyncAPIConsumer):

    @action()
    async def an_async_action(self, some=None, **kwargs):
        # do something async
        return {'response with': 'some message'}, status.HTTP_RESPONSE_OK

    @action()
    def a_sync_action(self, pk=None, **kwargs):
        # do something sync
        return {'response with': 'some message'}, status.HTTP_RESPONSE_OK
```

Consumer that is bound to a Model.

Inheriting from `GenericAsyncAPIConsumer` we have access to methods like `get_queryset` and `get_object`, this way we can perform operations in our django models though custom actions.

```
# serializers.py
from rest_framework import serializers
from django.contrib.auth.models import User

class UserSerializer(serializers.ModelSerializer):

    class Meta:
        model = User
        fields = ["id", "username", "email", "password"]
        extra_kwargs = {'password': {'write_only': True}}

    def create(self, validated_data):
        user = User(
            email=validated_data['email'],
            username=validated_data['username']
        )
        user.set_password(validated_data['password'])
        user.save()
        return user
```

```
# consumers.py
from django.contrib.auth.models import User
from .serializers import UserSerializer
from djangochannelsrestframework.generics import GenericAsyncAPIConsumer
from djangochannelsrestframework.decorators import action

class UserConsumer(GenericAsyncAPIConsumer):
    queryset = User.objects.all()
    serializer_class = UserSerializer

    @action()
    async def send_email(self, pk=None, to=None, **kwargs):
        user = await database_sync_to_async(self.get_object)(pk=pk)
        # ... do some stuff
        # remember to wrap all db actions in `database_sync_to_async`
        return {}, 200  # return the content and the response code.

    @action()  # if the method is not async it is already wrapped in `database_sync_to_
↪async`
    def publish(self, pk=None, **kwargs):
        user = self.get_object(pk=pk)
        # ...
        return {'pk': pk}, 200
```

1.3.2.3 Observer model instance

This mixin consumer lets you subscribe to all changes of a specific instance, and also gives you access to the `retrieve` action.

```
# serializers.py
from rest_framework import serializers
from django.contrib.auth.models import User
class UserSerializer(serializers.ModelSerializer):

    class Meta:
        model = User
        fields = ["id", "username", "email", "password"]
        extra_kwargs = {'password': {'write_only': True}}

    def create(self, validated_data):
        user = User(
            email=validated_data['email'],
            username=validated_data['username']
        )
        user.set_password(validated_data['password'])
        user.save()
        return user
```

```
# consumers.py
from django.contrib.auth.models import User
from .serializers import UserSerializer
from djangochannelsrestframework.generics import GenericAsyncAPIConsumer
from djangochannelsrestframework.observer.generics import ObserverModelInstanceMixin

class UserConsumer(ObserverModelInstanceMixin, GenericAsyncAPIConsumer):
    queryset = User.objects.all()
    serializer_class = UserSerializer
```

```
# routing.py
from django.urls import re_path
from . import consumers

websocket_urlpatterns = [
    re_path(r"^ws/$", consumers.UserConsumer.as_asgi()),
]
```

How to use it

First we will create the web socket instance in `javascript`.

```
const ws = new WebSocket("ws://localhost:8000/ws/")

ws.onmessage = function(e){
    console.log(e)
}
```

Note: We must have a few users in our database for testing, if not, create them.

Retrieve action.

```
ws.send(JSON.stringify({
    action: "retrieve",
    request_id: new Date().getTime(),
    pk: 1,
}))
/* The return response will be something like this.
{
    "action": "list",
    "errors": [],
    "response_status": 200,
    "request_id": 1550050,
    "data": {'email': '1@example.com', 'id': 1, 'username': 'test 1'},
}
*/
```

Subscription

1. Subscribe to a specific instance.

```
ws.send(JSON.stringify({
    action: "retrieve",
    request_id: new Date().getTime(),
    pk: 1,
}))
/* After subscribing the response will be something like this.
{
    "action": "subscribe_instance",
    "errors": [],
    "response_status": 201,
    "request_id": 1550050,
    "data": null,
}
*/
```

2. Changing the model instance in from the shell will fire the subscription event.

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.get(pk=1)
>>> user.username = "edited user name"
>>> user.save()
```

3. After saving the model instance, in the console, we will see the subscription message.

```
{
    action: "update",
```

(continues on next page)

(continued from previous page)

```
    errors: [],
    response_status: 200,
    request_id: 1550050,
    data: {email: '1@example.com', id: 1, username: 'edited user name'},
}
```

Todo

- More detail example.

1.3.2.4 View as consumer

Introduction

Suppose we already have a functional API that uses Django Rest Framework, and we want to add some websocket functionality. We can use the `view_as_consumer` decorator for accessing the same REST methods.

Creating the serializers.

```
# serializers.py
from django.contrib.auth.models import User
from rest_framework import serializers


class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ["id", "username", "email"]
```

Creating the view sets.

```
# views.py
from rest_framework.viewsets import ModelViewSet
from django.contrib.auth.models import User
from .serializers import UserSerializer


class UserViewSet(ModelViewSet):
    queryset = User.objects.all()
    serializer_class = UserSerializer
```

Routing the consumer

Using the same `UserViewSet` we can map some basic websocket actions to the REST methods. The mapped actions are:

- `create` - PUT
- `update` - PATCH
- `list` - GET
- `retrieve` - GET

```
# routing.py
from django.urls import re_path
from djangochannelsrestframework.consumers import view_as_consumer
from .views import UserViewSet

websocket_urlpatterns = [
    re_path(r"^user/$", view_as_consumer(UserViewSet.as_view()))
]
```

Manual testing the output.

Now we will have a websocket client in javascript listening to the messages, after subscribing to the comment activity. This code block can be used in the browser console.

Note:

In production the `ws:` is `wss:`, we can check it with the following code:

```
const ws_schema = window.location.protocol === "http:" ? "ws:" : "wss:";
```

```
const ws = new WebSocket("ws://localhost:8000/user/")
const ws.onopen = function(){
    ws.send(JSON.stringify({
        action: "list",
        request_id: new Date().getTime(),
    }))
}
const ws.onmessage = function(e){
    console.log(e)
}
```

> **Warning:** At this point we should have some users in our database, otherwise create them

In the console we will have the following response assuming that we have some users in our database.

```
{
    error: [],
    data: [
```

(continues on next page)

(continued from previous page)

```
        {username: "user 1", id: 1, email: "1@example.com"},
        {username: "user 2", id: 2, email: "2@example.com"},
    ],
    action: "list",
    response_status: 200,
    request_id: 15050500
}
```

1.3.2.5 Model observer

Subscribing to all instances of a model.

