**Django Rest Framework - Blog API**

In this tutorial we will build the API for a *blog app* featuring CRUD (Create-Read-Update-Delete) functionality with [Django Rest Framework](http://www.django-rest-framework.org/).

It’s helpful–but not required–to have previous experience with [Django](https://www.djangoproject.com/).

**SPAs**

Modern web applications are increasingly built as Single-Page-Applications (SPAs) which feature a distinct frontend and backend. As opposed to the traditional monolith approach in Django itself–and other web framework like Ruby on Rails–SPAs require a backend API that can then be consumed by multiple frontends as needed. This approach works well when a company needs multiple frontend applications–mobile web, iOS, Android–to run off the same database. It is also arguably more “future-proof” since the frontend can change to whatever the flavor-of-the-month JavaScript framework is, but the backend remains stable the whole time.

The downside is that it takes more time and code to create a separate frontend/backend for projects.

**RESTful APIs**

An API (Application Programming Interface) provides an interface for developers to interact with an application’s database. Instead of just giving someone full access to a database, an API sets up rules, permissions, and endpoints for various functionality: login, logout, reading a list of blogs, individual blog details, and so on.

The traditional way to construct a web API is via REST (Representational State Transfer), a well-established architecture for how websites can communicate with one another. Since computers communicate via the web this means using the HTTP protocol which supports a number of common “methods” (also called “verbs”) such as GET, PUT, POST, and DELETE.

There are also a host of related access codes that indicate whether a request was successful (200), redirected (301), missing (404), or worse (500).

**JSON**

It’s important to note that since an API is communicating with another computer the information being shared **is not** what would be sent for a standard web page. When your browser requests, for example, the Google homepage, it sends HTTP requests and receives HTTP responses with HTML, CSS, JavaScript, images, and so on.

An API is different. Typically we’re **only interested** in the data from a database. This data is often transformed into JSON format to be efficiently transmitted about. The API will also have a series of well-defined rules for how a frontend client can interact with it via a REST architecture. To register a new user the frontend framework will need access an API endpoint called, for example, /api/register. This API endpoint contains both a specific URL route and its own set of permissions.

**Setup**

We will start the tutorial by building the models for a blog app in Django. Then we can add Django Rest Framework to transform it into a RESTful API.

First create a new directory on your computer for our code. I’ll place it on the Desktop in a folder called api but you can place it anywhere. Then configure our project.

$ cd ~/Desktop

$ mkdir blogapi && cd blogapi

$ pip install django

(blogapi) $ django-admin.exe startproject blog\_project .

(blogapi) $ python manage.py startapp posts

We’re using Pipenv to install Django and activate a local environment. Then we create a new Django project called blogapi as well as our first app posts.

Since we’ve added a new app we need to tell Django about it. So make sure to add posts to our list of INSTALLED\_APPS in the settings.py file.

*# blog\_project/settings.py*

INSTALLED\_APPS **=** [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'posts',

]

Our database model will be deliberately quite basic. Let’s create four fields: title, content, created\_at, and updated\_at.

*# posts/models.py*

from django.db import models

**class** **Post**(models.Model):

title **=** models.CharField(max\_length**=**50)

content **=** models.TextField()

created\_at **=** models.DateTimeField(auto\_now\_add**=**True)

updated\_at **=** models.DateTimeField(auto\_now**=**True)

**def** **\_\_str\_\_**(self):

**return** self.title

Note that we’re also defining what the \_\_str\_\_ representation of the model should be which is a Django best practice.

Now update our database by first creating a new migration file and then applying it.

(blogapi) $ python manage.py makemigrations

(blogapi) $ python manage.py migrate

Good! We want to view our data in Django’s excellent built-in admin app so let’s add Post to it as follows.

