

Gastón C. Hillar

Django RESTful Web Services

The easiest way to build Python RESTful APIs and web services with Django



Packt

Preface

Python is definitely one of the most popular programming languages. It is open source, multiplatform, and you can use it to develop any kind of application, from websites and web services to artificial intelligence and machine learning applications. You will always find a framework or a set of packages in Python to make things easier for you in any domain. It is extremely easy to work with Python and its most popular web framework, Django, in the most important and popular cloud computing providers. Hence, Python is an excellent choice for developing modern and scalable RESTful Web Services that will run on the cloud.

REST (short for Representational State Transfer) is the architectural style that has been driving modern and scalable web development recently. If you want to be part of the world that builds complex web applications and mobile apps, you will need to develop and interact with RESTful Web Services. In many situations, you will have to design and develop a RESTful Web Service from scratch and maintain the API over time. A deep knowledge of RESTful Web Services is an essential skill for any software development job.

This book covers everything you need to know to develop and test a RESTful Web Service from scratch with the latest version of Django, Django REST framework, and Python. You will work with real-life examples in combination with Python packages that will allow you to simplify tasks.

You will learn to use a huge set of tools to test and develop a uniform, high-quality, and scalable RESTful Web Services. You will use object-oriented programming and modern Python 3.6 code to favor code reuse and simplify future maintenance. You will take advantage of automated testing to ensure that the coded RESTful Web Services run as expected.

This book will allow you to create your own RESTful Web Services for any domain with Django and Django REST framework in Python 3.6 or greater. You will learn the process for the most popular Python platforms: Linux, Windows, and macOS.

Who this book is for

This book is for Python developers who want to develop RESTful Web Services, also known as RESTful web APIs, with Python 3.6 or greater, and want to learn how to do it with the most popular Python web framework—Django.

What this book covers

Chapter 1, *Installing the Required Software and Tools*, shows how to get started in our journey toward creating RESTful Web Services with Python and its most popular web framework—Django. We will install and configure the environments, the software, and the tools required to create RESTful Web Services with Django and Django REST framework. We will learn the necessary steps in Linux, macOS, and Windows. We will create our first app with Django, we will take a first look at the Django folders, files, and configurations, and we will make the necessary changes to activate Django REST framework. In addition, we will introduce and install command-line and GUI tools that we will use to interact with the RESTful Web Services that we will design, code, and test in the forthcoming chapters.

Chapter 2, *Working with Models, Migrations, Serialization, and Deserialization*, describes how to design a RESTful Web Service to interact with a simple SQLite database and perform CRUD operations with toys. We will define the requirements for our web service, and we will understand the tasks performed by each HTTP method and the different scopes. We will create a model to represent and persist toys and execute migrations in Django to create the required tables in the database. We will analyze the tables and learn how to manage the serialization of toy instances into JSON representations with Django REST framework and the reverse process.

Chapter 3, *Creating API Views*, is about executing the first version of a simple Django RESTful Web Service that interacts with a SQLite database. We will write API views to process diverse HTTP requests on a collection of toys and on a specific toy. We will work with the following HTTP verbs: GET, POST, and PUT. We will configure the URL patterns list to route URLs to views. We will start the Django development server and use command-line tools (curl and HTTPie) to compose and send

diverse HTTP requests to our RESTful Web Service. We will learn how HTTP requests are processed in Django and our code. In addition, we will work with Postman, a GUI tool, to compose and send other HTTP requests to our RESTful Web Service.

[Chapter 4](#), *Using Generalized Behavior from the APIView Class*, presents different ways to improve our simple Django RESTful Web Service. We will take advantage of many features included in the Django REST framework to remove duplicate code and add many features for the web service. We will use model serializers, understand the different accepted and returned content types, and the importance of providing accurate responses to the HTTP OPTIONS requests. We will make the necessary changes to the existing code to enable diverse parsers and renderers. We will learn how things work under the hoods in Django REST framework. We will work with different content types and note how the RESTful Web Service improves compared to its previous versions.

[Chapter 5](#), *Understanding and Customizing the browsable API Feature*, explains how to use one of the additional features that Django REST framework adds to our RESTful Web Service—the browsable API. We will use a web browser to work with our first web service built with Django. We will learn to make HTTP GET, POST, PUT, OPTIONS, and DELETE requests with the browsable API. We will be able to easily test CRUD operations with a web browser. The browsable API will allow us to easily interact with our RESTful Web Service.

[Chapter 6](#), *Working with Advanced Relationships and Serialization*, shows how to define the requirements for a complex RESTful Web Service in which we needed to work with drone categories, drones, pilots, and competitions. We will create a new app with Django and configure the new web service. We will define many-to-one relationships between the models, and we will configure Django to work with a PostgreSQL database. We will execute migrations to generate tables with relationships between them. We will also analyze the generated database and configure serialization and deserialization for the models. We will define hyperlinks and work with class-based views. Then, we will take advantage of generic classes and generic views that generalize and mix predefined behaviors.

We will use routings and endpoints and prepare our RESTful Web Service to work with the browsable API. We will make many different HTTP requests to create and retrieve resources that have relationships between them.

[Chapter 7, Using Constraints, Filtering, Searching, Ordering, and Pagination](#), describes the usage of the browsable API feature to navigate through the API with resources and relationships. We will add unique constraints to improve the consistency of the models in our RESTful Web Service. We will understand the importance of paginating results and configure and test a global limit/offset pagination scheme with Django REST framework. Then, we will create our own customized pagination class to ensure that requests won't be able to require a huge number of elements on a single page. We will configure filter backend classes and incorporate code into the models to add filtering, searching, and ordering capabilities to the class-based views. We will create a customized filter and make requests to filter, search, and order results. Finally, we will use the browsable API to test pagination, filtering, and ordering.

[Chapter 8, Securing the API with Authentication and Permissions](#), presents the differences between authentication and permissions in Django, Django REST framework, and RESTful Web Services. We will analyze the authentication classes included in Django REST framework out of the box. We will follow the steps needed to provide security- and permissions-related data to models.

We will work with object-level permissions via customized permission classes and save information about users who make requests. We will configure permission policies and compose and send authenticated requests to understand how the permission policies work. We will use command-line tools and GUI tools to compose and send authenticated requests. We will browse the secure RESTful Web Service with the browsable API feature and work with a simple token-based authentication provided by Django REST framework to understand another way of authenticating requests.

[Chapter 9, Applying Throttling Rules and Versioning Management](#), focuses

on the importance of throttling rules and how we can combine them with authentication and permissions in Django, Django REST framework, and RESTful Web Services. We will analyze the throttling classes included in Django REST framework out of the box. We will follow the necessary steps to configure many throttling policies in Django REST framework. We will work with global and scope-related settings. Then, we will use command-line tools to compose and send many requests to test how the throttling rules work. We will understand versioning classes and we will configure a URL path versioning scheme to allow us to work with two versions of our RESTful Web Service. We will use command-line tools and the Browsable API to understand the differences between the two versions.

[Chapter 10](#), *Automating Tests*, shows how to automate tests for our RESTful Web Services developed with Django and Django REST framework. We will use different packages, tools, and configurations to perform tests. We will write the first round of unit tests for our RESTful Web Service, run them, and measure tests code coverage. Then, we will analyze tests code coverage reports and write new unit tests to improve the test code coverage. We will understand the new tests code coverage reports and learn the benefits of a good test code coverage.

[Appendix](#), *Solutions*, the right answers for the *Test Your Knowledge* sections of each chapter are included in the appendix.

To get the most out of this book

Any computer or device capable of running Python 3.6.3 or greater in Linux, macOS, or Windows.

Any computer or device capable of running a modern web browser compatible with HTML 5 and CSS 3 to work with the Browsable API feature included in Django REST framework.

Download the example code files

You can download the example code files for this book from your account at www.packtpub.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packtpub.com.
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Ziipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Django-RESTful-Web-Services>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: http://www.packtpub.com/sites/default/files/downloads/DjangoRESTfulWebServices_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
from django.shortcuts import render

# Create your views here.
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
from django.conf.urls import url, include

urlpatterns = [
    url(r'^$', include('drones.urls')),
    url(r'^api-auth/', include('rest_framework.urls'))
]
```

Any command-line input or output is written as follows:

```
http :8000/toys/
```

```
curl -iX GET localhost:8000/toys/3
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select System info from the Administration panel."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit : packtpub.com.

Installing the Required Software and Tools

In this chapter, we will start our journey towards creating RESTful Web Services with Python and its most popular web framework: Django. Python is one of the most popular and versatile programming languages. There are thousands of Python packages, which allow you to extend Python capabilities to any kind of domain you can imagine. You can work with Django and packages to easily build simple and complex RESTful Web Services with Python that can run on your favorite platform.

We will leverage your existing knowledge of Python and all of its packages to code the different pieces of your RESTful Web Services and their ecosystem. We will use object-oriented features to create code that is easier to maintain, understand, and reuse. We don't need to learn another programming language, we can use the one we already know and love: Python.

In this chapter, we will install and configure the environments and the required software and tools to create RESTful Web Services with Django and Django REST framework. We will learn the necessary steps in Linux, macOS, and Windows. We will gain an understanding of the following:

- Creating a virtual environment with Python 3.x and PEP 405
- Understanding the directory structure for a virtual environment
- Activating the virtual environment
- Deactivating the virtual environment
- Installing Django and Django REST framework in an isolated

environment

- Creating an app with Django
- Understanding Django folders, files, and configurations
- Installing Curl
- Installing HTTPie
- Installing the Postman REST client
- Installing Stoplight
- Installing iCurlHTTP

Creating a virtual environment with Python 3.x and PEP 405

Throughout this book, we will be working with different packages and libraries to create RESTful Web Services, and therefore it is convenient to work with Python virtual environments. Python 3.3 introduced lightweight virtual environments and they were improved in Python 3.4. We will work with these virtual environments, and therefore you will need Python 3.4 or greater. You can read more information about PEP 405 Python Virtual Environment, that introduced the `venv` module, here: <https://www.python.org/dev/peps/pep-0405>. All the examples in this book were tested on Python 3.6.2 on Linux, macOS, and Windows.

In case you decide to use the popular `virtualenv` (<https://pypi.python.org/pypi/virtualenv>) third-party virtual environment builder or the virtual environment options provided by your Python IDE, you just have to make sure that you activate your virtual environment with the appropriate mechanism whenever it is necessary to do so, instead of following the step explained to activate the virtual environment generated with the `venv` module integrated in Python.

Each virtual environment we create with `venv` is an isolated environment and it will have its own independent set of installed Python packages in its site directories (folders). When we create a virtual environment with `venv` in Python 3.4 and greater, `pip` is included in the new virtual environment. In Python 3.3, it was necessary to manually install `pip` after creating the virtual environment. Note that the instructions provided are compatible with Python 3.4 or greater, including Python 3.6.2.

In order to create a lightweight virtual environment, the first step is to select the target folder or directory for it. The following is the path we will

use in the example for Linux and macOS.

The target folder for the virtual environment will be the `HillarDjangoREST/01` folder within our home directory. For example, if our home directory in macOS or Linux is `/users/gaston`, the virtual environment will be created within `/users/gaston/HillarDjangoREST/01`. You can replace the specified path with your desired path in each command:



```
~/HillarDjangoREST/01
```

The following is the path we will use in the example for Windows. The target folder for the virtual environment will be the `HillarDjangoREST\01` folder within our user profile folder. For example, if our user profile folder is `c:\users\gaston`, the virtual environment will be created within `c:\Users\gaston\HillarDjangoREST\01`. You can replace the specified path with your desired path in each command:



```
%USERPROFILE%\HillarDjangoREST\01
```

In Windows PowerShell, the previous path would be as follows:



```
$env:UserProfile\HillarDjangoREST\01
```

Now, we will create a new virtual environment with `venv`. In order to do so, we have to use the `-m` option followed by the `venv` module name and the desired path to make Python run this module as a script and create a virtual environment in the specified path. The instructions are different depending on the platform in which we are creating the virtual environment.

Open Terminal in Linux or macOS and execute the following command to create a virtual environment:

```
python3 -m venv ~/HillarDjangoREST/01
```

In Windows, in Command Prompt, execute the following command to create a virtual environment:

```
python -m venv %USERPROFILE%\HillarDjangoREST\01
```

If you want to work with Windows PowerShell, execute the following command to create a virtual environment:

```
python -m venv $env:userprofile\HillarDjangoREST\01
```

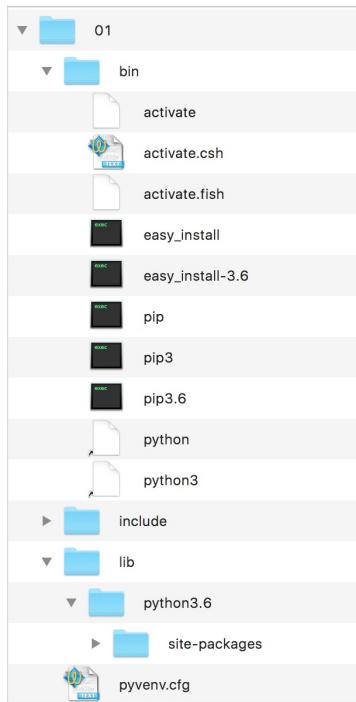
None of the previous commands produce any output. The script created the specified target folder and installed `pip` by invoking `ensurepip` because we didn't specify the `--without-pip` option.

Understanding the directory structure for a virtual environment

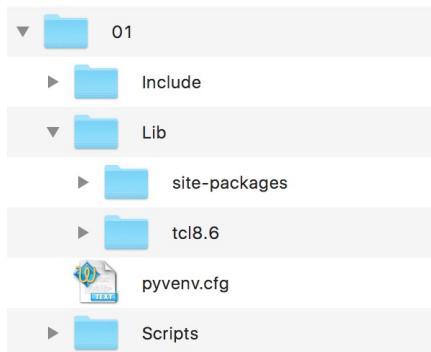
The specified target folder has a new directory tree that contains Python executable files and other files that indicate it is a PEP405 virtual environment.

In the root directory for the virtual environment, the `pyenv.cfg` configuration file specifies different options for the virtual environment and its existence is an indicator that we are in the root folder for a virtual environment. In Linux and macOS, the folder will have the following main subfolders: `bin`, `include`, `lib`, `lib/python3.6`, and `lib/python3.6/site-packages`. In Windows, the folder will have the following main subfolders: `Include`, `Lib`, `Lib\site-packages`, and `scripts`. The directory trees for the virtual environment in each platform are the same as the layout of the Python installation on these platforms.

The following diagram shows the folders and files in the directory trees generated for the `01` virtual environments in macOS and Linux platforms:



The following diagram shows the main folders in the directory trees generated for the virtual environment in Windows:



After we activate the virtual environment, we will install third-party packages into the virtual environment and the modules will be located in the lib/python3.6/site-packages or Lib\site-packages folder, based on the platform. The executables will be copied in the bin or scripts folder, based on the platform. The packages we install won't make changes to other virtual environments or our base Python environment.

Activating the virtual environment

Now that we have created a virtual environment, we will run a platform-specific script to activate it. After we activate the virtual environment, we will install packages that will only be available in this virtual environment. This way, we will work with an isolated environment in which all the packages we install won't affect our main Python environment.

Note that the results of this command will be accurate if you don't start a different shell than the default shell in the terminal session. If you have doubts, check your terminal configuration and preferences. Run the following command in the Terminal in Linux or macOS:

```
echo $SHELL
```

The command will display the name of the shell you are using in the Terminal. In macOS, the default is `/bin/bash` and this means you are working with the `bash` shell. Depending on the shell, you must run a different command to activate the virtual environment in Linux or macOS.

If your Terminal is configured to use the `bash` shell in Linux or macOS, run the following command to activate the virtual environment. The command also works for the `zsh` shell:

```
source ~/HillarDjangoREST/01/bin/activate
```

If your Terminal is configured to use either the `csh` or `tcsh` shell, run the following command to activate the virtual environment:

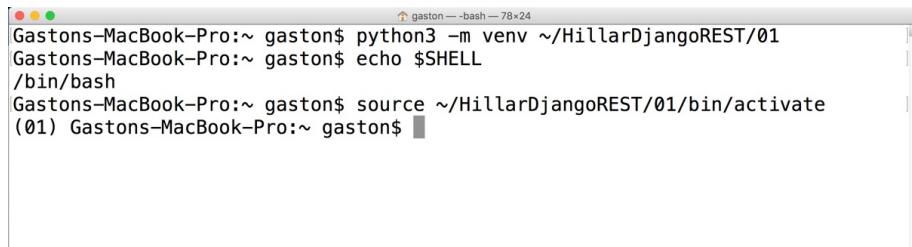
```
source ~/HillarDjangoREST/01/bin/activate.csh
```

If your Terminal is configured to use the `fish` shell, run the following command to activate the virtual environment:

```
source ~/HillarDjangoREST/01/bin/activate.fish
```

After you activate the virtual environment, Command Prompt will display the virtual environment root folder name enclosed in parentheses as a prefix of the default prompt to remind us that we are working in the virtual environment. In this case, we will see `(01)` as a prefix for the Command Prompt because the root folder for the activated virtual environment is `01`.

The following screenshot shows the virtual environment activated in a macOS Sierra Terminal with a `bash` shell, after executing the previously shown commands:

A screenshot of a macOS Sierra Terminal window titled "gaston". The window shows the following command history:

```
Gastons-MacBook-Pro:~ gaston$ python3 -m venv ~/HillarDjangoREST/01
Gastons-MacBook-Pro:~ gaston$ echo $SHELL
/bin/bash
Gastons-MacBook-Pro:~ gaston$ source ~/HillarDjangoREST/01/bin/activate
(01) Gastons-MacBook-Pro:~ gaston$
```

The terminal window has a standard OS X look with red, yellow, and green window controls.

As we can see from the previous screenshot, the prompt changed from

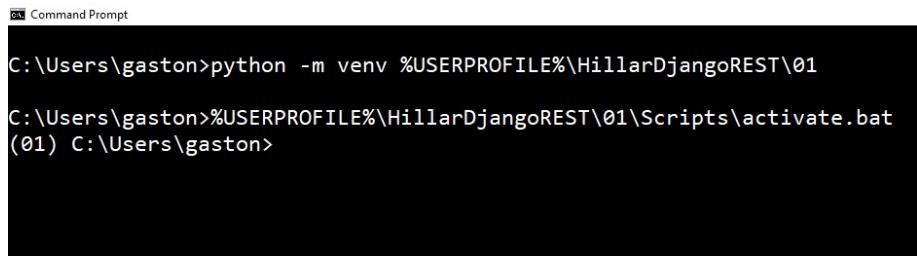
`Gastons-MacBook-Pro:~ gaston$` to `(01) Gastons-MacBook-Pro:~ gaston$` after the activation of the virtual environment.

In Windows, you can run either a batch file in the Command Prompt or a Windows PowerShell script to activate the virtual environment.

If you prefer Command Prompt, run the following command in the Windows command line to activate the virtual environment:

```
%USERPROFILE%\HillarDjangoREST\01\Scripts\activate.bat
```

The following screenshot shows the virtual environment activated in Windows 10 Command Prompt, after executing the previously shown commands:



A screenshot of a Windows Command Prompt window titled "Command Prompt". The window contains the following text:
C:\Users\gaston>python -m venv %USERPROFILE%\HillarDjangoREST\01
C:\Users\gaston>%USERPROFILE%\HillarDjangoREST\01\Scripts\activate.bat
(01) C:\Users\gaston>

As we can see from the previous screenshot, the prompt changed from `C:\Users\gaston` to `(01) C:\Users\gaston` after the activation of the virtual environment.

If you prefer Windows PowerShell, launch it and run the following commands to activate the virtual environment. Note that you must have scripts execution enabled in Windows PowerShell to be able to run the script:

```
cd $env:USERPROFILE  
HillarDjangoREST\01\Scripts\Activate.ps1
```

If you receive an error similar to the following lines, it means that you don't have scripts execution enabled:

```
C:\Users\gaston\HillarDjangoREST\01\Scripts\Activate.ps1 : File C:\Users\gas  
http://go.microsoft.com/fwlink/?LinkID=135170.  
At line:1 char:1  
+ C:\Users\gaston\HillarDjangoREST\01\Scripts\Activate.ps1  
+ -----  
+ CategoryInfo          : SecurityError: () [], PSSecurityException  
+ FullyQualifiedErrorId : UnauthorizedAccess
```

The Windows PowerShell default execution policy is `Restricted`. This policy allows the execution of individual commands but it doesn't run scripts. Thus, in case you want to work with Windows PowerShell, you will have to change the policy to allow the execution of scripts. It is very important to make sure that you understand the risks of the Windows PowerShell execution policies that allow you to run unsigned scripts. For more information about the different policies, check the following web page: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_execution_policies?view=powershell-6.

The following screenshot shows the virtual environment activated in a Windows 10 PowerShell, after executing the previously shown commands:



A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The window shows the command history for activating a virtual environment:

```
PS C:\WINDOWS\system32> python -m venv $env:UserProfile\HillarDjangoREST\01
PS C:\WINDOWS\system32> C:\Users\gaston\HillarDjangoREST\01\Scripts\Activate.ps1
(01) PS C:\WINDOWS\system32>
```

The prompt "(01)" indicates that the virtual environment is active.

Deactivating the virtual environment

It is extremely easy to deactivate a virtual environment generated by the previously explained process. The deactivation will remove all the changes made in the environment variables and will change the prompt back to its default message. Once you deactivate a virtual environment, you will go back to the default Python environment.

In macOS or Linux, just type `deactivate` and press *Enter*.

In a Windows Command Prompt, you have to run the `deactivate.bat` batch file included in the `scripts` folder. In our example, the full path for this file is `%USERPROFILE%\HillarDjangoREST\01\Scripts\deactivate.bat`.

In Windows PowerShell, you have to run the `Deactivate.ps1` script in the `scripts` folder. In our example, the full path for this file is `$env:userprofile\HillarDjangoREST\01\Scripts\Deactivate.ps1`. Remember that you must have scripts execution enabled in Windows PowerShell to be able to run the script.

The instructions in the next sections assume that the virtual environment we have created is activated.

Installing Django and Django REST frameworks in an isolated environment

We have created and activated a lightweight virtual environment. It is time to run many commands that will be the same for either Linux, macOS, or Windows.

First, run the following command to install the Django web framework:

```
pip install django==1.11.5
```

The last lines of the output will indicate that the `django` package has been successfully installed. The process will also install the `pytz` package that provides world time zone definitions. Take into account that you may also see a notice to upgrade `pip`. The next lines show a sample of the four last lines of the output generated by a successful `pip` installation:

```
Collecting django
Collecting pytz (from django)
  Installing collected packages: pytz, django
    Successfully installed django-1.11.5 pytz-2017.2
```

Now that we have installed the Django web framework, we can install Django REST framework. Django REST framework works on top of Django and provides us with a powerful and flexible toolkit to build RESTful Web Services. We just need to run the following command to install this package:

```
pip install djangorestframework==3.6.4
```

The last lines for the output will indicate that the `djangorestframework` package has been successfully installed, as shown here:

```
Collecting djangorestframework
  Installing collected packages: djangorestframework
    Successfully installed djangorestframework-3.6.4
```

After following the previous steps, we will have Django REST framework 3.6.4 and Django 1.11.5 installed in our virtual environment. We will install additional packages as we need them in the forthcoming chapters.

Creating an app with Django

Now, we will create our first app with Django and we will analyze the directory structure that Django creates. First, go to the root folder for the virtual environment: 01.

In Linux or macOS, enter the following command:

```
cd ~/HillarDjangoREST/01
```

If you prefer Command Prompt, run the following command in the Windows command line:

```
cd /d %USERPROFILE%\HillarDjangoREST\01
```

If you prefer Windows PowerShell, run the following command in Windows PowerShell:

```
cd /d $env:USERPROFILE\HillarDjangoREST\01
```

In Linux or macOS, run the following command to create a new Django project named `restful01`. The command won't produce any output:

```
python bin/django-admin.py startproject restful01
```

In Windows, in either Command Prompt or PowerShell, run the following command to create a new Django project named `restful01`. The command won't produce any output:

```
python Scripts\django-admin.py startproject restful01
```

The previous command creates a `restful01` folder with other subfolders and Python files. Now, go to the recently created `restful01` folder. Just execute the following command on any platform:

```
cd restful01
```

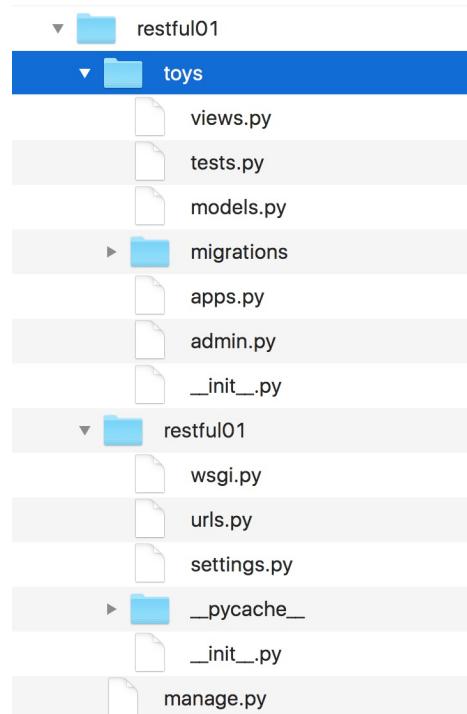
Then, run the following command to create a new Django app named `toys` within the `restful01` Django project. The command won't produce any output:

```
python manage.py startapp toys
```

The previous command creates a new `restful01/toys` subfolder, with the following files:

- `views.py`
- `tests.py`
- `models.py`
- `apps.py`
- `admin.py`
- `__init__.py`

In addition, the `restful01/toys` folder will have a `migrations` subfolder with an `__init__.py` Python script. The following diagram shows the folders and files in the directory tree, starting at the `restful01` folder with two subfolders - `toys` and `restful01`:



Understanding Django folders, files, and configurations

After we create our first Django project and then a Django app, there are many new folders and files. First, use your favorite editor or IDE to check the Python code in the `apps.py` file within the `restful01/toys` folder (`restful01\toys` in Windows). The following lines show the code for this file:

```
from django.apps import AppConfig

class ToysConfig(AppConfig):
    name = 'toys'
```

The code declares the `ToysConfig` class as a subclass of the `django.apps.AppConfig` class that represents a Django application and its configuration. The `ToysConfig` class just defines the `name` class attribute and sets its value to `'toys'`.

Now, we have to add `toys.apps.ToysConfig` as one of the installed apps in the `restful01/settings.py` file that configures settings for the `restful01` Django project. I built the previous string by concatenating many values as follows: `app name + .apps. + class name`, which is, `toys + .apps. + ToysConfig`. In addition, we have to add the `rest_framework` app to make it possible for us to use Django REST framework.

The `restful01/settings.py` file is a Python module with module-level variables that define the configuration of Django for the `restful01` project. We will make some changes to this Django settings file. Open the `restful01/settings.py` file and locate the highlighted lines that specify the strings list that declares the installed apps. The following code shows the

first lines for the `settings.py` file. Note that the file has more code:

```
"""
Django settings for restful01 project.

Generated by 'django-admin startproject' using Django 1.11.5.

For more information on this file, see
https://docs.djangoproject.com/en/1.11/topics/settings/

For the full list of settings and their values, see
https://docs.djangoproject.com/en/1.11/ref/settings/
"""

import os

# Build paths inside the project like this: os.path.join(BASE_DIR, ...)
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))

# Quick-start development settings - unsuitable for production
# See https://docs.djangoproject.com/en/1.11/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = '+uyg(tmneo+fpg+fcwmm&x(2x0gml8)=cs@$nijab%)y$a*xe'

# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True

ALLOWED_HOSTS = []

# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Add the following two strings to the `INSTALLED_APPS` strings list and save the changes to the `restful01/settings.py` file:

- 'rest_framework'
- 'toys.apps.ToysConfig'

The following lines show the new code that declares the `INSTALLED_APPS` string list with the added lines highlighted and with comments to understand what each added string means. The code file for the sample is included in the `hillar_django_restful_01` folder:

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    # Django REST framework  
    'rest_framework',  
    # Toys application  
    'toys.apps.ToysConfig',  
]
```

This way, we have added Django REST framework and the `toys` application to our initial Django project named `restful01`.

Installing tools

Now, we will leave Django for a while and we will install many tools that we will use to interact with the RESTful Web Services that we will develop throughout this book.

We will use the following different kinds of tools to compose and send HTTP requests and visualize the responses throughout our book:

- Command-line tools
- GUI tools
- Python code
- Web browser
- JavaScript code

You can use any other application that allows you to compose and send HTTP requests. There are many apps that run on tablets and smartphones that allow you to accomplish this task. However, we will focus our attention on the most useful tools when building RESTful Web Services with Django.

Installing Curl

We will start installing command-line tools. One of the key advantages of command-line tools is that you can easily run again the HTTP requests again after we have built them for the first time, and we don't need to use the mouse or tap the screen to run requests. We can also easily build a script with batch requests and run them.

As happens with any command-line tool, it can take more time to perform the first requests compared with GUI tools, but it becomes easier once we have performed many requests and we can easily reuse the commands we have written in the past to compose new requests.

Curl, also known as **cURL**, is a very popular open source command-line tool and library that allows us to easily transfer data. We can use the `curl` command-line tool to easily compose and send HTTP requests and check their responses.

In Linux or macOS, you can open a Terminal and start using `curl` from the command line.

In Windows, you have two options. You can work with `curl` in Command Prompt or you can decide to install curl as part of the Cygwin package installation option and execute it from the Cygwin terminal. You can read more about the Cygwin terminal and its installation procedure at: <http://cygwin.com/install.html>. Windows Powershell includes a `curl` alias that calls the `Invoke-WebRequest` command, and therefore, if you want to work with Windows Powershell with curl, it is necessary to remove the `curl` alias.

If you want to use the `curl` command within Command Prompt, you just need to download and unzip the latest version of the `curl` download page: <https://curl.haxx.se/download.html>. Make sure you download the version that includes SSL and SSH.

The following screenshot shows the available downloads for Windows. The Win64 - Generic section includes the versions that we can run in Command Prompt or Windows Powershell.

The `Win64_x86_64.7zip` file provides the binary version for `curl` version 7.55.1 with SSL and SSH support:

This screenshot shows the curl download page on curl.haxx.se. It displays three sections of packages:

- Win32 zip**:
 - Win32 zip 7.55.1 binary SSL SSH Dirk Paelinck
 - Win32 2000/XP MSI 7.46.0 binary SSL SSH Edward LoPinto
 - Win32 2000/XP zip 7.46.0 binary SSL SSH Edward LoPinto
- Win32 - cygwin**:
 - cygwin 7.55.1 binary SSL Cygwin 300 KB
 - cygwin 7.55.1 libcurl SSL Cygwin 206 KB
- Win64 - Generic**:
 - Win64 ia64 CAB 7.55.1 binary SSL Stefan Kanthak
 - Win64 x86_64 7zip 7.53.1 binary SSL Darren Owen
 - Win64 x86_64 zip 7.53.1 binary SSL SSH Marc Hörsken 849 KB
 - Win64 x86_64 CAB 7.55.1 binary SSL Stefan Kanthak
 - Win64 x86_64 7zip 7.55.1 binary SSL SSH Viktor Szakáts 1.86 MB
 - Win64 2000/XP x86_64 zip 7.46.0 binary SSL SSH Edward LoPinto
 - Win64 2000/XP x86_64 MSI 7.46.0 binary SSL SSH Edward LoPinto
- Win64 - cygwin**:
 - cygwin 7.55.1 binary SSL Cygwin 301 KB
 - cygwin 7.55.1 libcurl SSL Cygwin 196 KB

A note at the bottom states: "This colour means the packaged version is the latest stable version available (7.55.1)!"

After you unzip the `.7zip` or `.zip` file you have downloaded, you can include the folder in which `curl.exe` is included in your path. For example, if you unzip the `Win64_x86_64.7zip` file, you will find `curl.exe` in the `bin` folder. The following screenshot shows the results of executing `curl --version` on Command Prompt in Windows 10. The `--version` option makes curl display its version and all the libraries, protocols, and features it supports:

```
D:\Curl\curl-7.55.1-win64-mingw\bin>curl --version
curl 7.55.1 (x86_64-pc-win32) libcurl/7.55.1 OpenSSL/1.1.0f zlib/1.2.11 WinIDN libssh2/1.8.0 nghttp2/1.25.0
Release-Date: 2017-08-14
Protocols: dict file ftp ftps gopher http https imap imaps ldap ldaps pop3 pop3s rtsp scp sftp smtp smtpe telnet tftp
Features: AsynchDNS IDN IPv6 Largefile SSPI Kerberos SPNEGO NTLM SSL libz TLS-SRP HTTP2 HTTPS-proxy

D:\Curl\curl-7.55.1-win64-mingw\bin>
```

Installing HTTPie

Now, we will install HTTPie, a command-line HTTP client written in Python that makes it easy to send HTTP requests and uses a syntax that is easier than curl. By default, HTTPie displays colorized output and uses multiple lines to display the response details. In some cases, HTTPie makes it easier to understand the responses than the curl utility. However, one of the great disadvantages of HTTPie as a command-line utility is that it takes more time to load than curl, and therefore, if you want to code scripts with too many commands, you have to evaluate whether it makes sense to use HTTPie.

We just need to make sure we run the following command in the virtual environment we have just created and activated. This way, we will install HTTPie only for our virtual environment.

Run the following command in the terminal, Command Prompt, or Windows PowerShell to install the `httpie` package:

```
pip install --upgrade httpie
```

The last lines of the output will indicate that the `httpie` package has been successfully installed:

```
Collecting httpie
Collecting colorama>=0.2.4 (from httpie)
Collecting requests>=2.11.0 (from httpie)
Collecting Pygments>=2.1.3 (from httpie)
Collecting idna<2.7,>=2.5 (from requests>=2.11.0->httpie)
Collecting urllib3<1.23,>=1.21.1 (from requests>=2.11.0->httpie)
Collecting chardet<3.1.0,>=3.0.2 (from requests>=2.11.0->httpie)
Collecting certifi>=2017.4.17 (from requests>=2.11.0->httpie)
Installing collected packages: colorama, idna, urllib3, chardet, certifi, re
Successfully installed Pygments-2.2.0 certifi-2017.7.27.1 chardet-3.0.4 colo
```



If you don't remember how to activate the virtual environment that we created for this example, read the Activating the virtual environment section in this chapter.

Now, we will be able to use the `http` command to easily compose and send HTTP requests to our future RESTful Web Services build with Django.

The following screenshot shows the results of executing `http` on Command Prompt in Windows 10. HTTPie displays the valid options and indicates that a URL is required:

```
PS C:\Users\gaston\HillarDjangoREST\01\Scripts>http
(01) C:\Users\gaston\HillarDjangoREST\01\Scripts>http
usage: http [--json] [--form] [--pretty {all,colors,format,none}]
            [--style STYLE] [--print WHAT] [--headers] [--body] [--verbose]
            [--all] [--history-print WHAT] [--stream] [--output FILE]
            [--download] [--continue]
            [--session SESSION_NAME_OR_PATH | --session-read-only SESSION_NAME_OR_PATH]
            [--auth USER[:PASS]] [--auth-type {basic,digest}]
            [--proxy PROTOCOL:PROXY_URL] [--follow]
            [--max-redirects MAX_REDIRECTS] [--timeout SECONDS]
            [--check-status] [--verify VERIFY]
            [--ssl {ssl2.3,ssl3,tls1,tls1.1,tls1.2}] [--cert CERT]
            [--cert-key CERT_KEY] [--ignore-stdin] [--help] [--version]
            [--traceback] [--default-scheme DEFAULT_SCHEME] [--debug]
            [METHOD] URL [REQUEST_ITEM [REQUEST_ITEM ...]]
http: error: the following arguments are required: URL

(01) C:\Users\gaston\HillarDjangoREST\01\Scripts>
```

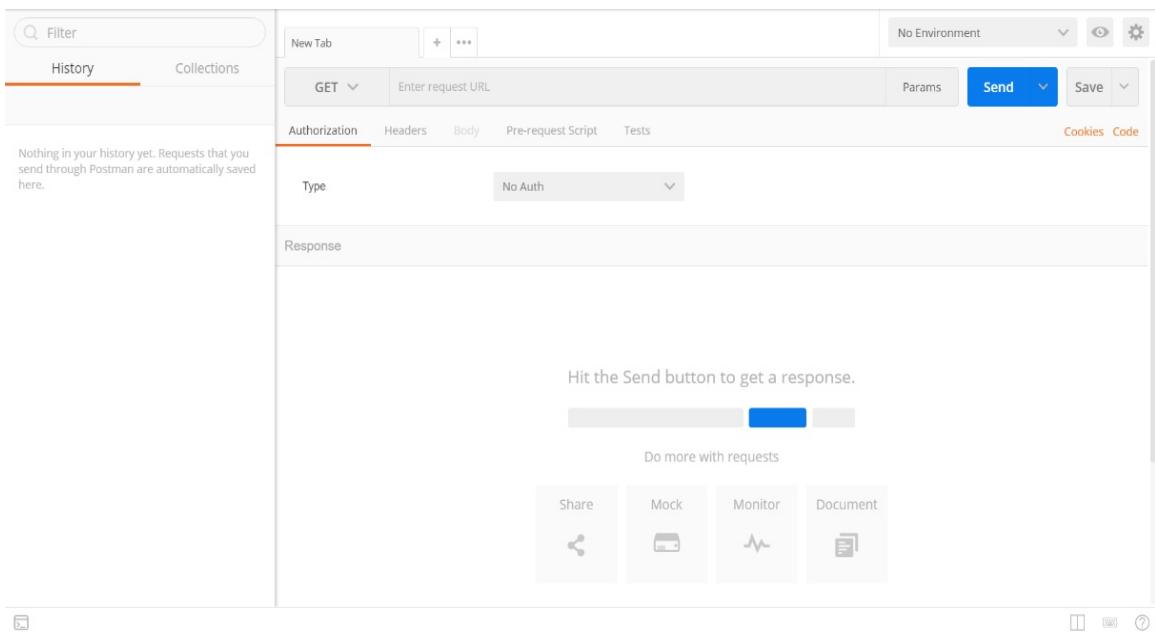
Installing the Postman REST client

So far, we have installed two terminal-based or command-line tools to compose and send HTTP requests to our Django development server: CURL and HTTPie. Now, we will start installing **Graphical User Interface (GUI)** tools.

Postman is a very popular API testing suite GUI tool that allows us to easily compose and send HTTP requests, among other features. Postman is available as a standalone app in Linux, macOS, and Windows. You can download the versions of the *Postman* app from the following URL: <https://www.getpostman.com>.

You can download and install Postman for free to compose and send HTTP requests to the RESTful Web Services we will build throughout this book. You just need to sign up to Postman. We won't be using any of the paid features provided by either Postman Pro or Postman Enterprise in our examples. All the instructions work with Postman 5.2.1 or greater.

The following screenshot shows the HTTP GET request builder in Postman:

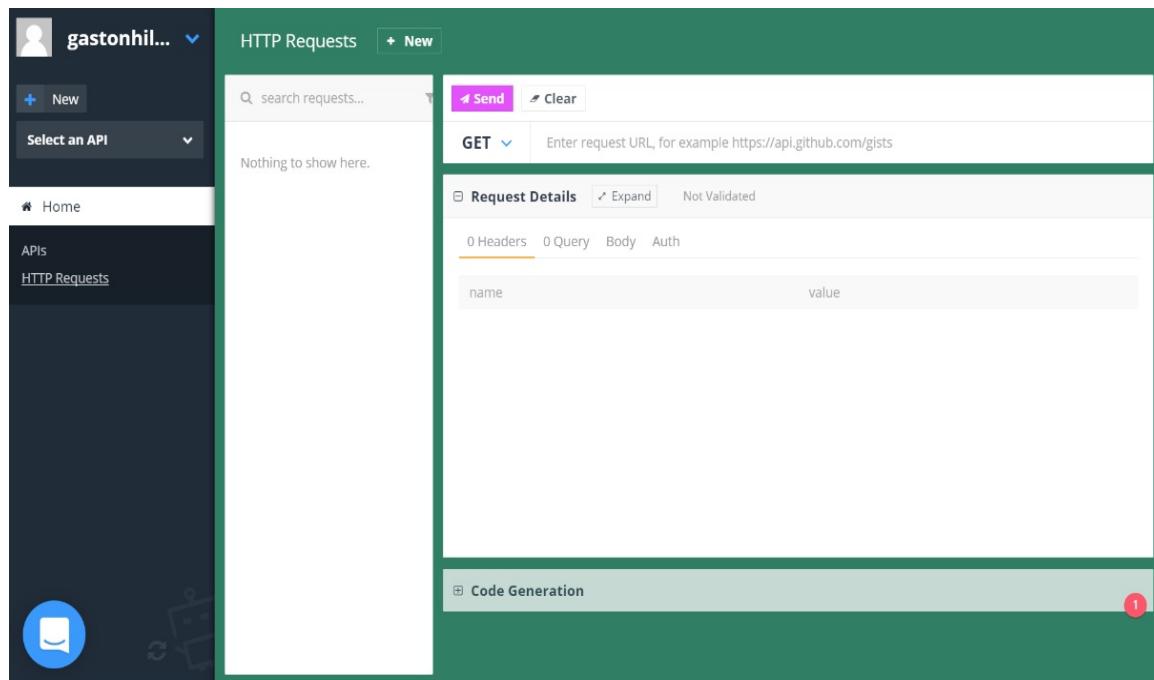


Installing Stoplight

Stoplight is a very useful GUI tool that focuses on helping architects and developers to model complex APIs. If we need to consume our RESTful Web Service in many different programming languages, we will find Stoplight extremely helpful. Stoplight provides an HTTP request maker that allows us to compose and send requests and generate the necessary code to make them in different programming languages, such as JavaScript, Swift, C#, PHP, Node, and Go, among others.