Introduction

In this first example, we will create a `user` model with a `comment` related model, create the serializers for each one. And create a `consumer` for the `user` model, with a model observer method for **all comment instances**.

Creating models.

We will have the following `models.py` file, with a user model, and a comment models that is related to the user.

```python
# models.py
from django.db import models
from django.contrib.auth.models import AbstractUser

class User(AbstractUser):
    pass

class Comment(models.Model):
    text = models.TextField()
    user = models.ForeignKey(User, related_name="comments", on_delete=models.CASCADE)
    date = models.DatetimeField(auto_now_add=True)
```

Creating the serializers.

In the `serializers.py` file, we will have the serializers for the models in the `models.py` file.

```python
# serializers.py
from rest_framework import serializers
from .models import User, Comment

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ["id", "username", "email"]

class CommentSerializer(serializers.ModelSerializer):
    class Meta:
```

(continues on next page)

(continued from previous page)

```
        model = Comment
        fields = ["id", "text", "user"]
```

Creating the consumers.

Now in the `consumers.py` file, we will create or websocket consumer for the users, with a model observer method for **all instances** of the `Comment` model.

These are the important methods of the class.

- A method, called `comment_activity` decorated with the `model_observer` decorator and as argument we will add the `Comment` model.
- A `subscribe_to_comment_activity` `action` to subscribe the `model_observer` method.
- A method (it can be named the same as the `model_observer` method) decorated with the `@comment_activity.serializer`, this will return the serializer based on the instance.

```
# consumers.py

from djangochannelsrestframework.consumers import GenericAsyncAPIConsumer
from djangochannelsrestframework.observer import model_observer
from djangochannelsrestframework.decorators import action

from .serializers import UserSerializer, CommentSerializer
from .models import User, Comment


class MyConsumer(GenericAsyncAPIConsumer):
    queryset = User.objects.all()
    serializer_class = UserSerializer

    @model_observer(Comments)
    async def comment_activity(
        self,
        message: CommentSerializer,
        observer=None,
        subscribing_request_ids=[]
        **kwargs
    ):
        await self.send_json(message.data)

    @comment_activity.serializer
    def comment_activity(self, instance: Comment, action, **kwargs) -> CommentSerializer:
        '''This will return the comment serializer'''
        return CommentSerializer(instance)

    @action()
    async def subscribe_to_comment_activity(self, request_id, **kwargs):
        await self.comment_activity.subscribe(request_id=request_id)
```

Manual testing the output.

Now we will have a websocket client in javascript listening to the messages, after subscribing to the comment activity. This code block can be used in the browser console.

Note:

In production the `ws:` is `wss:`, we can check it with the following code:

```
const ws_schema = window.location.protocol === "http:" ? "ws:" : "wss:";
```

```
const ws = new WebSocket("ws://localhost:8000/ws/my-consumer/")
const ws.onopen = function(){
    ws.send(JSON.stringify({
        action: "subscribe_to_comment_activity",
        request_id: new Date().getTime(),
    }))
}
const ws.onmessage = function(e){
    console.log(e)
}
```

In the IPython shell we will create some comments for different users and in the browser console we will see the log.

> **Warning:** At this point we should have some users in our database, otherwise create them

We will create a comment using the `user_1` and we will see the log in the browser console.

```
>>> from my_app.models import User, Comment
>>> user_1 = User.objects.get(pk=1)
>>> user_2 = User.objects.get(pk=2)
>>> Comment.objects.create(text="user 1 creates a new comment", user=user_1)
```

In the console log we will see something like this:

```
{
    action: "subscribe_to_comment_activity",
    errors: [],
    response_status: 200,
    request_id: 15606042,
    data: {
        id: 1,
        text: "user 1 creates a new comment",
        user: 1
    }
}
```

Now we will create a comment with the `user 2`.

```
>>> Comment.objects.create(text="user 2 creates a second comment", user=user_2)
```

In the console log we will see something like this:

```
{
    action: "subscribe_to_comment_activity",
    errors: [],
    response_status: 200,
    request_id: 15606042,
    data: {
        id: 2,
        text: "user 2 creates a second comment",
        user: 2,
    },
}
```

Conclusions

In this example we subscribed to **all instances** of the comment model, in the next section we will see how to filter them.

1.3.2.6 Filtered model observer

Subscribing to a filtered list of models.

Introduction

In this first example, we will create a `user` model with a `comment` related model, create the serializers for each one. And create a `consumer` for the `user` model, with a model observer method for watching all changes of the current user.

Creating models.

We will have the following `models.py` file, with a user model, and a comment models that is related to the user.

```python
# models.py
from django.db import models
from django.contrib.auth.models import AbstractUser

class User(AbstractUser):
    pass

class Comment(models.Model):
    text = models.TextField()
    user = models.ForeignKey(User, related_name="comments", on_delete=models.CASCADE)
    date = models.DatetimeField(auto_now_add=True)
```

Creating the serializers.

In the `serializers.py` file, we will have the serializers for the models in the `models.py` file.

```
# serializers.py
from rest_framework import serializers
from .models import User, Comment

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ["id", "username", "email"]

class CommentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Comment
        fields = ["id", "text", "user"]
```

Creating the consumers.

Now in the `consumers.py` file, we will create or websocket consumer for the users, with a model observer method for the `Comment` model, filtered for the current user.

These are the important methods of the class.

- A method, called `comment_activity` decorated with the `model_observer` decorator and as argument we will add the `Comment` model.
- A `subscribe_to_comment_activity` `action` to subscribe the `model_observer` method.
- A method (it can be named the same as the `model_observer` method) decorated with the `@comment_activity.serializer`, this will return the serializer based on the instance.

