BUILDING APIS
WITH
DJANGO
AND
DJANGO REST FRAMEWORK

Updated for Django 2.0+ and Python 3.6+

Building APIs with Django and Django Rest Framework

Building APIs with Django and DRF takes over where the Django tutorials stop.

In the Django tutorials, you built a regular Django polls app. We will rebuild an API for a similar app.

In the chapters to come, we will build a REST(ish) api with authorization, rate limiting, first with pure Django and then with DRF.

We will cover testing, continuous integration, documentation tools and API collaboration tools.

BUILDING APIS
WITH
DJANGO
AND
DJANGO REST FRAMEWORK

Updated for Django 2.0+ and Python 3.6+

Chapters:

Appendix

Introductions

Building APIs with Django and Django Rest Framework starts where the [Django "Polls" tutorial](https://docs.djangoproject.com/en/2.0/intro/tutorial01/) [https://docs.djangoproject.com/en/2.0/intro/tutorial01/] stops, and takes you through building the polls app, but this time using APIs. You will learn the basics of Django Rest Framework including serialization, views, generic views, viewsets, testing, access control. You will also learn about API documentation using swagger and raml.

Who is this book for?

If you have finished the Django "Polls" tutorial, and want to learn using DRF to build APIs, this book is perfect for you. This book assumes some knowledge of Django and Python, which you should have built if you have finished the "Poll" turtorial. No existing knowledge of DRF is assumed.

How to read this book?

The chapters are meant to be read in order. If you have existing knowledge of some chapters, you can quickly go through that chapter, but I highly recommend reading them in order as each chapter builds on the previous.

Setup, Models and Admin

In this tutorial we will walk through a process of creating an API for a basic poll application. We will be using Python 3.6.x, Django 2.0.x and Django Rest Framework 3.7.x for creating API.

First things first, let's install the required modules within a virtual environment.

```
 mkvirtualenv pollsapi
pip install Django
pip install djangorestframework
```

Creating a project

Earliest in order, to create a project we should move to the directory where we would like to store our code. For this go to command line and use cd command. Then trigger the startproject command.

```
django-admin startproject pollsapi
```

This command gives us a â€˜pollsapiâ€™ directoy. The contents of this directory look like this:

```
manage.py

pollsapi/
    __init__.py
    settings.py
    urls.py
    wsgi.py
```

Database setup

We will use SQlite database, which is already included with Python. The
`pollsapi/settings.py` file would already have the correct settings.

```python
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Now, use the migrate command which builds the needed database tables in
regard to the `django_pollsapi/settings.py` file.

```python
python manage.py migrate
```

Creating models

Before creating our database models, let us create our pollsapi App.

```
python manage.py startapp polls
```

The above command results in a 'polls' directory containing different files:

```
 admin.py
apps.py
models.py
tests.py
views.py
```

Step in to 'models.py' file and start writing the models. For creating the polls api we are going to create a `Poll` model, a `Choice` model and a `Vote` model. Once we are done with designing our models, the `models.py` file should look like this:

These models are the same as you would have seen in the Django introduction tutorial.

```python
from django.db import models
from django.contrib.auth.models import User


class Poll(models.Model):
    question = models.CharField(max_length=100)
    created_by = models.ForeignKey(User, on_delete=models.CASCADE)
    pub_date = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.question


class Choice(models.Model):
    poll = models.ForeignKey(Poll,
related_name='choices',on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=100)

    def __str__(self):
        return self.choice_text
```

```python
class Vote(models.Model):
    choice = models.ForeignKey(Choice, related_name='votes',
on_delete=models.CASCADE)
    poll = models.ForeignKey(Poll, on_delete=models.CASCADE)
    voted_by = models.ForeignKey(User, on_delete=models.CASCADE)

    class Meta:
        unique_together = ("poll", "voted_by")
```

The above models have been designed in such a way that, it would make our
API bulding a smooth process.

Activating models

With the simple lines of code in the 'models.py' Django can create a database schema and a Python database-access API which has the capability to access the objects of Poll, Choice, Vote. To create the database tables to our models, 'rest_framework' and 'pollsapi' app needs to be added to the "INSTALLED_APPS" in the 'django_pollsapi/settings' file.

```
INSTALLED_APPS = (
...
'rest_framework',
'polls',
)
```

Now, run the `makemigrations` command which will notify Django that new models have been created and those changes needs to be applied to the migration. Run `migrate` command to do the actual migration.

```
$ python manage.py makemigrations polls

$ python manage.py migrate
```

Create an empty `urls.py` in your `polls` app.

```
urlpatterns = [
]
```

Go to `pollsapi/urls.py` and include the polls urls.

```
urlpatterns = [
    re_path(r'^', include('polls.urls')),
]
```

Now you can runserver

```
$ python manage.py runserver
```

Goto any browser of your choice and hit the url `http://127.0.0.1:8000`

And we are in business, with a Django *Congratulations* page greeting us. (Though we haven't added any API endpoints yet.)

The install worked successfully! Congratulations!

You are seeing this page because DEBUG=True is in
your settings file and you have not configured any
URLs.

We will be adding API endpoints for creating and viewing polls in the next chapter.

Setting up the admin

You should register `Poll` and `Choice` in the admin like this.

```python
from django.contrib import admin

from .models import Poll, Choice

admin.site.register(Poll)
admin.site.register(Choice)
```

A simple API with pure Django

In this chapter, we will build an API with pure Django. We will not use Django Rest Framework (Or any other library). To start add some `Poll` using the admin.

The endpoints and the URLS

Our API will have two endpoints returning data in JSON format.

- `/polls/` GETs list of `Poll`
- `/polls/<id>/` GETs data of a specific `Poll`

Connecting urls to the views

Write two place holder view functions and connect them in your `urls.py`. We will finish `polls_list` and `polls_detail` shortly.

