**DJANGO REST FRAMEWORK LESSONS**

1. **Serialization**

Pip install Django

Pip install Djangorestframework

Pip install pygments

Pip install mysqlclient

Create a project: django-admin startproject tutorial

cd and then create an app that we'll use to create a simple Web API:

**python manage.py startapp snippets**

Add snippets app and the rest\_framework app to INSTALLED\_APPS. Edit the tutorial/settings.py

Create a snippets model to store snippets. Edit snippets/models.py:



Edit settings.py:

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.mysql',

'NAME': 'djangotutorial', #database name

'HOST':'localhost',

'USER':'root',

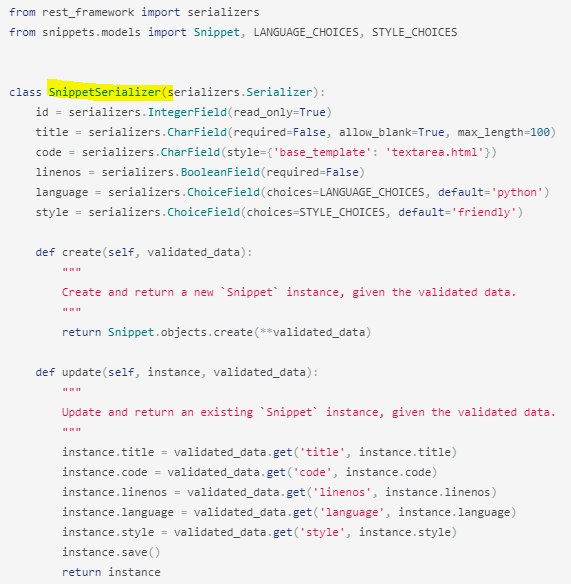
'PORT':'3306',

'PASSWORD':'root'

}

}

Now create a Serializer class to provide a way of serializing and deserializing the snippet instances into representations such as json. Create a file in the snippets directory named serializers.py and add the following:



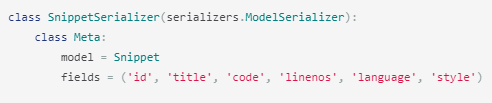
Make initial migrations:

Python manage.py makemigrations snippets

Python manage.py migrate

1. **Using Model Serializers**

Our SnippetSerializer class is replicating a lot of information that's also contained in the Snippet model. REST framework includes both Serializerclasses, and ModelSerializer classes. ModelSerializer class has default implementations for create and save methods.

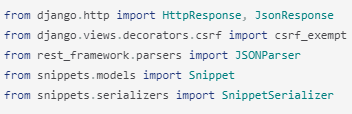


Try following using python manage.py shell:

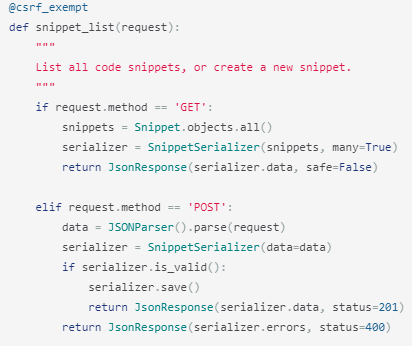


1. **Writing regular Django views**

Edit snippets/views.py:



Using snippets list API to list all snippets or create new snippet:

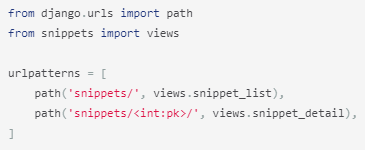


To be able to POST to this view from clients that won't have a CSRF token we need to mark the view as csrf\_exempt. Another view which corresponds to individual snippets (retrieve, update, delete):

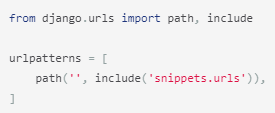


1. **Write the urls**

Create snippets/urls.py:



We also need to wire up the root urlconf, in the tutorial/urls.py file, to include our snippet app's URLs.



