

Django Rest Framework

Table of Contents

Introduction

DRF setup

RESTful Structure

DRF Quick Start

- **Model Serializer**
- **Update Views**
- **Update URLs**
- **Test**

Refactor for REST

- **GET**
- **Datetime Format**
- **POST**
- **Author Format**
- **Delete**

We will learn to build a REST API in Django through a very simple method. Just follow the below steps and your first API will be ready to get going with minimum code.

Django REST Framework (DRF) allows developers to rapidly build RESTful APIs. We will see how DRF is actually used and also clear some very basics of the web. This article assumes though, that you have a basic understanding of the working of the web. We will learn topics like Django for APIs, RESTful web services and then build up to DRF.

What is an API?

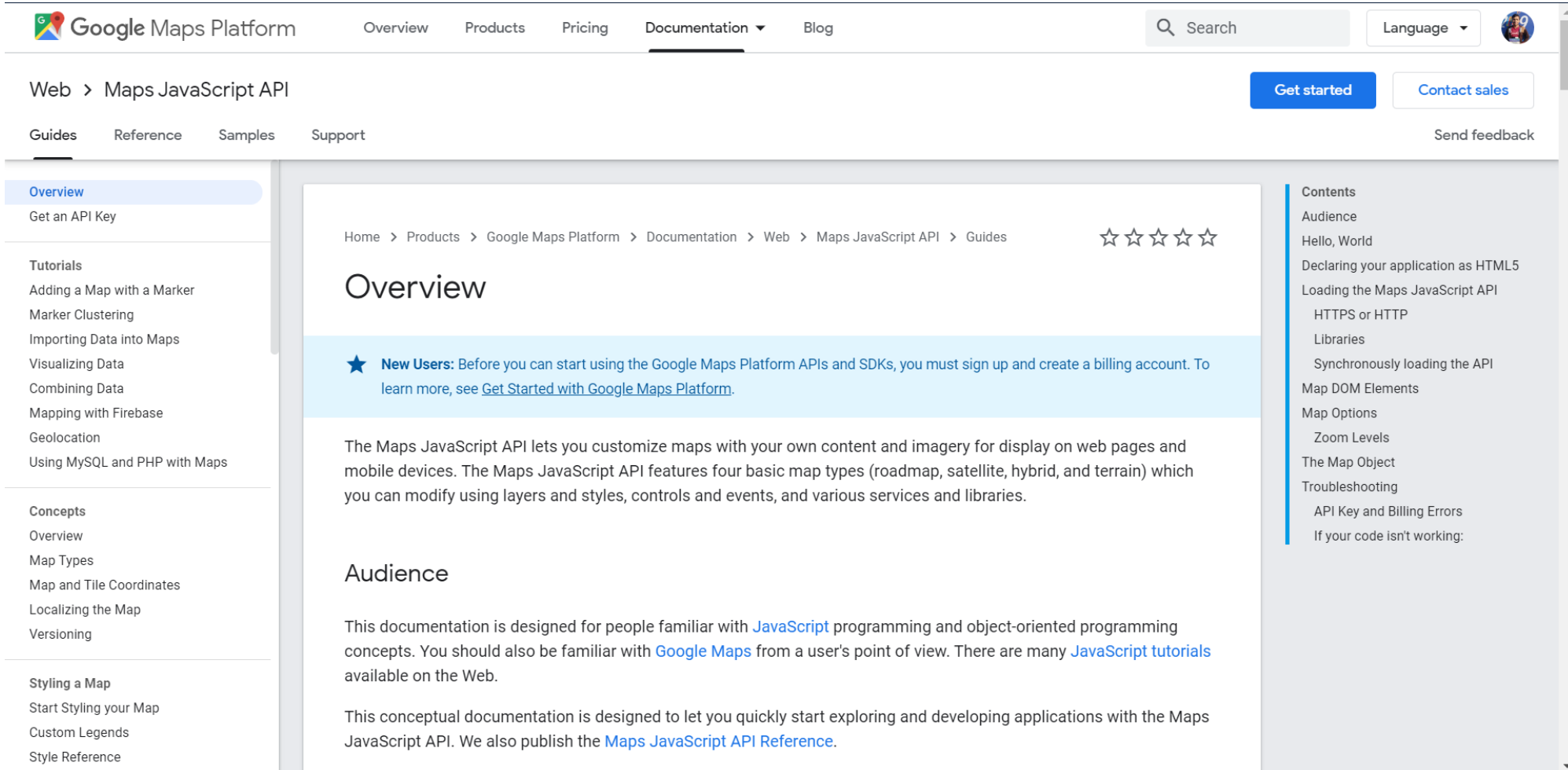
API is an acronym for Application Programming Interface. Two machines use it to communicate with each other. We will be dealing with Web APIs and then the definition changes to:

An API is used by two applications trying to communicate with each other over a network or Internet.

The API acts as a mediator between Django and other applications. other applications can be from Android, iOS, Web apps, browsers, etc. The API's main task is to receive data from other applications and provide them to the backend. This data is usually in **JSON format**.

Okay, so some of you may ask what about those so-called **Google APIs**. Okay, let's see. They are also APIs of the same nature. Understand it like Google providing the API as a service.

Here, you can see some of the common examples of APIs used by developers.



The screenshot shows the Google Maps Platform documentation page for the Maps JavaScript API. The page is titled "Overview" and includes a navigation menu on the left, a main content area, and a right-hand sidebar with a table of contents. The main content area features a "New Users" warning and two paragraphs of introductory text. The sidebar lists various topics such as "Audience", "Hello, World", "Declaring your application as HTML5", "Loading the Maps JavaScript API", "Map DOM Elements", "Map Options", "The Map Object", "Troubleshooting", and "If your code isn't working".

Google Maps Platform Overview Products Pricing Documentation Blog Search Language

Web > Maps JavaScript API Get started Contact sales

Guides Reference Samples Support Send feedback

Overview
Get an API Key

Tutorials
Adding a Map with a Marker
Marker Clustering
Importing Data into Maps
Visualizing Data
Combining Data
Mapping with Firebase
Geolocation
Using MySQL and PHP with Maps

Concepts
Overview
Map Types
Map and Tile Coordinates
Localizing the Map
Versioning

Styling a Map
Start Styling your Map
Custom Legends
Style Reference

Home > Products > Google Maps Platform > Documentation > Web > Maps JavaScript API > Guides ☆☆☆☆☆

Overview

★ **New Users:** Before you can start using the Google Maps Platform APIs and SDKs, you must sign up and create a billing account. To learn more, see [Get Started with Google Maps Platform](#).

The Maps JavaScript API lets you customize maps with your own content and imagery for display on web pages and mobile devices. The Maps JavaScript API features four basic map types (roadmap, satellite, hybrid, and terrain) which you can modify using layers and styles, controls and events, and various services and libraries.

Audience

This documentation is designed for people familiar with [JavaScript](#) programming and object-oriented programming concepts. You should also be familiar with [Google Maps](#) from a user's point of view. There are many [JavaScript tutorials](#) available on the Web.

This conceptual documentation is designed to let you quickly start exploring and developing applications with the Maps JavaScript API. We also publish the [Maps JavaScript API Reference](#).

Contents
Audience
Hello, World
Declaring your application as HTML5
Loading the Maps JavaScript API
 HTTPS or HTTP
 Libraries
 Synchronously loading the API
Map DOM Elements
Map Options
 Zoom Levels
The Map Object
Troubleshooting
 API Key and Billing Errors
 If your code isn't working:

What are RESTful APIs?

REST stands for Representational State Transfer. REST is an architecture on which we develop web services. Web services can be understood as your device connects to the internet. When you search for anything on Google or watch something on YouTube. These are web services where your device is communicating to a server. When these web services use REST Architecture, they are called RESTful Web Services. These web services use HTTP to transmit data between machines. now, back to the question, what are RESTful APIs?