```python
 # In views.py
def polls_list(request):
    pass


def polls_detail(request, pk):
    pass


# in urls.py
from django.urls import path
from .views import polls_list, polls_detail

urlpatterns = [
    path("polls/", polls_list, name="polls_list"),
    path("polls/<int:pk>/", polls_detail, name="polls_detail")
]
```

Writing the views

We will now write the `polls_list` and `polls_detail`

```python
from django.shortcuts import render, get_object_or_404
from django.http import JsonResponse

from .models import Poll

def polls_list(request):
    MAX_OBJECTS = 20
    polls = Poll.objects.all()[:20]
    data = {"results": list(polls.values("question",
"created_by__username", "pub_date"))}
    return JsonResponse(data)


def polls_detail(request, pk):
    poll = get_object_or_404(Poll, pk=pk)
    data = {"results": {
        "question": poll.question,
        "created_by": poll.created_by.username,
        "pub_date": poll.pub_date
    }}
    return JsonResponse(data)
```

This should be standard Django for you. `polls = Poll.objects.all()[:20]`
gets us upto 20 `Poll` objects. We get a list of dictionaries using `{"results":`
`list(polls.values("question", "created_by__username", "pub_date"))}`
and return it with a `JsonResponse`. A `JsonResponse` is a like `HttpResponse` with
`content-type=application/json`.

Similarly, *polls_detail* gets a specific Poll using `get_object_or_404(Poll,`
`pk=pk)`, and returns it wrapped in `JsonResponse`.

Using the API

You can now access the API using curl, wget, postman, browser or any other API consuming tools. Here us the response with curl.

```
$ curl http://localhost:8000/polls/

{"results": [{"pk": 1, "question": "What is the weight of an
unladen swallow?", "created_by__username": "shabda", "pub_date":
"2018-03-12T10:14:19.002Z"}, {"pk": 2, "question": "What do you
prefer, Flask or Django?", "created_by__username": "shabda",
"pub_date": "2018-03-12T10:15:55.949Z"}, {"pk": 3, "question":
"What is your favorite vacation spot?", "created_by__username":
"shabda", "pub_date": "2018-03-12T10:16:11.998Z"}]}
```

You should consider using postman or a similar tool. This is how your API looks in Postman.
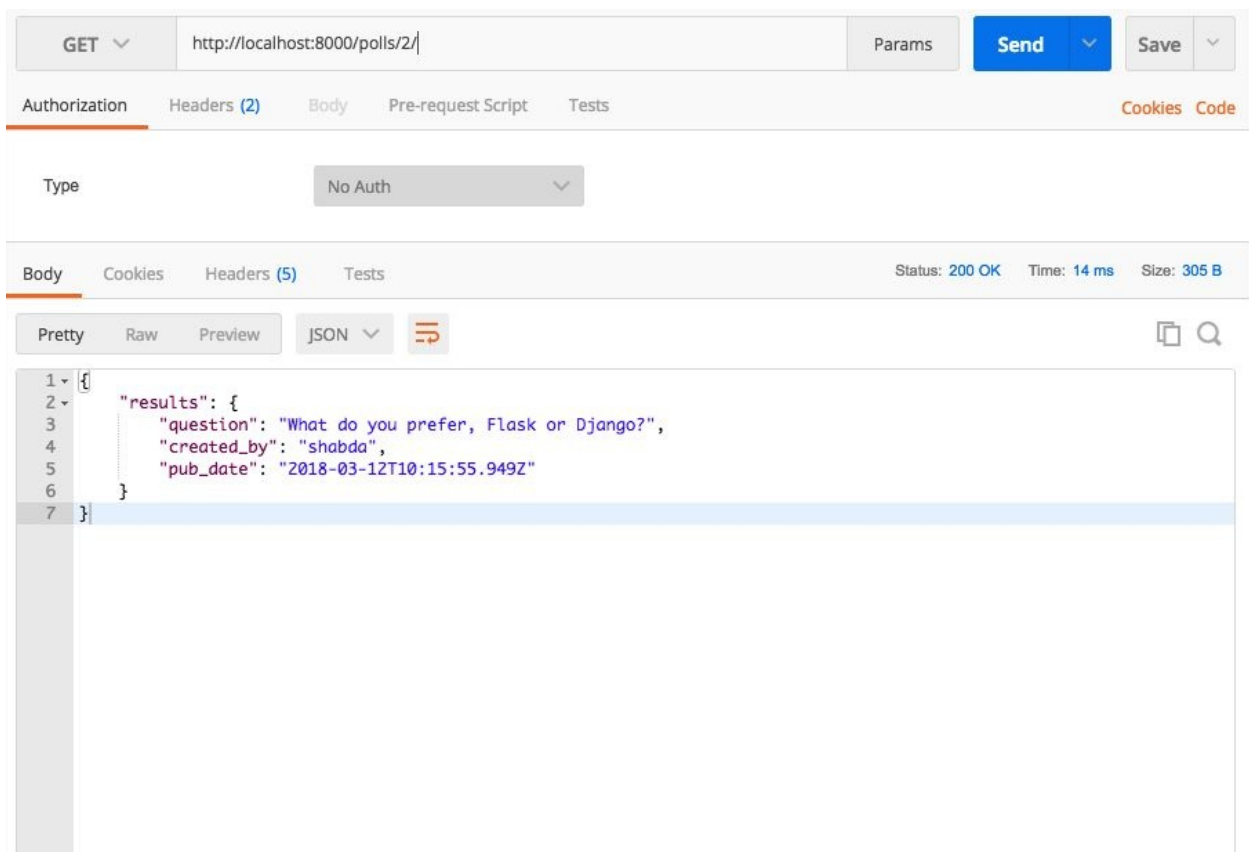
Why do we need DRF?

(DRF = Django Rest Framework)

We were able to build the API with just Django, without using DRF, so why do we need DRF? Almost always, you will need common tasks with your APIs, such as access control, serialization, rate limiting and more.

DRF provides a well thought out set of base components and convenient hook points for building APIs. We will be using DRF in the rest of the chapters.

Serializing and Deserializing Data

DRF makes the process of building web API's simple and flexible. With batteries included, it comes with well designed base classes which allows us to serialize and deserialize data.

Serialization and Deserialization

The first thing we need for our API is to provide a way to serialize model instances into representations. Serialization is the process of making a streamable representation of the data which we can transfer over the network. Deserialization is its reverse process.

Creating Serializers

Lets get started with creating serializer classes which will serialize and deserialize the model instances to json representations. Create a file named `polls/serializers.py`. We will use `ModelSerializer` which will reduce code duplication by automatically determing the set of fields and by creating implementations of the `create()` and `update()` methods.

Our `polls/serializers.py` looks like this.

```python
from rest_framework import serializers

from .models import Poll, Choice, Vote


class VoteSerializer(serializers.ModelSerializer):
    class Meta:
        model = Vote
        fields = '__all__'


class ChoiceSerializer(serializers.ModelSerializer):
    votes = VoteSerializer(many=True, required=False)

    class Meta:
        model = Choice
        fields = '__all__'


class PollSerializer(serializers.ModelSerializer):
    choices = ChoiceSerializer(many=True, read_only=True,
required=False)

    class Meta:
        model = Poll
        fields = '__all__'
```

The `PollSerializer` in detail

Our `PollSerializer` looks like this.

