## Part 1 – CRUD

If you’re reading this, you probably already have at least the application’s wireframe, and you want to add a REST API to enable basic operations on objects such as Create, Retrieve, Update and Delete (CRUD). That’s what we will do in this article.

For starters, let’s prepare an example application for managing things we lend to our friends. Then, we need to install Django REST Framework.

|  |  |
| --- | --- |
| 1 2 3 | ; cd your project’s directory and activate its virtualenv $ ./manage.py startapp rental $ pip install djangorestframework |

Next, we need to modify the INSTALLED\_APPS parameter in settings.py file.

settings.py

|  |  |
| --- | --- |
| 1 2 3 4 5 6 | INSTALLED\_APPS = [     *# previous apps*      'rental',     'rest\_framework', ] |

rental/models.py

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | from django.db import models  class Friend(models.Model):     name = models.CharField(max\_length=100)  class Belonging(models.Model):     name = models.CharField(max\_length=100)  class Borrowed(models.Model):     what = models.ForeignKey(Belonging, on\_delete=models.CASCADE)     to\_who = models.ForeignKey(Friend, on\_delete=models.CASCADE)     when = models.DateTimeField(auto\_now\_add=True)     returned = models.DateTimeField(null=True, blank=True) |

To make our objects available through the API, we need to perform a serialization – reflect the data contained in the object textually. The default format here is JSON, although DRF allows serialization to XML or YAML. The reverse process is called deserialization.

Both processes are defined in objects referred to as **serializers**. DRF offers developers with a convenient class to create serializers for Django models easily, so we have to provide only some basic information such as the model that will be served in the serializer and the fields to which we want to give access.

rental/serializers.py

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 | from rest\_framework import serializers from . import models  class FriendSerializer(serializers.ModelSerializer):     class Meta:         model = models.Friend         fields = ('id', 'name')  class BelongingSerializer(serializers.ModelSerializer):     class Meta:         model = models.Belonging         fields = ('id', 'name')  class BorrowedSerializer(serializers.ModelSerializer):     class Meta:         model = models.Borrowed         fields = ('id', 'what', 'to\_who', 'when', 'returned') |

Now we need to create views that will handle each of the operations we want to perform on our objects.

Consider for a moment their names and how to combine them with methods available in the HTTP standard.

**Create**– that one is rather straightforward. Standard support for it comes from the HTTP POST method. Because we’re creating a set element here (in particular, the object’s ID will be only determined now), we treat this method as an operation on the list: creating an element.

**Retrieve**– we have two options here: we can download a list of objects of a given type (list) or one specific object (retrieve). In both cases, GET will be the adequate HTTP method.

**Update**– There are two HTTP methods available here: PUT and PATCH. The difference between them is that according to its definition, PUT requires all attributes of the object – including those that have not changed. PATCH, on the other hand, allows entering only those fields that have actually changed, which is why it’s more popular. Using the PUT or PATCH method to update multiple objects is rare and DRF only supports updating single object in its default CRUD.

**Delete**– this deletes one or many objects. The HTTP method here will be DELETE. In practice, for security reasons, it’s usually not possible to remove several objects at the same time and again, DRF only supports this operation on single objects in its default CRUD. .

Let’s summarize all of the above:

|  |  |  |
| --- | --- | --- |
| **Operation** | **HTTP method** | **endpoint type** |
| Create | POST | list |
| Retrieve many | GET | list |
| Retrieve one | GET | detail |
| Update | PUT / PATCH | detail |
| Delete | DELETE | detail |

To support such a set of operations, DRF provides a handy tool – ViewSet. It takes the idea behind the standard class-based views from Django to a higher level. What it does is packing the above set into one class with the automatic creation of appropriate URL paths.

So let’s see how that looks like in practice:

To start, let’s create a ViewSet that will support our models. DRF provides the ModelViewSet thanks to which the required amount of code is reduced to the minimum:

rental/api\_views.py

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | from rest\_framework import viewsets from . import models from . import serializers  class FriendViewset(viewsets.ModelViewSet):     queryset = models.Friend.objects.all()     serializer\_class = serializers.FriendSerializer  class BelongingViewset(viewset.ModelViewSet):     queryset = models.Belonging.objects.all()     serializer\_class = serializers.BelongingSerializer  class BorrowedViewset(viewsets.ModelViewSet):     queryset = models.Borrowed.objects.all()     serializer\_class = serializers.BorrowedSerializer |

Then there’s the last part – connecting all this to the URL tree of our project. And here we get a very convenient tool as well – routers. DRF provides two of the most important classes that differ only in that one of them shows the API structure when downloading / (root), and the other doesn’t.