Warning: The user must be logged to subscribe this method, because we will access the `self.scope["user"]`

```
# consumers.py

from djangochannelsrestframework.consumers import GenericAsyncAPIConsumer
from djangochannelsrestframework.observer import model_observer
from djangochannelsrestframework.decorators import action

from .serializers import UserSerializer, CommentSerializer
from .models import User, Comment


class MyConsumer(GenericAsyncAPIConsumer):
    queryset = User.objects.all()
    serializer_class = UserSerializer

    @model_observer(Comments)
    async def comment_activity(
        self,
```

(continues on next page)

(continued from previous page)

```
        message: CommentSerializer,
        observer=None,
        subscribing_request_ids=[],
        **kwargs
    ):
        await self.send_json(message.data)

    @comment_activity.serializer
    def comment_activity(self, instance: Comment, action, **kwargs) -> CommentSerializer:
        '''This will return the comment serializer'''
        return CommentSerializer(instance)

    @comment_activity.groups_for_signal
    def comment_activity(self, instance: Comment, **kwargs):
        # this block of code is called very often *DO NOT make DB QUERIES HERE*
        yield f'-user__{instance.user_id}'  #! the string **user** is the ``Comment's``␣
↪user field.

    @comment_activity.groups_for_consumer
    def comment_activity(self, school=None, classroom=None, **kwargs):
        # This is called when you subscribe/unsubscribe
        yield f'-user__{self.scope["user"].pk}'

    @action()
    async def subscribe_to_comment_activity(self, request_id, **kwargs):
        # We will check if the user is authenticated for subscribing.
        if "user" in self.scope and self.scope["user"].is_authenticated:
            await self.comment_activity.subscribe(request_id=request_id)
```

Note: Without logging in we will have to access the `user` using the pk or any other unique field. Example:

```
...
class MyConsumer(GenericAsyncAPIConsumer):
    ...

    @action()
    async def subscribe_to_comment_activity(self, user_pk, **kwargs):
        # We will check if the user is authenticated for subscribing.
        user = await database_sync_to_async(User.objects.get)(pk=user_pk)
        await self.comment_activity.subscribe(user=user)
```

Manual testing the output.

Now we will have a websocket client in javascript listening to the messages, after subscribing to the comment activity. This code block can be used in the browser console.

Note:

In production the `ws:` is `wss:`, we can check it with the following code:

```
const ws_schema = window.location.protocol === "http:" ? "ws:" : "wss:";
```

```
const ws = new WebSocket("ws://localhost:8000/ws/my-consumer/")
const ws.onopen = function(){
    ws.send(JSON.stringify({
        action: "subscribe_to_comment_activity",
        request_id: new Date().getTime(),
    }))
}
const ws.onmessage = function(e){
    console.log(e)
}
```

Note:

The subscribe method doesn't require being logged:

```
const ws = new WebSocket("ws://localhost:8000/ws/my-consumer/")
const ws.onopen = function(){
    ws.send(JSON.stringify({
        action: "subscribe_to_comment_activity",
        request_id: new Date().getTime(),
        user_pk: 1, // This field is the argument in the
                    // subscribe method, and the pk correspond to the user.
    }))
}
const ws.onmessage = function(e){
    console.log(e)
}
```

In the IPython shell we will create some comments for different users and in the browser console we will see the log.

> **Warning:** At this point we should have some users in our database, otherwise create them

We will create a comment using the `user_1` and we will see the log in the browser console.

```
>>> from my_app.models import User, Comment
>>> user_1 = User.objects.get(pk=1)
>>> user_2 = User.objects.get(pk=2)
>>> Comment.objects.create(text="user 1 creates a new comment", user=user_1)
```

In the console log we will se something like this:

```
{
    action: "subscribe_to_comment_activity",
    errors: [],
    response_status: 200,
    request_id: 15606042,
    data: {
        id: 1,
        text: "user 1 creates a new comment",
        user: 1
    }
}
```

Now we will create a comment with the `user 2`.

```
>>> Comment.objects.create(text="user 2 creates a second comment", user=user_2)
```

In the console log we will see **nothing**, because this comment was created by the `user_2`.

Conclusions

In this example we subscribe to the filtered instances of the comment model.

1.3.3 Consumers

@action(*atomic=None*, ***kwargs*)

Mark a method as an action.

Note: Should be used as a method decorator eg: *@action()*

It can be used on both *async* and *sync* methods.

```
from djangochannelsrestframework.decorators import action

class MyConsumer(AsyncAPIConsumer):
    queryset = User.objects.all()
    serializer_class = UserSerializer

    @action()
    async def delete_user(self, request_id, user_pk, **kwargs):
        ...
```

Methods decorated with *@action()* will be called when a json message arrives from the client with a matching *action* name.

The default way of sending a message to call an action is:

```
{
 action: "delete_user",
 request_id: 42,
```

(continues on next page)

(continued from previous page)

```
 user_pk: 82
}
```

You can alter how `AsyncAPIConsumer` matches the action using the `get_action_name()` method.

When using on *sync* methods you can provide an additional option *atomic=True* to forcefully wrap the method in a transaction. The default value for atomic is determined by django's default db *ATOMIC_REQUESTS* setting.

Parameters **atomic** (*Optional[bool]*) –

class AsyncAPIConsumer(**args*, ***kwargs*)

This provides an async API consumer that is very inspired by DjangoRestFrameworks ViewSets.

permission_classes **An array for Permission classes**

async add_group(*name*)

Add a group to the set of groups this consumer is subscribed to.

Parameters **name** (*str*) –

async check_permissions(*action*, ***kwargs*)

Check if the action should be permitted. Raises an appropriate exception if the request is not permitted.

Parameters **action** (*str*) –

async get_action_name(*content*, ***kwargs*)

Retrieves the action name from the json message.

Returns a tuple of the action name and the argumetns that is passed to the action.

Override this method if you do not want to use *{"action": "action_name"}* as the way to describe actions.

Parameters **content** (*Dict*) –

Return type *Tuple*[*Optional*[str], *Dict*]

async get_permissions(*action*, ***kwargs*)

Instantiates and returns the list of permissions that this view requires.

Parameters **action** (*str*) –

async handle_action(*action*, *request_id*, ***kwargs*)

Handle a call for a given action.

This method checks permissions and handles exceptions sending them back over the ws connection to the client.

If there is no action listed on the consumer for this action name a *MethodNotAllowed* error is sent back over the ws connection.

Parameters

- **action** (*str*) –
- **request_id** (*str*) –

async handle_exception(*exc*, *action*, *request_id*)

Handle any exception that occurs, by sending an appropriate message

Parameters

- **exc** (*Exception*) –
- **action** (*str*) –

async receive_json(*content*, ***kwargs*)
Called with decoded JSON content.

Parameters **content** (*Dict*) –

async remove_group(*name*)
Remove a group to the set of groups this consumer is subscribed to.

Parameters **name** (*str*) –

async reply(*action*, *data=None*, *errors=None*, *status=200*, *request_id=None*)
Send a json response back to the client.

You should aim to include the *request_id* if possible as this helps clients link messages they have sent to responses.

Parameters **action** (*str*) –

class GenericAsyncAPIConsumer(**args*, ***kwargs*)
Base class for all other generic views, this subclasses AsyncAPIConsumer.

queryset
will be accesed when the method *get_queryset* is called.

serializer_class
it should correspond with the *queryset* model, it will be useded for the return response.

lookup_field
field used in the *get_object* method. Optional.

lookup_url_kwarg
url parameter used it for the lookup.

filter_queryset(*queryset*, ***kwargs*)
Given a queryset, filter it with whichever filter backend is in use.

