

Django Rest Framework

Intro

For serializing python objects you can just use the python json module. For querysets you need to use django's serializers module.

Full example

Coding for entrepreneurs

Srvup project <https://www.codingforentrepreneurs.com/projects/srvup-membership/>

Django rest for srvup <https://www.codingforentrepreneurs.com/projects/django-rest-framework/>

IOs app for srvup <https://www.codingforentrepreneurs.com/projects/django-ios/>

Built in Django

There is some embedded functionality in Django for serialization.

```
from django.core import serializers

data = serializers.serialize("xml", SomeModel.objects.all())
```

or you can also use a serializer object directly:

```
XMLSerializer = serializers.get_serializer("xml")

xml_serializer = XMLSerializer()

xml_serializer.serialize(queryset)

data = xml_serializer.getvalue()
```

Have in mind:

simplejson and json don't work with django objects well. (So you can't use them for serializing querysets. On the other hand Django's built-in [serializers](#) can only serialize querysets filled with django objects:

```
data = serializers.serialize('json', self.get_queryset())

return HttpResponse(data, mimetype="application/json")
```

In case, the data you want to serialize contain a mix of django objects and dicts inside you need to do something. For example convert your model instances to dictionary using the Django built in `model_to_dict` method. Or use the `values()` Django method and then serialize to json.

Introduction

The point is to build the basic CRUD operations of your models. Django rest provides built in functionality for various things, for example convenient serialization of models with relationships, handling the serialization format (for example datetimes are converted to a format that can be easily parsed with javascript) etc.

The tutorial

Function based views

api/serializers.py

```
1 from rest_framework import serializers
2
3 from task.models import Task
4
5
6 class TaskSerializer(serializers.ModelSerializer):
7
8     class Meta:
9         model = Task
10        fields = ('title', 'description', 'completed')
```

Its very similar in construction with forms. You can either have a serializer for a model or not.

api/urls.py

```
1 from django.conf.urls import patterns, url
2
3 urlpatterns = patterns(
4     'api.views',
5     url(r'^tasks/$', 'task_list', name='task_list'),
6     url(r'^tasks/(?P<pk>[0-9]+)$', 'task_detail', name='task_detail'))
```

api/models.py

```
1 from django.db import models
2
3 class Task(models.Model):
4     completed = models.BooleanField(default=False)
5     title = models.CharField(max_length=100)
6     description = models.TextField()
```

Views.py in your app

```
1 from rest_framework import status
2 from rest_framework.decorators import api_view
3 from rest_framework.response import Response
4
5 from task.models import Task
6 from api.serializers import TaskSerializer
7
8
9 @api_view(['GET', 'POST'])
10 def task_list(request):
11     """
12     List all tasks, or create a new task.
13     """
14     if request.method == 'GET':
15         tasks = Task.objects.all()
16         serializer = TaskSerializer(tasks)
17         return Response(serializer.data)
18
19     elif request.method == 'POST':
20         serializer = TaskSerializer(data=request.DATA)
21         if serializer.is_valid():
22             serializer.save()
23             return Response(serializer.data, status=status.HTTP_201_CREATED)
24         else:
25             return Response(
26                 serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

The views

We deal with the serializer as if it was a normal django model

This is an example in which the serializer is connected with a model.

@api_view

We use the **@api_view** decorator which declares what http methods this view can deal with. If there is another method then it will be rejected and Django-rest will handle this.

serializer = TaskSerializer(tasks)

serializer = TaskSerializer(data=request.DATA)

We create a serializer instance by defining a queryset of the model or a dictionary with the appropriate data

serializer.data

It returns data as json

serializer.is_valid

runs validation checks that are built into the model similar with the form.is_valid

serializer.save()

it saves the model entry to the database

serializer.errors

a json string with the validation errors

Notice that along with the json string we return also a status code that defines the status of our response. But he doesn't return a status code for a successful get request.

serializer=TaskSerializer(task, data=request.DATA)

This way the serializer is instantiated from an existing task and is updated with the data from the request.

If it passes all the validation that is built into the model.

```

29 @api_view(['GET', 'PUT', 'DELETE'])
30 def task_detail(request, pk):
31     """
32     Get, update, or delete a specific task
33     """
34     try:
35         task = Task.objects.get(pk=pk)
36     except Task.DoesNotExist:
37         return Response(status=status.HTTP_404_NOT_FOUND)
38
39     if request.method == 'GET':
40         serializer = TaskSerializer(task)
41         return Response(serializer.data)
42
43     elif request.method == 'PUT':
44         serializer = TaskSerializer(task, data=request.DATA)
45         if serializer.is_valid():
46             serializer.save()
47             return Response(serializer.data)
48         else:
49             return Response(
50                 serializer.errors, status=status.HTTP_400_BAD_REQUEST)
51
52     elif request.method == 'DELETE':
53         task.delete()
54         return Response(status=status.HTTP_204_NO_CONTENT)

```

@api_view and APIView view wrappers

Without using the @api view decorator: It's worth noting that there are a couple of edge cases we're not dealing with properly at the moment. If we send malformed json, or if a request is made with a method that the view doesn't handle, then we'll end up with a 500 "server error" response. Still, this'll do for now.

This is what the @api_view decorator handles. It's a wrapper for the view. The APIView is the same but for class based views. The wrappers also provide behaviour such as returning 405 Method Not Allowed responses when appropriate, and handling any ParseError exception that occurs when accessing request.data with malformed input.

Notice that the class variables that are used in class based views to define a lot of things (like authentication_classes, permission_classes etc) are defined as decorators for function based views.

Request, Response and Content negotiation

- Django rest examines the Content-Type header of a request to determine the content type of the sent data and use the respective parser to read them. (for example convert the sent json to python native data types using the JSONParser)
- Django rest examines the Accept header or the format suffix of a request to determine the content type of the data the client wants and uses the respective parser to generate that data (for example convert the native python data types to json)

You can override the defaults by setting a global parser class, or by defining a parser for specific views.

```

request.POST # Only handles form data. Only works for 'POST' method.
request.data # Handles arbitrary data. Works for 'POST', 'PUT' and 'PATCH' methods.
return Response(data) # Renders to content type as requested by the client.

```

```

class Authenticate(generics.CreateAPIView):
    serializer_class = AuthSerializer

    def create(self, request):
        # the following is wrong, use request.data instead of request.POST
        serializer = AuthSerializer(request.POST) #Use request.data

```

Request object

REST framework introduces a Request object that extends the regular HttpRequest, and provides more flexible request parsing. The core functionality of the Request object is the **request.data** attribute, which is similar to request.POST, but more useful for working with Web APIs. request.POST only handles parsing form multipart data. request.data instead will handle either form data, or json data, or whatever other parsers you have configured.

Response object

REST framework also introduces a Response object, which is a type of TemplateResponse that takes unrendered content and uses content negotiation to determine the correct content type to return to the client.

Notice that you don't have to explicitly use `JSONResponse` you can just use `Response` and the content type will be decided based on **content negotiation** (reading the request's **Accept header** or the **format suffix of the request's url**). Also no need to `jsonparse` the request [`data = JSONParser().parse(request)`] we can directly do `serializer = SnippetSerializer(data=request.data)`

There is no need to tie our requests or responses to a given content type. `request.data` can handle incoming json requests, but it can also handle other formats. Similarly we're returning response objects with data, but allowing REST framework to render the response into the correct content type for us. No need to return a `JSONResponse` and also no need to `jsonparse` the request `data = JSONParser().parse(request)` we can directly do `serializer = SnippetSerializer(data=request.data)`

So if we want to control the format of the response we can do that using the **format_suffix_patterns** and then use something like <http://example.com/api/items/4/.json>. Notice that in this case you need to define also **format=None** as a view argument (to not take the accept header into account but use the suffix instead?). Or by using the **Accept header**: (using http) `http http://127.0.0.1:8000/snippets/ Accept:application/json # Request JSON` `http http://127.0.0.1:8000/snippets/ Accept:text/html # Request HTML` Similarly, we can control the format of the response that we send, using the **Content-Type header**. Have in mind that you can post using form data or using json data.

Response

Unlike regular `HttpResponse` objects, you do not instantiate `Response` objects with rendered content. Instead you pass in unrendered data, which may consist of any Python primitives. The renderers used by the `Response` class cannot natively handle complex datatypes such as Django model instances, so you need to serialize the data into primitive datatypes before creating the `Response` object. You can use REST framework's `Serializer` classes to perform this data serialization, or use your own custom serialization.

Wrapping API views

REST framework provides two wrappers you can use to write API views.

- The `@api_view` decorator for working with function based views.
- The `APIView` class for working with class-based views.

These wrappers provide a few bits of functionality such as making sure you receive `Request` instances in your view, and adding context to `Response` objects so that content negotiation can be performed.

The wrappers also provide behaviour such as returning 405 Method Not Allowed responses when appropriate, and handling any `ParseError` exception that occurs when accessing `request.data` with malformed input.

Serialize data

Instantiate a serializer with python datatypes (deserialized data): **`serializer = SnippetSerializer(data=data)`** we create a serializer instance with those datatypes. The `data` argument can be the `request.data`. The request data is python native datatypes.

- **`serializer.data`** Returns the python dictionary with the serializer's data. Notice that the data are python native datatypes.
- **`serializer.is_valid()`**
- **`serializer.errors`**
- **`serializer.validated_data`**
- **`serializer.save()`** It calls the `create` method of the `Serializer` class

`serializer = SnippetSerializer(Snippet.objects.all(), many=True)` We can also serialize many objects at once using a queryset and the `'many'` argument.

Deserialize data

Notice that the django rest Request object deserializes the request data to python datatypes into the request.data dictionary. So you don't have to externally use jsonparser for the request data.

```
from rest_framework.parsers import JSONParser
from django.utils.six import BytesIO
stream = BytesIO(content)
data = JSONParser().parse(stream) #data is python datatypes
serializer = SnippetSerializer(data=data)
serializer.is_valid()
serializer.save()
```

JSONParser

It seems that the jsonParser.parse method expects a file like object. So we create a stream of the json data using the BytesIO and we pass it to the parser. Which creates python native datatypes. Notice that we can do `data = JSONParser().parse(request)` so the request can be consumed in a stream like fashion

JsonResponse and JSONRenderer

Usually you don't have to use these methods, django rest will serialize the data to the proper content type from content negotiation.

- *JSONResponse*

`return JSONResponse(serializer.data)` it is used to return the json response to the client. The jsonResponse renders the python data to json. It uses the Jsonrenderer.render in its init function.

- *JSONRenderer*

`content = JSONRenderer().render(serializer.data)` The python dictionary is rendered to json. Now the data are in json format (strings have been encoded from Unicode using utf-8 encoding), not python native datatypes (a False is now false, a u'string' is now string etc). Now the data have been serialized.

Adding optional format suffixes to our URLs

To take advantage of the fact that our responses are no longer hardwired to a single content type let's add support for format suffixes to our API endpoints. Using format suffixes gives us URLs that explicitly refer to a given format, and means our API will be able to handle URLs such as `http://example.com/api/items/4.json`.

```
def snippet_list(request, format=None):
    ...
def snippet_detail(request, pk, format=None):
    ...

urlpatterns = [
    path('snippets/', views.snippet_list),
    path('snippets/<int:pk>', views.snippet_detail),]
urlpatterns = format_suffix_patterns(urlpatterns)
```

To make our views use the suffixes we need to use an extra argument named "format". Then we need to be able to extract the suffix from the url and to do so we use the format_suffix_patterns function

The client controls the format of the response that it wants:

- either by using the **Accept header**:
`http http://127.0.0.1:8000/snippets/ Accept:application/json # Request JSON`
`http http://127.0.0.1:8000/snippets/ Accept:text/html # Request HTML`
- Or by appending a format suffix:
`http http://127.0.0.1:8000/snippets.json # JSON suffix`
`http http://127.0.0.1:8000/snippets.api # Browsable API suffix`

Similarly, we can control the format of the response that we send from the server, using the **Content-Type header**. In responses, a Content-Type header tells the client what the content type of the returned content actually is. In requests, (such as POST or PUT), the client tells the server what type of data is actually sent.

Browsable api

Because the API chooses the content type of the response based on the client request, it will, by default, return an HTML-formatted representation of the resource when that resource is requested by a web browser. This allows for the API to return a fully web-browsable HTML representation. Having a web-browsable API is a huge usability win, and makes developing and using your API much easier. It also dramatically lowers the barrier-to-entry for other developers wanting to inspect and work with your API.

Using class based views

Writing your views from specialized to more generic format (here for the detailed view of an object)

1. Function based views
2. APIView class based view
3. Using mixin class based views
4. Using the already mixed generic class based view

1. Using function based views

```
@api_view(['GET', 'PUT', 'DELETE'])
def snippet_detail(request, pk):
    """
    Retrieve, update or delete a code snippet.
    """
    try:
        snippet = Snippet.objects.get(pk=pk)
    except Snippet.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    if request.method == 'GET':
        serializer = SnippetSerializer(snippet)
        return Response(serializer.data)

    elif request.method == 'PUT':
        serializer = SnippetSerializer(snippet, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    elif request.method == 'DELETE':
        snippet.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

2. Using the APIView class based view

```
class SnippetDetail(APIView):
    """
    Retrieve, update or delete a snippet instance.
    """
    def get_object(self, pk):
        try:
            return Snippet.objects.get(pk=pk)
        except Snippet.DoesNotExist:
            raise Http404

    def get(self, request, pk, format=None):
        snippet = self.get_object(pk)
        serializer = SnippetSerializer(snippet)
        return Response(serializer.data)

    def put(self, request, pk, format=None):
        snippet = self.get_object(pk)
        serializer = SnippetSerializer(snippet, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    def delete(self, request, pk, format=None):
        snippet = self.get_object(pk)
        snippet.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

3. Using mixins

```
class SnippetDetail(mixins.RetrieveModelMixin,
                    mixins.UpdateModelMixin,
                    mixins.DestroyModelMixin,
                    generics.GenericAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer

    def get(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)

    def put(self, request, *args, **kwargs):
        return self.update(request, *args, **kwargs)

    def delete(self, request, *args, **kwargs):
        return self.destroy(request, *args, **kwargs)
```

4. Using generic class based views

```
class SnippetDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer
```

Have in mind that there are also the read only generic views

- `generics.RetrieveAPIView`
 - `generics.ListAPIView`
- apart from the read/write
- `RetrieveUpdateDestroyAPIView`
 - `ListCreateAPIView`.

RetrieveUpdateDestroyAPIView and ListCreateAPIView generic classes can be replaced with the ModelViewSet

Using mixins

<pre>from snippets.models import Snippet from snippets.serializers import SnippetSerializer from rest_framework import mixins from rest_framework import generics class SnippetList(mixins.ListModelMixin, mixins.CreateModelMixin, generics.GenericAPIView): queryset = Snippet.objects.all() serializer_class = SnippetSerializer def get(self, request, *args, **kwargs): return self.list(request, *args, **kwargs) def post(self, request, *args, **kwargs): return self.create(request, *args, **kwargs)</pre>	<p>For the list and create view we inherit the core functionality from the GenericAPIView and from the mixins we take the <i>list</i> and <i>create</i> methods that we use in the get and post methods respectively.</p> <p>Similarly for a view that handles retrieve, update, delete we use the RetrieveModelMixin, UpdateModelMixin, DestroyModelMixin along with the GenericAPIView. From the mixins we take the retrieve, update and delete methods that we use in the get, put and delete methods.</p> <p>But this structure can be further simplified as you can see in the next table using the ListCreateAPIView and the RetrieveUpdateDestroyAPIView</p> <p>Notice that we can also use the ListAPIView or RetrieveAPIView (and probably the others also) on their own. We can just create a MyModelList class that inherits from ListAPIView and returns the models list.</p> <p>Notice that you can use the get_queryset method (the accessor of queryset) to define the queryset that returns the items to be listed. The difference is that the accessor returns an immutable queryset.</p>
---	---

CURL

<p>Check easily your views using CURL</p> <pre>curl http://localhost:9000/api/tasks/ curl -X POST http://localhost:9000/api/tasks/ -d "title=hello world&description=a whole new world" curl -X PUT http://localhost:9000/api/tasks/1 -d "title=hello world&description=be nice" curl -X PUT http://localhost:9000/api/tasks/1 -d "title=hello world&description=be nice&completed=True" curl -X DELETE http://localhost:9000/api/tasks/1</pre>

Httpie

http --form POST http://127.0.0.1:8000/snippets/ key=value

http --json POST http://127.0.0.1:8000/snippets/ key=value

http -a admin:password123 POST http://127.0.0.1:8000/snippets/ key=value

Authentication

Authentication

As it is right now, the serialized data passed to the serializer do not contain any information about the user that sent the data. This information is contained in the request, so we have to extract it in the view and pass it to the serializer's create method in order to use it when it saves the task. This way the task object (that requires a FK to user) will have its user. So the serializer must also have one more field that references the user (similarly we can extract any information from the request and pass it to the serializer).

Just for having in mind: In the user serializer we must also add a field which will be a relation field to the task model. This way the user deserialized data will contain also the related tasks. If we also add permissions which check if a user is authenticated the authentication of users is done with the default authentications of Django-rest which are SessionAuthentication and BasicAuthentication. You can change the authentication method if you want.

When we interact with the API through the web browser, we can login, and the browser session will then provide the required authentication for the requests. If we're interacting with the API programmatically (curl, httpie) we need to explicitly provide the authentication credentials on each request.

Authentication is always run at the very start of the view, before the permission and throttling checks occur, and before any other code is allowed to proceed. Note: Don't forget that authentication by itself won't allow or disallow an incoming request, it simply identifies the credentials that the request was made with.

<pre>from django.db import models class Task(models.Model): owner = models.ForeignKey('auth.User', related_name='tasks') completed = models.BooleanField(default=False) title = models.CharField(max_length=100) description = models.TextField() Serializer.py from rest_framework import serializers from task.models import Task class TaskSerializer(serializers.ModelSerializer): owner = serializers.ReadOnlyField('owner.username') class Meta: model = Task fields = ('title', 'description', 'completed', 'owner')</pre> <pre>views.py from rest_framework.generics import (ListCreateAPIView, RetrieveUpdateDestroyAPIView) from task.models import Task from api.serializers import TaskSerializer from api.permissions import IsOwnerOrReadOnly class TaskMixin(object): queryset = Task.objects.all() serializer_class = TaskSerializer permission_classes = (IsOwnerOrReadOnly,) def perform_create(self, serializer): serializer.save(owner=self.request.user) # def pre_save(self, obj): # obj.owner = self.request.user class TaskList(TaskMixin, ListCreateAPIView): pass class TaskDetail(TaskMixin, RetrieveUpdateDestroyAPIView): pass # Notice the above schema, where we use a "mixin" class to add some custom functionality and then the task list and task detail views inherit from that mixin too.</pre> <p>self and obj view method arguments Arguments of a method of class based view "self" refers of course to the View and contains all the variables and methods that the view has inherited as well as the Django added data like self.request. The second argument is an instance of the model which is currently handled. It's better to declare it with the name "obj" in order not to confuse it with the object argument of the mixin class which is a python internal thing.</p>	<p>We add an owner field to the model, so only the owner can update the task.</p> <p>The serializer is like a model. You define its fields. If it is connected with a model then you can define its fields to be the model fields using the Meta class. You can also define extra fields like you do in models but instead of owner=model.FieldType you do it with owner=serializer.FieldType.</p> <p>Explicitly define a serializer field In this case the owner = serializers.ReadOnlyField('owner.username') or (source='owner.username') uses the username property of the owner field of the model. <u>You do that instead of just defining it in the fields because in the later case it would return the id of the user.</u> We wanted to return the username so we explicitly defined the owner field. <u>This way the json response will contain the username of the user.</u> It is also read only. We only want to get the value of the owner field from the request object. Not from the posted json data. (We could have also used CharField(read_only=True) here.)</p> <p>Passing information from the request to the serializer. The proposed method is Overriding perform_create method, alternatively overriding the pre_save method</p> <p>Overriding perform_create method Another way that is described in the tutorial is to override the perform_create() method of the list view class. Probably this is the one that is more appropriate since <u>it is invoked only on create action, not on update.</u></p> <p>This way the create method of the respective serializer class will be passed an extra argument named "owner" along with the validated data from the request</p> <p>Overriding pre_save method We create one new class based view in order to define a pre save functionality by creating a pre_save method. This will execute before creation and update of an object (a task object as it is defined by the queryset) since these methods use the save(). Obj is a reference to the specific object that is used (the one that has been collected by its pk which has been retrieved from the url). So before the creation (but also an update) we set the owner model field to the current user.</p>
<pre>def perform_create(self, serializer): serializer.save(owner=self.request.user)</pre>	

Token based authentication

The authenticated user sends a token in every request as a Header.

View based authentication

We can define a distinct authentication for a specific view by defining explicitly an authentication class different from the default one.

In the views we import the **TokenAuthentication** class and define it in the **authentication_classes** tuple. That tells this view that we are going to use as authentication only token authentication.

At this point if a user makes a request without a token the data will still be returned. In order to return data only for authenticated users we need to add also a Permission to our viewset. We use the built in **IsAuthenticated** permission. Now if a user doesn't provide a token then a 401 response will be returned with the following data:

```
{"detail": "Authentication credentials were not provided."}
```

Views.py

```
13 from rest_framework import viewsets
12 from rest_framework.authentication import TokenAuthentication
11 from rest_framework.permissions import IsAuthenticated
10
9 from documents.models import Document
8
7 from .serializers import DocumentSerializer
6
5
4 class DocumentViewSet(viewsets.ModelViewSet):
3     queryset = Document.objects.all()
2     serializer_class = DocumentSerializer
1     authentication_classes = (TokenAuthentication,)
14     permission_classes = (IsAuthenticated, )
```

There is a separate Django rest app for it. You install it and migrate.

Then you can create tokens for your users

```
from django.contrib.auth.models import User
buddy = User.objects.get(username='buddy')
```

```
from rest_framework.authtoken.models import Token
```

```
token = Token.objects.create(user=buddy)
```

```
print token.key
```

Probably you create a new token for every user login or until the token expires. Where do you store these tokens? In your database. Then what's the fundamental difference between session and session data? Native mobile apps don't use cookies so cookie based session authentication doesn't work there.

Permissions

Some standard build in permissions:

AllowAny, IsOwnerOrReadOnly, IsAuthenticated, IsAuthenticatedOrReadOnly, IsAdminUser, DjangoModelPermissions, DjangoModelPermissionsOrAnonReadOnly, DjangoObjectPermissions, Custom Permissions (for role based permissions for example)

permissions.py

```
1 from rest_framework.permissions import BasePermission, SAFE_METHODS
2
3
4 class IsOwnerOrReadOnly(BasePermission):
5     def has_object_permission(self, request, view, obj):
6         if request.method in SAFE_METHODS:
7             return True
8
9         return obj.owner == request.user
```

So what we do with this permission is this: If there is a safe method then RETURN True, so the view will be executed.

But if the method is PUT or POST which is used for updating a task, then the permission method returns True only if the owner of the current obj that is handled in the view is the same with the current user. Only then the view will be executed and the obj will be updated. Otherwise we will get a **403 unauthorized response**. This way if you are a different user or there is no user (anonymous user) a 403 will be returned.

Safe methods: GET, HEAD, OPTIONS

Permissions

The permissions define who can access a given view.

In a permissions.py file inside the app. Django rest has built in permissions that you can use or you can create your own custom ones. We create a method that returns true or false. **Then in the view we define a `permission_classes` tuple. Every permission in this tuple must return true in order for the view to be executed.** What methods of the permission are being called in order to be evaluated for true or false? Every method of the permission class? Here the self is not the View since this is a permission. So there is also a view argument. The obj refers to the object of the view that uses this permission.

The permission **IsAuthenticatedOrReadOnly**, will ensure that authenticated requests get read-write access, and unauthenticated requests get read-only access. In the tutorial they use it for list and detail views. The detail view has also the custom **IsOwnerOrReadOnly** which is object level permission.

default permissions

In settings.py

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': ('rest_framework.permissions.IsAdminUser',),
    'PAGE_SIZE': 10}
```

Notice that you can define some default permissions to be valid for the whole API by defining them in settings. For example if we want the API to be accessible only by admin users we use the `IsAdminUser` permission.

```
urlpatterns += [
```

`url(r'^api-auth/', include('rest_framework.urls', namespace='rest_framework'))`, This url just adds a login view to the browsable API so we can login and use it, because now there is the need to send also a user along with the request.

```
]
```

Using only the **IsAuthenticatedOrReadOnly** a not logged in user can't create a snippet. If he makes a post request with the appropriate data (the request.user will be an anonymous user) no snippet will be created. **A 403 response will be returned.** If there is a logged in user, he could PUT the data and update a snippet. In order to allow only the owner to modify his snippets we must add the **IsOwnerOrReadOnly** permission. In the detail view I think that we could only use the second permission.

Role based permissions

```
1 from rest_framework.permissions import BasePermission
2
3 class IsAdmin(BasePermission):
4     def has_permission(self, request, view):
5         return request.user.is_admin
6
7 class IsEditor(BasePermission):
8     def has_permission(self, request, view):
9         return request.user.is_editor
10
11 class IsUser(BasePermission):
12     def has_permission(self, request, view):
13         return request.user.is_user
```

In most cases, role based access would involve some specific business logic. For instance, you might have roles such as user, editor and admin.

To restrict access based on custom parameters like these. You can write a custom permission class by inheriting the `BasePermission` class and overriding the `has_permission` method.

```
12 # all types of users are allowed to access this view
13 class APIView3(APIView):
14     permission_classes = (IsAdmin | IsEditor | IsUser,)
```

Viewsets and Routers

Viewsets

`ViewSet` classes provide a set of pre built views (list and detail views). This way we combine all of the functionality: list, create (from `ListCreateAPIView`) and get, put, delete (from `RetrieveUpdateDestroyAPIView`) in one View. They provide operations such as read, or update, and not method handlers such as get or put.

```
from snippets.views import SnippetViewSet, api_root

snippet_list = SnippetViewSet.as_view({
    'get': 'list',
    'post': 'create'
})

snippet_detail = SnippetViewSet.as_view({
    'get': 'retrieve',
    'put': 'update',
    'patch': 'partial_update',
    'delete': 'destroy'
})

urlpatterns = format_suffix_patterns([
    path('', api_root),
    path('snippets/', snippet_list, name='snippet-list'),
    path('snippets/<int:pk>', snippet_detail, name='snippet-detail')
])
```

A ViewSet class is only bound to a set of method handlers at the last moment, when it is instantiated into a set of views, typically by using a Router class which handles the complexities of defining the URL conf for you.

```

views
from rest_framework import viewsets
from rest_framework.routers import DefaultRouter

from task.models import Task
from api.serializers import TaskSerializer
from api.permissions import IsOwnerOrReadOnly

class TaskMixin(object):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer
    permission_classes = (IsOwnerOrReadOnly,)

    def perform_create(self, serializer):
        serializer.save(owner=self.request.user)

class TaskViewSet(TaskMixin, viewsets.ModelViewSet):
    pass

# Here we explicitly define the set of views (the view names). In the tutorial they put this in the urls.py before the urlpatterns. Notice that we create more than one view from the same ViewSet. These views along with the urlpatterns can be replaced with the use of a router.
task_list = TaskViewSet.as_view({
    'get': 'list',
    'post': 'create'
})
task_detail = TaskViewSet.as_view({
    'get': 'retrieve',
    'put': 'update',
    'patch': 'partial_update',
    'delete': 'destroy'
})

```

We can further simplify the previous view structure and replace the TaskList and TaskDetail views with one view that inherits from viewsets. The **TaskViewSet**. Again its name is quite descriptive. **Is all the views for the Task model**. This way we combine all of the functionality: list, create (from TaskList) and get, put, delete (from TaskDetail) in one View.

Mapping http methods to viewset actions

Then we need to define the callables (the **viewset actions**) that will be called in the urls and we will define there which http verb calls which action of the viewset (which are methods of the mixin classes). So if there is a get request for the tasklist the list method of the ListModelMixin will be called.

Urls.py

```

from django.conf.urls import patterns, url
from rest_framework.urlpatterns import format_suffix_patterns

from api.views import task_list, task_detail

urlpatterns = patterns(
    'api.views',
    url(r'^tasks/$', task_list, name='task_list'),
    url(r'^tasks/(?P<pk>[0-9]+)$', task_detail, name='task_detail'),
)

```

ModelViewSet and ReadOnlyModelViewSet

If we inherit from the **viewsets.ReadOnlyModelViewSet** instead of the viewsets.ModelViewSet then the viewset will have only the read only operations (it will not have the create, delete, update). With this we can create a UserViewSet viewset. So you could not create, update or delete a user using the API.

Custom endpoints (actions) with the @action decorator

<pre> class UserViewSet(viewsets.ModelViewSet): """ A viewset that provides the standard actions """ queryset = User.objects.all() serializer_class = UserSerializer @action(detail=True, methods=['post']) def set_password(self, request, pk=None): user = self.get_object() serializer = PasswordSerializer(data=request.data) if serializer.is_valid(): user.set_password(serializer.validated_data['password']) user.save() return Response({'status': 'password set'}) else: return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST) @action(detail=False) def recent_users(self, request): recent_users = User.objects.all().order_by('-last_login') page = self.paginate_queryset(recent_users) if page is not None: serializer = self.get_serializer(page, many=True) return self.get_paginated_response(serializer.data) serializer = self.get_serializer(recent_users, many=True) return Response(serializer.data) </pre>	<p>Notice that we've also used the <code>@action</code> decorator to create a custom action, named <code>highlight</code>. This decorator can be used to add any custom endpoints that don't fit into the standard create/update/delete style actions. <u>The URLs for custom actions by default depend on the method name itself</u>. If you want to change the way url should be constructed, you can include <code>url_path</code> as a decorator keyword argument.</p> <p>The two new actions will then be available at the urls <code>^users/{pk}/set_password/\$</code> <code>^users/recent_users</code> (I guess this one)</p> <p>to view all extra actions, call the <code>.get_extra_actions()</code> method.</p>
--	--

The handler methods only get bound to the actions when we define the URLConf.

Routers

If you use a viewset you can use router to automatically generate the urls for it.

Notice: If we need more control over the API URLs we can simply drop down to using regular class based views, and writing the URL conf explicitly.

<p>In the urls.py</p> <pre> from rest_framework.routers import DefaultRouter # This part replaces the need to explicitly define the task_1 ist and task_detail views of the previous chapter. task_router = DefaultRouter() task_router.register(r'tasks', TaskViewSet) </pre> <p>We say that we want to associate the TaskViewSet with the 'tasks' prefix. The router creates automatically the urls for the taskViewSet (so for the task model eventually since the viewset uses the serializer which is connected with the task model). So there is no need to define a <code>urlpatterns</code> for these views. The viewset urls will be served under the <code>api/tasks/</code> url. The <code>r'tasks'</code> is the URL prefix for the views</p>	<p>In the urls we used the common include api urls and then defining the api urls in the respective file. We replace this include by including the router. The api/urls will be created automatically by the router.</p> <p>Projects/urls.py</p> <pre> from django.conf.urls import patterns, include, url from api.views import task_router from django.contrib import admin admin.autodiscover() urlpatterns = patterns('', url(r'^api/', include(task_router.urls)), url(r'^admin/', include(admin.site.urls)),) </pre> <p>The DefaultRouter class automatically creates the API root view for us, so we can now delete the <code>api_root</code> method from our views module.</p>
<p>api/urls</p>	<p>A better configuration</p> <p>Another configuration is to use the normal include in the projects api like: <code>include(api.urls)</code> and in the <code>api/urls.py</code> to register the router. So the built in urls will work on <code>/api/files/</code> urls, like <code>api/files/1</code></p>

```

from django.conf.urls import include, url
1 from django.contrib import admin
2
3 from rest_framework.routers import DefaultRouter
4
5 from .views import DocumentViewSet
6
7 router = DefaultRouter()
8
9 router.register(r'files', DocumentViewSet)
10
11 urlpatterns = [
12     url(r'^$', include(router.urls)),
13 ]

```

Auth libraries

In Django rest, authentication always runs at the very start of the view, before the permission and throttling checks occur, and before any other code is allowed to proceed. Don't forget that authentication by itself won't allow or disallow an incoming request, it simply identifies the credentials that the request was made with.

Rest framework supports various auth schemes. The authentication schemes are always defined as a list of classes. REST framework will attempt to authenticate with each class in the list, and will set `request.user` and `request.auth` using the return value of the first class that successfully authenticates.

- The `request.user` property will typically be set to an instance of the `contrib.auth` package's `User` class.
- The `request.auth` property is used for any additional authentication information, for example, it may be used to represent an authentication token that the request was signed with.

dj-rest-auth

The most common library is [Dj-rest-auth](#). It uses [Django-allauth](#) for social authentication and [Django-simple-jwt](#) for jwt handling.

You can configure jwt related stuff, with simple-jwt configuration. For example modify the claims you want in the jwt. Expose a `verify-jwt` endpoint so that other services can verify a jwt they have without having access to your signing key (although you want all your microservices to be able to verify a jwt on their own to avoid the internal traffic) etc.

Your other services (other than your auth service) can use simple-jwt and have a `JWTStatelessUserAuthentication` backend to obtain a user from a validated token instead of a database lookup. When a request with a jwt header comes they construct a user object from it. so the `request.user` is now created from the jwt. If your other services don't use Django but fastApi you have to use another library to verify the jwt, for example `pyjwt` and extract the user details from it.

You can add any info you want in the jwt claims, for example the `user.role = "editor"` or something similar. Then you can have role based permissions in your services and use them for authorization on per view or per service basis.

Dj-rest-auth exposes a refresh endpoint to which you send a refresh token and if it's valid, you receive a new jwt token. so you can implement the jwt refresh logic in your front end (on axios interceptors for example). If you get a 401 "Token is invalid or expired" response, send the refresh token to the refresh endpoint. If you get a new jwt resend the previous request with the new jwt. If the refresh token was expired, then you get a 401 "Token is invalid or expired". In this case you log the user out and redirect him to the login page. In any case, if you get a 403 unauthorized access response you don't do anything jwt related. It means that your jwt is valid but you don't have permissions to access that resource.

If you use axios you can use axios interceptors (that run before any then or catch logic), and add the refresh logic there. you can intercept requests before they are sent and add logic. You can intercept responses too. this is an example <https://www.bezkoder.com/react-refresh-token/>

<https://srini-dev.hashnode.dev/authentication-refresh-token-flow-with-nextjs-typescript-react-query-and-axios-interceptors>
(Authentication & Refresh token flow with Nextjs, Typescript, React Query and axios interceptors.)

Django-simple-jwt

Stateless User Authentication

The JWTStatelessUserAuthentication backend's authenticate method does not perform a database lookup to obtain a user instance. Instead, it returns a `rest_framework_simplejwt.models.TokenUser` instance which acts as a stateless user object backed only by a validated token instead of a record in a database. This can facilitate developing single sign-on functionality between separately hosted Django apps which all share the same token secret key.

Signing key

Since Simple JWT defaults to using 256-bit HMAC signing, the `SIGNING_KEY` setting defaults to the value of the `SECRET_KEY` setting for your django project.

HMAC isn't symmetric encryption, it doesn't use a private-public key pair, but only one key. You can change the algorithm to RSA if you want to have a public private key. The public key in this case, is the `VERIFYING_KEY` setting.

Return 401 if jwt expired or invalid

Modify jwt claims

Issue a new jwt using a refresh token sent to a refresh view

Token types

Access token + refresh token, sliding tokens (they contain both an expiration claim and a refresh expiration claim). Sliding tokens offer a more convenient experience to users of tokens with the trade-offs of being less secure and, in the case that the blacklist app is being used, less performant.

A refresh token HAS to be stored on the server side. You shouldn't leverage the "self-contained" property of JWT for a refresh token. Doing so leaves you with no way to revoke refresh tokens other than changing your private key. Also, store it in http only cookie for security.

Blacklist app

If you enable it, when a refresh or sliding token is generated, is added to an outstanding list (not expired list). There is also the blacklist list. In any authenticated request when the jwt needs to be authenticated, the refresh and sliding tokens are checked if they are in the blacklist. So you have a db lookup for every with request if you use sliding tokens.

You can implement instant logout with the blacklist app. You have to use sliding tokens for that. In logout you move the token to the blacklist. Then a subsequent request with the same jwt will return a 403. The disadvantage is that you need sliding tokens

which means db lookup at every auth request. So it's better to use short lived access tokens with longer lived refresh tokens and you only make a db lookup when the refresh token is sent. I guess that the front end should send the refresh token before the access token expires so that it receives a new one. Then every other service will use that one.

The blacklist app also provides a management command, `flushexpiredtokens`, which will delete any tokens from the outstanding list and blacklist that have expired. You need a daily cron job for that.

Old

Django rest framework offers a built in token authentication mechanism that you could use. This mechanism has though some limitations:

- DRF tokens are limited to one per user. This does not facilitate securely signing in from multiple devices, as the token is shared. It also requires all devices to be logged out if a server-side logout is required (i.e. the token is deleted).
- DRF tokens are stored unencrypted in the database. This would allow an attacker unrestricted access to an account with a token if the database were compromised.
- DRF tokens track their creation time, but have no inbuilt mechanism for tokens expiring.

External libraries like `knox` and `simple jwt` offer solutions to these issues along with additional functionality. For example they provide django rest framework endpoints to handle basic actions like registration, login, logout, password reset etc.

Libraries

- **dj-rest-auth** It is a maintained fork of `django-rest-auth` which uses `django-allauth`. `django-rest-auth` is not actively maintained, `dj-rest-auth` is. You can use `jwt` and social authentication. It uses `django-rest-framework-simplejwt` for `jwt` authentication. Seems the best choice as of late 2020.
- **simplejwt + djoser + python-all-auth** this would be an alternative for implementing `jwt` or social authentication using a pool of libraries.

