**Dockerized Django Back-end API with Angular Front-end Tutorial (Part 1)**

* 8th December 2018
* / [Development](https://dragonprogrammer.com/category/development/)
* / By [Dragos Stanciu](https://dragonprogrammer.com/author/dnstanciu/)
* / [6 COMMENTS](https://dragonprogrammer.com/dockerized-django-api-angular-tutorial/#comments)

In this multipart tutorial, you will learn how to build a production-ready REST API using Python and the Django REST Framework (DRF). You will interact with the API using an Angular app as the front-end layer. We will also cover authentication and deploying the stack as a Docker container in a “live” production environment.

**Tutorial Table of Contents**

1. Building a Basic API with Django REST Framework in Docker (this post)
2. [Adding Detail Views to a Django DRF API in Docker](https://dragonprogrammer.com/adding-detail-views-django-drf-api/)
3. [Building an Angular App in a Secure and Compact Docker Image](https://dragonprogrammer.com/building-angular-docker/)
4. [Connecting an Angular App to a REST API](https://dragonprogrammer.com/connecting-angular-api/)
5. [Improving an Angular Todo App to Manage Tasks via a REST API](https://dragonprogrammer.com/angular-app-manage-tasks-api/)
6. [Securing a Django REST Framework API Using Auth0](https://dragonprogrammer.com/securing-django-api-auth0/)
7. [Securing an Angular App Using Auth0 and JWT Authentication](https://dragonprogrammer.com/securing-angular-auth0/)
8. [Adding Object Level Permissions in Django REST Framework](https://dragonprogrammer.com/object-level-permissions-drf/)
9. [Preparing a Django DRF API for Production in Docker](https://dragonprogrammer.com/django-drf-api-production-docker/)
10. [Preparing an Angular App for Production in Docker](https://dragonprogrammer.com/angular-app-production-docker/)
11. [Building a Docker Image in Google Cloud Build](https://dragonprogrammer.com/building-docker-image-google-cloud-build/)
12. [Building a Docker Image from a Private GitHub or Bitbucket Repository in Google Cloud Build](https://dragonprogrammer.com/docker-image-private-repo-cloud-build/)
13. [Connecting Django to a PostgreSQL Database in Cloud SQL](https://dragonprogrammer.com/connect-django-database-cloud-sql/)
14. [Deploying a Docker Container on a VM in Google Cloud Platform](https://dragonprogrammer.com/deploy-docker-container-gcp/)
15. [Serving an Angular App’s Static Files from Google Cloud Storage](https://dragonprogrammer.com/serve-angular-files-cloud-storage/)

**GitHub repo**: <https://github.com/dnstanciu/drf-angular-docker-tutorial>

**Overview**

The Django back-end will be purely an API that is consumed by our Angular Single-Page Application (SPA).

We will use [Auth0](https://auth0.com/) as our authentication provider for the API and Angular app.

The whole application stack will be bundled as a Docker image. We will also learn how to deploy the image as a container using Google Cloud Platform.

**Technologies Used**

* Django 2.1.5
* Django REST framework 3.9.0
* Angular 7.2
* Docker Engine 18.09.0

In this first part of the tutorial, we will build a basic REST API using the Django Rest Framework (DRF) in Docker.

The API back-end will allow a user to create and update tasks as part of a simple Todo application.

Django is a mature and powerful Python web framework and allows fast development given its existing packages provided by the Django community.

[Django REST framework](https://www.django-rest-framework.org/) is a flexible tool for building web APIs. A few of its advantages are as follows:

* out-of-the-box web browsable API – this is especially useful for developers.
* authentication policies
* serialization
* good documentation

Docker is a tool that allows you to bundle your application and all of its libraries and dependencies into a Docker image. This image can then be easily deployed as a container in the cloud using a cloud service provider such as Google Cloud Platform.

If you don’t currently use Docker, definitely do yourself a favour and invest a bit of time to learn it. Especially if you’re doing web development, it will save you a lot of time troubleshooting and checking that development and production dependencies coincide.

**Initial Project Setup**

Before we get into building the actual API, let’s do some preparations.

The following workflow has worked well for me for more complex projects so I hope you’ll also find it useful. Let me know in the comments if you have any tips on how I can make the process smoother.

First of all, we’ll create a repository for our application.

I will use this Github repo for the tutorial: <https://github.com/dnstanciu/drf-angular-docker-tutorial>

I recommend you follow along, and if something doesn’t work, you can checkout the repo at various stages in the tutorial using the tags I’ve created.

In a later part of the tutorial I’ll also show you how to build Docker images from private GitHub or Bitbucket repositories.

I’ll use only one repository for both front-end and back-end code. One could separate the two layers in two repos, but because the only consumer of the API is the Angular app and because of the app’s relatively simple complexity, it makes sense to keep everything in one repo for this tutorial.

**1. Creating the Dockerfile for the Django API**

First of all, in the repo root, let’s create a directory for holding the Django back-end code. In the terminal, run:

$ mkdir django

Next, in the django directory, create a requirements.txt. This will contain the dependencies for our API. For now, add the following to this file:

|  |
| --- |
|  |
| Django==2.1.4 | |
|  | |

|  |
| --- |
| djangorestframework==3.9.0 |
|  |

|  |
| --- |
| psycopg2 |

[view raw](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50/raw/8be679344f35d9e15bbcf8b11e3967f992248484/requirements.txt) [requirements.txt](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50#file-requirements-txt) hosted with ❤ by [GitHub](https://github.com)

This simply means we add the Django, DRF and Python PostgreSQL adapter packages to be later included in our Docker image.

Then, also in the django directory, create a Dockerfile file with the contents:

|  |
| --- |
|  |
| FROM python:3 | |
|  | |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Set environment variables |
|  |

|  |
| --- |
| ENV PYTHONUNBUFFERED 1 |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| COPY requirements.txt / |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Install dependencies. |
|  |

|  |
| --- |
| RUN pip install -r /requirements.txt |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Set work directory. |
|  |

|  |
| --- |
| RUN mkdir /code |
|  |

|  |
| --- |
| WORKDIR /code |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Copy project code. |
|  |

|  |
| --- |
| COPY . /code/ |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| EXPOSE 80 |

[view raw](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50/raw/8be679344f35d9e15bbcf8b11e3967f992248484/Dockerfile) [Dockerfile](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50#file-dockerfile) hosted with ❤ by [GitHub](https://github.com)

This Dockerfile uses the basic Python 3 Docker image as a starting point.

Setting PYTHONUNBUFFERED to 1 means that log messages will be immediately dumped to the standard output stream instead of being buffered.

We then proceed to copy the requirements.txt file to the Docker image and install the dependencies using pip.

Next, we create a /code directory on the image which will hold the project code. We set that directory as the working directory on the image.

Afterwards, we copy the project code (i.e. what’s in the django directory in the repository) in the /code directory using the COPY command.

Further, port 80 is exposed on the container which means that the container will listen on port 80 at runtime.

Lastly, we specify that we want to start the Django development web server when the container will be started. We will use this for local testing, but in production we need to use a proper web server such as [uWSGI](https://uwsgi-docs.readthedocs.io/en/latest/) or [Gunicorn](https://gunicorn.org/).

From the django directory, test that you can successfully build the image from the terminal using:

$ docker build .