A RESTful API acts as a translator between two machines communicating over a Web service. This is just like an API but it's working on a RESTful Web service. Web developers program REST API such that server can receive data from applications. These applications can be web-apps, Android/iOS apps, etc. RESTful APIs today return JSON files which can be interpreted by a variety of devices.

What is Django REST Framework?

So, as we learned in previous sections, DRF is an acronym for Django REST Framework. (stating the obvious) It's used to develop REST APIs for Django. Yup, DRF is used to develop **RESTful APIs** which is both easy and a smart way.

DRF is a framework built upon the Django Framework. It is not a separate framework. You can say that it is a tool which alongside Django is used to **develop RESTful APIs**. It increases the development speed. It also addresses various security issues natively.

DRF setup

Install:

```
$ pip install djangorestframework
```

Update settings.py:

```
INSTALLED_APPS = (  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'talk',  
    'rest_framework'  
)
```

RESTful Structure:

In a RESTful API, endpoints (URLs) define the structure of the API and how end users access data from our application using the HTTP methods: GET, POST, PUT, DELETE. Endpoints should be logically organized around collections and elements, both of which are resources. In our case, we have one single resource, posts, so we will use the following URLs - /posts/ and /posts/<id> for collections and elements, respectively.

	GET	POST	PUT	DELETE
/posts/	Show all posts	Add new post	Update all posts	Delete all posts
/posts/<id>	Show <id>	N/A	Update <id>	Delete id

DRF Quick Start

Let's get our new API up and running!

Model Serializer

DRF's Serializers convert model instances to Python dictionaries, which can then be rendered in various API appropriate formats - like JSON or XML. Similar to the Django ModelForm class, DRF comes with a concise format for its Serializers, the ModelSerializer class. It's simple to use: Just tell it which fields you want to use from the model:

```
from rest_framework import serializers
from talk.models import Post

class PostSerializer(serializers.ModelSerializer):

    class Meta:
        model = Post
        fields = ('id', 'author', 'text', 'created', 'updated')
```

Save this as serializers.py within the “talk” directory.

Update Views

We need to refactor our current views to fit the RESTful paradigm. Comment out the current views and add in:

```
from django.http import HttpResponse
from rest_framework.decorators import api_view
from rest_framework.response import Response
from talk.models import Post
from talk.serializers import PostSerializer
from talk.forms import PostForm

def home(request):
    tpl_vars = {'form': PostForm()}
    return render(request, 'talk/index.html', tpl_vars)
```



```
@api_view(['GET'])  
def post_collection(request):  
    if request.method == 'GET':  
        posts = Post.objects.all()  
        serializer = PostSerializer(posts, many=True)  
        return Response(serializer.data)
```

```
@api_view(['GET'])  
def post_element(request, pk):  
    try:  
        post = Post.objects.get(pk=pk)  
    except Post.DoesNotExist:  
        return HttpResponse(status=404)  
  
    if request.method == 'GET':  
        serializer = PostSerializer(post)  
        return Response(serializer.data)
```


What's happening here:

- 1. First, the `@api_view` decorator checks that the appropriate HTTP request is passed into the view function. Right now, we're only supporting GET requests.**
- 2. Then, the view either grabs all the data, if it's for the collection, or just a single post, if it's for an element.**
- 3. Finally, the data is serialized to JSON and returned.**

Update URLs

Let's wire up some new URLs:

```
# Talk urls
from django.conf.urls import patterns, url

urlpatterns = patterns(
    'talk.views',
    url(r'^$', 'home'),

    # api
    url(r'^api/v1/posts/$', 'post_collection'),
    url(r'^api/v1/posts/(?P<pk>[0-9]+)$', 'post_element')
)
```

Test

We're now ready for our first test!