*# posts/admin.py*

from django.contrib import admin

**from** . models import Post

admin.site.register(Post)

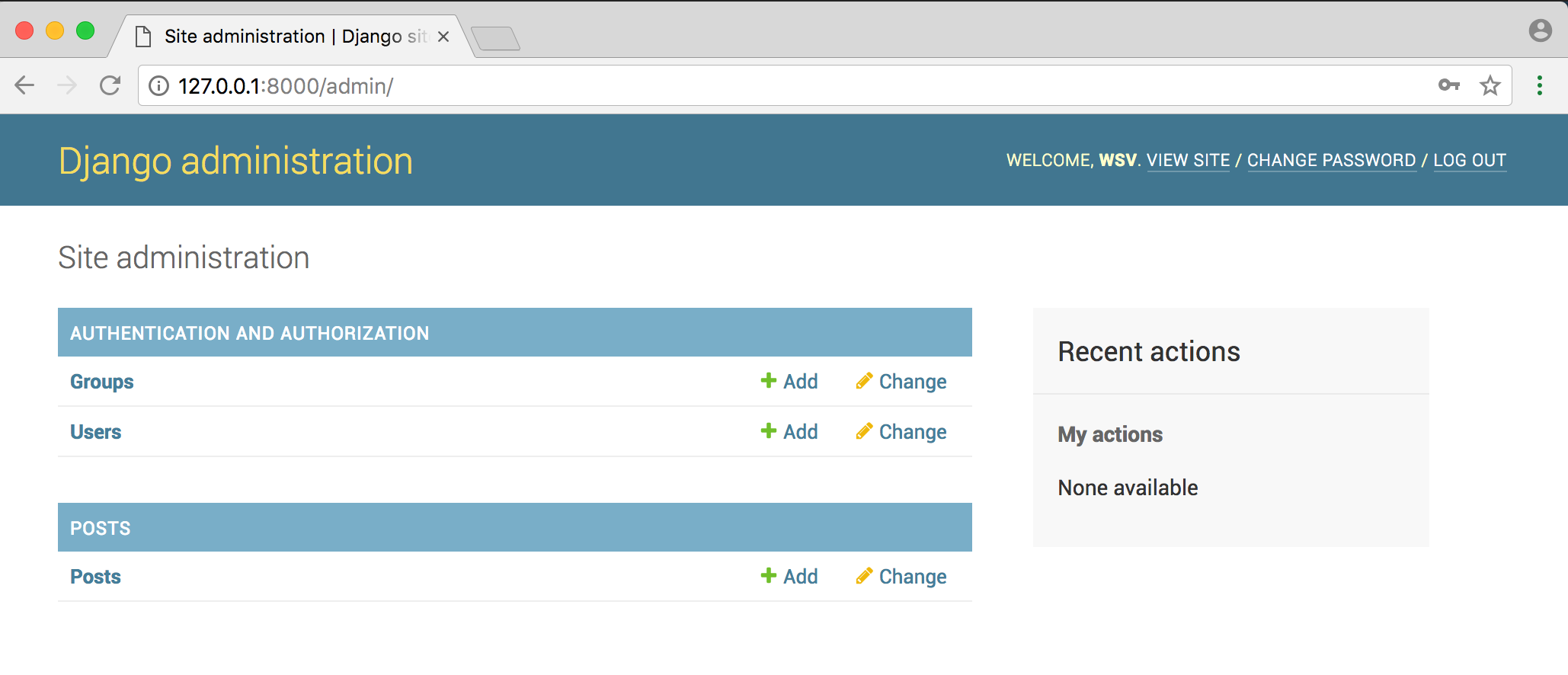
Then create a superuser account so we can login. Type the command below and enter all the prompts.