As a side note, there is a reason why we copy the project code after we install the dependencies.

The reason is that Docker caches the build steps of the image in order to save image build time.

Copying the project code after dependencies are installed means that Docker will cache the image to include the installed dependencies. This means that when you change your project’s source code, Docker will rebuild the image from the point where we copy the project’s code onto the image, without reinstalling dependencies.

**2. Creating the docker-compose File**

In this tutorial, we’ll run multiple containers locally for development purposes. In production, we’ll be running a single container.

Using [Docker Compose](https://docs.docker.com/compose/), we can run and coordinate multiple Docker containers. One could also use a Compose file as a basis for migrating towards a multi-container deployment setup using [Kubernetes](https://kubernetes.io/), but this is out of scope for this tutorial series.

So, let’s create a docker-compose.yaml file in the root of the repository with the following contents:

|  |
| --- |
|  |
| version: "3" | |
|  | |

|  |
| --- |
|  |
|  |

|  |
| --- |
| services: |
|  |

|  |
| --- |
| dj: |
|  |

|  |
| --- |
| container\_name: dj |
|  |

|  |
| --- |
| build: django |
|  |

|  |
| --- |
| command: python manage.py runserver 0.0.0.0:80 |
|  |

|  |
| --- |
| volumes: |
|  |

|  |
| --- |
| - ./django:/code |
|  |

|  |
| --- |
| ports: |
|  |

|  |
| --- |
| - "80:80" |

[view raw](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50/raw/8be679344f35d9e15bbcf8b11e3967f992248484/docker-compose_v1.yaml) [docker-compose\_v1.yaml](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50#file-docker-compose_v1-yaml) hosted with ❤ by [GitHub](https://github.com)

Here we specify that we are using version 3 of the Compose file format.

In the services definition, we specify what configuration should be applied to each container.

For now, we only have one container that’s running Django.

Using container-name, we specify that the container should be named dj instead of a generated default name.

In build, we give the path to Django’s Dockerfile we created in the previous step.

The command definition tells the Docker container to start the Django development server on port 80 on container start-up.

In volumes, we map the django directory in our repo to the /code directory in the Docker container.

In ports, we map port 80 on the host machine with port 80 on the container in the HOST:CONTAINER format.

**Creating the Django Project and API App**

We are now ready to create our Django project and app for our API.

1. Change to the repo root.
2. Create a Django starter project by running the [docker-compose run](https://docs.docker.com/compose/reference/run/) command from the terminal:

$ docker-compose run dj django-admin.py startproject todoproj .

Compose will run django-admin.py startproject todoproj . in a container, using the dj service’s image and configuration. This will create the files representing a Django project.

FYI, the . at the end of the command will tell the django-admin.py startproject command to create the Django project in the existing directory, i.e. /code as we’ve set that to be the working directory in the Dockerfile.

1. Due to the volume mapping we’ve done in the Compose file, we can now find the new Django project files in the django directory of the repo. Let’s list them:

$ cd django

$ ls -l

total 24

-rw-r--r-- 1 dragos staff 310 5 Dec 12:53 Dockerfile

-rwxr-xr-x 1 dragos staff 540 5 Dec 13:11 manage.py

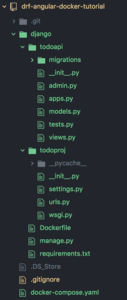
-rw-r--r-- 1 dragos staff 50 5 Dec 12:42 requirements.txt

drwxr-xr-x 6 dragos staff 192 5 Dec 13:11 todoproj

1. We can now create a Django app to hold our models and views. From the repo root, run the following:

$ docker-compose run dj python manage.py startapp todoapi

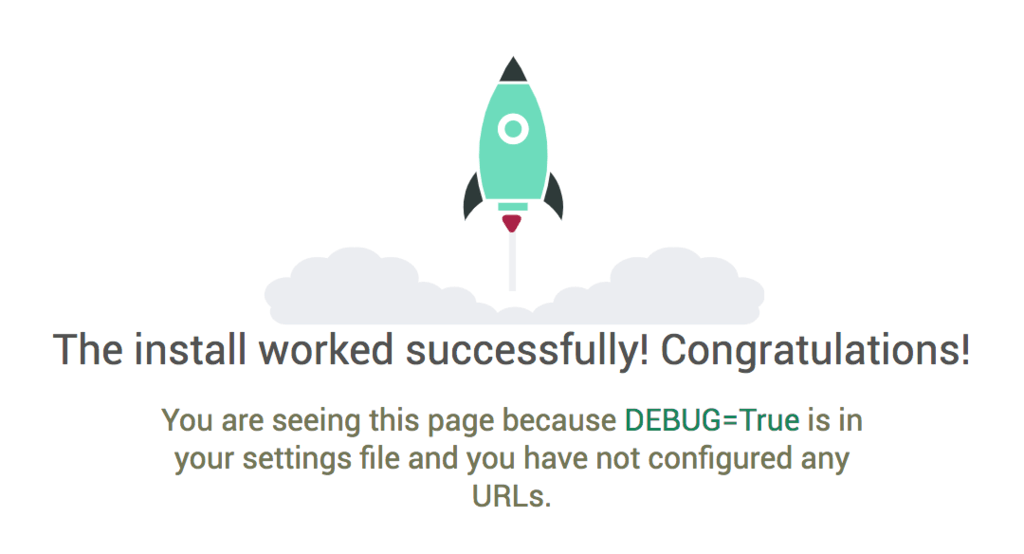
Here is the project file structure after running all of the above:

[](https://dragonprogrammer.com/wp-content/uploads/2018/12/project_file_structure.png)

1. You can now run the project and check that it works by running the following command from the repo root:

$ docker-compose up

Nice! If you now go to <http://localhost:80> you should see the Django welcome screen, meaning that the development server is running properly:



**Connecting to the PostgreSQL Database**

In this section, we will connect Django to a PostgreSQL database.

By default, Django uses SQLite as its database, which is bundled with Python. For both development and production, we want to use a more robust database engine, namely PostgreSQL.

In development, we’ll actually spin off a new Docker container that runs PostgreSQL. In production, we’ll use a PostgreSQL database hosted on Cloud SQL, which is a database service part of Google Cloud Platform.

1. Add a Database Service to docker-compose.

In this step, we’ll add a database service in our docker-compose file that the Django container will connect to.

Update your docker-compose.yaml file as follows:

|  |
| --- |
|  |
| version: "3" | |
|  | |

|  |
| --- |
|  |
|  |

|  |
| --- |
| services: |
|  |

|  |
| --- |
| db: |
|  |

|  |
| --- |
| image: postgres |
|  |

|  |
| --- |
| dj: |
|  |

|  |
| --- |
| container\_name: dj |
|  |

|  |
| --- |
| build: django |
|  |

|  |
| --- |
| command: python manage.py runserver 0.0.0.0:80 |
|  |

|  |
| --- |
| volumes: |
|  |

|  |
| --- |
| - ./django:/code |
|  |

|  |
| --- |
| ports: |
|  |

|  |
| --- |
| - "80:80" |
|  |

|  |
| --- |
| depends\_on: |
|  |

|  |
| --- |
| - db |

[view raw](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50/raw/8be679344f35d9e15bbcf8b11e3967f992248484/docker-compose_v2.yaml) [docker-compose\_v2.yaml](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50#file-docker-compose_v2-yaml) hosted with ❤ by [GitHub](https://github.com)

The db container uses the Docker postgres image. By default, this image exposes port 5432 (the postgres port) and includes a default postgres user and postgres database. We will use this database for the Django container.

