- Home
- Tutorial
  - Quickstart
  - 1 - Serialization
  - 2 - Requests and responses
  - 3 - Class based views
  - 4 - Authentication and permissions
  - 5 - Relationships and hyperlinked APIs
  - 6 - Viewsets and routers
- API Guide
  - Requests
  - Responses
  - Views
  - Generic views
  - Viewsets
  - Routers
  - Parsers
  - Renderers
  - Serializers
  - Serializer fields
  - Serializer relations
  - Validators
  - Authentication
  - Permissions
  - Caching
  - Throttling
  - Filtering
  - Pagination
  - Versioning
  - Content negotiation
  - Metadata
  - Schemas
  - Format suffixes
  - Returning URLs
  - Exceptions
  - Status codes
  - Testing
  - Settings
- Topics
  - Documenting your API
  - Internationalization
  - AJAX, CSRF & CORS
  - HTML & Forms
  - Browser Enhancements
  - The Browsable API
  - REST, Hypermedia & HATEOAS
- Community
  - Tutorials and Resources
  - Third Party Packages
  - Contributing to REST framework
  - Project management
  - Release Notes
  - 3.15 Announcement
  - 3.14 Announcement
  - 3.13 Announcement
  - 3.12 Announcement
  - 3.11 Announcement
  - 3.10 Announcement
  - 3.9 Announcement
  - 3.8 Announcement
  - 3.7 Announcement
  - 3.6 Announcement
  - 3.5 Announcement
  - 3.4 Announcement
  - 3.3 Announcement
  - 3.2 Announcement
  - 3.1 Announcement
  - 3.0 Announcement
  - Kickstarter Announcement
  - Mozilla Grant
  - Funding
  - Jobs

×

Documentation search

Search...

Close

- Django REST framework
- Funding
- Requirements
- Installation
- Example
- Quickstart
- Development
- Support
- Security
- License

Django REST Framework

Django REST framework is a powerful and flexible toolkit for building Web APIs.

Some reasons you might want to use REST framework:

- The Web browsable API is a huge usability win for your developers.

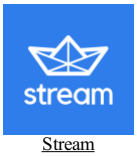
- Authentication policies including packages for OAuth1a and OAuth2.
- Serialization that supports both ORM and non-ORM data sources.
- Customizable all the way down - just use regular function-based views if you don't need the more powerful features.
- Extensive documentation, and great community support.
- Used and trusted by internationally recognised companies including Mozilla, Red Hat, Heroku, and Eventbrite.

Funding

REST framework is a *collaboratively funded project*. If you use REST framework commercially we strongly encourage you to invest in its continued development by **signing up for a paid plan**.

Every single sign-up helps us make REST framework long-term financially sustainable.


Sentry



Stream


Spacinov

Retool


bit.io


PostHog

CryptAPI


FEZTO


Svix

Many thanks to all our wonderful sponsors, and in particular to our premium backers, Sentry, Stream, Spacinov, Retool, bit.io, PostHog, CryptAPI, FEZTO, and Svix.

Requirements

REST framework requires the following:

- Python (3.6, 3.7, 3.8, 3.9, 3.10, 3.11)
- Django (3.0, 3.1, 3.2, 4.0, 4.1, 4.2, 5.0)

We **highly recommend** and only officially support the latest patch release of each Python and Django series.

The following packages are optional:

- PyYAML, uritemplate (5.1+, 3.0.0+) - Schema generation support.
- Markdown (3.0.0+) - Markdown support for the browsable API.
- Pygments (2.4.0+) - Add syntax highlighting to Markdown processing.
- django-filter (1.0.1+) - Filtering support.
- django-guardian (1.1.1+) - Object level permissions support.

Installation

Install using `pip`, including any optional packages you want...

```
pip install djangorestframework
pip install markdown       # Markdown support for the browsable API.
pip install django-filter  # Filtering support
```

...or clone the project from github.

```
git clone https://github.com/encode/django-rest-framework
```

Add `'rest_framework'` to your `INSTALLED_APPS` setting.

```
INSTALLED_APPS = [
    ...
    'rest_framework',
]
```

If you're intending to use the browsable API you'll probably also want to add REST framework's login and logout views. Add the following to your root `urls.py` file.

```
urlpatterns = [
    ...
    path('api-auth/', include('rest_framework.urls'))
]
```

Note that the URL path can be whatever you want.

Example

Let's take a look at a quick example of using REST framework to build a simple model-backed API.

We'll create a read-write API for accessing information on the users of our project.

Any global settings for a REST framework API are kept in a single configuration dictionary named `REST_FRAMEWORK`. Start off by adding the following to your `settings.py` module:

```
REST_FRAMEWORK = {
    # Use Django's standard `django.contrib.auth` permissions,
    # or allow read-only access for unauthenticated users.
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.DjangoModelPermissionsOrAnonReadOnly'
    ]
}
```

Don't forget to make sure you've also added `rest_framework` to your `INSTALLED_APPS`.