```
...

class PollSerializer(serializers.ModelSerializer):
    choices = ChoiceSerializer(many=True, read_only=True,
required=False)

    class Meta:
        model = Poll
        fields = '__all__'
```

What have we got with this? The `PollSerializer` class has a number of methods,

- A `is_valid(self, ..)` method which can tell if the data is sufficient and valid to create/update a model instance.
- A `save(self, ..)` method, which khows how to create or update an instance.
- A `create(self, validated_data, ..)` method which knows how to create an instance. This method can be overriden to customize the create behaviour.
- A `update(self, instance, validated_data, ..)` method which knows how to update an instance. This method can be overriden to customize the update behaviour.

Using the `PollSerializer`

Letâ€™s use the serializer to create a `Poll` object.

```
 In [1]: from polls.serializers import PollSerializer

In [2]: from polls.models import Poll

In [3]: poll_serializer = PollSerializer(data={"question": "Mojito
or Caipirinha?", "created_by": 1})

In [4]: poll_serializer.is_valid()
Out[4]: True

In [5]: poll = poll_serializer.save()

In [6]: poll.pk
Out[6]: 5
```

The `poll.pk` line tells us that the object has been commited to the DB. You can also use the serializer to update a `Poll` object.

```
 In [9]: poll_serializer = PollSerializer(instance=poll, data=
{"question": "Mojito, Caipirinha or margarita?", "created_by": 1})

In [10]: poll_serializer.is_valid()
Out[10]: True

In [11]: poll_serializer.save()
Out[11]: <Poll: Mojito, Caipirinha or margarita?>

In [12]: Poll.objects.get(pk=5).question
Out[12]: 'Mojito, Caipirinha or margarita?'
```

We can see that calling save on a Serializer with instance causes that instance to be updated. `Poll.objects.get(pk=5).question` verifies that the Poll was updated.

In the next chapter, we will use the serializers to write views.

Views and Generic Views

In this chapter, we will create views using `APIVIew`, and `generics.ListCreateAPIView` and family.

Creating Views with `APIView`

To start with, we will use the `APIView` to build the polls list and poll detail API we built in the chapter, [A simple API with pure Django](#).

Add this to a new file `polls/apiviews.py`

```python
from rest_framework.views import APIView
from rest_framework.response import Response
from django.shortcuts import get_object_or_404

from .models import Poll, Choice
from .serializers import PollSerializer

class PollList(APIView):
    def get(self, request):
        polls = Poll.objects.all()[:20]
        data = PollSerializer(polls, many=True).data
        return Response(data)


class PollDetail(APIView):
    def get(self, request, pk):
        poll = get_object_or_404(Poll, pk=pk)
        data = PollSerializer(poll).data
        return Response(data)
```

And change your `urls.py` to

```python
from django.urls import path

from .apiviews import PollList, PollDetail

urlpatterns = [
    path("polls/", PollList.as_view(), name="polls_list"),
    path("polls/<int:pk>/", PollDetail.as_view(),
name="polls_detail")
]
```

DRF comes with a browsable api, so you can directly open `http://localhost:8000/polls/` in the browser. It looks like this

```
Poll List                                                    OPTIONS   GET ▾

GET /polls/?format=api

HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "choices": [],
        "question": "What is the weight of an unladen swallow?",
        "pub_date": "2018-03-12T10:14:19.002337Z",
        "created_by": 1
    },
    {
        "id": 2,
        "choices": [],
        "question": "What do you prefer, Flask or Django?",
        "pub_date": "2018-03-12T10:15:55.949721Z",
        "created_by": 1
    },
    {
        "id": 3,
        "choices": [],
        "question": "What is your favorite vacation spot?",
        "pub_date": "2018-03-12T10:16:11.998387Z",
        "created_by": 1
    },
    {
        "id": 4,
        "choices": [],
        "question": "Mojito, Caipirinha or margarita?",
        "pub_date": "2018-03-13T09:24:52.709870Z",
        "created_by": 1
    },
    {
        "id": 5,
        "choices": [],
        "question": "Mojito, Caipirinha or margarita?",
        "pub_date": "2018-03-13T10:40:01.601316Z",
        "created_by": 1
    }
]
```

You can now do an `options` request to */polls/*, which gives

```
{
    "name": "Poll List",
    "description": "",
    "renders": [
        "application/json",
        "text/html"
    ],
    "parses": [
        "application/json",
        "application/x-www-form-urlencoded",
        "multipart/form-data"
    ]
}
```

This is how it looks like in postman.

Params   **Send** ∨   Save ∨

Authorization   Headers (2)   Body ●   Pre-request Script   Tests   Cookies   Code

Type   No Auth ∨

Body   Cookies   Headers (7)   Tests   Status: **200 OK**   Time: **82 ms**   Size: **393 B**

Pretty   Raw   Preview   JSON ∨

```json
1  {
2      "name": "Poll List",
3      "description": "",
4      "renders": [
5          "application/json",
6          "text/html"
7      ],
8      "parses": [
9          "application/json",
10         "application/x-www-form-urlencoded",
11         "multipart/form-data"
12      ]
13  }
```

Using DRF generic views to simplify code

The `PollList` and `PollDetail` get the work done, but there are bunch of common operations, we can do it in abstract away.

The generic views of Django Rest Framework help us in code reusablity. They infer the response format and allowed methods from the serilizer class and base class.

Change your `apiviews.py` to the below code, and leave urls.py as is.

```python
from rest_framework import generics

from .models import Poll, Choice
from .serializers import PollSerializer, ChoiceSerializer,\
    VoteSerializer


class PollList(generics.ListCreateAPIView):
    queryset = Poll.objects.all()
    serializer_class = PollSerializer


class PollDetail(generics.RetrieveDestroyAPIView):
    queryset = Poll.objects.all()
    serializer_class = PollSerializer
```

With this change, GET requests to `/polls/` and `/polls/<pk>/`, continue to work as was, but we have a more data available with OPTIONS.

Do an OPTIONs request to `/polls/`, and you will get a response like this.