(blogapi) $ python manage.py createsuperuser

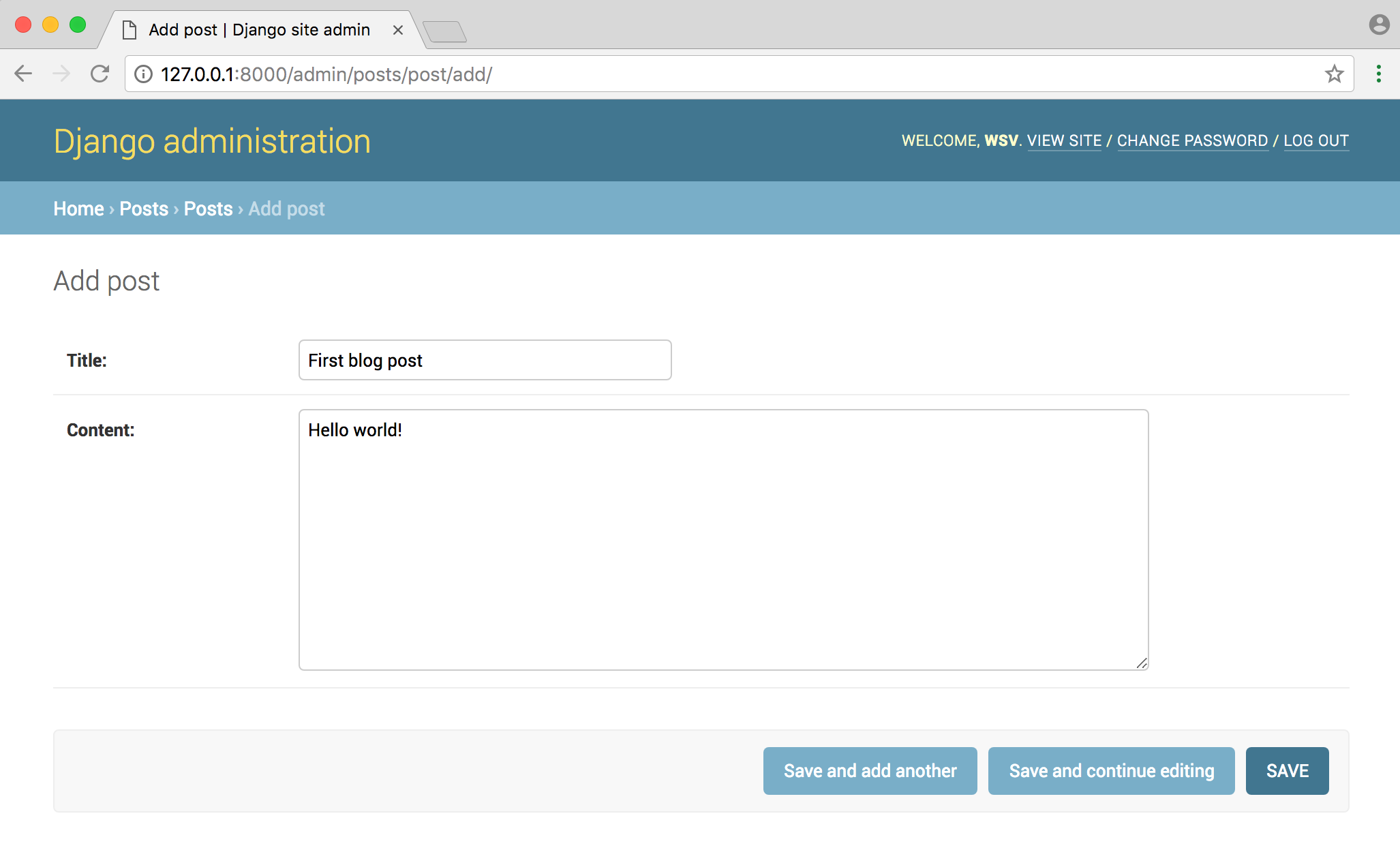
Now we can start up the local web server.