We're ready to create our API now. Here's our project's root `urls.py` module:

```
from django.urls import path, include
from django.contrib.auth.models import User
from rest_framework import routers, serializers, viewsets

# Serializers define the API representation.
class UserSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = User
        fields = ['url', 'username', 'email', 'is_staff']

# ViewSets define the view behavior.
class UserViewSet(viewsets.ModelViewSet):
```

```
    queryset = User.objects.all()
    serializer_class = UserSerializer

# Routers provide an easy way of automatically determining the URL conf.
router = routers.DefaultRouter()
router.register(r'users', UserViewSet)

# Wire up our API using automatic URL routing.
# Additionally, we include login URLs for the browsable API.
urlpatterns = [
    path('', include(router.urls)),
    path('api-auth/', include('rest_framework.urls', namespace='rest_framework'))
]
```

You can now open the API in your browser at http://127.0.0.1:8000/ , and view your new 'users' API. If you use the login control in the top right corner you'll also be able to add, create and delete users from the system.

Quickstart

Can't wait to get started? The quickstart guide is the fastest way to get up and running, and building APIs with REST framework.

Development

See the Contribution guidelines for information on how to clone the repository, run the test suite and contribute changes back to REST Framework.

Support

For support please see the REST framework discussion group , try the `#restframework` channel on `irc.libera.chat` , or raise a question on Stack Overflow , making sure to include the 'django-rest-framework' tag.

For priority support please sign up for a professional or premium sponsorship plan .

Security

Security issues are handled under the supervision of the Django security team .

Please report security issues by emailing security@djangoproject.com .

The project maintainers will then work with you to resolve any issues where required, prior to any public disclosure.

License

Documentation built with MkDocs .

- Third Party Packages
- Contributing to REST framework
- Project management
- Release Notes
- 3.15 Announcement
- 3.14 Announcement
- 3.13 Announcement
- 3.12 Announcement
- 3.11 Announcement
- 3.10 Announcement
- 3.9 Announcement
- 3.8 Announcement
- 3.7 Announcement
- 3.6 Announcement
- 3.5 Announcement
- 3.4 Announcement
- 3.3 Announcement
- 3.2 Announcement
- 3.1 Announcement
- 3.0 Announcement
- Kickstarter Announcement
- Mozilla Grant
- Funding
- Jobs

- Quickstart
- Project setup
- Serializers
- Views
- URLs
- Pagination
- Settings
- Testing our API

Quickstart

We're going to create a simple API to allow admin users to view and edit the users and groups in the system.

Project setup

Create a new Django project named `tutorial` , then start a new app called `quickstart` .

```
# Create the project directory
mkdir tutorial
cd tutorial

# Create a virtual environment to isolate our package dependencies locally
python3 -m venv env
source env/bin/activate  # On Windows use `env\Scripts\activate`

# Install Django and Django REST framework into the virtual environment
pip install django
pip install djangorestframework

# Set up a new project with a single application
django-admin startproject tutorial .  # Note the trailing '.' character
cd tutorial
django-admin startapp quickstart
cd ..
```

The project layout should look like:

```
$ pwd
<some path>/tutorial
$ find .
.
./tutorial
./tutorial/asgi.py
./tutorial/__init__.py
./tutorial/quickstart
./tutorial/quickstart/migrations
./tutorial/quickstart/migrations/__init__.py
./tutorial/quickstart/models.py
./tutorial/quickstart/__init__.py
./tutorial/quickstart/apps.py
./tutorial/quickstart/admin.py
./tutorial/quickstart/tests.py
./tutorial/quickstart/views.py
./tutorial/settings.py
./tutorial/urls.py
./tutorial/wsgi.py
./env
./env/...
./manage.py
```

It may look unusual that the application has been created within the project directory. Using the project's namespace avoids name clashes with external modules (a topic that goes outside the scope of the quickstart).

Now sync your database for the first time:

```
python manage.py migrate
```

We'll also create an initial user named `admin` with a password. We'll authenticate as that user later in our example.

```
python manage.py createsuperuser --username admin --email admin@example.com
```

Once you've set up a database and the initial user is created and ready to go, open up the app's directory and we'll get coding...

Serializers

First up we're going to define some serializers. Let's create a new module named `tutorial/quickstart/serializers.py` that we'll use for our data representations.

```
from django.contrib.auth.models import Group, User
from rest_framework import serializers
```

```
class UserSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = User
        fields = ['url', 'username', 'email', 'groups']


class GroupSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Group
        fields = ['url', 'name']
```

Notice that we're using hyperlinked relations in this case with `HyperlinkedModelSerializer` . You can also use primary key and various other relationships, but hyperlinking is good RESTful design.

Views

Right, we'd better write some views then. Open `tutorial/quickstart/views.py` and get typing.

```
from django.contrib.auth.models import Group, User
from rest_framework import permissions, viewsets

from tutorial.quickstart.serializers import GroupSerializer, UserSerializer


class UserViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows users to be viewed or edited.
    """
    queryset = User.objects.all().order_by('-date_joined')
    serializer_class = UserSerializer
    permission_classes = [permissions.IsAuthenticated]


class GroupViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows groups to be viewed or edited.
    """
    queryset = Group.objects.all().order_by('name')
    serializer_class = GroupSerializer
    permission_classes = [permissions.IsAuthenticated]
```

Rather than write multiple views we're grouping together all the common behavior into classes called `ViewSets` .