You are unlikely to want to override this method, although you may need to call it either from a list view, or from a custom *get_object* method if you want to apply the configured filtering backend to the default queryset.

Parameters

- **queryset** (*django.db.models.query.QuerySet*) – cached queryset to filter.
- **kwargs** – keyworded dictionary arguments.

Returns Filtered queryset.

Return type django.db.models.query.QuerySet

Todos: Implement

get_object(***kwargs*)
Returns the object the view is displaying.

You may want to override this if you need to provide non-standard queryset lookups. Eg if objects are referenced using multiple keyword arguments in the url conf.

Parameters **kwargs** – keyworded dictionary, it can be use it for filtering the queryset.

Returns Model object class.

Return type django.db.models.base.Model

Examples

```
>>> filtered_queryset = self.get_object(**{"field__gte": value})  # this way
↪you could filter from the frontend.
```

get_queryset(***kwargs*)
Get the list of items for this view. This must be an iterable, and may be a queryset. Defaults to using *self.queryset*.

This method should always be used rather than accessing *self.queryset* directly, as *self.queryset* gets evaluated only once, and those results are cached for all subsequent requests.

You may want to override this if you need to provide different querysets depending on the incoming request.

(Eg. return a list of items that is specific to the user)

Parameters **kwargs** – keyworded dictionary.

Returns Queryset attribute.

Return type django.db.models.query.QuerySet

get_serializer(*action_kwargs=None*, **args*, ***kwargs*)
Return the serializer instance that should be used for validating and deserializing input, and for serializing output.

Parameters

- **action_kwargs** (*Optional[Dict]*) – keyworded dictionary from the action.
- **args** – listed arguments.
- **kwargs** – keyworded dictionary arguments.

Returns Model serializer.

Return type rest_framework.serializers.Serializer

get_serializer_class(***kwargs*)
Return the class to use for the serializer. Defaults to using *self.serializer_class*.

You may want to override this if you need to provide different serializations depending on the incoming request.

(Eg. admins get full serialization, others get basic serialization)

Parameters **kwargs** – keyworded dictionary arguments.

Returns Model serializer class.

Return type *Type*[rest_framework.serializers.Serializer]

get_serializer_context(***kwargs*)
Extra context provided to the serializer class.

Parameters **kwargs** – keyworded dictionary arguments.

Returns Context dictionary, containing the scope and the consumer instance.

Return type *Dict*[str, *Any*]

view_as_consumer(*wrapped_view*, *mapped_actions=None*)
Wrap a django View to be used over a json ws connection.

```
websocket_urlpatterns = [
    re_path(r"^user/$", view_as_consumer(UserViewSet.as_view()))
]
```

This exposes the django view to your websocket connection so that you can send messages:

```
{
 action: "retrieve",
 request_id: 42,
 query: {pk: 92}
}
```

The default mapping for actions is:

- `create` - `PUT`
- `update` - `PATCH`
- `list` - `GET`
- `retrieve` - `GET`

Providing a *query* dict in the websocket messages results in the values of this dict being writen to the *GET* property of the request within your django view.

Providing a *parameters* dict within the websocket messages results in these values being passed as kwargs to the view method (in the same way that url parameters would normally be extracted).

Parameters

- **wrapped_view** (*Callable[[django.http.request.HttpRequest], django.http.response.HttpResponse]*) –
- **mapped_actions** (*Optional[Dict[str, str]]*) –

Return type djangochannelsrestframework.consumers.DjangoViewAsConsumer

1.3.4 Mixins

class CreateModelMixin

Create model mixin.

async create(*data*, ***kwargs*)

Create action.

Parameters **data** (*dict*) – model data to create.

Returns Tuple with the serializer data and the status code.

Return type *Tuple*[rest_framework.utils.serializer_helpers.ReturnDict, int]

Examples

```
#! consumers.py
from .models import User
from .serializers import UserSerializer
from djangochannelsrestframework import permissions
from djangochannelsrestframework.generics import GenericAsyncAPIConsumer
from djangochannelsrestframework.mixins import CreateModelMixin

class LiveConsumer(CreateModelMixin, GenericAsyncAPIConsumer):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    permission_classes = (permissions.AllowAny,)
```

```
#! routing.py
from django.urls import re_path
from .consumers import LiveConsumer

websocket_urlpatterns = [
    re_path(r'^ws/$', LiveConsumer.as_asgi()),
]
```

```
// html
const ws = new WebSocket("ws://localhost:8000/ws/")
ws.send(JSON.stringify({
    action: "create",
    request_id: new Date().getTime(),
    data: {
        username: "test",
        password1: "testpassword123",
        password2: "testpassword123",
    }
}))
/* The response will be something like this.
{
    "action": "create",
    "errors": [],
    "response_status": 201,
    "request_id": 150060530,
    "data": {'username': 'test', 'id': 42,},
}
*/
```

class DeleteModelMixin

Delete model mixin

async delete(***kwargs*)

Retrieve action.

Returns Tuple with the serializer data and the status code.

Return type *Tuple*[None, int]

Examples

```
#! consumers.py
from .models import User
from .serializers import UserSerializer
from djangochannelsrestframework import permissions
from djangochannelsrestframework.generics import GenericAsyncAPIConsumer
from djangochannelsrestframework.mixins import DeleteModelMixin

class LiveConsumer(DeleteModelMixin, GenericAsyncAPIConsumer):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    permission_classes = (permissions.AllowAny,)
```

```
#! routing.py
from django.urls import re_path
from .consumers import LiveConsumer

websocket_urlpatterns = [
    re_path(r'^ws/$', LiveConsumer.as_asgi()),
]
```

```
// html
const ws = new WebSocket("ws://localhost:8000/ws/")
ws.send(JSON.stringify({
    action: "delete",
    request_id: new Date().getTime(),
    pk: 1,
}))
/* The response will be something like this.
{
    "action": "delete",
    "errors": [],
    "response_status": 204,
    "request_id": 150000,
    "data": null,
}
*/
```

class ListModelMixin

List model mixin

async list(***kwargs*)

List action.

Returns Tuple with the list of serializer data and the status code.