Stoplight provides a web version and is also available as a standalone app in Linux, macOS, and Windows. You can download the versions of Stoplight from the following URL: <http://stoplight.io/>.

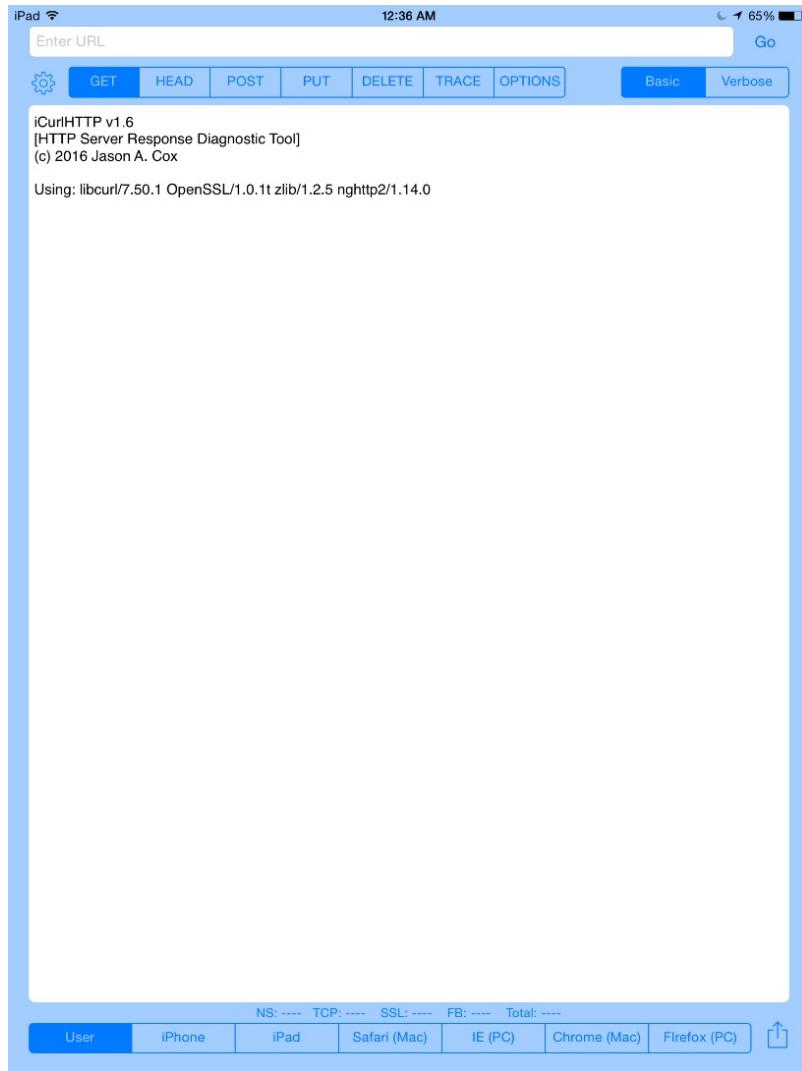
The following screenshot shows the HTTP GET request builder in Stoplight with the code generation at the bottom:



Installing iCurlHTTP

We can also use apps that can compose and send HTTP requests from mobile devices to work with our RESTful Web Services. For example, we can work with the iCurlHTTP app on iOS devices such as iPad and iPhone: <https://itunes.apple.com/us/app/icurlhttp/id611943891>. On Android devices, we can work with the *HTTP Request* app: <https://play.google.com/store/apps/details?id=air.http.request&hl=en>.

The following screenshot shows the UI for the iCurlHTTP app running on an iPad Pro:



At the time of writing, the mobile apps that allow you to compose and send HTTP requests do not provide all the features you can find in Postman or command-line utilities.

Test your knowledge

Let's see whether you can answer the following questions correctly:

1. After we activate a virtual environment, all the packages we install with `pip` are available:
 1. For all the virtual environments available in the computer or device that is running Python
 2. Only for the activated virtual environment
 3. For all the virtual environments created by the current user
2. HTTPie is a:
 1. Command-line HTTP server written in Python that makes it easy to create a RESTful Web Server
 2. Command-line utility that allows us to run queries against an SQLite database
 3. Command-line HTTP client written in Python that makes it easy to compose and send HTTP requests
3. Which of the following commands creates a new app named `books` in Django?
 1. `django startapp books`

2. `python django.py startapp books`
 3. `python manage.py startapp books`
4. In Django, a subclass of which of the following classes represents a Django application and its configuration?
 1. `django.apps.AppConfig`
 2. `django.application.configuration`
 3. `django.config.App`
5. Which of the following strings must be added to the `INSTALLED_APPS` string list in the `settings.py` file to enable Django REST framework?
 1. `'rest_framework'`
 2. `'django_rest_framework'`
 3. `'Django_REST_framework'`

The rights answers are included in the [Appendix](#), *Solutions*.

Summary

In this chapter, we learned the advantages of working with lightweight virtual environments in Python and we set up a virtual environment with Django and Django REST framework. We created an app with Django, we took a first look at the Django folders, files, and configurations, and we made the necessary changes to activate Django REST framework.

Then, we introduced and installed command-line and GUI tools that we will use to interact with the RESTful Web Services that we will design, code, test, and run in the forthcoming chapters.

Now that we have our environment ready to start working with Django REST framework, we will define the requirements for our first RESTful Web Service and we will work with models, migrations, serialization, and deserialization, which are the topics that we are going to discuss in the next chapter.

Working with Models, Migrations, Serialization, and Deserialization

In this chapter, we will define the requirements for our first RESTful Web Service. We will start working with Django, Django REST framework, Python, configurations, models, migrations, serialization, and deserialization. We will create a RESTful Web Service that performs **CRUD** (short for **C**reate, **R**ead, **U**pdate and **D**elete) operations on a simple SQLite database. We will be:

- Defining the requirements for our first RESTful Web Service
- Creating our first model
- Running our initial migration
- Understanding migrations
- Analyzing the database
- Understanding Django tables
- Controlling, serialization, and deserialization
- Working with the Django shell and diving deeply into serialization and deserialization

Defining the requirements for our first RESTful Web Service

Imagine a team of developers working on a mobile app for iOS and Android and requires a RESTful Web Service to perform CRUD operations with toys. We definitely don't want to use a mock web service and we don't want to spend time choosing and configuring an **ORM** (short for **Object-Relational Mapping**). We want to quickly build a RESTful Web Service and have it ready as soon as possible to start interacting with it in the mobile app.

We really want the toys to persist in a database but we don't need it to be production-ready. Therefore, we can use the simplest possible relational database, as long as we don't have to spend time performing complex installations or configurations.

Django REST framework, also known as **DRF**, will allow us to easily accomplish this task and start making HTTP requests to the first version of our RESTful Web Service. In this case, we will work with a very simple SQLite database, the default database for a new Django REST framework project.

First, we must specify the requirements for our main resource: a toy. We need the following attributes or fields for a toy entity:

- An integer identifier
- A name
- An optional description
- A toy category description, such as action figures, dolls, or

playsets

- A release date
- A bool value indicating whether the toy has been on the online store's homepage at least once

In addition, we want to have a timestamp with the date and time of the toy's addition to the database table, which will be generated to persist toys.

In a RESTful Web Service, each resource has its own unique URL. In our web service, each toy will have its own unique URL.

The following table shows the HTTP verbs, the scope, and the semantics of the methods that our first version of the web service must support. Each method is composed of an HTTP verb and a scope. All the methods have a well-defined meaning for toys and collections:

HTTP verb	Scope	Semantics
GET	Toy	Retrieve a single toy
GET	Collection of toys	Retrieve all the stored toys in the collection, sorted by their name in ascending order
POST	Collection of toys	Create a new toy in the collection
PUT	Toy	Update an existing toy
DELETE	Toy	Delete an existing toy

In the previous table, the GET HTTP verb appears twice but with two different scopes: toys and collection of toys. The first row shows a GET HTTP verb applied to a toy, that is, to a single resource. The second row shows a GET HTTP verb applied to a collection of toys, that is, to a collection of resources.

We want our web service to be able to differentiate collections from a single resource of the collection in the URLs. When we refer to a collection, we will use a slash (/) as the last character for the URL, as in `http://localhost:8000/toys/`. When we refer to a single resource of the collection we won't use a slash (/) as the last character for the URL, as in `http://localhost:8000/toys/5`.

Let's consider that `http://localhost:8000/toys/` is the URL for the collection of toys. If we add a number to the previous URL, we identify a specific toy with an ID or primary key equal to the specified numeric value. For example, `http://localhost:8000/toys/42` identifies the toy with an ID equal to 42.

We have to compose and send an HTTP request with the `POST` HTTP verb and `http://localhost:8000/toys/` request URL to create a new toy and add it to the toys collection. In this example, our RESTful Web Service will work with **JSON** (short for **JavaScript Object Notation**), and therefore we have to provide the JSON key-value pairs with the field names and the values to create the new toy. As a result of the request, the server will validate the provided values for the fields, make sure that it is a valid toy, and persist it in the database. The server will insert a new row with the new toy in the appropriate table and it will return a `201 Created` status code and a JSON body with the recently added toy serialized to JSON, including the assigned ID that was automatically generated by the database and assigned to the toy object:

```
POST http://localhost:8000/toys/
```

We have to compose and send an HTTP request with the `GET` HTTP verb and `http://localhost:8000/toys/{id}` request URL to retrieve the toy whose ID matches the specified numeric value in `{id}`. For example, if we use the request URL `http://localhost:8000/toys/25`, the server will retrieve the toy whose ID matches `25`. As a result of the request, the server will retrieve a toy with the specified ID from the database and create the appropriate toy object in Python. If a toy is found, the server will serialize the toy object into JSON, return a `200 OK` status code, and return a JSON body with the serialized toy object. If no toy matches the specified ID, the server will return only a `404 Not Found` status:

```
GET http://localhost:8000/toys/{id}
```

We have to compose and send an HTTP request with the `PUT` HTTP verb and request URL `http://localhost:8000/toys/{id}` to retrieve the toy whose ID matches the value in `{id}` and replace it with a toy created with the provided data. In addition, we have to provide the JSON key-value pairs with the field names and the values to create the new toy that will replace the existing one. As a result of the request, the server will validate the provided values for the fields, make sure that it is a valid toy, and replace the one that matches the specified ID with the new one in the database. The ID for the toy will be the same after the update operation. The server will update the existing row in the appropriate table and it will return a `200 OK` status code and a JSON body with the recently updated toy serialized to JSON. If we don't provide all the necessary data for the new toy, the server will return a `400 Bad Request` status code. If the server doesn't find a toy with the specified ID, the server will only return a `404 Not Found` status:

```
PUT http://localhost:8000/toys/{id}
```

We have to compose and send an HTTP request with the `DELETE` HTTP verb and request URL `http://localhost:8000/toys/{id}` to remove the toy whose ID matches the specified numeric value in `{id}`. For example, if we use the request URL `http://localhost:8000/toys/34`, the server will delete the toy whose ID matches `34`. As a result of the request, the server will retrieve a

toy with the specified ID from the database and create the appropriate toy object in Python. If a toy is found, the server will request the ORM delete the toy row associated with this toy object and the server will return a `204 No Content` status code. If no toy matches the specified ID, the server will return only a `404 Not Found` status:

```
DELETE http://localhost:8000/toys/{id}
```

1

▶ 1

Creating our first model

Now, we will create a simple `Toy` model in Django, which we will use to represent and persist toys. Open the `toys/models.py` file. The following lines show the initial code for this file with just one `import` statement and a comment that indicates we should create the models:

```
from django.db import models

# Create your models here.
```

The following lines show the new code that creates a `Toy` class, specifically, a `Toy` model in the `toys/models.py` file. The code file for the sample is included in the `hillar_django_restful_02_01` folder in the `restful01/toys/models.py` file:

```
from django.db import models

class Toy(models.Model):
    created = models.DateTimeField(auto_now_add=True)
    name = models.CharField(max_length=150, blank=False, default='')
    description = models.CharField(max_length=250, blank=True, default='')
    toy_category = models.CharField(max_length=200, blank=False, default='')
    release_date = models.DateTimeField()
    was_included_in_home = models.BooleanField(default=False)

    class Meta:
        ordering = ('name',)
```

The `Toy` class is a subclass of the `django.db.models.Model` class and defines the following attributes: `created`, `name`, `description`, `toy_category`, `release_date`, and `was_included_in_home`. Each of these attributes represents a database column or field.

Django automatically adds an auto-increment integer primary key column named `id` when it creates the database table related to the model. It is very important to notice that the model maps the underlying `id` column in an attribute named `pk` for the model.

We specified the field types, maximum lengths, and defaults for many attributes. The class declares a `Meta` inner class that declares an `ordering` attribute and sets its value to a tuple of `string` whose first value is the '`name`' string. This way, the inner class indicates to Django that, by default, we want the results ordered by the `name` attribute in ascending order.

Running our initial migration

Now, it is necessary to create the initial migration for the new `Toy` model we recently coded. We will also synchronize the SQLite database for the first time. By default, Django uses the popular self-contained and embedded SQLite database, and therefore we don't need to make changes in the initial ORM configuration. In this example, we will be working with this default configuration. Of course, we will upgrade to another database after we have a sample web service built with Django. We will only use SQLite for this example.

We just need to run the following Python script in the virtual environment that we activated in the previous chapter. Make sure you are in the `restful01` folder within the main folder for the virtual environment when you run the following command:

```
python manage.py makemigrations toys
```

The following lines show the output generated after running the previous command:

```
Migrations for 'toys':  
  toys/migrations/0001_initial.py:  
    - Create model Toy
```

The output indicates that the `restful01/toys/migrations/0001_initial.py` file includes the code to create the `Toy` model. The following lines show the code for this file that was automatically generated by Django. The code file for the sample is included in the `hillar_django_restful_02_01` folder in the `restful01/toys/migrations/0001_initial.py` file:

```
# -*- coding: utf-8 -*-
# Generated by Django 1.11.5 on 2017-10-08 05:19
from __future__ import unicode_literals

from django.db import migrations, models


class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='Toy',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False)),
                ('created', models.DateTimeField(auto_now_add=True)),
                ('name', models.CharField(default='', max_length=150)),
                ('description', models.CharField(blank=True, default='', max_length=300)),
                ('toy_category', models.CharField(default='', max_length=200)),
                ('release_date', models.DateTimeField()),
                ('was_included_in_home', models.BooleanField(default=False)),
            ],
            options={
                'ordering': ('name',),
            },
        ),
    ]
]
```



Understanding migrations

The automatically generated code defines a subclass of the `django.db.migrations.Migration` class named `Migration`, which defines an operation that creates the `Toy` model's table and includes it in the `operations` attribute. The call to the `migrations.createModel` method specifies the model's name, the fields, and the options to instruct the ORM to create a table that will allow the underlying database to persist the model.

The `fields` argument is a list of tuples that includes information about the field name, the field type, and additional attributes based on the data we provided in our model, that is, in the `Toy` class.

Now, run the following Python script to apply all the generated migrations. Make sure you are in the `restful01` folder within the main folder for the virtual environment when you run the following command:

```
python manage.py migrate
```

The following lines show the output generated after running the previous command:

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions, toys
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
```

```
Applying auth.0006_require_contenttypes_0002... OK
Applying auth.0007_alter_validators_add_error_messages... OK
Applying auth.0008_alter_user_username_max_length... OK
Applying sessions.0001_initial... OK
Applying toys.0001_initial... OK
```

After we run the previous command, we will notice that the root folder for our `restful01` project now has a `db.sqlite3` file that contains the SQLite database. We can use the SQLite command line or any other application that allows us to easily check the contents of the SQLite database to check the tables that Django generated.

The first migration will generate many tables required by Django and its installed apps before running the code that creates the table for the `Toys` model. These tables provide support for user authentication, permissions, groups, logs, and migration management. We will work with the models related to these tables after we add more features and security to our web services.

After the migration process creates all these Django tables in the underlying database, the first migration runs the Python code that creates the table required to persist our model. Thus, the last line of the running migrations section displays `Applying toys.0001_initial`.

Analyzing the database

In most modern Linux distributions and macOS, SQLite is already installed, and therefore you can run the `sqlite3` command-line utility.

In Windows, if you want to work with the `sqlite3.exe` command-line utility, you have to download the bundle of command-line tools for managing SQLite database files from the downloads section of the SQLite webpage at <http://www.sqlite.org/download.html>. For example, the ZIP file that includes the command-line tools for version 3.20.1 is `sqlite-tools-win32-x86-3200100.zip`. The name for the file changes with the SQLite version. You just need to make sure that you download the bundle of command-line tools and not the ZIP file that provides the SQLite DLLs. After you unzip the file, you can include the folder that includes the command-line tools in the PATH environment variable, or you can access the `sqlite3.exe` command-line utility by specifying the full path to it.

Run the following command to list the generated tables. The first argument, `db.sqlite3`, specifies the file that contains that SQLite database and the second argument indicates the command that we want the `sqlite3` command-line utility to run against the specified database:

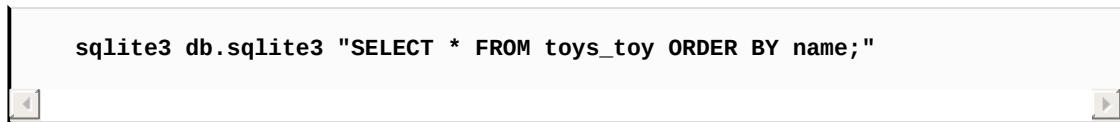
```
sqlite3 db.sqlite3 ".tables"
```

The following lines show the output for the previous command with the list of tables that Django generated in the SQLite database:

<code>auth_group</code>	<code>django_admin_log</code>
<code>auth_group_permissions</code>	<code>django_content_type</code>
<code>auth_permission</code>	<code>django_migrations</code>
<code>auth_user</code>	<code>django_session</code>
<code>auth_user_groups</code>	<code>toys_toy</code>
<code>auth_user_user_permissions</code>	



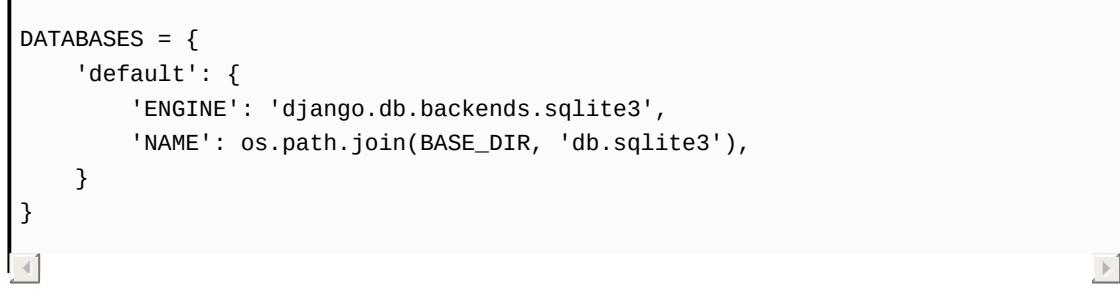
The following command will allow you to check the contents of the `toys_toy` table after we compose and send HTTP requests to the RESTful Web Service and the web service makes CRUD operations to the `toys_toy` table:



Instead of working with the SQLite command-line utility, you can use a GUI tool to check the contents of the SQLite database. DB Browser for SQLite is a useful, free, multiplatform GUI tool that allows us to easily check the database contents of an SQLite database in Linux, macOS, and Windows. You can read more information about this tool and download its different versions from <http://sqlitebrowser.org>. Once you have installed the tool, you just need to open the `db.sqlite3` file and you can check the database structure and browse the data for the different tables. After we start working with the first version of our web service, you need to check the contents of the `toys_toy` table with this tool.

You can also use the database tools included with your favorite IDE to check the contents of the SQLite database.

The SQLite database engine and the database file name are specified in the `restful01/settings.py` Python file. The following lines show the declaration of the `DATABASES` dictionary, which contains the settings for all the databases that Django uses. The nested dictionary maps the database named `default` with the `django.db.backends.sqlite3` database engine and the `db.sqlite3` database file located in the `BASE_DIR` folder (`restful01`):





After we execute the migrations, the SQLite database will have the following tables. Django uses prefixes to identify the modules and applications that each table belongs to. The tables that start with the `auth_` prefix belong to the Django authentication module. The table that starts with the `toys_` prefix belongs to our `toys` application. If we add more models to our `toys` application, Django will create new tables with the `toys_` prefix:

- `auth_group`: Stores authentication groups
- `auth_group_permissions`: Stores permissions for authentication groups
- `auth_permission`: Stores permissions for authentication
- `auth_user`: Stores authentication users
- `auth_user_groups`: Stores authentication user groups
- `auth_user_groups_permissions`: Stores permissions for authentication user groups
- `django_admin_log`: Stores the Django administrator log
- `django_content_type`: Stores Django content types
- `django_migrations`: Stores the scripts generated by Django migrations and the date and time at which they were applied
- `django_session`: Stores Django sessions
- `toys_toy`: Persists the `Toys` model
- `sqlite_sequence`: Stores sequences for SQLite primary keys with autoincrement fields

Understanding the table generated by Django

The `toys_toy` table persists in the database the `Toy` class we recently created, specifically, the `Toy` model. Django's integrated ORM generated the `toys_toy` table based on our `Toy` model.

Run the following command to retrieve the SQL used to create the `toys_toy` table:

```
sqlite3 db.sqlite3 ".schema toys_toy"
```

The following lines show the output for the previous command together with the SQL that the migrations process executed, to create the `toys_toy` table that persists the `Toy` model. The next lines are formatted to make it easier to understand the SQL code. Notice that the output from the command is formatted in a different way:

```
CREATE TABLE IF NOT EXISTS "toys_toy"
(
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "created" datetime NOT NULL,
    "name" varchar(150) NOT NULL,
    "description" varchar(250) NOT NULL,
    "toy_category" varchar(200) NOT NULL,
    "release_date" datetime NOT NULL,
    "was_included_in_home" bool NOT NULL
);
```

The `toys_toy` table has the following columns (also known as fields) with their SQLite types, all of them not nullable:

- `id`: The integer primary key, an autoincrement row
- `created`: `DateTime`
- `name`: `varchar(150)`
- `description`: `varchar(250)`
- `toy_category`: `varchar(200)`
- `release_date`: `DateTime`
- `was_included_in_home`: `bool`

Controlling, serialization, and deserialization

Our RESTful Web Service has to be able to serialize and deserialize the `Toy` instances into JSON representations. In Django REST framework, we just need to create a serializer class for the `Toy` instances to manage serialization to JSON and deserialization from JSON. Now, we will dive deep into the serialization and deserialization process in Django REST framework. It is very important to understand how it works because it is one of the most important components for all the RESTful Web Services we will build.

Django REST framework uses a two-phase process for serialization. The serializers are mediators between the model instances and Python primitives. Parser and renderers handle as mediators between Python primitives and HTTP requests and responses. We will configure our mediator between the `Toy` model instances and Python primitives by creating a subclass of the `rest_framework.serializers.Serializer` class to declare the fields and the necessary methods to manage serialization and deserialization.

We will repeat some of the information about the fields that we have included in the `Toy` model so that we understand all the things that we can configure in a subclass of the `Serializer` class. However, we will work with shortcuts, which will reduce boilerplate code later in the following examples. We will write less code in the following examples by using the `ModelSerializer` class.

Now, go to the `restful01/toys` folder and create a new Python code file named `serializers.py`. The following lines show the code that declares the new `ToySerializer` class. The code file for the sample is included in the `hillar_django_restful_02_01` folder in the `restful01/toys/serializers.py` file:

```
from rest_framework import serializers
from toys.models import Toy

class ToySerializer(serializers.Serializer):
    pk = serializers.IntegerField(read_only=True)
    name = serializers.CharField(max_length=150)
    description = serializers.CharField(max_length=250)
    release_date = serializers.DateTimeField()
    toy_category = serializers.CharField(max_length=200)
    was_included_in_home = serializers.BooleanField(required=False)

    def create(self, validated_data):
        return Toy.objects.create(**validated_data)

    def update(self, instance, validated_data):
        instance.name = validated_data.get('name', instance.name)
        instance.description = validated_data.get('description', instance.description)
        instance.release_date = validated_data.get('release_date', instance.release_date)
        instance.toy_category = validated_data.get('toy_category', instance.toy_category)
        instance.was_included_in_home = validated_data.get('was_included_in_home', instance.was_included_in_home)
        instance.save()
        return instance
```

The `ToySerializer` class declares the attributes that represent the fields that we want to be serialized. Notice that we have omitted the `created` attribute that was present in the `Toy` model. When there is a call to the `save` method that `ToySerializer` inherits from the `serializers.Serializer` superclass, the overridden `create` and `update` methods define how to create a new instance or update an existing instance. In fact, these methods must be implemented in our class because they only raise a `NotImplementedError` exception in their base declaration in the `serializers.Serializer` superclass.

The `create` method receives the validated data in the `validated_data` argument. The code creates and returns a new `Toy` instance based on the received validated data.

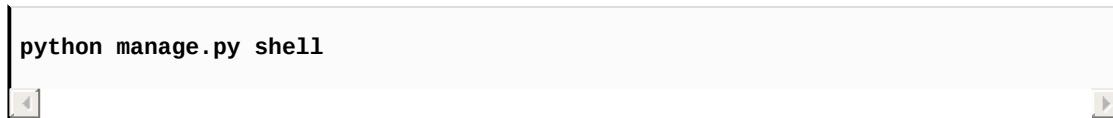
The `update` method receives an existing `Toy` instance that is being updated and the new validated data in the `instance` and `validated_data` arguments. The code updates the values for the attributes of the instance with the updated attribute values retrieved from the validated data. Finally, the code calls

the `save` method for the updated `toy` instance and returns the updated and saved instance.

Working with the Django shell and diving deeply into serialization and deserialization

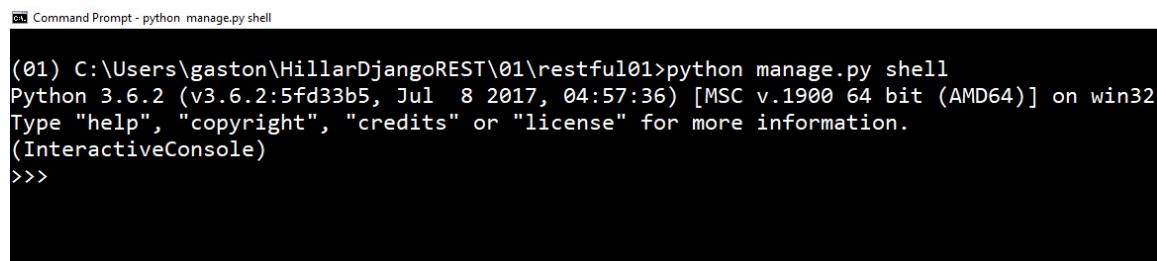
We can launch our default Python interactive shell in our virtual environment and make all the Django project modules available before it starts. This way, we can check that the serializer works as expected. We will do this to understand how serialization works in Django.

Run the following command to launch the interactive shell. Make sure you are within the `restful01` folder in the terminal, Command Prompt, or Windows Powershell:



```
python manage.py shell
```

You will notice a line that says `(InteractiveConsole)` is displayed after the usual lines that introduce your default Python interactive shell. The following screenshot shows the Django shell launched in a Windows command prompt:



```
C:\ Command Prompt - python manage.py shell
(01) C:\Users\gaston\HillarDjangoREST\01\restful01>python manage.py shell
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>
```

Enter the following code in the Python interactive shell to import all the things we will need to test the `Toy` model and its serializer. The code file for

the sample is included in the `hillar_django_restful_02_01` folder, in the `restful01/toy_serializers_test_01.py` file:

```
from datetime import datetime
from django.utils import timezone
from django.utils.six import BytesIO
from rest_framework.renderers import JSONRenderer
from rest_framework.parsers import JSONParser
from toys.models import Toy
from toys.serializers import ToySerializer
```

Enter the following code to create two instances of the `Toy` model and save them. The code file for the sample is included in the `hillar_django_restful_02_01` folder, in the `restful01/toy_serializers_test_01.py` file:

```
toy_release_date = timezone.make_aware(datetime.now(), timezone.get_current_time
toy1 = Toy(name='Snoopy talking action figure', description='Snoopy speaks five
toy1.save()
toy2 = Toy(name='Hawaiian Barbie', description='Barbie loves Hawaii', release_da
toy2.save()
```

After we execute the preceding code, we can check the SQLite database with the previously introduced command-line or GUI tools to check the contents of the `toys_toy` table. We will notice the table has two rows and columns with the values we have provided to the different attributes of the `Toy` instances. The following screenshot shows the results of browsing the data of the `toys_toy` table with the DB Browser for SQLite GUI utility. We can see that two rows were inserted.

Enter the following code in the interactive shell to check the values for the primary keys or identifiers for the saved `Toy` instances, and the value of their `name` and `was_included_in_home_attribute` attributes. The code also checks the value of the `created` attribute, which includes the date and time at which Django saved each instance to the database. The code file for the sample is included in the `hillar_django_restful_02_01` folder, in the `restful01/toy_serializers_test_01.py` file:

```

print(toy1.pk)
print(toy1.name)
print(toy1.created)
print(toy1.was_included_in_home)
print(toy2.pk)
print(toy2.name)
print(toy2.created)
print(toy2.was_included_in_home)

```

The following screenshot shows sample results of the previously shown code:

DB Browser for SQLite - C:\Users\gaston\HillarDjangoREST\01\restful01\db.sqlite3

File Edit View Help

New Database Open Database Write Changes Revert Changes

Database Structure Browse Data Edit Pragmas Execute SQL

Table: toys_toy

	id	created	name	description	toy_category	release_date	was_included_in_home
...	1	2017-10-09 15:11:51...	Snoopy talking action figure	Snoopy speaks five languages	Action figures	2017-10-09 ...	0
...	2	2017-10-09 15:12:12...	Hawaiian Barbie	Barbie loves Hawaii	Dolls	2017-10-09 ...	1

Now, let's write the following code to serialize the first `Toy` instance (`toy1`). The code file for the sample is included in the `hillar_django_restful_02_01` folder, in the `restful01/toy_serializers_test_01.py` file:

```

serializer_for_toy1 = ToySerializer(toy1)
print(serializer_for_toy1.data)

```

The following lines show the generated dictionary, specifically, a `rest_framework.utils.serializer_helpers.ReturnDict` instance, stored in the `serializer_for_toy1.data` attribute. The next lines show the results with easily understood formatting:

```

{
    'pk': 1,
    'name': 'Snoopy talking action figure',
    'description': 'Snoopy speaks five languages',
    'release_date': '2017-10-09T12:11:37.090335Z',
    'toy_category': 'Action figures',
    'was_included_in_home': False
}

```

Now, let's serialize the second `Toy` instance (`toy2`). The code file for the sample is included in the `hillar_django_restful_02_01` folder, in the `restful01/toy_serializers_test_01.py` file:

```
serializer_for_toy2 = ToySerializer(toy2)
print(serializer_for_toy2.data)
```

The following lines show the generated dictionary stored in the `serializer_for_toy2.data` attribute. The next lines show the results with easily understood formatting:

```
{
    'pk': 2,
    'name': 'Hawaiian Barbie',
    'description': 'Barbie loves Hawaii',
    'release_date': '2017-10-09T12:11:37.090335Z',
    'toy_category': 'Dolls',
    'was_included_in_home': True
}
```

We can easily render the dictionaries held in the `data` attribute into JSON with the help of the `rest_framework.renderers.JSONRenderer` class. The following lines create an instance of this class and then call the `render` method to render the dictionaries held in the `data` attribute into JSON. The code file for the sample is included in the `hillar_django_restful_02_01` folder, in the `restful01/toy_serializers_test_01.py` file:

```
json_renderer = JSONRenderer()
toy1_rendered_into_json = json_renderer.render(serializer_for_toy1.data)
toy2_rendered_into_json = json_renderer.render(serializer_for_toy2.data)
print(toy1_rendered_into_json)
print(toy2_rendered_into_json)
```

The following lines show the output generated from the two calls to the `render` method:

```
b'{"pk":1,"name":"Snoopy talking action figure","description":"Snoopy speaks\n>>> print(toy2_rendered_into_json)\nb'{"pk":2,"name":"Hawaiian Barbie","description":"Barbie loves Hawaii","rele
```

Now, we will work in the opposite direction: from serialized data to the population of a `Toy` instance. The following lines generate a new `Toy` instance from a JSON string (serialized data), that is, the code deserializes and parses the data. The code file for the sample is included in the `hillar_django_restful_02_01` folder, in the `restful01/toy_serializers_test_01.py` file:

```
json_string_for_new_toy = '{"name":"Clash Royale play set","description":"6 figu\njson_bytes_for_new_toy = bytes(json_string_for_new_toy, encoding="UTF-8")\nstream_for_new_toy = BytesIO(json_bytes_for_new_toy)\nparser = JSONParser()\nparsed_new_toy = parser.parse(stream_for_new_toy)\nprint(parsed_new_toy)
```

The first line creates a new string with the JSON that defines a new toy (`json_string_for_new_toy`). The next line converts the string to `bytes` and saves the results of the conversion in the `json_bytes_for_new_toy` variable. The `django.utils.six.BytesIO` class provides a buffered I/O implementation using an in-memory bytes buffer. The code uses this class to create a stream from the previously generated JSON bytes with the serialized data, `json_bytes_for_new_toy`, and saves the generated stream instance in the `stream_for_new_toy` variable.

We can easily deserialize and parse a stream into a Python model with the help of the `rest_framework.parsers.JSONParser` class. The next line creates an instance of this class and then calls the `parse` method with `stream_for_new_toy` as an argument, parses the stream into Python native datatypes, and saves the results in the `parsed_new_toy` variable.

After executing the previous lines, `parsed_new_toy` holds a Python dictionary, parsed from the stream. The following lines show the output generated after executing the preceding code snippet. The next lines show the results

with easily understood formatting:

```
{  
  
    'name': 'Clash Royale play set',  
  
    'description': '6 figures from Clash Royale',  
  
    'release_date': '2017-10-09T12:10:00.776594Z',  
  
    'toy_category': 'Playset',  
  
    'was_included_in_home': False  
  
}
```

The following lines use the `ToySerializer` class to generate a fully populated `Toy` instance named `toy3` from the Python dictionary, parsed from the stream. The code file for the sample is included in the `hillar_django_restful_02_01` folder, in the `restful01/toy_serializers_test_01.py` file:

```
new_toy_serializer = ToySerializer(data=parsed_new_toy)  
if new_toy_serializer.is_valid():  
    toy3 = new_toy_serializer.save()  
    print(toy3.name)
```

First, the code creates an instance of the `ToySerializer` class with the Python dictionary that we previously parsed from the stream (`parsed_new_toy`) passed as the `data` keyword argument. Then, the code calls the `is_valid` method to check whether the data is valid.

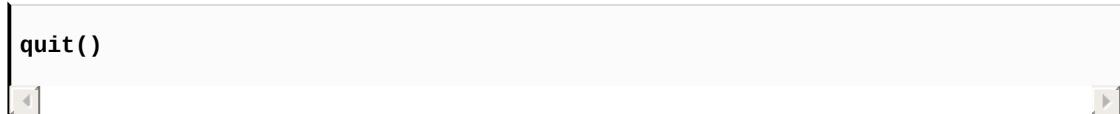
Note that we must always call `is_valid` before we attempt to access the serialized data representation when we pass a `data`

keyword argument in the creation of a serializer.

If the method returns `true`, we can access the serialized representation in the `data` attribute, and therefore, the code calls the `save` method that persists the new instance. In this case, it is a new `Toy` instance, and therefore the code to the `save` method inserts the corresponding row in the database and returns a fully populated `Toy` instance, saved in the `toy3` local variable. Then, the code prints one of the attributes from the fully populated `Toy` instance. After executing the previous code, we fully populated a new `Toy` instance: `toy3`.

As we can see from the previous code, Django REST framework makes it easy to serialize from objects to JSON and deserialize from JSON to objects, which are core requirements for our RESTful Web Service that has to perform CRUD operations.

Enter the following command to leave the Django shell with the Django project modules that we loaded to test serialization and deserialization:



A screenshot of a terminal window. The text 'quit()' is typed into the input field. The window has a standard OS X style with a title bar and scroll bars.

Test your knowledge

1. In Django REST framework, serializers are:
 1. Mediators between the view functions and Python primitives
 2. Mediators between the URLs and view functions
 3. Mediators between the model instances and Python primitives
2. If we want to create a simple `Toy` model that we will use to represent and persist toys in Django REST framework, we can create:
 1. A `Toy` class as a subclass of the `djangorestframework.models.Model` class
 2. A `Toy` class as a subclass of the `django.db.models.Model` class
 3. A `Toy` function in the `restframeworkmodels.py` file
3. In Django REST framework, parsers and renderers:
 1. Handle as mediators between model instances and Python primitives
 2. Handle as mediators between Python primitives and HTTP requests and responses
 3. Handle as mediators between the view functions and

Python primitives.

4. Which of the following commands starts the Django shell?
 1. `python manage.py shell`
 2. `python django.py shell`
 3. `django shell`

5. If we have a Django application named `computers` and a model called `memory`, what is the name of the table that Django's ORM will create to persist the model in the database?
 1. `computers_memories`
 2. `memory_computers`
 3. `computers_memory`

The rights answers are included in the [Appendix](#), *Solutions*.

Summary

In this chapter, we designed a RESTful Web Service to interact with a simple SQLite database and perform CRUD operations with toys. We defined the requirements for our web service and we understood the tasks performed by each HTTP method and the different scopes.

We created a model to represent and persist toys, and we executed migrations in Django to create the required tables in an SQLite database. We analyzed the tables that Django generated. We learned to manage serialization of toy instances into JSON representations with Django REST framework and the reverse process.

Now that we understand models, migrations, serialization, and deserialization with Django and Django REST framework, we will create Django views combined with serializer classes and start making HTTP requests to our web service. We will cover these topics in [chapter 3](#), *Creating API Views*.

Creating API Views

In this chapter, we have to run our first version of a RESTful Web Service powered by Django. We will write API views to process different HTTP requests and we will perform HTTP requests with command-line and GUI tools. We will analyze how Django and Django REST framework process each HTTP request. We will gain an understanding of:

- Creating Django views combined with serializer classes
- CRUD operations with Django views and the request methods
- Launching Django's development server
- Making HTTP GET requests that target a collection of instances with command-line tools
- Making HTTP GET requests that target a single instance with command-line tools
- Making HTTP GET requests with command-line tools
- Making HTTP POST requests with command-line tools
- Making HTTP PUT requests with command-line tools
- Making HTTP DELETE requests with command-line tools
- Making HTTP GET requests with Postman
- Making HTTP POST requests with Postman

Creating Django views combined with serializer classes

We have created the necessary model and its serializer. It is time to code the necessary elements to process HTTP requests and produce HTTP responses. Now, we will create Django views that use the `ToySerializer` class that we created previously to return JSON representations of the entities for each HTTP request that our web service will handle. Open the `toys/views.py` file. The following lines show the initial code for this file with just one import statement and a comment that indicates we should create the views:

```
from django.shortcuts import render

# Create your views here.
```

We will create our first version of the web service and we will use functions to keep the code as simple as possible. We will work with classes and more complex code in later examples. First, it is very important to understand how Django and Django REST framework work by way of a simple example.

Now, write the following lines in the `restful01/toys/views.py` file to create a `JSONResponse` class and declare two functions: `toy_list` and `toy_detail`. The code file for the sample is included in the `hillar_django_restful_03_01` folder, in the `restful01/toys/views.py` file:

```
from django.shortcuts import render
from django.http import JsonResponse
from django.views.decorators.csrf import csrf_exempt
```

```
from rest_framework.renderers import JSONRenderer
from rest_framework.parsers import JSONParser
from rest_framework import status
from toys.models import Toy
from toys.serializers import ToySerializer


class JsonResponse(HttpResponse):
    def __init__(self, data, **kwargs):
        content = JSONRenderer().render(data)
        kwargs['content_type'] = 'application/json'
        super(JsonResponse, self).__init__(content, **kwargs)


@csrf_exempt
def toy_list(request):
    if request.method == 'GET':
        toys = Toy.objects.all()
        toys_serializer = ToySerializer(toys, many=True)
        return JsonResponse(toys_serializer.data)

    elif request.method == 'POST':
        toy_data = JSONParser().parse(request)
        toy_serializer = ToySerializer(data=toy_data)
        if toy_serializer.is_valid():
            toy_serializer.save()
            return JsonResponse(toy_serializer.data,
                                status=status.HTTP_201_CREATED)
        return JsonResponse(toy_serializer.errors,
                            status=status.HTTP_400_BAD_REQUEST)