The depends\_on key tells Compose that the dj (Django) service is dependent on the db service.

1. Modify [settings.py](http://settings.py) in the Django Project.

Next, we need to tell Django to use the PostgreSQL database instead of the default SQLite one.

In the Django project directory, edit the todoproj/settings.py file and replace the DATABASES = {...} section with the following:

|  |
| --- |
|  |
| DATABASES = { | |
|  | |

|  |
| --- |
| 'default': { |
|  |

|  |
| --- |
| 'ENGINE': 'django.db.backends.postgresql', |
|  |

|  |
| --- |
| 'NAME': 'postgres', |
|  |

|  |
| --- |
| 'USER': 'postgres', |
|  |

|  |
| --- |
| 'HOST': 'db', |
|  |

|  |
| --- |
| 'PORT': 5432, |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |

[view raw](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50/raw/8be679344f35d9e15bbcf8b11e3967f992248484/settings_databases.py) [settings\_databases.py](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50#file-settings_databases-py) hosted with ❤ by [GitHub](https://github.com)

1. From the repo root, re-run the docker-compose up command.
2. List running containers.

To see that the two containers are running, execute the docker ps in a new terminal:

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

4862ac29cce4 drf-angular-docker-tutorial\_dj "python manage.py ru…" 15 minutes ago Up 15 minutes 0.0.0.0:80->80/tcp dj

83195aa165c6 postgres "docker-entrypoint.s…" 15 minutes ago Up 15 minutes 5432/tcp drf-angular-docker-tutorial\_db\_1

1. Shutdown services using docker-compose down from the repo root, in a new terminal:

$ docker-compose down

Stopping dj ... done

Stopping drf-angular-docker-tutorial\_db\_1 ... done

Removing dj ... done

Removing drf-angular-docker-tutorial\_db\_1 ... done

Removing network drf-angular-docker-tutorial\_default

Great job! Now you have a nice Dockerized development environment, with two containers: one for Django and the other for your PostgreSQL database.

You can checkout the code at this point in the tutorial using:

$ git checkout v1.0

**Creating the Data Model**

In this section, we will create our model, which is essentially the database structure for our app.

Since our API will allow listing and updating tasks as part of a Todo app, let’s create a simple Task model in the todoapi/models.py file:

|  |
| --- |
|  |
| from django.db import models | |
|  | |

|  |
| --- |
|  |
|  |

|  |
| --- |
| from datetime import date |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Create the Task class to describe the model. |
|  |

|  |
| --- |
| class Task(models.Model): |
|  |

|  |
| --- |
| """Stores a task.""" |
|  |

|  |
| --- |
| title = models.CharField(max\_length=50) |
|  |

|  |
| --- |
| content = models.CharField(max\_length=50) |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Date the task was created. |
|  |

|  |
| --- |
| created\_on = models.DateField(default=date.today) |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Due date. |
|  |

|  |
| --- |
| due\_date = models.DateField(default=date.today) |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Meta data about the database table. |
|  |

|  |
| --- |
| class Meta: |
|  |

|  |
| --- |
| # Set the table name. |
|  |

|  |
| --- |
| db\_table = 'task' |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Set default ordering |
|  |

|  |
| --- |
| ordering = ['id'] |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Define what to output when the model is printed as a string. |
|  |

|  |
| --- |
| def \_\_str\_\_(self): |
|  |

|  |
| --- |
| return self.title |

[view raw](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50/raw/8be679344f35d9e15bbcf8b11e3967f992248484/models.py) [models.py](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50#file-models-py) hosted with ❤ by [GitHub](https://github.com)

For the Django project to recognise the todoapi application, we need to add it to INSTALLED\_APPS in todoproj/settings.py like so:

|  |
| --- |
|  |
| INSTALLED\_APPS = [ | |
|  | |

|  |
| --- |
| 'django.contrib.admin', |
|  |

|  |
| --- |
| 'django.contrib.auth', |
|  |

|  |
| --- |
| 'django.contrib.contenttypes', |
|  |

|  |
| --- |
| 'django.contrib.sessions', |
|  |

|  |
| --- |
| 'django.contrib.messages', |
|  |

|  |
| --- |
| 'django.contrib.staticfiles', |
|  |

|  |
| --- |
| 'rest\_framework', |
|  |

|  |
| --- |
| 'todoapi', |
|  |

|  |
| --- |
| ] |

[view raw](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50/raw/8be679344f35d9e15bbcf8b11e3967f992248484/settings_installed_apps_v2.py) [settings\_installed\_apps\_v2.py](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50#file-settings_installed_apps_v2-py) hosted with ❤ by [GitHub](https://github.com)

We now need to create and apply the database migrations in order to update our database schema.

To do this, we first connect to the Django container. First spin up the containers with docker-compose up as usual, then from a new terminal, run:

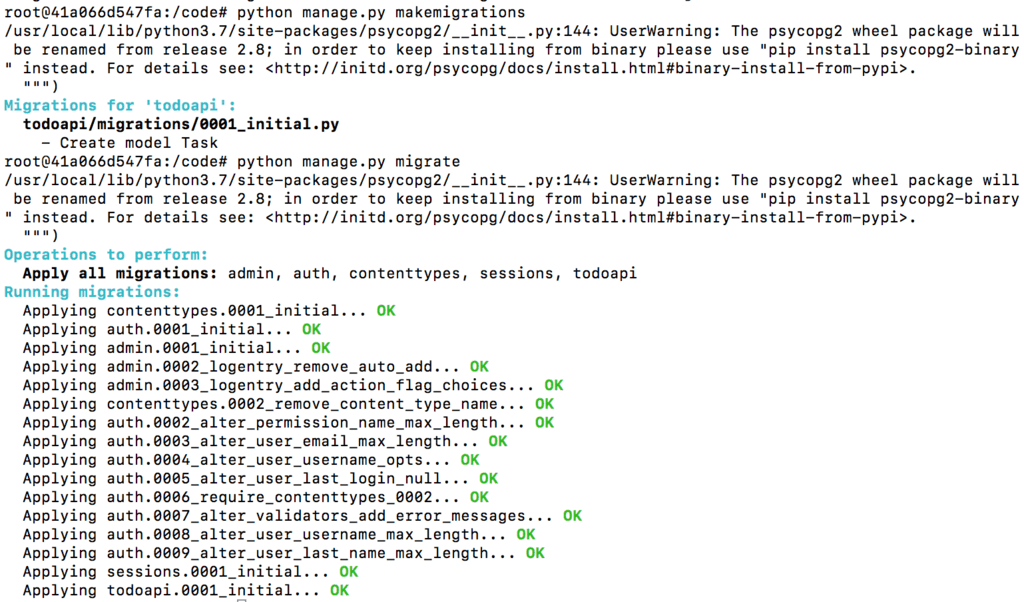
$ docker exec -it dj bash

The docker exec command allows you to run commands inside an existing container.

The -it part is a combination of -i and -t, which stands for --interactive (“Keep STDIN open even if not attached”) and --tty (“Allocate a pseudo-TTY”), respectively.