The **django-rest-framework-jwt** uses the Django-rest convention and adds the read JWT to the **request.auth** variable. The check for authentication is done by this library using the `JSONWebTokenAuthentication` authentication class, you don't do it manually.

Authentication example back and front

Back end

jwt authentication example using `knox` (an `simplejwt` alternative)

Think of it as creating views for the user model. We create a register serializer (which is a user model serializer with one additional field, the password), but we want highly customized views so we don't use the `ModelViewSet` (to create, list, update etc. a user) but specialized individual views. The user model serializer is only used to retrieve a user (without retrieving his password).

Registering a user

<pre> 5 # User Serializer 6 class UserSerializer(serializers.ModelSerializer): 7 class Meta: 8 model = User 9 fields = ('id', 'username', 'email') 10 11 # Register Serializer 12 class RegisterSerializer(serializers.ModelSerializer): 13 class Meta: 14 model = User 15 fields = ('id', 'username', 'email', 'password') 16 extra_kwargs = {'password': {'write_only': True}} 17 18 def create(self, validated_data): 19 user = User.objects.create_user(validated_data 20 ['username'], validated_data['email'], 21 validated_data['password']) 22 23 return user </pre>	<p>We create a registerSerializer which will create a User object with the validated data.</p> <p>We create a register view (here he calls it registerAPI).</p>
<pre> from knox import AuthToken 6 # Register API 7 class RegisterAPI(generics.GenericAPIView): 8 serializer_class = RegisterSerializer 9 10 def post(self, request, *args, **kwargs): 11 serializer = self.get_serializer(data=request.data) 12 serializer.is_valid(raise_exception=True) 13 user = serializer.save() 14 return Response({ 15 "user": UserSerializer(user, 16 context=self.get_serializer_context()).data, 17 "token": AuthToken.objects.create(user) 18 }) </pre>	<p>The context dictionary can be used within any serializer field logic, such as a custom <code>.to_representation()</code> method, by accessing the <code>self.context</code> attribute.</p> <p><code>get_serializer_context(self)</code> Returns a dictionary containing any extra context that should be supplied to the serializer. Defaults to including 'request', 'view' and 'format' keys.</p> <p># Return a 400 response if the data was invalid. <code>serializer.is_valid(raise_exception=True)</code></p> <p>In this example he uses the knox package for managing authentication.</p> <p>Then we set up the urls and we can make a post request to create a user.</p>

Login user

We have to create a distinct serializer to handle the login data received from a login request.

<pre> 23 # Login Serializer 24 class LoginSerializer(serializers.Serializer): 25 username = serializers.CharField() 26 password = serializers.CharField 27 28 def validate(self, data): 29 user = authenticate(**data) 30 if user and user.is_active: 31 return user 32 raise serializers.ValidationError("Incorrect 33 Credentials") 34 35 # Login API 36 class LoginAPI(generics.GenericAPIView): 37 serializer_class = LoginSerializer 38 39 def post(self, request, *args, **kwargs): 40 serializer = self.get_serializer(data=request.data) 41 serializer.is_valid(raise_exception=True) 42 user = serializer.validated_data 43 return Response({ 44 "user": UserSerializer(user, 45 context=self.get_serializer_context()).data, 46 "token": AuthToken.objects.create(user) 47 }) </pre>	<p>Use <code>authenticate()</code> to verify a set of credentials. It takes credentials as keyword arguments, username and password for the default case, checks them against each authentication backend, and returns a User object if the credentials are valid for a backend.</p> <p>To login, a user sends a username and a password. The password is send as plain text, but is encrypted by django before stored in the database. If the user is authenticated successfully a token is created for that user and returned in the response.</p>
--	--

Accessing user specific content

The typical way to check if a user is authenticated in django rest is using the build in permissions. If the user is not authenticated then he would not be able to access a view that uses the isAuthenticated permission. The actual authentication check is performed transparently depending on your authentication backend. If the user is authenticated, he will be added to the request object as request.user

Get a user example

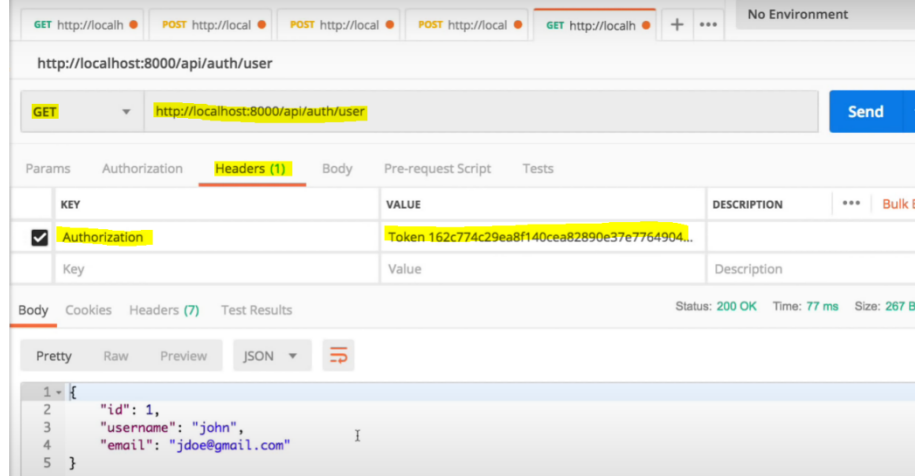
```
32 # Get User API
33 class UserAPI(generics.RetrieveAPIView):
34     permission_classes = [
35         permissions.IsAuthenticated,
36     ]
37     serializer_class = UserSerializer
38
39     def get_object(self):
40         return self.request.user
```

Notice that since we only want to get a user, we create a RetrieveAPIView. We use the isAuthenticated permission. This will check the token sent with the request as a header, and will try to validate it against the existing users. (if the authentication backend is token authentication). The request to this endpoint will not contain any data, just an authorization header with a token. django is able to get the user from the received token. This is done transparently.

We can test the endpoints with postman. The token must be added as a request header like -> Authorization: Token the-token

If you don't provide a token you will get a message "authentication credentials were not provided".

If you give an invalid token "Invalid token"



The screenshot shows a Postman interface for a GET request to `http://localhost:8000/api/auth/user`. The 'Headers' tab is active, showing an 'Authorization' header with the value 'Token 162c774c29ea8f140cea82890e37e7764904...'. The 'Body' tab shows a JSON response: `{ "id": 1, "username": "john", "email": "jdoe@gmail.com" }`. The status is 200 OK, time is 77 ms, and size is 267 B.

user specific content example

```
1 from leads.models import Lead
2 from rest_framework import viewsets, permissions
3 from .serializers import LeadSerializer
4
5 # Lead Viewset
6 class LeadViewSet(viewsets.ModelViewSet):
7     permission_classes = [
8         permissions.IsAuthenticated,
9     ]
10
11     serializer_class = LeadSerializer
12
13     def get_queryset(self):
14         return self.request.user.leads.all()
15
16     def perform_create(self, serializer):
17         serializer.save(owner=self.request.user)
```

Notice that when we want the viewset to return the items of the logged in user, we override the `get_queryset` method (instead of defining a `queryset` class variable).

We also use the `IsAuthenticated` permission.

User logout

http://localhost:8000/api/auth/logout		Knox offers a build in view for that. You should make a post request to that url with the token in the header. It deletes the token from the database.
POST	http://localhost:8000/api/auth/logout	
Params	Authorization	Headers (1)
KEY	VALUE	
<input checked="" type="checkbox"/> Authorization	Token c112affdfd56fc206585c8b87bdbac7ee9478...	
Key	Value	

Tips

Object of type 'AuthToken' is not JSON serializable

Solution: "token": AuthToken.objects.create(user)[1]

Front end

We use privateRoutes

Every time the main app component is loaded (in the componentDidMount lifecycle method) we should make a call to the get user endpoint (sending the token) to see if a user is returned. This is done whenever we reload the whole page. Is this necessary? When the user is logged in we store the token not only in redux state but in local storage too, which doesn't empty on reload. Isn't enough to login get a token and just use that token for every request if it is deleted in the server then the user will logout. Whats the point of the continuous checking?

On reload page the redux state is initialized and the token value is taken from local storage. But we need to check if this token is still valid. Probably for avoiding the following case: if for any reason the token is deleted in the server without a client action (logout) then the client would have an invalid token. But it will not be able to receive private data from the server, in any call for private data the call will fail and the token will be deleted from local storage. So you can implement this automatic check on reload to avoid that small erroneous behavior.

Not only for that. In the shown example, on failure to a protected route for example to get the todos, (that will fail if you call it with invalid token) he doesn't initializes the state by dispatching an action.

Actions

USER_LOADING

USER_LOADED

AUTH_FAIL

First we load a user, if he is logged in we will dispatch the user_loaded action if not the auth_fail one.

In the auth_fail we remove the token from the local storage.

```

18 case USER_LOADING:
19   return {
20     ...state,
21     isLoading: true
22   };
23 case USER_LOADED:
24   return {
25     ...state,
26     isAuthenticated: true,
27     isLoading: false,
28     user: action.payload
29   };
30 case LOGIN_SUCCESS:
31   localStorage.setItem("token",
32     action.payload.token);
33   return {
34     ...state,
35     ...action.payload,
36     isAuthenticated: true,
37     isLoading: false
38   };
39 case AUTH_ERROR:
40 case LOGIN_FAIL:
41   localStorage.removeItem("token");
42   return {

```

A part of the auth reducer to see the basic logic.
 On login we store the token to local storage.
 On fail to authenticate, we set up the state to the initial state.

Register user example

In the register component

```

onSubmit = e => {
  e.preventDefault();
  const { username, email, password, password2 } =
  this.state;
  if (password !== password2) {
    this.props.createMessage({ passwordNotMatch:
      "Passwords do not match" });
  } else {
    const newUser = {
      email,
      username,
      password
    };
    this.props.register(newUser);
  }
};

```

On form submit, calls the register action with a newUser object containing the email, username and psw. The form's input fields are connected with the component's state (it is a controlled component)

<p>Register</p> <pre> export const register = ({ username, password, email }) => dispatch => { // Headers const config = { headers: { "Content-Type": "application/json" } }; // Request Body const body = JSON.stringify({ username, email, password }); axios .post("/api/auth/register", body, config) .then(res => { dispatch({ type: REGISTER_SUCCESS, payload: res.data }); }) .catch(err => { dispatch(returnErrors(err.response.data, err.response.status)); }); dispatch({ type: REGISTER_FAIL, payload: null }); } </pre>	<p>action</p> <p>Catches the new User object properties, stringify them and posts them to the register url. Depending on the result it dispatches the corresponding action</p> <p>For the redirection after a successful registration, in the register component's render method we check if is authenticated and redirect to root.</p> <pre> render() { if (this.props.isAuthenticated) { return <Redirect to="/" />; } } </pre> <p>So when you register it automatically redirects you</p>
---	--

Protected endpoints

<pre> 22 // DELETE LEAD 23 export const deleteLead = id => (dispatch, getState) => { 24 axios 25 .delete(`/api/leads/\${id}/`, tokenConfig(getState)) 26 .then(res => { 27 dispatch(createMessage({ deleteLead: "Lead Deleted" })); 28 dispatch({ 29 type: DELETE_LEAD, 30 payload: id 31 }); 32 }) 33 .catch(err => console.log(err)); 34 }; </pre>	<p>Any protected route must send the token as a header. So he made a function the tokenConfig that gets the token from the redux state and adds it to header using a config variable (as is the standard for adding headers to axios calls). Then he just calls that function in the axios call for all protected endpoints</p>
--	---

Renderers

Content negotiation

The set of valid renderers for a view is always defined as a list of classes. When a view is entered REST framework will perform content negotiation on the incoming request, and determine the most appropriate renderer to satisfy the request.

The basic process of content negotiation involves examining the request's Accept header, to determine which media types it expects in the response. Optionally, format suffixes on the URL may be used to explicitly request a particular representation. For example the URL http://example.com/api/users_count.json might be an endpoint that always returns JSON data.

Apart from json you can send a response in html either already rendered (pre-rendered) html, or html to be rendered using django templates. Notice that if the request doesn't contain an Accept header the first defined default rendered will be used. So in general you can serve both html and json from your api depending on the requested view. If your API includes views that can serve both regular webpages and API responses depending on the request, then you might consider making TemplateHTMLRenderer your default renderer, in order to play nicely with older browsers that send broken accept headers.

The renderers used by the Response class cannot natively handle complex datatypes such as Django model instances, so you need to serialize the data into primitive datatypes before creating the Response object. You can use REST framework's Serializer classes to perform this data serialization, or use your own custom serialization.

The default renderers can be set globally in settings or individually for a specific view using the **renderer_classes** tuple for class based views and the **@renderer_classes** decorator for function based views.

There are various renderer classes:
JSONRenderer, **TemplateHTMLRenderer**, **StaticHTMLRenderer** (for sending pre-rendered html which is actually a string), **BrowsableAPIRenderer**, **AdminRenderer**, **HTMLFormRenderer** (Renders data returned by a serializer into an HTML form). There are also **Custom renderers**.

A view can return more than one type of media types depending on the accepted media type that can be accessed from the request.accepted renderer. So you can say **if request.accepted_renderer == 'html'** :

Format suffixes

Using format suffixes gives us URLs that explicitly refer to a given format, and means our API will be able to handle URLs such as *http://example.com/api/items/4.json*.

Format suffixes as url parameters

?format=html and ?format=json

Apart from the url format suffixes you can also use a url parameter to define the format. This behavior is controlled using the URL_FORMAT_OVERRIDE setting.

By default, the API will return the format specified by the headers, which in the case of the browser is HTML. The format can be specified using ?format= in the request, so you can look at the raw JSON response in a browser by adding ?format=json to the URL

Rendering HTML

<pre>class ProfileList(APIView): renderer_classes = [TemplateHTMLRenderer] template_name = 'profile_list.html' def get(self, request): queryset = Profile.objects.all() return Response({'profiles': queryset})</pre>	The TemplateHTMLRenderer class expects the response to contain a dictionary of context data, and renders an HTML page based on a template that must be specified either in the view or on the response.
<pre>from rest_framework import renderers from rest_framework.response import Response class SnippetHighlight(generics.GenericAPIView): queryset = Snippet.objects.all() renderer_classes = (renderers.StaticHTMLRenderer,) def get(self, request, *args, **kwargs): snippet = self.get_object() return Response(snippet.highlighted)</pre>	We want to have a view that returns only one (or some) of the fields of the instance in a pre-rendered html. To do so we don't use a built in view. We create our own inheriting from the base generic view and override the get method to return only the fields that we want. The snippet.highlighted field that we return must be a string (html code in a string like "<h1>test</h1>") in order to be used as a pre-rendered html as the StaticHTMLRenderer defines that this view returns pre-rendered html code.

Rendering Forms

Serializers may be rendered as forms by using the `render_form` template tag, and including the serializer instance as context to the template

<pre>class ProfileDetail(APIView): renderer_classes = [TemplateHTMLRenderer] template_name = 'profile_detail.html' def get(self, request, pk): profile = get_object_or_404(Profile, pk=pk) serializer = ProfileSerializer(profile) return Response({'serializer': serializer, 'profile': profile}) def post(self, request, pk): profile = get_object_or_404(Profile, pk=pk) serializer = ProfileSerializer(profile, data=request.data) if not serializer.is_valid(): return Response({'serializer': serializer, 'profile': profile}) serializer.save() return redirect('profile-list')</pre>	<p>Profile_detail.html</p> <pre>{% load rest_framework %} <html><body> <h1>Profile - {{ profile.name }}</h1> <form action="{% url 'profile-detail' pk=profile.pk %}" method="POST"> {% csrf_token %} {% render_form serializer %} <input type="submit" value="Save"> </form> </body></html></pre>
--	---

The `render_form` tag takes an optional `template_pack` argument, that specifies which template directory should be used for rendering the form and form fields. The built-in styles are horizontal, vertical, and inline

<pre>class LoginSerializer(serializers.Serializer): email = serializers.EmailField(max_length=100, style={'placeholder': 'Email', 'autofocus': True}) password = serializers.CharField(max_length=100, style={'input_type': 'password', 'placeholder': 'Password'}) remember_me = serializers.BooleanField()</pre>	<p>Notice that in this example we create a new serializer just for creating the form (while in the previous case the serializer was a model serializer)</p> <p>And then in the template:</p> <pre>{% render_form serializer template_pack='rest_framework/vertical' %}</pre>
--	--

Serializer fields can have their rendering style customized by using the `style` keyword argument.