@csrf_exempt
def toy_detail(request, pk):
    try:
        toy = Toy.objects.get(pk=pk)
    except Toy.DoesNotExist:
        return HttpResponse(status=status.HTTP_404_NOT_FOUND)

    if request.method == 'GET':
        toy_serializer = ToySerializer(toy)
        return JsonResponse(toy_serializer.data)

    elif request.method == 'PUT':
        toy_data = JSONParser().parse(request)
        toy_serializer = ToySerializer(toy, data=toy_data)
        if toy_serializer.is_valid():
            toy_serializer.save()
            return JsonResponse(toy_serializer.data)
        return JsonResponse(toy_serializer.errors,
```

```
status=status.HTTP_400_BAD_REQUEST)

elif request.method == 'DELETE':
    toy.delete()
    return HttpResponseRedirect(status=status.HTTP_204_NO_CONTENT)
```

The highlighted lines show the expressions that evaluate the value of the `request.method` attribute to determine the actions to be performed based on the HTTP verb. The `JSONResponse` class is a subclass of the `django.http.HttpResponse` class. The `django.http.HttpResponse` superclass represents an HTTP response with string content.

The `JSONResponse` class renders its content in JSON. The class just declares the `__init__` method that creates a `rest_framework.renderers.JSONRenderer` instance and calls its `render` method to render the received data in JSON and save the returned byte string in the `content` local variable. Then, the code adds the `'content_type'` key to the response header with `'application/json'` as its value. Finally, the code calls the initializer for the base class with the JSON byte string and the key-value pair added to the header. This way, the class represents a JSON response that we use in the two functions to easily return a JSON response in each HTTP request our web service will process. Since Django 1.7, the `django.http.JsonResponse` class has accomplished the same goal. However, we created our own class for educational purposes in this example as well as to understand the difference between an `HttpResponse` and a `JSONResponse`.

The code uses the `@csrf_exempt` decorator in the two functions to ensure that the view sets a **CSRF** (short for **Cross-Site Request Forgery**) cookie. We do this to make it easier to test this example, which doesn't represent a production-ready web service. We will add security features to our RESTful Web Service later. Of course, it is very important to understand that we should never put a web service into production before configuring security and throttling rules.

Note that the previous code has many problems that we will analyze and fix in the forthcoming chapters. However, first, we need to understand how some basic things work.

Understanding CRUD operations with Django views and the request methods

When the Django server receives an HTTP request, Django creates an `HttpRequest` instance, specifically a `django.http.HttpRequest` object. This instance contains metadata about the request, and this metadata includes an HTTP verb such as GET, POST, or PUT. The `method` attribute provides a string representing the HTTP verb or method used in the request.

When Django loads the appropriate view that will process the request, it passes the `HttpRequest` instance as the first argument to the `view` function. The `view` function has to return an `HttpResponse` instance, specifically a `django.http.HttpResponse` instance.

The `toy_list` function lists all the toys or creates a new toy. The function receives an `HttpRequest` instance in the `request` argument. The function is capable of processing two HTTP verbs: `GET` and `POST`. The code checks the value of the `request.method` attribute to determine the code to be executed based on the HTTP verb.

If the HTTP verb is `GET`, the expression `request.method == 'GET'` will evaluate to `True` and the code has to list all the toys. The code will retrieve all the `Toy` objects from the database, use the `ToySerializer` to serialize all of them and return a `JSONResponse` instance built with the data generated by the `ToySerializer` serializer. The code creates the `ToySerializer` instance with the `many=True` argument to specify that multiple instances have to be serialized and not just one. Under the hood, Django uses a `ListSerializer` instance when the `many` argument value is set to `True`. This way, Django is capable of serializing a list of objects.

If the HTTP verb is `POST`, the code has to create a new toy based on the JSON data that is included in the body of the HTTP request. First, the code uses a `JSONParser` instance and calls its `parse` method with the `request` parameter that the `toy_list` function receives as an argument to parse the toy data provided as `JSON` data in the request body and saves the results in the `toy_data` local variable. Then, the code creates a `ToySerializer` instance with the previously retrieved data and calls the `is_valid` method to determine whether the `Toy` instance is valid or not. If the instance is valid, the code calls the `save` method to persist the instance in the database and returns a `JSONResponse` with the saved data in its body and a status equal to `status.HTTP_201_CREATED`, that is, `201 Created`.

Whenever we have to return a specific status different from the default `200 OK` status, it is a good practice to use the module variables defined in the `rest_framework.status` module and avoid using hard-coded numeric values. If you see `status=status.HTTP_201_CREATED`, as in the sample code, it is easy to understand that the status is an HTTP `201 created` status. If you read `status=201`, you have to remember what the number `201` stands for in the HTTP status codes.

The `toy_detail` function retrieves, updates, or deletes an existing toy. The function receives an `HttpRequest` instance in the `request` argument and the identifier for the toy to be retrieved, updated, or deleted in the `pk` argument. The function is capable of processing three HTTP verbs: `GET`, `PUT`, and `DELETE`. The code checks the value of the `request.method` attribute to determine the code to be executed based on the HTTP verb.

No matter what the HTTP verb is, the `toy_detail` function calls the `Toy.objects.get` method with the received `pk` as the `pk` argument to retrieve a `Toy` instance from the database based on the specified identifier, and saves it in the `toy` local variable. In case a toy with the specified identifier doesn't exist in the database, the code returns an `HttpResponse` with its status set to `status.HTTP_404_NOT_FOUND`, that is, `404 Not Found`.

If the HTTP verb is `GET`, the code creates a `ToySerializer` instance with `toy` as an argument and returns the data for the serialized toy in a `JSONResponse` that

will include the default HTTP `200 OK` status. The code returns the retrieved toy serialized as JSON in the response body.

If the HTTP verb is `PUT`, the code has to create a new toy based on the JSON data that is included in the HTTP request and use it to replace an existing toy. First, the code uses a `JSONParser` instance and calls its `parse` method with `request` as an argument to parse the toy data provided as JSON data in the request and saves the results in the `toy_data` local variable. Then, the code creates a `ToySerializer` instance with the `toy` instance previously retrieved from the database (`toy`) and the retrieved data that will replace the existing data (`toy_data`). Then, the code calls the `is_valid` method to determine whether the `toy` instance is valid or not. If the instance is valid, the code calls the `save` method to persist the instance with the replaced values in the database and returns a `JSONResponse` with the saved data serialized as JSON in its body and the default HTTP `200 OK` status. If the parsed data doesn't generate a valid `toy` instance, the code returns a `JSONResponse` with a status equal to `status.HTTP_400_BAD_REQUEST`, that is `400 Bad Request`.

If the HTTP verb is `DELETE`, the code calls the `delete` method for the `toy` instance previously retrieved from the database (`toy`). The call to the `delete` method erases the underlying row in the `toys_toy` table that we analyzed in the previous chapter. Thus, the toy won't be available anymore. Then, the code returns a `JSONResponse` with a status equal to `status.HTTP_204_NO_CONTENT` that is, `204 No Content`.

Routing URLs to Django views and functions

Now, we have to create a new Python file named `urls.py` in the `toys` folder, specifically, the `toys/urls.py` file. The following lines show the code for this file, which defines the URL patterns that specify the regular expressions that have to be matched in the request to run a specific function previously defined in the `views.py` file. The code file for the sample is included in the `hillar_django_restful_03_01` folder, in the `restful01/toys/urls.py` file:

```
from django.conf.urls import url
from toys import views

urlpatterns = [
    url(r'^toys/$', views.toy_list),
    url(r'^toys/(?P<pk>[0-9]+)$', views.toy_detail),
]
```

The `urlpatterns` list makes it possible to route URLs to views. The code calls the `django.conf.urls.url` function with the regular expression that has to be matched and the `view` function defined in the `views` module as arguments to create a `RegexURLPattern` instance for each entry in the `urlpatterns` list.

Now, we have to replace the code in the `urls.py` file in the `restful01` folder, specifically, the `restful01/urls.py` file. The file defines the root URL configurations, and therefore we must include the URL patterns declared in the previously coded `toys/urls.py` file. The following lines show the new code for the `restful01/urls.py` file. The code file for the sample is included in the `hillar_django_restful_03_01` folder, in the `restful01/urls.py` file:

```
from django.conf.urls import url, include

urlpatterns = [
```

```
|     url(r'^$', include('toys.urls')),  
| ]  
|<1>
```

```
>1
```

Launching Django's development server

Now, we can launch Django's development server to compose and send HTTP requests to our unsecured web service. Remember that we will add security later.

Execute the following command in a Linux or macOS Terminal, or in the Windows Command Prompt or Powershell that has our previously created virtual environment activated. Make sure you are in the `restful01` folder within the virtual environment's main folder:

```
python manage.py runserver
```

The following lines show the output after we execute the previous command. The development server is listening at port 8000:

```
Performing system checks...

System check identified no issues (0 silenced).
October 09, 2017 - 18:42:30
Django version 1.11.5, using settings 'restful01.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

With the previous command, we will start the Django development server and we will only be able to access it on our development computer. The previous command starts the development server at the default IP address, that is, `127.0.0.1` (`localhost`). It is not possible to access this IP address from other computers or devices connected to our LAN. Thus, if we want to make HTTP requests to our API from other computers or devices

connected to our LAN, we should use the development computer IP address, `0.0.0.0` (for IPv4 configurations) or `::` (for IPv6 configurations) as the desired IP address for our development server.

If we specify `0.0.0.0` as the desired IP address for IPv4 configurations, the development server will listen on every interface on port 8000. When we specify `::` for IPv6 configurations, it will have the same effect. In addition, it is necessary to open the default port `8000` in our firewalls (software and/or hardware) and configure port-forwarding to the computer that is running the development server. The following command launches Django's development server in an IPv4 configuration and allows requests to be made from other computers and devices connected to our LAN:

```
python manage.py runserver 0.0.0.0:8000
```

If you decide to compose and send HTTP requests from other computers or devices connected to the LAN, remember that you have to use the development computer's assigned IP address instead of `localhost`. For example, if the computer's assigned IPv4 IP address is `192.168.2.103`, instead of `localhost:8000`, you should use `192.168.2.103:8000`. Of course, you can also use the hostname instead of the IP address.

The previously explained configurations are very important because mobile devices might be the consumers of our RESTful Web Services and we will always want to test the apps that make use of our web services and APIs in our development environments.

Making HTTP GET requests that target a collection of instances

In [Chapter 1](#), *Installing the Required Software and Tools*, we installed command-line and GUI tools that were going to allow us to compose and send HTTP requests to the web services we were going to build throughout this book. Now, we will use the curl utility to make HTTP GET requests, specifically, HTTP GET requests that target a collection of toys. In case curl is not included in the path, make sure you replace curl with the full path to this utility.

Make sure you leave the Django development server running. Don't close the terminal or Command Prompt that is running this development server. Open a new Terminal in Linux or macOS, or a Command Prompt in Windows, and run the following command. It is very important that you enter the ending slash (/) because /toys won't match any of the patterns specified in `urlpatterns` in the `toys/urls.py` file. We aren't going to use options to follow redirects. Thus, we must enter `/toys/`, including the ending slash (/).

```
curl -X GET localhost:8000/toys/
```

The previous command will compose and send the following HTTP request: `GET http://localhost:8000/toys/`. The request is the simplest case in our RESTful Web Service because it will match and run the `views.toy_list` function, that is, the `toy_list` function we coded within the `toys/views.py` file. The function just receives `request` as a parameter because the URL pattern doesn't include any parameters. As the HTTP verb for the request is `GET`, the `request.method` property is equal to '`GET`', and therefore, the function will

execute the code that retrieves all the `Toy` objects and generates a JSON response with all of these `Toy` objects serialized.

The following lines show an example response for the HTTP request, with three `Toy` objects in the JSON response:

```
[{"pk":3,"name":"Clash Royale play set","description":"6 figures from Clash"}]
```

As we might notice from the previous response, the curl utility displays the JSON response in a single line, and therefore, it is a bit difficult to read. It is possible to use different tools, including some Python scripts, to provide a better format to the response. However, we will use the HTTPie command-line tool we installed in our virtual environment for this purpose later.

In this case, we know that the value of the `Content-Type` header key of the response is `application/json`. However, in case we want more details about the response, we can use the `-i` option to request curl to print the HTTP response headers and their key-value pairs. We can combine the `-i` and `-x` options by entering `-ix`.

Go back to the terminal in Linux or macOS, or the Command prompt in Windows, and run the following command:

```
curl -ix GET localhost:8000/toys/
```

The following lines show an example response for the HTTP request. The first lines show the HTTP response headers, including the status (`200 OK`) and the `Content-Type: application/json`. After the HTTP response headers, we can see the details for the three `Toy` objects in the JSON response:

```
HTTP/1.0 200 OK
Date: Tue, 10 Oct 2017 00:53:41 GMT
Server: WSGIServer/0.2 CPython/3.6.2
Content-Type: application/json
```

```
X-Frame-Options: SAMEORIGIN
Content-Length: 548

[{"pk":3,"name":"Clash Royale play set","description":"6 figures from Clash Roya
```

After we run the two requests, we will see the following lines in the window running the Django development server. The output indicates that the server received two HTTP requests with the `GET` verb and `/toys/` as the URI. The server processed both HTTP requests, returned a status code equal to `200`, and the response length was equal to `548` characters.

The response length might be different because the value for the primary key assigned to each toy will have an incidence in the response length. The first number after `HTTP/1.1.`" indicates the returned status code (`200`) and the second number the response length (`548`):

```
[09/Oct/2017 22:12:37] "GET /toys/ HTTP/1.1" 200 548
[09/Oct/2017 22:12:40] "GET /toys/ HTTP/1.1" 200 548
```

The following image shows two Terminal windows side-by-side on macOS. The Terminal window on the left-hand side is running the Django development server and displays the received and processed HTTP requests. The Terminal window on the right-hand side is running `curl` commands to generate the HTTP requests. It is a good idea to use a similar configuration to check the output while we compose and send the HTTP requests. Notice that the JSON outputs are a bit difficult to read because they don't use syntax highlighting:

```

(01) Gastons-MacBook-Pro:restful01 gaston$ python manage.py runserver 0.0.0.0:8000
Performing system checks...

System check identified no issues (0 silenced).
October 10, 2017 - 02:41:16
Django version 1.11.5, using settings 'restful01.settings'
Starting development server at http://0.0.0.0:8000/
Quit the server with CONTROL-C.
[10/Oct/2017 02:41:18] "GET /toys/ HTTP/1.1" 200 548
[10/Oct/2017 02:41:25] "GET /toys/ HTTP/1.1" 200 548

(01) Gastons-MacBook-Pro:bin gaston$ curl -X GET localhost:8000/toys/
[{"pk":3,"name":"Clash Royale play set","description":"6 figures from Clash Royale","release_date":"2017-10-09T12:10:00.776594Z","toy_category":"Playset","was_included_in_home":false},{"pk":2,"name":"Hawaiian Barbie","description":"Barbie loves Hawaii","release_date":"2017-10-09T12:11:37.090335Z","toy_category":"Dolls","was_included_in_home":true}, {"pk":1,"name":"Snoopy talking action figure","description":"Snoopy speaks five languages","release_date":"2017-10-09T12:11:37.090335Z","toy_category":"Action figures","was_included_in_home":false}](01) Gastons-MacBook-Pro:bin gaston$
(01) Gastons-MacBook-Pro:bin gaston$ curl -iX GET localhost:8000/toys/
HTTP/1.0 200 OK
Date: Tue, 10 Oct 2017 02:41:25 GMT
Server: WSGI Server/0.2 CPython/3.6.2
Content-Type: application/json
X-Frame-Options: SAMEORIGIN
Content-Length: 548

[{"pk":3,"name":"Clash Royale play set","description":"6 figures from Clash Royale","release_date":"2017-10-09T12:10:00.776594Z","toy_category":"Playset","was_included_in_home":false}, {"pk":2,"name":"Hawaiian Barbie","description":"Barbie loves Hawaii","release_date":"2017-10-09T12:11:37.090335Z","toy_category":"Dolls","was_included_in_home":true}, {"pk":1,"name":"Snoopy talking action figure","description":"Snoopy speaks five languages","release_date":"2017-10-09T12:11:37.090335Z","toy_category":"Action figures","was_included_in_home":false}](01) Gastons-MacBook-Pro:bin gaston$
(01) Gastons-MacBook-Pro:bin gaston$
```

Now, open a new Terminal in Linux or macOS, or a new Command Prompt in Windows, and activate the virtual environment we created. This way, you will be able to access the **HTTPie** utility we installed within the virtual environment.

We will use the `http` command to easily compose and send HTTP requests to `localhost:8000` and test the RESTful Web Service. **HTTPie** supports curl-like shorthand for `localhost`, and therefore we can use `:8000` as a shorthand that expands to `http://localhost:8000`. Run the following command and remember to enter the ending slash (/):

```
http :8000/toys/
```

The previous command will compose and send the following HTTP request: `GET http://localhost:8000/toys/`. The request is the same one we previously composed with the `curl` command. However, in this case, the **HTTPie** utility will display a colorized output and it will use multiple lines to display the JSON response, without any additional tweaks. The previous command is equivalent to the following command that specifies the `GET` method after `http`:

```
http :8000/toys/
```

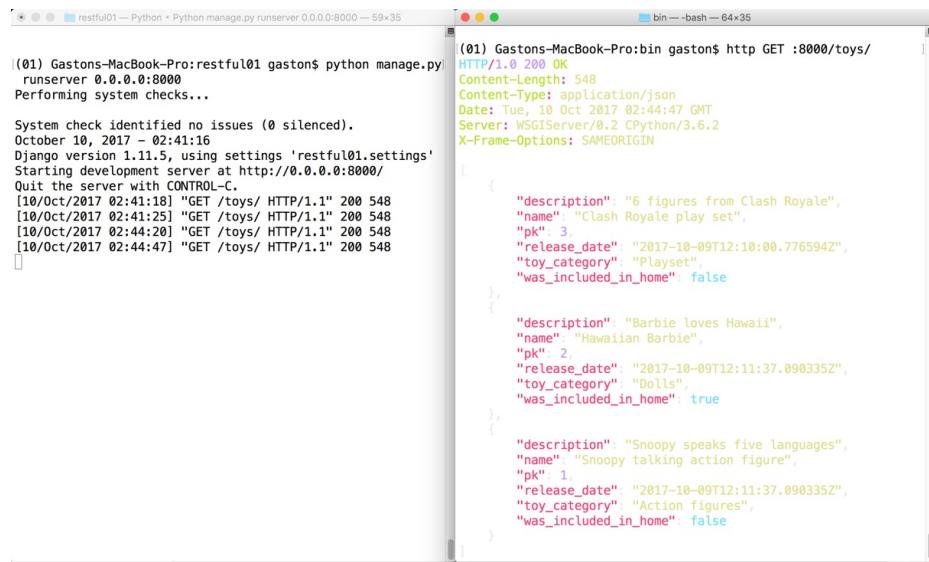
The following lines show an example response for the HTTP request, with the headers and the three `Toy` objects in the JSON response. It is indeed easier to understand the response compared with the results that were generated when we composed the HTTP request with curl. HTTPie automatically formats the JSON data received as a response and applies syntax highlighting, specifically, both colors and formatting:

```
HTTP/1.0 200 OK
Content-Length: 548
Content-Type: application/json
Date: Tue, 10 Oct 2017 01:26:52 GMT
Server: WSGIServer/0.2 CPython/3.6.2
X-Frame-Options: SAMEORIGIN

[
  {
    "description": "6 figures from Clash Royale",
    "name": "Clash Royale play set",
    "pk": 3,
    "release_date": "2017-10-09T12:10:00.776594Z",
    "toy_category": "Playset",
    "was_included_in_home": false
  },
  {
    "description": "Barbie loves Hawaii",
    "name": "Hawaiian Barbie",
    "pk": 2,
    "release_date": "2017-10-09T12:11:37.090335Z",
    "toy_category": "Dolls",
    "was_included_in_home": true
  },
  {
    "description": "Snoopy speaks five languages",
    "name": "Snoopy talking action figure",
    "pk": 1,
    "release_date": "2017-10-09T12:11:37.090335Z",
    "toy_category": "Action figures",
    "was_included_in_home": false
  }
]
```

We can achieve the same results by combining the output generated with the curl command with other utilities. However, HTTPie provides us exactly what we need for working with RESTful Web Services such as the one we are building with Django. We will use HTTPie to compose and send HTTP requests, but we will always provide the equivalent curl command. Remember that curl is faster when you need to execute it many times, such as when you prepare automated scripts.

The following image shows two Terminal windows side-by-side on macOS. The Terminal window on the left-hand side is running the Django development server and displays the received and processed HTTP requests. The Terminal window on the right-hand side is running HTTPie commands to generate the HTTP requests. Notice that the JSON output is easier to read compared to the output generated by the curl command:



```
(01) Gastons-MacBook-Pro:restful01 gaston$ python manage.py runserver 0.0.0.0:8000
Performing system checks...
System check identified no issues (0 silenced).
October 10, 2017 - 02:41:16
Django version 1.11.5, using settings 'restful01.settings'
Starting development server at http://0.0.0.0:8000/
Quit the server with CONTROL-C.
[10/Oct/2017 02:41:18] "GET /toys/ HTTP/1.1" 200 548
[10/Oct/2017 02:41:25] "GET /toys/ HTTP/1.1" 200 548
[10/Oct/2017 02:44:20] "GET /toys/ HTTP/1.1" 200 548
[10/Oct/2017 02:44:47] "GET /toys/ HTTP/1.1" 200 548
```

```
(01) Gastons-MacBook-Pro:bin gaston$ http GET :8000/toys/
HTTP/1.0 200 OK
Content-Length: 548
Content-Type: application/json
Date: Tue, 10 Oct 2017 02:44:47 GMT
Server: WSGIServer/0.2 CPython/3.6.2
X-Frame-Options: SAMEORIGIN

[{"description": "6 figures from Clash Royale", "name": "Clash Royale play set", "pk": 3, "release_date": "2017-10-09T12:10:00.776594Z", "toy_category": "Playset", "was_included_in_home": false}, {"description": "Barbie loves Hawaii", "name": "Hawaiian Barbie", "pk": 2, "release_date": "2017-09T12:11:37.090335Z", "toy_category": "Dolls", "was_included_in_home": true}, {"description": "Snoopy speaks five languages", "name": "Snoopy talking action figure", "pk": 1, "release_date": "2017-10-09T12:11:37.090335Z", "toy_category": "Action figures", "was_included_in_home": false}]
```

We can execute the `http` command with the `-b` option in case we don't want to include the header in the response. For example, the following line performs the same HTTP request but doesn't display the header in the response output, and therefore, the output will just display the JSON response:

```
http -b :8000/toys/
```



Making HTTP GET requests that target a single instance

Now, we will make HTTP GET requests that target a single `toy` instance. We will select one of the toys from the previous list and we will compose an HTTP request to retrieve only the chosen toy. For example, in the previous list, the first toy has a `pk` value equal to `3` because the results are ordered by the toy's name in ascending order. Run the following command to retrieve this toy. Use the `pk` value you have retrieved in the previous command for the first toy, as the `pk` number might be different if you execute the sample code or the commands more than once or you make changes to the `toys_toy` table. In this case, you don't have to enter an ending slash `(/)` because `/toys/3/` won't match any of the patterns specified in `urlpatterns` in the `toys/urls.py` file:

```
http :8000/toys/3
```

The following is the equivalent `curl` command:

```
curl -iX GET localhost:8000/toys/3
```

The previous commands will compose and send the following HTTP request: `GET http://localhost:8000/toys/3/`. The request has a number after `/toys/`, and therefore, it will match `'^toys/(?P<pk>[0-9]+)$'` and run the `views.toy_detail` function, that is, the `toy_detail` function declared within the `toys/views.py` file. The function receives `request` and `pk` as parameters because the URL pattern passes the number specified after `/toys/` in the `pk` parameter.

As the HTTP verb for the request is `GET`, the `request.method` property is equal

to 'GET', and therefore, the `toy_detail` function will execute the code that retrieves the `Toy` object whose primary key matches the `pk` value received as an argument and, if found, generates a JSON response with this `Toy` object serialized. The following lines show an example response for the HTTP request, with the `Toy` object that matches the `pk` value in the JSON response:

```
HTTP/1.0 200 OK
Content-Length: 182
Content-Type: application/json
Date: Tue, 10 Oct 2017 04:24:35 GMT
Server: WSGIServer/0.2 CPython/3.6.2
X-Frame-Options: SAMEORIGIN

{
    "description": "6 figures from Clash Royale",
    "name": "Clash Royale play set",
    "pk": 3,
    "release_date": "2017-10-09T12:10:00.776594Z",
    "toy_category": "Playset",
    "was_included_in_home": false
}
```

Now, we will compose and send an HTTP request to retrieve a toy that doesn't exist. For example, in the previous list, there is no toy with a `pk` value equal to `17500`. Run the following command to try to retrieve this toy. Make sure you use a `pk` value that doesn't exist. We must make sure that the utilities display the headers as part of the response because the response won't have a body:

```
http :8000/toys/17500
```

The following is the equivalent `curl` command:

```
curl -iX GET localhost:8000/toys/17500
```

The previous commands will compose and send the following HTTP request: `GET http://localhost:8000/toys/17500`. The request is the same as the

previous one we analyzed, with a different number for the `pk` parameter. The server will run the `views.toy_detail` function, that is, the `toy_detail` function declared within the `toys/views.py` file. The function will execute the code that retrieves the `Toy` object whose primary key matches the `pk` value received as an argument and a `Toy.DoesNotExist` exception will be thrown and captured because there is no toy with the specified `pk` value. Thus, the code will return an HTTP `404 Not Found` status code. The following lines show an example header response for the HTTP request:

```
HTTP/1.0 404 Not Found
Content-Length: 0
Content-Type: text/html; charset=utf-8
Date: Tue, 10 Oct 2017 15:54:59 GMT
Server: WSGIServer/0.2 CPython/3.6.2
X-Frame-Options: SAMEORIGIN
```

Making HTTP POST requests

Now, we will compose and send an HTTP request to create a new toy:

```
http POST :8000/toys/ name="PvZ 2 puzzle" description="Plants vs
Zombies 2 puzzle" toy_category="Puzzle" was_included_in_home=false
release_date="2017-10-08T01:01:00.776594Z"
```

The following is the equivalent `curl` command. It is very important to use the `-H "Content-Type: application/json"` option to indicate to curl that it should send the data specified after the `-d` option as `application/json` instead of the default `application/x-www-form-urlencoded`:

```
curl -iX POST -H "Content-Type: application/json" -d '{"name": "PvZ
2 puzzle", "description": "Plants vs Zombies 2 puzzle",
"toy_category": "Puzzle", "was_included_in_home": "false",
"release_date": "2017-10-08T01:01:00.776594Z"}'
localhost:8000/toys/
```

The previous commands will compose and send the following HTTP request: `POST http://localhost:8000/toys/` with the following JSON key-value pairs:

```
{
  "name": "PvZ 2 puzzle",
  "description": "Plants vs Zombies 2 puzzle",
  "toy_category": "Puzzle",
  "was_included_in_home": "false",
  "release_date": "2017-10-08T01:01:00.776594Z"
}
```

The request specifies `/toys/`, and therefore, it will match the `'^toys/$'` regular expression and run the `views.toy_list` function, that is, the `toy_detail`

function declared within the `toys/views.py` file. The function just receives `request` as a parameter because the URL pattern doesn't include any parameters. As the HTTP verb for the request is `POST`, the `request.method` property is equal to '`POST`', and therefore, the function will execute the code that parses the JSON data received in the request. Then, the function creates a new `Toy` and, if the data is valid, it saves the new `Toy` to the `toys_toy` table in the SQLite database. If the new `Toy` was successfully persisted in the database, the function returns an HTTP `201 Created` status code and the recently persisted `Toy` serialized to JSON in the response body. The following lines show an example response for the HTTP request, with the new `Toy` object in the JSON response:

```
HTTP/1.0 201 Created
Content-Length: 171
Content-Type: application/json
Date: Tue, 10 Oct 2017 16:27:57 GMT
Server: WSGIServer/0.2 CPython/3.6.2
X-Frame-Options: SAMEORIGIN

{
    "description": "Plants vs Zombies 2 puzzle",
    "name": "PvZ 2 puzzle",
    "pk": 4,
    "release_date": "2017-10-08T01:01:00.776594Z",
    "toy_category": "Puzzle",
    "was_included_in_home": false
}
```

Making HTTP PUT requests

Now, we will compose and send an HTTP request to update an existing toy, specifically, the previously added toy. We have to check the value assigned to `pk` in the previous response and replace `4` in the command with the returned value. For example, if the value for `pk` was `4`, you should use `:8000/toys/4` instead of `:8000/toys/4`:

```
http PUT :8000/toys/4 name="PvZ 3 puzzle" description="Plants vs Zombies 3 puzzle"
```

The following is the equivalent `curl` command. As with the previous `curl` example, it is very important to use the `-H "Content-Type: application/json"` option to indicate `curl` to send the data specified after the `-d` option as `application/json` instead of the default `application/x-www-form-urlencoded`:

```
curl -iX PUT -H "Content-Type: application/json" -d '{"name": "PvZ 3 puzzle", "de
```

The previous commands will compose and send the following HTTP request: `PUT http://localhost:8000/toys/4` with the following JSON key-value pairs:

```
{
    "name": "PvZ 3 puzzle",
    "description": "Plants vs Zombies 3 puzzle",
    "toy_category": "Puzzles & Games",
    "was_included_in_home": "false",
    "release_date": "2017-10-08T01:01:00.776594Z"
}
```

The request has a number after `/toys/`, and therefore, it will match the `'^toys/(?P<pk>[0-9]+)$'` regular expression and run the `views.toy_detail`

function, that is, the `toy_detail` function declared within the `toys/views.py` file. The function receives `request` and `pk` as parameters because the URL pattern passes the number specified after `/toys/` in the `pk` parameter. As the HTTP verb for the request is `PUT`, the `request.method` property is equal to `'PUT'`, and therefore, the function will execute the code that parses the JSON data received in the request. Then, the function will create a `Toy` instance from this data and update the existing toy in the database. If the toy was successfully updated in the database, the function returns an HTTP `200 OK` status code and the recently updated `Toy` serialized to JSON in the response body. The following lines show an example response for the HTTP request, with the updated `Toy` object in the JSON response:

```
HTTP/1.0 200 OK
Content-Length: 180
Content-Type: application/json
Date: Tue, 10 Oct 2017 17:06:43 GMT
Server: WSGIServer/0.2 CPython/3.6.2
X-Frame-Options: SAMEORIGIN

{
    "description": "Plants vs Zombies 3 puzzle",
    "name": "PvZ 3 puzzle",
    "pk": 4,
    "release_date": "2017-10-08T01:01:00.776594Z",
    "toy_category": "Puzzles & Games",
    "was_included_in_home": false
}
```

In order to successfully process a `PUT` HTTP request that updates an existing toy, we must provide values for all the required fields. We will compose and send an HTTP request to try to update an existing toy, and we will fail to do so because we will just provide a value for the name. As in the previous request, we will use the value assigned to `pk` in the last toy we added:

```
http PUT :8000/toys/4 name="PvZ 4 puzzle"
```

The following is the equivalent `curl` command:

```
curl -iX PUT -H "Content-Type: application/json" -d '{"name": "PvZ 4 puzzle"}' localhost:8000/toys/4
```

The previous commands will compose and send the following HTTP request: `PUT http://localhost:8000/toys/4` with the following JSON key-value pair:

```
{  
    "name": "PvZ 4 puzzle",  
}
```

The request will execute the same code we explained for the previous request. As we didn't provide all the required values for a `Toy` instance, the `toy_serializer.is_valid()` method will return `False` and the function will return an `HTTP 400 Bad Request` status code and the details generated in the `toy_serializer.errors` attribute serialized to JSON in the response body. The following lines show an example response for the HTTP request, with the required fields that our request didn't include values in the JSON response (`description`, `release_date`, and `toy_category`):

```
HTTP/1.0 400 Bad Request  
Content-Length: 129  
Content-Type: application/json  
Date: Tue, 10 Oct 2017 17:23:46 GMT  
Server: WSGIServer/0.2 CPython/3.6.2  
X-Frame-Options: SAMEORIGIN  
  
{  
    "description": [  
        "This field is required."  
    ],  
    "release_date": [  
        "This field is required."  
    ],  
    "toy_category": [  
        "This field is required."  
    ]  
}
```

When we want our API to be able to update a single field for an existing resource, in this case, an existing toy, we should provide an implementation for the PATCH method. The PUT method is meant to replace an entire resource and the PATCH method is meant to apply a delta to an existing resource. We can write code in the handler for the PUT method to apply a delta to an existing resource, but it is a better practice to use the PATCH method for this specific task. We will work with the PATCH method later.

Making HTTP DELETE requests

Now, we will compose and send an HTTP request to delete an existing toy, specifically, the last toy we added. As in our last HTTP request, we have to check the value assigned to `pk` in the previous response and replace `4` in the command with the returned value:

```
http DELETE :8000/toys/4
```



The following is the equivalent `curl` command:

```
curl -iX DELETE localhost:8000/toys/4
```



The previous commands will compose and send the following HTTP request: `DELETE http://localhost:8000/toys/4`. The request has a number after `/toys/`, and therefore, it will match the `'^toys/(?P<pk>[0-9]+)$'` regular expression and run the `views.toy_detail` function, that is, the `toy_detail` function declared within the `toys/views.py` file. The function receives `request` and `pk` as parameters because the URL pattern passes the number specified after `/toys/` in the `pk` parameter. As the HTTP verb for the request is `DELETE`, the `request.method` property is equal to `'DELETE'`, and therefore, the function will execute the code that parses the JSON data received in the request. Then, the function creates a `Toy` instance from this data and deletes the existing toy in the database. If the toy was successfully deleted in the database, the function returns an HTTP `204 No Content` status code. The following lines show an example response to the HTTP request after successfully deleting an existing toy:

```
HTTP/1.0 204 No Content
Content-Length: 0
Content-Type: text/html; charset=utf-8
```

Date: Tue, 10 Oct 2017 17:45:40 GMT
Server: WSGIServer/0.2 CPython/3.6.2
X-Frame-Options: SAMEORIGIN



Making HTTP GET requests with Postman

Now, we will use one of the GUI tools we installed in [Chapter 1, Installing the Required Software and Tools](#), specifically Postman. We will use this GUI tool to compose and send HTTP requests to the web service.

The first time you execute Postman, you will see a modal that provides shortcuts to the most common operations. Make sure you close this modal so that we can focus on the main UI for Postman.

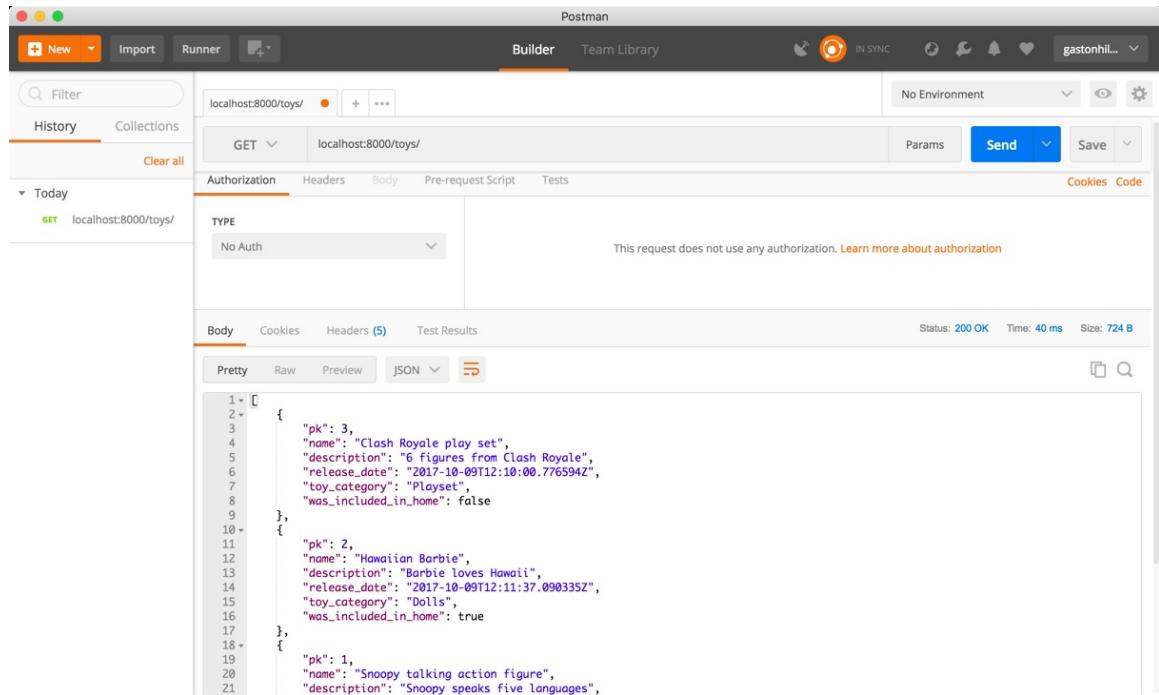
We will use the Builder tab in Postman to easily compose and send diverse HTTP requests to `localhost:8000` and test the RESTful Web Service with this GUI tool. Postman doesn't support curl-like shorthand for localhost, and therefore, we cannot use the same shorthand we have been using when composing requests with HTTPie.

Select GET in the drop-down menu on the left-hand side of the Enter request URL textbox, and enter `localhost:8000/toys/` in this textbox on the right-hand side of the drop-down menu. Then, click Send and Postman will display the following information:

- Status: 200 OK.
- Time: The time it took for the request to be processed.
- Size: The approximate response size (sum of body size plus headers size).
- Body: The response body with all the toys formatted as JSON with syntax highlighting. The default view for the body is Pretty

and it activates syntax highlighting.

The following screenshot shows the JSON response body in Postman for the HTTP GET request to `localhost:8000/toys/`.



The screenshot shows the Postman application interface. In the top navigation bar, 'Builder' is selected. The main workspace shows a GET request to 'localhost:8000/toys/'. The 'Body' tab is active, displaying a JSON array of toy objects. The JSON is formatted with line numbers and syntax highlighting. The array contains three objects, each representing a toy with fields like pk, name, description, release_date, toy_category, and was_included_in_home. The status bar at the bottom right indicates a 200 OK status, 40 ms time, and 724 B size.

```
1 [  
2 {  
3     "pk": 3,  
4     "name": "Clash Royale play set",  
5     "description": "6 figures from Clash Royale",  
6     "release_date": "2017-10-09T12:18:00.776594Z",  
7     "toy_category": "Playset",  
8     "was_included_in_home": false  
9 },  
10 {  
11     "pk": 2,  
12     "name": "Hawaiian Barbie",  
13     "description": "Barbie loves Hawaii",  
14     "release_date": "2017-10-09T12:11:37.090335Z",  
15     "toy_category": "Dolls",  
16     "was_included_in_home": true  
17 },  
18 {  
19     "pk": 1,  
20     "name": "Snoopy talking action figure",  
21     "description": "Snoopy speaks five languages",  
22 }
```

Click on the Headers tab on the right-hand side of the Body and Cookies tab to read the response headers. The following screenshot shows the layout for the response headers that Postman displays for the previous response. Notice that Postman displays the Status on the right-hand side of the response and doesn't include it as the first line of the key-value pairs that compose the headers, as when we worked with both the `curl` and `http` command-line utilities.

The screenshot shows the Postman application interface. At the top, there is a header bar with the URL "localhost:8000/toys/" and several icons. Below the header is a toolbar with "GET", "localhost:8000/toys/", "Params", "Send", and "Save" buttons. A dropdown menu for "Environment" is set to "No Environment".

The main area has tabs for "Authorization", "Headers", "Body", "Pre-request Script", "Tests", "Cookies", and "Code". The "Authorization" tab is currently selected. It displays a dropdown menu for "TYPE" with "No Auth" selected. A note states: "This request does not use any authorization. [Learn more about authorization](#)".

Below the tabs, there are buttons for "Body", "Cookies", "Headers (5)", and "Test Results". The "Headers (5)" button is highlighted with an orange underline. To the right, status information is shown: "Status: 200 OK", "Time: 40 ms", and "Size: 724 B".