Lastly, bash is the command to run in the container.

Now that we are in the container, let’s create the migration files using python manage.py makemigrations and apply them using python manage.py migrate:



**Creating the API Serializer**

[Serializers](https://www.django-rest-framework.org/api-guide/serializers/) allow conversion from received data in formats such as JSON to Python objects and viceversa.

Create a new file todoapi/serializers.py containing:

|  |
| --- |
|  |
| from rest\_framework import serializers | |
|  | |

|  |
| --- |
| from .models import Task |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| class TaskSerializer(serializers.ModelSerializer): |
|  |

|  |
| --- |
| class Meta: |
|  |

|  |
| --- |
| model = Task |
|  |

|  |
| --- |
| fields = ('title', 'content', 'created\_on', 'due\_date') |

[view raw](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50/raw/8be679344f35d9e15bbcf8b11e3967f992248484/serializers.py) [serializers.py](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50#file-serializers-py) hosted with ❤ by [GitHub](https://github.com)

The ModelSerializer class easily creates a serializer with fields corresponding to our Task model fields.

The fields attribute is used to specify which fields we want to use in the model serializer.

**Creating the API View**

Next, we need to implement a View in order to handle incoming API requests.

In Django, a View is simply a function that takes an HTTP request and returns an HTTP response.

For our initial version of the view, we’ll use the APIView from the Django REST framework to implement handling GET requests to list all our Task objects. Modify todoapi/views.py as follows:

|  |
| --- |
|  |
| from .models import Task | |
|  | |

|  |
| --- |
| from .serializers import TaskSerializer |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| from rest\_framework.views import APIView |
|  |

|  |
| --- |
| from rest\_framework.response import Response |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| class TaskList(APIView): |
|  |

|  |
| --- |
| """ |
|  |

|  |
| --- |
| View all tasks. |
|  |

|  |
| --- |
| """ |
|  |

|  |
| --- |
| def get(self, request, format=None): |
|  |

|  |
| --- |
| """ |
|  |

|  |
| --- |
| Return a list of all tasks. |
|  |

|  |
| --- |
| """ |
|  |

|  |
| --- |
| tasks = Task.objects.all() |
|  |

|  |
| --- |
| serializer = TaskSerializer(tasks, many=True) |
|  |

|  |
| --- |
| return Response(serializer.data) |

[view raw](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50/raw/8be679344f35d9e15bbcf8b11e3967f992248484/views_v1.py) [views\_v1.py](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50#file-views_v1-py) hosted with ❤ by [GitHub](https://github.com)

We define a get() method to handle incoming GET requests.

Task.objects.all() queries the database and retrieves all Task objects as a Django queryset.

We then pass this queryset to an instance of TaskSerializer, specifying many=True. This tells the serializer that we want to serialize a collection of objects, and not just a single instance.

Lastly, with the return Response(serializer.data), we return the list of serialized Task objects.

**Testing the API View**

There are a couple of more things to do before we can test our API:

1. We need to tell Django to map the TaskList API view to a URL. Modify todoproj/urls.py as follows:

|  |
| --- |
|  |
| from django.contrib import admin | |
|  | |

|  |
| --- |
| from django.urls import path |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| from todoapi import views as todoapi\_views |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| urlpatterns = [ |
|  |

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

|  |
| --- |
| path('api/task/', todoapi\_views.TaskList.as\_view(), name='task-list'), |
|  |

|  |
| --- |
| ] |

[view raw](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50/raw/8be679344f35d9e15bbcf8b11e3967f992248484/urls.py) [urls.py](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50#file-urls-py) hosted with ❤ by [GitHub](https://github.com)

Here we simply map the api/task/ endpoint to a callable TaskList view. A callable view simply means that the view takes a request object and returns a response object. We get this callable view by using the as\_view() function.

1. Add rest\_framework to our list of INSTALLED\_APPS in todoproj/settings.py:

|  |
| --- |
|  |
| INSTALLED\_APPS = [ | |
|  | |

|  |
| --- |
| 'django.contrib.admin', |
|  |

|  |
| --- |
| 'django.contrib.auth', |
|  |

|  |
| --- |
| 'django.contrib.contenttypes', |
|  |

|  |
| --- |
| 'django.contrib.sessions', |
|  |

|  |
| --- |
| 'django.contrib.messages', |
|  |

|  |
| --- |
| 'django.contrib.staticfiles', |
|  |

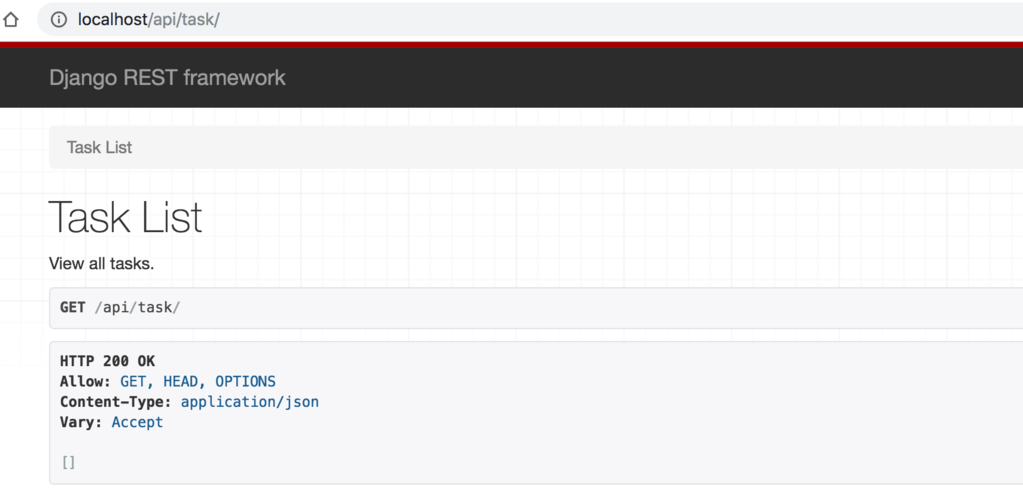
|  |
| --- |
| 'rest\_framework', |
|  |

|  |
| --- |
| 'todoapi', |
|  |

|  |
| --- |
| ] |

[view raw](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50/raw/8be679344f35d9e15bbcf8b11e3967f992248484/settings_installed_apps_v2.py) [settings\_installed\_apps\_v2.py](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50#file-settings_installed_apps_v2-py) hosted with ❤ by [GitHub](https://github.com)

If you now restart the Django container, you can go to <http://localhost/api/task/> to see if the endpoint works. Because we haven’t actually created any Task objects yet, the response will be an empty list.



Nice! You can checkout the code at this point using:

$ git checkout v1.1

If you get an error such as “*relation task does not exist*”, it means you need to apply migrations to your database.

**Creating Initial Dummy Data**

To make sure that the API can actually return data, we can create a data migration file that will populate the database with a few dummy Task items.