<pre>author = serializers.HyperlinkedRelatedField(queryset=User.objects.all(), style={'base_template': 'input.html'})</pre>	<p>Handling ChoiceField with large numbers of items.</p> <p>When a relationship or ChoiceField has too many items, rendering the widget containing all the options can become very slow, and cause the browsable API rendering to perform poorly. The simplest option in this case is to replace the select input with a standard text input.</p>
---	---

Returning a single field in a pre-rendered html response

	<p>The “highlighted” field</p> <p>The highlight field is a TextField that uses the text from another field of the model (the code field) and styles it using the pygments python module. To create it we override the save method of the Snippet model. First we do the things that we want, we define the value of the field and</p>
--	---

<pre>def save(self, *args, **kwargs): """ Use the 'pygments' library to create a highlighted HTML representation of the code snippet. """ lexer = get_lexer_by_name(self.language) linenos = self.linenos and 'table' or False options = self.title and {'title': self.title} or {} formatter = HtmlFormatter(style=self.style, linenos=linenos, full=True, **options) self.highlighted = highlight(self.code, lexer, formatter) super(Snippet, self).save(*args, **kwargs) url(r'^snippets/(?P<pk>[0-9]+)/highlight/\$', views.SnippetHighlight.as_view(), name='snippet-highlight'),</pre>	<p>then we call the super save method in order to do its things and save the fields (along with the value that we have defined for the highlighted field)</p> <p>Then we add it to the serializer (a HyperlinkedModelSerializer) as a url field (HyperlinkedIdentityField) which points to the snippet-highlight view as it is named in the urls.</p>
---	---

Creating a viewset for the Snippet views

<pre>from rest_framework.decorators import detail_route class SnippetViewSet(viewsets.ModelViewSet): """ This viewset automatically provides 'list', 'create', 'retrieve', 'update' and 'destroy' actions. Additionally we also provide an extra 'highlight' action. """ queryset = Snippet.objects.all() serializer_class = SnippetSerializer permission_classes = (permissions.IsAuthenticatedOrReadOnly, IsOwnerOrReadOnly,) @detail_route(renderer_classes=[renderers.StaticHTMLRenderer]) def highlight(self, request, *args, **kwargs): snippet = self.get_object() return Response(snippet.highlighted) def perform_create(self, serializer): serializer.save(owner=self.request.user)</pre>	<p>@detail_route</p> <p>With this decorator we define a custom action for the viewset. Notice that we can define a different renderer for this action. By default it responds to GET requests. By default its url is the name of the method (here /highlight/). This custom action replaces the custom generic view snippet-highlight as defined in the previous case.</p> <p>We add also the perform_create method that we had added in the SnippetList view, in order to add the owner field to the snippet serializer. We had it in the list view since this view handles also the create action (post method) apart from the list action. So by defining it in the viewset, it will be executed before the create action of the viewset (whatever class it comes from).</p>
---	--

Serializers

Serializer and ModelSerializer

<pre> from rest_framework import serializers from snippets.models import Snippet, LANGUAGE_CHOICES, STYLE_CHOICES class SnippetSerializer(serializers.Serializer): pk = serializers.IntegerField(read_only=True) title = serializers.CharField(required=False, allow_blank=True, max_length=100) code = serializers.CharField(style={'base_template': 'textarea.html'}) linenos = serializers.BooleanField(required=False) language = serializers.ChoiceField(choices=LANGUAGE_CHOICES, default='python') style = serializers.ChoiceField(choices=STYLE_CHOICES, default='friendly') def create(self, validated_data): """ Create and return a new `Snippet` instance, given the validated data. """ return Snippet.objects.create(**validated_data) def update(self, instance, validated_data): """ Update and return an existing `Snippet` instance, given the validated data. """ instance.title = validated_data.get('title', instance.title) instance.code = validated_data.get('code', instance.code) instance.linenos = validated_data.get('linenos', instance.linenos) instance.language = validated_data.get('language', instance.language) instance.style = validated_data.get('style', instance.style) instance.save() return instance </pre> <p>You can create explicitly a serializer for a model instead of using the <code>ModelSerializer</code> class (just for demonstration here)</p>	<p><u>Serializers allow complex data such as queriesets and model instances to be converted to native Python datatypes that can then be easily rendered into JSON, XML or other content types.</u> Serializers also provide deserialization, allowing parsed data to be converted back into complex types, after first validating the incoming data.</p> <p>Have in mind the term non-model serializers for referring to serializers that don't refer to a Django model. Have in mind this SO question about the process of using serializers for non model data: http://stackoverflow.com/questions/13603027/django-rest-framework-non-model-serializer</p> <p>A note about permissions in non model objects: http://jsatt.com/blog/abusing-django-rest-framework-part-1-non-model-endpoints/ The one caveat when using <code>APIView</code> is to watch what permissions are applied to the view. Since it is not associated with a model, any permission that expects a model object to be available, like the <code>DjangoModelPermission</code> that is applied by default, will fail, so you should be explicit about permissions (here he defined an empty tuple).</p>
---	---

HyperlinkedModelSerializer

The `HyperlinkedModelSerializer` class is similar to the `ModelSerializer` class except that it uses hyperlinks to represent relationships, rather than primary keys. By default adds a url field to the serializer that points to the detail view of the object that is serialized. This url field is a `HyperlinkedIdentityField`. Any relationships on the model will be represented using a `HyperlinkedRelatedField` serializer field.

The `HyperlinkedModelSerializer` has the following differences from `ModelSerializer`:

- It does not include the id field by default.
- It includes a url field, using `HyperlinkedIdentityField`.
- Relationships use `HyperlinkedRelatedField`, instead of `PrimaryKeyRelatedField`.

Raising an exception on invalid data

The `.is_valid()` method takes an optional `raise_exception` flag that will cause it to raise a `serializers.ValidationError` exception if there are validation errors.

These exceptions are automatically dealt with by the default exception handler that REST framework provides, and will return HTTP 400 Bad Request responses by default.

Return a 400 response if the data was invalid.

`serializer.is_valid(raise_exception=True)`

Serializer relations

Dealing with relationships between entities is one of the more challenging aspects of Web API design. There are a number of different ways that we might choose to represent a relationship:

- Using primary keys.
- Using hyperlinking between entities.

- Using a unique identifying slug field on the related entity.
- Using the default string representation of the related entity.
- Nesting the related entity inside the parent representation.
- Some other custom representation.

REST framework supports all of these styles, and can apply them across forward or reverse relationships, or apply them across custom managers such as generic foreign keys.

Reverse relationship

<p>The relationship</p> <pre>class Album(models.Model): album_name = models.CharField(max_length=100) artist = models.CharField(max_length=100) class Track(models.Model): album = models.ForeignKey(Album, related_name='tracks') order = models.IntegerField() title = models.CharField(max_length=100) duration = models.IntegerField() class Meta: unique_together = ('album', 'order') ordering = ['order'] def __unicode__(self): return '%d: %s' % (self.order, self.title)</pre> <p>Related fields Arguments many=True must be used in any –to Many relation. The foreign key is actually One to Many so we need to use the many=True. The queryset argument is only ever required for writable relationship field. So for read-only fields it isn't needed. In this case it would be queryset = models.Track.objects.all() I only saw it collecting all the related entries.</p>	<p>Reverse relation <u>Manually add the related field to the serializer.</u> When we want to serialize the related model (the one that doesn't contain the foreign key) we should manually define a “field” in the serializer which will define the entries with which this entry is related (it will return the entries of the reverse relation). You must also explicitly define it in the fields argument.</p> <p>StringRelatedField</p> <pre>class AlbumSerializer(serializers.ModelSerializer): tracks = serializers.StringRelatedField(many=True) class Meta: model = Album fields = ('album_name', 'artist', 'tracks')</pre> <p>When the albumserializer serializes the data it returns a list with the unicode__ returns of the related model.</p> <pre>{ 'album_name': 'Things We Lost In The Fire', 'artist': 'Low', 'tracks': ['1: Sunflower', '2: Whitetail', '3: Dinosaur Act', ...] }</pre>
<p>PrimaryKeyRelatedField Read-write</p> <pre>class AlbumSerializer(serializers.ModelSerializer): tracks = serializers.PrimaryKeyRelatedField(many=True, read_only=True) class Meta: model = Album fields = ('album_name', 'artist', 'tracks')</pre> <p>It returns a list with the pks of the related models. By default this field is read-write, although you can change this behavior using the read_only flag. So if you save the serializer the related models will not be created and saved.</p> <p>SlugRelatedField Read-write You define which field of the target related model you want the serializer to return</p>	<p>HyperlinkedRelatedField Read-write</p> <div style="display: flex; align-items: flex-start;"> <pre>class AlbumSerializer(serializers.ModelSerializer): tracks = serializers.HyperlinkedRelatedField(many=True, read_only=True, view_name='track-detail') class Meta: model = Album fields = ('album_name', 'artist', 'tracks')</pre> <pre>{ 'album_name': 'Graceland', 'artist': 'Paul Simon', 'tracks': ['http://www.example.com/api/tracks/45/', 'http://www.example.com/api/tracks/46/', 'http://www.example.com/api/tracks/47/', ...] }</pre> </div> <p>It returns a list with the related model detail views urls. You can customize this field to create more complex urls with more passed parameters (more lookup fields)</p> <p>HyperlinkedIdentityField Always read-only It is actually a url field.</p> <p>Nested relationships Read only (except if you override create and update methods) In this case you define the whole serializer of the related model and in the serialization the whole related model serializer is returned.</p>

```
class AlbumSerializer(serializers.ModelSerializer):
    tracks = serializers.SlugRelatedField(
        many=True,
        read_only=True,
        slug_field='title'
    )
```

Writable Nested

You can save a model with a nested relationship and the related models will also be saved. But to do this you need to override the **create** and/or **update** methods.

```
class AlbumSerializer(serializers.ModelSerializer):
    tracks = TrackSerializer(many=True)

    class Meta:
        model = Album
        fields = ('album_name', 'artist', 'tracks')

    def create(self, validated_data):
        tracks_data = validated_data.pop('tracks')
        album = Album.objects.create(**validated_data)
        for track_data in tracks_data:
            Track.objects.create(album=album, **track_data)
        return album
```

Notice that we remove the tracks data from the validated data and since there is only the rest (the album data) we use it to create an album instance. Then from the tracks data we create the track objects.

This way when we save the serializer apart from the album object the related tracks objects will also be created.

```
class AlbumSerializer(serializers.ModelSerializer):
    tracks = TrackSerializer(many=True, read_only=True)

    class Meta:
        model = Album
        fields = ('album_name', 'artist', 'tracks')
```

```
{
    'album_name': 'The Grey Album',
    'artist': 'Danger Mouse',
    'tracks': [
        {'order': 1, 'title': 'Public Service Announcement', 'duration': 245},
        {'order': 2, 'title': 'What More Can I Say', 'duration': 264},
        {'order': 3, 'title': 'Encore', 'duration': 159},
        ...
    ],
}
```

Custom representation of a relation

```
class TrackListingField(serializers.RelatedField):
    def to_representation(self, value):
        duration = time.strftime('%M:%S', time.gmtime(value.duration))
        return 'Track %d: %s (%s)' % (value.order, value.name, duration)

class AlbumSerializer(serializers.ModelSerializer):
    tracks = TrackListingField(many=True)

    class Meta:
        model = Album
        fields = ('album_name', 'artist', 'tracks')
```

You can define a custom string to be returned in the serialized data, using the **RelatedField** class and the **to_representation** method

Forward relationships


```

from rest_framework import serializers
from models import City, County, State, Zip

class CitySerializer(serializers.ModelSerializer):
    class Meta:
        model = City
        fields = ('id', 'name')

class CountySerializer(serializers.ModelSerializer):
    class Meta:
        model = County
        fields = ('id', 'name')

class StateSerializer(serializers.ModelSerializer):
    class Meta:
        model = State
        fields = ('id', 'name', 'state_abbr')

class ZipSerializers(serializers.ModelSerializer):
    city = CitySerializer()
    county = CountySerializer()
    state = StateSerializer()

    class Meta:
        model = Zip
        fields = ('id', 'zipcode', 'city', 'county', 'state')

```

We use the serializer field types mentioned before.

Here he uses the nested relationship option.

Validation

Validation in Django REST framework serializers is handled a little differently to how validation works in Django's ModelForm class. With ModelForm the validation is performed partially on the form, and partially on the model instance. With REST framework the validation is performed entirely on the serializer class.

The ModelSerializer automatically generates the built in validation classes for its fields (from the fields attributes like “unique”). You can inspect the validations that will be generated by printing the repr attribute of the serializer (print(repr(serializer))). You can disable this functionality if you want. If you use the Serializer class instead, then you must implement the validation rules explicitly like in the following example.

```

from rest_framework.validators import UniqueForYearValidator

class ExampleSerializer(serializers.Serializer):
    # ...
    class Meta:
        # Blog posts should have a slug that is unique for the current year.
        validators = [
            UniqueForYearValidator(
                queryset=BlogPostItem.objects.all(),
                field='slug',
                date_field='published'
            )
        ]

```

Example of using a built in validation class “UniqueForYearValidator”.

Throttling

Throttling is similar to permissions, in that it determines if a request should be authorized.

Django rest provides application-level throttling suitable for

- different business tiers and
- basic protections against service over-use

not suitable for

- brute forcing or denial-of-service attacks

The application-level throttling that REST framework provides should not be considered a security measure or protection against brute forcing or denial-of-service attacks. Deliberately malicious actors will always be able to spoof IP origins. In addition to this, the built-in throttling implementations are implemented using Django's cache framework, and use non-atomic operations to determine the request rate, which may sometimes result in some fuzziness.

Throttles indicate a temporary state, and are used to control the rate of requests that clients can make to an API.

As with permissions, multiple throttles may be used.

Throttle options:

- Rate of requests
- Bandwidth
- Number of accessed records

You can have multiple throttles

- to impose different constraints on different user tiers.
- to impose different constraints on different parts of the API (using the `ScopedRateThrottle` built in class)
- to impose both burst throttling rates, and sustained throttling rates. For example, you might want to limit a user to a maximum of 60 requests per minute, and 1000 requests per day.

Implementation

- Django rest builds unique keys for the entity to be throttled. For example for the an unauthenticated user using his IP and for authenticated users using their user id. This key is stored in django's cache. It uses the default cache by default, but you can choose another one. If a request is throttled, then a `Retry-After` header is added to the response (if the `wait` method is implemented in the throttle class).
So in every request, your service will communicate with the caching service (one reason to not rely on application-level throttling for handling brute force attacks).
- You have to declare the default throttling classes you want to use in the settings (used by all view). Django rest offers some built in throttle classes. You can implement per view throttle using a throttle's tuple similar to permissions.
- The built-in throttle implementations are open to race conditions, so under high concurrency they may allow a few extra requests through. If your project relies on guaranteeing the number of requests during concurrent requests, you will need to implement your own throttle class.

The X-Forwarded-For HTTP header and REMOTE_ADDR WSGI variable are used to uniquely identify client IP addresses for throttling. If the X-Forwarded-For header is present then it will be used, otherwise the value of the `REMOTE_ADDR` variable from the WSGI environment will be used.

<pre> REST_FRAMEWORK = { 'DEFAULT_THROTTLE_CLASSES': ['rest_framework.throttling.AnonRateThrottle', 'rest_framework.throttling.UserRateThrottle'], 'DEFAULT_THROTTLE_RATES': { 'anon': '100/day', 'user': '1000/day' } } </pre>	<p>Here we use two classes for all the views of our api. With the default throttle rates setting, we modify the built in properties of these classes (anon for the first and user for the second one).</p> <p>The rate descriptions used in DEFAULT_THROTTLE_RATES may include second, minute, hour or day as the throttle period.</p>
---	--

Filtering

Have in mind the [haystack](#) package for searching filtering etc. It can be combined with many search backends like Elasticsearch.

Django rest framework

- You can override the `get_queryset` method to return parameter specific objects (for example user related objects only) where `user_id` is extracted from url for example.
- Generic filtering: You can use filter backends which filter the returned queryset (so they work in addition to the previous method). You use a 3rd party library called Django-filter. It allows you to easily construct complex searches and filters. You can define default filters and per view filters. Have in mind that Generic filters can also present themselves as HTML controls in the browsable API and admin API. Note that if a filter backend is configured for a view, then as well as being used to filter list views, it will also be used to filter the querysets used for returning a single object.
- Custom generic filtering: You can also provide your own generic filtering backend, or write an installable app for other developers to use. To do so override `BaseFilterBackend`, and override the `.filter_queryset(self, request, queryset, view)` method. The method should return a new, filtered queryset.

An example of equality based filtering

http://example.com/api/products?category=clothing&in_stock=True

<pre> class ProductList(generics.ListAPIView): queryset = Product.objects.all() serializer_class = ProductSerializer filter_backends = [DjangoFilterBackend] filterset_fields = ['category', 'in_stock'] </pre>	<p>If all you need is simple equality-based filtering, you can set a <code>filterset_fields</code> attribute on the view, or viewset, listing the set of fields you wish to filter against. This will automatically create a <code>FilterSet</code> class for the given fields, and will allow you to make requests such as:</p> <p>http://example.com/api/products?category=clothing&in_stock=True</p>
---	---

Searching

<http://example.com/api/users?search=russell>

The `SearchFilter` class supports simple single query parameter based searching, and is based on the Django admin's search functionality.

Ordering

<http://example.com/api/users?ordering=account.username>

The `OrderingFilter` class supports simple query parameter controlled ordering of results.

It's recommended that you explicitly specify which fields the API should allowing in the ordering filter. You can do this by setting an `ordering_fields` attribute on the view

If an ordering attribute is set on the view, this will be used as the default ordering.

Pagination

The pagination API can support either:

- Pagination links that are provided as part of the content of the response.
- Pagination links that are included in response headers, such as Content-Range or Link.

Pagination is only performed automatically if you're using the generic views or viewsets. If you're using a regular `APIView`, you'll need to call into the pagination API yourself to ensure you return a paginated response. See the source code for the `mixins.ListModelMixin` and `generics.GenericAPIView` classes for an example.

Pagination can be turned off.

By default using the pagination classes will cause HTML pagination controls to be displayed in the browsable API.

You can set up pagination defaults

<pre>REST_FRAMEWORK = { 'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.LimitOffsetPagination', 'PAGE_SIZE': 100 }</pre>	<p>The pagination style may be set globally, using the <code>DEFAULT_PAGINATION_CLASS</code> and <code>PAGE_SIZE</code> setting keys. For example, to use the built-in limit/offset pagination, you would do something like this:</p>
<pre>class BillingRecordsView(generics.ListAPIView): queryset = Billing.objects.all() serializer_class = BillingRecordsSerializer pagination_class = LargeResultsSetPagination</pre>	<p>You can also set the pagination class on an individual view by using the <code>pagination_class</code> attribute.</p>

It offers built in pagination classes. You can build your own custom ones too.

PageNumberPagination class

GET <https://api.example.org/accounts/?page=4>

<pre>HTTP 200 OK { "count": 1023, "next": "https://api.example.org/accounts/?page=5", "previous": "https://api.example.org/accounts/?page=3", "results": [...] }</pre>		
--	--	--

LimitOffsetPagination class

GET <https://api.example.org/accounts/?limit=100&offset=400>

<pre> HTTP 200 OK { "count": 1023, "next": "https://api.example.org/accounts/?limit=100&offset=500", "previous": "https://api.example.org/accounts/?limit=100&offset=300", "results": [...] }</pre>	
---	--

CursorPagination class

The cursor-based pagination presents an opaque "cursor" indicator that the client may use to page through the result set. This pagination style only presents forward and reverse controls, and does not allow the client to navigate to arbitrary positions.

We can modify the way the style of the response (for example the next and previous attributes can be children of a links attribute)

Versioning

Versioning is determined by the incoming client request, and may either be based on the request URL, or based on the request headers.

When API versioning is enabled, the request.version attribute will contain a string that corresponds to the version requested in the incoming client request. By default, versioning is not enabled, and request.version will always return None.