Return type *Tuple*[rest_framework.utils.serializer_helpers.ReturnList, int]

Examples

```python
#! consumers.py
from .models import User
from .serializers import UserSerializer
from djangochannelsrestframework import permissions
from djangochannelsrestframework.generics import GenericAsyncAPIConsumer
from djangochannelsrestframework.mixins import ListModelMixin

class LiveConsumer(ListModelMixin, GenericAsyncAPIConsumer):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    permission_classes = (permissions.AllowAny,)
```

```python
#! routing.py
from django.urls import re_path
from .consumers import LiveConsumer

websocket_urlpatterns = [
    re_path(r'^ws/$', LiveConsumer.as_asgi()),
]
```

```javascript
// html
const ws = new WebSocket("ws://localhost:8000/ws/")
ws.send(JSON.stringify({
    action: "list",
    request_id: new Date().getTime(),
}))
/* The response will be something like this.
{
    "action": "list",
    "errors": [],
    "response_status": 200,
    "request_id": 1500000,
    "data": [
        {"email": "42@example.com", "id": 1, "username": "test1"},
        {"email": "45@example.com", "id": 2, "username": "test2"},
    ],
}
*/
```

class PaginatedModelListMixin

async list(***kwargs*)

List action.

Returns Tuple with the list of serializer data and the status code.

Examples

```
#! consumers.py
from .models import User
from .serializers import UserSerializer
from djangochannelsrestframework import permissions
from djangochannelsrestframework.generics import GenericAsyncAPIConsumer
from djangochannelsrestframework.mixins import ListModelMixin

class LiveConsumer(ListModelMixin, GenericAsyncAPIConsumer):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    permission_classes = (permissions.AllowAny,)
```

```
#! routing.py
from django.urls import re_path
from .consumers import LiveConsumer

websocket_urlpatterns = [
    re_path(r'^ws/$', LiveConsumer.as_asgi()),
]
```

```
// html
const ws = new WebSocket("ws://localhost:8000/ws/")
ws.send(JSON.stringify({
    action: "list",
    request_id: new Date().getTime(),
}))
/* The response will be something like this.
{
    "action": "list",
    "errors": [],
    "response_status": 200,
    "request_id": 1500000,
    "data": [
        {"email": "42@example.com", "id": 1, "username": "test1"},
        {"email": "45@example.com", "id": 2, "username": "test2"},
    ],
}
*/
```

property paginator: Optional[any]
Gets the paginator class

Returns Pagination class. Optional.

class PatchModelMixin
Patch model mixin

async patch(*data*, ***kwargs*)
Patch action.

Returns Tuple with the serializer data and the status code.

Parameters **data** (*dict*) –

Return type *Tuple*[rest_framework.utils.serializer_helpers.ReturnDict, int]

Examples

```
#! consumers.py
from .models import User
from .serializers import UserSerializer
from djangochannelsrestframework import permissions
from djangochannelsrestframework.generics import GenericAsyncAPIConsumer
from djangochannelsrestframework.mixins import PatchModelMixin

class LiveConsumer(PatchModelMixin, GenericAsyncAPIConsumer):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    permission_classes = (permissions.AllowAny,)
```

```
#! routing.py
from django.urls import re_path
from .consumers import LiveConsumer

websocket_urlpatterns = [
    re_path(r'^ws/$', LiveConsumer.as_asgi()),
]
```

```
// html
const ws = new WebSocket("ws://localhost:8000/ws/")
ws.send(JSON.stringify({
    action: "patch",
    request_id: new Date().getTime(),
    pk: 1,
    data: {
        email: "00@example.com",
    },
}))
/* The response will be something like this.
{
    "action": "patch",
    "errors": [],
    "response_status": 200,
    "request_id": 150000,
    "data": {"email": "00@example.com", "id": 1, "username": "test1"},
}
*/
```

class RetrieveModelMixin

Retrieve model mixin

async retrieve(***kwargs*)

Retrieve action.

Returns Tuple with the serializer data and the status code.

Return type *Tuple*[rest_framework.utils.serializer_helpers.ReturnDict, int]

Examples

```
#! consumers.py
from .models import User
from .serializers import UserSerializer
from djangochannelsrestframework import permissions
from djangochannelsrestframework.generics import GenericAsyncAPIConsumer
from djangochannelsrestframework.mixins import RetrieveModelMixin

class LiveConsumer(RetrieveModelMixin, GenericAsyncAPIConsumer):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    permission_classes = (permissions.AllowAny,)
```

```
#! routing.py
from django.urls import re_path
from .consumers import LiveConsumer

websocket_urlpatterns = [
    re_path(r'^ws/$', LiveConsumer.as_asgi()),
]
```

```
// html
const ws = new WebSocket("ws://localhost:8000/ws/")
ws.send(JSON.stringify({
    action: "retrieve",
    request_id: new Date().getTime(),
    pk: 1,
}))
/* The response will be something like this.
{
    "action": "retrieve",
    "errors": [],
    "response_status": 200,
    "request_id": 1500000,
    "data": {"email": "42@example.com", "id": 1, "username": "test1"},
}
*/
```

class StreamedPaginatedListMixin

async list(*action*, *request_id*, ***kwargs*)

List action.

Returns Tuple with the list of serializer data and the status code.

Examples

```python
#! consumers.py
from .models import User
from .serializers import UserSerializer
from djangochannelsrestframework import permissions
from djangochannelsrestframework.generics import GenericAsyncAPIConsumer
from djangochannelsrestframework.mixins import ListModelMixin

class LiveConsumer(ListModelMixin, GenericAsyncAPIConsumer):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    permission_classes = (permissions.AllowAny,)
```

```python
#! routing.py
from django.urls import re_path
from .consumers import LiveConsumer

websocket_urlpatterns = [
    re_path(r'^ws/$', LiveConsumer.as_asgi()),
]
```

```javascript
// html
const ws = new WebSocket("ws://localhost:8000/ws/")
ws.send(JSON.stringify({
    action: "list",
    request_id: new Date().getTime(),
}))
/* The response will be something like this.
{
    "action": "list",
    "errors": [],
    "response_status": 200,
    "request_id": 1500000,
    "data": [
        {"email": "42@example.com", "id": 1, "username": "test1"},
        {"email": "45@example.com", "id": 2, "username": "test2"},
    ],
}
*/
```

class UpdateModelMixin

Update model mixin

async update(*data*, ***kwargs*)

Retrieve action.

Returns Tuple with the serializer data and the status code.