(blogapi) $ python manage.py runserver

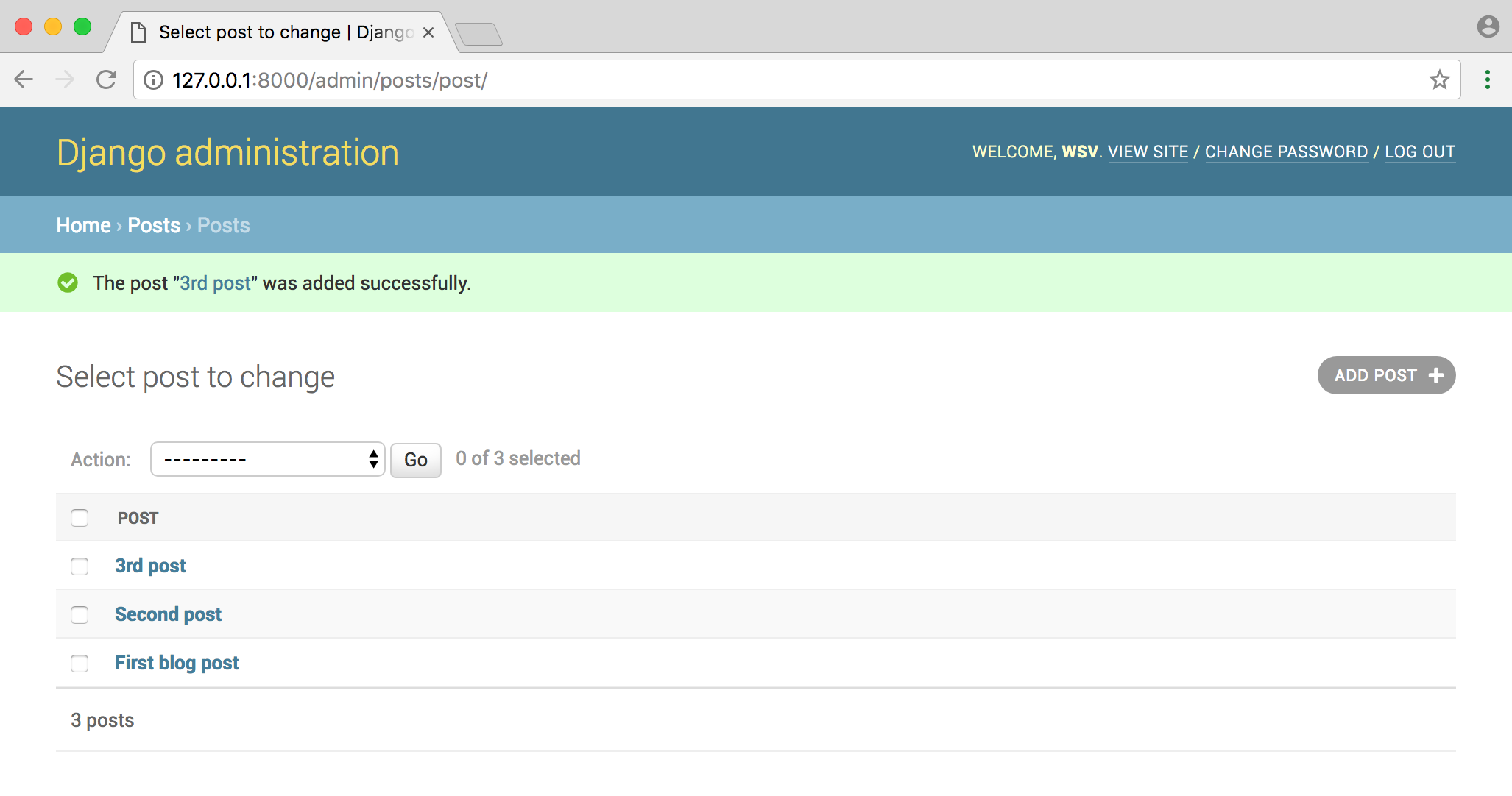
Navigate to <localhost:8000/admin> and login with your superuser credentials.



Click on “+ Add” button next to Posts and enter in some new content.



I’ve created three new blog posts that you can see here.



And we’re done with the Django part! Since we’re creating an API we will not need to create templates and views. Instead it’s time to add Django Rest Framework to take care of transforming our model data into an API.

**Django Rest Framework**

DRF takes care of the heavy lifting of transforming our database models into a RESTful API. There are two main steps to this process: first a serializer is used to transform the data into JSON so it can be sent over the internet, then a View is used to define what data is sent.

Let’s see it in action. First stop the local server with Control+c and use Pipenv to install Django Rest Framework.

(blogapi) $ pip install djangorestframework==3.10

Then add it to the INSTALLED\_APPS section of our settings.py file.

*# blog\_project/settings.py*

INSTALLED\_APPS **=** [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'rest\_framework',

'posts',

]

I like to add a space between third-party apps like rest\_framework and my own apps like posts.

Now create a new serializers.py file in our posts app.

(blogapi) $ touch posts/serializers.py

Remember that the serializer is used to convert our data into JSON format. That’s it. Here’s what it looks like.

*# posts/serializers.py*

from rest\_framework import serializers

from . import models

**class** **PostSerializer**(serializers.ModelSerializer):

**class** **Meta**:

fields **=** ('id', 'title', 'content', 'created\_at', 'updated\_at',)

model **=** models.Post

On the top two files we’re importing serializers from DRF and our models. Next we create a serializer class and create a Meta class within it. The fields controls which database attributes are available. In this case we’re exposing all our fields including id which is the primary key Django automatically adds to all database records.