We can easily break these down into individual views if we need to, but using viewsets keeps the view logic nicely organized as well as being very concise.

URLs

Okay, now let's wire up the API URLs. On to `tutorial/urls.py` ...

```
from django.urls import include, path
from rest_framework import routers

from tutorial.quickstart import views

router = routers.DefaultRouter()
router.register(r'users', views.UserViewSet)
router.register(r'groups', views.GroupViewSet)

# Wire up our API using automatic URL routing.
# Additionally, we include login URLs for the browsable API.
urlpatterns = [
    path('', include(router.urls)),
    path('api-auth/', include('rest_framework.urls', namespace='rest_framework'))
]
```

Because we're using viewsets instead of views, we can automatically generate the URL conf for our API, by simply registering the viewsets with a router class.

Again, if we need more control over the API URLs we can simply drop down to using regular class-based views, and writing the URL conf explicitly.

Finally, we're including default login and logout views for use with the browsable API. That's optional, but useful if your API requires authentication and you want to use the browsable API.

Pagination

Pagination allows you to control how many objects per page are returned. To enable it add the following lines to `tutorial/settings.py`

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 10
}
```

Settings

Add `'rest_framework'` to `INSTALLED_APPS` . The settings module will be in `tutorial/settings.py`

```
INSTALLED_APPS = [
    ...
    'rest_framework',
]
```

Okay, we're done.

Testing our API

We're now ready to test the API we've built. Let's fire up the server from the command line.

```
python manage.py runserver
```

We can now access our API, both from the command-line, using tools like `curl` ...

```
bash: curl -u admin -H 'Accept: application/json; indent=4' http://127.0.0.1:8000/users/
Enter host password for user 'admin':
{
    "count": 1,
    "next": null,
    "previous": null,
    "results": [
        {
            "url": "http://127.0.0.1:8000/users/1/",
            "username": "admin",
            "email": "admin@example.com",
            "groups": []
        }
    ]
}
```

Or using the httpie , command line tool...

```
bash: http -a admin http://127.0.0.1:8000/users/
http: password for admin@127.0.0.1:8000::
$HTTP/1.1 200 OK
```

```
...
{
    "count": 1,
    "next": null,
    "previous": null,
    "results": [
        {
            "email": "admin@example.com",
            "groups": [],
            "url": "http://127.0.0.1:8000/users/1/",
            "username": "admin"
        }
    ]
}
```

Or directly through the browser, by going to the URL `http://127.0.0.1:8000/users/` ...

If you're working through the browser, make sure to login using the control in the top right corner.

Great, that was easy!

If you want to get a more in depth understanding of how REST framework fits together head on over to the tutorial, or start browsing the API guide.

Documentation built with MkDocs.

GitHub Next Previous Search Django REST framework

- Home
- Tutorial
  - Quickstart
  - 1 - Serialization
  - 2 - Requests and responses
  - 3 - Class based views
  - 4 - Authentication and permissions
  - 5 - Relationships and hyperlinked APIs
  - 6 - Viewsets and routers
- API Guide
  - Requests
  - Responses
  - Views
  - Generic views
  - Viewsets
  - Routers
  - Parsers
  - Renderers
  - Serializers
  - Serializer fields
  - Serializer relations
  - Validators
  - Authentication
  - Permissions
  - Caching
  - Throttling
  - Filtering
  - Pagination
  - Versioning
  - Content negotiation
  - Metadata
  - Schemas
  - Format suffixes
  - Returning URLs
  - Exceptions
  - Status codes
  - Testing
  - Settings
- Topics
  - Documenting your API
  - Internationalization
  - AJAX, CSRF & CORS
  - HTML & Forms
  - Browser Enhancements
  - The Browsable API
  - REST, Hypermedia & HATEOAS
- Community
  - Tutorials and Resources
  - Third Party Packages
  - Contributing to REST framework
  - Project management
  - Release Notes
  - 3.15 Announcement
  - 3.14 Announcement
  - 3.13 Announcement
  - 3.12 Announcement
  - 3.11 Announcement
  - 3.10 Announcement
  - 3.9 Announcement
  - 3.8 Announcement
  - 3.7 Announcement
  - 3.6 Announcement
  - 3.5 Announcement
  - 3.4 Announcement
  - 3.3 Announcement
  - 3.2 Announcement
  - 3.1 Announcement
  - 3.0 Announcement
  - Kickstarter Announcement
  - Mozilla Grant
  - Funding
  - Jobs

×

Documentation search

Search...

Close

- Django REST framework
- Funding
- Requirements
- Installation
- Example
- Quickstart
- Development
- Support
- Security
- License

Django REST Framework

Django REST framework is a powerful and flexible toolkit for building Web APIs.

Some reasons you might want to use REST framework:

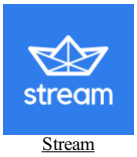
- The Web browsable API is a huge usability win for your developers.
- Authentication policies including packages for OAuth1a and OAuth2.
- Serialization that supports both ORM and non-ORM data sources.
- Customizable all the way down - just use regular function-based views if you don't need the more powerful features.
- Extensive documentation, and great community support.
- Used and trusted by internationally recognised companies including Mozilla, Red Hat, Heroku, and Eventbrite.

Funding

REST framework is a *collaboratively funded project*. If you use REST framework commercially we strongly encourage you to invest in its continued development by **signing up for a paid plan**.

Every single sign-up helps us make REST framework long-term financially sustainable.


Sentry


Stream


Spacinov


Retool


bit.io


PostHog


CryptAPI


FEZTO


Svix

Many thanks to all our wonderful sponsors, and in particular to our premium backers, Sentry, Stream, Spacinov, Retool, bit.io, PostHog, CryptAPI, FEZTO, and Svix.

Requirements

REST framework requires the following:

- Python (3.6, 3.7, 3.8, 3.9, 3.10, 3.11)
- Django (3.0, 3.1, 3.2, 4.0, 4.1, 4.2, 5.0)

We **highly recommend** and only officially support the latest patch release of each Python and Django series.

The following packages are optional:

- PyYAML, uritemplate (5.1+, 3.0.0+) - Schema generation support.
- Markdown (3.0.0+) - Markdown support for the browsable API.
- Pygments (2.4.0+) - Add syntax highlighting to Markdown processing.
- django-filter (1.0.1+) - Filtering support.
- django-guardian (1.1.1+) - Object level permissions support.

Installation

Install using `pip`, including any optional packages you want...