```json
{
    "name": "Poll List",
    "description": "",
    "renders": [
        "application/json",
        "text/html"
    ],
    "parses": [
        "application/json",
        "application/x-www-form-urlencoded",
        "multipart/form-data"
    ],
```

```
    "actions": {
        "POST": {
            "id": {
                "type": "integer",
                "required": false,
                "read_only": true,
                "label": "ID"
            },
            // ...
            },
            "question": {
                "type": "string",
                "required": true,
                "read_only": false,
                "label": "Question",
                "max_length": 100
            },
            "pub_date": {
                "type": "datetime",
                "required": false,
                "read_only": true,
                "label": "Pub date"
            },
            "created_by": {
                "type": "field",
                "required": true,
                "read_only": false,
                "label": "Created by"
            }
        }
    }
}
```

This tells us

- Our API now accepts POST
- The required data fields
- The type of each data field.

Pretty nifty! This is what it looks like in Postman.

Send ∨ **Save** ∨

Authorization   Headers (2)   Body ●   Pre-request Script   Tests   Cookies Code

Type   No Auth   ∨

Body   Cookies   Headers (7)   Tests   Status: **200 OK**   Time: **16 ms**   Size: **1.59 KB**

Pretty   Raw   Preview   JSON ∨   ⇥

```json
1  {
2      "name": "Poll List",
3      "description": "",
4      "renders": [
5          "application/json",
6          "text/html"
7      ],
8      "parses": [
9          "application/json",
10         "application/x-www-form-urlencoded",
11         "multipart/form-data"
12     ],
13     "actions": {
14         "POST": {
15             "id": {
16                 "type": "integer",
17                 "required": false,
18                 "read_only": true,
19                 "label": "ID"
20             },
21             "choices": {
22                 "type": "field",
23                 "required": false,
24                 "read_only": true,
25                 "label": "Choices",
26                 "child": {
27                     "type": "nested object",
28                     "required": false,
29                     "read_only": true,
30                     "children": {
31                         "id": {
32                             "type": "integer",
33                             "required": false,
34                             "read_only": true,
35                             "label": "ID"
```

More generic views

Let us add the view to create choices and for voting. We will look more closely at this code shortly.

```python
 from rest_framework import generics

from .models import Poll, Choice
from .serializers import PollSerializer, ChoiceSerializer,
VoteSerializer


class PollList(generics.ListCreateAPIView):
    queryset = Poll.objects.all()
    serializer_class = PollSerializer


class PollDetail(generics.RetrieveDestroyAPIView):
    queryset = Poll.objects.all()
    serializer_class = PollSerializer


class ChoiceList(generics.ListCreateAPIView):
    queryset = Choice.objects.all()
    serializer_class = ChoiceSerializer


class CreateVote(generics.CreateAPIView):
    serializer_class = VoteSerializer
```

Connect the new apiviews to urls.py.

```python
 # ...
urlpatterns = [
    # ...
    path("choices/", ChoiceList.as_view(), name="choice_list"),
    path("vote/", CreateVote.as_view(), name="create_vote"),

]
```

There is a lot going on here, let us look at the attributes we need to override or set.

- queryset: This determines the initial queryset. The queryset can be further filtered, sliced or ordered by the view.

- serializer_class: This will be used for validating and deserializing the input and for serializing the output.

We have used three different classes from rest_framework.generic. The names of the classes are representative of what they do, but lets quickly look at them.

- ListCreateAPIView: Get a list of entities, or create them. Allows GET and POST
- RetrieveDestroyAPIView: Retrieve and inidvidual entity details, or delete the entity. Allows GET and DELETE
- CreateAPIView: Allows creating entities, but not listing them. Allows POST.

Create some choices by POSTing to /choices/.

```json
{
    "choice_text": "Flask",
    "poll": 2
}
```

The response looks like this

```json
{
    "id": 4,
    "votes": [],
    "choice_text": "Flask",
    "poll": 2
}
```

You can also retrieve the Poll to by doing a GET to /poll/<pk>/. You should get something like this

```json
{
    "id": 2,
    "choices": [
        {
            "id": 3,
            "votes": [],
            "choice_text": "Django",
            "poll": 2
        },
        {
            "id": 4,
            "votes": [],
            "choice_text": "Flask",
```

```
            "poll": 2
        }
    ],
    "question": "What do you prefer, Flask or Django?",
    "pub_date": "2018-03-12T10:15:55.949721Z",
    "created_by": 1
}
```

If you make a mistake while POSTing, the API will warn you. POST a json with `choice_text` missing to `/choices/`.

```
{
    "poll": 2
}
```

You will get a response like this

```
{
    "poll": [
        "This field is required."
    ]
}
```

Check the status code is 400 Bad Request.

Next Steps

We have working API at this point, but we can simplify our API with a better URL design and remove some code duplication using viewsets. We will be doing that in the next chapter.

More views and viewsets

A better URL structure

We have three API endpoints

- `/polls/` and `/polls/<pk>/`
- `/choices/`
- `/vote/`

They get the work done, but we can make our API more intuitive by nesting them correctly. Our redesigned urls look like this:

- `/polls/` and `/polls/<pk>`
- `/polls/<pk>/choices/` to GET the choices for a specific poll, and to create choices for a specific poll. (Idenitfied by the `<pk>`)
- `/polls/<pk>/choices/<choice_pk>/vote/` - To vote for the choice identified by `<choice_pk>` under poll with `<pk>`.

Changing the views

We will make changes to `ChoiceList` and `CreateVote`, because the `/polls/` and `/polls/<pk>` have not changed.

```python
 from rest_framework import generics
from rest_framework.views import APIView
from rest_framework import status
from rest_framework.response import Response

from .models import Poll, Choice
from .serializers import PollSerializer, ChoiceSerializer,
VoteSerializer

# ...
# PollList and PollDetail views

class ChoiceList(generics.ListCreateAPIView):
    def get_queryset(self):
        queryset = Choice.objects.filter(poll_id=self.kwargs["pk"])
        return queryset
    serializer_class = ChoiceSerializer


class CreateVote(APIView):

    def post(self, request, pk, choice_pk):
        voted_by = request.data.get("voted_by")
        data = {'choice': choice_pk, 'poll': pk, 'voted_by':
voted_by}
        serializer = VoteSerializer(data=data)
        if serializer.is_valid():
            vote = serializer.save()
            return Response(serializer.data,
status=status.HTTP_201_CREATED)
        else:
            return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
```

And change your urls.py to a nested structure.

```python
 #...
urlpatterns = [
    path("polls/<int:pk>/choices/", ChoiceList.as_view(),
name="choice_list"),
    path("polls/<int:pk>/choices/<int:choice_pk>/vote/",
```

```
    CreateVote.as_view(), name="create_vote"),

]
```

You can see the changes by doing a GET to
`http://localhost:8000/polls/1/choices/`, which should give you.