The "Headers (5)" section lists the following headers:

- Content-Length → 548
- Content-Type → application/json
- Date → Tue, 10 Oct 2017 18:27:44 GMT
- Server → WSGIServer/0.2 CPython/3.6.2
- X-Frame-Options → SAMEORIGIN

Making HTTP POST requests with Postman

Now, we will use the Builder tab in Postman to compose and send an HTTP POST request to create a new toy. Perform the following steps:

1. Click on the plus (+) button on the right-hand side of the tab that displayed the previous request. This way, you will create a new tab.
2. Select Request in the New drop-down menu located in the upper-left corner.
3. Select POST in the drop-down menu on the left-hand side of the Enter request URL textbox.
4. Enter `localhost:8000/toys/` in that textbox on the right-hand side of the drop-down menu.
5. Click Body on the right-hand side of Authorization and Headers, within the panel that composes the request.
6. Activate the raw radio button and select JSON (`application/json`) in the drop-down menu on the right-hand side of the binary radio button. Postman will automatically add a `content-type = application/json` header, and therefore, you will notice the Headers tab will be renamed to Headers (1), indicating to us that there is one key-value pair specified for the request headers.
7. Enter the following lines in the textbox below the radio buttons, within the Body tab:

```
{  
    "name": "Wonderboy puzzle",  
    "description": "The Dragon's Trap puzzle",  
    "toy_category": "Puzzles & Games",  
    "was_included_in_home": "false",  
    "release_date": "2017-10-03T01:01:00.776594Z"  
}
```

The following screenshot shows the request body in Postman:

The screenshot shows the Postman application interface. At the top, there are two tabs both labeled "localhost:8000/toys/". To the right of the tabs are buttons for "No Environment", "Send", and "Save". Below the tabs, there are buttons for "POST", "Params", "Cookies", and "Code". Underneath these buttons, there are tabs for "Authorization", "Headers (1)", "Body", "Pre-request Script", and "Tests". The "Body" tab is currently selected and has a blue dot next to it. Below the tabs, there are radio buttons for "form-data", "x-www-form-urlencoded", "raw", and "binary", with "raw" selected. To the right of these buttons is a dropdown menu set to "JSON (application/json)". The main body area contains the JSON code shown in the first code block.

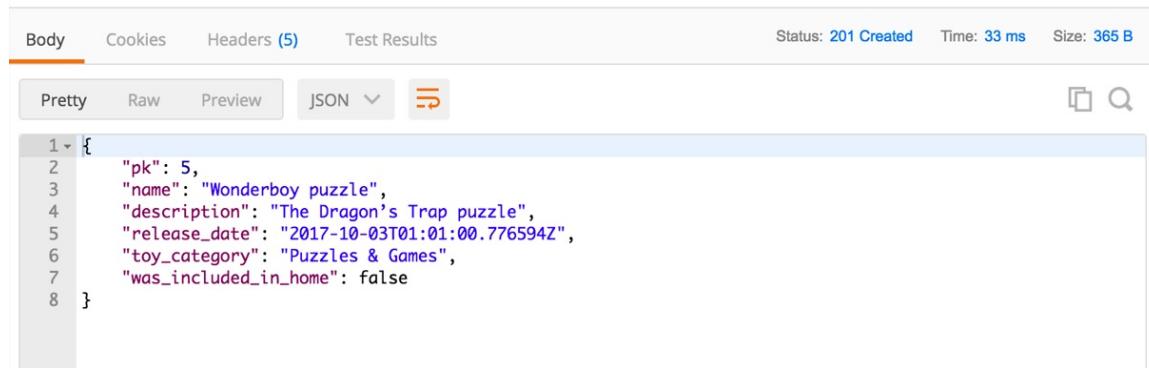
```
1 {  
2     "name": "Wonderboy puzzle",  
3     "description": "The Dragon's Trap puzzle",  
4     "toy_category": "Puzzles & Games",  
5     "was_included_in_home": "false",  
6     "release_date": "2017-10-03T01:01:00.776594Z"  
7 }  
8
```

We followed the necessary steps to create an HTTP POST request with a JSON body that specifies the necessary key-value pairs to create a new toy. Click Send and Postman will display the following information:

- Status: 201 Created
- Time: The time it took for the request to be processed
- Size: The approximate response size (sum of body size plus headers size)
- Body: The response body with the recently added toy formatted as

JSON with syntax highlighting

The following screenshot shows the JSON response body in Postman for the HTTP POST request:

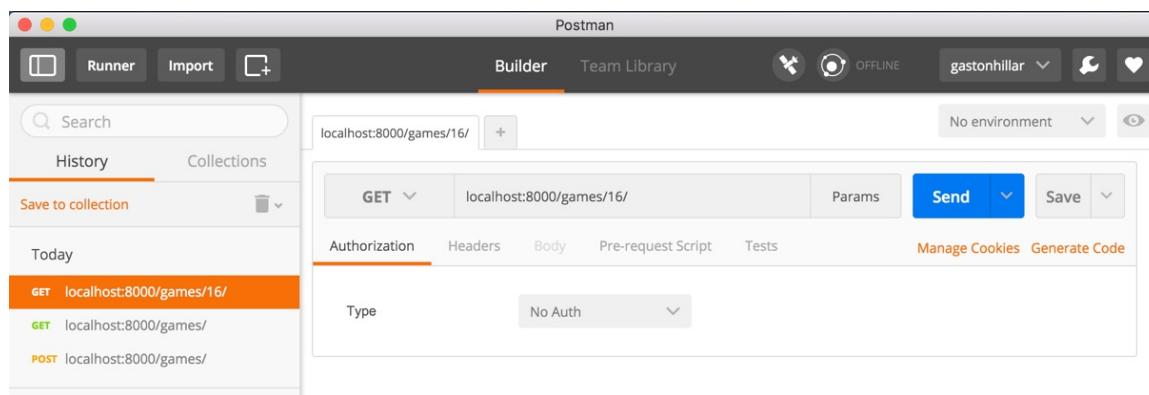


A screenshot of the Postman application interface. The top navigation bar includes tabs for 'Body' (which is selected), 'Cookies', 'Headers (5)', and 'Test Results'. On the right side of the header, it shows 'Status: 201 Created', 'Time: 33 ms', and 'Size: 365 B'. Below the header, there are buttons for 'Pretty', 'Raw', 'Preview', and 'JSON' (with a dropdown arrow). To the right of these buttons are icons for copy and search. The main content area displays a JSON object with line numbers on the left. The JSON content is:

```
1 {  
2   "pk": 5,  
3   "name": "Wonderboy puzzle",  
4   "description": "The Dragon's Trap puzzle",  
5   "release_date": "2017-10-03T01:00:776594Z",  
6   "toy_category": "Puzzles & Games",  
7   "was_included_in_home": false  
8 }
```

If we want to compose and send an HTTP PUT request with Postman, it is necessary to follow the previously explained steps to provide JSON data within the request body.

One of the nice features included in Postman is that we can easily review and run the HTTP requests we have made again by browsing the saved History shown on the left-hand side of the Postman window. The History panel displays a list with the HTTP verb followed by the URL for each HTTP request we have composed and sent. We just need to click on the desired HTTP request and click Send to run it again. The following screenshot shows the many HTTP requests in the History panel and the first HTTP GET request that was executed selected so it can be easily resent:



Test your knowledge

Let's see whether you can answer the following questions correctly:

1. The `urlpatterns` list declared in the `urls.py` file makes it possible to:
 1. Route URLs to Django models
 2. Route URLs to Django views
 3. Route URLs to Python primitives
2. When the Django server receives an HTTP request, Django creates an instance of which of the following classes?
 1. `django.restframework.HttpRequest`
 2. `django.http.HttpRequest`
 3. `django.http.Request`
3. A view function has to return an instance of which of the following classes?
 1. `django.http.HttpResponse`
 2. `django.http.Response`
 3. `django.restfremework.HttpResponse`
4. Whenever you have to return a specific status different from the default `200 OK` status, it is a good practice to use the module

variables defined in which of the following modules?

1. `rest_framework.HttpStatus`
 2. `django.status`
 3. `rest_framework.status`
5. If you want to retrieve a Toy instance whose primary key value is equal to `10` and save it in the `toy` variable, which line of code would you write?
1. `toy = Toy.get_by(pk=10)`
 2. `toy = Toy.objects.all(pk=10)`
 3. `toy = Toy.objects.get(pk=pk)`

The rights answers are included in the [Appendix](#), *Solutions*.

Summary

In this chapter, we executed our first version of a simple Django RESTful Web Service that interacts with an SQLite database. We wrote API views to process diverse HTTP requests on a collection of toys and on a specific toy. We worked with the following HTTP verbs: GET, POST, and PUT. We configured the URL patterns list to route URLs to views.

Then, we started the Django development server and we used command-line tools (curl and HTTPie) to compose and send diverse HTTP requests to our RESTful Web Service. We learned how HTTP requests were processed in Django and our code. Finally, we worked with Postman, a GUI tool, to compose and send other HTTP requests to our RESTful Web Service.

Now that we understand the basics of a RESTful Web Service with Django REST framework and a simple SQLite database, we will work with a seriously powerful PostgreSQL database, use class-based views instead of function views, and we will take advantage of advanced features included in Django REST framework to work with different content types, without writing a huge amount of code. We will cover these topics in the next chapter.

Using Generalized Behavior from the APIView Class

In this chapter, we will improve our simple RESTful Web Service. We will make it possible for it to work with diverse content types without writing a huge amount of code. We will take advantage of advanced features and generalized behaviors included in the Django REST framework to enable multiple parsers and renderers. We will gain an understanding of:

- Taking advantage of model serializers
- Understanding accepted and returned content types
- Making unsupported HTTP OPTIONS requests with command-line tools
- Understanding decorators that work as wrappers
- Using decorators to enable different parsers and renderers
- Taking advantage of content negotiation classes
- Making supported HTTP OPTIONS requests with command-line tools
- Working with different content types
- Sending HTTP requests with unsupported HTTP verbs

Taking advantage of model serializers

In [Chapter 1](#), *Installing the Required Software and Tools*, we created the toy model (the `Toy` class) and its serializer (the `ToySerializer` class). When we wrote the code for the `ToySerializer` class, we had to declare many attributes with the same names that we used in the `Toy` class. The `ToySerializer` class is a subclass of the `rest_framework.serializers.Serializer` superclass; it declares attributes that we manually mapped to the appropriate types, and overrides the `create` and `update` methods. However, we repeated a lot of code and information that was already included in the toy model, such as the types and the `max_length` values that specify the maximum length for each string field.

Now, we will take advantage of model serializers to simplify code and to avoid repeating information that is already included in the model. We will create a new version of the existing `ToySerializer` class that will inherit from the `rest_framework.serializers.ModelSerializer` superclass instead of inheriting from the `rest_framework.serializers.Serializer` superclass.

The `ModelSerializer` class automatically populates a set of default fields and default validators by retrieving metadata from the related model class that we must specify. In addition, the `ModelSerializer` class provides default implementations for the `create` and `update` methods. In this case, we will take advantage of these default implementations because they will be suitable to provide our necessary `create` and `update` methods.

Go to the `restful01/toys` folder and open the `serializers.py` file. The code file for the sample is included in the `hillar_django_restful_04_01` folder, in the `restful01/toys/serializers.py` file. Replace the code in this file with the following code that declares the new version of the `ToySerializer` class:

```
from rest_framework import serializers
from toys.models import Toy

class ToySerializer(serializers.ModelSerializer):
    class Meta:
        model = Toy
        fields = ('id',
                  'name',
                  'description',
                  'release_date',
                  'toy_category',
                  'was_included_in_home')
```



The new version of the `ToySerializer` class declares a `Meta` inner class that declares the following two attributes:

- `model`: This attribute specifies the model related to the serializer, that is, the `Toy` class
- `fields`: This attribute specifies a tuple of `string` whose values indicate the field names that we want to include in the serialization from the related model (the `Toy` class)

The new version of the `ToySerializer` class doesn't need to override either the `create` or `update` methods because the generic behavior provided by the `ModelSerializer` class will be enough in this case. The `ModelSerializer` superclass provides implementations for both methods.

With the changes we have made, we removed a nice amount of code from the `ToySerializer` class. In the new version, we just had to specify the related model and the desired set of fields in a tuple. Now, the types and `max_length` values related to the toy fields are only included in the `Toy` class.

If you have previous experience with the Django Web framework, you will realize that the `Serializer` and `ModelSerializer` classes in the Django REST framework are similar to the `Form` and `ModelForm` classes in Django.

You can press *Ctrl + C* to quit Django's development server and execute the command that we learned in [Chapter 3](#), *Creating API Views*, to run the server to start it again. In this case, we just edited one file, and in case you didn't stop the development server, Django will detect the changes when we save the changes to the file and it will automatically restart the server.

The following lines show sample output that you will see after you save the changes in the edited Python file. The lines indicate that Django has restarted the development server and successfully performed a system check that identified no issues:

```
System check identified no issues (0 silenced).
October 13, 2017 - 04:11:13
Django version 1.11.5, using settings 'restful01.settings'
Starting development server at http://0.0.0.0:8000/
Quit the server with CONTROL-C.
```

You can use the command-line and GUI tools we used in [Chapter 3](#), *Creating API Views*, to test the new version of our RESTful Web Service that takes advantage of model serializers. The behavior will be the same as in the previous version. However, we definitely have less code to maintain and we have removed duplicated data.

Understanding accepted and returned content types

So far, our RESTful Web Service has been working with JSON for the response body. The code we wrote in the `toys/views.py` file in [Chapter 3, Creating API Views](#), declares a `JSONResponse` class and two function-based views. These functions return a `JSONResponse` when it is necessary to return JSON data and a `django.HttpResponse.HttpResponse` instance when the response is just an HTTP status code. No matter the accepted content type specified in the HTTP request header, the view functions always provide the same content in the response body: JSON.

Run the following command to retrieve all the toys with the `Accept` request header key set to `text/html`. Remember that the virtual environment we have created in [Chapter 3, Creating API Views](#), must be activated in order to run the next `http` command:

```
http :8000/toys/ Accept:text/html
```

The following is the equivalent `curl` command:

```
curl -H "Accept: text/html" -iX GET localhost:8000/toys/
```

The previous commands will compose and send the following HTTP request: `GET http://localhost:8000/toys/`. These commands specify the `text/html` value for the `Accept` key in the request header. This way, the HTTP request indicates that it accepts a response of `text/html`.

The header response for the request will include the following line:

```
Content-Type: application/json
```

Now, run the following command to retrieve all the toys with different values with the `Accept` request header key set to `text/html`.

Run the following command to retrieve all the toys with the `Accept` request header key set to `application/json`:

```
http :8000/toys/ Accept:application/json
```

The following is the equivalent `curl` command:

```
curl -H "Accept: application/json" -iX GET localhost:8000/toys/
```

The previous commands will compose and send the following HTTP request: `GET http://localhost:8000/toys/`. These commands specify the `application/json` value for the `Accept` key in the request header. This way, the HTTP request indicates that it accepts a response of `application/json`.

The header response for the request will include the following line:

```
Content-Type: application/json
```

The first group of commands defined the `text/html` value for the `Accept` request header key. The second group of commands defined the `application/json` value for the `Accept` request header key. However, both produced the same results and the responses were always in the JSON format. The view functions don't take into account the value specified for the `Accept` request header key in the HTTP requests. No matter the value indicated for the `Accept` request header key, the response is always in the JSON format.

We want to provide support for other formats. However, we don't want to write a huge amount of code to do so. Thus, we will take advantage of additional features included in the Django REST framework that will make it easy for us to support additional formats for our RESTful Web Service.

Making unsupported HTTP OPTIONS requests with command-line tools

Sometimes, we don't know which are the HTTP methods or verbs that a resource or resource collection supports in a RESTful Web Service. In order to provide a solution to this problem, we can compose and send an HTTP request with the `OPTIONS` HTTP verb and the URL for the resource or the resource collection.

If the RESTful Web Service implements the `OPTIONS` HTTP verb for a resource or resource collection, it will build a response with an `Allow` key in the response header. The value for this key will include a comma-separated list of HTTP verbs or methods that it supports. In addition, the response header will include additional information about other supported options, such as the content type it is capable of parsing from the request and the content type it is capable of rendering in the response.

For example, if we want to know which HTTP verbs the `toys` collection supports, we can run the following command:

```
http OPTIONS :8000/toys/
```

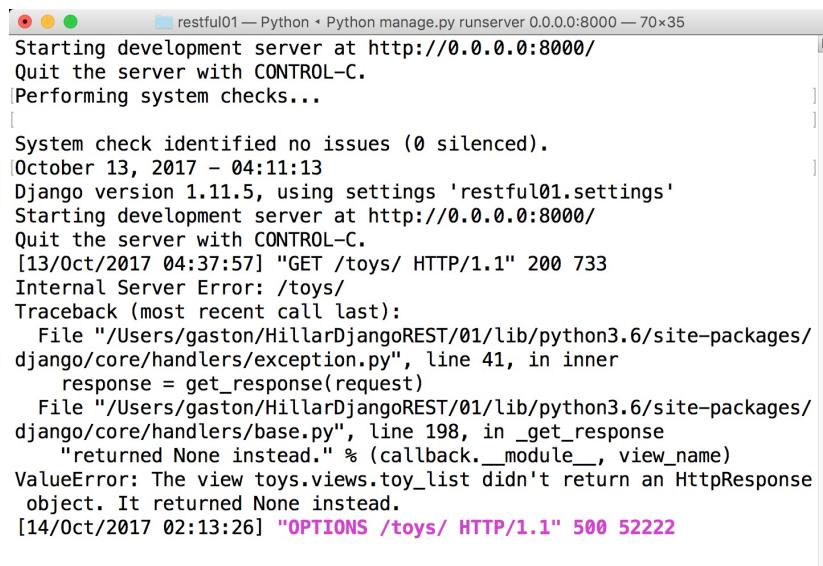
Notice that the command will generate an error in the Django development server.

The following is the equivalent curl command:

```
curl -iX OPTIONS localhost:8000/toys/
```

The previous command will compose and send the following HTTP request: `OPTIONS http://localhost:8000/toys/`. The request specifies `/toys/`, and therefore, it will match the `'^toys/$'` regular expression and run the `views.toy_list` function, that is, the `toy_list` function declared within the `toys/views.py` file. This function only runs code when the `request.method` is equal to either `'GET'` or `'POST'`. In this case, `request.method` is equal to `'OPTIONS'`, and therefore, the function won't run any code. The function won't return the expected `HttpResponse` instance.

The lack of the expected `HttpResponse` instance generates an internal server error in Django's development server. The console output for the development server will display details about the internal server error and a traceback similar to the one shown in the next screenshot. The last lines indicate that there is a `ValueError` because the `toys_list` function didn't return an `HttpResponse` instance and returned `None` instead:



```
restful01 — Python — Python manage.py runserver 0.0.0.0:8000 — 70x35
Starting development server at http://0.0.0.0:8000/
Quit the server with CONTROL-C.
Performing system checks...
[...]
System check identified no issues (0 silenced).
[October 13, 2017 - 04:11:13
Django version 1.11.5, using settings 'restful01.settings'
Starting development server at http://0.0.0.0:8000/
Quit the server with CONTROL-C.
[13/Oct/2017 04:37:57] "GET /toys/ HTTP/1.1" 200 733
Internal Server Error: /toys/
Traceback (most recent call last):
  File "/Users/gaston/HillarDjangoREST/01/lib/python3.6/site-packages/
django/core/handlers/exception.py", line 41, in inner
    response = get_response(request)
  File "/Users/gaston/HillarDjangoREST/01/lib/python3.6/site-packages/
django/core/handlers/base.py", line 198, in _get_response
    "returned None instead." % (callback.__module__, view_name)
ValueError: The view toys.views.toy_list didn't return an HttpResponseRedirect
object. It returned None instead.
[14/Oct/2017 02:13:26] "OPTIONS /toys/ HTTP/1.1" 500 52222
```

The following lines show the header for the output displayed as a result of the HTTP request. The response also includes a detailed HTML document with a huge amount of information about the error because the debug mode is activated for Django. We receive an `HTTP 500 Internal Server Error` status code. Obviously, we don't want all this information to be provided in a production-ready web service, in which we will deactivate the debug mode:

```
HTTP/1.0 500 Internal Server Error
Content-Length: 52222
Content-Type: text/html
Date: Tue, 10 Oct 2017 17:46:34 GMT
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Cookie
X-Frame-Options: SAMEORIGIN
```

We don't want our web service to provide a response with an HTTP 500 Internal Server Error status code when we receive a request with the `OPTIONS` verb to either a valid resource or resource collection. Obviously, we want to provide a more consistent web service and we want to provide an accurate response when we receive a request with the `OPTIONS` verbs, for either a toy resource or the toys collection.

If we compose and send an HTTP request with the `OPTIONS` verb for an existing toy resource, we will see the same error in the console output for the development server and a similar response with the HTTP 500 Internal Server Error status code. The `views.toy_detail` function only runs code when the `request.method` is equal to `'GET'`, `'PUT'`, or `'DELETE'`. Thus, as happened with the previous case, the `toys_detail` function won't return an `HttpResponse` instance and it will return `None` instead.

The following commands will produce the explained error when we try to see the options offered for the toy resource whose `id` or primary key is equal to 2. Make sure you replace 2 with a primary key value of an existing toy in your configuration:

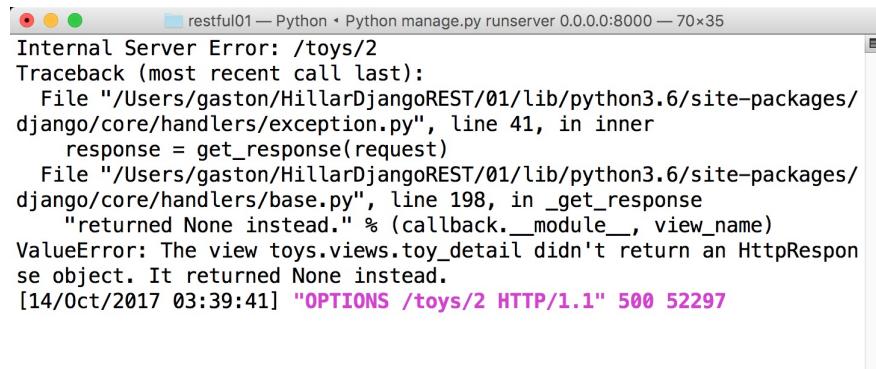
```
http OPTIONS :8000/toys/2
```

The following is the equivalent curl command:

```
curl -iX OPTIONS localhost:8000/toys/2
```

The following screenshot shows the details of the internal server error and

a traceback displayed in the console output for the development server after we run the previous HTTP request:



The screenshot shows a terminal window titled "restful01 — Python" with the command "Python manage.py runserver 0.0.0.0:8000" running. The window displays an Internal Server Error for the URL "/toys/2". The traceback points to the "django/core/handlers/exception.py" file at line 41, where it calls "get_response(request)". It then traces back to "django/core/handlers/base.py" at line 198, which calls "_get_response". The error message states: "returned None instead." % (callback.__module__, view_name). A ValueError is then raised: "The view toys.views.toy_detail didn't return an HttpResponse object. It returned None instead." The log entry shows the error occurred on [14/Oct/2017 03:39:41] with the status line "OPTIONS /toys/2 HTTP/1.1" 500 52297.

```
Internal Server Error: /toys/2
Traceback (most recent call last):
  File "/Users/gaston/HillarDjangoREST/01/lib/python3.6/site-packages/
django/core/handlers/exception.py", line 41, in inner
    response = get_response(request)
  File "/Users/gaston/HillarDjangoREST/01/lib/python3.6/site-packages/
django/core/handlers/base.py", line 198, in _get_response
    "returned None instead." % (callback.__module__, view_name)
ValueError: The view toys.views.toy_detail didn't return an HttpResponse
object. It returned None instead.
[14/Oct/2017 03:39:41] "OPTIONS /toys/2 HTTP/1.1" 500 52297
```

Understanding decorators that work as wrappers

Now, we will make a few changes to the code in the `toys/views.py` file to provide support for the `OPTIONS` verb in our RESTful Web Service. Specifically, we will take advantage of a decorator provided by the Django REST framework.

We will use the `@api_view` decorator that is declared in the `rest_framework.decorators` module. We will apply this decorator to our function-based views: `toys_list` and `toys_detail`.

The `@api_view` decorator allows us to specify which are the HTTP verbs that the function to which it is applied can process. If the request that has been routed to the view function has an HTTP verb that isn't included in the string list specified as the `http_method_names` argument for the `@api_view` decorator, the default behavior returns a response with an HTTP `405 Method Not Allowed` status code.

This way, we make sure that whenever the RESTful Web Service receives an HTTP verb that isn't considered within our function views, we won't generate an unexpected and undesired error in Django. The decorator generates the appropriate response for the unsupported HTTP verbs or methods. In addition, by reading the declaration of our function views, we can easily understand which HTTP verbs are handled by the function.

It is very important to understand what happens under the hood whenever we use the `@api_view` decorator. This decorator is a wrapper that converts a function-based view into a subclass of the `rest_framework.views.APIView` class. This class is the base class for all the views in the Django REST framework.

We will work with class-based views in the forthcoming examples and we will have the same benefits we have analyzed for the function-based views that use the decorator.

In addition, the decorator uses the string list we specify with the supported HTTP verbs to build the response for a request with the `OPTIONS` HTTP verb. The automatically generated response includes the supported method, and the parser and the render capabilities. In other words, the response includes the format that the function is capable of understanding and the format that the function can generate for the response.

As previously explained, the current version of our RESTful Web Service is only capable of rendering JSON as its output. The usage of the decorator makes sure that we always receive an instance of the `rest_framework.request.Request` class in the `request` argument when Django calls our view function. In addition, the decorator handles the `ParserError` exceptions when our function views access the `request.data` attribute and there are parsing problems.

Using decorators to enable different parsers and renderers

We will make changes to just one file. After you save the changes, Django's development server will automatically restart. However, you can decide to stop Django's development server and start it again after you finish all the necessary changes.

We will make the necessary changes to use the previously introduced `@api_view` decorator to make it possible for the RESTful Web Service to work with different parsers and renderers, by taking advantage of generalized behaviors provided by the `APIView` class.

Now, go to the `restful01/toys` folder and open the `views.py` file. Replace the code in this file with the following lines. However, take into account that many lines have been removed, such as the lines that declared the `JSONResponse` class. The code file for the sample is included in the `hillar_django_restful_04_02` folder, in the `restful01/toys/views.py` file:

```
from django.shortcuts import render
from rest_framework import status
from toys.models import Toy
from toys.serializers import ToySerializer
from rest_framework.decorators import api_view
from rest_framework.response import Response

@api_view(['GET', 'POST'])
def toy_list(request):
    if request.method == 'GET':
        toys = Toy.objects.all()
        toys_serializer = ToySerializer(toys, many=True)
        return Response(toys_serializer.data)

    elif request.method == 'POST':
        toy_serializer = ToySerializer(data=request.data)
```

```

        if toy_serializer.is_valid():
            toy_serializer.save()
            return Response(toy_serializer.data, status=status.HTTP_201_CREATED)
        return Response(toy_serializer.errors, status=status.HTTP_400_BAD_REQUEST)

@api_view(['GET', 'PUT', 'DELETE'])
def toy_detail(request, pk):
    try:
        toy = Toy.objects.get(pk=pk)
    except Toy.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    if request.method == 'GET':
        toy_serializer = ToySerializer(toy)
        return Response(toy_serializer.data)

    elif request.method == 'PUT':
        toy_serializer = ToySerializer(toy, data=request.data)
        if toy_serializer.is_valid():
            toy_serializer.save()
            return Response(toy_serializer.data)
        return Response(toy_serializer.errors, status=status.HTTP_400_BAD_REQUEST)

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

```

The new code applies the `@api_view` decorator for the two functions: `toy_list` and `toy_detail`. In addition, the new code removes the `JSONResponse` class and uses the more generic `rest_framework.response.Response` class.

We had to remove the usage of the `rest_framework.parsers.JSONParser` class in the functions to make it possible to work with different parsers. This way, we stopped working with a parser that only works with JSON. In the older version of the code, the `toy_list` function executed the following two lines when the `request.method` attribute was equal to 'POST':

```

toy_data = JSONParser().parse(request)
toy_serializer = ToySerializer(data=toy_data)

```

In the new code, we removed the first line that called the `JSONParser().parse`

method that was only capable of parsing JSON content. The new code replaces the two previous lines with the following single line that passes `request.data` as the `data` argument to create a new `ToySerializer` instance:

```
toy_serializer = ToySerializer(data=request.data)
```

In the older version of the code, the `toy_detail` function executed the following two lines when the `request.method` attribute was equal to '`PUT`':

```
toy_data = JSONParser().parse(request)
toy_serializer = ToySerializer(toy, data=toy_data)
```

We made edits that are similar to the changes done for the code in the `toy_list` function. We removed the first line that called the `JSONParser().parse` method that was only capable of parsing JSON content. The new code replaces the two previous lines with the following single line that passes `toy` as the first argument and `request.data` as the `data` argument to create a new `ToySerializer` instance:

```
toy_serializer = ToySerializer(toy, data=request.data)
```

Taking advantage of content negotiation classes

The `APIView` class defines default settings for each view that we can override by specifying the desired values in the settings module, that is, the `restful01/settings.py` file. It is also possible to override the class attributes in subclasses. In this case, we won't make changes in the settings module, but we have to understand which are the default settings that the `APIView` class uses. We added the `@api_view` decorator, and it automatically makes the `APIView` use these settings.

The value for the `DEFAULT_PARSER_CLASSES` setting key specifies a tuple of string whose values indicate the default classes that we want to use for parsing backends. The following lines show the default values:

```
(  
    'rest_framework.parsers.JSONParser',  
    'rest_framework.parsers.FormParser',  
    'rest_framework.parsers.MultiPartParser'  
)
```

When we use the `@api_view` decorator, the RESTful Web Service will be able to handle any of the following content types through the appropriate parsers. Thus, we will be able to work with the `request.data` attribute to retrieve the keys and values for each of these content types:

- `application/json`: Parsed by the `rest_framework.parsers.JSONParser` class
- `application/x-www-form-urlencoded`: Parsed by the `rest_framework.parsers.FormParser` class
- `multipart/form-data`: Parsed by the `rest_framework.parsers.MultiPartParser`

```
class
```

When we access the `request.data` attribute in the functions, the Django REST framework examines the value for the `Content-Type` header in the incoming request and determines the appropriate parser to parse the request content. If we use the previously explained default values, the Django REST Framework will be able to parse all of the previously listed content types. Notice that the request must specify the appropriate value for the `Content-Type` key in the request header.

The value for the `DEFAULT_RENDERER_CLASSES` setting key specifies a tuple of string whose values indicate the default classes that we want to use for rendering backends. The following lines show the default values:

```
(  
    'rest_framework.renderers.JSONRenderer',  
    'rest_framework.renderers.BrowsableAPIRenderer',  
)
```

When we use the `@api_view` decorator, the RESTful Web Service will be able to render any of the following content types through the appropriate renderers. We made the necessary changes to work with a `rest_framework.response.Response` instance to be able to work with these content types:

- `application/json`: Rendered by the `rest_framework.response.JSONRenderer` class
- `text/html`: Rendered by the `rest_framework.response.BrowsableAPIRenderer` class

So far, we understand the default settings for parsers and renderers. There is an additional part of this puzzle that must select the appropriate renderer for the response based on the requirements specified in the incoming

request.

By default, the value for the `DEFAULT_CONTENT_NEGOTIATION_CLASS` is the `rest_framework.negotiation.DefaultContentNegotiation` class. When we use the decorator, the web service will use this content negotiation class to select the appropriate renderer for the response, based on the incoming request. This way, when a request specifies that it will accept `text/html`, the content negotiation class selects the `rest_framework.renderers.BrowsableAPIRenderer` to render the response and generate `text/html` instead of `application/json`.

In the old version of the code, we used the `JSONResponse` and `HttpResponse` classes in the functions. The new version replaced the usages of both classes with the `rest_framework.response.Response` class. This way, the code takes advantage of the content negotiation features. The `Response` class renders the provided data into the appropriate content type and returns it to the client that made the request.

Making supported HTTP OPTIONS requests with command-line tools

Now, we will take advantage of all the changes we've made in the code and we will compose and send HTTP requests to make our RESTful Web Service work with different content types. Make sure you've saved all the changes. In case you stopped Django's development server, you will have to start it again as we learned in [Chapter 3, *Creating API Views*](#), in the section *Launching Django's development server*, to start running the Django development server.

We want to know which HTTP verbs the toys collection supports, that is, we want to take advantage of the `OPTIONS` verb. Run the following command. This time, the command won't produce errors. Remember that the virtual environment we have created in the previous chapters must be activated in order to run the next `http` command:

```
http OPTIONS :8000/toys/
```

The following is the equivalent `curl` command:

```
curl -iX OPTIONS localhost:8000/toys/
```

The previous command will compose and send the following HTTP request: `OPTIONS http://localhost:8000/toys/`. The request will end up running the `views.toy_list` function, that is, the `toy_list` function declared within the `toys/views.py` file. We added the `@api_view` decorator to this function, and therefore, the function is capable of determining the supported HTTP

verbs, the enabled parsing and rendering options. The following lines show the output:

```
HTTP/1.0 200 OK
Allow: POST, OPTIONS, GET
Content-Length: 167
Content-Type: application/json
Date: Mon, 16 Oct 2017 04:28:32 GMT
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
    "description": "",
    "name": "Toy List",
    "parses": [
        "application/json",
        "application/x-www-form-urlencoded",
        "multipart/form-data"
    ],
    "renders": [
        "application/json",
        "text/html"
    ]
}
```

The response header includes an `Allow` key with a comma-separated list of HTTP verbs supported by the resource collection as its value: `POST, OPTIONS, GET`. Our request didn't specify the allowed content type, and therefore, the function rendered the response with the default `application/json` content type.

The response body specifies the `Content-type` that the resource collection is capable of parsing in the values for the `"parses"` key and the `Content-type` that the resource collection is capable of rendering in the values for the `"renders"` key.

Run the following command to compose and send an HTTP request with the `OPTIONS` verb for a toy resource. Don't forget to replace `2` with a primary key value of an existing toy in your configuration:

```
http OPTIONS :8000/toys/2
```

The following is the equivalent curl command:

```
curl -iX OPTIONS localhost:8000/toys/2
```

The previous command will compose and send the following HTTP request: `OPTIONS http://localhost:8000/toys/2`. The request will end up running the `views.toy_detail` function, that is, the `toy_detail` function declared within the `toys/views.py` file. We also added the `@api_view` decorator to this function, and therefore, it is capable of determining the supported HTTP verbs, the enabled parsing and rendering options. The following lines show a sample output:

```
HTTP/1.0 200 OK
Allow: DELETE, PUT, OPTIONS, GET
Content-Length: 169
Content-Type: application/json
Date: Mon, 16 Oct 2017 04:30:04 GMT
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
    "description": "",
    "name": "Toy Detail",
    "parses": [
        "application/json",
        "application/x-www-form-urlencoded",
        "multipart/form-data"
    ],
    "renders": [
        "application/json",
        "text/html"
    ]
}
```

The response header includes an `Allow` key with a comma-separated list of

HTTP verbs supported by the resource as its value: `DELETE`, `PUT`, `OPTIONS`, `GET`. The response body specifies the `content-type` that the resource is capable of parsing in the values for the `"parses"` key and the `content-type` that the resource collection is capable of rendering in the values for the `"renders"` key. The resource and the resource collection can parse and render the same content types because everything is handled by the decorator and the `APIView` class.

Working with different content types

In [Chapter 3](#), *Creating API Views*, when we composed and sent `POST` and `PUT` commands, we had to use the `-H "Content-Type: application/json"` option to indicate curl to send the data specified after the `-d` option as `application/json`. We had to use this option because the default content-type in curl is `application/x-www-form-urlencoded`.

Now, our RESTful Web Service goes beyond JSON and it can also parse `application/x-www-form-urlencoded` and `multipart/form-data` data specified in the `POST` and `PUT` requests. Hence, we can compose and send a `POST` command that sends the data as `application/x-www-form-urlencoded`.

We will compose and send an HTTP request to create a new toy. In this case, we will use the `-f` option for HTTP.

This option serializes data items from the command line as form fields and sets the `Content-Type` header key to the `application/x-www-form-urlencoded` value. Run the next command:

```
http -f POST :8000/toys/ name="Ken in Rome" description="Ken loves Rome" toy_cat
```

The following is the equivalent curl command that creates a new toy. Notice that we don't use the `-H` option and curl will send the data in the default `application/x-www-form-urlencoded`:

```
curl -iX POST -d '{"name": "Ken in Rome", "description": "Ken loves Rome", "toy_cat": "cat"}'
```

The previous commands will compose and send the following HTTP

request: POST http://localhost:8000/toys/ with the Content-Type header key set to the application/x-www-form-urlencoded value and the following data:

```
name=Ken+in+Rome&description=Ken+loves+Rome&toy_category=Dolls&was_included_in_h
```

The request specifies /toys/, and therefore, it will match the '^toys/\$' regular expression and Django will run the `views.toy_list` function, that is, the updated `toy_detail` function declared within the `toys/views.py` file. The HTTP verb for the request is `POST`, and therefore, the `request.method` property is equal to '`POST`'. The function will execute the code that creates a `ToySerializer` instance and passes `request.data` as the `data` argument to create the new instance.

The `rest_framework.parsers.FormParser` class will parse the data received in the request, the code creates a new `Toy` and, if the data is valid, it saves the new `Toy`. If the new `Toy` instance was successfully persisted in the database, the function returns an HTTP `201 Created` status code and the recently persisted `Toy` serialized to JSON in the response body. The following lines show an example response for the HTTP request, with the new `Toy` object in the JSON response:

```
HTTP/1.0 201 Created
Allow: GET, OPTIONS, POST
Content-Length: 157
Content-Type: application/json
Date: Mon, 16 Oct 2017 04:40:02 GMT
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN
{
    "description": "Ken loves Rome",
    "id": 6,
    "name": "Ken in Rome",
    "release_date": "2017-10-09T12:11:37.090335Z",
    "toy_category": "Dolls",
    "was_included_in_home": false
}
```

Sending HTTP requests with unsupported HTTP verbs

Now, we will compose and send HTTP requests with an HTTP verb that isn't supported for the toys resource collection. Run the following command:

```
http PATCH :8000/toys/
```



The following is the equivalent `curl` command:

```
curl -iX PATCH localhost:8000/toys/
```



The previous command will compose and send the following HTTP request: `PATCH http://localhost:8000/toys/`. The request will try to run the `views.toy_list` function, that is, the `toy_list` function declared within the `toys/views.py` file. The `@api_view` decorator we added to this function doesn't include '`PATCH`' in the string list with the allowed HTTP verbs. The default behavior when this happens in the `APIView` class is to return an HTTP 405 Method Not Allowed status code. The following lines show a sample output with the response from the previous request. A JSON content provides a `detail` key with a string value that indicates the `PATCH` method is not allowed in the response body:

```
HTTP/1.0 405 Method Not Allowed
Allow: GET, OPTIONS, POST
Content-Length: 42
Content-Type: application/json
Date: Mon, 16 Oct 2017 04:41:35 GMT
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Accept, Cookie
```

```
X-Frame-Options: SAMEORIGIN

{
    "detail": "Method \"PATCH\" not allowed."
}
```



Test your knowledge

Let's see whether you can answer the following questions correctly:

1. The `@api_view` decorator declared in the `rest_framework.decorators` module allows you to:
 1. Specify which is the model related to the function based view
 2. Specify which are the HTTP verbs that the function based view to which it is applied can process
 3. Specify which is the serializer related to the function based view
2. The `@api_view` decorator is a wrapper that converts a function based view into a subclass of which of the following classes:
 1. `django.Http.Response.HttpResponse`
 2. `rest_framework.views.APIView`
 3. `rest_framework.serializers.Serializer`
3. Which of the following settings key in the `REST_FRAMEWORK` dictionary allows you to override the global setting with a tuple of string whose values indicate the default classes that you want to use for parsing backends:

1. 'DEFAULT_PARSER_CLASSES'
 2. 'GLOBAL_PARSER_CLASSES'
 3. 'REST_FRAMEWORK_PARSING_CLASSES'
-
4. Which of the following classes is able to parse application/json content type when we work with the `@api_view` decorator and its default settings:
 1. `django.parsers.JSONParser`
 2. `rest_framework.classes.JSONParser`
 3. `rest_framework.parsers.JSONParser`

 5. Which of the following classes is able to parse application/x-www-form-urlencoded content type when we work with the `@api_view` decorator and its default settings:
 1. `django.parsers.XWWWUrlEncodedParser`
 2. `rest_framework.classes.XWWWUrlEncodedParser`
 3. `rest_framework.parsers.FormParser`

The rights answers are included in the [Appendix](#), *Solutions*.

Summary

In this chapter, we improved our simple Django RESTful Web Service. We took advantage of many features included in the Django REST framework to remove duplicate code and to add many features for the web service. We just needed to edit a few lines of code to enable an important amount of features.

First, we took advantage of model serializers. Then, we understood the different accepted and returned content types and the importance of providing accurate responses to the HTTP OPTIONS requests.

We incorporated the `@api_view` decorator and made the necessary changes to the existing code to enable diverse parsers and renderers. We understood how things worked under the hood in the Django REST framework. We worked with different content types and noticed the improvement of the RESTful Web Service compared with its previous versions.

Now that we understand how easy it is to work with different content types with the Django REST framework, we will work with one of the most interesting and powerful features: the browsable API. We will cover this topic in [chapter 5, *Understanding and Customizing the Browsable API Feature*](#).

Understanding and Customizing the Browsable API Feature

In this chapter, we will work with one of the most interesting and powerful features included in the Django REST framework: the browsable API. This feature makes it easy for us to interact with our RESTful Web Services through any web browser. We will gain an understanding of:

- Understanding the possibility of rendering text/HTML content
- Using a web browser to work with our web service
- Making HTTP GET requests with the browsable API
- Making HTTP POST requests with the browsable API
- Making HTTP PUT requests with the browsable API
- Making HTTP DELETE requests with the browsable API
- Making HTTP OPTIONS requests with the browsable API

Understanding the possibility of rendering text/HTML content

In [Chapter 4, Using Generalized Behavior from the APIView Class](#), we made many changes to make it possible for the simple RESTful Web Service to work with a content negotiation class and provide many content renderers. We used the default configuration for the Django REST framework that includes a renderer that produces `text/html` content.