<pre> def get_serializer_class(self): if self.request.version == 'v1': return AccountSerializerVersion1 return AccountSerializer</pre>		How you vary the API behavior is up to you, but one example you might typically want is to switch to a different serialization style in a newer version. For example:
--	--	---

Reversing urls gets automatically the version (if url version is used)

AcceptHeaderVersioning class (recommended)

versioning based on accept headers is generally considered as best practice

<pre> GET /bookings/ HTTP/1.1 Host: example.com Accept: application/json; version=1.0</pre>		This scheme requires the client to specify the version as part of the media type in the Accept header. The version is included as a media type parameter, that supplements the main media type.
---	--	---

url veriosning

<pre> GET /v1/something/ HTTP/1.1 Host: example.com Accept: application/json</pre>		
--	--	--

Host name veriosning

```
GET /bookings/ HTTP/1.1
Host: v1.example.com
Accept: application/json
```

Query parameter versioning

Hostname based versioning can be particularly useful if you have requirements to route incoming requests to different servers based on the version, as you can configure different DNS records for different API versions.

```
GET /something/?version=0.1 HTTP/1.1
Host: example.com
Accept: application/json
```

Exceptions

REST framework's views handle various exceptions, and deal with returning appropriate error responses. This means that when an exception is raised in a view, Django rest will return an appropriate error response object. Examples of such responses for one typical not allowed exception and a validation exception on the right.

```
HTTP/1.1 405 Method Not Allowed
Content-Type: application/json
Content-Length: 42
```

```
{"detail": "Method 'DELETE' not allowed."}
```

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Content-Length: 94
```

```
{"amount": ["A valid integer is required."], "description": ["This field may not be blank."]}
```

The handled exceptions are:

- Subclasses of `APIException` raised inside REST framework (there are built in exceptions and you can implement your own too).
- Django's `Http404` exception.
- Django's `PermissionDenied` exception.

Creating your own exceptions

```
from rest_framework.exceptions import APIException

class ServiceUnavailable(APIException):
    status_code = 503
    default_detail = 'Service temporarily unavailable, try again later.'
    default_code = 'service_unavailable'
```

Generic Error Views

Django REST Framework provides two error views suitable for providing generic JSON 500 Server Error and 400 Bad Request responses. (Django's default error views provide HTML responses, which may not be appropriate for an API-only application.) you define them as handlers (see docs).

Have in mind

You can implement custom exception handling by creating a handler function that converts exceptions raised in your API views into response objects. This allows you to control the style of error responses used by your API. You do this by implementing and defining in settings your own `exception_handler` function.

Status codes

Use the built in status codes from the `status` module instead of integers. It's easier to understand. For example `status.HTTP_304_NOT_MODIFIED` instead of 304.

Testing

REST framework includes a few helper classes that extend Django's existing test framework, and improve support for making API requests.

APIRequestFactory

It extends the django's existing one `RequestFactory` class.

```
# Create a JSON POST request
factory = APIRequestFactory()
request = factory.post('/notes/', {'title': 'new idea'}, format='json')
```

Important

Note: When using `APIRequestFactory`, the object that is returned is Django's standard `HttpRequest`, and not REST framework's `Request` object, which is only generated once the view is called.

To forcibly authenticate a request, use the `force_authenticate()` method

You can also force csrf validation

APIClient

```
from django.urls import reverse
from rest_framework import status
from rest_framework.test import APITestCase
from myproject.apps.core.models import Account

class AccountTests(APITestCase):
    def test_create_account(self):
        """
        Ensure we can create a new account object.
        """
        url = reverse('account-list')
        data = {'name': 'DabApps'}
        response = self.client.post(url, data, format='json')
        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
        self.assertEqual(Account.objects.count(), 1)
        self.assertEqual(Account.objects.get().name, 'DabApps')
```

RequestsClient

REST framework also includes a client for interacting with your application using the popular Python library, requests. This may be useful if:

- You are expecting to interface with the API primarily from another Python service, and want to test the service at the same level as the client will see.
- You want to write tests in such a way that they can also be run against a staging or live environment.

API Test cases

REST framework includes the following test case classes, that mirror the existing Django's test case classes, but use APIClient instead of Django's default Client.

API Schemas

You create a schema from your API. This schema describes your API endpoints. There is a number of schema formats available like Core JSON, Open API/Swagger, HAL, and JSON Hyper-Schema.

The schema can be used for:

- Automatic Documentation creation
- Automatic client libraries creation

A client can use this schema to build their API requests including URLs, query parameters, and the request body. When an API changes, they simply have to update the schema and all clients will adapt accordingly. The client doesn't have to write a single line of code.

API schemas are a useful tool that allow for a range of use cases, including generating reference documentation, or driving dynamic client libraries that can interact with your API. Django REST Framework provides support for automatic generation of OpenAPI schemas.

REST framework provides built-in support for generating OpenAPI schemas, which can be used both for creating dynamic clients (coreAPI) and with tools that allow you to build API documentation.

JSON schema

<div>An example of a JSON schema</div> <pre>{ "type": "object", "properties": { "id": { "type": "number" }, "title": { "type": "string" }, "urlSlug": { "type": "string" }, "body": { "type": "string" } }, "required": ["id"] }</pre>	<div>And here is JSON data that matches that schema</div> <pre>{ "id": 5, "title": "JSON Hyper-Schema", "urlSlug": "json-hyper-schema", "body": "My long post about JSON Hyper-Schema..." }</pre>	<div>JSON Schema describes JSON data. It's like a database schema for JSON and can be used to validate a JSON instance before it's sent to an API.</div> <div>Once your API consumers have a JSON schema, they can use one of the many JSON Schema libraries to validate their JSON data.</div>
--	---	---

JSON hyper schema

Here's a JSON Hyper-Schema document for a blog post.

```
{
  "type": "object",
  "properties": {
    "id": {
      "type": "number"
    },
    "title": {
      "type": "string"
    },
    "urlSlug": {
      "type": "string"
    },
    "post": {
      "type": "string"
    }
  },
  "required": ["id"],
  "base": "[http://api.dashron.com/](http://api.dashron.com/)",
  "links": [{
    "rel": "self",
    "href": "posts/{id}",
    "templateRequired": ["id"]
  }]
}
```

links is an array of Link Description Objects (LDOs). An LDO contains all the information necessary to describe the actions available to your client.

Documentation

drf-spectacular

drf-spectacular is a library that generates automatically an OpenAPI 3.0 schema from your Django-rest project. You can customize the schema generation.

Your api client needs access to the internet so that it gets the swagger UI or Redoc from CDNs. Alternatively you can include these files in your service with an optional additional package, the drf-spectacular-sidecar. Usage is as follows:

```
$ pip install drf-spectacular[sidecar]
```

(another solution is to use the built-in browsable API without the need to use other documentation packages. But it isn't so rich and it doesn't show you a table of contents for your api. You have to know the endpoints). The browsable API that the REST framework provides makes it possible for your API to be entirely self-describing. The documentation for each API endpoint can be provided simply by visiting the URL in your browser.

Client generation (with Core API)

Core API is a format-independent Document Object Model for representing Web APIs. Core API can be used to interact with any API that exposes a supported Schema or Hypermedia format. It can be used to represent either Schema or Hypermedia responses, and allows you to interact with an API at the layer of an application interface, rather than a network interface. Core API currently has implementations available for Core JSON, Open API/Swagger, HAL, and JSON Hyper-Schema. There is a command line tool that you can use to interact with APIs exposing any of these formats, as well as a Python client library. The dynamic client library is always up to date with the API, and client code focuses solely on the interface being provided, rather than dealing with network details and encodings.

The command line client

```
$ pip install coreapi-cli
$ coreapi get http://api.example.org/
<Pastebin API "http://127.0.0.1:8000/">
snippets: {
  create(code, [title], [linenos], [language], [style])
  destroy(pk)
  highlight(pk)
  list([page])
  partial_update(pk, [title], [code], [linenos], [language], [style])
  retrieve(pk)
  update(pk, code, [title], [linenos], [language], [style])
}
users: {
  list([page])
  retrieve(pk)
}
```

To start inspecting and interacting with an API the schema must first be loaded from the network.

This will then load the schema, displaying the resulting Document. This Document includes all the available interactions that may be made against the API.

<pre>\$ coreapi action users list [{ "url": "http://127.0.0.1:8000/users/2/", "id": 2, "username": "aziz", "snippets": [] }, ...]</pre>	<p>To interact with the API, use the <u>action command</u>.</p> <p>By default the command line client only includes support for reading Core JSON schemas, however it includes a plugin system for installing additional codecs.</p> <p>Codecs are responsible for encoding or decoding Documents.</p>
<pre>\$ coreapi credentials add <domain> <credentials string> For example \$ coreapi credentials add api.example.org "Token 9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b"</pre>	<p>The <u>credentials command</u> is used to manage the request Authentication: header. Any credentials added are always linked to a particular domain, so as to ensure that credentials are not leaked across differing APIs.</p>

The python client

<pre>\$ pip install coreapi import coreapi client = coreapi.Client() schema = client.get('https://api.example.org/')</pre>	<p>In order to start working with an API, we first need a Client instance. Once we have a Client instance, we can fetch an API schema from the network. The object returned from this call will be a <u>Document instance</u>, which is a representation of the API schema.</p>
<pre>client = coreapi.Client() schema = client.get('https://api.example.org/') action = ['api-token-auth', 'create'] params = {"username": "example", "password": "secret"} result = client.action(schema, action, params) auth = coreapi.auth.TokenAuthentication(scheme='JWT', token=result['token']) client = coreapi.Client(auth=auth)</pre>	<p>When using TokenAuthentication you'll probably need to implement a login flow using the CoreAPI client. A suggested pattern for this would be to initially make an unauthenticated client request to an "obtain token" endpoint For example, using the "Django REST framework JWT" package</p>

The javascript client

There are two separate JavaScript resources that you need to include in your HTML pages in order to use the JavaScript client library. These are a static coreapi.js file, which contains the code for the dynamic client library, and a templated schema.js resource, which exposes your API schema.

You'll either want to include the API schema in your codebase directly, by copying it from the schema.js resource, or else load the schema asynchronously.

<pre>// Setup some globally accessible state window.client = new coreapi.Client(); window.loggedIn = false; function loginUser(username, password) { let action = ["api-token-auth", "obtain-token"]; let params = {username: "example", email: "example@example.com"}; client.action(schema, action, params).then(function(result) { // On success, instantiate an authenticated client. let auth = window.coreapi.auth.TokenAuthentication({ scheme: 'JWT', token: result['token'], }) window.client = coreapi.Client({auth: auth}); window.loggedIn = true; }).catch(function(error) { // Handle error case where eg. user provides incorrect credentials. }) }</pre>	<p>In order to interact with the API you'll need a client instance. Typically you'll also want to provide some authentication credentials when instantiating the client. You can use sessionauthentication, tokenauthentication and basicauthentication. This is an example of using tokenauthentication.</p>
<pre>let action = ["users", "list"]; client.action(schema, action).then(function(result) { // Return value is in 'result'})</pre>	<p>Then you can use the client.</p>

Tips

In a nutshell

Api view

- it ensures that the view receives an instance of the django rest framework's Request object (instead of the standard Django HttpRequest object)
- it ensures that the view returns an instance of the django rest framework's Response object (instead of a Django HttpResponse object)
- It provides behaviour such as returning 405 Method Not Allowed responses when appropriate,
- and handling any ParseError exception that occurs when accessing request.data with malformed input.

Request object

The core functionality of the Request object is the *request.data* attribute. request.data instead will handle either form data, or json data, or whatever other parsers you have configured.

Response object

takes unrendered content and uses content negotiation to determine the correct content type to return to the client.

The client controls the type of the response

<ul style="list-style-type: none">● either by using the Accept header: http http://127.0.0.1:8000/snippets/ Accept:application/json # Request JSON http http://127.0.0.1:8000/snippets/ Accept:text/html # Request HTML● Or by appending a format suffix: http http://127.0.0.1:8000/snippets.json # JSON suffix http http://127.0.0.1:8000/snippets.api # Browsable API suffix	content negotiation (reading the request's accept header or the format suffix of the request's url)
--	---

<pre>from rest_framework.urlpatterns import format_suffix_patterns def snippet_list(request, format=None): ... def snippet_detail(request, pk, format=None): ... urlpatterns = [path('snippets/', views.snippet_list), path('snippets/<int:pk>', views.snippet_detail),] urlpatterns = format_suffix_patterns(urlpatterns)</pre>	To make our views use the suffixes we need to use an extra argument named "format". Then we need to be able to extract the suffix from the url and to do so we use the format_suffix_patterns function
--	---

Because the API chooses the content type of the response based on the client request, it will, by default, return an HTML-formatted representation of the resource when that resource is requested by a web browser. This allows for the API to return a fully web-browsable HTML representation.

In a permissions.py file inside the app. Django rest has built in permissions that you can use or you can create your own custom ones. We create a method that returns true or false. Then in the view we define a permission_classes tuple. **Every permission in this tuple must return true in order for the view to be executed.**

Viewsets

A ViewSet class is only bound to a set of method handlers at the last moment, when it is instantiated into a set of views.

If you use a viewset you can use router to automatically generate the urls for it.

RetrieveUpdateDestroyAPIView and ListCreateAPIView generic classes can be replaced with the ModelViewSet

<pre>from rest_framework.generics import (ListCreateAPIView, RetrieveUpdateDestroyAPIView) from task.models import Task from api.serializers import TaskSerializer from api.permissions import IsOwnerOrReadOnly class TaskMixin(object): queryset = Task.objects.all() serializer_class = TaskSerializer permission_classes = (IsOwnerOrReadOnly,) def perform_create(self, serializer): serializer.save(owner=self.request.user)</pre>	<p>The previous can be replaced by this:</p> <pre>class TaskMixin(object): queryset = Task.objects.all() serializer_class = TaskSerializer permission_classes = (IsOwnerOrReadOnly,) def perform_create(self, serializer): serializer.save(owner=self.request.user) class TaskViewSet(TaskMixin, viewsets.ModelViewSet): pass</pre>
---	---

<pre># def pre_save(self, obj): # obj.owner = self.request.user class TaskList(TaskMixin, ListCreateAPIView): pass class TaskDetail(TaskMixin, RetrieveUpdateDestroyAPIView): pass</pre> <p># Notice the above schema, where we use a “mixin” class to add some custom functionality and then the task list and task detail views inherit from that mixin too.</p> <p>Have in mind that there are also the read only generic views</p> <pre>generics.RetrieveAPIView generics.ListAPIView</pre>	<p>Have in mind that there is also the read only version of modelviewset, the ReadOnlyModelViewSet.</p> <p>Have also in mind that there is the ViewSet class that is more generic since it doesn't need to be connected with a model.</p>
---	---

Authentication is always run at the very start of the view, before the permission and throttling checks occur, and before any other code is allowed to proceed. Note: Don't forget that authentication by itself won't allow or disallow an incoming request, it simply identifies the credentials that the request was made with.

Token authentication. There is a separate Django rest app for it. You install it and migrate. Then you can create tokens for your users

```
from django.contrib.auth.models import User
buddy = User.objects.get(username='buddy')

from rest_framework.authtoken.models import Token

token = Token.objects.create(user=buddy)
print token.key
```

If the request doesn't contain an Accept header the first defined default renderer will be used. The default renderers can be set globally in settings or individually for a specific view

Some bullets

- Serializer's main goal is to transform django objects to python native objects and vice versa.
- For every piece of data that you send to the server there must be a serializer for. The reason is that the serializer offers some very useful build in functionality like `is_valid()` and many more.
- The Serializer can take complex objects like django model instances and return python native datatypes that can be used by the response object (or manually rendered)

`serializer.data` is Python native datatypes

`JSONRenderer().render(serializer.data)` we can render the python native datatypes to json

- The serializer can also do the opposite. It can take python native datatypes and build django model instances from them.

`stream = io.BytesIO(content)`

`data = JSONParser().parse(stream)` json data is now python native datatypes

`serializer = SnippetSerializer(data=data)` `serializer.is_valid()` `serializer.save()`

- But notice: rest framework will automatically determine the content type of the response based on content negotiation and will automatically convert the request data to native datatypes in `request.data`. No need to return a `JSONResponse` and also no need to `jsonparse` the request [`data = JSONParser().parse(request)`]. We can directly do `serializer = SnippetSerializer(data=request.data)`
- Content negotiation: reading the request's accept header or the format suffix of the request's url
- You pass primitive datatypes to the response object. Return `Response(serializer.data)`

- Don't forget that authentication by itself won't allow or disallow an incoming request, it simply identifies the credentials that the request was made with. To disallow a request use the `isAuthenticated` permission in a view.
- You can restrict the entries the list view returns (or any view that inherits from `GenericAPIView`) by overriding the `get_queryset()` method
- The `DefaultRouter` class automatically creates the API root view for us

Packages

- `Djoser`
djoser library provides a set of Django Rest Framework views to handle basic actions such as registration, login, logout, password reset and account activation. If you use jwt you also need to install the `django-rest-framework-simplejwt`
- `django-rest-framework-simplejwt` (for jwt auth)
- `corsheaders` (for adding cors headers)
- `django-rest-pandas`
Serves up your Pandas dataframes via the Django REST Framework for use in client-side (i.e. d3.js) visualizations and offline analysis (e.g. Excel)

More tips

Returning URLs

Django rest docs guideline: As a rule, it's probably better practice to return absolute URLs from your Web APIs, such as `http://example.com/foobar`, rather than returning relative URLs, such as `/foobar`. This offers a lot of advantages for your clients. To do so use the functions `reverse` and `reverse_lazy`. These are same as in Django but they return a fully qualified uri.

Csrf in django rest

It's worth noting that Django's standard `RequestFactory` doesn't need to include this option, because when using regular Django the CSRF validation takes place in middleware, which is not run when testing views directly. When using REST framework, CSRF validation takes place inside the view

Access query set parameters

`http://example.com/api/purchases?username=denvercoder9`

```
def get_queryset(self):
    ...
    username = self.request.query_params.get('username')
```

access url parameters

```
re_path('^purchases/(?P<username>+)/$', PurchaseList.as_view()),
```

```
def get_queryset(self):
    ...
    username = self.kwargs['username']
```

Browsable API

It offers a base django template with some predefined blocks that you can override for customization.

CSRF protection

- Ensure that the 'safe' HTTP operations, such as GET, HEAD and OPTIONS cannot be used to alter any server-side state.

- Ensure that any 'unsafe' HTTP operations, such as POST, PUT, PATCH and DELETE, always require a valid CSRF token. In order to make AJAX requests, you need to include CSRF token in the HTTP header, as described in the Django documentation.

CORS

The best way to deal with CORS in REST framework is to add the required response headers in middleware. This ensures that CORS is supported transparently, without having to change any behavior in your views. Otto Yiu maintains the `django-cors-headers` package, which is known to work correctly with REST framework APIs.

Posting data with json or form encoding data

Basically there are three ways defined by the Content-Type request header to send the HTML data to the server

1. `application/x-www-form-urlencoded` (data becomes key, value separated by &)
2. `multipart/form-data` (the data is sent in chunks. Conventionally people use this to upload files)
3. `application/json`

- When submitting through JSON, you cannot send files. Well you can encode them to base64 encoding and send it as string but it would be larger than in binary format.

- Json data can be smaller

Content type: application/x-www-form-urlencoded partners[]=Apple&partners[]=Microsoft&partners[]=Activision	Content type: application/json { "partners": ["Apple", "Microsoft", "Activision"] }
--	--

# POST using form data http --form POST http://127.0.0.1:8000/snippets/ code="print(123)"	# POST using JSON http --json POST http://127.0.0.1:8000/snippets/ code="print(456)"
--	---

Regular django views

If we want to fully customize things (the views, the urls etc), we can move in the opposite direction to more low level refactoring (from viewsets and class based views to normal Django views) and write ourselves the views as normal Django views.

Format_suffix_patterns

<pre>urls from django.conf.urls import patterns, include, url from rest_framework.urlpatterns import format_suffix_patterns from zip import views urlpatterns = patterns('', url(r'^api/\$', views.ZipList.as_view()), url(r'^api/(?P<pk>[0-9]+)/\$', views.ZipDetail.as_view()),), format_suffix_patterns(urlpatterns))</pre>	<p>Have in mind the <code>Format_suffix_patterns</code> using it you can define a suffix in the url to define the format of the response that you want like : <code>/api/1/.json</code></p> 
--	--

`print(repr(serializer))`

it prints the representation of the serializer. Its field types. Very useful to see what is happening.

reverse url

There is a Django-rest **reverse** function for reversing fully qualified urls, like in django