- 1. Fire up the server, then navigate to:
<http://127.0.0.1:8000/api/v1/posts/?format=json>.**
- 2. Now let's check out theBrowsable API. Navigate to
<http://127.0.0.1:8000/api/v1/posts/>**
- 3. So, With no extra work on our end we automatically get this nice, human-readable output of our API. Nice! This is a huge win for DRF.**
- 4. How about an element? Try:
<http://127.0.0.1:8000/api/v1/posts/1>**

Before moving on you may have noticed that the author field is an `id` rather than the actual `username`. We'll address this shortly. For now, let's wire up our new API so that it works with our current application's Templates.

Refactor for REST

GET

On the initial page load, we want to display all posts. To do that, add the following AJAX request:



load_posts()

// Load all posts on page load

```
function load_posts() {
  $.ajax({
    url : "api/v1/posts/", // the endpoint
    type : "GET", // http method
    // handle a successful response
    success : function(json) {
      for (var i = 0; i < json.length; i++) {
        console.log(json[i])
        $("#talk").prepend("<li id='post-
"+json[i].id+"'><strong>"+json[i].text+"</strong> - <em>
"+json[i].author+"</em> - <span> "+json[i].created+
        "</span> - <a id='delete-post-"+json[i].id+"'>delete
me</a></li>");
      }
    },
  },
```

```
// handle a non-successful response  
error : function(xhr,errmsg,err) {  
    $('#results').html("<div class='alert-box alert radius' data-  
alert>Oops! We have encountered an error: "+errmsg+  
    " <a href='#' class='close'>&times;</a></div>"); // add the  
error to the dom  
    console.log(xhr.status + ": " + xhr.responseText); // provide a  
bit more info about the error to the console  
    }  
});  
};
```

You've seen all this before. Notice how we're handling a success: Since the API sends back a number of objects, we need to iterate through them, appending each to the DOM. We also changed `json[i].postpk` to `json[i].id` as we are serializing the post id.

Test this out. Fire up the server, log in, then check out the posts.

Besides the author being displayed as an id, take note of the datetime format. This is not what we want, right? We want a readable datetime format. Let's update that...

Datetime Format

We can use an awesome JavaScript library called MomentJS to easily format the date anyway we want.

First, we need to import the library to our index.html file:

HTML Code:

```
<!-- scripts -->  
<script  
src="http://cdnjs.cloudflare.com/ajax/libs/moment.js/2.8.2/moment.min.js"></script>  
<script src="static/scripts/main.js"></script>
```

Then update the for loop in main.js:

JavaScript Code:

```
for (var i = 0; i < json.length; i++) {  
    dateString = convert_to_readable_date(json[i].created)  
    $("#talk").prepend("<li id='post-  
"+json[i].id+"'><strong>"+json[i].text+  
    "</strong> - <em> "+json[i].author+"</em> - <span>  
"+dateString+  
    "</span> - <a id='delete-post-"+json[i].id+"'>delete  
me</a></li>");  
}
```

Here we pass the date string to a new function called `convert_to_readable_date()`, which needs to be added:

JavaScript:

```
// convert ugly date to human readable date  
function convert_to_readable_date(date_time_string) {  
    var newDate = moment(date_time_string).format('MM/DD/YYYY,  
h:mm:ss a')  
    return newDate  
}
```

That's it. Refresh the browser. The datetime format should now look something like this - 08/22/2014, 6:48:29 pm.

POST

POST requests are handled in similar fashion. Before messing with the serializer, let's test it first by just updating the views. Maybe we'll get lucky and it will just work.