```
[
    {
        "id": 1,
        "votes": [],
        "choice_text": "Flask",
        "poll": 1
    },
    {
        "id": 2,
        "votes": [
        ],
        "choice_text": "Django",
        "poll": 1
    }
]
```

You can vote for choices 2, of poll 1 by doing a POST to
`http://localhost:8000/polls/1/choices/2/vote/` with data `{"voted_by":
1}`.

```
{
    "id": 2,
    "choice": 2,
    "poll": 1,
    "voted_by": 1
}
```

Lets get back to `ChoiceList`.

```python
 # urls.py
#...
urlpatterns = [
    # ...
    path("polls/<int:pk>/choices/", ChoiceList.as_view(),
name="choice_list"),
]

# views.py
# ...
```

```python
class ChoiceList(generics.ListCreateAPIView):
    def get_queryset(self):
        queryset = Choice.objects.filter(poll_id=self.kwargs["pk"])
        return queryset
    serializer_class = ChoiceSerializer
```

From the urls, we pass on `pk` to `ChoiceList`. We override the `get_queryset` method, to filter on choices with this `poll_id`, and let DRF handle the rest.

And for `CreateVote`,

```python
# urls.py
#...
urlpatterns = [
    # ...
    path("polls/<int:pk>/choices/<int:choice_pk>/vote/",
CreateVote.as_view(), name="polls_list"),
]

# views.py
# ...

class CreateVote(APIView):

    def post(self, request, pk, choice_pk):
        voted_by = request.data.get("voted_by")
        data = {'choice': choice_pk, 'poll': pk, 'voted_by':
voted_by}
        serializer = VoteSerializer(data=data)
        if serializer.is_valid():
            vote = serializer.save()
            return Response(serializer.data,
status=status.HTTP_201_CREATED)
        else:
            return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
```

We pass on poll id and choice id. We subclass this from `APIView`, rather than a generic view, because we competely customize the behaviour. This is similar to our earlier `APIView`, where in we are passing the data to a serializer, and saving or returning an error depending on whether the serializer is valid.

Introducing Viewsets and Routers

Our urls are looking good, and we have a views with very little code duplication, but we can do better.

The `/polls/` and `/polls/<pk>/` urls require two view classes, with the same serializer and base queryset. We can group them into a viewset, and connect them to the urls using a router.

This is what it will look like:

```python
 # urls.py
# ...
from rest_framework.routers import DefaultRouter

router = DefaultRouter()
router.register('polls', PollViewSet, base_name='polls')


urlpatterns = [
    # ...
]

urlpatterns += router.urls

# views.py
# ...
from rest_framework import viewsets

from .models import Poll, Choice
from .serializers import PollSerializer, ChoiceSerializer,
VoteSerializer


class PollViewSet(viewsets.ModelViewSet):
    queryset = Poll.objects.all()
    serializer_class = PollSerializer
```

There is no change at all to the urls or to the responses. You can verify this by doing a GET to `/polls/` and `/polls/<pk>/`.

Choosing the base class to use

We have seen 4 ways to build API views until now

- Pure Django views
- `APIView` subclasses
- `generics.*` subclasses
- `viewsets.ModelViewSet`

So which one should you use when? My rule of thumb is,

- Use `viewsets.ModelViewSet` when you are goin to allow all or most of CRUD operations on a model.
- Use `generics.*` when you only want to allow some operations on a model
- Use `APIView` when you want to completely customize the behaviour.

Next steps

In the next chapter, we will look at adding access control to our apis.

Access Control

In this chapter, we will add access control to our APIs, and add APIs to create and authenticate users.

Right now our APIs are completely permissive. Anyone can create, access and delete anything. We want to add these access controls.

- A user must be authenticated to access a poll or the list of polls.
- Only an authenicated users can create a poll.
- Only an authenticated user can create a choice.
- Authenticated users can create choices only for polls they have created.
- Authenticated users can delete only polls they have created.
- Only an authenticated user can vote. Users can vote for other people's polls.

To enable the access control, we need to add two more APIs

- API to create a user, we will call this endpoint `/users/`
- API to verify a user and get a token to identify them, we will call this endpoint `/login/`

Creating a user

We will add an user serializer, which will allow creating. Add the following code to `serializers.py`.

```python
class UserSerializer(serializers.ModelSerializer):

    class Meta:
        model = User
        fields = ('username', 'email', 'password')
        extra_kwargs = {'password': {'write_only': True}}

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

We have overriden the ModelSerializer methodâ€™s `create()` to save the `User` instances. We ensure that we set the password correctly using `user.set_password`, rather than setting the raw password as the hash. We also donâ€™t want to get back the password in response which we ensure using `extra_kwargs = {'password': {'write_only': True}}`.

Let us also add views to the User Serializer for creating the user and connect it to the urls.py

```python
 # in apiviews.py
# ...
from .serializers import PollSerializer, ChoiceSerializer,
VoteSerializer, UserSerializer

# ...
class UserCreate(generics.CreateAPIView):
    serializer_class = UserSerializer

# in urls.py
# ...
from .apiviews import PollViewSet, ChoiceList, CreateVote,
UserCreate
```

```
urlpatterns = [
    # ...
    path("users/", UserCreate.as_view(), name="user_create"),
]
```

We can test this api by posting to `/users/` with this json.

```json
{
    "username": "nate.silver",
    "email": "nate.silver@example.com",
    "password": "FiveThirtyEight"
}
```

Which give back this response.

```json
{
    "username": "nate.silver",
    "email": "nate.silver@example.com"
}
```

Try posting the same json, and you will get a error response (HTTP status code 400)

```json
{
    "username": [
        "A user with that username already exists."
    ]
}
```

Authentication scheme setup

With Django Rest Framework, we can set up a default authentication scheme which is applied to all views using `DEFAULT_AUTHENTICATION_CLASSES`. We will use the token authentication in this tutorial. In your settings.py, add this.

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.TokenAuthentication',
        'rest_framework.authentication.SessionAuthentication',
    )
}
```

You also need to enable `rest_framework.authtoken` app, the so update `INSTALLED_APPS` in your settings.py.

```
INSTALLED_APPS = (
    ...
    'rest_framework.authtoken'
)
```

Run `python manage.py migrate` to create the new tables.

```
REST_FRAMEWORK = {
    # ...
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    )
}
```

Also, dont forget to give exemption to `UserCreate` view for authentication by overriding the global setting. The `UserCreate` in `polls/apiviews.py` should look as follows.