First, create an empty migration file while connected to your Django container:

$ python manage.py makemigrations --empty todoapi --name dummy\_tasks

Now open the new file in todoapi/migrations/0002\_dummy\_tasks.py and add the following:

|  |
| --- |
|  |
| from django.db import migrations | |
|  | |

|  |
| --- |
|  |
|  |

|  |
| --- |
| def create\_dummy\_tasks(apps, schema\_editor): |
|  |

|  |
| --- |
| Task = apps.get\_model('todoapi', 'Task') |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| Task(title='Workout', content='Squat, bench press, deadlift.').save() |
|  |

|  |
| --- |
| Task(title='Shopping', content='Whey protein, creatine, BCAAs.').save() |
|  |

|  |
| --- |
| Task(title='Counting', content='1, 2, 3 and so on.').save() |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| class Migration(migrations.Migration): |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| dependencies = [ |
|  |

|  |
| --- |
| ('todoapi', '0001\_initial'), |
|  |

|  |
| --- |
| ] |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| operations = [ |
|  |

|  |
| --- |
| migrations.RunPython(create\_dummy\_tasks), |
|  |

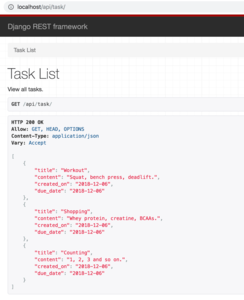
|  |
| --- |
| ] |

[view raw](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50/raw/8be679344f35d9e15bbcf8b11e3967f992248484/0002_dummy_tasks.py) [0002\_dummy\_tasks.py](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50#file-0002_dummy_tasks-py) hosted with ❤ by [GitHub](https://github.com)

To apply this migration, run from inside the container:

$ python manage.py migrate

If you now refresh the page at <http://localhost/api/task/>, you should see the dummy data returned by the API:

[](https://dragonprogrammer.com/wp-content/uploads/2018/12/task_api_dummy_data.png)

**Simplifying the API View**

In this last section of this part of the tutorial, we will simplify the API view we wrote previously.

Edit the todoapi/views.py file and replace the contents with:

|  |
| --- |
|  |
| from .models import Task | |
|  | |

|  |
| --- |
| from .serializers import TaskSerializer |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| from rest\_framework import generics |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| class TaskList(generics.ListAPIView): |
|  |

|  |
| --- |
| """ |
|  |

|  |
| --- |
| View all tasks. |
|  |

|  |
| --- |
| """ |
|  |

|  |
| --- |
| queryset = Task.objects.all() |
|  |

|  |
| --- |
| serializer = TaskSerializer |

[view raw](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50/raw/8be679344f35d9e15bbcf8b11e3967f992248484/views_v2.py) [views\_v2.py](https://gist.github.com/dnstanciu/131e95b3dd83475fc0e65a9b177a2c50#file-views_v2-py) hosted with ❤ by [GitHub](https://github.com)

Pretty concise, right? Here we’re making use of the [ListAPIView](https://www.django-rest-framework.org/api-guide/generic-views/#listapiview) class. This creates a read-only endpoint that represents a collection of items, thus handling GET requests. We only need to specify the queryset and the serializer\_class for this view.

ListAPIView is a generic view provided by DRF. Generic views make use of inheritance and are useful for building common API behaviours to interact with your database models.

You can read more about them in the [generic views](https://www.django-rest-framework.org/api-guide/generic-views/) documentation. As a bonus for you, a good resource for seeing what each class-based view inherits is the website [Classy DRF](http://www.cdrf.co/).

You can get to this stage of the code using:

$ git checkout v1.2

**Summary**

In this part of the tutorial, we’ve built a simple REST API using the Django REST Framework. The app is dockerized, with two Docker containers: one for the Django API and one for the PostgreSQL database that Django uses.

In the next part of the tutorial, we will look at creating and updating individual Task objects using the API.

**Adding Detail Views to a Django DRF API in Docker (Part 2)**

* 10th December 2018
* / [Development](https://dragonprogrammer.com/category/development/)
* / By [Dragos Stanciu](https://dragonprogrammer.com/author/dnstanciu/)

In this part of the tutorial, we will extend the basic Django DRF API built in the last part to handle creating, updating, and deleting items.

*This post is part of the* [*Dockerized Django Back-end API with Angular Front-end Tutorial*](https://dragonprogrammer.com/dockerized-django-api-angular-tutorial/)*. Check out all the parts of the tutorial there.*

To get to the point where we’ve finished in the last tutorial, use:

$ git checkout v1.2

**Implementing the POST Method Using APIView**

At the moment, our API is read-only as we can only retrieve Task objects. In this section, we’ll implement a POST method for creating new tasks.

Instead of using the generic ListAPIView view, let’s go back to the basic APIView to see things more explicitly. Modify todoapi/views.py as below:

|  |
| --- |
|  |
| from .models import Task | |
|  | |

|  |
| --- |
| from .serializers import TaskSerializer |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| from rest\_framework.views import APIView |
|  |

|  |
| --- |
| from rest\_framework.response import Response |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| class TaskList(APIView): |
|  |

|  |
| --- |
| """ |
|  |

|  |
| --- |
| View all tasks. |
|  |

|  |
| --- |
| """ |
|  |

|  |
| --- |
| def get(self, request, format=None): |
|  |

|  |
| --- |
| """ |
|  |

|  |
| --- |
| Return a list of all tasks. |
|  |

|  |
| --- |
| """ |
|  |

|  |
| --- |
| tasks = Task.objects.all() |
|  |

|  |
| --- |
| serializer = TaskSerializer(tasks, many=True) |
|  |

|  |
| --- |
| return Response(serializer.data) |

[view raw](https://gist.github.com/dnstanciu/18e1499fd7970d647d6cd7911bb4b754/raw/3fb5a55e10a134aede925dfae5d37599ad135949/views_list_apiview.py) [views\_list\_apiview.py](https://gist.github.com/dnstanciu/18e1499fd7970d647d6cd7911bb4b754#file-views_list_apiview-py) hosted with ❤ by [GitHub](https://github.com)

Let’s add this import to todoapi/views.py:

from rest\_framework import status

Then, add the POST method to the TaskList class:

|  |
| --- |
|  |
| def post(self, request, format=None): | |
|  | |

|  |
| --- |
| """ |
|  |

|  |
| --- |
| Create a task. |
|  |

|  |
| --- |
| """ |
|  |

|  |
| --- |
| serializer = TaskSerializer(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) |

[view raw](https://gist.github.com/dnstanciu/18e1499fd7970d647d6cd7911bb4b754/raw/3fb5a55e10a134aede925dfae5d37599ad135949/views_list_apiview_post.py) [views\_list\_apiview\_post.py](https://gist.github.com/dnstanciu/18e1499fd7970d647d6cd7911bb4b754#file-views_list_apiview_post-py) hosted with ❤ by [GitHub](https://github.com)

Here, we check that the deserialization was valid before we save the new task to the database. If all goes OK, we send back the created task with a HTTP 201 Created status. Otherwise, we send a 400 Bad Request with the errors from the serializer.

Let’s test the POST method using [Postman](https://www.getpostman.com/).

Open Postman and on a new tab, select POST from the drop-down request menu.

Next, enter <http://localhost/api/task/> as the request URL. After, click the *Body* subtab and choose *raw*. To the right of this, select *JSON (application/json)* from the drop-down list instead of *Text*.