```
@api_view(('GET',))
def api_root(request, format=None):
    return Response({
        'users': reverse('user-list', request=request, format=format),
        'snippets': reverse('snippet-list', request=request, format=format)
    })
```

`url(r'^$', views.api_root)`, We can add a url for the root API (not requesting any specific model)

pagination

we can paginate the list views using a setting. We could also customize the pagination style if we needed

```
REST_FRAMEWORK = {  
    'PAGE_SIZE': 10  
}
```

Custom field widgets

```
class SnippetSerializer(serializers.Serializer):  
    pk = serializers.IntegerField(read_only=True)  
    title = serializers.CharField(required=False, allow_blank=True, max_length=100)  
    code = serializers.CharField(style={'base_template': 'textarea.html'})  
    linenos = serializers.BooleanField(required=False)  
    language = serializers.ChoiceField(choices=LANGUAGE_CHOICES, default='python')  
    style = serializers.ChoiceField(choices=STYLE_CHOICES, default='friendly')
```

Notice the `style={'base_template': 'textarea.html'}` it defines how the field should appear in html. Its equivalent to using `widget=widgets.Textarea` on a Django Form class

django-rest-pandas

Have in mind the django-rest-pandas library. Serves up your Pandas dataframes via the Django REST Framework for use in client-side (i.e. d3.js) visualizations and offline analysis (e.g. Excel)

<https://godjango.com/87-saving-data-from-strava/>

see also blog post for front end pip

Schema

Usually you don't have to use these methods, django rest will get the proper content type from content negotiation. API schemas are a useful tool that allow for a range of use cases, including generating reference documentation, or driving dynamic client libraries that can interact with your API. Django REST Framework provides support for automatic generation of OpenAPI schemas.

Django channels 1.x

Django 3.0 vs Channels

<https://github.com/django/channels/issues/1416> a general discussion about this issue

Intro

The key part of Channels is introducing a standardised way to run **event-triggered pieces of code**, and a standardised way to **route messages via named channels** that hits the right balance between flexibility and simplicity. Channels is a **distributed system** that can have many workers.

Channels, in a nutshell, replaces the “guts” of Django — the request/response cycle — with messages that are sent across channels. Channels allows Django to support WebSockets in a way that’s very similar to traditional HTTP views. Channels also allow for background tasks that run on the same servers as the rest of Django. HTTP requests continue to behave the same as before, but also get routed over channels.

Channels are essentially task queues: messages get pushed onto the channel by producers, and then given to one of the consumers listening on that channel. If you’ve used channels in Go, the concept should be fairly familiar. The main difference is that Django channels work across a network and allow producers and consumers to run transparently across many dynos and/or machines. This network layer is called the channel layer. Channels is designed to use Redis as its preferred channel layer, though there is support for other types (and a first-class API for creating custom channel layers).

Channels continues to handle HTTP(S) requests just fine, but it does it in a complete different way. this isn’t too different from running Celery with Django, where you’d run a Celery worker alongside a WSGI server. Now, though, all tasks — HTTP requests, WebSockets, background tasks — get run in the worker.

[The core of the system is, unsurprisingly, a data structure called a channel. What is a channel? It is an ordered, first-in first-out queue with message expiry and at-most-once delivery to only one listener at a time. You can think of it as analogous to a task queue - messages are put onto the channel by producers, and then given to just one of the consumers listening to that channel. Inside a network, we identify channels uniquely by a name string - you can send to any named channel from any machine connected to the same channel backend. If two different machines both write to the http.request channel, they’re writing into the same channel.]

It can be used to implement websocket connections with Django. Also used for running background processes. Channels is a general-purpose utility for running background tasks. Thus, many features that used to require Celery or Python-RQ could be done using Channels instead. Channels can’t replace dedicated task queues entirely ([See the documentation for full details.](#)) but can make common background tasks much simpler. Plan is to include Channels in **Django 1.10**, which is scheduled for release in summer 2016.

Why we need a task queue for handling websockets?

Your application might have many workers (processes) located in different machines in a cluster. Each process handles a number of websockets. So if you want to broadcast to all of these websockets based on an event that is triggered in one of your processes, you need an inter process communication mechanism.

Channel: Named FIFO task queue. Every connection you open comes with channels going in both directions. Channels go from web server to django and from django to the webserver. Sockets have a websocket.receive channel that goes to django and a websocket.send channel that comes from django and goes to the process that has the specific socket open waiting for data. Those “return” channels are named individually. They have only one valid “subscriber”.

From another context but probably relevant:

Connection: A connection is a TCP connection between your application and the RabbitMQ broker.

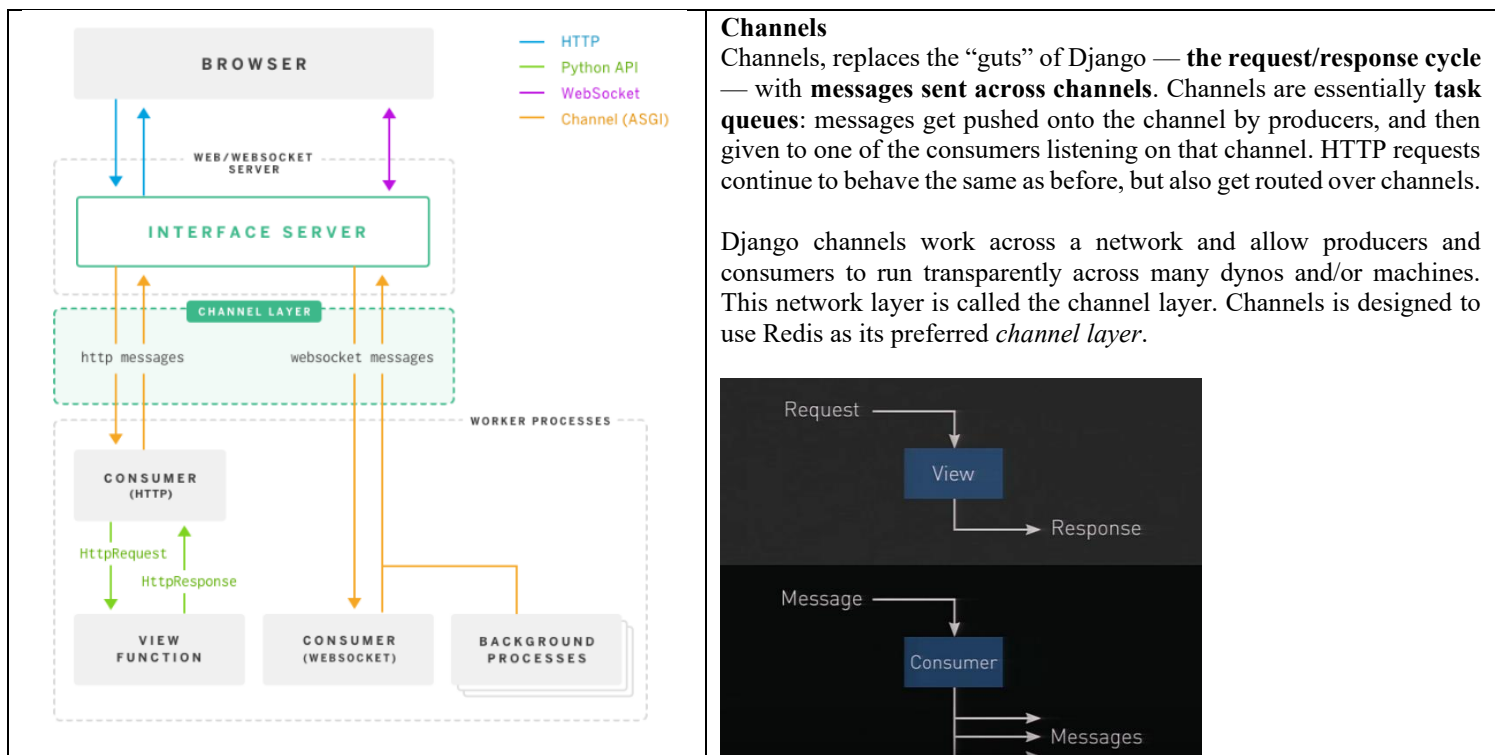
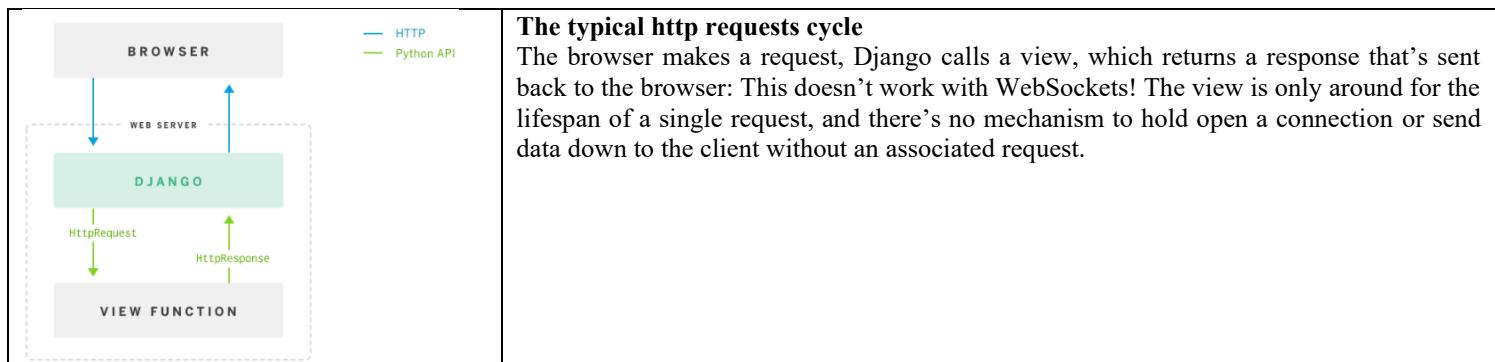
Channel: A channel is a virtual connection inside a connection. When you are publishing or consuming messages from a queue - it's all done over a channel.

Django – wsgi, django channels - asgi

Django is agnostic about its actual connection to the webserver. It provides a basic wsgi adapter (A WSGI Handler, which turns the raw request into a `HttpRequest` object, and a `HttpResponse` into a raw response, wsgi is the Python spec for delivering HTTP requests from the web server to the python application and sending responses back to the web server), which can be plugged into any wsgi container. When HTTP requests arrive at the wsgi container, it forwards them on to Django's entry point, which runs the normal routes/views/middleware/etc.

When I first got into Django in 2007 the preferred container was flup, using an scgi connection to the webserver. Nowadays, it seems like uwsgi over HTTP is preferred. CGI was an option for deployment since inception, but I don't know anyone that actually ran their Django apps over CGI, since it's many times slower than using uwsgi.

Django Channels extends wsgi into **asgi** (ASynchronous Gateway Interface), which provides both **`send()`** and **`receive_many()`** callables to the application server. **In other words, Django can now send messages, unsolicited (not asked for; given or done voluntarily), back to the webserver, which presumably has a websocket connection back to the browser.**



ASGI

(It defines a spec about how daphne communicates with the application code). ASGI aims to define a common interface for protocol servers (HTTP/WebSocket protocol servers like daphne) and application code (like Django Channels and the consumers it wraps) to communicate, with two key concepts:

- A *channel layer* API for sending and receiving *messages* between different system components
- Formats for those messages for defined protocols (e.g. HTTP, WebSocket)

channel layer

Rather than provide a server with an application object, as in WSGI (the Django code; (the application code), returns an object to the wsgi server), with ASGI you provide both servers (web dynos) and applications (worker dynos) with channel layer objects (for example redis objects). The API is pretty simple - the key mechanics are just two functions, *send()* and *receive_many()*.

Most Django users won't need to look under the hood to see what's going on. `manage.py runserver` will just launch daphne and a worker thread and tie them together with an in-memory backend.

Django will still support WSGI of course - ASGI is there for deployments that want features that fall outside of the WSGI envelope, like long-polling, WebSockets or HTTP/2 - but you will also be able to run entirely on ASGI if you want. ASGI is also designed to be able to interoperate with WSGI, running as either a WSGI application or providing a WSGI server, so it should allow Django Channels to slip in to most stacks with relative ease.

<http://www.machinalis.com/blog/introduction-to-django-channels/>

Synopsis

A default Django site follows the request-response model: A request comes in, it is routed to the proper view, the view generates a response, the response is sent to the client, and everything is done in a single process. This works just fine for most applications, but it has its limitations. If the request is complex, it might hold the worker process for a long time, making subsequent requests wait for their turn. That's why we use Celery to do things like generating thumbnails: the image upload comes in, we enqueue the thumbnail generation task and respond to the client right away, while the Celery worker takes care of the image on its own process.

Django channels provides a different model: An event-oriented model. In this model, instead of requests and response we have just events. An event with a request is received, then it is given to the proper event handler which generates a new event with the response that is going to be sent to the client.

However, the crucial part is that you can run code (and so send on channels) in response to any event - and that includes ones you create. You can trigger on model saves, on other incoming messages, or from code paths inside views and forms.

Channels splits Django into two parts: the front-end interface server (called Daphne), and the backend message consumers. Daphne replaces the WSGI server. So rather than running in just a single process tied to a WSGI server, Django runs in three separate layers

- **Interface server (daphne)**, a HTTP/WebSocket protocol server that does the job of terminating client connections and encoding requests and packets as messages on channels. So in heroku daphne (<https://github.com/andrewgodwin/daphne>) runs in a web dyno and Django.channels (<https://github.com/andrewgodwin/channels>) runs in a worker dyno. They communicate through redis (which runs elsewhere).
- **The channel backend** (mentioned also as *channel layer* in my notes) responsible for transporting messages. Notice that you can have more than one channel layers
- **The workers**, which run websocket handlers, views and background tasks when messages arrive on routed channels. You can run as many as you want (depending on your infrastructure)

Interface server and the workers communicate through ASGI protocol which is similar to WSGI but runs over a network and allows for more protocol types. The http/websocket server communicates with the Django (application) code using a “middleware” layer, called channel layer which by default is redis (or in memory space for single server situations like development). In heroku the http/websocket server runs in a web dyno and your Django code (dhancho channels) runs in a worker dyno. In a production scenario, you’d usually run worker servers as a separate cluster from the interface servers, though of course you can run both as separate processes on one machine too.

You can’t wait around inside a consumer and block on the results. You cannot wait on an event. The consumer is only executed once an event is available. You can send tasks off and keep going, but you cannot ever block waiting on a channel in a consumer

One thing channels do not, however, is guarantee delivery. If you need certainty that tasks will complete, use a system designed for this with retries and persistence (e.g. Celery) , or alternatively make a management command that checks for completion and re-submits a message to the channel if nothing is completed (rolling your own retry logic, essentially).

A **channel** is a mailbox where messages can be sent to. Each channel has a name. Anyone who has the name of a channel can send a message to the channel.

A **group** is a group of related channels. A group has a name. Anyone who has the name of a group can add/remove a channel to the group by name and send a message to all channels in the group. It is not possible to enumerate what channels are in a particular group.

All code runs synchronously

Channels does not introduce asyncio, gevent, or any other async code to your Django code; **all of your business logic runs synchronously in a worker process or thread.**

Do I need to worry about making all my code async-friendly?

No, all your code runs synchronously without any sockets or event loops to block. You can use async code within a Django view or channel consumer if you like - for example, to fetch lots of URLs in parallel - but it doesn’t affect the overall deployed site.

Channels changes the way Django runs to be “event oriented” - rather than just responding to requests, instead Django responds to a wide array of events sent on channels. There’s still no persistent state - each event handler, or consumer as we call them, is called independently in a way much like a view is called.

Install and run

When you run Django out of the box after you have installed channels, it will be set up in the default layout - where all HTTP requests (on the http.request channel) are routed to the Django view layer - nothing will be different to how things worked in the past with a WSGI-based Django, and your views and static file serving (from runserver will work as normal). **If you don’t specify a consumer for http.request the normal Django view layer to serve HTTP requests.**

http request handling

So an http request arrives at daphne. Daphne transform connections (HTTP, WebSockets, etc.) into messages on channels (adds to the message some additional attributes and methods as I understand) and puts it in a queue (called channel) meaning that it puts it to a redis list as a task. The Django code (the consumer) listens to this redis list and gets this message from the queue. It processes it, produces a response and puts it in a new queue (a new channel) in redis. Daphne gets this response message from the queue (it knows in which request it responds to) and sends it to the client with http.

websocket message handling

The process is similar with a websocket message. The difference is that when daphne gets a message from Django code, it sends it to the proper websocket connections that maintains with the clients. For example in the chat case. There are 5 websocket connections to daphne. 3 are in one Group (in one websocket url), and the other 2 in another Group. One user sends a message, daphne receives it, puts in a queue, consumer that listens to the queue process it and creates a response that has to be send to this room (this websocket url). It adds the response message to a new queue in which daphne listens to, daphne gets it and sends it to that websocket url (which 3 users use). So it sends it to 3 websocket connections. You can also send unsolicited messages from Django code to any group.

Channels as deployment pattern

Actually Channels is also a deployment pattern. You have to use a web dyno and a worker dyno. Your web server is daphne and your Django code runs as a worker. So in normal Django if you wanted to run a background process (and didn't want to make everything else wait for it to finish) you had to do it with code outside of Django and create a new worker dyno or use some multithreaded / multiprocessed deployment scheme (gunicorn etc) and create a new worker thread/process. With channels this is embedded functionality. You can run background tasks in the same server that Django is installed.

Tips

Scope and Events

Channels and ASGI split up incoming connections into two components: a scope, and a series of events. The scope is a set of details about a single incoming connection.

- For HTTP, the scope just lasts a single request. For WebSocket, it lasts for the lifetime of the socket (but changes if the socket closes and reconnects).
- During the lifetime of this scope, a series of events occur. These represent user interactions - making a HTTP request, for example, or sending a WebSocket frame.
- Within the lifetime of a scope, you will have one application instance handling all the events from it. You can persist things onto the application instance as well (apart from or on top of the things that are stored in the scope of the connection that this application instance handles). A consumer represents an application instance. Consumers live for the duration of a scope.
- Every event is of a specific type. These types are represented by consumer methods. For example the connect event is handled by the connect method of a consumer.

Synchronous and Asynchronous Consumers

Underneath, Channels is running on a fully asynchronous event loop. So your asgi applications run in a thread inside a server process. This thread creates an event loop.

- If consumer code is synchronous (defs instead of async defs and sync instead of async consumer class) then each connection creates a new thread (and probably a new event loop that isn't used?). If it is asynchronous though, everything runs concurrently in one thread using the event loop. Each request creates a new application instance that runs in the same thread.
- If you have blocking operations in your asynchronous consumers then you either block the event loop or you can run it in a threadpool. If you have blocking operations in a sync consumer then its thread is blocked but other consumers can run in other threads (although you can efficiently use only a limited number of threads).

Deployment

It is good practice to use a common path prefix like /ws/ to distinguish WebSocket connections from ordinary HTTP connections because it will make deploying Channels to a production environment in certain configurations easier.

In particular for large sites it will be possible to configure a production-grade HTTP server like nginx to route requests based on path to either (1) a production-grade WSGI server like Gunicorn+Django for ordinary HTTP requests or (2) a production-grade ASGI server like Daphne+Channels for WebSocket requests.

Note that for smaller sites you can use a simpler deployment strategy where Daphne serves all requests - HTTP and WebSocket - rather than having a separate WSGI server. In this deployment configuration no common path prefix like `/ws/` is necessary.

Each socket or connection to your application is handled by an application instance. (A consumer is an ASGI application instance)

Each application instance has a unique channel name (and can join groups)

asynchronous consumers

If you have an asynchronous consumer then every new request that is routed to that consumer on that machine, is handled by one thread and the event loop, without having to create a new thread for each request. This leads to a higher level of performance. (Many application instances are handled by one thread)

So if you have many workers then each one has its own event loop right? Probably Yes.

heroku deployment differences with regular Django app

- Because Channels apps need both a HTTP/WebSocket server and a backend channel consumer, the Procfile needs to define both of these types. And, when we do our initial deploy, we'll need to make sure both process types are running (Heroku will only start the web dyno by default)
- Apart from heroku-postgresql you need also heroku-redis (the default channel layer)
- The key is that Channels splits processes between those responsible for **handling connections** (daphne), and those responsible for **handling channel messages** (runworker). This means that there are 2 concerns in scaling:
The throughput of channels — HTTP requests, WebSocket messages, or custom channel messages — is mostly determined by the number of worker dynos. So, if you need to handle greater request volume, you'll do it by scaling up worker dynos (e.g. heroku ps:scale worker=3).
The level of concurrency — the number of current open connections — will mostly be limited by the scale of the front-end web dyno. So, if you needed to handle more concurrent WebSocket connections, you'd scale up the web dyno (e.g. heroku ps:scale worker=2). Daphne is quite capable of handling many hundreds of concurrent connections on a Standard-1X dyno

Vs gevent, asyncio etc

In channels all the code you write for consumers runs synchronously. You can do all the blocking filesystem calls and CPU-bound tasks you like and all you'll do is block the one worker you're running on; the other worker processes will just keep on going and handling other messages.

Channels still uses asynchronous code, but it confines it to the **interface layer** - the processes that serve HTTP, WebSocket and other requests. These do indeed use asynchronous frameworks (currently, asyncio and Twisted) to handle managing all the concurrent connections, but they're also fixed pieces of code; as an end developer, you'll likely never have to touch them.

Does channels replace Celery+RabbitMQ/Queue service?

Channels is a general-purpose utility for running background tasks. Thus, many features that used to require Celery or Python-RQ could be done using Channels instead. Channels can't replace dedicated task queues entirely: it has some important limitations, including at-most-once delivery (delivery is not guaranteed), that don't make it suitable for all use cases. Still, Channels can make common background tasks much simpler. For example, you could easily use Channels to perform image thumbnailing, send out emails, tweets, or SMSes, run expensive data calculations, and more." However, per my conversations with Andrew on twitter, he doesn't recommend using channels as a celery/rq replacement as delivery is not guaranteed. If you are going to use it as one, **you need to at least add your own retry mechanism.**

If you've worked with Django before, you already know how important this project is. Supporting the features mentioned with Django's current implementation requires libraries like [Celery](#) (to handle complex tasks outside the realm of a request), or [Node.js](#), [django-websocket-redis](#), or [gevent-socketio](#) to support WebSockets. With the exception of Celery (which is a defacto standard) all the other implementations are non-standard way of working around the limitations of Django with problems of their own.

I did some Django years ago and if I remember correctly Django wasn't a "server" ie long running process but more like CGI. So how does it work exactly ? is there some python code running as a demon polling socket events and interacting with the actual Django app ?

The underlying model for all Python Web apps -- WSGI -- is CGI-like. But the actual deployment of WSGI almost always involves a long-running process or processes, which keep the application code in memory and then expose the appropriate WSGI API to whatever Web server you're using. Switching that from "expose WSGI API" to "expose ASGI API" gets you Channels :)

Channels event loop

Django channels provides a different model: An event-oriented model. In this model, instead of requests and response we have just events. An event with a request is received, then it is given to the proper event handler which generates a new event with the response that is going to be sent to the client. These probably happen in the daphne part, I mean the event loop. But the event model can be applied to other scenarios and not just mimicking the request-response model. For instance, suppose that a hardware sensor is triggered due to external condition, this generates an event which is given to the event handler, which in turn generates another event to notify whomever is interested in the occurrence of the original event.

When do I have to start thinking about sharding?

Load tests showed good response times for 10 requests at once. If the sockets are not doing anything the number goes up dramatically.

One machine should normally be able to handle a couple hundred sockets

Channel

The core of the system is, unsurprisingly, a datastructure called a channel. It's an ordered, first-in first-out queue with message expiry and at-most-once delivery to only one listener at a time. Several producers write messages into a channel (which is identified by a name), and when a consumer that was subscribed to that channel becomes available, it picks the first message that came into the queue.

At least once: The call executes at least once as long as the server machine does not fail. These semantics require very little overhead and are easy to implement. The client machine continues to send call requests to the server machine until it gets an acknowledgement. If one or more acknowledgements are lost, the server may execute the call multiple times. This approach works only if the requested operation is idempotent, that is, multiple invocations of it return the same result. Servers that implement only idempotent operations must be stateless, that is, must not change global state in response to client requests. Thus, RPC (remote procedure call) systems that support these semantics rely on the design of stateless servers. an example case of "at-least-once" would be something like updating a database with a particular value (if you happen to write the same value to the database twice in a row, that really isn't going to have any effect on anything)

At most once: By at-most-once we say that either one consumer gets the message or nobody does (if the channel implementation crashes, let's say). The call executes at most once - either it does not execute at all or it executes exactly once depending on whether the server machine goes down. Unlike the previous semantics, these semantics require the detection of duplicate packets, but work for non-idempotent operations. An example case where you want to do "at-most-once" would be something like payments (you wouldn't want to accidentally bill someone's credit card twice),

The channels have capacity, so a load of producers can write lots of messages into a channel with no consumers and then a consumer can come along later and will start getting served those queued messages.

2 Types of channels

There are two types of channels: the **regular channels** used to get messages to consumers (we use these channels for routing. We connect them with specific Django functions), and **response channels** (`message.reply_channel`) which are unique per client and their name contains a “!” (for example `http.response!f5G3fE21f`). The ! character signifies a *single-reader channel*. Single-reader channels are only ever connected to by a single process). Only the interface server is listening on the response channels, and it knows which channel is connected to which connection (client) so it know who to send the responses to. When you have scaled up and have many servers and workers then there is a difference in the handling of the channel types. Any worker can take a message from regular channels. But a response channel must have its messages sent to the interface server that listens to it (and knows in which client to send it).

Regular channels:

- **websocket.connect** channel
A message is sent to this channel the first time a new client (i.e. browser) connects via a WebSocket.
- **websocket.receive** channel
A message is sent to this channel when a websocket packet is received from a client (These are just messages received from the browser; remember that channels are one-direction)
- **websocket.disconnect** channel
A message is sent to this channel when the client disconnects
- **http.message** channel
A message is sent to this channel when an http request arrive

Add a message to a channel: What does this do exactly?

What I guess: Each channel represents a separate event loop. When we add a message to a channel we actually add an event in the event loop (which runs in the interface server in case of regular channels and in the worker in case of response channels). The events of the event loop are represented in a datastore (for example in redis list). A consumer listens to this list. When there is an event in the list it gets it. The event is the message datatype or it contains the data to recreate the message datatype from them. The channel is the datatype that “describes” this process.

Message

messages must be made of serializable types (must be json serializable). message objects have a “**content**” attribute which is always a dict of data, and a “**channel**” attribute which is the channel it came from, as well as **some others**

Send http message

```
# Listens on http.request
def my_consumer(message):
    #Decode the request from message format to a Request object
    django_request = AsgiRequest(message)
    # Run a normal django view
    django_response = view(django_request)
    # Encode the response into message format
    for chunk in AsgiHandler.encode_response(django_response):
        message.reply_channel.send(chunk)
```

Send to Group

```
Group('chat-'+label).send({'text': json.dumps(m.as_dict())})
```

Usually you leave normal HTTP up to Django’s built-in consumers that plug it into the view/template system, but you can override it to add functionality if you want.

`message.content['get']` gets the query string parameters of the url. The get dictionary. `{"foo": ["bar"]}`

`message.content['path']` gets the path of the message. For example if this message represents an http request <http://host/some/path?foo=bar> this will be `/some/path`

`message.reply_channel` it is the reply channel automatically created for that specific message. This channel is in the form `http.response.o4F2h2Fd` where the last number is a user specific number. There is this channel and we can add messages to it without knowing the number. We just use the `message.reply_channel` to get the channel.

`message.reply_channel.send()`

--	--

HttpResponse.channel_encode() encode a text in the proper encoding in order to be send to a channel

Messages sent to channels must be json serializable (meaning that must be serialized to key value pairs). Messages are in key value format (as described by the ASGI spec). channels can transform from ASGI request to Django request object (using the AsgiRequest class). This is done for messages sent from interface server to the worker. And can translate httpresponse objects to ASGI messages (to be sent from the workers to the interface server).

Groups

Named sets of channels you broadcast to

It is used for broadcasting messages. You send a message to the group and it broadcasts it to all of its channels.

This is how you could do the same thing if channels didn't offer the Group abstraction. You actually add each reply channel name in a redis set. Then you create a django post save signal on a model and then iterate through the redis set, get the Channel for each channel name and send a message to it.

```
redis_conn = redis.Redis("localhost", 6379)

@receiver(post_save, sender=BlogUpdate)
def send_update(sender, instance, **kwargs):
    # Loop through all response channels and send the update
    for reply_channel in redis_conn.smembers("readers"):
        Channel(reply_channel).send(
            id=instance.id,
            content=instance.content,
        )

# Connected to websocket.connect
def ws_connect(message):
    # Add to reader set
    redis_conn.sadd("readers", message.reply_channel.name)
```

But there are some issues with the manual approach. You have on disconnect to remove channels and also in case of interface server failure that didn't send disconnect messages you must check the time of each message and remove it. All this can be done automatically by the Group. (In the background it creates a redis set and do all these things)

```
@receiver(post_save, sender=BlogUpdate)
def send_update(sender, instance, **kwargs):
    Group("liveblog").send(
        id=instance.id,
        content=instance.content,
    )

# Connected to websocket.connect
def ws_connect(message):
    # Add to reader group
    Group("liveblog").add(message.reply_channel)

# Connected to websocket.disconnect
def ws_disconnect(message):
    # Remove from reader group on clean disconnect
    Group("liveblog").discard(message.reply_channel)
```

Authentication

Channels is network-transparent and can run on multiple workers, so you can't just store things locally in global variables or similar. If something is stored locally on one worker (for example the information that a user is logged in) then if the next request of this user is handled by another worker, then the user will appear to be logged out. So we must persist data for each unique reply channel (each unique websocket connection). This is what Django's session framework does for HTTP requests, using a cookie as the key. Wouldn't it be useful if we could get a **session** using the **reply_channel as a key**?

Handily, as WebSockets start off using the HTTP protocol, they have a lot of familiar features, including a **path**, **GET parameters**, and **cookies**

We do our authentication inside our consumer functions

@channel_session

You get access to *message.channel_session*

@http_session decorator

You get access to a user's normal Django session using the `http_session` decorator - that gives you a *message.http_session* attribute that behaves just like `request.session`.

@http_session_user

You can go one further and use `http_session_user` which will provide a *message.user* attribute as well as the session attribute

Notice that you only get the detailed HTTP information (I guess http cookies, meaning http session, meaning user authentication) during the connect message of a WebSocket connection. This means lower bandwidth for the other messages but also means **we'll have to grab the user in the connection handler and then store it in the session.**

@ channel_session_user

loads the user from the channel session rather than the HTTP session.

transfer_user

And you get a function called *transfer_user* which replicates a user from one session to another.

@channel_session_user_from_http

It combines all of these, meaning that gets the user from the http session, stores it in a channel_session like `message.channel_session['user'] = user`. Then in websocket receive consumers we can get the user from the channel_session. But we can skip the long typing of `user = message.channel_session['user']` and use directly `message.user` if we use the `@channel_session_user` which loads the user from the channel_session. But we do this at once with the `@channel_session_user_from_http` which loads the user from http session, stores it in the channel session and retrieves it from there. So if you use it in the connect consumer you can then retrieve the user from channel session in the other consumers

<pre># In consumers.py from channels import Channel, Group from channels.sessions import channel_session from channels.auth import http_session_user, channel_session_user, channel_session_user_from_http # Connected to websocket.connect @channel_session_user_from_http def ws_add(message): # Add them to the right group Group("chat-%s" % message.user.username[0]).add(message.reply_channel) # Connected to websocket.receive @channel_session_user def ws_message(message): Group("chat-%s" % message.user.username[0]).send({ "text": message['text'], }) # Connected to websocket.disconnect @channel_session_user def ws_disconnect(message): Group("chat-%s" % message.user.username[0]).discard(message.reply_channel)</pre>	This is how it's used:
--	------------------------

Have in mind: Channels can use Django sessions either from cookies (if you're running your websocket server on the same port as your main site, using something like Daphne), or from a `session_key` GET parameter (passing the session id as a get parameter since the message doesn't contain any http information like the cookie which identifies the session), which works if you want to keep running your HTTP requests through a WSGI server and offload WebSockets to a second server process on another port.

Routing

When Django accepts an HTTP request, it consults the root URLconf to lookup a view function, and then calls the view function to handle the request. Similarly, when Channels accepts a WebSocket connection, it consults the root routing configuration to lookup a consumer, and then calls various functions on the consumer to handle events from the connection.

<pre>http_routing = [route("http.request", poll_consumer, path=r"^/poll/\$", method=r"^POST\$"),] chat_routing = [route("websocket.connect", chat_connect, path=r"^(?P<room>[a-zA-Z0-9_]+)/\$"), route("websocket.disconnect", chat_disconnect),] routing = [# You can use a string import path as the first argument as well. include(chat_routing, path=r"^/chat/"), include(http_routing),]</pre>	<p>Much like urls, you route using regular expressions; the main difference is that because the path is not special-cased - Channels doesn't know that it's a URL - you have to start patterns with the root /, and end includes without a / so that when the patterns combine, they work correctly.</p> <p>The room parameter is passed to the consumer as in urls matched parameters pass to views.</p> <pre># Connected to websocket.connect @channel_session_user_from_http def ws_add(message, room): # Add them to the right group Group("chat-%s" % room).add(message.reply_channel)</pre>
--	---

Custom channels

<pre># In consumers.py from channels import Channel from channels.sessions import channel_session from .models import ChatMessage # Connected to chat-messages def msg_consumer(message): # Save to model ChatMessage.objects.create(room=message.content['room'], message=message.content['message'],) # Broadcast to listening sockets Group("chat-%s" % room).send({ "text": message.content['message'], }) # Connected to websocket.connect @channel_session def ws_connect(message): # Work out room name from path (ignore slashes) room = message.content['path'].strip("/") # Save room in session and add us to the group message.channel_session['room'] = room Group("chat-%s" % room).add(message.reply_channel) # Connected to websocket.receive @channel_session def ws_message(message): # Stick the message onto the processing queue Channel("chat-messages").send({ "room": channel_session['room'], "message": message['text'], }) # Connected to websocket.disconnect @channel_session def ws_disconnect(message): Group("chat-%s" % message.channel_session['room']).discard(message.reply_channel)</pre> <p><i>consumer that listens to the channel we created (set up in routing)</i></p> <p><i>create a Channel named chat-messages</i></p>	<p>We create a channel named "chat-messages" and we add messages to it on message receive. We set up routing so the msg_consumer function listens to that channel.</p> <p>This way the sending process/consumer (he means the receive consumer right?) can move on immediately and not spend time waiting for the database save and the (slow on some backends) Group.send() call.</p> <p>Note that we could add messages onto the chat-messages channel from anywhere; inside a View, inside another model's post_save signal, inside a management command run via cron. If we wanted to write a bot, too, we could put its listening logic inside the chat-messages consumer, as every message would pass through it.</p>
---	--

Enforce Ordering

Because Channels is a distributed system that can have many workers, by default it just processes messages in the order the workers get them off the queue. It's entirely feasible for a WebSocket interface server to send out a connect and a receive message close enough together that a second worker will pick up and start processing the receive message before the first worker has finished processing the connect worker.

@enforce_ordering

All WebSocket messages contain an order key, and this decorator uses that to make sure that messages are consumed in the right order, in one of two modes:

- **Slight ordering:** Message 0 (websocket.connect) is done first, all others are unordered (smaller cost than strict mode)
- **Strict ordering:** All messages are consumed strictly in sequence

The decorator uses `channel_session` to keep track of what numbered messages have been processed

Generally you'll want to use slight mode for most session-based WebSocket and other "continuous protocol" things. Slight ordering does mean that it's possible for a disconnect message to get processed before a receive message, but that's fine in this case; the client is disconnecting anyway, they don't care about those pending messages.

Deployment

Architecture

In particular for large sites it will be possible to configure a production-grade HTTP server like nginx to route requests based on path to either (1) a production-grade WSGI server like Gunicorn+Django for ordinary HTTP requests or (2) a production-grade ASGI server like Daphne+Channels for WebSocket requests.

Note that for smaller sites you can use a simpler deployment strategy where Daphne serves all requests - HTTP and WebSocket - rather than having a separate WSGI server.

If you leave a project with the default settings (no `CHANNEL_LAYERS`), it'll just run and work like a normal WSGI app. When you want to enable channels in production, you need to do three things:

- Set up a channel backend
- Run worker servers
- Run interface servers

You can have a wsgi server (uwsgi/gunicorn etc) and a separate asgi server (in this case you need a separate interface server in front of them to route requests appropriately). Or you can have one that serves all traffic both http and websockets, http2 like Daphne.

If you use Daphne for all traffic, it auto-negotiates between HTTP and WebSocket, so there's no need to have your WebSockets on a separate port or path (and they'll be able to share cookies with your normal view code, which isn't possible if you separate by domain rather than path).

Run a channel backend

We recommend you always have at least ten workers for each Redis server to ensure good distribution.

The IPC backend, works between processes on the same machine but not over the network. The IPC backend uses POSIX shared memory segments and semaphores in order to allow different processes on the same machine to communicate with each other.

Run worker servers.

You can run many workers (with `manage.py runworker`) in one machine. (They will all be under the same process but each one will have its own thread as I understood). Each worker server is single-threaded so it's recommended you run around one or two per core on each machine. It's safe to run as many concurrent workers on the same machine as you like, as they don't open any ports (all they do is talk to the channel backend). They are executed concurrently (in each core), not co-operatively like coroutines; they are "paused" and continue from the os.

In a more complex project, you won't want all your channels being served by the same workers, especially if you have long-running tasks. It's possible to tell workers to either limit themselves to just certain channel names or ignore specific channels using the `--only-channels` and `--exclude-channels` options.

Run interface servers

To run Daphne, it just needs to be supplied with a channel backend, in much the same way a WSGI server needs to be given an application.

Note: Make sure you run the commands (*manage.py runworker* and *daphne my_project.asgi:channel_layer*) inside an init system or a program like supervisord that can take care of restarting the process when it exits;

Heroku Real time chat app

There are a bunch of rooms, and everyone in the same room can chat, in real-time, with each other (using WebSockets).

Models and views

<pre>class Room(models.Model): name = models.TextField() label = models.SlugField(unique=True) class Message(models.Model): room = models.ForeignKey(Room, related_name='messages') handle = models.TextField() message = models.TextField() timestamp = models.DateTimeField(default=timezone.now, db_index=True)</pre>	<pre>def chat_room(request, label): # If the room with the given label doesn't exist, automatically create it # upon first visit (a la etherpad). room, created = Room.objects.get_or_create(label=label) # We want to show the last 50 messages, ordered most-recent-last messages = reversed(room.messages.order_by('-timestamp')[:50]) return render(request, "chat/room.html", { 'room': room, 'messages': messages, })</pre>
---	---

Opening websocket in js

Like HTTP and HTTPS, the WebSocket protocol comes in secure (WSS) and insecure (WS) flavors.