Next we need to create our views. Just as Django has generic class based views, so too DRF has [generic views](http://www.django-rest-framework.org/api-guide/generic-views/) we can use. Let’s add a view to list all blog posts and a detail view for a specific post.

Update the views.py file in posts as follows.

*# posts/views.py*

from rest\_framework import generics

from .models import Post

from .serializers import PostSerializer

**class** **PostList**(generics.ListAPIView):

queryset **=** Post.objects.all()

serializer\_class **=** PostSerializer

**class** **PostDetail**(generics.RetrieveAPIView):

queryset **=** Post.objects.all()

serializer\_class **=** PostSerializer

At the top we import generics from DRF as well as our models and serializers files. Then we create two views: PostList and PostDetail. Both are just for GETs however RetrieveAPIView is for a **single instance** of a model. The complete list of [generic views](http://www.django-rest-framework.org/api-guide/generic-views/#generic-views) is on the DRF site.

The final piece is urls. We need to create the url routes–known as **endpoints** in an API–where the data is available.

Start at the project-level urls.py file.

*# blog\_project/urls.py*

from django.contrib import admin

from django.urls import include, path

urlpatterns **=** [

path('admin/', admin.site.urls),

path('api/', include('posts.urls')),

]

We’ve added include to the second line of imports and then created a path called api/ for our posts app.

Next create our posts app urls.py file.

(blogapi) $ touch posts/urls.py

And then include the code below.

*# posts/urls.py*

from django.urls import path

from . import views

urlpatterns **=** [

path('', views.PostList.as\_view()),

path('<int:pk>/', views.PostDetail.as\_view()),

]

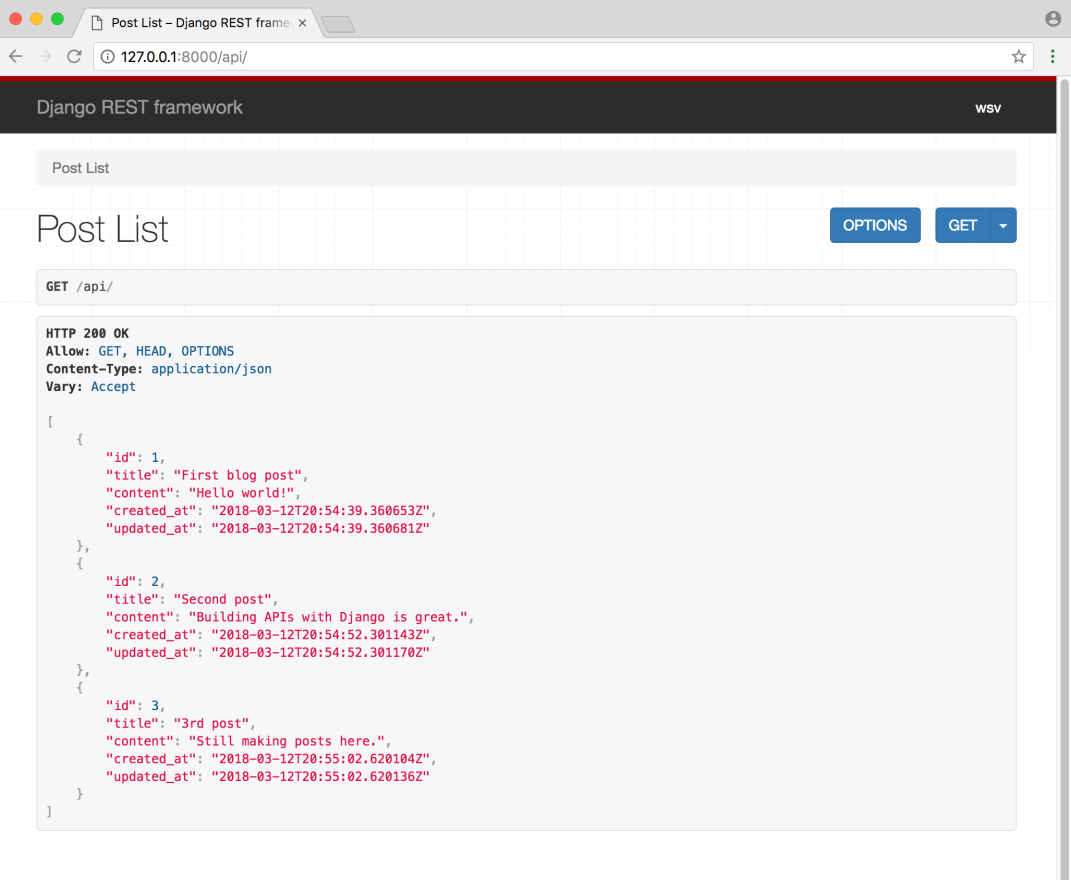
All blog routes will be at api/ so our PostList which has the empty string '' will be at api/ and postDetail at api/# where # represents the primary key of the entry. For example, the first blog post has a primary id of 1 so it will be at the route api/1, the second post at api/2, and so on.