Now enter the following JSON body in the text box below:

|  |
| --- |
|  |
| { | |
|  | |

|  |
| --- |
| "title":"My Task Title", |
|  |

|  |
| --- |
| "content":"The task content." |
|  |

|  |
| --- |
| } |

[view raw](https://gist.github.com/dnstanciu/18e1499fd7970d647d6cd7911bb4b754/raw/3fb5a55e10a134aede925dfae5d37599ad135949/task_post.json) [task\_post.json](https://gist.github.com/dnstanciu/18e1499fd7970d647d6cd7911bb4b754#file-task_post-json) hosted with ❤ by [GitHub](https://github.com)

Click *Send*. At the bottom of the screen, you should see the returned JSON string:

|  |
| --- |
|  |
| { | |
|  | |

|  |
| --- |
| "title": "My Task Title", |
|  |

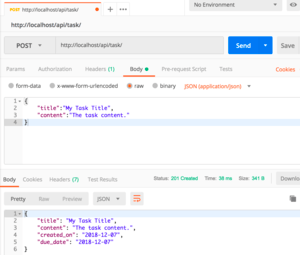
|  |
| --- |
| "content": "The task content.", |
|  |

|  |
| --- |
| "created\_on": "2018-12-07", |
|  |

|  |
| --- |
| "due\_date": "2018-12-07" |
|  |

|  |
| --- |
| } |

[view raw](https://gist.github.com/dnstanciu/18e1499fd7970d647d6cd7911bb4b754/raw/3fb5a55e10a134aede925dfae5d37599ad135949/task_post_returned.json) [task\_post\_returned.json](https://gist.github.com/dnstanciu/18e1499fd7970d647d6cd7911bb4b754#file-task_post_returned-json) hosted with ❤ by [GitHub](https://github.com)

[](https://dragonprogrammer.com/wp-content/uploads/2018/12/postman_POST_test.png)

If you now refresh the <http://localhost/api/task/> page in your browser, you should see your new task returned with the other dummy data. Alternatively, you can also send a GET request from Postman by choosing GET from the request menu.

You can checkout the code at this stage:

$ git checkout v1.3

**Implementing the POST Method Using ListCreateAPIView**

Actually, we don’t need to explicitly implement the POST method. The ListCreateAPIView generic view provides the POST method out-of-the-box. Modify todoapi/views.py as follows:

|  |
| --- |
|  |
| from .models import Task | |
|  | |

|  |
| --- |
| from .serializers import TaskSerializer |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| from rest\_framework import generics |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| class TaskList(generics.ListCreateAPIView): |
|  |

|  |
| --- |
| """ |
|  |

|  |
| --- |
| Lists and creates tasks. |
|  |

|  |
| --- |
| """ |
|  |

|  |
| --- |
| queryset = Task.objects.all() |
|  |

|  |
| --- |
| serializer\_class = TaskSerializer |

[view raw](https://gist.github.com/dnstanciu/18e1499fd7970d647d6cd7911bb4b754/raw/3fb5a55e10a134aede925dfae5d37599ad135949/views_list_generic.py) [views\_list\_generic.py](https://gist.github.com/dnstanciu/18e1499fd7970d647d6cd7911bb4b754#file-views_list_generic-py) hosted with ❤ by [GitHub](https://github.com)

The ListCreateAPIView does all the validation behind the scenes using the specified serializer.

**Handling Updates and Deletions**

In this section, we will implement handling individual object details. This is needed if we want to delete or update a specific Task object.

Firstly, we need to modify the TaskSerializer class to also return the ID of the task. Change todoapi/serializers.py:

|  |
| --- |
|  |
| class TaskSerializer(serializers.ModelSerializer): | |
|  | |

|  |
| --- |
| class Meta: |
|  |

|  |
| --- |
| model = Task |
|  |

|  |
| --- |
| fields = ('id', 'title', 'content', 'created\_on', 'due\_date') |

[view raw](https://gist.github.com/dnstanciu/18e1499fd7970d647d6cd7911bb4b754/raw/3fb5a55e10a134aede925dfae5d37599ad135949/serializers.py) [serializers.py](https://gist.github.com/dnstanciu/18e1499fd7970d647d6cd7911bb4b754#file-serializers-py) hosted with ❤ by [GitHub](https://github.com)

**Implementation using APIView**

Let’s implement a new TaskDetail view in todoapi/views.py. Add the imports:

|  |
| --- |
|  |
| from django.http import Http404 | |
|  | |

|  |
| --- |
| from rest\_framework.views import APIView |
|  |

|  |
| --- |
| from rest\_framework.response import Response |
|  |

|  |
| --- |
| from rest\_framework import status |

[view raw](https://gist.github.com/dnstanciu/18e1499fd7970d647d6cd7911bb4b754/raw/3fb5a55e10a134aede925dfae5d37599ad135949/views_detail_apiview_imports.py) [views\_detail\_apiview\_imports.py](https://gist.github.com/dnstanciu/18e1499fd7970d647d6cd7911bb4b754#file-views_detail_apiview_imports-py) hosted with ❤ by [GitHub](https://github.com)

And the new detail view:

|  |
| --- |
|  |
| class TaskDetail(APIView): | |
|  | |

|  |
| --- |
| """ |
|  |

|  |
| --- |
| Returns a single Task and allows updates and deletion of a Task. |
|  |

|  |
| --- |
| """ |
|  |

|  |
| --- |
| def get\_object(self, task\_id): |
|  |

|  |
| --- |
| try: |
|  |

|  |
| --- |
| return Task.objects.get(pk=task\_id) |
|  |

|  |
| --- |
| except Task.DoesNotExist: |
|  |

|  |
| --- |
| raise Http404 |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| def get(self, request, task\_id, format=None): |
|  |

|  |
| --- |
| task = self.get\_object(task\_id) |
|  |

|  |
| --- |
| serializer = TaskSerializer(task) |
|  |

|  |
| --- |
| return Response(serializer.data) |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| def put(self, request, task\_id, format=None): |
|  |

|  |
| --- |
| task = self.get\_object(task\_id) |
|  |

|  |
| --- |
| serializer = TaskSerializer(task, data=request.data) |
|  |

|  |
| --- |
| if serializer.is\_valid(): |
|  |

|  |
| --- |
| serializer.save() |
|  |

|  |
| --- |
| return Response(serializer.data) |
|  |

|  |
| --- |
| return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST) |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| def delete(self, request, task\_id, format=None): |
|  |

|  |
| --- |
| task = self.get\_object(task\_id) |
|  |

|  |
| --- |
| task.delete() |
|  |

|  |
| --- |
| return Response(status=status.HTTP\_204\_NO\_CONTENT) |

[view raw](https://gist.github.com/dnstanciu/18e1499fd7970d647d6cd7911bb4b754/raw/3fb5a55e10a134aede925dfae5d37599ad135949/views_detail_apiview.py) [views\_detail\_apiview.py](https://gist.github.com/dnstanciu/18e1499fd7970d647d6cd7911bb4b754#file-views_detail_apiview-py) hosted with ❤ by [GitHub](https://github.com)

We also need to map the new view to a URL, so edit todoproj/urls.py:

|  |
| --- |
|  |
| urlpatterns = [ | |
|  | |

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

|  |
| --- |
| path('api/task/', todoapi\_views.TaskList.as\_view(), name='task-list'), |
|  |

|  |
| --- |
| path('api/task/<int:task\_id>/', todoapi\_views.TaskDetail.as\_view(), name='task-detail'), |
|  |

|  |
| --- |
| ] |

[view raw](https://gist.github.com/dnstanciu/18e1499fd7970d647d6cd7911bb4b754/raw/3fb5a55e10a134aede925dfae5d37599ad135949/urls.py) [urls.py](https://gist.github.com/dnstanciu/18e1499fd7970d647d6cd7911bb4b754#file-urls-py) hosted with ❤ by [GitHub](https://github.com)

We are now able to send GET, PUT and DELETE requests to a specific task using a link such as /api/task/1/. This allows us to retrieve, update or remove an existing task. The task\_id is a URL keyword argument used for object lookup.