The `rest_framework.response.BrowsableAPIRenderer` class is responsible for rendering the `text/html` content. This class makes it possible for us to browse the API. The Django REST framework includes a feature that generates an interactive and human-friendly HTML output for the different resources when the request specifies `text/html` as the value for the `Content-Type` key in the request header. This feature is known as the browsable API because it enables us to use a web browser to navigate through the API and easily make different types of HTTP requests.

The browsable API feature is extremely useful when we have to test the RESTful Web Services that perform CRUD operations on a database, such as the one we have been developing in [chapter 4, Using Generalized Behavior from the APIView Class](#).

Now, we will compose and send HTTP requests that will make the RESTful Web Service user the `BrowsableAPIRenderer` class to provide `text/html` content in the response. This way, we will understand how the browsable API works before we jump into the web browser and we start using and customizing this feature. In case you stopped Django's development server, you will have to start it again as we learned in [Chapter 3, Creating API Views](#), in the section *Launching Django's development server*, to start running the Django development server.

Run the following command to retrieve all the toys with the `Accept` request header key set to `text/html`. Remember that the virtual environment we created in the previous chapters must be activated in order to run the next `http` command:

```
http -v :8000/toys/ "Accept:text/html"
```

The following is the equivalent `curl` command:

```
curl -vH "Accept: text/html" -iX GET localhost:8000/toys/
```

The previous commands will compose and send the following HTTP request: `GET http://localhost:8000/toys/`. These commands specify the `text/html` value for the `Accept` key in the request header. This way, the HTTP request indicates that it accepts a response of `text/html`.

In both cases, we specified the `-v` option that provides a verbose output and prints the details of the request that has been made. For example, the following are the first lines of the output generated by the `http` command:

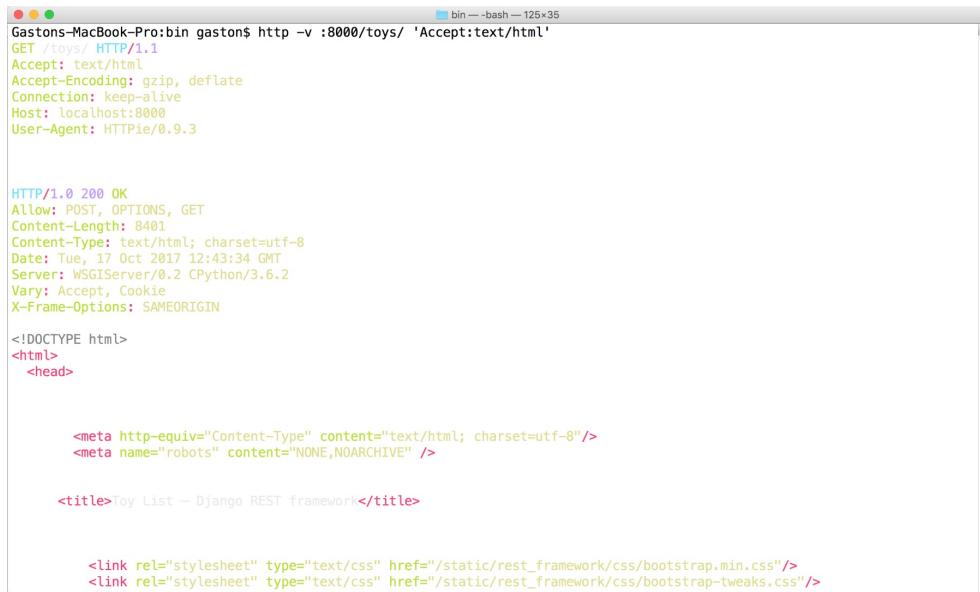
```
GET /toys/ HTTP/1.1
Accept: text/html
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8000
User-Agent: HTTPie/0.9.3
```

The second line prints the value for the `Accept` key included in the request header, `text/html`. The header response for the request will include the following line:

```
Content-Type: text/html; charset=utf-8
```

The previous commands will compose and send the following HTTP

`request: GET http://localhost:8000/toys/`. The request will end up running the `views.toy_list` function, that is, the `toy_list` function declared within the `toys/views.py` file. The content negotiation class selected the `BrowsableAPIRenderer` class to provide `text/html` content in the response. The following lines show the first lines of the output for the `http` command:



```
Gastons-MacBook-Pro:bin gaston$ http -v :8000/toys/ 'Accept:text/html'
GET /toys/ HTTP/1.1
Accept: text/html
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8000
User-Agent: HTTPie/0.9.3

HTTP/1.0 200 OK
Allow: POST, OPTIONS, GET
Content-Length: 8401
Content-Type: text/html; charset=utf-8
Date: Tue, 17 Oct 2017 12:43:34 GMT
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

<!DOCTYPE html>
<html>
  <head>

    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <meta name="robots" content="NONE,NODRAWNE" />

    <title>Toy List - Django REST framework</title>

    <link rel="stylesheet" type="text/css" href="/static/rest_framework/css/bootstrap.min.css"/>
    <link rel="stylesheet" type="text/css" href="/static/rest_framework/css/bootstrap-tweaks.css"/>

  </head>
  <body>

    <h1>Django REST framework</h1>
    <h2>Toy List</h2>
    <table>
      <thead>
        <tr>
          <th>Name</th>
          <th>Type</th>
          <th>Description</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>Plush</td>
          <td>Stuffed Animal</td>
          <td>A soft toy made of plush material, often shaped like an animal or character. They are typically used for comfort or play. Plush toys are often collectible and can be quite expensive.</td>
        </tr>
        <tr>
          <td>Action Figure</td>
          <td>Collectible Toy</td>
          <td>A figure representing a character from a movie, television show, or video game. Action figures are usually made of plastic and have articulated joints, allowing them to be posed in different ways. They are often highly detailed and come with accessories like weapons or vehicles.</td>
        </tr>
        <tr>
          <td>Board Game</td>
          <td>Game</td>
          <td>A game played on a board with pieces and tokens. It requires strategy and luck to win. Board games are often played with other people and can be a social activity. Some popular board games include Monopoly, Clue, and Scrabble.</td>
        </tr>
        <tr>
          <td>Card Game</td>
          <td>Game</td>
          <td>A game played with cards. It can be a solo game or a game played with other people. Card games can be simple or complex, and some require skill while others are more luck-based. Some popular card games include poker, blackjack, and Uno.</td>
        </tr>
        <tr>
          <td>Video Game</td>
          <td>Game</td>
          <td>A game played on a computer or console. It can be a single-player game or a multi-player game. Video games can be action-oriented, puzzle-solving, or simulation-based. Some popular video games include Minecraft, Call of Duty, and Grand Theft Auto V.</td>
        </tr>
      </tbody>
    </table>
    <div>
      <a href="#">Create New Toy</a>
    </div>
  </body>
</html>
```

We can easily detect from the previous output that the Django REST framework provides an HTML web page as a response to our previous requests. If we enter any URL for a resource collection or resource in any web browser, the browser will perform an HTTP GET request that requires an HTML response, that is, the `Accept` request header key will be set to `text/html`. The web service built with the Django REST framework will provide an HTML response and the browser will render the web page.

By default, the `BrowsableAPIRenderer` class uses the Bootstrap popular frontend component library. You can read more about Bootstrap here: <http://getbootstrap.com>. The web page might include the following elements:

- Diverse buttons to perform other requests to the resource or resource collection
- A section that displays the resource or resource collection content in JSON

- Forms with fields that allow us to submit data for `POST`, `PUT`, and `PATCH` requests

The Django REST framework uses templates and themes to render the pages for the browsable API. It is possible to customize many settings to tailor the output to our specific requirements.

Using a web browser to work with our web service

Let's start browsing our RESTful Web Service. Open a web browser and enter `http://localhost:8000/toys/`. The browser will compose and send a `GET` request to `http://localhost:8000/toys/` with `text/html` as the desired content type and the returned HTML web page will be rendered.

Under the hood, the web service will compose and send an HTTP GET request to `http://localhost:8000/toys/` with `application/json` as the content type and the headers, and the JSON returned by this request will be rendered as part of the content of the web page. The following screenshot shows the rendered web page with the resource collection description, Toy List:

The screenshot shows a web browser displaying the Toy List API response. The title bar says "Django REST framework". The main content area has a header "Toy List" with "OPTIONS" and "GET" buttons. Below that is a "GET /toys/" button. The response body shows the following JSON data:

```
HTTP 200 OK
Allow: POST, OPTIONS, GET
Content-Type: application/json
Vary: Accept

[{"id": 3, "name": "Clash Royale play set", "description": "6 figures from Clash Royale", "release_date": "2017-10-09T12:10:00.776594Z", "toy_category": "Playset", "was_included_in_home": false}, {"id": 2, "name": "Hawaiian Barbie", "description": "Barbie loves Hawaii", "release_date": "2017-10-09T12:11:37.090335Z", "toy_category": "Dolls", "was_included_in_home": true}, {"id": 6, "name": "Ken in Rome", "description": null, "release_date": null, "toy_category": null, "was_included_in_home": null}]
```

When we work with the browsable API, Django uses the information

about the allowed methods for a resource or resource collection to render the appropriate buttons to allow us to execute the related requests. In the previous screenshot, you will notice that there are two buttons on the right-hand side of the resource description (Toy List): OPTIONS and GET. We will use the different buttons to make additional requests to the RESTful Web Service.

If you decide to browse the web service in a web browser that is being executed on another computer or device connected to the LAN, you will have to use the assigned IP address to the computer that is running Django's development server instead of `localhost`. For example, if Django's development server is running on a computer whose assigned IPv4 IP address is `192.168.2.125`, instead of `http://localhost:8000/toys/`, you should use `http://192.168.2.125:8000/toys/`. You can also use the hostname instead of the IPv4 address or an IPv6 address.

One of the nicest features of the browsable API is that it makes it extremely easy to test a RESTful Web Service from a mobile device.

As a disclaimer, I must say that once you learn how to take advantage of the browsable API, you will never want to work with a framework that doesn't provide a feature like this one.

Making HTTP GET requests with the browsable API

We just made an HTTP GET request to retrieve the toys resource collection with the browsable API. Now, we will compose and send another HTTP GET request for an existing toy resource with the web browser.

Enter the URL for an existing toy resource, such as

`http://localhost:8000/toys/3`. Make sure you replace `3` with the `id` or primary key of an existing toy in the previously rendered Toy List. Django will compose and send a `GET` request to `http://localhost:8000/toys/3` and the rendered web page will display the results of its execution, that is, the headers and the JSON data for the toy resource. The following screenshot shows the rendered web page after entering the URL in a web browser with the resource description, Toy Detail:

The screenshot shows the Django REST framework browsable API interface. At the top, there's a header bar with the text "Django REST framework". Below it, a navigation bar shows "Toy List" and "Toy Detail". The main content area is titled "Toy Detail" and has a URL field containing "GET /toys/3". To the right of the URL field are three buttons: "DELETE" (red), "OPTIONS" (blue), and "GET" (blue). Below the URL field, there's a status message "HTTP 200 OK" followed by the standard HTTP headers: "Allow: PUT, DELETE, OPTIONS, GET", "Content-Type: application/json", and "Vary: Accept". The main content area displays a JSON object representing a toy resource:

```
{  
    "id": 3,  
    "name": "Clash Royale play set",  
    "description": "6 figures from Clash Royale",  
    "release_date": "2017-10-09T12:10:00.776594Z",  
    "toy_category": "Playset",  
    "was_included_in_home": false  
}
```

At the bottom of the page, there's a form for editing the resource. It has a "Media type:" dropdown set to "application/json" and a "Content:" text area. A "PUT" button is located at the bottom right of the form.

At the right-hand side of the resource description, the browsable API

shows a GET drop-down button. This button allows us to make a `GET` request to `/toys/3` again. If we click or tap the down arrow, we can select the json option and the browsable API will display the raw JSON results of a `GET` request to `/toys/3` without the headers. In fact, the browser will go to `http://localhost:8000/toys/3?format=json` and the Django REST framework will display the raw JSON results because the value for the `format` query parameter is set to `json`. The following screenshot shows the results of making that request:



```
{"id":3,"name":"Clash Royale play set","description":"6 figures from Clash Royale","release_date":"2017-10-09T12:10:00.776594Z","toy_category":"Playset","was_included_in_home":false}
```

Enter the URL for a non-existing toy resource, such as

`http://localhost:8000/toys/250`. Make sure you replace `250` with the `id` or primary key of the toy that doesn't exist in the previously rendered Toy List. Django will compose and send a `GET` request to

`http://localhost:8000/toys/250` and the rendered web page will display the results of its execution, that is, the header with the `HTTP 404 Not found` status code.

The following screenshot shows the rendered web page after entering the URL in a web browser:

Toy List / Toy Detail

Toy Detail

[DELETE](#)

[OPTIONS](#)

[GET](#) ▾

[GET /toys/250](#)

HTTP 404 Not Found
Allow: PUT, DELETE, OPTIONS, GET
Content-Type: application/json
Vary: Accept

Media type:

application/json

Content:

[PUT](#)

Making HTTP POST requests with the browsable API

Now, we want to use the browsable API to compose and send an HTTP POST request to our RESTful Web Service to create a new toy. Go to the following URL in your web browser, `http://localhost:8000/toys/`. At the bottom of the rendered web page, the browsable API displays the following controls to allow us to compose and send a `POST` request to `/toys/`:

- Media type: This dropdown allows us to select the desired parser. The list will be generated based on the configured supported parsers in the Django REST framework for our web service.
- Content: This text area allows us to enter the text for the body that will be sent with the POST request. The content must be compatible with the selected value for the media type dropdown.
- POST: This button will use the selected media type and the entered content to compose and send an HTTP POST request with the appropriate header key/value pairs and content.

The following screenshot shows the previously explained controls at the bottom of the rendered web page:

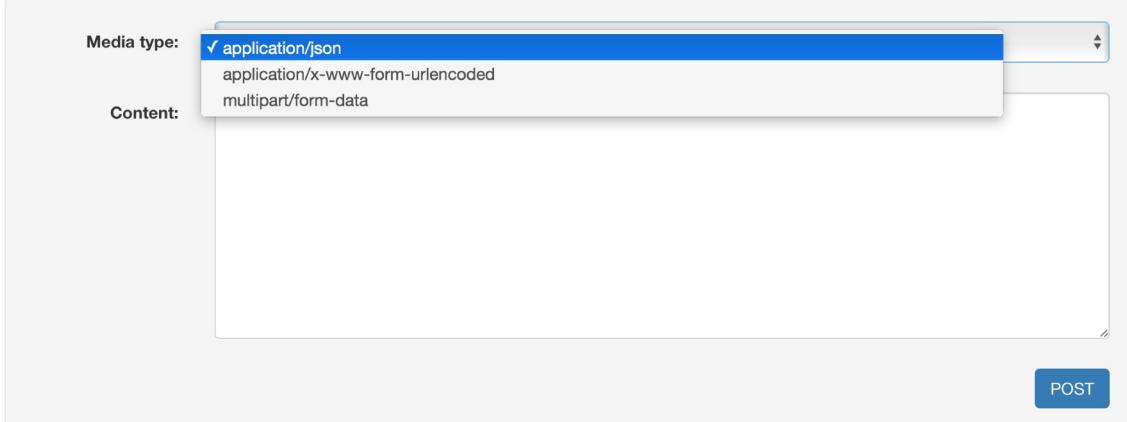
Django REST framework

```
        "was_included_in_home": false
    },
{
    "id": 5,
    "name": "Wonderboy puzzle",
    "description": "The Dragon's Trap puzzle",
    "release_date": "2017-10-03T01:01:00.776594Z",
    "toy_category": "Puzzles & Games",
    "was_included_in_home": false
}
]
```

Media type: **application/json**

Content:

POST



We enabled all the default parsers provided by the Django REST framework for our RESTful Web Service, and therefore, the Media type drop-down will provide us with the following options:

- application/json
- application/x-www-form-urlencoded
- multipart/form-data

Select application/json in the Media type dropdown and enter the following JSON content in the Content text area:

```
{
    "name": "Surfer girl",
    "description": "Surfer girl doll",
    "toy_category": "Dolls",
    "was_included_in_home": "false",
    "release_date": "2017-10-29T12:11:25.090335Z"
}
```





Click or tap POST. The browsable API will compose and send an HTTP POST request to `/toys/` with the previously specified data as a JSON body, and we will see the results of the call in the web browser.

The following screenshot shows a web browser displaying the HTTP status code `201 Created` in the response and the previously explained dropdown and text area with the POST button to allow us to continue composing and sending POST requests to `/toys/`:

A screenshot of a web browser window. The title bar says "Toy List". The main content area displays the response to a POST request to "/toys/". It shows the HTTP status code "HTTP 201 Created" and the response headers "Allow: OPTIONS, POST, GET", "Content-Type: application/json", and "Vary: Accept". The response body is a JSON object representing a toy: {"id": 7, "name": "Surfer girl", "description": "Surfer girl doll", "release_date": "2017-10-29T12:11:25.090335Z", "toy_category": "Dolls", "was_included_in_home": false}. Below this, there is a form for composing a new POST request. It has a "Media type:" dropdown set to "application/json" and a "Content:" text area. A "POST" button is located at the bottom right of the form.

In this case, we entered the JSON key/value pairs as we did when we composed and sent HTTP POST requests with command-line and GUI tools. However, we will learn to configure the browsable API to provide us with a form with fields to make it even easier to perform operations on our RESTful Web Service.

Making HTTP PUT requests with the browsable API

Now, we want to use the browsable API to compose and send an HTTP PUT request to our RESTful Web Service to replace an existing toy with a new one. First, go to the URL for an existing toy resource, such as

`http://localhost:8000/toys/7`. Make sure you replace `7` with the `id` or primary key of an existing toy in the previously rendered Toy List. The HTML web page that displays the results of an HTTP GET request to `/toys/7` plus additional details and controls will be rendered.

At the bottom of the rendered web page, the browsable API displays the controls to compose and send a `POST` request to `/toys/` followed by the controls to compose and send a `PUT` request to `/toys/7`. The controls for the `PUT` request are the same that we already analyzed for the `POST` request. The `PUT` button will use the selected media type and the entered content to compose and send an HTTP PUT request with the appropriate header key/value pairs and content.

The following screenshot shows the controls to compose and send an HTTP PUT request at the bottom of the rendered web page:

Toy List / Toy Detail

Toy Detail

DELETE **OPTIONS** **GET** ▾

GET /toys/7

```

HTTP 200 OK
Allow: DELETE, OPTIONS, GET, PUT
Content-Type: application/json
Vary: Accept

{
    "id": 7,
    "name": "Surfer girl",
    "description": "Surfer girl doll",
    "release_date": "2017-10-29T12:11:25.090335Z",
    "toy_category": "Dolls",
    "was_included_in_home": false
}
  
```

Media type: application/json

Content:

PUT

In this example, we took advantage of the features included in the Django REST framework to build the OPTIONS response that indicates which HTTP verbs are allowed for each resource and resource collection. Thus, the browsable API only offers us the possibility to compose and send a POST and PUT methods. The POST method is applied to the resource collection while the PUT method is applied to a single resource. The browsable API doesn't provide the controls to compose and send an HTTP PATCH method on a resource because the code hasn't specified that this verb is accepted as a resource.

Select application/json in the Media type dropdown and enter the following JSON content in the Content text area. Remember that the HTTP PUT method replaces an existing resource with a new one, and

therefore, we must specify the values for all the fields and not just for the fields that we want to update:

```
{  
    "name": "Surfer girl",  
    "description": "Surfer girl doll (includes pink surfboard)",  
    "toy_category": "Dolls",  
    "was_included_in_home": "false",  
    "release_date": "2017-10-29T12:11:25.090335Z"  
}
```

Click or tap PUT. The browsable API will compose and send an HTTP `PUT` request to `/toys/7` with the previously specified data as a JSON body and we will see the results of the call in the web browser. The following screenshot shows a web browser displaying the HTTP status code `200 OK` in the response, and the controls to allow us to send a new PUT request, if necessary:

Toy List / Toy Detail

Toy Detail

DELETE

OPTIONS

GET

PUT /toys/7

HTTP 200 OK

Allow: DELETE, OPTIONS, GET, PUT
Content-Type: application/json
Vary: Accept

```
{  
    "id": 7,  
    "name": "Surfer girl",  
    "description": "Surfer girl doll (includes pink surfboard)",  
    "release_date": "2017-10-29T12:11:25.090335Z",  
    "toy_category": "Dolls",  
    "was_included_in_home": false  
}
```

Media type: application/json

Content:

PUT

Making HTTP OPTIONS requests with the browsable API

Now, we want to use the browsable API to compose and send an HTTP OPTIONS request to our RESTful Web Service to check the allowed HTTP verbs, the available renderers, and parsers for a toy resource. First, go to the URL for an existing toy resource, such as

`http://localhost:8000/toys/7`. Make sure you replace `7` with the `id` or primary key of an existing toy in the previously rendered Toy List. The HTML web page that displays the results of an HTTP GET request to `/toys/7` plus additional details and controls will be rendered.

At the right-hand side of the Toy Detail title, you will see an OPTIONS button. Click or tap this button. The browsable API will compose and send an HTTP `OPTIONS` request to `/toys/7` and we will see the results of the call in the web browser. The following screenshot shows a web browser displaying the HTTP status code `200 OK` in the response, the allowed HTTP verbs, the content types that the toy resource is capable of rendering as values for the `renders` key, and the content types that the toy resource is capable of parsing as values for the `parses` key:

The screenshot shows a web browser window with the title "Toy Detail". At the top right are three buttons: "DELETE" (red), "OPTIONS" (blue), and "GET" (blue with a dropdown arrow). Below the title is a header bar with the URL "OPTIONS /toys/7". The main content area displays an HTTP response:

```
HTTP 200 OK
Allow: DELETE, OPTIONS, GET, PUT
Content-Type: application/json
Vary: Accept

{
  "name": "Toy Detail",
  "description": "",
  "renders": [
    "application/json",
    "text/html"
  ],
  "parses": [
    "application/json",
    "application/x-www-form-urlencoded",
    "multipart/form-data"
  ]
}
```

We can also compose and send an HTTP OPTIONS request to our RESTful Web Service to check the allowed HTTP verbs, the available renderers, and parsers for the toys resource collection. First, go to the URL for the toys resource collection: `http://localhost:8000/toys/`. The HTML web page that displays the results of an HTTP GET request to `/toys/`, plus additional details and controls, will be rendered.

At the right-hand side of the Toy Detail title, you will see an OPTIONS button. Click or tap this button. The browsable API will compose and send an HTTP `OPTIONS` request to `/toys/` with the previously specified data as a JSON body and we will see the results of the call in the web browser. The following screenshot shows a web browser displaying the HTTP status code `200 OK` in the response, the allowed HTTP verbs, the content types that the toys resource collection is capable of rendering as values for the `renders` key, and the content types that the toys resource collection is capable of parsing as values for the `parses` key:

The screenshot shows a browsable API interface with the following details:

- Resource:** Toy List
- HTTP Method:** OPTIONS /toys/
- Response Headers:**
 - HTTP 200 OK
 - Allow: OPTIONS, POST, GET
 - Content-Type: application/json
 - Vary: Accept
- Response Body (JSON):**

```
{  
  "name": "Toy List",  
  "description": "",  
  "renders": [  
    "application/json",  
    "text/html"  
  ],  
  "parses": [  
    "application/json",  
    "application/x-www-form-urlencoded",  
    "multipart/form-data"  
  ]  
}
```

It is always a good idea to check that all the allowed verbs returned by an HTTP OPTIONS request to a specific resource or resource collection are coded. The browsable API makes it easy for us to test whether the requests for all the supported verbs are working OK. Then, we can automate testing, which is a topic we will learn in the forthcoming chapters.

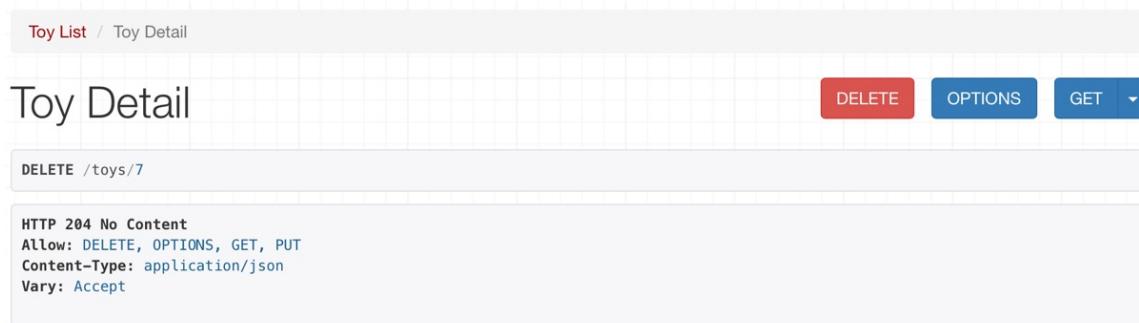
Making HTTP DELETE requests with the browsable API

Now, we want to use the browsable API to compose and send an HTTP DELETE request to our RESTful Web Service to delete an existing toy resource. First, go to the URL for an existing toy resource, such as

`http://localhost:8000/toys/7`. Make sure you replace `7` with the `id` or primary key of an existing toy in the previously rendered Toy List. The HTML web page that displays the results of an HTTP GET request to `/toys/7`, plus additional details and controls, will be rendered.

At the right-hand side of the Toy Detail title, you will see a DELETE button. Click or tap this button. The web page will display a modal requesting confirmation to delete the toy resource. Click or tap the DELETE button in this modal.

The browsable API will compose and send an HTTP `DELETE` request to `/toys/7` and we will see the results of the call in the web browser. The following screenshot shows a web browser displaying the HTTP status code `204 No Content` in the response:



Now, go to the URL for the toys resource collection:

`http://localhost:8000/toys/`. The HTML web page that displays the results of an HTTP GET request to `/toys/` plus additional details and controls will be

rendered. The recently deleted toy has been removed from the database. Thus, the list will not include the deleted toy. The following screenshot shows a web browser displaying the HTTP status code `200 OK` in the response and the list of toys without the recently deleted toy:

```
GET /toys/  
  
HTTP 200 OK  
Allow: OPTIONS, POST, GET  
Content-Type: application/json  
Vary: Accept  
  
[  
  {  
    "id": 3,  
    "name": "Clash Royale play set",  
    "description": "6 figures from Clash Royale",  
    "release_date": "2017-10-09T12:10:00.776594Z",  
    "toy_category": "Playset",  
    "was_included_in_home": false  
  },  
  {  
    "id": 2,  
    "name": "Hawaiian Barbie",  
    "description": "Barbie loves Hawaii",  
    "release_date": "2017-10-09T12:11:37.090335Z",  
    "toy_category": "Dolls",  
    "was_included_in_home": true  
  },  
  {  
    "id": 6,  
    "name": "Ken in Rome",  
    "description": "Ken loves Rome",  
    "release_date": "2017-10-09T12:11:37.090335Z",  
    "toy_category": "Dolls",  
    "was_included_in_home": false  
  },  
  {  
    "id": 1,  
    "name": "Snoopy talking action figure",  
    "description": "Snoopy speaks five languages",  
    "release_date": "2017-10-09T12:11:37.090335Z",  
    "toy_category": "Action figures",  
    "was_included_in_home": false  
  },  
  {  
    "id": 5,  
    "name": "Wonderboy puzzle",  
    "description": "The Dragon's Trap puzzle",  
    "release_date": "2017-10-03T01:01:00.776594Z",  
    "toy_category": "Puzzles & Games",  
    "was_included_in_home": false  
  }]  
]
```

The browsable API allowed us to compose and send many HTTP requests to our web service by clicking or tapping buttons on a web browser. We could check that all the operations are working as expected in our RESTful Web Service. However, we had to enter JSON content and we couldn't click on hyperlinks to navigate through entities. For example, we couldn't click on a toy's `id` to perform an HTTP GET request to retrieve this specific toy.

We will definitely improve this situation and we will take full advantage

of many additional features included in the browsable API as we create additional RESTful Web Services. We will do this in the forthcoming chapters. We have just started working with the browsable API.

Test your knowledge

Let's see whether you can answer the following questions correctly:

1. Which of the following classes is responsible for rendering the `text/html` content:
 1. The `rest_framework.response.HtmlRenderer` class
 2. The `rest_framework.response.TextHtmlAPIRenderer` class
 3. The `rest_framework.response.BrowsableAPIRenderer` class
2. By default, the browsable API uses the following web component library:
 1. Bootstrap
 2. ReactJS
 3. AngularJS
3. When we enter the URL of an existing resource in a web browser, the browsable API:
 1. Returns a web page with just the JSON response for an HTTP GET request to the resource
 2. Returns a web page with a section that displays the JSON response for an HTTP GET request to the resource and diverse buttons to perform other requests to the resource

3. Returns a web page with a section that displays the JSON response for an HTTP OPTIONS request to the resource and diverse buttons to perform other requests to the resource
4. When we enter the URL of a non-existing resource in a web browser, the browsable API:
 1. Renders a web page that displays an `HTTP 404 not found` header
 2. Displays a plain text message with an `HTTP 404 not found` error
 3. Renders a web page with the last toy resource that was available
5. If we enter the following URL, `http://localhost:8000/toys/10?format=json`, and there is a toy resource whose `id` is equal to `10`, the browsable API will display:
 1. The raw JSON results of an HTTP GET request to
`http://localhost:8000/toys/`
 2. The raw JSON results of an HTTP GET request to
`http://localhost:8000/toys/10`
 3. The same web page that would be rendered if we entered
`http://localhost:8000/toys/10`

The rights answers are included in the Appendix, *Solutions*.

Summary

In this chapter, we understood some of the additional features that the Django REST framework adds to our RESTful Web Service, the browsable API. We used a web browser to work with our first web service built with Django.

We learned to make HTTP GET, POST, PUT, OPTIONS, and DELETE requests with the browsable API. We were able to easily test CRUD operations with a web browser. The browsable API allowed us to easily interact with our RESTful Web Service. We will take advantage of additional features in the forthcoming chapters.

Now that we understand how easy it is to take advantage of the browsable API with the Django REST framework, we will move on to more advanced scenarios and we will start a new RESTful Web Service. We will work with advanced relationships and serialization. We will cover these topics in [Chapter 6, *Working with Advanced Relationships and Serialization*](#).

Working with Advanced Relationships and Serialization

In this chapter, we will create a complex RESTful Web Service that will persist data in a PostgreSQL database. We will work with different types of relationships between the resources and we will take advantage of generic classes and generic views provided by the Django REST framework to reduce the amount of boilerplate code. We will gain an understanding of:

- Defining the requirements for a complex RESTful Web Service
- Creating a new app with Django
- Configuring a new web service
- Defining many-to-one relationships with `models.ForeignKey`
- Installing PostgreSQL
- Running migrations that generate relationships
- Analyzing the database
- Configuring serialization and deserialization with relationships
- Defining hyperlinks with `serializers.HyperlinkedModelSerializer`
- Working with class-based views
- Taking advantage of generic classes and generic views
- Generalizing and mixing behavior

- Working with routing and endpoints
- Making requests that interact with resources that have relationships

Defining the requirements for a complex RESTful Web Service

So far, our RESTful Web Service performed CRUD operations on a single database table. Now, we want to create a more complex RESTful Web Service with the Django REST framework to interact with a complex database model.

A drone is an **IoT** (short for **Internet of Things**) device that interacts with many sensors and actuators, including digital electronic speed controllers linked to engines, propellers, and sensors. A drone is also known as an **Unnamed Aerial Vehicle (UAV)**. We will code a RESTful Web Service that will allow us to register competitions for drones that are grouped into drone categories. In our previous RESTful Web Service, we had toys grouped in toy categories and we used a string field to specify the toy category for a toy. In this case, we want to be able to easily retrieve all the drones that belong to a specific drone category. Thus, we will have a relationship between a drone and a drone category.

We must be able to perform CRUD operations on diverse resources and resource collections. Many resources have relationships with other resources, and therefore, we won't work with simple models. We will learn how to establish different kinds of relationships between the models.

The following list enumerates the resources and the model name we will use to represent them in a Django REST framework:

- Drone categories (`DroneCategory` model)
- Drones (`Drone` model)
- Pilots (`Pilot` model)

- Competitions (`competition` model)

The drone category (`droneCategory` model) just requires a name.

We need the following data for a drone (`drone` model):

- A foreign key to a drone category (`droneCategory` model)
- A name
- A manufacturing date
- A bool value indicating whether the drone participated in at least one competition or not
- A timestamp with the date and time in which the drone was inserted in the database

We need the following data for a pilot (`pilot` model):

- A name
- A gender value
- An integer value with the number of races in which the pilot participated
- A timestamp with the date and time in which the pilot was inserted in the database

We need the following data for the competition (`competition` model):

- A foreign key to a pilot (`pilot` model)

- A foreign key to a drone (`drone` model)
- A distance value (measured in feet)
- A date in which the drone controlled by the pilot reached the specified distance value

We will use diverse options that the Django REST framework provides us to materialize the relationship between resources. This way, we will be able to analyze different configurations that will make it possible for us to know which is the best option based on the specific requirements of new web services that we will develop in the future.

The following table shows the HTTP verbs, the scope and the semantics for the methods that our new RESTful Web Service must support. Each method is composed by an HTTP verb and a scope. All the methods have well-defined meanings for all the resources and resource collections. In this case, we will implement the PATCH HTTP verb for all the resources:

HTTP verb	Scope	Semantics
GET	Drone category	Retrieve a single drone category. The drone category must include a list of URLs for each drone resource that belongs to the drone category.
GET	Collection of drone categories	Retrieve all the stored drone categories in the collection, sorted by their name in ascending order. Each drone category must include a list of URLs for each drone resource that belongs to the drone

		category.
POST	Collection of drone categories	Create a new drone category in the collection.
PUT	Drone category	Update an existing drone category.
PATCH	Drone category	Update one or more fields of an existing drone category.
DELETE	Drone category	Delete an existing drone category.
GET	Drone	Retrieve a single drone. The drone must include its drone category description.
GET	Collection of drones	Retrieve all the stored drones in the collection, sorted by their name in ascending order. Each drone must include its drone category description.
POST	Collection of drones	Create a new drone in the collection.
PUT	Drone	Update an existing drone.
PATCH	Drone	Update one or more fields of an existing drone.
DELETE	Drone	Delete an existing drone.

GET	Pilot	Retrieve a single pilot. The pilot must include a list of the registered competitions, sorted by distance in descending order. The list must include all the details for the competition in which the pilots and his related drone participated.
GET	Collection of pilots	Retrieve all the stored pilots in the collection, sorted by their name in ascending order. Each pilot must include a list of the registered competitions, sorted by distance in descending order. The list must include all the details for the competition in which the pilot and his related drone participated.
POST	Collection of pilots	Create a new pilot in the collection.
PUT	Pilot	Update an existing pilot.
PATCH	Pilot	Update one or more fields of an existing pilot.
DELETE	Pilot	Delete an existing pilot.
GET	Competition	Retrieve a single competition. The competition must include the pilot's name that made the drone reach a specific distance and the drone's name.

GET	Collection of competitions	Retrieve all the stored competitions in the collection, sorted by distance in descending order. Each competition must include the pilot's name that made the drone reach a specific distance and the drone's name.
POST	Collection of competitions	Create a new competition in the collection. The competition must be related to an existing pilot and an existing drone.
PUT	Competition	Update an existing competition.
PATCH	Competition	Update one or more fields of an existing competition.
DELETE	Competition	Delete an existing competition.

In the previous table, we have a huge number of methods and scopes. The following table enumerates the URIs for each scope mentioned in the previous table, where `{id}` has to be replaced with the numeric `id` or primary key of the resource:

Scope	URI
Collection of drone categories	/drone-categories/
Drone category	/drone-category/{id}
Collection of drones	/drones/

Drone	/drone/{id}
Collection of pilots	/pilots/
Pilot	/pilot/{id}
Collection of competitions	/competitions/
Competition	/competition/{id}

Let's consider that `http://localhost:8000/` is the URL for the RESTful Web Service running on Django's development server. We have to compose and send an HTTP request with the following HTTP verb (`GET`) and request URL (`http://localhost:8000/competitions/`) to retrieve all the stored competitions in the collection.

```
GET http://localhost:8000/competitions/
```

Our RESTful Web Service must be able to update a single field for an existing resource. In order to make this possible, we will implement the `PATCH` method. Remember that the `PUT` method is meant to replace an entire resource and the `PATCH` method is meant to apply a delta to an existing resource, that is, to update one or more fields for an existing resource.

We definitely want our RESTful Web Service to support the `OPTIONS` method for all the resources and collections of resources. This way, we will provide a consistent web service.

We will use the **ORM** (short for **Object-Relational Mapping**) included in Django. In addition, we will take advantage of many features and reusable elements included in the latest version of the Django REST framework to make it easy to build our web service without writing a huge amount of

code.

In this case, we will work with a PostgreSQL database. However, in case you don't want to spend time installing PostgreSQL, you can skip the changes we make in the Django REST framework ORM configuration and continue working with the default SQLite database, as we did with our first RESTful Web Service.

Creating a new app with Django

Now, we will create a new app with Django. We will follow the same steps we learned in [chapter 1, *Installing the Required Software and Tools*](#), in the *Creating an app with Django* section. However, in order to avoid repeating many steps, we will use the same `restful01` project we had created in that chapter. Hence, we will just add a new app to an existing project.

Make sure you quit Django's development server. Remember that you just need to press `Ctrl + C` in the terminal or command-prompt window in which it is running. In case you weren't running Django's development server, make sure the virtual environment in which we have been working in the previous chapter is activated. Then, go to the `restful01` folder within the `01` folder (the root folder for our virtual environment). The following commands use the default paths to go to this folder. In case you have used a different path for your virtual environment, make sure you replace the base path with the appropriate one.

In Linux or macOS, enter the following command:

```
cd ~/HillarDjangoREST/01/restful01
```

If you prefer the Command Prompt, run the following command in the Windows command line:

```
cd /d %USERPROFILE%\HillarDjangoREST\01\restful01
```

If you prefer Windows PowerShell, run the following command in

Windows PowerShell:

```
cd /d $env:USERPROFILE\HillarDjangoREST\01\restful01
```

Then, run the following command to create a new Django app named `drones` within the `restful01` Django project. The command won't produce any output:

```
python manage.py startapp drones
```

The previous command creates a new `restful01/drones` sub-folder, with the following files:

- `views.py`
- `tests.py`
- `models.py`
- `apps.py`
- `admin.py`
- `__init__.py`

In addition, the `restful01/drones` folder will have a `migrations` sub-folder with an `__init__.py` Python script. The structure is the same as the one we analyzed in [Chapter 1, *Installing the Required Software and Tools*](#), in the *Understanding Django folders, files and configurations* section.

Configuring a new web service

We added a new Django app to our existing Django project. Use your favorite editor or IDE to check the Python code in the `apps.py` file within the `restful01/drones` folder (`restful01\drones` in Windows). The following lines show the code for this file:

```
from django.apps import AppConfig

class DronesConfig(AppConfig):
    name = 'drones'
```

The code declares the `DronesConfig` class as a subclass of the `django.apps.AppConfig` class that represents a Django application and its configuration. The `DronesConfig` class just defines the `name` class attribute and sets its value to `'drones'`.

Now, we have to add `drones.apps.DronesConfig` as one of the installed apps in the `restful01/settings.py` file that configures settings for the `restful01` Django project. I built the previous string by concatenating many values as follows: app name + `.apps.` + class name, which is, `drones` + `.apps.` + `DronesConfig`.

We already added the `rest_framework` app to make it possible for us to use the Django REST framework in the `restful01/settings.py` file. However, in case you decided to create a new Django project from scratch by following all the steps we learned in chapter 1, Installing the Required Software and Tools, make sure you don't forget to add the `rest_framework` app.

Open the `restful01/settings.py` file that declares module-level variables that

define the configuration of Django for the `restful01` project. We will make some changes to this Django settings file. Locate the lines that assign a strings list to `INSTALLED_APPS` to declare the installed apps.

Remove the following line from the `INSTALLED_APPS` strings list. This way, Django won't consider this app anymore:

```
'toys.apps.ToysConfig',
```

Add the following string to the `INSTALLED_APPS` strings list and save the changes to the `restful01/settings.py` file:

```
'drones.apps.DronesConfig',
```

The following lines show the new code that declares the `INSTALLED_APPS` strings list with the added line highlighted and with comments to understand what each added string means. The code file for the sample is included in the `hillar_django_restful_06_01` folder, in the `restful01/settings.py` file:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Django REST framework
    'rest_framework',
    # Drones application
    'drones.apps.DronesConfig',
]
```

This way, we have added the `drones` application to our initial Django project named `restful01`.

Defining many-to-one relationships with models.ForeignKey

Now, we will create the models that we will use to represent and persist the drone categories, drones, pilots, and competitions, and their relationships. Open the `drones/models.py` file and replace its contents with the following code. The lines that declare fields related to other models are highlighted in the code listing. The code file for the sample is included in the `hillar_django_restful_06_01` folder, in the `restful01/drones/models.py` file.