```
// W$(function() {
```

When we're using HTTPS, use WSS too.

```
var ws_scheme = window.location.protocol == "https:" ? "wss" : "ws";
```

```
var chatsock = new ReconnectingWebSocket(ws_scheme + '://' +
window.location.host + "/chat" + window.location.pathname);
```

```
chatsock.onmessage = function(message) {
```

```
var data = JSON.parse(message.data);
```

```
var chat = $("#chat")
```

```
var ele = $('<tr></tr>')
```

```
ele.append( $('<td></td>').text(data.timestamp) )
```

```
ele.append( $('<td></td>').text(data.handle) )
```

```
ele.append( $('<td></td>').text(data.message) )
```

```
chat.append(ele)
```

```
};
```

new ReconnectingWebSocket

Because Heroku's router has a 60-second timeout, I use a little shim around the browser WebSocket that automatically reconnects if the socket gets disconnected.

We wire up a callback so that when the form's submitted, we send data over the WebSocket (rather than POSTing it)

chatsock.onmessage

We need to wire up a callback to fire when new data is received on the WebSocket. (the timestamp is added by the server)

If I run this code now, it won't work — there's nothing listening to WebSocket connections, just HTTP.

<pre> \$("#chatform").on("submit", function(event) { var message = { handle: \$('#handle').val(), message: \$('#message').val(), } chatsock.send(JSON.stringify(message)); \$('#message').val('').focus(); return false; }); }); </pre>	
--	--

Channelify the app – Set up

To “channel-ify” this app, we’ll need to do three things: install Channels, set up a channel layer, define channel routing, and modify our project to run under Channels (rather than WSGI).

1. Install channels

<pre> pip install channels add "channels" to your INSTALLED_APPS </pre>	<p>Installing Channels allows Django to run in “channel mode”, swapping out the request/response cycle with the channel-based architecture described above. (For backwards-compatibility, you can still run Django in WSGI mode, but WebSockets and all the other Channel features won’t work in this backwards-compatible mode.)</p>
---	---

2. Choose channel layer

<pre> pip install asgi_redis </pre> <p>Then we define our <i>channel layer</i> in the CHANNEL_LAYERS setting</p> <pre> CHANNEL_LAYERS = { "default": { "BACKEND": "asgi_redis.RedisChannelLayer", "CONFIG": { "hosts": [os.environ.get('REDIS_URL', 'redis://localhost:6379')], }, "ROUTING": "chat.routing.channel_routing", }, } </pre>	<p>This is the transport mechanism that Channels uses to pass messages from producers (message senders, which is daphne- the HTTP/WebSocket protocol server) to consumers (consumers code and view code).</p> <p>It’s a type of message queue with some specific properties. Redis is the default one but you can also use in-memory and database-backed channels layers for development or low traffic sites. The Redis channel layer is implemented in a different package (its not built-in in Channels), we’ll need to install it: <code>pip install asgi_redis</code>.</p>
---	---

3. Channel routing

<p>routing.py</p> <pre> channel_routing = { #we will do this later } </pre>	<p>In CHANNEL_LAYERS settings, we’ve told Channel where to look for our channel routing — <code>chat.routing.channel_routing</code>. Channel routing is a very similar concept to URL routing: URL routing maps URLs to view functions; channel routing maps channels to consumer functions. Notice that all websockets no matter what their “url” is will get routed and matched by our routing</p>
---	--

Our app has both a `urls.py` and a `routing.py`: we’re using the same app to handle HTTP requests and WebSockets. This is expected, and typical: Channels apps are still Django apps, so all the features you except from Django — views, forms, models, etc. — continue to work as they did pre-Channels.

4. Run Django with channels

```
asgi.py
import os
import channels.asgi

os.environ.setdefault("DJANGO_SETTINGS_MODULE", "chat.settings")
channel_layer = channels.asgi.get_channel_layer()
```

```
$ daphne chat.asgi:channel_layer --port 8888
```

```
$ python manage.py runworker //run a worker
```

Finally, we need to swap out Django's HTTP/WSGI-based request handler, for the one built into channels. This is based around an emerging standard called **ASGI (Asynchronous Server Gateway Interface)**, so we'll define that handler in an asgi.py file.

Daphne

Run the app under channels. The Channels interface server is called Daphne.

**** If we now visit `http://localhost:8888/` we should see.... nothing happen. This might be confusing, until you remember that Channels splits Django into two parts:**

- the **front-end interface server**, Daphne
- the **backend message consumers**.

So to actually handle HTTP requests, we need to run a worker

Now, requests should go through. This illustrates something pretty neat: Channels continues to handle HTTP(S) requests just fine, but it does it in a complete different way. This isn't too different from running Celery with Django, where you'd run a Celery worker alongside a WSGI server. Now, though, all tasks — **HTTP requests, WebSockets, background tasks** — **get run in the worker**. By the way, we can still run `python manage.py runserver` for easy local testing. When we do, Channels simply runs Daphne and a worker in the same process as separate threads. The channel layer can be in-memory storage.

Websocket consumers

What we actually do here is to create 3 channels (essentially 3 task queues) and we subscribe consumers to each channel. So whenever there is a message in a websocket the message is added to the receive channel and whenever a consumer is available will get the message from the channel and process it.

Channels maps WebSocket connections to three channels:

1. A message is sent to the **websocket.connect** channel the first time a new client (i.e. browser) connects via a WebSocket.
2. Each message the client sends on the socket gets sent across the **websocket.receive** channel. (These are just messages received from the browser; remember that channels are one-direction. We'll see how to send messages back to the client in a bit.)
3. Finally, when the client disconnects, a message is sent to **websocket.disconnect**

```
So in routing.py
from . import consumers

channel_routing = {
    'websocket.connect': consumers.ws_connect,
    'websocket.receive': consumers.ws_receive,
    'websocket.disconnect': consumers.ws_disconnect,
}
```

We just connect each channel to an appropriate function. By convention we'll put these functions in a *consumers.py*

Notice: Each websocket connection has a url like *ws://host/chat/label*. Each room has its own specific label (which is mapped to model's label attribute). But in routing all websocket urls are routed to 3 functions. So we must extract the specific label in the functions code. On the contrary using `urls.py` to route http, `urls.py` do this for us. Eventually websocket routing will match urls routing in the future. Seems it has been done. See Routing chapter. It uses regular expressions and includes similar to `urls`.

```
consumers.py
from channels import Group
from channels.sessions import channel_session
from .models import Room

@channel_session
def ws_connect(message):
    prefix, label = message['path'].strip('/').split('/')
    room = Room.objects.get(label=label)
    Group('chat-' + label).add(message.reply_channel)
    message.channel_session['room'] = room.label
```

For clarity, I've stripped all the error handling and logging from this code.

message.reply_channel

Channels adds to each message a `reply_channel` attribute which is the channel that responds to this specific message. Channels creates this channel for us. **Notice that a channel is one direction so we need to create a new channel to respond to each message.**

Group

We create a Group (named `chat+'room label'`) and add all response channels of the messages that have been send to that room.

<pre>@channel_session def ws_receive(message): label = message.channel_session['room'] room = Room.objects.get(label=label) data = json.loads(message['text']) m = room.messages.create(handle=data['handle'], message=data['message']) Group('chat-'+label).send({'text': json.dumps(m.as_dict())}) @channel_session def ws_disconnect(message): label = message.channel_session['room'] Group('chat-'+label).discard(message.reply_channel)</pre>	<p>Session</p> <p>Notice that subsequent messages (receive/disconnect) won't contain the URL anymore (since the connection is already active). So there is no way to extract the label of subsequent messages, so we can't identify in which room it belongs. We overcome this by adding a session variable to each message using the message.channel_session.</p>
<p>Channel sessions are very similar to Django's session framework. Adding the <code>@channel_session</code> decorator to a consumer is all we need to make sessions work. (The documentation has more details on how channel sessions work). In receive and disconnect functions we extract the label from the session.</p> <p>In receive, we read the message text we save it to the db, and then using the Group.send(message text) we send this message to all the reply channels attached to this Group.</p>	

Misc

Channel 2.0

The channel layer (redis or other) can't scale to huge amounts of traffic but it can be ok up to let's say 100 requests per second (why not?). you can replace it with something else after that

Synchronous vs asynchronous consumers

The ChatConsumer that we have written is currently synchronous. Synchronous consumers are convenient because they can call regular synchronous I/O functions such as those that access Django models without writing special code. However asynchronous consumers can provide a higher level of performance since they don't need create additional threads when handling requests.

Note:

Even if ChatConsumer did access Django models or other synchronous code it would still be possible to rewrite it as asynchronous. Utilities like `asgiref.sync.sync_to_async` (Utility class which turns a synchronous callable into an awaitable that runs in a threadpool) and `channels.db.database_sync_to_async` (subclass of `sync_to_async`, so the concept is the same) can be used to call synchronous code from an asynchronous consumer. The performance gains however would be less than if it only used async-native libraries.

Channels and groups

A channel is a mailbox where messages can be sent to. Each channel has a name. Anyone who has the name of a channel can send a message to the channel.

A group is a group of related channels. A group has a name. Anyone who has the name of a group can add/remove a channel to the group by name and send a message to all channels in the group. It is not possible to enumerate what channels are in a particular group.

Backing store

Channels can use a variety of options as a backing store for the channel layer.

- Redis (channels_redis package)
- POSIX shared memory IPC (asgi_ipc package) only works between processes on the same machine
- Database

- Others ???

Async Django

Async views

The main benefit is **the ability to serve hundreds of connections without using Python threads** (without using one thread per request but use one thread with an event loop). This allows you to use:

- Slow streaming
- Long polling
- Other exciting response types

These benefits are possible if you run the async views under ASGI. Under a WSGI server, async views will run in their own, one-off event loop. This means you can use async features, like concurrent async HTTP requests, without any issues, but you will not get the benefits of an async stack (slow streaming, long polling etc)

In both ASGI and WSGI mode, you can still safely use asynchronous support to run code concurrently rather than serially. This is especially handy when dealing with external APIs or data stores.

You will only get the benefits of a fully-asynchronous request stack if you have no synchronous middleware loaded into your site. If there is a piece of synchronous middleware, then Django must use a thread per request to safely emulate a synchronous environment for it.

Rules for full async stack

1. Run under ASGI
2. No use of synchronous middleware

No async support for ORM and other django parts in django 3.0

New in Django 4.1: With some exceptions, Django can run ORM queries asynchronously

All QuerySet methods that cause a SQL query to occur have an a-prefixed asynchronous variant (for example aget or afilter)

async for is supported on all QuerySets (including the output of values() and values_list().)

Transactions do not yet work in async mode. If you have a piece of code that needs transactions behavior, we recommend you write that piece as a single synchronous function and call it using sync_to_async().

Note

- In python you can't just call a sync function from an async function. It will block the thread, the event loop
- You can't call an async function from a sync function without creating a new event loop

sync_to_async

You have to use the sync_to_async() adapter to interact with the sync parts of Django

This runs the sync code in its own python thread instead of running it in the current thread which has a running event loop. It actually uses the `loop.run_in_executor` function and runs the function in a threadpool. It returns the result back to the main thread with the event loop. Notice that the calling async function will not block but will continue its execution.

<pre>from asgiref.sync import sync_to_async results = await sync_to_async(Blog.objects.get, thread_sensitive=True)(pk=123)</pre>	<p>If you want to call a part of Django that is still synchronous, like the ORM, you will need to wrap it in a sync_to_async() call.</p>
<pre>from asgiref.sync import sync_to_async def _get_blog(pk): return Blog.objects.select_related('author').get(pk=pk) get_blog = sync_to_async(_get_blog, thread_sensitive=True)</pre>	<p>You may find it easier to move any ORM code into its own function and call that entire function using sync_to_async()</p>

If you accidentally try to call a part of Django that is still synchronous-only from an async view, you will trigger Django's asynchronous safety protection to protect your data from corruption.

write your code that talks to async-unsafe functions in its own, sync function, and call that using `asgiref.sync.sync_to_async()` (or any other way of running sync code in its own thread).

Notice that many libraries, specifically database adapters, require that they are accessed in the same thread that they were created in. Also a lot of existing Django code assumes it all runs in the same thread, e.g. middleware adding things to a request for later use in views. This is what the `thread_sensitive = True` tries to achieve. With this argument the sync function will run in the same thread as all other `thread_sensitive` functions. Thread-sensitive mode is quite special, and does a lot of work to run all functions in the same thread.

(Have in mind that django creates a new database connection for every thread that uses the ORM. What happens in the `thread_sensitive = True` case? All ORM calls are made by the same thread sequentially? This would mean that the new ORM calls have to wait, and that you don't have the problem of multiple simultaneous database connections)

async_to_sync

Takes an async function and returns a sync function that wraps it. The wrapped async function will execute on a different thread to the calling code.

Declaring an async view

Any view can be declared async by making the callable part of it return a coroutine:

For function based views: `async def my_view()`

For class based views: make the `__call__` method an async function, `async def __call__(self)`

ASGI

Use an ASGI web server

ASGI is backwards-compatible with WSGI, making it a good excuse to switch from a WSGI server like Gunicorn or uWSGI to an ASGI server like Uvicorn or Daphne even if you're not ready to switch to writing asynchronous apps.

In development

<pre>pip install uvicorn uvicorn {name of your project}.asgi:application --reload</pre>	Django will run your async views if you're using the built-in development server, but it won't actually run them asynchronously, so you have to use an asgi web server for development too.
---	--

Production

In production, be sure to use Gunicorn to manage Uvicorn in order to take advantage of both concurrency (via Uvicorn) and parallelism (via Gunicorn workers):

<pre>gunicorn -w 3 -k uvicorn.workers.UvicornWorker hello_async.asgi:application</pre>	
--	--

Misc

call an async function

`await`

`loop.create_task`

To call an async function, you must either use the `await` keyword from another async function or call `create_task()` directly from the event loop, which can be grabbed from `asyncio.get_event_loop()`

<pre> async def http_call_async(): for num in range(1, 6): await asyncio.sleep(1) print(num) async with httpx.AsyncClient() as client: r = await client.get("https://httpbin.org/") print(r) async def async_view(request): loop = asyncio.get_event_loop() loop.create_task(http_call_async()) return HttpResponse("Non-blocking HTTP request") </pre>	<p>Why not using await to call the “http_call_async”? I guess you could, it uses the second method for demonstration probably.</p>
--	--

asyncio.gather

As of Python 3.9, asyncio.wait has a parameter called return_when, which you can use to control when the event loop should yield back to you. asyncio.gather does not have such parameter, the event loop only get back to you when all tasks have finished/failed. All tasks in a group scheduled with .gather can be cancelled.

<pre> async def smoke_some_meats(request): results = await asyncio.gather(*[get_smokables(), get_flavor()]) total = await asyncio.gather(*[smoke(results[0], results[1])]) return HttpResponse(f"Smoked {total[0]} meats with {results[1]}!") </pre>	<p>Here the smoke async function takes as arguments the values returned from get_smokables and get_flavor async functions. That's why we should use asyncio.gather which waits until each async task is complete</p>
--	--

asyncio.run(coroutine())

<pre> main.py 1 import asyncio 2 3 async def main(): 4 5 async def test(): 6 await asyncio.sleep(5) 7 return 1 8 9 res = await test() 10 print(res) 11 12 asyncio.run(main()) </pre>	<p>In this case you couldn't just do await write() because await works only inside an async function.</p>
--	---

async with

<pre> async def write_genre(file_name): """ Uses genrenator from binaryjazz.us to write a random genre to the name of the given file """ async with aiohttp.ClientSession() as session: async with session.get("https://binaryjazz.us/wp-json/genrenator/v1/genre/") as response: genre = await response.json() async with aiofiles.open(file_name, "w") as new_file: print(f'Writing "{genre}" to "{file_name}"...') await new_file.write(genre) </pre>	<p>allows awaiting async responses and file operations.</p>
--	---

async for

iterates over an asynchronous stream

Httpx

Why not use the requests package? As I understand you can't make async http calls with requests. You need to use an async http client.

Asyncio.gather

As the name suggests, asyncio.gather mainly focuses on gathering the results. It waits on a bunch of futures and returns their results in a given order.

asyncio.wait just waits on the futures. And instead of giving you the results directly, it gives done and pending tasks. You have to manually collect the values.

Notice

If you want to make async calls to multiple urls, it's far better to create a task for each call, and execute the list of tasks with asyncio.gather

Sync example

```
>>> import httpx
>>> r = httpx.get('https://www.example.org/')
>>> r
<Response [200 OK]>
>>> r.status_code
200
>>> r.headers['content-type']
'text/html; charset=UTF-8'
>>> r.text
'<!doctype html>\n<html>\n<head>\n<title>Example Domain</title>...'
```

Async example

```
>>> import httpx
>>> async with httpx.AsyncClient() as client:
...     r = await client.get('https://www.example.org/')
...
>>> r
<Response [200 OK]>
```

HTTPX is a fully featured HTTP client for Python 3, which provides sync and async APIs, and support for both HTTP/1.1 and HTTP/2.

pip install httpx
import httpx

Couldn't you do the same with another http client? What does the http client needs to have in order to be able to be called asynchronously?