**Implementation using RetrieveUpdateDestroyAPIView**

Finally, let’s use generic views and refactor the TaskDetail class to be more compact. For this, update todoapi/views.py using the [RetrieveUpdateDestroyAPIView](https://www.django-rest-framework.org/api-guide/generic-views/#retrieveupdatedestroyapiview) view:

|  |
| --- |
|  |
| class TaskDetail(generics.RetrieveUpdateDestroyAPIView): | |
|  | |

|  |
| --- |
| """ |
|  |

|  |
| --- |
| Returns a single Task and allows updates and deletion of a Task. |
|  |

|  |
| --- |
| """ |
|  |

|  |
| --- |
| queryset = Task.objects.all() |
|  |

|  |
| --- |
| serializer\_class = TaskSerializer |
|  |

|  |
| --- |
| lookup\_url\_kwarg = 'task\_id' |

[view raw](https://gist.github.com/dnstanciu/18e1499fd7970d647d6cd7911bb4b754/raw/3fb5a55e10a134aede925dfae5d37599ad135949/views_detail_generic.py) [views\_detail\_generic.py](https://gist.github.com/dnstanciu/18e1499fd7970d647d6cd7911bb4b754#file-views_detail_generic-py) hosted with ❤ by [GitHub](https://github.com)

Here, lookup\_url\_kwarg specifies which URL keyword argument should be used for object lookup.

In the end, we have the same functionality as the previous more explicit implementation using APIView.

To get to this point in the tutorial, use:

$ git checkout v1.4

**Summary**

In this part of the tutorial, we have extended our Django DRF API to handle creation, update and deletion of individual Task items.

In the next part, we’ll build a dockerized Angular app that will be used to interact with the Django API.

Credit: I liked the approach used in [this blog post](https://www.andreagrandi.it/2017/03/12/creating-a-production-ready-api-with-python-and-django-rest-framework-part-3/) of first explaining how to implement the HTTP methods manually before using generic views, so I also followed it in this tutorial.

# Building an Angular App in a Secure and Compact Docker Image (Part 3)

* 12th December 2018
* / [Development](https://dragonprogrammer.com/category/development/)
* / By [Dragos Stanciu](https://dragonprogrammer.com/author/dnstanciu/)
* / [4 COMMENTS](https://dragonprogrammer.com/building-angular-docker/#comments)

This article will teach you how you can use Docker to build and run an Angular application.

We will create a lightweight and efficient Docker image using Docker’s multi-stage builds feature to serve the app using NGINX.

This post is part of the [*Dockerized Django Back-end API with Angular Front-end Tutorial*](https://dragonprogrammer.com/dockerized-django-api-angular-tutorial/). Check out all the parts of the tutorial there.

In the [last part](https://dragonprogrammer.com/adding-detail-views-django-drf-api/) of the tutorial, we’ve created a simple dockerized Django API that can create Todo tasks. In this blog post, we’ll move towards the front-end and build the basis of our Angular app to interact with our API.

To get the code to where we left off in the last blog post, use:

$ git checkout v1.4

## Creating an Empty Angular App to Dockerize

Firstly, let’s create a new directory in our repo root to hold all Angular related code and cd inside it:

$ mkdir angular

$ cd angular

Next, let’s start an initial Angular project using the [Angular CLI](https://cli.angular.io/).

If you don’t already have the Angular CLI, install it using:

$ npm install -g @angular/cli@latest

Next, create a new application using:

$ ng new angular-app

Select Yes for Angular routing. For the stylesheet format, I chose CSS, but you can select whatever you want.

Run the app to ensure it compiles and works correctly.

$ cd angular-app

$ ng serve

Visit <http://localhost:4200> in your browser to confirm the skeleton app works.

You can also checkout the code with the skeleton app using:

$ git checkout v1.5

When developing the app, it’s sufficient to just use ng serve, as the app is dynamically rebuilt on file changes.

When it comes to moving towards a production deployment, setting up a Docker image makes sense as you’ll see shortly.

## Creating the Docker Image in Stages

In this section, we will build a Docker image that can be used to serve our Angular app.

The image will be built in two stages:

1. Build Stage – use a Node Alpine Docker image to compile our app.
2. Delivery Stage – serve files from build stage using an NGINX Alpine Docker image.

### Stage 1 – Compiling the Angular App

Create a Dockerfile file in the angular directory of the repo root.

In the first stage, we will use a node Alpine image as our base Docker image:

FROM node:11.4.0-alpine as builder

We name this stage builder so we can reference it later in the second stage.

Then, we create and set a working directory:

RUN mkdir /app

WORKDIR /app

After, we copy and install our app dependencies:

# Copy app dependencies.

COPY angular-app/package.json angular-app/package-lock.json /app/angular-app/

# Install app dependencies.

RUN npm install --prefix angular-app

The --prefix option installs the dependencies in the angular-app directory.

Lastly, we copy the app files and build it in ./dist/out:

# Copy app files.

COPY . /app

# Build app

RUN npm run build --prefix angular-app -- --output-path=./dist/out

We copy the app files after installing dependencies so that if package.json doesn’t change, Docker will cache the installation step so later image builds will be faster.

Stage 1 looks as follows:

|  |
| --- |
|  |
| ### Stage 1: build ### | |
|  | |

|  |
| --- |
|  |
|  |

|  |
| --- |
| FROM node:11.4.0-alpine as builder |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Set working directory. |
|  |

|  |
| --- |
| RUN mkdir /app |
|  |

|  |
| --- |
| WORKDIR /app |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Copy app dependencies. |
|  |

|  |
| --- |
| COPY angular-app/package.json angular-app/package-lock.json /app/angular-app/ |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Install app dependencies. |
|  |

|  |
| --- |
| RUN npm install --prefix angular-app |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Copy app files. |
|  |

|  |
| --- |
| COPY . /app |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Build app |
|  |

|  |
| --- |
| RUN npm run build --prefix angular-app -- --output-path=./dist/out |

[view raw](https://gist.github.com/dnstanciu/604f014dc7227874f94d2e2452b78f8a/raw/b0ffc8ea11b5520a8f1c2eed750e98a5fca33632/Dockerfile_stage_1) [Dockerfile\_stage\_1](https://gist.github.com/dnstanciu/604f014dc7227874f94d2e2452b78f8a#file-dockerfile_stage_1) hosted with ❤ by [GitHub](https://github.com)

#### Ignoring files using .dockerignore

Before moving on to the second stage, there is one issue that can cause potential problems.

The second COPY command copies everything, including the node\_modules. This means that the dependencies we’ve installed on the Docker image using RUN npm install will be overwritten.