**Browsable API**

Time to view our work and check out a DRF killer feature. Start up the server.

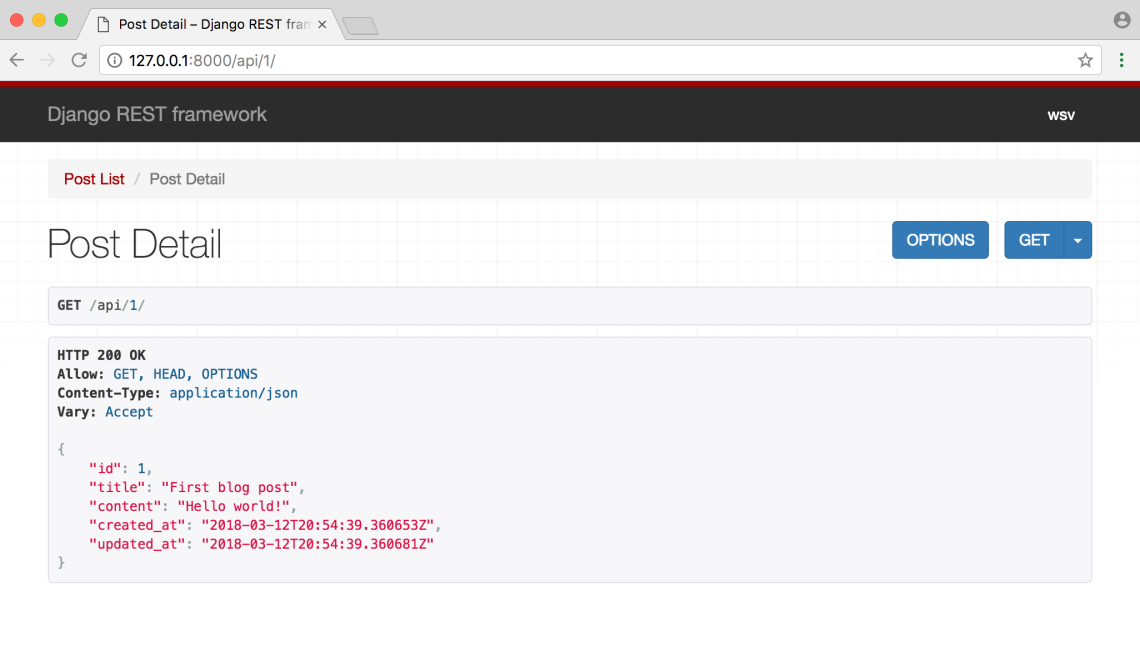
(blogapi) $ python manage.py runserver

Then go to <http://127.0.0.1:8000/api/>.



Check out that! The api/ endpoint displays all three of my blog posts in JSON format. It also shows in the header that only GET, HEAD, OPTIONS are allowed. No POSTing of data.

Now go to <http://127.0.0.1:8000/api/1/> and you’ll see only the data for the first entry.



**Implementing CRUD**

I promised at the beginning of the article that this tutorial would cover not just reading/getting content but the full CRUD syntax. Watch how easy DRF makes it to transform our API into one that supports CRUD!

Open up the posts/views.py file and change class PostDetail(generics.RetrieveAPIView) to class PostDetail(generics.RetrieveUpdateDestroyAPIView).