```
pip install djangorestframework
pip install markdown       # Markdown support for the browsable API.
pip install django-filter  # Filtering support
```

...or clone the project from github.

```
git clone https://github.com/encode/django-rest-framework
```

Add `'rest_framework'` to your `INSTALLED_APPS` setting.

```
INSTALLED_APPS = [
    ...
    'rest_framework',
]
```

If you're intending to use the browsable API you'll probably also want to add REST framework's login and logout views. Add the following to your root `urls.py` file.

```
urlpatterns = [
    ...
    path('api-auth/', include('rest_framework.urls'))
]
```

Note that the URL path can be whatever you want.

Example

Let's take a look at a quick example of using REST framework to build a simple model-backed API.

We'll create a read-write API for accessing information on the users of our project.

Any global settings for a REST framework API are kept in a single configuration dictionary named `REST_FRAMEWORK` . Start off by adding the following to your `settings.py` module:

```
REST_FRAMEWORK = {
    # Use Django's standard `django.contrib.auth` permissions,
    # or allow read-only access for unauthenticated users.
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.DjangoModelPermissionsOrAnonReadOnly'
    ]
}
```

Don't forget to make sure you've also added `rest_framework` to your `INSTALLED_APPS` .

We're ready to create our API now. Here's our project's root `urls.py` module:

```
from django.urls import path, include
from django.contrib.auth.models import User
from rest_framework import routers, serializers, viewsets

# Serializers define the API representation.
class UserSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = User
        fields = ['url', 'username', 'email', 'is_staff']

# ViewSets define the view behavior.
class UserViewSet(viewsets.ModelViewSet):
    queryset = User.objects.all()
    serializer_class = UserSerializer

# Routers provide an easy way of automatically determining the URL conf.
router = routers.DefaultRouter()
router.register(r'users', UserViewSet)

# Wire up our API using automatic URL routing.
# Additionally, we include login URLs for the browsable API.
urlpatterns = [
    path('', include(router.urls)),
    path('api-auth/', include('rest_framework.urls', namespace='rest_framework'))
]
```

You can now open the API in your browser at http://127.0.0.1:8000/ , and view your new 'users' API. If you use the login control in the top right corner you'll also be able to add, create and delete users from the system.

Quickstart

Can't wait to get started? The quickstart guide is the fastest way to get up and running, and building APIs with REST framework.

Development

See the Contribution guidelines for information on how to clone the repository, run the test suite and contribute changes back to REST Framework.

Support

For support please see the REST framework discussion group , try the `#restframework` channel on `irc.libera.chat` , or raise a question on Stack Overflow , making sure to include the 'django-rest-framework' tag.

For priority support please sign up for a professional or premium sponsorship plan .

Security

Security issues are handled under the supervision of the Django security team .

Please report security issues by emailing security@djangoproject.com .

The project maintainers will then work with you to resolve any issues where required, prior to any public disclosure.

License

Documentation built with MkDocs .

- Caching
- Throttling
- Filtering
- Pagination
- Versioning
- Content negotiation
- Metadata
- Schemas
- Format suffixes
- Returning URLs
- Exceptions
- Status codes
- Testing
- Settings

- Topics
  - Documenting your API
  - Internationalization
  - AJAX, CSRF & CORS
  - HTML & Forms
  - Browser Enhancements
  - The Browsable API
  - REST, Hypermedia & HATEOAS
- Community
  - Tutorials and Resources
  - Third Party Packages
  - Contributing to REST framework
  - Project management
  - Release Notes
  - 3.15 Announcement
  - 3.14 Announcement
  - 3.13 Announcement
  - 3.12 Announcement
  - 3.11 Announcement
  - 3.10 Announcement
  - 3.9 Announcement
  - 3.8 Announcement
  - 3.7 Announcement
  - 3.6 Announcement
  - 3.5 Announcement
  - 3.4 Announcement
  - 3.3 Announcement
  - 3.2 Announcement
  - 3.1 Announcement
  - 3.0 Announcement
  - Kickstarter Announcement
  - Mozilla Grant
  - Funding
  - Jobs

- Tutorial 1: Serialization
- Introduction
- Setting up a new environment
- Getting started
- Creating a model to work with
- Creating a Serializer class
- Working with Serializers
- Using ModelSerializers
- Writing regular Django views using our Serializer
- Testing our first attempt at a Web API
- Where are we now

Tutorial 1: Serialization

Introduction

This tutorial will cover creating a simple pastebin code highlighting Web API. Along the way it will introduce the various components that make up REST framework, and give you a comprehensive understanding of how everything fits together.

The tutorial is fairly in-depth, so you should probably get a cookie and a cup of your favorite brew before getting started. If you just want a quick overview, you should head over to the quickstart documentation instead.

Note : The code for this tutorial is available in the encode/rest-framework-tutorial repository on GitHub. The completed implementation is also online as a sandbox version for testing, available here .

Setting up a new environment

Before we do anything else we'll create a new virtual environment, using venv . This will make sure our package configuration is kept nicely isolated from any other projects we're working on.