```
from django.db import models

class DroneCategory(models.Model):
    name = models.CharField(max_length=250)

    class Meta:
        ordering = ('name',)

    def __str__(self):
        return self.name


class Drone(models.Model):
    name = models.CharField(max_length=250)
    drone_category = models.ForeignKey(
        DroneCategory,
        related_name='drones',
        on_delete=models.CASCADE)
    manufacturing_date = models.DateTimeField()
    has_it_competed = models.BooleanField(default=False)
    inserted_timestamp = models.DateTimeField(auto_now_add=True)

    class Meta:
        ordering = ('name',)

    def __str__(self):
```

```

        return self.name

class Pilot(models.Model):
    MALE = 'M'
    FEMALE = 'F'
    GENDER_CHOICES = (
        (MALE, 'Male'),
        (FEMALE, 'Female'),
    )
    name = models.CharField(max_length=150, blank=False, default='')
    gender = models.CharField(
        max_length=2,
        choices=GENDER_CHOICES,
        default=MALE,
    )
    races_count = models.IntegerField()
    inserted_timestamp = models.DateTimeField(auto_now_add=True)

    class Meta:
        ordering = ('name',)

    def __str__(self):
        return self.name


class Competition(models.Model):
    pilot = models.ForeignKey(
        Pilot,
        related_name='competitions',
        on_delete=models.CASCADE)
    drone = models.ForeignKey(
        Drone,
        on_delete=models.CASCADE)
    distance_in_feet = models.IntegerField()
    distance_achievement_date = models.DateTimeField()

    class Meta:
        # Order by distance in descending order
        ordering = ('-distance_in_feet',)

```

The code declares the following four models, specifically, four classes that are subclasses of the `django.db.models.Model` class:

- `DroneCategory`

- Drone
- Pilot
- Competition

Django automatically adds an auto-increment integer primary key column named `id` when it creates the database table related to each model. We specify the field types, maximum lengths, and defaults for many attributes.

Each class declares a `Meta` inner class that declares an ordering attribute. The `Meta` inner class declared within the `Competition` class specifies '`-distance_in_feet`' as the value of the `ordering` tuple, with a dash as a prefix of the field name to order by `distance_in_feet` in descending order, instead of the default ascending order.

The `DroneCategory`, `Drone`, and `Pilot` classes declare a `__str__` method that returns the contents of the `name` attribute that provides the name or title for each of these models. This way, whenever Django needs to provide a human readable representation for the model, it will call this method and return its result.

The `Drone` model declares the `drone_category` field with the following line:

```
drone_category = models.ForeignKey(  
    DroneCategory,  
    related_name='drones',  
    on_delete=models.CASCADE)
```

The previous line uses the `django.db.models.ForeignKey` class to provide a many-to-one relationship to the `DroneCategory` model. The '`drones`' value specified for the `related_name` argument creates a backwards relation from the `DroneCategory` model to the `Drone` model. This value indicates the name to use for the relation from the related `DroneCategory` object back to a `Drone` object.

This way, we will be able to easily access all the drones that belong to a specific drone category. Whenever we delete a drone category, we want all the drones that belong to this category to be deleted, too, and therefore, we specified the `models.CASCADE` value for the `on_delete` argument.

The `competition` model declares the `pilot` field with the following line:

```
pilot = models.ForeignKey(  
    Pilot,  
    related_name='competitions',  
    on_delete=models.CASCADE)
```

The previous line uses the `django.db.models.ForeignKey` class to provide a many-to-one relationship to the `Pilot` model. The '`competitions`' value specified for the `related_name` argument creates a backwards relation from the `Pilot` model to the `competition` model. This value indicates the name to use for the relation from the related Pilot object back to a Competition object.

This way, we will be able to easily access all the competitions in which a specific pilot participated with his drone. Whenever we delete a pilot, we want all the competitions in which this pilot participated to be deleted, too, and therefore, we specified the `models.CASCADE` value for the `on_delete` argument.

The `competition` model declares the `drone` field with the following line:

```
drone = models.ForeignKey(  
    Drone,  
    on_delete=models.CASCADE)
```

The previous line uses the `django.db.models.ForeignKey` class to provide a many-to-one relationship to the `Drone` model. In this case, we don't create a backwards relation because we don't need it. Thus, we don't specify a value for the `related_name` argument. Whenever we delete a drone, we want all the competitions in which the drone participated to be deleted too, and

therefore, we specified the `models.CASCADE` value for the `on_delete` argument.

Installing PostgreSQL

In order to work with this example, you have to download and install PostgreSQL, in case you aren't already running it in your development computer or in a development server. PostgreSQL is available on multiple platforms and offers versions for Linux, macOS and Windows, among other operating systems. You can download and install this relational database management system from the Download section of its web page:

<http://www.postgresql.org>.

The next samples that work with a PostgreSQL database have been tested in PostgreSQL versions 9.6.x and PostgreSQL 10.x. Make sure you work with any of these versions.

There are interactive installers built by EnterpriseDB and BigSQL for macOS and Windows. In case you are working with macOS, Postgres.app provides a really easy way to install and use PostgreSQL on this operating system. You can read more about Postgres.app and download it from its web page: <http://postgresapp.com>.

After you finish the installation process, you have to make sure that the PostgreSQL `bin` folder is included in the PATH environmental variable. You should be able to execute the `psql` command-line utility from your current Linux or macOS terminal, the Windows command prompt, or Windows PowerShell.

In case the `bin` folder isn't included in the PATH, you will receive an error indicating that the `pg_config` file cannot be found when trying to install the `psycopg2` Python package. In addition, you will have to use the full path to each of the PostgreSQL command-line tools that we will use in the next steps.

Running migrations that generate relationships

We must create the initial migration for the new models we recently coded. We just need to run the following Python scripts and we will also synchronize the database for the first time. As we learned from our previous RESTful Web Service sample, by default, Django uses an SQLite database.

In this new example, we will be working with a PostgreSQL database. However, in case you want to use an SQLite, you can skip all the next steps related to PostgreSQL, its configuration in Django, and jump to the migrations generation command. You will also have to use the SQLite utilities instead of the PostgreSQL tools to analyze the database.

We will use the PostgreSQL command-line tools to create a new database named `toys`. In case you already have a PostgreSQL database with this name, make sure that you use another name in all the commands and configurations. You can perform the same tasks with any PostgreSQL GUI tool or any database administration tool that supports PostgreSQL.

In case you are developing in Linux, it is necessary to run the commands as the `postgres` user. Run the following command in Windows or macOS to create a new database named `drones`. Notice that the command won't produce any output and that you must have the `bin` folder for PostgreSQL command-line tools in your path:

```
createdb drones
```

In Linux, run the following command to use the `postgres` user:

```
sudo -u postgres createdb drones
```

Now, we will use the psql command-line tool to run some SQL statements to create a specific user that we will use in Django and assign the necessary roles for it. In Windows or macOS, run the following command to launch the psql tool:

```
psql
```

In macOS, you might need to run the following command to launch psql with the `postgres` user in case the previous command doesn't work, as it will depend on the way in which you installed and configured PostgreSQL:

```
sudo -u postgres psql
```

In Linux, run the following command to start psql with the `postgres` user:

```
sudo -u psql
```

Then, run the following SQL statements and finally enter `\q` to exit the psql command-line tool.

Replace the username with your desired username to use in the new database, and password with your chosen password. We will specify the selected username and password in the Django settings for our web service.

You don't need to run the steps in case you are already working with a specific user in PostgreSQL and you have already granted privileges to the database for the user:

```
CREATE ROLE username WITH LOGIN PASSWORD 'password';
GRANT ALL PRIVILEGES ON DATABASE drones TO username;
```

```
ALTER USER username CREATEDB;
\q
```

You will see the following output for the previous commands:

```
CREATE ROLE
GRANT
ALTER ROLE
```

The default SQLite database engine and the database file name are specified in the `restful01/settings.py` Python file. The following lines show the default lines that configure the database:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

We will work with PostgreSQL instead of SQLite for this example, and therefore, we must replace the declaration of the `DATABASES` dictionary with the following lines. The nested dictionary maps the database named `default` with the `django.db.backends.postgresql` database engine, the desired database name and its settings. In this case, we will create a database named `drones`. Make sure you specify the desired database name in the value for the `'NAME'` key and that you configure the user, password, host, and port based on your PostgreSQL configuration. In case you followed the previously explained steps to configure the PostgreSQL database, use the settings specified in these steps. The code file for the sample is included in the `hillar_django_restful_06_01` folder, in the `restful01/settings.py` file:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        # Replace drones with your desired database name
        'NAME': 'drones',
```

```
# Replace username with your desired user name
'USER': 'username',
# Replace password with your desired password
'PASSWORD': 'password',
# Replace 127.0.0.1 with the PostgreSQL host
'HOST': '127.0.0.1',
# Replace 5432 with the PostgreSQL configured port
# in case you aren't using the default port
'PORT': '5432',
}
}
```

In case you decided to continue working with SQLite, you don't need to make the previous changes and you can continue using the default configuration.

We don't want the migrations process to take into account our models related to toys from the previous web service, and therefore, we will make changes to the code in the `urls.py` file in the `restful01` folder, specifically, the `restful01/urls.py` file. The file defines the root URL configurations, and therefore, we must remove the URL patterns declared in the `toys/urls.py` file. The following lines show the new code for the `restful01/urls.py` file. The code file for the sample is included in the `hillar_django_restful_06_01` folder, in the `restful01/urls.py` file:

```
from django.conf.urls import url, include

urlpatterns = [
]
```

In order to use PostgreSQL, it is necessary to install a Python-PostgreSQL Database Adapter that Django will use to interact with a PostgreSQL database: the Psycopg2 package (`psycopg2`).

In macOS, we have to make sure that the PostgreSQL `bin` folder is included in the `PATH` environmental variable. For example, in case the path to the `bin` folder is `/Applications/Postgres.app/Contents/Versions/latest/bin`, we must execute the following command to add this folder to the `PATH` environmental variable:

```
export PATH=$PATH:/Applications/Postgres.app/Contents/Versions/latest/bin
```

Once we make sure that the PostgreSQL `bin` folder is included in the `PATH` environmental variable, we just need to run the following command to install this package. Make sure the virtual environment is activated before running the command:

```
pip install psycopg2
```

The last lines for the output will indicate that the `psycopg2` package has been successfully installed:

```
Collecting psycopg2
  Installing collected packages: psycopg2
    Successfully installed psycopg2-2.7.3.2
```

Now, run the following Python script to generate the migrations that will allow us to synchronize the PostgreSQL database for the first time. We will run the migrations for the `drones` application:

```
python manage.py makemigrations drones
```

The following lines show the output generated after running the previous command:

```
Migrations for 'drones':
drones/migrations/0001_initial.py
- Create model Competition
- Create model Drone
- Create model DroneCategory
- Create model Pilot
- Add field drone_category to drone
- Add field drone to competition
- Add field pilot to competition
```

The output indicates that the `restful01/drones/migrations/0001_initial.py` file includes the code to create the `Competition`, `Drone`, `DroneCategory`, and `Pilot` models. The following lines show the code for this file that was automatically generated by Django. The code file for the sample is included in the `hillar_django_restful_06_01` folder, in the `restful01/drones/migrations/0001_initial.py` file:

```
# -*- coding: utf-8 -*-
# Generated by Django 1.11.5 on 2017-11-02 02:55
from __future__ import unicode_literals

from django.db import migrations, models
import django.db.models.deletion


class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='Competition',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False)),
                ('distance_in_feet', models.IntegerField()),
                ('distance_achievement_date', models.DateTimeField()),
            ],
            options={
                'ordering': ('-distance_in_feet',),
            },
        ),
        migrations.CreateModel(
            name='Drone',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False)),
                ('name', models.CharField(max_length=250)),
                ('manufacturing_date', models.DateTimeField()),
                ('has_it_competed', models.BooleanField(default=False)),
                ('inserted_timestamp', models.DateTimeField(auto_now_add=True)),
            ],
            options={
                'ordering': ('name',),
            }
        )
    ]
}
```

```
        },
    ),
    migrations.CreateModel(
        name='DroneCategory',
        fields=[
            ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False)),
            ('name', models.CharField(max_length=250)),
        ],
        options={
            'ordering': ('name',),
        },
    ),
    migrations.CreateModel(
        name='Pilot',
        fields=[
            ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False)),
            ('name', models.CharField(default='', max_length=50)),
            ('gender', models.CharField(choices=[('M', 'Male'), ('F', 'Female')], max_length=1)),
            ('races_count', models.IntegerField()),
            ('inserted_timestamp', models.DateTimeField(auto_now_add=True)),
        ],
        options={
            'ordering': ('name',),
        },
    ),
    migrations.AddField(
        model_name='drone',
        name='drone_category',
        field=models.ForeignKey(on_delete=django.db.models.deletion.CASCADE),
    ),
    migrations.AddField(
        model_name='competition',
        name='drone',
        field=models.ForeignKey(on_delete=django.db.models.deletion.CASCADE),
    ),
    migrations.AddField(
        model_name='competition',
        name='pilot',
        field=models.ForeignKey(on_delete=django.db.models.deletion.CASCADE),
    ),
],
]
```

The code defines a subclass of the `django.db.migrations.Migration` class named `Migration` that defines an `operations` list with many calls to `migrations.CreateModel`. Each `migrations.CreateModel` call will create the table for each of the related models.

Notice that Django has automatically added an `id` field for each of the models. The `operations` are executed in the same order in which they are displayed in the list. The code creates `competition`, `Drone`, `DroneCategory`, `Pilot` and finally adds the following fields with foreign keys:

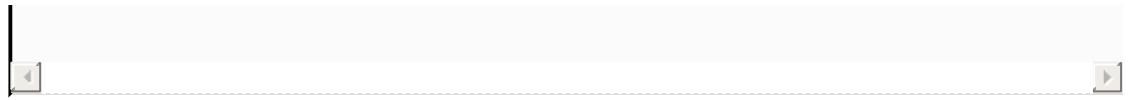
- The `drone_category` field to the `Drone` model with the foreign key to the `DroneCategory` model
- The `drone` field to the `competition` model with the foreign key to the `Drone` model
- The `pilot` field to the `competition` model with the foreign key to the `Pilot` model

Now, run the following python script to apply all the generated migrations:

```
python manage.py migrate
```

The following lines show the output generated after running the previous command:

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, drones, sessions
    Running migrations:
      Applying contenttypes.0001_initial... OK
      Applying auth.0001_initial... OK
      Applying admin.0001_initial... OK
      Applying admin.0002_logentry_remove_auto_add... OK
      Applying contenttypes.0002_remove_content_type_name... OK
      Applying auth.0002_alter_permission_name_max_length... OK
      Applying auth.0003_alter_user_email_max_length... OK
      Applying auth.0004_alter_user_username_opts... OK
      Applying auth.0005_alter_user_last_login_null... OK
      Applying auth.0006_require_contenttypes_0002... OK
      Applying auth.0007_alter_validators_add_error_messages... OK
      Applying auth.0008_alter_user_username_max_length... OK
      Applying drones.0001_initial... OK
      Applying sessions.0001_initial... OK
```



Analyzing the database

After we have run the previous command, we can use the PostgreSQL command-line tools or any other application that allows us to easily check the contents of a PostgreSQL database, to check the tables that Django generated. **If you decided to continue working with SQLite, we already learned how to check the tables in the previous chapters.**

Run the following command to list the generated tables:

```
psql --username=username --dbname=drones --command="\dt"
```

The following lines show the output with all the generated table names:

List of relations			
Schema	Name	Type	Owner
public	auth_group	table	username
public	auth_group_permissions	table	username
public	auth_permission	table	username
public	auth_user	table	username
public	auth_user_groups	table	username
public	auth_user_user_permissions	table	username
public	django_admin_log	table	username
public	django_content_type	table	username
public	django_migrations	table	username
public	django_session	table	username
public	drones_competition	table	username
public	drones_drone	table	username
public	drones_dronecategory	table	username
public	drones_pilot	table	username
(14 rows)			

In our previous example, Django used the `toys_` prefix for the table related

to the `toys` application we had created. In this case, the application name is `drones`, and therefore, Django uses the `drones_` prefix for the following four tables that are related to the application:

- `drones_drone`: This table persists the `drone` model
- `drones_dronecategory`: This table persists the `droneCategory` model
- `drones_pilot`: This table persists the `pilot` model
- `drones_competition`: This table persists the `competition` model

Django's integrated ORM generated these tables and the foreign keys based on the information included in our models and the code generated during the migrations process.

The following commands will allow you to check the contents of the four tables after we compose and send different HTTP requests to the RESTful Web Service, and these calls end up making CRUD operations to the four tables. The commands assume that you are running PostgreSQL on the same computer in which you are running the command:

```
psql --username=username --dbname=drones --command="SELECT * FROM drones_dro
psql --username=username --dbname=drones --command="SELECT * FROM drones_dro
psql --username=username --dbname=drones --command="SELECT * FROM drones_pil
psql --username=username --dbname=drones --command="SELECT * FROM drones_com
```

Instead of working with the PostgreSQL command-line utility, you can use a GUI tool to check the contents of the PostgreSQL database. You can use also the database tools included in your favorite IDE to check the contents for the SQLite database.

As happened in our previous example, Django generated additional tables that it requires to support the web framework and the authentication features that we will use later.

Configuring serialization and deserialization with relationships

The new RESTful Web Service must be able to serialize the `DroneCategory`, `Drone`, `Pilot`, and `Competition` instances into JSON representations and vice versa. In this case, we must pay special attention to the relationships between the different models when we create the serializer classes to manage serialization to JSON and deserialization from JSON.

In our last version of the previous RESTful Web Service, we created a subclass of the `rest_framework.serializers.ModelSerializer` class to make it easier to generate a serializer and reduce boilerplate code. In this case, we will also declare one class that inherits from `ModelSerializer`. The other three classes will inherit from the

`rest_framework.serializers.HyperlinkedModelSerializer` class.

The `HyperlinkedModelSerializer` is a type of `ModelSerializer` that uses hyperlinked relationships instead of primary key relationships, and therefore, it represents the relationships to other model instances with hyperlinks instead of primary key values. In addition, the `HyperlinkedModelSerializer` generates a field named `url` with the URL for the resource as its value. As happens with `ModelSerializer`, the `HyperlinkedModelSerializer` class provides default implementations for the `create` and `update` methods.

Now, go to the `restful01/drones` folder and create a new Python code file named `serializers.py`. The following lines show the code that declares the new `DroneCategorySerializer` class. Notice that we will add more classes to this file later. The code file for the sample is included in the `hillar_django_restful_06_01` folder in the `restful01/drones/serializers.py` file:

```
from rest_framework import serializers
from drones.models import DroneCategory
from drones.models import Drone
from drones.models import Pilot
from drones.models import Competition
import drones.views

class DroneCategorySerializer(serializers.HyperlinkedModelSerializer):
    drones = serializers.HyperlinkedRelatedField(
        many=True,
        read_only=True,
        view_name='drone-detail')

    class Meta:
        model = DroneCategory
        fields = (
            'url',
            'pk',
            'name',
            'drones')
```

The `DroneCategorySerializer` class is a subclass of the `HyperlinkedModelSerializer` class. The `DroneCategorySerializer` class declares a `drones` attribute that holds an instance of `serializers.HyperlinkedRelatedField` with `many` and `read_only` equal to `True`. This way, the code defines a one-to-many relationship that is read-only.

The code uses the `drones` name that we specified as the `related_name` string value when we created the `drone_category` field as a `models.ForeignKey` instance in the `Drone` model. This way, the `drones` field will provide us with an array of hyperlinks to each drone that belongs to the drone category.

The `view_name` value is '`drone-detail`' to indicate the browsable API feature to use the drone detail view to render the hyperlink when the user clicks or taps on it. This way, we make it possible for the browsable API to allow us to browse between related models.

The `DroneCategorySerializer` class declares a `Meta` inner class that declares the following two attributes:

- `model`: This attribute specifies the model related to the serializer, that is, the `DroneCategory` class.
- `fields`: This attribute specifies a tuple of string whose values indicate the field names that we want to include in the serialization from the model related to the serializer, that is, the `DroneCategory` class. We want to include both the primary key and the URL, and therefore, the code specified both '`pk`' and '`url`' as members of the tuple. In addition, we want to include the name and the field that provides hyperlinks to each drone that belongs to the drone category. Thus, the code also specifies '`name`' and '`drones`' as members of the tuple.

There is no need to override either the `create` or `update` methods because the generic behavior will be enough in this case. The `HyperlinkedModelSerializer` superclass provides implementations for both methods.

Defining hyperlinks with serializers.HyperlinkedModelSerializer

Now, add the following code to the `serializers.py` file to declare the `DroneSerializer` class. The code file for the sample is included in the `hillar_django_restful_06_01` folder in the `restful01/drones/serializers.py` file:

```
class DroneSerializer(serializers.HyperlinkedModelSerializer):
    # Display the category name
    drone_category = serializers.SlugRelatedField(queryset=DroneCategory.objects

    class Meta:
        model = Drone
        fields = (
            'url',
            'name',
            'drone_category',
            'manufacturing_date',
            'has_it_competed',
            'inserted_timestamp')
```

The `DroneSerializer` class is a subclass of the `HyperlinkedModelSerializer` class. The `DroneSerializer` class declares a `drone_category` attribute that holds an instance of `serializers.SlugRelatedField` with its `queryset` argument set to `DroneCategory.objects.all()` and its `slug_field` argument set to `'name'`.

A `slugRelatedField` is a read-write field that represents the target of the relationship by a unique slug attribute, that is, the description. In the Drone model, we created the `drone_category` field as a `models.ForeignKey` instance.

We want to display the drone category name as the description (slug field)

for the related `DroneCategory`, and therefore, we specified `'name'` as the value for the `slug_field` argument. The browsable API has to display the possible options for the related drone category in a dropdown field in a form, and Django will use the expression specified in the `queryset` argument to retrieve all the possible instances and display their specified slug field.

The `DroneSerializer` class declares a `Meta` inner class that declares two attributes:

- `model`: The `Drone` class.
- `fields`. In this case, we don't want to include the primary key, and therefore, the tuple doesn't include the `'pk'` string. The `drone_category` field will render the `name` field for the related `DroneCategory`.

Now, add the following code to the `serializers.py` file to declare the `CompetitionSerializer` class. The code file for the sample is included in the `hillar_django_restful_06_01` folder in the `restful01/drones/serializers.py` file.

```
class CompetitionSerializer(serializers.HyperlinkedModelSerializer):  
    # Display all the details for the related drone  
    drone = DroneSerializer()  
    class Meta:  
        model = Competition  
        fields = (  
            'url',  
            'pk',  
            'distance_in_feet',  
            'distance_achievement_date',  
            'drone')
```

The `CompetitionSerializer` class is a subclass of the `HyperlinkedModelSerializer` class. We will use the `CompetitionSerializer` class to serialize `Competition` instances related to a `Pilot`, that is, to display all the competitions in which a specific `Pilot` has participated when we serialize a `Pilot`. We want to display all the details for the related `Drone`, but we don't include the related

`Pilot` because the `Pilot` will use this `competitionSerializer` serializer.

The `competitionSerializer` class declares a `drone` attribute that holds an instance of the previously coded `DroneSerializer` class. We created the `drone` field as a `models.ForeignKey` instance in the `Competition` model and we want to serialize the same data for the drone that we coded in the `DroneSerializer` class.

The `competitionSerializer` class declares a `Meta` inner class that declares two attributes: `model` and `fields`. The `model` attribute specifies the model related to the serializer, that is, the `Competition` class. As previously explained, we don't include the '`Pilot`' field name in the `fields` tuple of string to avoid serializing the Pilot again. We will use a `PilotSerializer` as a master and the `CompetitionSerializer` as the detail.

Now, add the following code to the `serializers.py` file to declare the `PilotSerializer` class. The code file for the sample is included in the `hillar_django_restful_06_01` folder in the `restful01/drones/serializers.py` file:

```
class PilotSerializer(serializers.HyperlinkedModelSerializer):
    competitions = CompetitionSerializer(many=True, read_only=True)
    gender = serializers.ChoiceField(
        choices=Pilot.GENDER_CHOICES)
    gender_description = serializers.CharField(
        source='get_gender_display',
        read_only=True)

    class Meta:
        model = Pilot
        fields = (
            'url',
            'name',
            'gender',
            'gender_description',
            'races_count',
            'inserted_timestamp',
            'competitions')
```

The `PilotSerializer` class is a subclass of the `HyperlinkedModelSerializer` class. We will use the `PilotSerializer` class to serialize `Pilot` instances and we will

use the previously coded `CompetitionSerializer` class to serialize all the `Competition` instances related to the `Pilot`.

The `PilotSerializer` class declares a `competitions` attribute that holds an instance of the previously coded `CompetitionSerializer` class. The `many` argument is set to `True` because it is a one-to-many relationship (one `Pilot` has many related `Competition` instances).

We use the `competitions` name that we specified as the `related_name` string value when we created the `Pilot` field as a `models.ForeignKey` instance in the `Competition` model. This way, the `competitions` field will render each `Competition` that belongs to the `Pilot` by using the previously declared `CompetitionSerializer`.

The `Pilot` model declared `gender` as an instance of `models.CharField` with the `choices` attribute set to the `Pilot.GENDER_CHOICES` string tuple. The `PilotSerializer` class declares a `gender` attribute that holds an instance of `serializers.ChoiceField` with the `choices` argument set to the `Pilot.GENDER_CHOICES` string tuple.

In addition, the class declares a `gender_description` attribute with `read_only` set to `True` and the `source` argument set to `'get_gender_display'`. The `source` string is built with the `get_` prefix followed by the field name, `gender`, and the `_display` suffix. This way, the read-only `gender_description` attribute will render the description for the gender choices instead of the single char stored values.

The `PilotSerializer` class declares a `Meta` inner class that declares two attributes: `model` and `fields`. The `model` attribute specifies the model related to the serializer, that is, the `Pilot` class. We will use the `PilotSerializer` class as a master and the `CompetitionSerializer` class as the detail. As happened with the previous serializers, the `fields` attribute specifies a tuple of string whose values indicate the field names that we want to include in the serialization from the related model.

Finally, add the following code to the `serializers.py` file to declare the `PilotCompetitionSerializer` class. The code file for the sample is included in the `hillar_django_restful_06_01` folder in the `restful01/drones/serializers.py` file:

```
class PilotCompetitionSerializer(serializers.ModelSerializer):
    # Display the pilot's name
    pilot = serializers.SlugRelatedField(queryset=Pilot.objects.all(), slug_field='name')
    # Display the drone's name
    drone = serializers.SlugRelatedField(queryset=Drone.objects.all(), slug_field='name')

    class Meta:
        model = Competition
        fields = (
            'url',
            'pk',
            'distance_in_feet',
            'distance_achievement_date',
            'pilot',
            'drone')
```

The `PilotCompetitionSerializer` class is a subclass of the `ModelSerializer` class. We will use the `CompetitionSerializer` class to serialize `Competition` instances. Previously, we created the `CompetitionSerializer` class to serialize `Competition` instances as the detail of a `Pilot`. We will use the new `PilotCompetitionSerializer` class whenever we want to display the related `Pilot` name and the related `Drone` name.

In the other serializer class, the `CompetitionSerializer` class, we didn't include any information related to the `Pilot` and we included all the details for the drone. This new class is an example of the flexibility that we have when we declare more than one serializer for the same model.

The `PilotCompetitionSerializer` class declares a `pilot` attribute that holds an instance of `serializers.SlugRelatedField` with its `queryset` argument set to `Pilot.objects.all()` and its `slug_field` argument set to `'name'`. We created the `pilot` field as a `models.ForeignKey` instance in the `Competition` model and we want to display the Pilot's name as the description (slug field) for the related `Pilot`. Thus, we specified `'name'` as the `slug_field`. When the browsable API has to display the possible options for the related pilot in a dropdown in a form, Django will use the expression specified in the `queryset` argument to retrieve all the possible pilots and display their specified slug field.

The `PilotCompetitionSerializer` class declares a `drone` attribute that holds an instance of `serializers.SlugRelatedField` with its `queryset` argument set to `Drone.objects.all()` and its `slug_field` argument set to `'name'`. We created the `drone` field as a `models.ForeignKey` instance in the `Competition` model and we want to display the drone's name as the description (slug field) for the related `Drone`.

We have coded all the necessary serializers for our new RESTful Web Service. The following table summarizes the serializers we have. Notice that we have two different serializers for the `Competition` model:

Serializer class name	Superclass	Related model
<code>DroneCategorySerializer</code>	<code>HyperlinkedModelSerializer</code>	<code>DroneCategory</code>
<code>DroneSerializer</code>	<code>HyperlinkedModelSerializer</code>	<code>Drone</code>
<code>CompetitionSerializer</code>	<code>HyperlinkedModelSerializer</code>	<code>Competition</code>
<code>PilotSerializer</code>	<code>HyperlinkedModelSerializer</code>	<code>Pilot</code>
<code>PilotCompetitionSerializer</code>	<code>ModelSerializer</code>	<code>Competition</code>

Working with class-based views

We will write our RESTful Web Service by coding class-based views. We will take advantage of a set of generic views that we can use as our base classes for our class-based views to reduce the required code to the minimum and reuse the behavior that has been generalized in the Django REST framework.

We will create subclasses of the two following generic class views declared in the `rest_framework.generics` module:

- `ListCreateAPIView`: This class view implements the `get` method that retrieves a listing of a queryset and the `post` method that creates a model instance
- `RetrieveUpdateDestroyAPIView`: This class view implements the `get`, `delete`, `put`, and `patch` methods to retrieve, delete, completely update, or partially update a model instance

Those two generic views are composed by combining reusable bits of behavior in the Django REST framework implemented as mixin classes declared in the `rest_framework.mixins` module. We can create a class that uses multiple inheritance and combine the features provided by many of these mixin classes.

The following line shows the declaration of the `ListCreateAPIView` class as the composition of `ListModelMixin`, `CreateModelMixin`, and `rest_framework.generics.GenericAPIView`:

```
class ListCreateAPIView(mixins.ListModelMixin,  
                      mixins.CreateModelMixin,  
                      GenericAPIView):
```

The following line shows the declaration of the `RetrieveUpdateDestroyAPIView` class as the composition of `RetrieveModelMixin`, `UpdateModelMixin`, `DestroyModelMixin`, and `rest_framework.generics.GenericAPIView`:

```
class RetrieveUpdateDestroyAPIView(mixins.RetrieveModelMixin,  
                                   mixins.UpdateModelMixin,  
                                   mixins.DestroyModelMixin,  
                                   GenericAPIView):
```

Taking advantage of generic classes and viewsets

Now, we will create many Django class-based views that will use the previously explained generic classes combined with the serializer classes to return JSON representations for each HTTP request that our RESTful Web Service will handle.

We will just have to specify a queryset that retrieves all the objects in the `queryset` attribute and the serializer class in the `serializer_class` attribute for each subclass that we declare. The behavior coded in the generic classes will do the rest for us. In addition, we will declare a `name` attribute with the string name we will use to identify the view.

Go to the `restful01/drones` folder and open the `views.py` file. Replace the code in this file with the following code that declares the required imports and the class-based views for our web service. We will add more classes to this file later. The code file for the sample is included in the

`hillar_django_restful_06_01` folder in the `restful01/drones/views.py` file:

```
from django.shortcuts import render
from rest_framework import generics
from rest_framework.response import Response
from rest_framework.reverse import reverse
from drones.models import DroneCategory
from drones.models import Drone
from drones.models import Pilot
from drones.models import Competition
from drones.serializers import DroneCategorySerializer
from drones.serializers import DroneSerializer
from drones.serializers import PilotSerializer
from drones.serializers import PilotCompetitionSerializer


class DroneCategoryList(generics.ListCreateAPIView):
    queryset = DroneCategory.objects.all()
```

```
    serializer_class = DroneCategorySerializer
    name = 'dronecategory-list'

class DroneCategoryDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = DroneCategory.objects.all()
    serializer_class = DroneCategorySerializer
    name = 'dronecategory-detail'

class DroneList(generics.ListCreateAPIView):
    queryset = Drone.objects.all()
    serializer_class = DroneSerializer
    name = 'drone-list'

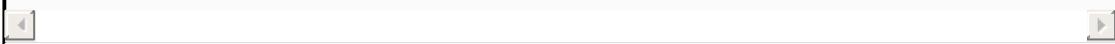
class DroneDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Drone.objects.all()
    serializer_class = DroneSerializer
    name = 'drone-detail'

class PilotList(generics.ListCreateAPIView):
    queryset = Pilot.objects.all()
    serializer_class = PilotSerializer
    name = 'pilot-list'

class PilotDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Pilot.objects.all()
    serializer_class = PilotSerializer
    name = 'pilot-detail'

class CompetitionList(generics.ListCreateAPIView):
    queryset = Competition.objects.all()
    serializer_class = PilotCompetitionSerializer
    name = 'competition-list'

class CompetitionDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Competition.objects.all()
    serializer_class = PilotCompetitionSerializer
    name = 'competition-detail'
```



Generalizing and mixing behavior

The previous classes took advantage of generalized behavior. Each of the superclasses for the classes mixed different generalized behaviors. The following table summarizes the HTTP verbs that each class-based view is going to process and the scope to which it applies. Notice that we can execute the `OPTIONS` HTTP verb on any of the scopes:

Scope	Class-based view name	HTTP verbs that it will process
Collection of drone categories: <code>/drone-categories/</code>	<code>DroneCategoryList</code>	GET, POST, and <code>OPTIONS</code>
Drone category: <code>/drone-category/{id}</code>	<code>DroneCategoryDetail</code>	GET, PUT, PATCH, DELETE, and <code>OPTIONS</code>
Collection of drones: <code>/drones/</code>	<code>DroneList</code>	GET, POST, and <code>OPTIONS</code>
Drone: <code>/drone/{id}</code>	<code>DroneDetail</code>	GET, PUT, PATCH, DELETE, and <code>OPTIONS</code>
Collection of Pilots: <code>/pilots/</code>	<code>PilotList</code>	GET, POST and <code>OPTIONS</code>
		GET, PUT, PATCH,

Pilot: /Pilot/{id}	PilotDetail	DELETE and OPTIONS
Collection of competitions: /competitions/	CompetitionList	GET, POST and OPTIONS
Score: /competition/{id}	CompetitionDetail	GET, PUT, PATCH, DELETE and OPTIONS

Working with routing and endpoints

We want to create an endpoint for the root of our web service to make it easy to browse the resource collections and resources provided by our web service with the browsable API feature and understand how everything works. Add the following code to the `views.py` file in the `restful01/drones` folder to declare the `ApiRoot` class as a subclass of the `generics.GenericAPIView` class. The code file for the sample is included in the `hillar_django_restful_06_01` folder in the `restful01/drones/views.py` file:

```
class ApiRoot(generics.GenericAPIView):
    name = 'api-root'
    def get(self, request, *args, **kwargs):
        return Response({
            'drone-categories': reverse(DroneCategoryList.name, request=request),
            'drones': reverse(DroneList.name, request=request),
            'pilots': reverse(PilotList.name, request=request),
            'competitions': reverse(CompetitionList.name, request=request)
        })
```

The `ApiRoot` class is a subclass of the `rest_framework.generics.GenericAPIView` class and declares the `get` method. The `GenericAPIView` class is the base class for all the other generic views we have previously analyzed.

The `ApiRoot` class defines the `get` method that returns a `Response` object with key/value pairs of strings that provide a descriptive name for the view and its URL, generated with the `rest_framework.reverse.reverse` function. This URL resolver function returns a fully qualified URL for the view.

Go to the `restful01/drones` folder and create a new file named `urls.py`. Write the following code in this new file. The following lines show the code for this file that defines the URL patterns that specifies the regular expressions

that have to be matched in the request to run a specific method for a class-based view defined in the `views.py` file.

In the previous example, we specified a function that represented a view. In this case, we call the `as_view` method for each appropriate class-based view. The code file for the sample is included in the `hillar_django_restful_06_01` folder in the `restful01/drones/urls.py` file:

```
from django.conf.urls import url
from drones import views

urlpatterns = [
    url(r'^drone-categories/$',
        views.DroneCategoryList.as_view(),
        name=views.DroneCategoryList.name),
    url(r'^drone-categories/(?P<pk>[0-9]+)$',
        views.DroneCategoryDetail.as_view(),
        name=views.DroneCategoryDetail.name),
    url(r'^drones/$',
        views.DroneList.as_view(),
        name=views.DroneList.name),
    url(r'^drones/(?P<pk>[0-9]+)$',
        views.DroneDetail.as_view(),
        name=views.DroneDetail.name),
    url(r'^pilots/$',
        views.PilotList.as_view(),
        name=views.PilotList.name),
    url(r'^pilots/(?P<pk>[0-9]+)$',
        views.PilotDetail.as_view(),
        name=views.PilotDetail.name),
    url(r'^competitions/$',
        views.CompetitionList.as_view(),
        name=views.CompetitionList.name),
    url(r'^competitions/(?P<pk>[0-9]+)$',
        views.CompetitionDetail.as_view(),
        name=views.CompetitionDetail.name),
    url(r'^$', 
        views.ApiRoot.as_view(),
        name=views.ApiRoot.name),
]
```

Now, we have to replace the code in the `urls.py` file in the `restful01` folder, specifically, the `restful01/urls.py` file. The file defines the root URL

configurations, and therefore, we must include the URL patterns declared in the previously coded `drones/urls.py` file. The following lines show the new code for the `restful01/urls.py` file. The code file for the sample is included in the `hillar_django_restful_06_01` folder, in the `restful01/urls.py` file:

```
from django.conf.urls import url, include  
  
urlpatterns = [  
    url(r'^$', include('drones.urls')),  
]
```

Making requests that interact with resources that have relationships

Now, we will use the HTTP command or its curl equivalents to compose and send HTTP requests to the recently coded RESTful Web Service that allows us to work with drone categories, drones, pilots, and competitions. We will use JSON for the requests that require additional data. Remember that you can perform the same tasks with your favorite GUI- based tool or with the browsable API.

Launch Django's development server to compose and send HTTP requests to our new unsecure Web service. We will definitely add security later. In case you don't remember how to start Django's development server, check the instructions in [chapter 3, *Creating API Views*](#), in the *Launching Django's development server* section.

First, we will compose and send an HTTP POST request to create a new drone category:

```
http POST :8000/drone-categories/ name="Quadcopter"
```

The following is the equivalent curl command:

```
curl -iX POST -H "Content-Type: application/json" -d '{"name": "Quadcopter"}' loc
```

The previous command will compose and send a POST HTTP request with the specified JSON key/value pair. The request specifies /drone-categories/, and therefore, it will match the '^drone-categories/\$' regular expression and

run the `post` method for the `views.DroneCategoryList` class based view. Remember that the method is defined in the `ListCreateAPIView` superclass and it ends up calling the `create` method defined in `mixins.CreateModelMixin`.

If the new `DroneCategory` instance was successfully persisted in the database, the call to the method will return an `HTTP 201 Created` status code and the recently persisted `DroneCategory` serialized to JSON in the response body. The following line shows a sample response for the HTTP request with the new `DroneCategory` object in the JSON response. Notice that the response body includes both the primary key, `pk`, and the URL, `url`, for the created category. The `drones` array is empty because there aren't drones related to the recently created drone category yet. The response doesn't include the header to focus on the body:

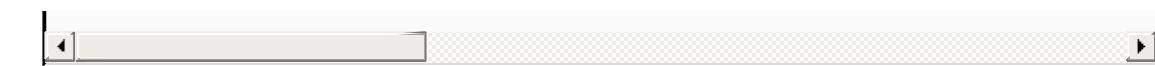
```
{  
    "drones": [],  
    "name": "Quadcopter",  
    "pk": 1,  
    "url": "http://localhost:8000/drone-categories/1"  
}
```

Now, we will compose and send HTTP requests to create two drones that belong to the drone category we recently created: `quadcopter`. We specify the `drone_category` value with the name of the desired drone category. The database table that persists the `Drone` model (the `drones_drone` table) will save the value of the primary key of the related `DroneCategory` whose name value matches the one we provide:

```
http POST :8000/drones/ name="WonderDrone" drone_category="Quadcopter" manuf  
http POST :8000/drones/ name="Atom" drone_category="Quadcopter" manufacturin
```

The following are the equivalent curl commands:

```
curl -iX POST -H "Content-Type: application/json" -d '{"name": "WonderDrone",  
curl -iX POST -H "Content-Type: application/json" -d '{"name": "Atom", "drone
```



The previous commands will compose and send two `POST` HTTP requests with the specified JSON key/value pairs. The request specifies `/toys/`, and therefore, it will match the `'^toys/$'` regular expression and run the `post` method for the `views.DroneList` class based view.

The following lines show sample responses for the two HTTP requests with the new `drone` objects in the JSON responses. Notice that the response includes only the URL, `url`, for the created drones and doesn't include the primary key. The value for `drone_category` is the `name` for the related `DroneCategory`:



We can run the commands explained in the *Analyzing the database* section to check the rows that were inserted in the tables that Django created in the PostgreSQL database to persist the models. The `drone_category_id` column for the `drones_drone` table saves the value of the primary key of the related row in the `drones_drone_category` table. The `DroneSerializer` class uses the `slugRelatedField` to display the `name` value for the related `DroneCategory`. The following screenshot uses the `psql` command-line utility to query the contents for the `drones_drone_category` and the `drones_drone` table in a

PostgreSQL database after running the HTTP requests:

```
bin — psql --username=username --dbname=drones — 125x36
drones=> SELECT * FROM drones_dronecategory;
 id | name
----+
 1 | Quadcopter
(1 row)

drones=> SELECT * FROM drones_drone;
 id | name | manufacturing_date | has_it_competed | inserted_timestamp | drone_category_id
----+
 1 | WonderDrone | 2017-07-19 23:02:00.716312-03 | f | 2017-11-02 22:58:49.135737-03 | 1
 2 | Atom | 2017-08-17 23:02:00.716312-03 | f | 2017-11-02 22:59:31.108031-03 | 1
(2 rows)
```

Now, we will compose and send an HTTP request to retrieve the drone category that contains the two drones we created. Don't forget to replace `1` with the primary key value of the drone category whose name is equal to 'Quadcopter' in your configuration:

```
http :8000/drone-categories/1
```

The following is the equivalent curl command:

```
curl -iX GET localhost:8000/drone-categories/1
```

The previous commands will compose and send the following HTTP request: `GET http://localhost:8000/drone-categories/1`. The request has a number after `/drone-categories/`, and therefore, it will match the `'^drone-categories/(?P<pk>[0-9]+)$'` regular expression and run the `get` method for the `views.DroneCategoryDetail` class-based view.

Remember that the method is defined in the `RetrieveUpdateDestroyAPIView` superclass and it ends up calling the `retrieve` method defined in `mixins.RetrieveModelMixin`. The following lines show a sample response for the HTTP request, with the `droneCategory` object and the hyperlinks of the related drones in the JSON response:

```
HTTP/1.0 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Length: 154
```

```
Content-Type: application/json
Date: Fri, 03 Nov 2017 02:58:33 GMT
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
    "drones": [
        "http://localhost:8000/drones/2",
        "http://localhost:8000/drones/1"
    ],
    "name": "Quadcopter",
    "pk": 1,
    "url": "http://localhost:8000/drone-categories/1"
}
```

The `DroneCategorySerializer` class defined the `drones` attribute as a `HyperlinkedRelatedField`, and therefore, the serializer renders the URL for each related `drone` instance in the value for the `drones` array. Later, we will display the results in a web browser through the browsable API and we will be able to click or tap on the hyperlink to see the details for each drone.

Now, we will compose and send an HTTP `POST` request to create a drone related to a drone category name that doesn't exist: '`Octocopter`':

```
http POST :8000/drones/ name="Noisy Drone" drone_category="Octocopter" manuf
```

The following is the equivalent `curl` command:

```
curl -iX POST -H "Content-Type: application/json" -d '{"name": "Noisy Drone",
```

Django won't be able to retrieve a `DroneCategory` instance whose `name` is equal to the specified value: `Octocopter`. Hence, as a result of the previous request, we will receive a `400 Bad Request` status code in the response header and a message related to the value specified for the `drone_category` key in the JSON body. The following lines show a sample response:

```
HTTP/1.0 400 Bad Request
Allow: GET, POST, HEAD, OPTIONS
Content-Length: 66
Content-Type: application/json
Date: Fri, 03 Nov 2017 03:15:07 GMT
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
    "drone_category": [
        "Object with name=Octocopter does not exist."
    ]
}
```

Now, we will compose and send HTTP requests to create two pilots:

```
http POST :8000/pilots/ name="Penelope Pitstop" gender="F" races_count=0
http POST :8000/pilots/ name="Peter Perfect" gender="M" races_count=0
```

The following are the equivalent curl commands:

```
curl -iX POST -H "Content-Type: application/json" -d '{"name": "Penelope Pitstop"
curl -iX POST -H "Content-Type: application/json" -d '{"name": "Peter Perfect
```

The previous commands will compose and send two HTTP `POST` requests with the specified JSON key/value pairs. The request specifies `/pilots/`, and therefore, it will match the `'^Pilots/$'` regular expression and run the `post` method for the `views.PilotList` class-based view.

The following lines show sample responses for the two HTTP requests with the new `Pilot` objects in the JSON responses. Notice that the response includes only the `url`, `url`, for the created pilots and doesn't include the primary key. The value for `gender_description` is the choice description for the `gender` char. The `competitions` array is empty because there aren't competitions related to each new `Pilot` yet. The responses don't include the headers:

```
{  
    "url": "http://localhost:8000/pilots/1",  
    "name": "Penelope Pitstop",  
    "gender": "F",  
    "gender_description": "Female",  
    "races_count": 0,  
    "inserted_timestamp": "2017-11-03T03:22:36.399433Z",  
    "competitions": []  
}  
{  
    "url": "http://localhost:8000/pilots/2",  
    "name": "Peter Perfect",  
    "gender": "M",  
    "gender_description": "Male",  
    "races_count": 0,  
    "inserted_timestamp": "2017-11-03T03:23:02.276186Z",  
    "competitions": []  
}
```

Now, we will compose and send many HTTP POST requests to create three competitions:

```
http POST :8000/competitions/ distance_in_feet=800 distance_achievement_date  
http POST :8000/competitions/ distance_in_feet=2800 distance_achievement_date  
http POST :8000/competitions/ distance_in_feet=790 distance_achievement_date
```

The following are the equivalent curl commands:

```
curl -iX POST -H "Content-Type: application/json" -d '{"distance_in_feet":800}' http://localhost:8000/competitions/  
curl -iX POST -H "Content-Type: application/json" -d '{"distance_in_feet":2800}' http://localhost:8000/competitions/  
curl -iX POST -H "Content-Type: application/json" -d '{"distance_in_feet":790}' http://localhost:8000/competitions/
```

The previous commands will compose and send three HTTP POST requests with the specified JSON key/value pairs. The request specifies /competitions/, and therefore, it will match the '^competitions/\$' regular expression and run the post method for the views.CompetitionList class based view.

The following lines show sample responses for the three HTTP requests

with the new `Competition` objects in the JSON responses. Django REST framework uses the `PilotCompetitionSerializer` class to generate the JSON response. Hence, the value for `drone` is the name for the related `Drone` instance and the value for `pilot` is the name for the related `Pilot` instance. The `PilotCompetitionSerializer` class used `slugRelatedField` for both fields, and therefore, we can specify the names as the values for both the `drone` and `pilot` keys. The responses don't include the headers:

```
{  
    "distance_achievement_date": "2017-10-20T05:03:20.776594Z",  
    "distance_in_feet": 800,  
    "drone": "Atom",  
    "pilot": "Penelope Pitstop",  
    "pk": 1,  
    "url": "http://localhost:8000/competitions/1"  
}  
{  
    "distance_achievement_date": "2017-10-21T06:02:23.776594Z",  
    "distance_in_feet": 2800,  
    "drone": "WonderDrone",  
    "pilot": "Penelope Pitstop",  
    "pk": 2,  
    "url": "http://localhost:8000/competitions/2"  
}  
{  
    "distance_achievement_date": "2017-10-20T05:43:20.776594Z",  
    "distance_in_feet": 790,  
    "drone": "Atom",  
    "pilot": "Peter Perfect",  
    "pk": 3,  
    "url": "http://localhost:8000/competitions/3"  
}
```

We can run the commands explained in the *Analyzing the database* section to check the rows that were inserted in the tables that Django created in the PostgreSQL database to persist the models. The `drone_id` column for the `drones_competition` table saves the value of the primary key of the related row in the `drones_drone` table. In addition, the `pilot_id` column for the `drones_competition` table saves the value of the primary key of the related row in the `drones_pilot` table. The following screenshot uses the `psql` command-line utility to query the contents for the `drones_drone_category`, `drones_drone`, `drones_pilot`, and `drones_competition` tables in a PostgreSQL database after

running the HTTP requests:

```
bin — psql --username=username --dbname=drones — 125x36
drones=> SELECT * FROM drones_dronecategory;
 id |      name
----+-----
  1 | Quadcopter
(1 row)

drones=> SELECT * FROM drones_drone;
 id |      name |      manufacturing_date | has_it_competed |      inserted_timestamp | drone_category_id
----+-----+-----+-----+-----+-----+
  1 | WonderDrone | 2017-07-19 23:02:00.716312-03 | f | 2017-11-02 22:58:49.135737-03 | 1
  2 | Atom | 2017-08-17 23:02:00.716312-03 | f | 2017-11-02 22:59:31.108031-03 | 1
(2 rows)

drones=> SELECT * FROM drones_pilot;
 id |      name | gender | races_count |      inserted_timestamp
----+-----+-----+-----+-----+
  1 | Penelope Pitstop | F | 0 | 2017-11-03 00:22:36.399433-03
  2 | Peter Perfect | M | 0 | 2017-11-03 00:23:02.276186-03
(2 rows)

drones=> SELECT * FROM drones_competition;
 id | distance_in_feet |      distance_achievement_date | drone_id | pilot_id
----+-----+-----+-----+-----+
  1 | 800 | 2017-10-20 02:03:20.776594-03 | 2 | 1
  2 | 2800 | 2017-10-21 03:02:23.776594-03 | 1 | 1
  3 | 790 | 2017-10-20 02:43:20.776594-03 | 2 | 2
(3 rows)

drones=>
```

Now, we will compose and send an HTTP `GET` request to retrieve a specific pilot that participated in two competitions, that is, the pilot resource whose id or primary key is equal to `1`. Don't forget to replace `1` with the primary key value of the Pilot whose name is equal to '`Penelope Pitstop`' in your configuration:

```
http :8000/pilots/1
```

The following is the equivalent curl command:

```
curl -iX GET localhost:8000/Pilots/1
```

The previous commands will compose and send the following HTTP request: `GET http://localhost:8000/Pilots/1`. The request has a number after `/Pilots/`, and therefore, it will match the '`^Pilots/(\?P<pk>[0-9]+)$`' regular expression and run the `get` method for the `views.PilotDetailView` class-based view.

Remember that the `get` method is defined in the `RetrieveUpdateDestroyAPIView`

superclass and it ends up calling the `retrieve` method defined in `mixins.RetrieveModelMixin`. The following lines show a sample response for the HTTP request, with the `pilot` object, the related `competition` objects and the `drone` object related to each `competition` object in the JSON response:

```
HTTP/1.0 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Length: 909
Content-Type: application/json
Date: Fri, 03 Nov 2017 04:40:43 GMT
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
    "competitions": [
        {
            "distance_achievement_date": "2017-10-
                21T06:02:23.776594Z",
            "distance_in_feet": 2800,
            "drone": {
                "drone_category": "Quadcopter",
                "has_it_competed": false,
                "inserted_timestamp": "2017-11-
                    03T01:58:49.135737Z",
                "manufacturing_date": "2017-07-
                    20T02:02:00.716312Z",
                "name": "WonderDrone",
                "url": "http://localhost:8000/drones/1"
            },
            "pk": 2,
            "url": "http://localhost:8000/competitions/2"
        },
        {
            "distance_achievement_date": "2017-10-
                20T05:03:20.776594Z",
            "distance_in_feet": 800,
            "drone": {
                "drone_category": "Quadcopter",
                "has_it_competed": false,
                "inserted_timestamp": "2017-11-
                    03T01:59:31.108031Z",
                "manufacturing_date": "2017-08-
                    18T02:02:00.716312Z",
                "name": "Atom",
                "url": "http://localhost:8000/drones/2"
            },
            "pk": 3,
            "url": "http://localhost:8000/competitions/3"
        }
    ]
}
```

```
        "pk": 1,
        "url": "http://localhost:8000/competitions/1"
    }
],
"gender": "F",
"gender_description": "Female",
"inserted_timestamp": "2017-11-03T03:22:36.399433Z",
"name": "Penelope Pitstop",
"races_count": 0,
"url": "http://localhost:8000/pilots/1"
}
```

The `PilotSerializer` class defined the `competitions` attribute as a `CompetitionSerializer` instance with the `many` argument equal to `True`. Hence, this serializer renders each `competition` related to the `Pilot`. The `CompetitionSerializer` class defined the `drone` attribute as a `DroneSerializer`, and therefore, this serializer renders each `drone` related to the `competition`.

Later, we will render the results in a web browser through the browsable API and we will be able to click or tap on the hyperlink of each of the related resources. However, in this case, we also see all their details without having to follow the hyperlink in the JSON response body.

Test your knowledge

Let's see whether you can answer the following questions correctly.

1. The `related_name` argument for the `django.db.models.ForeignKey` class initializer specifies:
 1. The name to use for the relation from the related object back to this object
 2. The related model class name
 3. The related model primary key attribute name
2. If we use the following line to declare the `pilot` field in the `Competition` model: `pilot = models.ForeignKey(Pilot, related_name='competitions', on_delete=models.CASCADE)`. What will Django's ORM do whenever we delete a specific Pilot?
 1. All the related competitions in which this `Pilot` participated will remain without changes in the database
 2. All the related competitions in which this `Pilot` participated will be deleted too
 3. All the related pilots that are related to the `Competition` will be deleted too
3. The `rest_framework.serializers.HyperlinkedModelSerializer` class is a type of `ModelSerializer` that represents the relationships to other model

instances with:

1. Primary key values
 2. Foreign key values
 3. Hyperlinks
-
4. Which of the following attributes defined in the Meta inner class for a class that inherits from `ModelSerializer` specifies the model related to the serializer that is being declared:
 1. related-model
 2. model
 3. main-model

 5. Which of the following attributes defined in the Meta inner class for a class that inherits from `ModelSerializer` specifies the tuple of string whose values indicate the field names that we want to include in the serialization from the model related to the serializer:
 1. included-fields
 2. fields
 3. serialized-fields

The rights answers are included in the [Appendix](#), *Solutions*.

Summary

In this chapter, we defined the requirements for a complex RESTful Web Service in which we needed to work with drone categories, drones, pilots, and competitions. We created a new app with Django and configured the new web service.

We defined many-to-one relationships between the models with `models.ForeignKey`. We configured Django to work with a PostgreSQL database. We executed migrations that generated tables with relationships between them. We analyzed the generated database and we configured serialization and deserialization for the models. We declared two serializers for a single model to understand the flexibility we have with Django and the Django REST framework.

We defined hyperlinks with the help of the `HyperlinkedModelSerializer` class and we worked with class-based views. In this case, we took advantage of generic classes and generic views that generalize and mix predefined behaviors. We worked with routings and endpoints and we prepared our RESTful Web Service to work with the browsable API. We made many different HTTP requests to create and retrieve resources that have relationships between them. We did everything without writing a huge amount of code.

Now that we understand how to work with class-based views, and to take advantage of generalized behaviors and materialize complex relationships, we will add constraints, filtering, searching, ordering, and pagination features to our RESTful Web Service. We will cover these topics in the next chapter.

Using Constraints, Filtering, Searching, Ordering, and Pagination

In this chapter, we will take advantage of many features included in the Django REST framework to add constraints, pagination, filtering, searching, and ordering features to our RESTful Web Service. We will add a huge amount of features with a few lines of code. We will gain an understanding of:

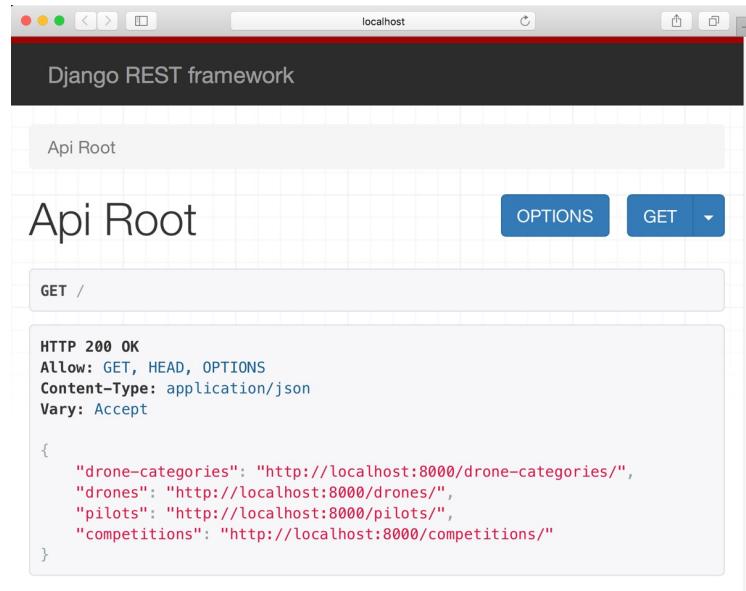
- Browsing the API with resources and relationships
- Defining unique constraints
- Working with unique constraints
- Understanding pagination
- Configuring pagination classes
- Making requests that paginate results
- Working with customized pagination classes
- Making requests that use customized paginated results
- Configuring filter backend classes
- Adding filtering, searching, and ordering
- Working with different types of Django filters
- Making requests that filter results

- Composing requests that filter and order results
- Making requests that perform starts with searches
- Using the browsable API to test pagination, filtering, searching, and ordering

Browsing the API with resources and relationships

We will take advantage of the browsable API feature that we introduced in [Chapter 5, *Understanding and Customizing the Browsable API Feature*](#), with our new web service. Let's start browsing our new RESTful Web Service. Open a web browser and enter `http://localhost:8000`. The browser will compose and send a `GET` request to `/` with `text/html` as the desired content type, and the returned HTML web page will be rendered.

The request will end up executing the `GET` method defined in the `ApiRoot` class within the `views.py` file. The following screenshot shows the rendered web page with the resource description Api Root:



The Api Root renders the following hyperlinks:

- `http://localhost:8000/drone-categories/`: The collection of drone categories

- `http://localhost:8000/drones/`: The collection of drones
- `http://localhost:8000/pilots/`: The collection of pilots
- `http://localhost:8000/competitions/`: The collection of competitions

We can easily access each resource collection by clicking or tapping on the appropriate hyperlink. Once we access each resource collection, we can perform operations on the different resources throughout the browsable API. Whenever we visit any of the resource collections, we can use the breadcrumb to go back to the Api Root that lists all the hyperlinks.

Our new RESTful Web Service takes advantage of many generic views. These views provide many features for the browsable API that weren't included when we worked with function-based views, and we will be able to use forms to easily compose and send HTTP POST requests.

Click or tap on the URL at the right-hand side of drone-categories and the web browser will go to `http://localhost:8000/drone-categories/`. As a result, Django will render the web page for the Drone Category List. At the bottom of the web page, there are two tabs to make an HTTP POST request: Raw data and HTML form. By default, the HTML form tab is activated and displays an automatically generated form with a textbox to enter the value for the Name field to create a new drone category. We can use this form to easily compose and send an HTTP POST request without having to deal with the raw JSON data as we did when working with the browsable API and our previous web service. The following screenshot shows the HTML form to create a new drone category:

The screenshot shows a browsable API interface for a Django REST framework application. At the top, there's a header bar with "Django REST framework". Below it, a navigation bar shows "Api Root" and "Drone Category List". On the right side of the header, there are "OPTIONS" and "GET" buttons. The main content area has a title "Drone Category List" and a sub-section "GET /drone-categories/". Below this, a JSON response is shown:

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "url": "http://localhost:8000/drone-categories/1",
        "pk": 1,
        "name": "Quadcopter",
        "drones": [
            "http://localhost:8000/drones/2",
            "http://localhost:8000/drones/1"
        ]
    }
]
```

At the bottom, there are two buttons: "Raw data" and "HTML form". Under "HTML form", there is a text input field labeled "Name" and a "POST" button.

HTML forms make it really easy to generate requests to test our RESTful web service with the browsable API.

Enter the following value in the Name textbox: `octocopter`. Then, click or tap POST to create a new drone category. The browsable API will compose and send an HTTP `POST` request to `/drone-categories/` with the specified data. Then, we will see the results of this request in the web browser. The following screenshot shows the rendered web page with the results of the previous operation, with an HTTP status code of `201 Created` in the response and the previously explained HTML form with the POST button that allows us to continue composing and sending HTTP `POST` requests to `/drone-categories/`:

The screenshot shows the Django REST framework's API browser interface. At the top, a header bar says "Django REST framework". Below it, a breadcrumb navigation shows "Api Root / Drone Category List". On the right, there are buttons for "OPTIONS", "GET", and a dropdown menu. The main content area has a "POST /drone-categories/" button. Underneath, a JSON response is shown for an "HTTP 201 Created" status:

```

HTTP 201 Created
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Location: http://localhost:8000/drone-categories/2
Vary: Accept

{
    "url": "http://localhost:8000/drone-categories/2",
    "pk": 2,
    "name": "Octocopter",
    "drones": []
}

```

At the bottom, there are two tabs: "Raw data" (selected) and "HTML form". The "HTML form" tab shows a form with a "Name" field containing "Octocopter" and a "Drone category" dropdown. A "POST" button is at the bottom right.

Now, you can go back to the Api Root by clicking on the link on the breadcrumb and use the HTML forms to create drones, pilots, and finally, competitions. For example, go to the Api Root and click or tap on the URL at the right-hand side of drones and the web browser will go to <http://localhost:8000/drones/>. As a result, Django will render the web page for the Drone List. At the bottom of the web page, there are two tabs to make an HTTP POST request: Raw data and HTML form. By default, the HTML form tab is activated and displays an automatically generated form with the appropriate controls for the following fields:

- Name
- Drone category
- Manufacturing date
- Has it competed

The Drone category field provides a drop-down with all the existing drone categories so that we can select one of them for our new drone. The Has it competed field provides a checkbox because the underlying field is

Boolean.

We can use this form to easily compose and send an HTTP `POST` request without having to deal with the raw JSON data as we did when working with the browsable API and our previous web service. The following screenshot shows the HTML form to create a new drone:

The screenshot shows a web-based form for creating a new drone. At the top right, there are two tabs: "Raw data" and "HTML form", with "HTML form" being the active tab. The form itself has the following fields:

- Name:** An input field containing the value "Drone1".
- Drone category:** A dropdown menu set to "Octocopter".
- Manufacturing date:** An input field containing the value "2023-01-01".
- Has it competed:** A checkbox that is checked.

At the bottom right of the form is a blue "POST" button.

Defining unique constraints

The RESTful Web Service doesn't use any constraints, and therefore, it is possible to create many drone categories with the same name. We don't want to have many drone categories with the same name. Each drone category name must be unique in the database table that persists drone categories (the `drones_dronecategory` table). We also want drones and pilots to have unique names. Hence, we will make the necessary changes to add unique constraints to each of the following fields:

- The name field of the `DroneCategory` model
- The name field of the `Drone` model
- The name field of the `Pilot` model

We will learn the necessary steps to edit existing models and add constraints to fields that are already persisted in tables and to propagate the changes in the underlying database by running the already analyzed migrations process.

Make sure you quit Django's development server. Remember that you just need to press `Ctrl + C` in the terminal or Command Prompt window in which it is running. We have to edit the models and then execute migrations before starting Django's development server again.

Now, we will edit the existing code that declares the models to add unique constraints to the `name` field for the models that we use to represent and persist the drone categories, drones, and pilots. Open the `drones/models.py` file and replace the code that declares the `DroneCategory`, `Drone`, and `Pilot` classes with the following code. The lines that were edited are highlighted in the code listing. The code for the `competition` class remains without

changes. The code file for the sample is included in the `hillar_django_restful_07_01` folder, in the `restful01/drones/models.py` file:

```
class DroneCategory(models.Model):
    name = models.CharField(max_length=250, unique=True)

    class Meta:
        ordering = ('name',)

    def __str__(self):
        return self.name


class Drone(models.Model):
    name = models.CharField(max_length=250, unique=True)
    drone_category = models.ForeignKey(
        DroneCategory,
        related_name='drones',
        on_delete=models.CASCADE)
    manufacturing_date = models.DateTimeField()
    has_it_competed = models.BooleanField(default=False)
    inserted_timestamp = models.DateTimeField(auto_now_add=True)

    class Meta:
        ordering = ('name',)

    def __str__(self):
        return self.name


class Pilot(models.Model):
    MALE = 'M'
    FEMALE = 'F'
    GENDER_CHOICES = (
        (MALE, 'Male'),
        (FEMALE, 'Female'),
    )
    name = models.CharField(max_length=150, blank=False, unique=True)
    gender = models.CharField(
        max_length=2,
        choices=GENDER_CHOICES,
        default=MALE,
    )
    races_count = models.IntegerField()
    inserted_timestamp = models.DateTimeField(auto_now_add=True)

    class Meta:
        ordering = ('name',)
```

```
def __str__(self):  
    return self.name
```

We added `unique=True` as one of the named arguments for each call to the `models.CharField` initializer. This way, we specify that the fields must be unique, and Django's ORM will translate this into a requirement for the creation of the necessary unique constraints for the fields in the underlying database tables.

Now, it is necessary to execute the migrations that will generate the unique constraints we added for the fields in the models in the database. This time, the migrations process will synchronize the database with the changes we made in the models, and therefore, the process will apply a delta. Run the following Python script:

```
python manage.py makemigrations drones
```

The following lines show the output generated after running the previous command:

```
Migrations for 'drones':  
drones/migrations/0002_auto_20171104_0246.py  
- Alter field name on drone  
- Alter field name on dronecategory  
- Alter field name on pilot
```

The lines in the output indicate that the `drones/migrations/0002_auto_20171104_0246.py` file includes the code to alter the fields called `name` on `drone`, `dronecategory`, and `pilot`. It is important to take into account that the Python filename generated by the migrations process encodes the date and time, and therefore, the name will be different when you run the code in your development computer.

The following lines show the code for the file that was automatically generated by Django. The code file for the sample is included in the

hillar_django_restful_07_01 folder, in the
restful01/drones/migrations/0002_auto_20171104_0246.py file:

```
# -*- coding: utf-8 -*-
# Generated by Django 1.11.5 on 2017-11-04 02:46
from __future__ import unicode_literals

from django.db import migrations, models


class Migration(migrations.Migration):

    dependencies = [
        ('drones', '0001_initial'),
    ]

    operations = [
        migrations.AlterField(
            model_name='drone',
            name='name',
            field=models.CharField(max_length=250, unique=True),
        ),
        migrations.AlterField(
            model_name='dronecategory',
            name='name',
            field=models.CharField(max_length=250, unique=True),
        ),
        migrations.AlterField(
            model_name='pilot',
            name='name',
            field=models.CharField(max_length=50, unique=True),
        ),
    ]

```

The code defines a subclass of the `django.db.migrations.Migration` class called `Migration`, which defines an `operations` list with many `migrations.AlterField` instances. Each `migrations.AlterField` instance will alter the field in the table for each of the related models: `drone`, `dronecategory`, and `pilot`.

Now, run the following Python script to execute all the generated migrations and apply the changes in the underlying database tables:

```
| python manage.py migrate
```

The following lines show the output generated after running the previous command. Notice that the order in which the migrations are executed can differ in your development computer:

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, drones, sessions
Running migrations:
  Applying drones.0002_auto_20171104_0246... OK
```

After we run the previous command, we will have unique indexes on the `name` fields for the following tables in the PostgreSQL database:

- `drones_drone`
- `drones_dronecategory`
- `drones_pilot`

We can use the PostgreSQL command-line tools or any other application that allows us to easily check the contents of the PostgreSQL database to check the tables that Django updated. If you are working with an SQLite or any other database with this example, make sure you use the commands or tools related to the database you are using.

The following screenshot shows a list of the indexes for each of the previously enumerated tables in the SQLPro for Postgres GUI tool. Each table has a new unique index for the `name` field:

drones_drone	
▶	id integer, not null
▶	name character varying(250), not null
▶	manufacturing_date timestamp with time zone, not null
▶	has_it_competed boolean, not null
▶	inserted_timestamp timestamp with time zone, not null
▶	drone_category_id integer, not null
▼	Indexes
▶	drones_drone_pkey
▶	drones_drone_name_85faecee_uniq
▶	drones_drone_name_85faecee_like
▶	drones_drone_drone_category_id_214d98cf
▼	Triggers
▼	
drones_dronecategory	
▶	id integer, not null
▶	name character varying(250), not null
▼	Indexes
▶	drones_dronecategory_pkey
▶	drones_dronecategory_name_dedead86_uniq
▶	drones_dronecategory_name_dedead86_like
▶	Triggers
▼	
drones_pilot	
▶	id integer, not null
▶	name character varying(50), not null
▶	gender character varying(2), not null
▶	races_count integer, not null
▶	inserted_timestamp timestamp with time zone, not null
▼	Indexes
▶	drones_pilot_pkey
▶	drones_pilot_name_3b56f2a1_uniq
▶	drones_pilot_name_3b56f2a1_like

The following are the names generated for the new unique indexes in the sample database:

- The `drones_drone_name_85faecee_uniq` index for the `drones_drone` table
- The `drones_drone_dronecategory_name_dedead86_uniq` index for the `drones_dronecategory` table
- The `drones_pilot_name_3b56f2a1_uniq` index for the `drones_pilot` table

Working with unique constraints

Now, we can launch Django's development server to compose and send HTTP requests to understand how unique constraints work when applied to our models. Execute any of the following two commands, based on your needs, to access the API in other devices or computers connected to your LAN. Remember that we analyzed the difference between them in chapter 3, *Creating API Views*, in the *Launching Django's development server* section:

```
python manage.py runserver  
python manage.py runserver 0.0.0.0:8000
```

After we run any of the previous commands, the development server will start listening at port 8000.

Now, we will compose and send an HTTP request to create a drone category with a name that already exists: 'quadcopter', as shown below:

```
http POST :8000/drone-categories/ name="Quadcopter"
```

The following is the equivalent curl command:

```
curl -iX POST -H "Content-Type: application/json" -d '{"name": "Quadcopter"}' 1
```

Django won't be able to persist a `DroneCategory` instance whose `name` is equal to the specified value because it violates the unique constraint we just added to the `name` field for the `DroneCategory` model. As a result of the request,

we will receive a `400 Bad Request` status code in the response header and a message related to the value specified for the `name` field in the JSON body: "drone category with this name already exists." The following lines show the detailed response:

```
HTTP/1.0 400 Bad Request
Allow: GET, POST, HEAD, OPTIONS
Content-Length: 58
Content-Type: application/json
Date: Sun, 05 Nov 2017 04:00:42 GMT
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
    "name": [
        "drone category with this name already exists."
    ]
}
```

We made the necessary changes to avoid duplicate values for the `name` field in drone categories, drones, or pilots. Whenever we specify the name for any of these resources, we will be referencing the same unique resource, because duplicates aren't possible.

Now, we will compose and send an HTTP request to create a pilot with a name that already exists: 'Penelope Pitstop', as shown below:

```
http POST :8000/pilots/ name="Penelope Pitstop" gender="F"
races_count=0
```

The following is the equivalent `curl` command:

```
curl -iX POST -H "Content-Type: application/json" -d
'{"name":"Penelope Pitstop", "gender":"F", "races_count": 0}'
localhost:8000/pilots/
```

The previous command will compose and send an HTTP `POST` request with the specified JSON key-value pairs. The request specifies `/pilots/`, and therefore, it will match the '`^pilots/$`' regular expression and will run the `post` method for the `views.PilotList` class-based view. Django won't be able to persist a `Pilot` instance whose `name` is equal to the specified value because it violates the unique constraint we just added to the `name` field for the `Pilot` model. As a result of the request, we will receive a `400 Bad Request` status code in the response header and a message related to the value specified for the `name` field in the JSON body: "pilot with this name already exists." The following lines show the detailed response:

```
HTTP/1.0 400 Bad Request
Allow: GET, POST, HEAD, OPTIONS
Content-Length: 49
Content-Type: application/json
Date: Sun, 05 Nov 2017 04:13:37 GMT
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
    "name": [
        "pilot with this name already exists."
    ]
}
```

If we generate the HTTP `POST` request with the help of the HTML form in the browsable API, we will see the error message displayed below the Name field in the form, as shown in the next screenshot:

Django REST framework

Pilot List

OPTIONS GET ▾

POST /pilots/

```
HTTP 400 Bad Request
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "name": [
        "pilot with this name already exists."
    ]
}
```

Raw data HTML form

Name pilot with this name already exists.

Gender

Races count

POST

Understanding pagination

So far, we have been working with a database that has just a few rows, and therefore, the HTTP `GET` requests to the different resource collections for our RESTful Web Service don't have problems with the amount of data in the JSON body of the responses. However, this situation changes as the number of rows in the database tables increases.

Let's imagine we have 300 rows in the `drones_pilots` table that persists pilots. We don't want to retrieve the data for 300 pilots whenever we make an HTTP `GET` request to `localhost:8000/pilots/`. Instead, we just take advantage of the pagination features available in the Django REST framework to make it easy to specify how we want the large result sets to be split into individual pages of data. This way, each request will retrieve only one page of data, instead of the entire result set. For example, we can make the necessary configurations to retrieve only the data for a page of a maximum of four pilots.

Whenever we enable a pagination scheme, the HTTP `GET` requests must specify the pieces of data that they want to retrieve, that is, the details for the specific pages, based on predefined pagination schemes. In addition, it is extremely useful to have data about the total number of resources, the next page, and the previous one, in the response body. This way, the user or the application that is consuming the RESTful Web Service knows the additional requests that need to be made to retrieve the required pages.

We can work with page numbers and the client can request a specific page number in the HTTP `GET` request. Each page will include a maximum amount of resources. For example, if we request the first page for the 300 pilots, the web service will return the first four pilots in the response body. The second page will return the pilots from the fifth to the eighth position in the response body.

Another option is to specify an offset combined with a limit. For example, if we request a page with an offset equal to 0 and a limit of 4, the web service will return the first four pilots in the response body. A second request with an offset equal to 4 and a limit of 4 will return the pilots from the fifth to the eighth position in the response body.

Right now, each of the database tables that persist the models we have defined has a few rows. However, after we start working with our web service in a real-life production environment, we will have hundreds of competitions, pilots, drones, and drone categories. Hence, we will definitely have to deal with large result sets. We will usually have the same situation in most RESTful Web Services, and therefore, it is very important to work with pagination mechanisms.

Configuring pagination classes

The Django REST framework provides many options to enable pagination. First, we will set up one of the customizable pagination styles included in the Django REST framework to include a maximum of four resources in each individual page of data.

Our RESTful Web Service uses the generic views that work with **mixin** classes. These classes are prepared to build paginated responses based on specific settings in the Django REST framework configuration. Hence, our RESTful Web Service will automatically take into account the pagination settings we configured, without requiring additional changes in the code.

Open the `restful01/restful01/settings.py` file that declares module-level variables that define the configuration of Django for the `restful01` project. We will make some changes to this Django settings file. The code file for the sample is included in the `hillar_django_restful_07_01` folder, in the `restful01/restful01/settings.py` file. Add the following lines that declare a dictionary named `REST_FRAMEWORK` with key-value pairs that configure the global pagination settings:

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
        'rest_framework.pagination.LimitOffsetPagination',
    'PAGE_SIZE': 4
}
```

Save the changes and Django's development server will recognize the edits and start again with the new pagination settings enabled. The new dictionary has two string keys: `'DEFAULT_PAGINATION_CLASS'` and `'PAGE_SIZE'`. The value for the `'DEFAULT_PAGINATION_CLASS'` key specifies a global setting with the default pagination class that the generic views will use to provide paginated responses. In this case, we will use the

`rest_framework.pagination.LimitOffsetPagination` class that provides a limit/offset-based style.

This pagination style works with a `limit` parameter that indicates the maximum number of items to return and an `offset` that specifies the starting position of the query. The value for the `PAGE_SIZE` settings key specifies a global setting with the default value for the `limit`, also known as the page size. In this case, the value is set to 4, and therefore, the maximum number of resources returned in a single request will be four. We can specify a different limit when we perform the HTTP request by specifying the desired value in the `limit` query parameter. We can configure the class to have a maximum `limit` value in order to avoid undesired huge result sets. This way, we can make sure that the user won't be able to specify a large number for the `limit` value. However, we will make this specific configuration later.

Now, we will compose and send many HTTP `POST` requests to create nine additional drones related to the two drone categories we created: `Quadcopter` and `Octocopter`. This way, we will have a total of 11 drones (two existing drones, plus nine additional drones) to test the limit/offset pagination mechanism we have enabled:

```
http POST :8000/drones/ name="Need for Speed" drone_category="Quadcopter" ma
http POST :8000/drones/ name="Eclipse" drone_category="Octocopter" manufactu
http POST :8000/drones/ name="Gossamer Albatross" drone_category="Quadcopter"
http POST :8000/drones/ name="Dassault Falcon 7X" drone_category="Octocopter"
http POST :8000/drones/ name="Gulfstream I" drone_category="Quadcopter" manu
http POST :8000/drones/ name="RV-3" drone_category="Octocopter" manufacturin
http POST :8000/drones/ name="Dusty" drone_category="Quadcopter" manufacturi
http POST :8000/drones/ name="Ripslinger" drone_category="Octocopter" manufa
http POST :8000/drones/ name="Skipper" drone_category="Quadcopter" manufactu
```

The following are the equivalent `curl` commands:

```
curl -iX POST -H "Content-Type: application/json" -d '{"name": "Need for Speed",
curl -iX POST -H "Content-Type: application/json" -d '{"name": "Eclipse", "dr
curl -iX POST -H "Content-Type: application/json" -d '{"name": "Gossamer Alba
curl -iX POST -H "Content-Type: application/json" -d '{"name": "Dassault Falc
```

```
curl -iX POST -H "Content-Type: application/json" -d '{"name":"Gulfstream I"}'
curl -iX POST -H "Content-Type: application/json" -d '{"name":"RV-3", "drone": "I"}'
curl -iX POST -H "Content-Type: application/json" -d '{"name":"Dusty", "drone": "II"}'
curl -iX POST -H "Content-Type: application/json" -d '{"name":"Ripslinger", "drone": "III"}'
curl -iX POST -H "Content-Type: application/json" -d '{"name":"Skipper", "drone": "IV"}'
```

The previous commands will compose and send nine HTTP `POST` requests with the specified JSON key-value pairs. The requests specify `/drones/`, and therefore, they will match the `'^drones/$'` regular expression and run the `post` method for the `views.DroneList` class-based view.

Making requests that paginate results

Now, we will compose and send an HTTP `GET` request to retrieve all the drones. The new pagination settings will take effect and we will only retrieve the first page for the drones resource collection:

```
http GET :8000/drones/
```

The following is the equivalent `curl` command:

```
curl -iX GET localhost:8000/drones/
```

The previous commands will compose and send an HTTP `GET` request. The request specifies `/drones/`, and therefore, it will match the '`^drones/$`' regular expression and run the `get` method for the `views.DroneList` class-based view. The method executed in the generic view will use the new settings we added to enable the offset/limit pagination, and the result will provide us with the first four drone resources. However, the response body looks different than in the previous HTTP `GET` requests we made to any resource collection. The following lines show the sample response that we will analyze in detail. Don't forget that the drones are being sorted by the name field, in ascending order:

```
HTTP/1.0 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Length: 958
Content-Type: application/json
Date: Mon, 06 Nov 2017 23:08:36 GMT
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Accept, Cookie
```

```
X-Frame-Options: SAMEORIGIN

{
  "count": 11,
  "next": "http://localhost:8000/drones/?limit=4&offset=4",
  "previous": null,
  "results": [
    {
      "drone_category": "Quadcopter",
      "has_it_competed": false,
      "inserted_timestamp": "2017-11-03T01:59:31.108031Z",
      "manufacturing_date": "2017-08-18T02:02:00.716312Z",
      "name": "Atom",
      "url": "http://localhost:8000/drones/2"
    },
    {
      "drone_category": "Octocopter",
      "has_it_competed": false,
      "inserted_timestamp": "2017-11-06T20:25:30.357127Z",
      "manufacturing_date": "2017-04-18T02:02:00.716312Z",
      "name": "Dassault Falcon 7X",
      "url": "http://localhost:8000/drones/6"
    },
    {
      "drone_category": "Quadcopter",
      "has_it_competed": false,
      "inserted_timestamp": "2017-11-06T20:25:31.049833Z",
      "manufacturing_date": "2017-07-20T02:02:00.716312Z",
      "name": "Dusty",
      "url": "http://localhost:8000/drones/9"
    },
    {
      "drone_category": "Octocopter",
      "has_it_competed": false,
      "inserted_timestamp": "2017-11-06T20:25:29.909965Z",
      "manufacturing_date": "2017-02-18T02:02:00.716312Z",
      "name": "Eclipse",
      "url": "http://localhost:8000/drones/4"
    }
  ]
}
```

The response has a `200 OK` status code in the header and the following keys in the response body:

- `count`: The value indicates the total number of drones for the query.

- `next`: The value provides a link to the next page.
- `previous`: The value provides a link to the previous page. In this case, the response includes the first page of the result set, and therefore, the link to the previous page is `null`.
- `results`: The value provides an array of JSON representations of `Drone` instances that compose the requested page. In this case, the four drones belong to the first page of the result set.

In the previous HTTP `GET` request, we didn't specify any values for either the `limit` or `offset` parameters. We specified `4` as the default value for the `limit` parameter in the global settings and the generic views use this configuration value and provide us with the first page. Whenever we don't specify any `offset` value, the default `offset` is equal to `0` and the `get` method will return the first page.