1. **Testing the APIs**

python manage.py runserver

Open postman or pip install httpie

Hit: <http://127.0.0.1:8000/snippets/> or <http://127.0.0.1:8000/snippets/>2/

1. **Requests and Responses**

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 like request.POST, but more useful for working with Web APIs.



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.



1. **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.



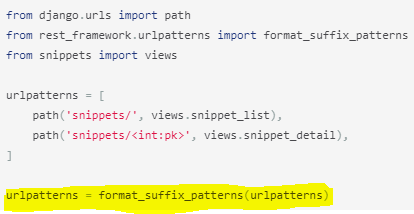


 request.data can handle incoming json requests, but it can also handle other formats. Similarly, we're returning response objects with data, but allowing REST framework to render the response into the correct content type for us.

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>. Add a format keyword argument to both views:

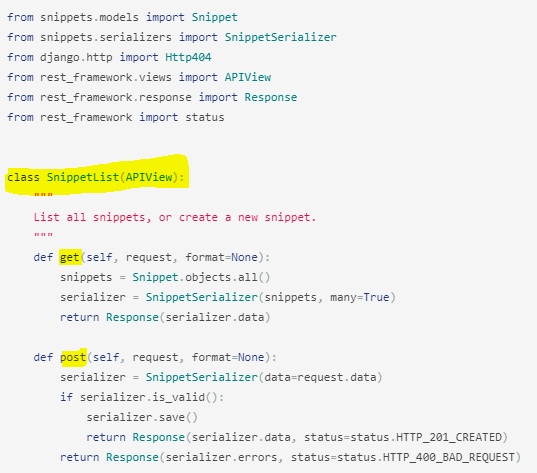


Append a set of format\_suffix\_patterns in addition to the existing URLs in snippets/urls.py



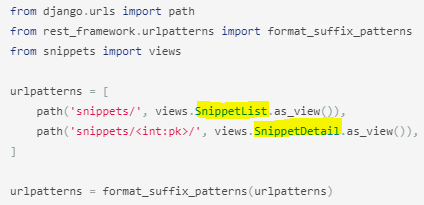
1. **Class based views**

Edit Snippets/views.py





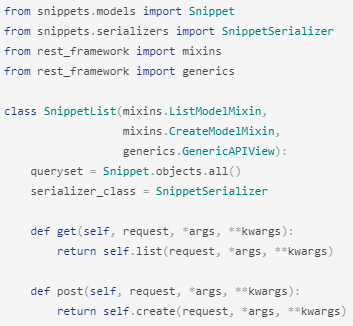
Edit snippets/urls.py:



APIs still work same as before.

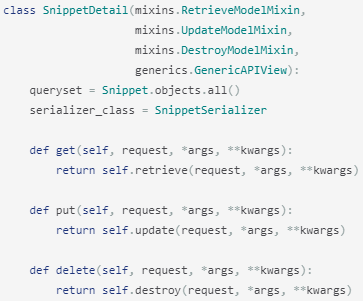
1. **Using Mixins**

The create/retrieve/update/delete operations that we've been using so far are going to be similar for any model-backed API views we create. Those bits of common behavior are implemented in REST framework's mixin classes. Edit views.py file:



We're building our view using GenericAPIView, and adding in ListModelMixin and CreateModelMixin.

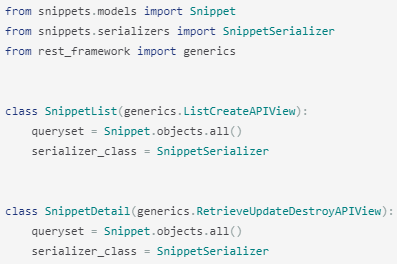
The base class provides the core functionality, and the mixin classes provide the .list() and .create() actions. We're then explicitly binding the get and post methods to the appropriate actions.



we're using the GenericAPIView class to provide the core functionality, and adding in mixins to provide the .retrieve(), .update() and .destroy() actions.

1. **Using generic class-based views**

REST framework provides a set of already mixed-in generic views that we can use to trim down our views.py.



1. **User Authentication and permissions**
2. **Fdffwfww**
3. **Fdffwfww**