```
class UserCreate(generics.CreateAPIView):
    authentication_classes = ()
    permission_classes = ()
    serializer_class = UserSerializer
```

Note the `authentication_classes = ()` and `permission_classes = ()` to exempt `UserCreate` from global authentication scheme.

We want to ensure that tokens are created when user is created in `UserCreate`

view, so we update the `UserSerializer`. Change your `serializers.py` like this

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

class UserSerializer(serializers.ModelSerializer):

    class Meta:
        model = User
        fields = ('username', 'email', 'password')
        extra_kwargs = {'password': {'write_only': True}}

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

The login API

Since we have added `rest_framework.authentication.TokenAuthentication`, we will need to set an header like this `Authorization: Token c2a84953f47288ac1943a3f389a6034e395ad940` to auhenticate. We need an API where a user can give their username and password, and get a token back.

We will not be adding a serializer, because we never save a token using this API.

Add a view and connect it to urls.

```python
 # in apiviews.py
# ...
from django.contrib.auth import authenticate

class LoginView(APIView):
    permission_classes = ()

    def post(self, request,):
        username = request.data.get("username")
        password = request.data.get("password")
        user = authenticate(username=username, password=password)
        if user:
            return Response({"token": user.auth_token.key})
        else:
            return Response({"error": "Wrong Credentials"},
status=status.HTTP_400_BAD_REQUEST)


# in urls.py
# ...

from .apiviews import PollViewSet, ChoiceList, CreateVote,
UserCreate, LoginView


urlpatterns = [
    path("login/", LoginView.as_view(), name="login"),
    # ...
]
```

Do a POST with a correct username and password, and you will get a response like this.

```
{
    "token": "c300998d0e2d1b8b4ed9215589df4497de12000c"
}
```

POST with a incorrect username and password, and you will get a response like this, with a HTTP status of 400.

```
{
    "error": "Wrong Credentials"
}
```

Fine grained access control

Try accessing the `/polls/` API without any header. You will get an error with a http status code of `HTTP 401 Unauthorized` like this.

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

Add an authorization header `Authorization: Token <your token>`, and you can access the API.

From now onwards we will use a HTTP header like this, `Authorization: Token <your token>` in all further requests.

We have two remaining things we need to enforce.

- Authenticated users can create choices only for polls they have created.
- Authenticated users can delete only polls they have created.

We will do that by overriding `PollViewSet.destroy` and `ChoiceList.post`.

```python
class PollViewSet(viewsets.ModelViewSet):
    # ...

    def destroy(self, request, *args, **kwargs):
        poll = Poll.objects.get(pk=self.kwargs["pk"])
        if not request.user == poll.created_by:
            raise PermissionDenied("You can not delete this poll.")
        return super().destroy(request, *args, **kwargs)


class ChoiceList(generics.ListCreateAPIView):
    # ...

    def post(self, request, *args, **kwargs):
        poll = Poll.objects.get(pk=self.kwargs["pk"])
        if not request.user == poll.created_by:
            raise PermissionDenied("You can not create choice for
this poll.")
        return super().post(request, *args, **kwargs)
```

In both cases, we are checkeding the `request.user` against the expected user,

and raising as `PermissionDenied` if it does not match.

You can check this by doing a DELETE on someone elses `Poll`. You will get an error with `HTTP 403 Forbidden` and response.

```
{
    "detail": "You can not delete this poll."
}
```

Similarly trying to create choice for someone else's `Poll` will get an error with `HTTP 403 Forbidden` and response

```
{
    "detail": "You can not create choice for this poll."
}
```

Next steps:

In the next chapter we will look at adding tests for our API and serializers. We will also look at how to use `flake8` and run our tests in a CI environment.

Testing and Continuous Integeration

In this chapter we will add test to our API.

DRF provides a few important classes which makes testing APIs simpler. We will be using these classes later in the chapter in our tests.

- `APIRequestFactory`: This is similar to Django's `RequestFactory`. It allows you to create requests with any http method, which you can then pass on to any view method and compare responses.
- `APIClient`: similar to Django's `Client`. You can GET or POST a URL, and test responses.
- `APITestCase`: similar to Django's `TestCase`. Most of your tests will subclass this.

Now lets us write test cases to our polls application.

Creating Test Requests

Django's 'Requestfactory' has the capability to create request instances which allow us in testing view functions individually. Django Rest Framework has a class called 'APIRequestFactory' which extends the standard Django's 'RequestFactory'. This class contains almost all the http verbs like .get(), .post(), .put(), .patch() et all.

Syntax for Post request:

```
factory = APIRequestFactory()
request = factory.post(uri, post data)
```

Lets add a test for the polls list.

```python
from rest_framework.test import APITestCase
from rest_framework.test import APIRequestFactory

from polls import apiviews


class TestPoll(APITestCase):
    def setUp(self):
        self.factory = APIRequestFactory()
        self.view = apiviews.PollViewSet.as_view({'get': 'list'})
        self.uri = '/polls/'

    def test_list(self):
        request = self.factory.get(self.uri)
        response = self.view(request)
        self.assertEqual(response.status_code, 200,
                         'Expected Response Code 200, received {0} instead.'
                         .format(response.status_code))
```

In the above lines of code, we are trying to access the PollList view. We are asserting that the HTTP response code is 200.

Now run the test command.

```
python manage.py test
```

And it will display the below message.

```
 Creating test database for alias 'default'...
System check identified no issues (0 silenced).
F
================================================================
FAIL: test_list (polls.tests.TestPoll)
-------------------------------------------------------------------
---
Traceback (most recent call last):
  File "/Users/shabda/repos/building-api-
django/pollsapi/polls/tests.py", line 19, in test_list
    .format(response.status_code))
AssertionError: 401 != 200 : Expected Response Code 200, received
401 instead.

-------------------------------------------------------------------
---
Ran 1 test in 0.002s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

Ouch! Our test failed. This happened because the view is not accessable without authentication. So we need to create a user and test the view after getting authenticated.

Testing APIs with authentication

To test apis with authentication, a test user needs to be created so that we can make requests in context of that user. Letâ€™s create a test user. Change your tests to

```python
from django.contrib.auth import get_user_model
# ...

class TestPoll(APITestCase):
    def setUp(self):
        # ...
        self.user = self.setup_user()

    @staticmethod
    def setup_user():
        User = get_user_model()
        return User.objects.create_user(
            'test',
            email='testuser@test.com',
            password='test'
        )

    def test_list(self):
        request = self.factory.get(self.uri)
        request.user = self.user
        response = self.view(request)
        self.assertEqual(response.status_code, 200,
                         'Expected Response Code 200, received {0} instead.'
                         .format(response.status_code))
```

Now run the test command.

```
python manage.py test
```

You should get this response

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
-----------------------------------------------------------------
---
Ran 1 test in 0.119s
```

```
OK
Destroying test database for alias 'default'...
```

Using `APIClient`

The same test can be written using `APIClient`. It has `get`, `.post` and family. Unlike creating requests first, with `APIClient` you can GET ot POST to a url directly and get a response.

Add a test like this:

```python
from rest_framework.test import APIClient

# ...


class TestPoll(APITestCase):
    def setUp(self):
        self.client = APIClient()
        # ...

    # ...
    def test_list2(self):
        response = self.client.get(self.uri)
        self.assertEqual(response.status_code, 200,
                         'Expected Response Code 200, received {0} instead.'
                         .format(response.status_code))
```

Let us test it now.

```
python manage.py test polls.tests.TestPoll


Creating test database for alias 'default'...
System check identified no issues (0 silenced).
F
======================================================================
FAIL: test_list2 (polls.tests.TestPoll)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/shabda/repos/building-api-django/pollsapi/polls/tests.py", line 37, in test_list2
    .format(response.status_code))
AssertionError: 401 != 200 : Expected Response Code 200, received 401 instead.
```

```
----------------------------------------------------------------------
---
Ran 1 test in 0.136s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

We are seeing the same failure we saw in the test with `APIRequestFactory`. You can login a `APIClient` by calling `APIClient.login`. Lets update the test.

```python
class TestPoll(APITestCase):
    # ...

    def test_list2(self):
        self.client.login(username="test", password="test")
        response = self.client.get(self.uri)
        self.assertEqual(response.status_code, 200,
                        'Expected Response Code 200, received {0}
instead.'
                        .format(response.status_code))
```

```
 python manage.py test polls.tests.TestPoll
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
----------------------------------------------------------------------
---
Ran 1 test in 0.260s

OK
Destroying test database for alias 'default'...
```

VoilÃ ! The test passed successfully.

`.post` and create

We now know how to test our GET APIs. We can use the the `APIClient` with `.post` method this time.

Let us try creating a new poll by sending the â€˜questionâ€™, and â €˜created_byâ€™ parameters which are needs in the POST method. The test function looks as follows.

```python
class TestPoll(APITestCase):

    # ...
    def test_create(self):
        self.client.login(username="test", password="test")
        params = {
            "question": "How are you?",
            "created_by": 1
            }
        response = self.client.post(self.uri, params)
        self.assertEqual(response.status_code, 201,
                        'Expected Response Code 201, received {0}
instead.'
                        .format(response.status_code))
```

We are asserting that the the http code is 201 if the test passes succesfully. Lets run the tests.

```
 python manage.py test polls.tests.TestPoll.test_create

Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
-------------------------------------------------------------------
---
Ran 1 test in 0.267s

OK
Destroying test database for alias 'default'...
```

Time to celebrate with the API :)

Continuous integration with CircleCI

We have the tests, but we also want it to run on every commit. If you are using Github, CircleCI provides a very well in integrated service to run your tests. We will use Circleci. v2

We can configure our application to use Circle CI by adding a file named `.circleci/circle.yml` which is a YAML(a human-readable data serialization format) text file. It automatically detects when a commit has been made and pushed to a Github repository that is using CircleCI, and each time this happens, it will try to build the project and runs tests. The build failure or success is notified to the developer.

Setting up CircleCI

- Sign-in: To get started with Circle CI we can sign-in with our github account on circleci.com.
- Activate Github webhook: Once the Signup process gets completed we need to enable the service hook in the github profile page.
- Add circle.yml: We should add the yml file to the project.

Writing circle.yml file

In order for circle CI to build our project we need to tell the system a little bit about it. we will be needed to add a file named `.circleci/config.yml` to the root of our repository. We also need to create a `pollsapi/requirements.txt` to define our dependencies.

Add this to your `pollsapi/requirements.txt`

```
Django==2.0.3
djangorestframework==3.7.7
```

And then add this to `.circleci/config.yml`

```yaml
version: 2
jobs:
  build:
    docker:
      # specify the version you desire here
      - image: circleci/python:3.6.1


    working_directory: ~/repo

    steps:
      - checkout

      # Download and cache dependencies
      - restore_cache:
          keys:
          - v1-dependencies-{{ checksum "pollsapi/requirements.txt" }}
          # fallback to using the latest cache if no exact match is found
          - v1-dependencies-

      - run:
          name: install dependencies
          command: |
            python3 -m venv venv
            . venv/bin/activate
            pip install -r pollsapi/requirements.txt

      - save_cache:
          paths:
```

```
        - ./venv
      key: v1-dependencies-{{ checksum "requirements.txt" }}

  - run:
      name: run tests
      command: |
        . venv/bin/activate
        cd pollsapi
        python manage.py test

  - store_artifacts:
      path: test-reports
      destination: test-reports
```

Below are the important keywords that are used in writting circle.yml file.

- `image`: Defines the base image including the language and version to use
- `run`: It specifies a `command` which will be run to setup environent and run tests. `pip install -r pollsapi/requirements.txt` sets up the environment and `pip install -r pollsapi/requirements.txt`

If everything passed successfully, you should see a green checkmark

Congratulations, you have tests running in a CI environment.

From now onwards whenever we push our code to our repository a new build will be created for it and the tests will run.

We are at the end of the first part of our book. You can read the appendix, which tell about some documentation tools and api consumption tools. Go forward and build some amazing apps and apis.

Testing and Using API with Postman

In this chapter, weâ€™ll learn how to use the Postman app for testing our APIs.

Postman can be installed from [the Postman site](https://www.getpostman.com/) [https://www.getpostman.com/]. It is a versatile tool for working with APIs.

In this books, you will be creating and using APIs. Weâ€™ll see how we can make use of Postman for this.