Update the `post_collection()` function in `views.py`:

```
@api_view(['GET', 'POST'])  
def post_collection(request):  
    if request.method == 'GET':  
        posts = Post.objects.all()  
        serializer = PostSerializer(posts, many=True)  
        return Response(serializer.data)  
    elif request.method == 'POST':  
        data = {'text': request.DATA.get('the_post'), 'author':  
request.user.pk}  
        serializer = PostSerializer(data=data)  
        if serializer.is_valid():  
            serializer.save()  
            return Response(serializer.data, status=status.HTTP_201_CREATED)  
        return Response(serializer.errors,  
status=status.HTTP_400_BAD_REQUEST)
```

Also add the following import:

from rest_framework import status

What's happening here:

- 1. `request.DATA` extends Django's `HttpRequest`, returning the content from the request body. Read more about it [here](#).**
- 2. If the deserialization process works, we return a response with a code of `201` (created).**
- 3. On the other hand, if the deserialization process fails, we return a `400` response.**

Update the endpoint in the `create_post()` function

From:

JavaScript Code:

```
url : "create_post/", // the endpoint
```

To:

JavaScript Code:

```
url : "api/v1/posts/", // the endpoint
```

Test it out in the browser. It should work. Don't forget to update the handling of the dates correctly as well as changing json.postpk to json.id:

```
success : function(json) {
  $('#post-text').val(''); // remove the value from the input
  console.log(json); // log the returned json to the console
  dateString = convert_to_readable_date(json.created)
  $("#talk").prepend("<li id='post-
"+json.id+"'><strong>"+json.text+"</strong> - <em> "+
  json.author+"</em> - <span> "+dateString+
  "</span> - <a id='delete-post-"+json.id+"'>delete me</a></li>");
  console.log("success"); // another sanity check
},
```


Author Format

Now's a good time to pause and address the author id vs. username issue. We have a few options:

1. Be really **RESTFUL** and make another call to get the user info, which is not good for performance.
2. Utilize the **SlugRelatedField** relation.

Let's go with the latter option. Update the serializer:

```
from django.contrib.auth.models import User
from rest_framework import serializers
from talk.models import Post

class PostSerializer(serializers.ModelSerializer):
    author = serializers.SlugRelatedField(
        queryset=User.objects.all(), slug_field='username'
    )

    class Meta:
        model = Post
        fields = ('id', 'author', 'text', 'created', 'updated')
```

What's happening here?

1. The **SlugRelatedField** allows us to change the target of the author field from id to username.
2. Also, by default the target field - **username** - is both readable and writeable so out-of-the-box this relation will work for both **GET** and **POST** requests.

Update the data variable in the views as well:

```
data = {'text': request.DATA.get('the_post'), 'author':  
request.user}
```

Test again. You should now see the author's username. Make sure both **GET and **POST** requests are working correctly.**

Delete

Before changing or adding anything, test it out. Try the delete link. What happens? You should get a 404 error. Any idea why that would be? Or where to go to find out what the issue is? How about the `delete_post` function in our JavaScript file:

```
url : "delete_post/", // the endpoint
```

That URL does not exist. Before we update it, ask yourself - “Should we target the collection or an individual element?”. If you’re unsure, scroll back up and look at the RESTful Structure table. Unless we want to delete all posts, then we need to hit the element endpoint:

```
url : "api/v1/posts/" + post_primary_key, // the endpoint
```

Test again. Now what happens? You should see a 405 error - 405: {"detail": "Method 'DELETE' not allowed."} - since the view is not setup to handle a DELETE request.



```
@api_view(['GET', 'DELETE'])  
def post_element(request, pk):  
    try:  
        post = Post.objects.get(pk=pk)  
    except Post.DoesNotExist:  
        return HttpResponse(status=404)  
  
    if request.method == 'GET':  
        serializer = PostSerializer(post)  
        return Response(serializer.data)  
  
    elif request.method == 'DELETE':  
        post.delete()  
        return Response(status=status.HTTP_204_NO_CONTENT)
```

With the **DELETE HTTP** verb added, we can handle the request by removing the post with the **delete()** method and returning a 204 response. Does it work? Only one way to find out. This time when you test make sure that (a) the post is actually deleted and removed from the DOM and (b) that a 204 status code is returned (you can confirm this in the Network tab within Chrome Developer Tools).