*# posts/views.py*

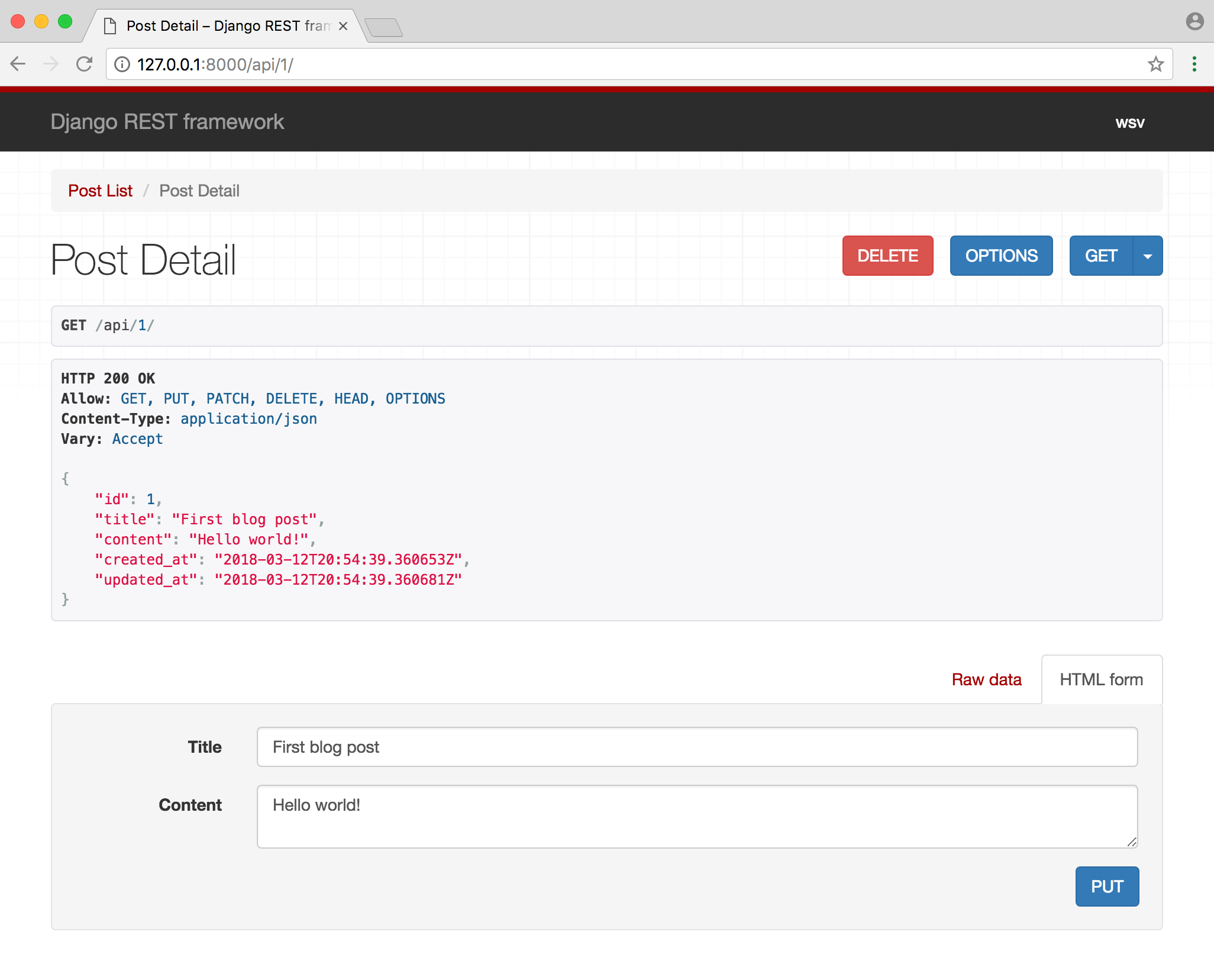
...