Making HTTP request

Postman is pretty intutive, but the image below should make the app easy to understand.
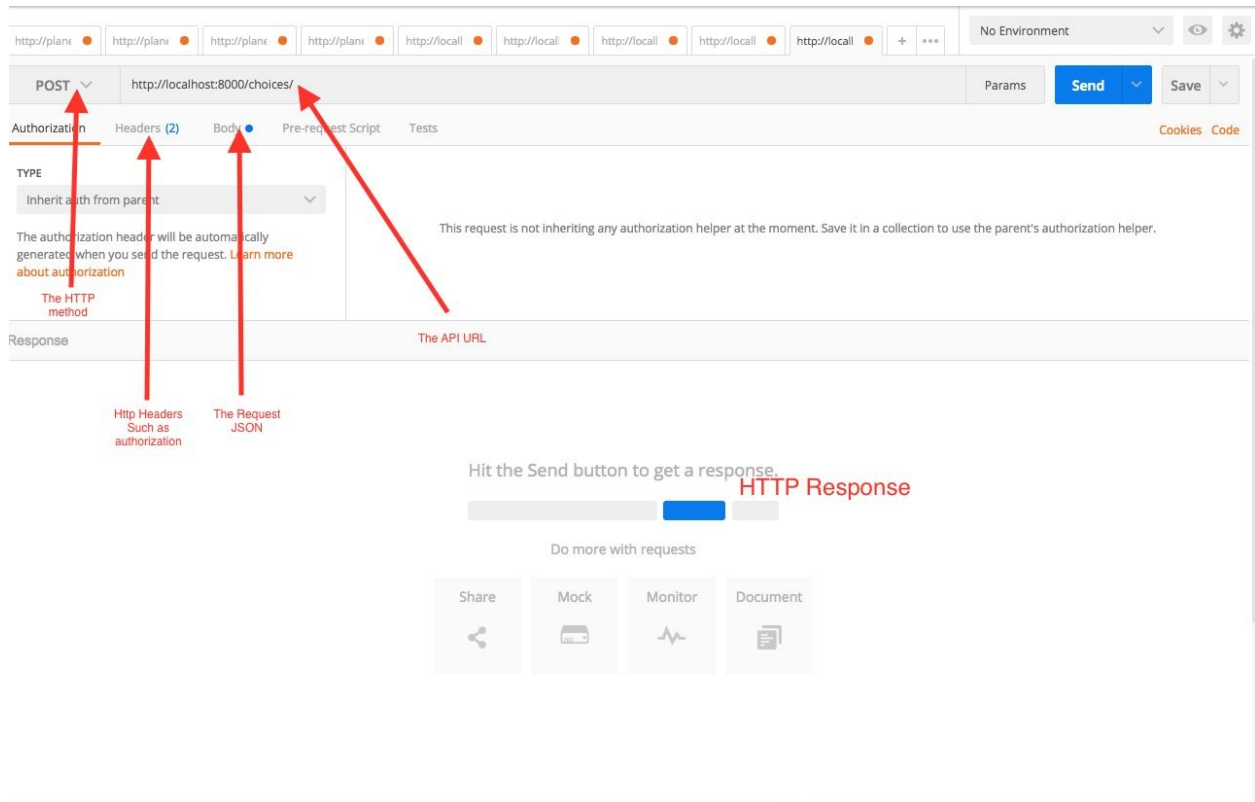
There are 4 key elements in making an HTTP request.

1. URL:
   This specifies to which URL we need to make a request for. In other terms where our API endpoint resides.

2. Method:
   Each API endpoint has a method which serves itâ€™s purpose. The methods for eg., can be GET for retrieving some data, POST for creating or updating, DELETE for deleting a record.

3. Headers:
   Headers provide required information about the request or the response or about the object sent in the body. Some times we use authentication

headers too, in order to access the API endpoint.

4. Body:

    The request body is where we send the object. The object which may be required for the service.

Response

Response is available in the bottom section, usually in a JSON format, but may also vary depending up on the API service.

Collections

We can save all the relative API endpoints to collections. In our example, we can save all our polls related endpoints as a collection or all the users related endpoints as another collection. This way all the APIs are organized.

Authentication

Postman also supports few authentication mechanisms like Basic Auth, Digest Auth and Oauth1. This allows us to use these authentication methods for the APIs.

Documenting APIs (with Swagger and more)

In this chapter we will see how to document our API.

As you build your API, you would need to document the API to collaborate with other people. In most companies and teams, the developer using the API is different from the one building them. API documentation and collaboration tools, become even more important in such an environment.

Swagger is a tool used to understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. In simple terms, with swagger you can see what all API end points are available for a web application. You can use swagger for testing the requests and responses of the API endpoints.

DRF comes with its own tool, coreapi, for documenting and interacting with the API.

We will use both coreapi and swagger to document our API.

Adding swagger documentation

Install django-rest-swagger

```
pip install django-rest-swagger
```

Update your `settings.py`

```python
INSTALLED_APPS = [
    # ...
    'polls',
    'rest_framework_swagger',
]
```

Add swagger to your urls.

```python
from rest_framework_swagger.views import get_swagger_view

schema_view = get_swagger_view(title='Polls API')

# ...
urlpatterns = [
    # ...
    path(r'swagger-docs/', schema_view),
]
```

Navigate to */swagger-docs/*. And your swagger docs are ready in all their glory.

Polls API

login   Show/Hide | List Operations | Expand Operations

| POST | /login/ |

polls   Show/Hide | List Operations | Expand Operations

GET	/polls/
POST	/polls/
DELETE	/polls/{id}/
GET	/polls/{id}/
PATCH	/polls/{id}/
PUT	/polls/{id}/
GET	/polls/{id}/choices/
POST	/polls/{id}/choices/
POST	/polls/{id}/choices/{choice_pk}/vote/

users   Show/Hide | List Operations | Expand Operations

| POST | /users/ |

[BASE URL:]

Using coreapi for documentation

Install coreapi

```
pip install coreapi
```

Add coreapi urls to your urls.

```python
from rest_framework.documentation import include_docs_urls
# ...

urlpatterns = [
    # ...
    path(r'docs/', include_docs_urls(title='Polls API')),
]
```

And your coreapi docs are ready in all their glory.

Polls API

Sidebar Navigation

- ⊙ **login** ⌄
  - › create
- ⊙ **polls** ⌄
  - › list
  - › create
  - › read
  - › update
  - › partial_update
  - › delete
  - › choices > list
  - › choices > create
  - › choices > vote > create
- ⊙ **users** ⌄
  - › create

Polls API

login

create   ⇄ INTERACT
`POST`   `/login/`

polls

list   ⇄ INTERACT
`GET`   `/polls/`

create   ⇄ INTERACT
`POST`   `/polls/`

Request Body

The request body should be a `"application/json"` encoded object, containing the following items.

Parameter	Description
question `required`	
created_by `required`	

Index