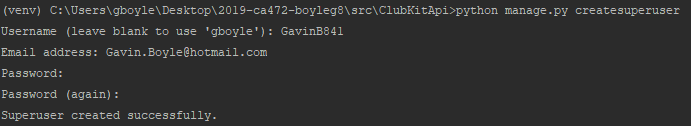
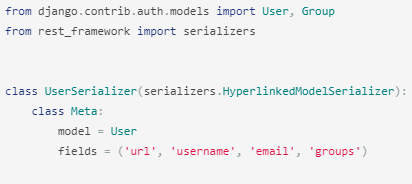
**Create Super Users & Test**

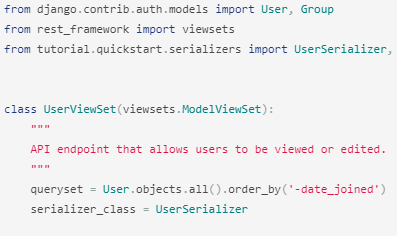
* Sync database for the first time:
  + *Python manage.py migrate*
* Create super user:
  + *Python manage.py createsuperuser*



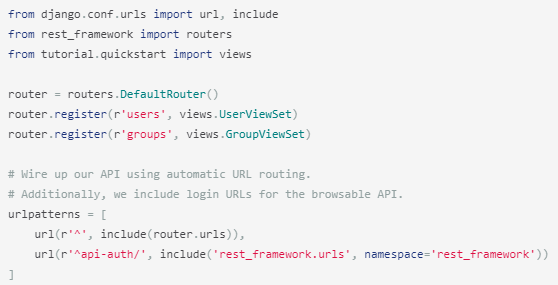
* Create serializers.py within api directory:
  + Add the following code:



* Add the following into views.py



* Add the following to urls.py



* Add to settings.py:

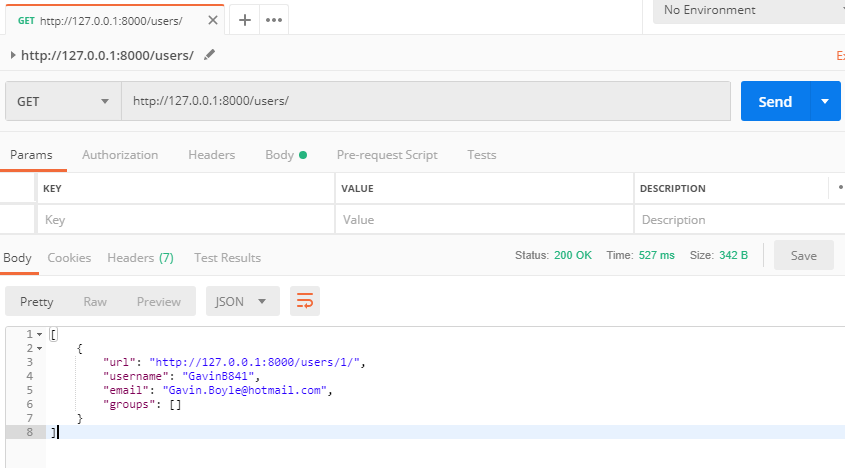


**Test**

Run server: *python manage.py runserver*

Open postman :

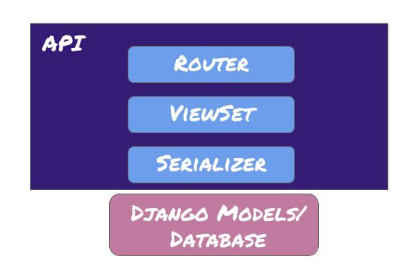
GET: [*http://127.0.0.1:8000/users/*](http://127.0.0.1:8000/users/)



### Basic Architecture

A DRF API is composed of 3 layers: the serializer, the viewset, and the router.

* **Serializer:** converts the information stored in the database and defined by the [Django models](https://docs.djangoproject.com/en/2.0/topics/db/models/) into a format which is more easily transmitted via an API
* **Viewset**: defines the functions (read, create, update, delete) which will be available via the API
* **Router**: defines the URLs which will provide access to each viewset



**Serializers**

Django models intuitively represent data stored in your database, but an API will need to transmit information in a less complex structure. While your data will be represented as instances of your Model classes in your Python code, it needs to be translated into a format like JSON in order to be communicated over an API.

The DRF serializer handles this translation. When a user submits information (such as creating a new instance) through the API, the serializer takes the data, validates it, and converts it into something Django can slot into a Model instance. Similarly, when a user accesses information via the API the relevant instances are fed into the serializer, which parses them into a format that can easily be fed out as JSON to the user.

The most common form that a DRF serializer will take is one that is tied directly to a Django model:

class ThingSerializer(serializers.ModelSerializer):

class Meta:

model = Thing

fields = (‘name’, )

Setting fields allows you to specify exactly which fields are accessible using this serializer. Alternatively, exclude can be set instead of fields, which will include all of the model’s fields except those listed in exclude.

Serializers are an incredibly flexible and powerful component of DRF. While attaching a serializer to a model is the most common use, serializers can be used to make any kind of Python data structure available via the API according to defined parameters.

### ViewSets

A given serializer will parse information in both directions (reads and writes), but the ViewSet is where the available operations are defined. The most common ViewSet is the ModelViewSet, which has the following built-in operations:

* Create an instance: create()
* Retrieve/Read an instance: retrieve()
* Update an instance (all fields or only selected fields): update() or partial\_update()
* Destroy/Delete an instance: destroy()
* List instances (paginated by default): list()

Each of these associated functions can be overwritten if different behavior is desired, but the standard functionality works with minimal code, as follows:

class ThingViewSet(viewsets.ModelViewSet):

queryset = Thing.objects.all()

serializer\_class = ThingSerializer

If you need more customization, you can use generic viewsets instead of the ModelViewSet or even individual custom views.

### Routers

Finally, the router provides the surface layer of your API. To avoid creating endless “list”, “detail” and “edit” URLs, the DRF routers bundle all the URLs needed for a given viewset into one line per viewset, like so:

# Initialize the DRF router; only once per urls.py file from rest\_framework import routers`

router = routers.DefaultRouter()

# Register the viewset

router.register(r'thing', main\_api.ThingViewSet)

Then, all of the viewsets you registered with the router can be added to the usual url\_patterns:

url\_patterns += url(r'^', include(router.urls))

And you’re up and running! Your API can now be accessed just like any of your other django pages. Next, you’ll want to make sure people can find out how to use it.

**Ref**

<https://www.django-rest-framework.org/tutorial/quickstart/>

<https://www.caktusgroup.com/blog/2018/02/26/basics-django-rest-framework/>