We can create a .dockerignore file in the angular directory to tell Docker that the node\_modules directory should be ignored when copying files. Just create the file with the contents:

|  |
| --- |
|  |
| angular-app/node\_modules | |

[view raw](https://gist.github.com/dnstanciu/604f014dc7227874f94d2e2452b78f8a/raw/b0ffc8ea11b5520a8f1c2eed750e98a5fca33632/.dockerignore) [.dockerignore](https://gist.github.com/dnstanciu/604f014dc7227874f94d2e2452b78f8a#file-dockerignore) hosted with ❤ by [GitHub](https://github.com)

Thanks to Lukas for this tip from his [article](https://malcoded.com/posts/angular-docker).

### Stage 2 – Serving the Angular App Using NGINX

Using [multi-stage builds](https://docs.docker.com/develop/develop-images/multistage-build/), we can start from a new image with NGINX installed and just copy the built app from the builder stage.

Therefore, we use an NGINX Alpine image as the base:

FROM nginx:1.15.7-alpine

Then, we remove the default NGINX website:

RUN rm -rf /usr/share/nginx/html/\*

After, we copy the compiled app from the builder image to the NGINX public folder:

COPY --from=builder /app/angular-app/dist/out /usr/share/nginx/html

Lastly, we need to copy an NGINX configuration file to the Docker image. We’ll create this file next.

COPY ./nginx/nginx.conf /etc/nginx/conf.d/default.conf

In the end, stage 2 will be added right after stage 1 in the same Dockerfile, and looks as follows:

|  |
| --- |
|  |
| ### Stage 2: delivery ### | |
|  | |

|  |
| --- |
|  |
|  |

|  |
| --- |
| FROM nginx:1.15.7-alpine |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Remove default nginx website |
|  |

|  |
| --- |
| RUN rm -rf /usr/share/nginx/html/\* |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Copy output directory from builder to nginx image. |
|  |

|  |
| --- |
| COPY --from=builder /app/angular-app/dist/out /usr/share/nginx/html |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Copy nginx configuration file. |
|  |

|  |
| --- |
| COPY ./nginx/nginx.conf /etc/nginx/conf.d/default.conf |

[view raw](https://gist.github.com/dnstanciu/604f014dc7227874f94d2e2452b78f8a/raw/b0ffc8ea11b5520a8f1c2eed750e98a5fca33632/Dockerfile_stage_2) [Dockerfile\_stage\_2](https://gist.github.com/dnstanciu/604f014dc7227874f94d2e2452b78f8a#file-dockerfile_stage_2) hosted with ❤ by [GitHub](https://github.com)

## Creating a Default NGINX Configuration File

We need to inform NGINX which files to serve. Create a new directory under the angular one called nginx.

$ mkdir nginx

Inside this directory, create a NGINX configuration file called nginx.conf with the contents:

|  |
| --- |
|  |
| server { | |
|  | |

|  |
| --- |
| listen 80; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| gzip on; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| location / { |
|  |

|  |
| --- |
| root /usr/share/nginx/html; |
|  |

|  |
| --- |
| index index.html index.htm; |
|  |

|  |
| --- |
| try\_files $uri $uri/ /index.html =404; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |

[view raw](https://gist.github.com/dnstanciu/604f014dc7227874f94d2e2452b78f8a/raw/b0ffc8ea11b5520a8f1c2eed750e98a5fca33632/nginx.conf) [nginx.conf](https://gist.github.com/dnstanciu/604f014dc7227874f94d2e2452b78f8a#file-nginx-conf) hosted with ❤ by [GitHub](https://github.com)

Here, we tell NGINX to listen on port 80 and for requested files that can’t be found, serve index.html.

## Updating the docker-compose File

We can now spin up a container serving the Angular app via NGINX using the docker-compose.yaml file we created in the first part of the tutorial.

In the repo root, update docker-compose.yaml as follows:

|  |
| --- |
|  |
| version: "3" | |
|  | |

|  |
| --- |
|  |
|  |

|  |
| --- |
| services: |
|  |

|  |
| --- |
| db: |
|  |

|  |
| --- |
| image: postgres |
|  |

|  |
| --- |
| dj: |
|  |

|  |
| --- |
| container\_name: dj |
|  |

|  |
| --- |
| build: django |
|  |

|  |
| --- |
| command: python manage.py runserver 0.0.0.0:80 |
|  |

|  |
| --- |
| volumes: |
|  |

|  |
| --- |
| - ./django:/code |
|  |

|  |
| --- |
| ports: |
|  |

|  |
| --- |
| - "80:80" |
|  |

|  |
| --- |
| depends\_on: |
|  |

|  |
| --- |
| - db |
|  |

|  |
| --- |
| ng: |
|  |

|  |
| --- |
| container\_name: ng |
|  |

|  |
| --- |
| build: angular |
|  |

|  |
| --- |
| ports: |
|  |

|  |
| --- |
| - "8080:80" |

[view raw](https://gist.github.com/dnstanciu/604f014dc7227874f94d2e2452b78f8a/raw/b0ffc8ea11b5520a8f1c2eed750e98a5fca33632/docker-compose.yaml) [docker-compose.yaml](https://gist.github.com/dnstanciu/604f014dc7227874f94d2e2452b78f8a#file-docker-compose-yaml) hosted with ❤ by [GitHub](https://github.com)

We’ve just added a new service called ng and told Docker to map port 8080 on the host to port 80 on the container, where NGINX listens for requests.

If you execute docker-compose up from the repo root, you can then see a new ng container running concurrently with the Django and the PostgreSQL ones:

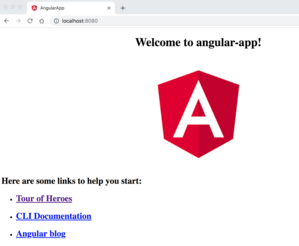
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

67933b65acac drf-angular-docker-tutorial\_dj "python manage.py ru…" 5 minutes ago Up 5 minutes 0.0.0.0:80->80/tcp dj

c478693e2dd3 postgres "docker-entrypoint.s…" 5 minutes ago Up 5 minutes 5432/tcp drf-angular-docker-tutorial\_db\_1

a6b6c7dc7ab8 drf-angular-docker-tutorial\_ng "nginx -g 'daemon of…" 5 minutes ago Up 5 minutes 0.0.0.0:8080->80/tcp ng

If you visit <http://localhost:8080>, you can see the app being served via NGINX.

[](https://dragonprogrammer.com/wp-content/uploads/2018/12/angular_running.png)

Awesome! To get to this point of the tutorial, just:

$ git checkout v1.6

## Summary

In this part of the tutorial, we’ve learned how to build an Angular application using Docker and how to serve it in an NGINX container.

In the next blog post, we’ll see how we can consume the Django REST API using the Angular app.

Credit: I used the following resources as a basis for this article:

* [Create efficient Angular Docker images with Multi Stage Builds](https://medium.com/@avatsaev/create-efficient-angular-docker-images-with-multi-stage-builds-907e2be3008d)
* [Create a small and secured Angular Docker image based on nginx](https://medium.com/@chemidy/create-a-small-and-secure-angular-docker-image-based-on-nginx-93452cb08aa2)
* [Run Angular in a Docker Container using Multi-Stage builds](https://malcoded.com/posts/angular-docker)