Our viewsets will be hooked up as follows:

api.py (global, next to settings.py)

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 | from rest\_framework import routers from core import views as myapp\_views  router = routers.DefaultRouter() router.register(r'friends', myapp\_views.FriendViewset) router.register(r'belongings', myapp\_views.BelongingViewset) router.register(r'borrowings', myapp\_views.BorrowedViewset) |

urls.py (global)

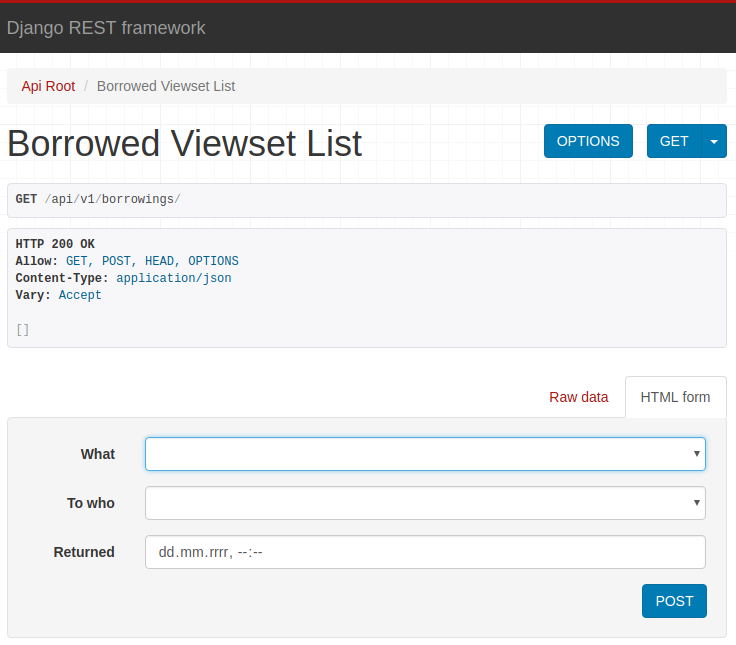
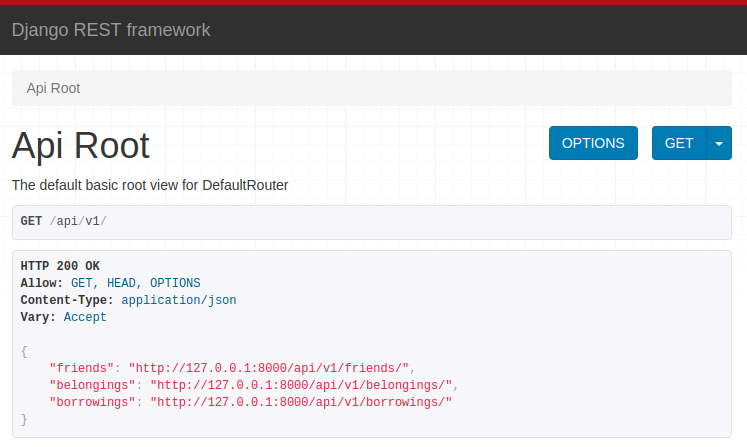
|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | from django.urls import include, path from django.contrib import admin from .api import router  urlpatterns = [     path('admin/', admin.site.urls),     path('api/v1/', include(router.urls)), ] |

Let’s test the API we created.

|  |  |
| --- | --- |
| 1 | $ ./manage.py makemigrations $ ./manage.py migrate $ ./manage.py runserver |

**Open this address in your browser (in standard configuration): https://127.0.0.1:8000/api/v1/**

DRF automatically creates views that allow performing API queries from the browser level:



Experiment and check the effects in the django-admin panel.

## Conclusion

At this point, we receive a ready API that supports CRUD for our models. Please note that we don’t have any security against unauthorized access here yet.

We will deal with the user login and registration process in the next article in this series.

Be sure to catch up with the work we’ve completed in other parts of the series:

* [Django REST Framework Tutorial – CRUD](https://sunscrapers.com/blog/ultimate-tutorial-django-rest-framework-part-1/)
* [Django REST Framework Tutorial – Login and Authentiction](https://sunscrapers.com/blog/django-rest-framework-login-and-authentication/)
* [Django REST Framework Tutorial – Custom Fields](https://sunscrapers.com/blog/django-rest-framework-tutorial-part-3-custom-fields/)
* [Django REST Framework Tutorial – Pagination](https://sunscrapers.com/blog/the-ultimate-tutorial-for-django-rest-framework-pagination-part-4/)
* [Django REST Framework Tutorial – Filters and Filtering](https://sunscrapers.com/blog/the-ultimate-tutorial-for-django-rest-framework-filtering-part-5/)
* [Django REST Framework Tutorial – Functional Endpoints and API Nesting](https://sunscrapers.com/blog/the-ultimate-tutorial-for-django-rest-framework-functional-endpoints-and-api-nesting-part-6/)
* [Django REST Framework Tutorial – Selective Fields and Related Objects](https://sunscrapers.com/blog/the-ultimate-tutorial-for-django-rest-framework-selective-fields-and-related-objects-part-7/)