```
python3 -m venv env
source env/bin/activate
```

Now that we're inside a virtual environment, we can install our package requirements.

```
pip install django
pip install djangorestframework
pip install pygments  # We'll be using this for the code highlighting
```

Note: To exit the virtual environment at any time, just type `deactivate` . For more information see the venv documentation .

Getting started

Okay, we're ready to get coding. To get started, let's create a new project to work with.

```
cd ~
django-admin startproject tutorial
cd tutorial
```

Once that's done we can create an app that we'll use to create a simple Web API.

```
python manage.py startapp snippets
```

We'll need to add our new `snippets` app and the `rest_framework` app to `INSTALLED_APPS` . Let's edit the `tutorial/settings.py` file:

```
INSTALLED_APPS = [
    ...
    'rest_framework',
    'snippets',
]
```

Okay, we're ready to roll.

Creating a model to work with

For the purposes of this tutorial we're going to start by creating a simple `Snippet` model that is used to store code snippets. Go ahead and edit the `snippets/models.py` file. Note: Good programming practices include comments. Although you will find them in our repository version of this tutorial code, we have omitted them here to focus on the code itself.

```
from django.db import models
from pygments.lexers import get_all_lexers
from pygments.styles import get_all_styles

LEXERS = [item for item in get_all_lexers() if item[1]]
LANGUAGE_CHOICES = sorted([(item[1][0], item[0]) for item in LEXERS])
STYLE_CHOICES = sorted([(item, item) for item in get_all_styles()])


class Snippet(models.Model):
    created = models.DateTimeField(auto_now_add=True)
    title = models.CharField(max_length=100, blank=True, default='')
    code = models.TextField()
    linenos = models.BooleanField(default=False)
    language = models.CharField(choices=LANGUAGE_CHOICES, default='python', max_length=100)
    style = models.CharField(choices=STYLE_CHOICES, default='friendly', max_length=100)

    class Meta:
        ordering = ['created']
```

We'll also need to create an initial migration for our snippet model, and sync the database for the first time.

```
python manage.py makemigrations snippets
python manage.py migrate snippets
```

Creating a Serializer class

The first thing we need to get started on our Web API is to provide a way of serializing and deserializing the snippet instances into representations such as `json` . We can do this by declaring serializers that work very similar to Django's forms. Create a file in the `snippets` directory named `serializers.py` and add the following.

```
from rest_framework import serializers
from snippets.models import Snippet, LANGUAGE_CHOICES, STYLE_CHOICES


class SnippetSerializer(serializers.Serializer):
    id = serializers.IntegerField(read_only=True)
    title = serializers.CharField(required=False, allow_blank=True, max_length=100)
    code = serializers.CharField(style={'base_template': 'textarea.html'})
    linenos = serializers.BooleanField(required=False)
    language = serializers.ChoiceField(choices=LANGUAGE_CHOICES, default='python')
    style = serializers.ChoiceField(choices=STYLE_CHOICES, default='friendly')

    def create(self, validated_data):
        """
        Create and return a new `Snippet` instance, given the validated data.
        """
        return Snippet.objects.create(**validated_data)

    def update(self, instance, validated_data):
        """
        Update and return an existing `Snippet` instance, given the validated data.
        """
        instance.title = validated_data.get('title', instance.title)
        instance.code = validated_data.get('code', instance.code)
        instance.linenos = validated_data.get('linenos', instance.linenos)
        instance.language = validated_data.get('language', instance.language)
        instance.style = validated_data.get('style', instance.style)
        instance.save()
        return instance
```

The first part of the serializer class defines the fields that get serialized/deserialized. The `create()` and `update()` methods define how fully fledged instances are created or modified when calling `serializer.save()`

A serializer class is very similar to a Django `Form` class, and includes similar validation flags on the various fields, such as `required` , `max_length` and `default` .

The field flags can also control how the serializer should be displayed in certain circumstances, such as when rendering to HTML. The `{'base_template': 'textarea.html'}` flag above is equivalent to using `widget=widgets.Textarea` on a Django `Form` class. This is particularly useful for controlling how the browsable API should be displayed, as we'll see later in the tutorial.

We can actually also save ourselves some time by using the `ModelSerializer` class, as we'll see later, but for now we'll keep our serializer definition explicit.

Working with Serializers

Before we go any further we'll familiarize ourselves with using our new Serializer class. Let's drop into the Django shell.

```
python manage.py shell
```

Okay, once we've got a few imports out of the way, let's create a couple of code snippets to work with.

```
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from rest_framework.renderers import JSONRenderer
from rest_framework.parsers import JSONParser

snippet = Snippet(code='foo = "bar"\n')
snippet.save()

snippet = Snippet(code='print("hello, world")\n')
snippet.save()
```

We've now got a few snippet instances to play with. Let's take a look at serializing one of those instances.

```
serializer = SnippetSerializer(snippet)
serializer.data
# {'id': 2, 'title': '', 'code': 'print("hello, world")\n', 'linenos': False, 'language': 'python', 'style': 'friendly'}
```

At this point we've translated the model instance into Python native datatypes. To finalize the serialization process we render the data into `json` .

```
content = JSONRenderer().render(serializer.data)
content
# b'{"id": 2, "title": "", "code": "print(\\"hello, world\\")\\n", "linenos": false, "language": "python", "style": "friendly"}'
```

Deserialization is similar. First we parse a stream into Python native datatypes...

```
import io
```

```
stream = io.BytesIO(content)
data = JSONParser().parse(stream)
```

...then we restore those native datatypes into a fully populated object instance.

```
serializer = SnippetSerializer(data=data)
serializer.is_valid()
# True
serializer.validated_data
# OrderedDict([('title', ''), ('code', 'print("hello, world")\n'), ('linenos', False), ('language', 'python'), ('style', 'friendly')])
serializer.save()
# <Snippet: Snippet object>
```

Notice how similar the API is to working with forms. The similarity should become even more apparent when we start writing views that use our serializer.

We can also serialize querysets instead of model instances. To do so we simply add a `many=True` flag to the serializer arguments.

```
serializer = SnippetSerializer(Snippet.objects.all(), many=True)
serializer.data
# [OrderedDict([('id', 1), ('title', ''), ('code', 'foo = "bar"\n'), ('linenos', False), ('language', 'python'), ('style', 'friendly')]), OrderedDict([('id', 2), ('title', ''), (
```

Using ModelSerializers

Our `SnippetSerializer` class is replicating a lot of information that's also contained in the `Snippet` model. It would be nice if we could keep our code a bit more concise.

In the same way that Django provides both `Form` classes and `ModelForm` classes, REST framework includes both `Serializer` classes, and `ModelSerializer` classes.

Let's look at refactoring our serializer using the `ModelSerializer` class. Open the file `snippets/serializers.py` again, and replace the `SnippetSerializer` class with the following.

```
class SnippetSerializer(serializers.ModelSerializer):
    class Meta:
        model = Snippet
        fields = ['id', 'title', 'code', 'linenos', 'language', 'style']
```

One nice property that serializers have is that you can inspect all the fields in a serializer instance, by printing its representation. Open the Django shell with `python manage.py shell` , then try the following:

```
from snippets.serializers import SnippetSerializer
serializer = SnippetSerializer()
print(repr(serializer))
# SnippetSerializer():
#     id = IntegerField(label='ID', read_only=True)
#     title = CharField(allow_blank=True, max_length=100, required=False)
#     code = CharField(style={'base_template': 'textarea.html'})
#     linenos = BooleanField(required=False)
#     language = ChoiceField(choices=[('Clipper', 'FoxPro'), ('Cucumber', 'Gherkin'), ('RobotFramework', 'RobotFramework'), ('abap', 'ABAP'), ('ada', 'Ada')...
#     style = ChoiceField(choices=[('autumn', 'autumn'), ('borland', 'borland'), ('bw', 'bw'), ('colorful', 'colorful')...
```

It's important to remember that `ModelSerializer` classes don't do anything particularly magical, they are simply a shortcut for creating serializer classes:

- An automatically determined set of fields.
- Simple default implementations for the `create()` and `update()` methods.

Writing regular Django views using our Serializer

Let's see how we can write some API views using our new Serializer class. For the moment we won't use any of REST framework's other features, we'll just write the views as regular Django views.

Edit the `snippets/views.py` file, and add the following.

```
from django.http import HttpResponse, JsonResponse
from django.views.decorators.csrf import csrf_exempt
from rest_framework.parsers import JSONParser
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
```

The root of our API is going to be a view that supports listing all the existing snippets, or creating a new snippet.

```
@csrf_exempt
def snippet_list(request):
    """
    List all code snippets, or create a new snippet.
    """
    if request.method == 'GET':
        snippets = Snippet.objects.all()
        serializer = SnippetSerializer(snippets, many=True)
        return JsonResponse(serializer.data, safe=False)

    elif request.method == 'POST':
        data = JSONParser().parse(request)
        serializer = SnippetSerializer(data=data)
        if serializer.is_valid():
            serializer.save()
            return JsonResponse(serializer.data, status=201)
        return JsonResponse(serializer.errors, status=400)
```

Note that because we want 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` . This isn't something that you'd normally want to do, and REST framework views actually use more sensible behavior than this, but it'll do for our purposes right now.

We'll also need a view which corresponds to an individual snippet, and can be used to retrieve, update or delete the snippet.

```
@csrf_exempt
def snippet_detail(request, pk):
    """
    Retrieve, update or delete a code snippet.
    """
    try:
        snippet = Snippet.objects.get(pk=pk)
    except Snippet.DoesNotExist:
        return HttpResponse(status=404)

    if request.method == 'GET':
        serializer = SnippetSerializer(snippet)
        return JsonResponse(serializer.data)

    elif request.method == 'PUT':
        data = JSONParser().parse(request)
        serializer = SnippetSerializer(snippet, data=data)
        if serializer.is_valid():
            serializer.save()
            return JsonResponse(serializer.data)
        return JsonResponse(serializer.errors, status=400)

    elif request.method == 'DELETE':
        snippet.delete()
        return HttpResponse(status=204)
```

Finally we need to wire these views up. Create the `snippets/urls.py` file:

```
from django.urls import path
from snippets import views

urlpatterns = [
    path('snippets/', views.snippet_list),
    path('snippets/<int:pk>/', views.snippet_detail),
]
```

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

```
from django.urls import path, include

urlpatterns = [
    path('', include('snippets.urls')),
]
```

It's worth noting that there are a couple of edge cases we're not dealing with properly at the moment. If we send malformed `json` , or if a request is made with a method that the view doesn't handle, then we'll end up with a 500 "server error" response. Still, this'll do for now.

Testing our first attempt at a Web API

Now we can start up a sample server that serves our snippets.

Quit out of the shell...

```
quit()
```

...and start up Django's development server.

```
python manage.py runserver

Validating models...

0 errors found
Django version 4.0, using settings 'tutorial.settings'
Starting Development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

In another terminal window, we can test the server.

We can test our API using curl or httpie . Httpie is a user friendly http client that's written in Python. Let's install that.

You can install httpie using pip:

```
pip install httpie
```

Finally, we can get a list of all of the snippets:

```
http http://127.0.0.1:8000/snippets/

HTTP/1.1 200 OK
...
[
  {
    "id": 1,
    "title": "",
    "code": "foo = \"bar\"\n",
    "linenos": false,
    "language": "python",
    "style": "friendly"
  },
  {
    "id": 2,
    "title": "",
    "code": "print(\"hello, world\")\n",
    "linenos": false,
    "language": "python",
    "style": "friendly"
  }
]
```

Or we can get a particular snippet by referencing its id:

```
http http://127.0.0.1:8000/snippets/2/

HTTP/1.1 200 OK
...
{
  "id": 2,
  "title": "",
  "code": "print(\"hello, world\")\n",
  "linenos": false,
  "language": "python",
  "style": "friendly"
}
```

Similarly, you can have the same json displayed by visiting these URLs in a web browser.

Where are we now

We're doing okay so far, we've got a serialization API that feels pretty similar to Django's Forms API, and some regular Django views.

Our API views don't do anything particularly special at the moment, beyond serving `json` responses, and there are some error handling edge cases we'd still like to clean up, but it's a functioning Web API.

We'll see how we can start to improve things in part 2 of the tutorial .

Documentation built with MkDocs .

- Throttling
- Filtering
- Pagination
- Versioning
- Content negotiation
- Metadata
- Schemas
- Format suffixes
- Returning URLs
- Exceptions
- Status codes
- Testing
- Settings

- Topics
  - Documenting your API
  - Internationalization
  - AJAX, CSRF & CORS
  - HTML & Forms
  - Browser Enhancements
  - The Browsable API
  - REST, Hypermedia & HATEOAS
- Community
  - Tutorials and Resources
  - Third Party Packages
  - Contributing to REST framework
  - Project management
  - Release Notes
  - 3.15 Announcement
  - 3.14 Announcement
  - 3.13 Announcement
  - 3.12 Announcement
  - 3.11 Announcement
  - 3.10 Announcement
  - 3.9 Announcement
  - 3.8 Announcement
  - 3.7 Announcement
  - 3.6 Announcement
  - 3.5 Announcement
  - 3.4 Announcement
  - 3.3 Announcement
  - 3.2 Announcement
  - 3.1 Announcement
  - 3.0 Announcement
  - Kickstarter Announcement
  - Mozilla Grant
  - Funding
  - Jobs

- Tutorial 2: Requests and Responses
- Request objects
- Response objects
- Status codes
- Wrapping API views
- Pulling it all together
- Adding optional format suffixes to our URLs
- How's it looking?
- What's next?

Tutorial 2: Requests and Responses

From this point we're going to really start covering the core of REST framework. Let's introduce a couple of essential building blocks.