Parameters **data** (*dict*) –

Return type *Tuple*[rest_framework.utils.serializer_helpers.ReturnDict, int]

Examples

```
#! consumers.py
from .models import User
from .serializers import UserSerializer
from djangochannelsrestframework import permissions
from djangochannelsrestframework.generics import GenericAsyncAPIConsumer
from djangochannelsrestframework.mixins import UpdateModelMixin

class LiveConsumer(UpdateModelMixin, GenericAsyncAPIConsumer):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    permission_classes = (permissions.AllowAny,)
```

```
#! routing.py
from django.urls import re_path
from .consumers import LiveConsumer

websocket_urlpatterns = [
    re_path(r'^ws/$', LiveConsumer.as_asgi()),
]
```

```
// html
const ws = new WebSocket("ws://localhost:8000/ws/")
ws.send(JSON.stringify({
    action: "update",
    request_id: new Date().getTime(),
    pk: 1,
    data: {
        username: "test edited",
    },
}))
/* The response will be something like this.
{
    "action": "update",
    "errors": [],
    "response_status": 200,
    "request_id": 1500000,
    "data": {"email": "42@example.com", "id": 1, "username": "test edited"},
}
*/
```

1.3.5 Observer

@observer(*signal*, ***kwargs*)

Note: Should be used as a method decorator eg: *@observer(user_logged_in)*

The wrapped method will be called once for each consumer that has subscribed.

```
class AdminPortalLoginConsumer(AsyncAPIConsumer):
     async def accept(self, **kwargs):
         await self.handle_user_logged_in.subscribe()
         await super().accept()

     @observer(user_logged_in)
     async def handle_user_logged_in(self, message, observer=None, **kwargs):
         await self.send_json(message)
```

If the signal you are using supports filtering with *args* or *kwargs* these can be passed to the *@observer(signal, args..)*.

Parameters **signal** (*django.dispatch.dispatcher.Signal*) –

@model_observer(*model*, ***kwargs*)

Note: Should be used as a method decorator eg: *@model_observer(BlogPost)*

The resulted wrapped method body becomes the handler that is called on each subscribed consumer. The method itself is replaced with an instance of `djangochannelsrestframework.observer.model_observer.ModelObserver`

```
# consumers.py

from djangochannelsrestframework.consumers import GenericAsyncAPIConsumer
from djangochannelsrestframework.observer import model_observer
from djangochannelsrestframework.decorators import action

from .serializers import UserSerializer, CommentSerializer
from .models import User, Comment

class MyConsumer(GenericAsyncAPIConsumer):
    queryset = User.objects.all()
    serializer_class = UserSerializer

    @model_observer(Comments)
    async def comment_activity(self, message, observer=None, subscribing_request_
↪ids=[], **kwargs):
        for request_id in subscribing_request_ids:
            await self.send_json({"message": message, "request_id": request_id})

    @comment_activity.serializer
    def comment_activity(self, instance: Comment, action, **kwargs):
        return CommentSerializer(instance).data

    @action()
    async def subscribe_to_comment_activity(self, request_id, **kwargs):
        await self.comment_activity.subscribe(request_id=request_id)
```

If you only need to use a regulare Django Rest Framework Serializer class then there is a shorthand:

```
class MyConsumer(GenericAsyncAPIConsumer):
    queryset = User.objects.all()
```

(continues on next page)

(continued from previous page)

```
    serializer_class = UserSerializer

    @model_observer(Comments, serializer_class=CommentSerializer)
    async def comment_activity(self, message, action, subscribing_request_ids=[],␣
↪**kwargs):
        for request_id in subscribing_request_ids:
            await self.reply(data=message, action=action, request_id=request_id)

    @action()
    async def subscribe_to_comment_activity(self, request_id, **kwargs):
        await self.comment_activity.subscribe(request_id=request_id)
```

Parameters **model** (*Type[django.db.models.base.Model]*) –

class BaseObserver(*func*, *partition='*'*)

This is the Base Observer class that *Observer* and *ModelObserver* inherit from.

The decorators *@model_observer* and *@observer* replaced the wrapped method with an instance of these classes. You can then access the methods of this class using the method name that you wrapped.

Parameters **partition** (*str*) –

groups_for_consumer(*func*)

Note: Should be used as a method decorator eg: *@observed_handler.groups_for_consumer*

The decorated method is used when *subscribe()* and *unsubscribe()* are called to enumerate the corresponding groups to un/subscribe to.

The *args* and *kwargs* providing to *subscribe()* and *unsubscribe()* are passed here to enable this.

```
@classroom_change_handler.groups_for_consumer
def classroom_change_handler(self, school=None, classroom=None, **kwargs):
    # This is called when you subscribe/unsubscribe
    if school is not None:
        yield f'-school__{school.pk}'
    if classroom is not None:
        yield f'-pk__{classroom.pk}'

@action()
async def subscribe_to_classrooms_in_school(self, school_pk, request_id,␣
↪**kwargs):
    # check user has permission to do this
    await self.classroom_change_handler.subscribe(school=school, request_
↪id=request_id)

@action()
async def subscribe_to_classroom(self, classroom_pk, request_id, **kwargs):
    # check user has permission to do this
    await self.classroom_change_handler.subscribe(classroom=classroom, request_
↪id=request_id)
```

It is important that a corresponding *groups_for_signal()* method is provided that enumerates the groups that each event is sent to.

Parameters **func** (*Callable[[djangochannelsrestframework.observer.base_observer.BaseObserver, djangochannelsrestframework.consumers.AsyncAPIConsumer], Generator[str, None, None]]*) –

groups_for_signal(*func*)

Note: Should be used as a method decorator eg: *@observed_handler.groups_for_signal*

The decorated method is used whenever an event happens that the observer is observing (even if nothing is subscribed).

The role of this method is to enumerate the groups that the event should be sent over.

```
@classroom_change_handler.groups_for_signal
def classroom_change_handler(self, instance: models.Classroom, **kwargs):
    yield f'-school__{instance.school_id}'
    yield f'-pk__{instance.pk}'
```

It is important that a corresponding *groups_for_consumer()* method is provided to enable the consumers to correctly select which groups to subscribe to.

Parameters **func** (*Callable[[...], Generator[str, None, None]]*) –

serializer(*func*)

Note: Should be used as a method decorator eg: *@observed_handler.serializer*

The method that this wraps is evaluated just after the observer is triggered before the result is sent over the channel layer. That means you **DO NOT** have access to user or other request information.

The result of this method is what is sent over the channel layer. If you need to modify that with user specific information then you need to do that in the observer handler method.

```
class MyConsumer(GenericAsyncAPIConsumer):
    queryset = User.objects.all()
    serializer_class = UserSerializer

    @model_observer(Comments)
    async def comment_activity(self, message, observer=None, subscribing_
↪request_ids=[], **kwargs):
        ...

    @comment_activity.serializer
    def comment_activity(self, instance: Comment, action, **kwargs):
        return CommentSerializer(instance).data
```

The advantage of doing serialization at this point is that it happens only once even if 1000s of consumers are subscribed to the event.

async subscribe(*consumer*, **args*, *request_id=None*, ***kwargs*)
This should be called to subscribe the current consumer.

args and kwargs passed here are provided to the *groups_for_consumer()* method to enable custom partitioning of events.