**class** **PostDetail**(generics.RetrieveUpdateDestroyAPIView):

queryset **=** models.Post.objects.all()

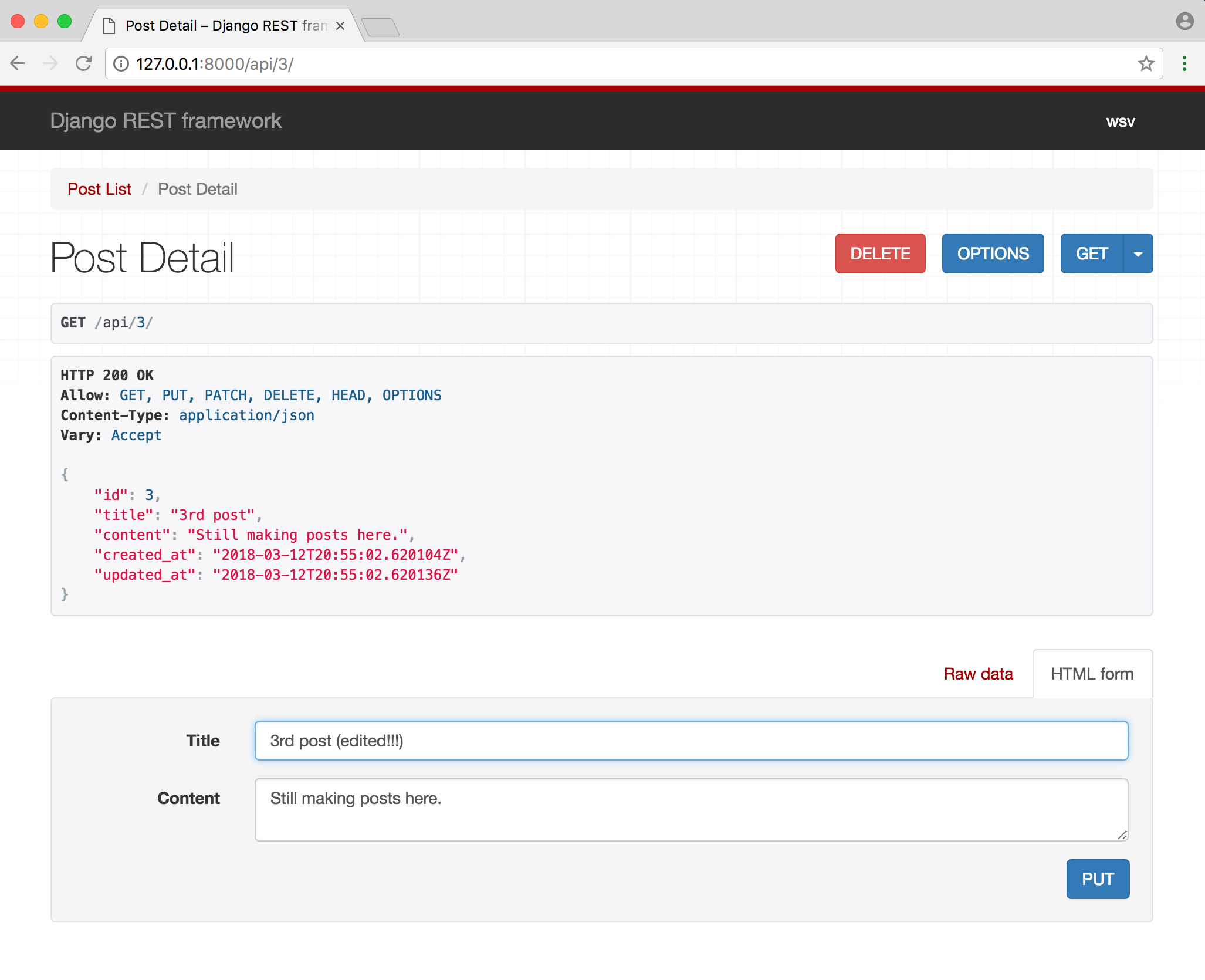
serializer\_class **=** serializers.PostSerializer

Now refresh the page at <http://127.0.0.1:8000/api/1/> and you can see updates in our graphical UI.



You can use the “Delete” button to delete content, “Put” to either update content, and “Get” to retrieve it as before. For example, navigate to the URL endpoint for our third post: <http://127.0.0.1:8000/api/3/>.

Then use the graphical interface for PUT at the bottom of the page to create a new entry.



And you can GET it after making the change:

