





 Hands-On RESTful Python Web Services - Second Edition

 [PREV](#)
[Improving testing coverage](#)

[NEXT](#) 
[Test your knowledge](#)

Running Django RESTful APIs on the cloud

One of the biggest drawbacks related to Django and Django REST Framework is that each HTTP request is blocking. Thus, whenever the Django server receives an HTTP request, it doesn't start working on any other HTTP requests in the incoming queue until the server sends the response for the first HTTP request is received.

However, one of the great advantages of RESTful Web Services is that they are stateless; that is, they shouldn't keep a client state on any server. Our API is a good example of a stateless RESTful Web Service. Thus, we can make the API run on as many servers as necessary to achieve our scalability goals. Obviously, we must take into account that we can easily transform the database server in our scalability bottleneck.

Nowadays, we have a huge number of cloud-based alternatives with which to deploy a RESTful Web Service that uses Django and Django REST Framework and make it extremely scalable. Just to mention a few examples, we have Heroku, PythonAnywhere, Google App Engine, OpenShift, AWS Elastic Beanstalk, and Microsoft Azure.

There are dozens of deployment options for Django and Django REST Framework, and the different coverages for stacks and procedures offered by each option are out of the scope of this book, which is focused on development tasks for RESTful APIs with the most popular Python frameworks. The most important cloud providers include instructions on how to deploy Django applications with diverse possible configurations. In addition, there are many options to use **WSGI** (short for **Web Server Gateway Interface**) servers that implement the web server side of the WSGI interface, which allow us to run Python web applications such as Django applications in production.

Of course, in a production environment, we will want to work with HTTPS instead of HTTP. We will have to configure the appropriate TLS certificates, also known as SSL certificates.

Our API is a good example of a stateless RESTful Web Service with Django, Django REST Framework, and PostgreSQL 10.5 that can be containerized in a Docker container. For example, we can produce an image with our application configured to run with NGINX, uWSGI, Redis, and Django. Thus, we can make the API run as a microservice.

We always have to make sure that we profile the API and the database before we deploy our first version of our API. It is very important to make sure that the generated queries run properly on the underlying database, and that the most popular queries do not end up in sequential scans. It is usually necessary to add the appropriate indexes to the tables in the database.

We have been using basic HTTP authentication. We can improve it with token-based authentication and configure additional authentication plugins available for Django and Django REST Framework.

Each platform includes detailed instructions to deploy our application. All of them will require us to generate the `requirements.txt` file that lists the application dependencies, together with their versions. This way, the platforms will be able to install all the necessary dependencies listed in the file. We have been updating this file each time we needed to install a new package in our virtual environment. However, it is a good idea to run the following `pip freeze` within the root folder of our virtual environment, `django01`, to generate the final `requirements.txt` file.

Run the following `pip freeze` to generate the `requirements.txt` file:

```
pip freeze > requirements.txt
```

The following lines show the contents of a sample generated `requirements.txt` file. Notice that the generated file also includes all the dependencies that were installed by the packages we specified in the original `requirements.txt` file:

```
atomicwrites==1.2.1
attrs==18.2.0
certifi==2018.10.15
chardet==3.0.4
coverage==4.5.2
Django==2.1.4
django-filter==2.0.0
djangorestframework==3.9.0
httplib==1.0.2
idna==2.7
more-itertools==4.3.0
pluggy==0.8.0
psycopg2==2.7.5
py==1.7.0
Pygments==2.2.0
pytest==4.0.2
pytest-cov==2.6.0
pytest-django==3.4.4
pytz==2018.6
requests==2.20.0
six==1.11.0
urllib3==1.24
```

We must make sure that we change the following line in the `settings.py` file:

```
DEBUG = True
```

We must always turn off the debug mode in production, and therefore, we must replace the previous line with the following one:

DEBUG = False

[Settings](#) / [Support](#) / [Sign Out](#)

◀ PREV
[Improving testing coverage](#)

NEXT ▶
[Test your knowledge](#)