If the request_id is passed to the subscribe method then the observer will track that request id and provide it to the handling method when an event happens.

```
class MyConsumer(GenericAsyncAPIConsumer):
    queryset = User.objects.all()
    serializer_class = UserSerializer

    @model_observer(Comments)
    async def comment_activity(self, message, observer=None, subscribing_
↪request_ids=[], **kwargs):
        ...

    @action()
    async def subscribe_to_comment_activity(self, request_id, **kwargs):
        await self.comment_activity.subscribe(request_id=request_id)
```

Parameters **consumer** (djangochannelsrestframework.consumers.AsyncAPIConsumer) –

Return type *Iterable*[str]

async unsubscribe(*consumer*, **args*, *request_id=None*, ***kwargs*)

This should be called to unsubscribe the current consumer.

args and kwargs passed here are provided to the *groups_for_consumer()* method to enable custom partitioning of events.

If the request_id is passed to the un-subscribe method then this will un-subscribe the requests with the same id that called the *subscribe()* method. If no *request_id* is provided then all subscribed requests for this consumer are un-subscribed.

```
class MyConsumer(GenericAsyncAPIConsumer):
    queryset = User.objects.all()
    serializer_class = UserSerializer

    @model_observer(Comments)
    async def comment_activity(self, message, observer=None, subscribing_
↪request_ids=[], **kwargs):
        ...

    @action()
    async def unsubscribe_to_comment_activity(self, request_id, **kwargs):
        await self.comment_activity.unsubscribe(request_id=request_id)
```

Parameters **consumer** (djangochannelsrestframework.consumers.AsyncAPIConsumer) –

Return type *Iterable*[str]

1.3.6 Permissions

class AllowAny
Allow any permission class

class BasePermission
Base permission class

Notes

You should extend this class and override the *has_permission* method to create your own permission class.

async has_permission (scope, consumer, action, **kwargs)

class IsAuthenticated
Allow authenticated only class

1.3.7 Tutorial

Djangochannelsrestframework allow you to use DRF serializers easily with django Channels v3 In this tutorial we will use this library to improve the chat tutorial from django Channels.

In this tutorial we redo the channels tutorial to use DCRF consumers.

1.3.7.1 Tutorial Part 1: Basic Setup

In this tutorial we will build a simple chat server. It will have two pages:

- An index view that lets you type the name of a chat room to join.
- A room view that lets you see messages posted in a particular chat room.

The room view will use a WebSocket to communicate with the Django server and listen for any messages that are posted.

We assume that you are familiar with basic concepts for building a Django site. If not we recommend you complete the Django tutorial first and then come back to this tutorial.

We assume that you have Django installed already and the Channels Tutorial made.

This will be the directory tree at the end of the Channels Tutorial and we will add the following python files:

- `serializers.py`
- `models.py`
- `routing.py`

```
mysite/
    manage.py
    mysite/
        __init__.py
        asgi.py
        settings.py
        urls.py
        wsgi.py
    chat/
```

(continues on next page)

(continued from previous page)

```
        __init__.py
        consumers.py
        models.py
        serializers.py
        routing.py
        templates/
            chat/
                index.html
                room.html
        tests.py
        urls.py
        views.py
```

Creating the Models

We will put the following code in the `models.py` file, to handle current rooms, messages and current users.

```
from django.db import models
from django.conf import settings
from django.contrib.auth.models import AbstractUser


class User(AbstractUser):
    pass


class Room(models.Model):
    name = models.CharField(max_length=255, null=False, blank=False, unique=True)
    host = models.ForeignKey(User, on_delete=models.CASCADE, related_name="rooms")
    current_users = models.ManyToManyField(User, related_name="current_rooms",␣
↪blank=True)

    def __str__(self):
        return f"Room({self.name} {self.host})"


class Message(models.Model):
    room = models.ForeignKey("chat.Room", on_delete=models.CASCADE, related_name=
↪"messages")
    text = models.TextField(max_length=500)
    user = models.ForeignKey(User, on_delete=models.CASCADE, related_name="messages")
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"Message({self.user} {self.room})"
```

Creating the Serializers

We will put the following code in the `serializers.py` file, to handle the serialization of the models created.

```
from .models import User, Room, Message
from rest_framework import serializers


class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        exclude = ["password"]


class MessageSerializer(serializers.ModelSerializer):
    created_at_formatted = serializers.SerializerMethodField()
    user = UserSerializer()

    class Meta:
        model = Message
        exclude = []
        depth = 1

    def get_created_at_formatted(self, obj:Message):
        return obj.created_at.strftime("%d-%m-%Y %H:%M:%S")

class RoomSerializer(serializers.ModelSerializer):
    last_message = serializers.SerializerMethodField()
    messages = MessageSerializer(many=True, read_only=True)

    class Meta:
        model = Room
        fields = ["pk", "name", "host", "messages", "current_users", "last_message"]
        depth = 1
        read_only_fields = ["messages", "last_message"]

    def get_last_message(self, obj:Room):
        return MessageSerializer(obj.messages.order_by('created_at').last()).data
```

Creating the Consumers

In the `consumers.py` file we will create only the room consumer for:

- Joining and leaving a room.
- Observing messages in that room.
- Observing the current users in the room.

```
import json
from django.shortcuts import get_object_or_404
from channels.generic.websocket import AsyncWebsocketConsumer
from channels.db import database_sync_to_async
```

(continues on next page)