Request objects

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 similar to `request.POST`, but more useful for working with Web APIs.

```
request.POST  # Only handles form data.  Only works for 'POST' method.
request.data  # Handles arbitrary data.  Works for 'POST', 'PUT' and 'PATCH' methods.
```

Response objects

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.

```
return Response(data)  # Renders to content type as requested by the client.
```

Status codes

Using numeric HTTP status codes in your views doesn't always make for obvious reading, and it's easy to not notice if you get an error code wrong. REST framework provides more explicit identifiers for each status code, such as `HTTP_400_BAD_REQUEST` in the `status` module. It's a good idea to use these throughout rather than using numeric identifiers.

Wrapping API views

REST framework provides two wrappers you can use to write API views.

1. The `@api_view` decorator for working with function based views.
2. 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 behavior such as returning `405 Method Not Allowed` responses when appropriate, and handling any `ParseError` exceptions that occur when accessing `request.data` with malformed input.

Pulling it all together

Okay, let's go ahead and start using these new components to refactor our views slightly.

```
from rest_framework import status
from rest_framework.decorators import api_view
from rest_framework.response import Response
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer


@api_view(['GET', 'POST'])
def snippet_list(request):
    """
    List all code snippets, or create a new snippet.
    """
    if request.method == 'GET':
        snippets = Snippet.objects.all()
        serializer = SnippetSerializer(snippets, many=True)
        return Response(serializer.data)

    elif request.method == 'POST':
        serializer = SnippetSerializer(data=request.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)
```

Our instance view is an improvement over the previous example. It's a little more concise, and the code now feels very similar to if we were working with the Forms API. We're also using named status codes, which makes the response meanings more obvious.

Here is the view for an individual snippet, in the `views.py` module.

```
@api_view(['GET', 'PUT', 'DELETE'])
def snippet_detail(request, pk):
    """
    Retrieve, update or delete a code snippet.
    """
    try:
        snippet = Snippet.objects.get(pk=pk)
    except Snippet.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    if request.method == 'GET':
        serializer = SnippetSerializer(snippet)
        return Response(serializer.data)

    elif request.method == 'PUT':
        serializer = SnippetSerializer(snippet, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    elif request.method == 'DELETE':
        snippet.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

This should all feel very familiar - it is not a lot different from working with regular Django views.

Notice that we're no longer explicitly tying our requests or responses to a given content type. `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.

Adding optional format suffixes to our URLs

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 .

Start by adding a `format` keyword argument to both of the views, like so.

```
def snippet_list(request, format=None):
```

and

```
def snippet_detail(request, pk, format=None):
```

Now update the `snippets/urls.py` file slightly, to append a set of `format_suffix_patterns` in addition to the existing URLs.

```
from django.urls import path
from rest_framework.urlpatterns import format_suffix_patterns
from snippets import views

urlpatterns = [
    path('snippets/', views.snippet_list),
    path('snippets/<int:pk>/', views.snippet_detail),
]

urlpatterns = format_suffix_patterns(urlpatterns)
```

We don't necessarily need to add these extra url patterns in, but it gives us a simple, clean way of referring to a specific format.

How's it looking?

Go ahead and test the API from the command line, as we did in tutorial part 1 . Everything is working pretty similarly, although we've got some nicer error handling if we send invalid requests.

We can get a list of all of the snippets, as before.

```
http http://127.0.0.1:8000/snippets/

HTTP/1.1 200 OK
...
[
  {
    "id": 1,
    "title": "",
    "code": "foo = \"bar\"\n",
    "linenos": false,
    "language": "python",
    "style": "friendly"
  },
  {
    "id": 2,
    "title": "",
    "code": "print(\"hello, world\")\n",
    "linenos": false,
    "language": "python",
    "style": "friendly"
  }
]
```

We can control the format of the response that we get back, either by using the `Accept` header:

```
http http://127.0.0.1:8000/snippets/ Accept:application/json  # Request JSON
http http://127.0.0.1:8000/snippets/ Accept:text/html         # Request HTML
```

Or by appending a format suffix:

```
http http://127.0.0.1:8000/snippets.json  # JSON suffix
http http://127.0.0.1:8000/snippets.api   # Browsable API suffix
```

Similarly, we can control the format of the request that we send, using the `Content-Type` header.

```
# POST using form data
http --form POST http://127.0.0.1:8000/snippets/ code="print(123)"

{
  "id": 3,
  "title": "",
  "code": "print(123)",
  "linenos": false,
  "language": "python",
  "style": "friendly"
}

# POST using JSON
http --json POST http://127.0.0.1:8000/snippets/ code="print(456)"

{
    "id": 4,
    "title": "",
    "code": "print(456)",
    "linenos": false,
    "language": "python",
    "style": "friendly"
}
```

If you add a `--debug` switch to the `http` requests above, you will be able to see the request type in request headers.

Now go and open the API in a web browser, by visiting http://127.0.0.1:8000/snippets/ .

Browsability

Because the API chooses the content type of the response based on the client request, it will, by default, return an HTML-formatted representation of the resource when that resource is requested by a web browser. This allows for the API to return a fully web-browsable HTML representation.

Having a web-browsable API is a huge usability win, and makes developing and using your API much easier. It also dramatically lowers the barrier-to-entry for other developers wanting to inspect and work with your API.

See the browsable api topic for more information about the browsable API feature and how to customize it.

What's next?

In tutorial part 3 , we'll start using class-based views, and see how generic views reduce the amount of code we need to write.