(continued from previous page)

```
from django.utils.timezone import now
from django.conf import settings
from typing import Generator
from djangochannelsrestframework.generics import GenericAsyncAPIConsumer,␣
↪AsyncAPIConsumer
from djangochannelsrestframework.observer.generics import (ObserverModelInstanceMixin,␣
↪action)
from djangochannelsrestframework.observer import model_observer

from .models import Room, Message, User
from .serializers import MessageSerializer, RoomSerializer, UserSerializer


class RoomConsumer(ObserverModelInstanceMixin, GenericAsyncAPIConsumer):
    queryset = Room.objects.all()
    serializer_class = RoomSerializer
    lookup_field = "pk"

    async def disconnect(self, code):
        if hasattr(self, "room_subscribe"):
            await self.remove_user_from_room(self.room_subscribe)
            await self.notify_users()
        await super().disconnect(code)

    @action()
    async def join_room(self, pk, **kwargs):
        self.room_subscribe = pk
        await self.add_user_to_room(pk)
        await self.notify_users()

    @action()
    async def leave_room(self, pk, **kwargs):
        await self.remove_user_from_room(pk)

    @action()
    async def create_message(self, message, **kwargs):
        room: Room = await self.get_room(pk=self.room_subscribe)
        await database_sync_to_async(Message.objects.create)(
            room=room,
            user=self.scope["user"],
            text=message
        )

    @action()
    async def subscribe_to_messages_in_room(self, pk, request_id, **kwargs):
        await self.message_activity.subscribe(room=pk, request_id=request_id)

    @model_observer(Message)
    async def message_activity(
        self,
        message,
        observer=None,
```

(continues on next page)

(continued from previous page)

```
        subscribing_request_ids = [],
        **kwargs
    ):
        """
        This is evaluated once for each subscribed consumer.
        The result of `@message_activity.serializer` is provided here as the message.
        """
        # since we provide the request_id when subscribing we can just loop over them␣
↪here.
        for request_id in subscribing_request_ids:
            message_body = dict(request_id=request_id)
            message_body.update(message)
            await self.send_json(message_body)

    @message_activity.groups_for_signal
    def message_activity(self, instance: Message, **kwargs):
        yield 'room__{instance.room_id}'
        yield f'pk__{instance.pk}'

    @message_activity.groups_for_consumer
    def message_activity(self, room=None, **kwargs):
        if room is not None:
            yield f'room__{room}'

    @message_activity.serializer
    def message_activity(self, instance:Message, action, **kwargs):
        """
        This is evaluated before the update is sent
        out to all the subscribing consumers.
        """
        return dict(data=MessageSerializer(instance).data, action=action.value,␣
↪pk=instance.pk)

    async def notify_users(self):
        room: Room = await self.get_room(self.room_subscribe)
        for group in self.groups:
            await self.channel_layer.group_send(
                group,
                {
                    'type':'update_users',
                    'usuarios':await self.current_users(room)
                }
            )

    async def update_users(self, event: dict):
        await self.send(text_data=json.dumps({'usuarios': event["usuarios"]}))

    @database_sync_to_async
    def get_room(self, pk: int) -> Room:
        return Room.objects.get(pk=pk)

    @database_sync_to_async
```

(continues on next page)

(continued from previous page)

```
    def current_users(self, room: Room):
        return [UserSerializer(user).data for user in room.current_users.all()]

    @database_sync_to_async
    def remove_user_from_room(self, room):
        user:User = self.scope["user"]
        user.current_rooms.remove(room)

    @database_sync_to_async
    def add_user_to_room(self, pk):
        user:User = self.scope["user"]
        if not user.current_rooms.filter(pk=self.room_subscribe).exists():
            user.current_rooms.add(Room.objects.get(pk=pk))
```

Routing the Websocket

```
from django.urls import re_path
from . import consumers


websocket_urlpatterns = [
    re_path(r'ws/chat/room/$', consumers.RoomConsumer.as_asgi()),
]
```

1.3.7.2 Tutorial Part 2: Templates

We will edit the views, urls and templates for posting a Room form, and joining it.

We will edit the `index.html` file, for posting a new room.

```
{% extends "chat/layout.html" %}

{% block content %}
    What chat room would you like to enter?<br>
    <form method="POST">
        <input id="room-name-input" name="name" type="text" size="100"><br>
        <input id="room-name-submit" type="button" value="Enter">
    </form>
{% endblock content %}
```

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.index, name='index'),
    path('room/<int:pk>/', views.room, name='room'),

]
```

Editing existing views

We will edit the `views.py`

```
from django.shortcuts import render, reverse, get_object_or_404
from django.views.generic import TemplateView
from django.http import HttpResponseRedirect
from .models import User, Room, Message

def index(request):
    if request.method == "POST":
        name = request.POST.get("name", None)
        if name:
            room = Room.objects.create(name=name, host=request.user)
            HttpResponseRedirect(reverse("room", args=[room.pk]))
    return render(request, 'chat/index.html')

def room(request, pk):
    room:Room = get_object_or_404(Room, pk=pk)
    return render(request, 'chat/room.html', {
        "room":room,
    })
```

```
{% extends "chat/layout.html" %}
{% load static %}


{% block content %}
    <textarea id="chat-log" cols="100" rows="20"></textarea><br>
    <input id="chat-message-input" type="text" size="100"><br>
    <input id="chat-message-submit" type="button" value="Send">
{% endblock content %}

{% block footer %}
    <script>
        const room_pk = "{{ room.pk }}";
        const request_id = "{{ request.sessions.session_key }}";

        const chatSocket = new WebSocket(`ws://${window.location.host}/ws/chat/`);


        chatSocket.onopen = function(){
            chatSocket.send(
                JSON.stringify({
                    pk:room_pk,
                    action:"join_room",
                    request_id:request_id,
                })
            );
                            chatSocket.send(
                JSON.stringify({
                    pk:room_pk,
                    action:"retrieve",
```

(continues on next page)

(continued from previous page)

```
                    request_id:request_id,
                })
            );
                        chatSocket.send(
                JSON.stringify({
                    pk:room_pk,
                    action:"subscribe_to_messages_in_room",
                    request_id:request_id,
                })
            );
                        chatSocket.send(
                JSON.stringify({
                    pk:room_pk,
                    action:"subscribe_instance",
                    request_id:request_id,
                })
            );
        };

        chatSocket.onmessage = function (e) {
            const data = JSON.parse(e.data);
            switch (data.action) {
                case "retrieve":
                    setRoom(old => data.data);
                    setMessages(old => data.messages);
                    break;
                case "create":
                    setMessages(old => [...old, data])
                    break;
                default:
                    break;
            }
            break;
        };

        chatSocket.onclose = function(e) {
            console.error('Chat socket closed unexpectedly');
        };

        $('#chat-message-input').focus();
        $('#chat-message-input').on('keyup', function(e){
            if (e.keyCode === 13) {  // enter, return
                document.querySelector('#chat-message-submit').click();
            }
        });

        $('#chat-message-submit').on('click', function(e){
            const message = $('#chat-message-input').val();
            chatSocket.send(JSON.stringify({
                message: message,
                action: "create_message",
                request_id: request_id
```

(continues on next page)

(continued from previous page)

```
            }));
            $('#chat-message-input').val('') ;
        });

</script>
{% endblock footer %}
```

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

O

P

Q

R

S

U

V