The previous request is equivalent to the following HTTP `GET` request that specifies `0` for the `offset` value. The result of the next command will be the same as the previous one:

```
http GET ":8000/drones/?offset=0"
```

The following is the equivalent `curl` command:

```
curl -iX GET "localhost:8000/drones/?offset=0"
```

The previous requests are equivalent to the following HTTP `GET` request that specifies `0` for the offset value and `4` for the limit value. The result of the next command will be the same as the previous two commands:

```
http GET ":8000/drones/?limit=4&offset=0"
```

The following is the equivalent `curl` command:

```
curl -iX GET "localhost:8000/drones/?limit=4&offset=0"
```

Now, we will compose and send an HTTP request to retrieve the next page, that is, the second page for the drones. We will use the value for the `next` key provided in the JSON body of the response from the previous requests. This value gives us the URL for the next page:

`http://localhost:8000/drones/?limit=4&offset=4`. Thus, we will compose and send an HTTP `GET` method to `/drones/` with the limit value set to 4 and the offset value set to 4 :

```
http GET ":8000/drones/?limit=4&offset=4"
```

The following is the equivalent `curl` command:

```
curl -iX GET "localhost:8000/drones/?limit=4&offset=4"
```

The result will provide us the second page of four drone resources as the value for the `results` key in the response body. In addition, we will see the values for the `count`, `previous`, and `next` keys that we analyzed in the previous requests. The following lines show the sample response:

```
HTTP/1.0 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Length: 1007
Content-Type: application/json
Date: Mon, 06 Nov 2017 23:31:34 GMT
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
    "count": 11,
    "next": "http://localhost:8000/drones/?limit=4&offset=8",
```

```
"previous": "http://localhost:8000/drones/?limit=4",
"results": [
  {
    "drone_category": "Quadcopter",
    "has_it_competed": false,
    "inserted_timestamp": "2017-11-06T20:25:30.127661Z",
    "manufacturing_date": "2017-03-20T02:02:00.716312Z",
    "name": "Gossamer Albatross",
    "url": "http://localhost:8000/drones/5"
  },
  {
    "drone_category": "Quadcopter",
    "has_it_competed": false,
    "inserted_timestamp": "2017-11-06T20:25:30.584031Z",
    "manufacturing_date": "2017-05-20T02:02:00.716312Z",
    "name": "Gulfstream I",
    "url": "http://localhost:8000/drones/7"
  },
  {
    "drone_category": "Quadcopter",
    "has_it_competed": false,
    "inserted_timestamp": "2017-11-06T20:25:29.636153Z",
    "manufacturing_date": "2017-01-20T02:02:00.716312Z",
    "name": "Need for Speed",
    "url": "http://localhost:8000/drones/3"
  },
  {
    "drone_category": "Octocopter",
    "has_it_competed": false,
    "inserted_timestamp": "2017-11-06T20:25:30.819695Z",
    "manufacturing_date": "2017-06-18T02:02:00.716312Z",
    "name": "RV-3",
    "url": "http://localhost:8000/drones/8"
  }
]
```

In this case, the result set is the second page, and therefore, we have a value for the `previous` key: `http://localhost:8000/drones/?limit=4`.

In the previous HTTP request, we specified values for both the `limit` and `offset` parameters. However, as we set the default value of `limit` to `4` in the global settings, the following request will produce the same results as the previous request:

```
http GET ":8000/drones/?offset=4"
```

The following is the equivalent `curl` command:

```
curl -iX GET "localhost:8000/drones/?offset=4"
```

Now, we will compose and send an HTTP request to retrieve the next page, that is, the third and last page for the drones. We will use the value for the `next` key provided in the JSON body of the response from the previous requests. This value gives us the URL for the next page

as `http://localhost:8000/drones/?limit=4&offset=8`. Thus, we will compose and send an HTTP `GET` method to `/drones/` with the limit value set to 4 and the offset value set to 8 :

```
http GET ":8000/drones/?limit=4&offset=8"
```

The following is the equivalent `curl` command:

```
curl -iX GET "localhost:8000/drones/?limit=4&offset=8"
```

The result will provide us with the third and last page of three drone resources as the value for the `results` key in the response body. In addition, we will see the values for the `count`, `previous`, and `next` keys that we analyzed in the previous requests. The following lines show the sample response:

```
HTTP/1.0 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Length: 747
Content-Type: application/json
Date: Tue, 07 Nov 2017 02:59:42 GMT
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN
```

```
{  
    "count": 11,  
    "next": null,  
    "previous": "http://localhost:8000/drones/?limit=4&offset=4",  
    "results": [  
        {  
            "drone_category": "Octocopter",  
            "has_it_competed": false,  
            "inserted_timestamp": "2017-11-06T20:25:31.279172Z",  
            "manufacturing_date": "2017-08-18T02:02:00.716312Z",  
            "name": "Ripslinger",  
            "url": "http://localhost:8000/drones/10"  
        },  
        {  
            "drone_category": "Quadcopter",  
            "has_it_competed": false,  
            "inserted_timestamp": "2017-11-06T20:25:31.511881Z",  
            "manufacturing_date": "2017-09-20T02:02:00.716312Z",  
            "name": "Skipper",  
            "url": "http://localhost:8000/drones/11"  
        },  
        {  
            "drone_category": "Quadcopter",  
            "has_it_competed": false,  
            "inserted_timestamp": "2017-11-03T01:58:49.135737Z",  
            "manufacturing_date": "2017-07-20T02:02:00.716312Z",  
            "name": "WonderDrone",  
            "url": "http://localhost:8000/drones/1"  
        }  
    ]  
}
```

In this case, the result set is the last page, and therefore, we have `null` as the value for the `next` key.

Working with customized pagination classes

We enabled pagination to limit the size for the result sets. However, any client or user is able to specify a large number for the `limit` value, such as `10000`, and generate a huge result set. In order to specify the maximum number that is accepted for the `limit` query parameter, it is necessary to create a customized version of the `limit/offset` pagination scheme that the Django REST framework provides us.

We made changes to the global configuration to use the `rest_framework.pagination.LimitOffsetPagination` class to handle paginated responses. This class declares a `max_limit` class attribute whose default value is equal to `None`, which means there is no upper bound for the `limit` value. We will indicate the upper bound value for the `limit` query parameter in the `max_limit` class attribute.

Make sure you quit Django's development server. Remember that you just need to press `Ctrl + C` in the terminal or Command Prompt in which it is running.

Go to the `restful01/drones` folder and create a new file named `custompagination.py`. Write the following code in this new file. The following lines show the code for this file that declares the new `LimitOffsetPaginationWithUpperBound` class. The code file for the sample is included in the `hillar_django_restful_07_02` folder in the `restful01/drones/custompagination.py` file:

```
from rest_framework.pagination import LimitOffsetPagination
class LimitOffsetPaginationWithUpperBound(LimitOffsetPagination):
    # Set the maximum limit value to 8
    max_limit = 8
```

The previous lines declare the `LimitOffsetPaginationWithUpperBound` class as a subclass of `rest_framework.pagination.LimitOffsetPagination`. This new class overrides the value assigned to the `max_limit` class attribute with 8.

Open the `restful01/restful01/settings.py` file and replace the line that specifies the value for the `DEFAULT_PAGINATION_CLASS` key in the `REST_FRAMEWORK` dictionary with the highlighted line. The following lines show the new declaration of the `REST_FRAMEWORK` dictionary. The code file for the sample is included in the `hillar_django_restful_07_02` folder in the `restful01/restful01/settings.py` file:

```
REST_FRAMEWORK = {  
    'DEFAULT_PAGINATION_CLASS':  
        'drones.custompagination.LimitOffsetPaginationWithUpperBound',  
    'PAGE_SIZE': 4  
}
```

This way, all the generic views will use the recently declared `drones.custompagination.LimitOffsetPaginationWithUpperBound` class that provides the limit/offset pagination scheme we have analyzed with an upper bound for the `limit` value equal to 8.

If any request specifies a value higher than 8 for the limit, the class will use the maximum limit value, that is, 8, and the RESTful Web Service will never return more than eight resources in a paginated response.

It is a good practice to configure a maximum limit to avoid generating responses with huge amounts of data that might generate important loads to the server running the RESTful Web Service. Note that we will learn to limit the usage of the resources of our RESTful Web Service in the forthcoming chapters. Pagination is just the beginning of a long story.

Making requests that use customized paginated results

Launch Django's development server. If you don't remember how to start Django's development server, check the instructions in [Chapter 3, *Creating API Views*](#), in the *Launching Django's development server* section.

Now, we will compose and send an HTTP `GET` request to retrieve the first page for the drones with the value for the `limit` query parameter set to `500`. This value is higher than the maximum limit we established:

```
http GET ":8000/drones/?limit=500"
```

The following is the equivalent `curl` command:

```
curl -iX GET "localhost:8000/drones/?limit=500"
```

The code in the `get` method for the `views.DroneList` class-based view will use the new settings we added to enable the customized offset/limit pagination, and the result will provide us with the first eight drone resources because the maximum value for the `limit` query is set to `8`. The value specified for the `limit` query parameter is greater than `8`, and therefore, the maximum value of `8` is used, instead of the value indicated in the request.

The key advantage of working with generic views is that we can easily customize the behavior for the methods defined in the mixins that compose these views with just a few lines of code. In this case, we took advantage of the pagination features available in the Django REST framework to specify

how we wanted large results sets to be split into individual pages of data. Then, we customized paginated results with just a few lines of code to make the limit/offset pagination scheme match our specific requirements.

Configuring filter backend classes

So far, we have been working with the entire queryset as the result set. For example, whenever we requested the drones resource collection, the RESTful Web Service worked with the entire resource collection and used the default sorting we had configured in the model. Now, we want our RESTful Web Service to be able to provide filtering, searching, and sorting features.

It is very important to understand that we have to be careful with the fields we configure to be available in the filtering, searching, and ordering features. The configuration will have an impact on the queries executed on the database, and therefore, we must make sure that we have the appropriate database optimizations, considering the queries that will be executed. Specific database optimizations are outside of the scope of this book, but you definitely must take them into account when you configure these features.

Make sure you quit Django's development server. Remember that you just need to press *Ctrl + C* in the terminal or Command Prompt window in which it is running.

Run the following command to install the `django-filter` package in our virtual environment. This package will enable us to use many field filtering features that we can easily customize in the Django REST framework. Make sure the virtual environment is activated, and run the following command:

```
pip install django-filter
```

The last lines of the output will indicate that the `django-filter` package has been successfully installed:

```
Collecting django-filter
  Downloading django_filter-1.1.0-py2.py3-none-any.whl
    Installing collected packages: django-filter
      Successfully installed django-filter-1.1.0
```

We will work with the following three classes:

- `rest_framework.filters.OrderingFilter`: This class allows the client to control how the results are ordered with a single query parameter. We can specify which fields may be ordered against.
- `django_filters.rest_framework.DjangoFilterBackend`: This class provides field filtering capabilities. We can specify the set of fields we want to be able to filter against, and the filter backend defined in the `django-filter` package will create a new `django_filters.rest_framework.FilterSet` class and associate it to the class-based view. It is also possible to create our own `rest_framework.filters.FilterSet` class, with more customized settings, and write our own code to associate it with the class-based view.
- `rest_framework.filters.SearchFilter`: This class provides single query parameter-based searching capabilities, and its behavior is based on the Django admin's search function. We can specify the set of fields we want to include for the search feature and the client will be able to filter items by making queries that search on these fields with a single query. It is useful when we want to make it possible for a request to search on multiple fields with a single query.

It is possible to configure the filter backends by including any of the

previously enumerated classes in a tuple and assigning it to the `filter_backends` class attribute for the generic view classes. In our RESTful Web Service, we want all our class-based views to use the same filter backends, and therefore, we will make changes in the global configuration.

Open the `restful01/restful01/settings.py` file that declares module-level variables that define the configuration of Django for the `restful01` project. We will make some changes to this Django settings file. Add the highlighted lines that declare the `'DEFAULT_FILTER_BACKENDS'` key and assign a tuple of strings as its value with the three classes we have analyzed. The following lines show the new declaration of the `REST_FRAMEWORK` dictionary. The code file for the sample is included in the `hillar_django_restful_07_03` folder in the `restful01/restful01/settings.py` file:

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
        'drones.custompagination.LimitOffsetPaginationWithUpperBound',
    'PAGE_SIZE': 4,
    'DEFAULT_FILTER_BACKENDS': (
        'django_filters.rest_framework.DjangoFilterBackend',
        'rest_framework.filters.OrderingFilter',
        'rest_framework.filters.SearchFilter',
    ),
}
```

Locate the lines that assign a string list to `INSTALLED_APPS` to declare the installed apps. Add the following string to the `INSTALLED_APPS` string list and save the changes to the `settings.py` file:

```
'django_filters',
```

The following lines show the new code that declares the `INSTALLED_APPS` string list with the added line highlighted and with comments to understand what each added string means. The code file for the sample is included in the `hillar_django_restful_07_03` folder in the `restful01/restful01/settings.py` file:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Django REST Framework
    'rest_framework',
    # Drones application
    'drones.apps.DronesConfig',
    # Django Filters,
    'django_filters',
]
```

This way, we have added the `django_filters` application to our Django project named `restful01`.

The default query parameter names are `search` for the search feature and `ordering` for the ordering feature. We can specify other names by setting the desired strings in the `SEARCH_PARAM` and the `ORDERING_PARAM` settings. In this case, we will work with the default values.

Adding filtering, searching, and ordering

Now, we will add the necessary code to configure the fields that we want to be included in the filtering, searching, and ordering features for each of the class-based views that retrieve the contents of each resource collection. Hence, we will make changes to all the classes with the `List` suffix in the `views.py` file: `DroneCategoryList`, `DroneList`, `PilotList`, and `CompetitionList`.

We will declare the following three class attributes in each of those classes:

- `filter_fields`: This attribute specifies a tuple of strings whose values indicate the field names that we want to be able to filter against. Under the hood, the Django REST framework will automatically create a `rest_framework.filters.FilterSet` class and associate it to the class-based view in which we are declaring the attribute. We will be able to filter against the field names included in the tuple of strings.
- `search_fields`: This attribute specifies a tuple of strings whose values indicate the text type field names that we want to include in the search feature. In all the usages, we will want to perform a starts-with match. In order to do this, we will include '^' as a prefix of the field name to indicate that we want to restrict the search behavior to a starts-with match.
- `ordering_fields`: This attribute specifies a tuple of strings whose values indicate the field names that the HTTP request can specify

to sort the results. If the request doesn't specify a field for ordering, the response will use the default ordering fields specified in the model that is related to the class-based view.

Open the `restful01/drones/views.py` file. Add the following code after the last line that declares the imports, before the declaration of the `DroneCategoryList` class. The code file for the sample is included in the `hillar_django_restful_07_03` folder in the `restful01/drones/views.py` file:

```
from rest_framework import filters
from django_filters import AllValuesFilter, DateTimeFilter, NumberFilter
```

Add the following highlighted lines to the `DroneList` class declared in the `views.py` file. The next lines show the new code that defines the class. The code file for the sample is included in the `hillar_django_restful_07_03` folder in the `restful01/drones/views.py` file:

```
class DroneCategoryList(generics.ListCreateAPIView):
    queryset = DroneCategory.objects.all()
    serializer_class = DroneCategorySerializer
    name = 'dronecategory-list'
    filter_fields = (
        'name',
    )
    search_fields = (
        '^name',
    )
    ordering_fields = (
        'name',
    )
```

The changes in the `DroneList` class are easy to understand. We will be able to filter, search, and order by the `name` field.

Add the following highlighted lines to the `DroneList` class declared in the `views.py` file. The next lines show the new code that defines the class. The

code file for the sample is included in the `hillar_django_restful_07_03` folder in the `restful01/drones/views.py` file:

```
class DroneList(generics.ListCreateAPIView):
    queryset = Drone.objects.all()
    serializer_class = DroneSerializer
    name = 'drone-list'
    filter_fields = (
        'name',
        'drone_category',
        'manufacturing_date',
        'has_it_competed',
    )
    search_fields = (
        '^name',
    )
    ordering_fields = (
        'name',
        'manufacturing_date',
    )
```

In the `DroneList` class, we specified many field names in the `filter_fields` attribute. We included '`drone_category`' in the string tuple, and therefore, we will be able to include the ID values for this field in the filter.

We will take advantage of other options for related models that will allow us to filter by fields of the related model later. This way, we will understand the different customizations available.

The `ordering_fields` attribute specifies two field names for the tuple of strings, and therefore, we will be able to order the results by either `name` or `manufacturing_date`. Don't forget that we must take into account database optimizations when enabling fields to order by.

Add the following highlighted lines to the `PilotList` class declared in the `views.py` file. The next lines show the new code that defines the class. The code file for the sample is included in the `hillar_django_restful_07_03` folder in the `restful01/drones/views.py` file:

```
class PilotList(generics.ListCreateAPIView):
    queryset = Pilot.objects.all()
    serializer_class = PilotSerializer
    name = 'pilot-list'
    filter_fields = (
        'name',
        'gender',
        'races_count',
    )
    search_fields = (
        '^name',
    )
    ordering_fields = (
        'name',
        'races_count'
    )
```

The `ordering_fields` attribute specifies two field names for the tuple of strings, and therefore, we will be able to order the results by either `name` or `races_count`.

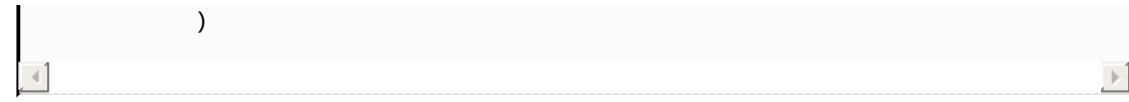
Working with different types of Django filters

Now, we will create a customized filter that we will apply to the `Competition` model. We will code the new `CompetitionFilter` class, specifically, a subclass of the `rest_framework.filters.FilterSet` class.

Open the `restful01/drones/views.py` file. Add the following code before the declaration of the `CompetitionList` class. The code file for the sample is included in the `hillar_django_restful_07_03` folder in the `restful01/drones/views.py` file:

```
class CompetitionFilter(filters.FilterSet):
    from_achievement_date = DateTimeFilter(
        name='distance_achievement_date', lookup_expr='gte')
    to_achievement_date = DateTimeFilter(
        name='distance_achievement_date', lookup_expr='lte')
    min_distance_in_feet = NumberFilter(
        name='distance_in_feet', lookup_expr='gte')
    max_distance_in_feet = NumberFilter(
        name='distance_in_feet', lookup_expr='lte')
    drone_name = AllValuesFilter(
        name='drone__name')
    pilot_name = AllValuesFilter(
        name='pilot__name')

    class Meta:
        model = Competition
        fields = (
            'distance_in_feet',
            'from_achievement_date',
            'to_achievement_date',
            'min_distance_in_feet',
            'max_distance_in_feet',
            # drone__name will be accessed as drone_name
            'drone_name',
            # pilot__name will be accessed as pilot_name
            'pilot_name',
```



The `CompetitionFilter` class declares the following class attributes:

- `from_achievement_date`: This attribute is a `django_filters.DateTimeFilter` instance that allows the request to filter the competitions whose `achievement_date` `DateTime` value is greater than or equal to the specified `DateTime` value. The value specified in the `name` argument indicates the field to which the `DateTime` filter is applied, `'distance_achievement_date'`, and the value for the `lookup_expr` argument indicates the lookup expression, `'gte'`, which means greater than or equal to.
- `to_achievement_date`: This attribute is a `django_filters.DateTimeFilter` instance that allows the request to filter the competitions whose `achievement_date` `DateTime` value is less than or equal to the specified `DateTime` value. The value specified in the `name` argument indicates the field to which the `DateTime` filter is applied, `'distance_achievement_date'`, and the value for the `lookup_expr` argument indicates the lookup expression, `'lte'`, which means less than or equal to.
- `min_distance_in_feet`: This attribute is a `django_filters.NumberFilter` instance that allows the request to filter the competitions whose `distance_in_feet` numeric value is greater than or equal to the specified number. The value for the `name` argument indicates the field to which the numeric filter is applied, `'distance_in_feet'`, and the value for the `lookup_expr` argument indicates the lookup expression, `'gte'`, which means greater than or equal to.
- `max_distance_in_feet`: This attribute is a `django_filters.NumberFilter`

instance that allows the request to filter the competitions whose `distance_in_feet` numeric value is less than or equal to the specified number. The value for the `name` argument indicates the field to which the numeric filter is applied, '`distance_in_feet`', and the value for the `lookup_expr` argument indicates the lookup expression, '`lte`', which means less than or equal to.

- `drone_name`: This attribute is a `djongo_filters.AllValuesFilter` instance that allows the request to filter the competitions whose drones' names match the specified string value. The value for the `name` argument indicates the field to which the filter is applied, '`drone__name`'. Notice that there is a double underscore (`_`) between `drone` and `name`, and you can read it as the `name` field for the `drone` model or simply replace the double underscore with a dot and read `drone.name`. The name uses Django's double underscore syntax. However, we don't want the request to use `drone__name` to specify the filter for the drone's name. Hence, the instance is stored in the class attribute named `drone_name`, with just a single underscore between `player` and `name`, to make it more user-friendly. We will make configurations to make the browsable API display a drop-down with all the possible values for the drone's name to use as a filter. The drop-down will only include the drones' names that have registered competitions.
- `pilot_name`: This attribute is a `djongo_filters.AllValuesFilter` instance that allows the request to filter the competitions whose pilots' names match the specified string value. The value for the `name` argument indicates the field to which the filter is applied, '`pilot__name`'. The name uses Django's double underscore syntax. As happened with `drone_name`, we don't want the request to use `pilot__name` to specify the filter for the pilot's name, and therefore,

we stored the instance in the class attribute named `pilot_name`, with just a single underscore between `pilot` and `name`. The browsable API will display a drop-down with all the possible values for the pilot's name to use as a filter. The drop-down will only include the pilots' names that have registered competitions because we used the `AllValuesFilter` class.

The `CompetitionFilter` class defines a `Meta` inner class that declares the following two attributes:

- `model`: This attribute specifies the model related to the filter set, that is, the `Competition` class.
- `fields`: This attribute specifies a tuple of strings whose values indicate the field names and filter names that we want to include in the filters for the related model. We included '`distance_in_feet`' and the names for all the previously explained filters. The string '`distance_in_feet`' refers to the field with this name. We want to apply the default numeric filter that will be built under the hood to allow the request to filter by an exact match on the `distance_in_feet` field. This way, the request will have plenty of options to filter competitions.

Now, add the following highlighted lines to the `CompetitionList` class declared in the `views.py` file. The next lines show the new code that defines the class. The code file for the sample is included in the `hillar_django_restful_07_03` folder in the `restful01/drones/views.py` file:

```
class CompetitionList(generics.ListCreateAPIView):  
    queryset = Competition.objects.all()  
    serializer_class = PilotCompetitionSerializer  
    name = 'competition-list'
```

```
filter_class = CompetitionFilter
ordering_fields = (
    'distance_in_feet',
    'distance_achievement_date',
)
```

The `filter_class` attribute specifies `CompetitionFilter` as its value, that is, the `FilterSet` subclass that declares the customized filters that we want to use for this class-based view. In this case, the code didn't specify a tuple of strings for the `filter_class` attribute because we have defined our own `FilterSet` subclass.

The `ordering_fields` tuple of strings specifies the two field names that the request will be able to use for ordering the competitions.

Making requests that filter results

Now we can launch Django's development server to compose and send HTTP requests to understand how to use the previously coded filters. Execute any of the following two commands, based on your needs, to access the API in other devices or computers connected to your LAN. Remember that we analyzed the difference between them in [Chapter 3, Creating API Views](#), in the *Launching Django's development server* section:

```
python manage.py runserver  
python manage.py runserver 0.0.0.0:8000
```

After we run any of the previous commands, the development server will start listening at port 8000.

Now, we will compose and send an HTTP request to retrieve all the drone categories whose `name` is equal to `Quadcopter`, as shown below:

```
http ":8000/drone-categories/?name=Quadcopter"
```

The following is the equivalent `curl` command:

```
curl -iX GET "localhost:8000/drone-categories/?name=Quadcopter"
```

The following lines show a sample response with the single drone category whose `name` matches the specified `name` string in the filter and the list of hyperlinks for the drones that belong to the category. The following

lines show the JSON response body without the headers. Notice that the results are paginated:

```
{  
    "count": 1,  
    "next": null,  
    "previous": null,  
    "results": [  
        {  
            "drones": [  
                "http://localhost:8000/drones/2",  
                "http://localhost:8000/drones/9",  
                "http://localhost:8000/drones/5",  
                "http://localhost:8000/drones/7",  
                "http://localhost:8000/drones/3",  
                "http://localhost:8000/drones/11",  
                "http://localhost:8000/drones/1"  
            ],  
            "name": "Quadcopter",  
            "pk": 1,  
            "url": "http://localhost:8000/drone-categories/1"  
        }  
    ]  
}
```

Composing requests that filter and order results

We will compose and send an HTTP request to retrieve all the drones whose related drone category ID is equal to `1` and whose value for the `has_it_competed` field is equal to `False`. The results must be sorted by `name` in descending order, and therefore, we specify `-name` as the value for the `ordering` query parameter.

The hyphen (-) before the field name indicates that the ordering feature must use descending order instead of the default ascending order.

Make sure you replace `1` with the `pk` value of the previously retrieved drone category named `Quadcopter`. The `has_it_competed` field is a `bool` field, and therefore, we have to use Python valid bool values (`True` and `False`) when specifying the desired values for the `bool` field in the filter:

```
http ":8000/drones/?  
drone_category=1&has_it_competed=False&ordering=-name"
```

The following is the equivalent `curl` command:

```
curl -iX GET "localhost:8000/drones/?  
drone_category=1&has_it_competed=False&ordering=-name"
```

The following lines show a sample response with the first four out of seven drones that match the specified criteria in the filter, sorted by name in descending order. Notice that the filters and the ordering have been combined with the previously configured pagination. The following lines show only the JSON response body, without the headers:

```
{  
    "count": 7,  
    "next": "http://localhost:8000/drones/?  
drone_category=1&has_it_competed=False&limit=4&offset=4&ordering=-  
name",  
    "previous": null,  
    "results": [  
        {  
            "drone_category": "Quadcopter",  
            "has_it_competed": false,  
            "inserted_timestamp": "2017-11-03T01:58:49.135737Z",  
            "manufacturing_date": "2017-07-20T02:02:00.716312Z",  
            "name": "WonderDrone",  
            "url": "http://localhost:8000/drones/1"  
        },  
        {  
            "drone_category": "Quadcopter",  
            "has_it_competed": false,  
            "inserted_timestamp": "2017-11-06T20:25:31.511881Z",  
            "manufacturing_date": "2017-09-20T02:02:00.716312Z",  
            "name": "Skipper",  
            "url": "http://localhost:8000/drones/11"  
        },  
        {  
            "drone_category": "Quadcopter",  
            "has_it_competed": false,  
            "inserted_timestamp": "2017-11-06T20:25:29.636153Z",  
            "manufacturing_date": "2017-01-20T02:02:00.716312Z",  
            "name": "Need for Speed",  
            "url": "http://localhost:8000/drones/3"  
        },  
        {  
            "drone_category": "Quadcopter",  
            "has_it_competed": false,  
            "inserted_timestamp": "2017-11-06T20:25:30.584031Z",  
            "manufacturing_date": "2017-05-20T02:02:00.716312Z",  
            "name": "Gulfstream I",  
            "url": "http://localhost:8000/drones/7"  
        }  
    ]  
}
```

*Notice that the response provides the value for the next key,
http://localhost:8000/drones/?drone_category=1&has_it_competed=False&limit=4&offset=4&ordering=-name. This URL includes the combination of pagination,*

filtering, and ordering query parameters.

In the `DroneList` class, we included '`drone_category`' as one of the strings in the `filter_fields` tuple of strings. Hence, we had to use the drone category ID in the filter.

Now, we will use a filter on the drone's name related to a competition. As previously explained, our `CompetitionFilter` class provides us a filter to the name of the related drone in the `drone_name` query parameter.

We will combine the filter with another filter on the pilot's name related to a competition. Remember that the class also provides us a filter to the name of the related pilot in the `pilot_name` query parameter. We will specify two conditions in the criteria, and the filters are combined with the `AND` operator. Hence, both conditions must be met. The pilot's name must be equal to '`Penelope Pitstop`' and the drone's name must be equal to '`WonderDrone`'. The following command generates a request with the explained filter:

```
http ":8000/competitions/?  
pilot_name=Penelope+Pitstop&drone_name=WonderDrone"
```

The following is the equivalent `curl` command:

```
curl -iX GET "localhost:8000/competitions/?  
pilot_name=Penelope+Pitstop&drone_name=WonderDrone"
```

The following lines show a sample response with the competition that matches the specified criteria in the filters. The following lines show only the JSON response body, without the headers:

```
{  
  "count": 1,  
  "next": null,  
  "previous": null,
```

```
"results": [
    {
        "distance_achievement_date": "2017-10-21T06:02:23.776594Z",
        "distance_in_feet": 2800,
        "drone": "WonderDrone",
        "pilot": "Penelope Pitstop",
        "pk": 2,
        "url": "http://localhost:8000/competitions/2"
    }
]
```

Now, we will compose and send an HTTP request to retrieve all the competitions that match the following criteria. In addition, we want the results ordered by `distance_achievement_date`, in descending order:

1. The `distance_achievement_date` is between `2017-10-18` and `2017-10-21`
2. The `distance_in_feet` value is between `700` and `900`

The following command will do the job:

```
http " :8000/competitions/? min_distance_in_feet=700&max_distance_in_feet=9000&f
```

The following is the equivalent `curl` command:

```
curl -iX GET "localhost:8000/competitions/?min_distance_in_feet=700&max_distance
```

The previously analyzed `CompetitionFilter` class allowed us to create a request like the previous one, in which we take advantage of the customized filters. The following lines show a sample response with the two competitions that match the specified criteria in the filters. We overrode the default ordering specified in the model with the `ordering` field indicated in the request. The following lines show only the JSON body response, without the headers:

```
{  
    "count": 2,  
    "next": null,  
    "previous": null,  
    "results": [  
        {  
            "distance_achievement_date":  
                "2017-10-20T05:03:20.776594Z",  
            "distance_in_feet": 800,  
            "drone": "Atom",  
            "pilot": "Penelope Pitstop",  
            "pk": 1,  
            "url": "http://localhost:8000/competitions/1"  
        },  
        {  
            "distance_achievement_date":  
                "2017-10-20T05:43:20.776594Z",  
            "distance_in_feet": 790,  
            "drone": "Atom",  
            "pilot": "Peter Perfect",  
            "pk": 3,  
            "url": "http://localhost:8000/competitions/3"  
        }  
    ]  
}
```



Making requests that perform starts with searches

Now, we will take advantage of searches that are configured to check whether a value starts with the specified characters. We will compose and send an HTTP request to retrieve all the pilots whose `name` starts with '`G`'.

The next request uses the search feature that we configured to restrict the search behavior to a starts-with match on the `name` field for the `drone` model:

```
http ":8000/drones/?search=G"
```

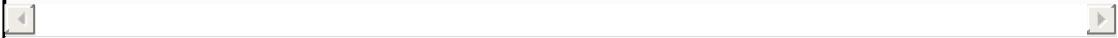
The following is the equivalent `curl` command:

```
curl -iX GET "localhost:8000/drones/?search=G"
```

The following lines show a sample response with the two drones that match the specified search criteria, that is, those drones whose `name` starts with '`G`'. The following lines show only the JSON response body, without the headers:

```
{
  "count": 2,
  "next": null,
  "previous": null,
  "results": [
    {
      "drone_category": "Quadcopter",
      "has_it_competed": false,
      "inserted_timestamp": "2017-11-06T20:25:30.127661Z",
      "manufacturing_date": "2017-03-20T02:02:00.716312Z",
      "name": "Gossamer Albatross",
```

```
        "url": "http://localhost:8000/drones/5"
    },
    {
        "drone_category": "Quadcopter",
        "has_it_competed": false,
        "inserted_timestamp": "2017-11-06T20:25:30.584031Z",
        "manufacturing_date": "2017-05-20T02:02:00.716312Z",
        "name": "Gulfstream I",
        "url": "http://localhost:8000/drones/7"
    }
]
```



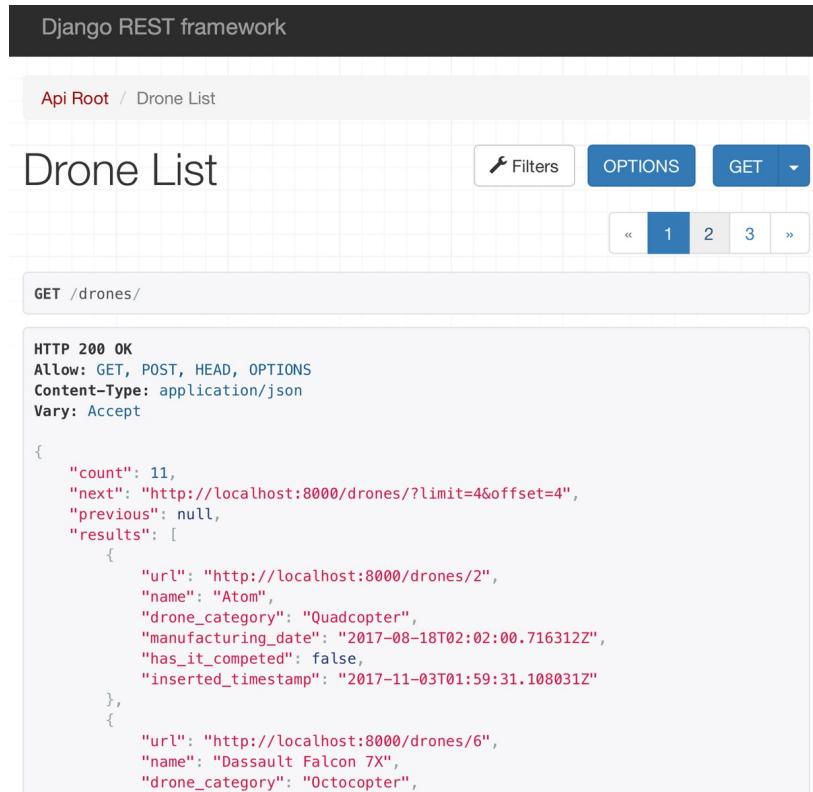
Using the browsable API to test pagination, filtering, searching, and ordering

We enabled pagination and we added filtering, searching, and ordering features to our RESTful Web Service. All of these new features have an impact on how each web page is rendered when working with the browsable API.

We can work with a web browser to easily test pagination, filtering, searching, and ordering with a few clicks or taps.

Open a web browser and go to `http://localhost:8000/drones/`. Replace `localhost` with the IP of the computer that is running Django's development server if you use another computer or device to run the browser. The browsable API will compose and send a `GET` request to `/drones/` and will display the results of its execution, that is, the headers and the JSON drones list.

We have configured pagination, and therefore, the rendered web page will include the default pagination template associated with the base pagination class we are using and will display the available page numbers in the upper-right corner of the web page. The following screenshot shows the rendered web page after entering the URL in a web browser with the resource description, Drone List, and the three pages generated with the limit/offset pagination scheme:



Now, go to `http://localhost:8000/competitions/`. The browsable API will compose and send a `GET` request to `/competitions/` and will display the results of its execution, that is, the headers and the JSON competitions list. The web page will include a Filters button at the right-hand side of the resource description, Competition List, and at the left-hand side of the OPTIONS button.

Click or tap on Filters, and the browsable API will render the Filter model with the appropriate controls for each filter that you can apply below Field Filters. In addition, the model will render the different ordering options below Ordering. The following screenshot shows the Filters model for the competitions:

The screenshot shows a 'Filters' dialog box on a Mac OS X interface. The 'Field filters' section contains several input fields:

- Distance in feet: An input field with a dropdown arrow.
- Distance achievement date is greater than or equal to: An input field with a dropdown arrow.
- Distance achievement date is less than or equal to: An input field with a dropdown arrow.
- Distance in feet is greater than or equal to: A dropdown menu set to 85, with up and down arrows.
- Distance in feet is less than or equal to: An input field with a dropdown arrow.
- Drone name: A dropdown menu set to Atom, with up and down arrows.
- Pilot name: A dropdown menu set to Penelope Pitstop, with up and down arrows.

A 'Submit' button is located below these fields. The 'Ordering' section contains four options, with the last one selected:

- distance_in_feet - ascending
- distance_in_feet - descending
- distance_achievement_date - ascending
- distance_achievement_date - descending (selected, indicated by a blue background and a checked checkbox)

The Drone name and Pilot name drop-downs only provide the related drones' names and pilots' names that have participated in competitions because we used the `AllValuesFilter` class for both filters. We can easily enter all the values for each desired filter that we want to apply and click or tap Submit. Then, click on Filters again, select the ordering option, and click Submit. The browsable API will compose and send the necessary HTTP request to apply the filters and ordering we have specified and it will render a web page with the first page of the results of the execution of the request.

The next screenshot shows the results of executing a request whose filters were composed with the previously explained model:

Competition List

GET /competitions/?distance_in_feet=&from_achievement_date=&to_achievement_date=&min_distance_in_feet=85&max_distance_in_feet=&drone_name=Atom

HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
{  
    "count": 1,  
    "next": null,  
    "previous": null,  
    "results": [  
        {  
            "url": "http://localhost:8000/competitions/1",  
            "pk": 1,  
            "distance_in_feet": 800,  
            "distance_achievement_date": "2017-10-20T05:03:20.776594Z",  
            "pilot": "Penelope Pitstop",  
            "drone": "Atom"  
        }  
    ]  
}
```

The following are the parameters for the HTTP `GET` request. Notice that the browsable API generates the query parameters but doesn't specify values for the filters that were left without values in the previous modal. When the query parameters don't specify values, they are ignored:

```
http://localhost:8000/competitions/?distance_in_feet=&drone_name=Atom&format=json
```

As happens whenever we have to test the different features included in our RESTful Web Service, the browsable API is also extremely helpful whenever we need to check filters and ordering.

Test your knowledge

Let's see whether you can answer the following questions correctly:

1. The `django_filters.rest_framework.DjangoFilterBackend` class provides:
 1. Control on how the results are ordered with a single query parameter
 2. Single query parameter-based searching capabilities, based on the Django admin's search function
 3. Field filtering capabilities

2. The `rest_framework.filters.SearchFilter` class provides:
 1. Control on how the results are ordered with a single query parameter
 2. Single query parameter-based searching capabilities, based on the Django admin's search function
 3. Field filtering capabilities

3. If we want to create a unique constraint, what must be added to a `models.CharField` initializer as one of the named arguments?
 1. `unique=True`
 2. `unique_constraint=True`
 3. `force_unique=True`

4. Which of the following class attributes specifies a tuple of strings whose values indicate the field names that we want to be able to filter against in a class-based view that inherits from

`generics.ListCreateAPIView`:

1. `filters`
2. `filtering_fields`
3. `filter_fields`

5. Which of the following class attributes specifies a tuple of strings whose values indicate the field names that the HTTP request can specify to sort the results in a class-based view that inherits from

`generics.ListCreateAPIView`:

1. `order_by`
2. `ordering_fields`
3. `order_fields`

The rights answers are included in the [Appendix](#), *Solutions*.

Summary

In this chapter, we used the browsable API feature to navigate through the API with resources and relationships. We added unique constraints to improve consistency for the models in our RESTful Web Service.

We understood the importance of paginating results and we configured and tested a global limit/offset pagination scheme with the Django REST framework. Then, we created our own customized pagination class to make sure that requests weren't able to acquire a huge amount of elements in a single page.

We configured filter backend classes and we added code to the models to add filtering, searching, and ordering capabilities to the class-based views. We created a customized filter and we made requests to filter, search, and order results, and we understood how everything worked under the hood. Finally, we used the browsable API to test pagination, filtering, and ordering.

Now that we improved our RESTful Web Service with unique constraints, paginated results, filtering, searching, and ordering features, we will secure the API with authentication and permissions. We will cover these topics in the next chapter.

Securing the API with Authentication and Permissions

In this chapter, we will understand the difference between authentication and permissions in the Django REST framework. We will start securing our RESTful Web Service by adding requirements for authentication schemes and specifying permission policies. We will gain an understanding of:

- Understanding authentication and permissions in Django, the Django REST framework, and RESTful Web Services
- Authentication classes
- Security and permissions-related data to models
- Working with object-level permissions via customized permission classes
- Saving information about users that make requests
- Setting permissions policies
- Creating the superuser for Django
- Creating a user for Django
- Making authenticated requests
- Browsing the secured API with the required authentication

- Working with token-based authentication
- Generating and using tokens

Understanding authentication and permissions in Django, the Django REST framework, and RESTful Web Services

Right now, our sample RESTful Web Service processes all the incoming requests without requiring any kind of authentication, that is, any user can perform requests. The Django REST framework allows us to easily use diverse authentication schemes to identify a user that originated the request or a token that signed the request. Then, we can use these credentials to apply permission and throttling policies that will determine whether the request must be permitted or not.

We already know how configurations work with the Django REST framework. We can apply a global setting and override it if necessary in the appropriate class-based views. Hence, we can set the default authentication schemes in the global settings and override them whenever required for specific scenarios.

The settings allow us to declare a list of classes that specify the authentication schemes to be used for all the incoming HTTP requests. The Django REST framework will use all the specified classes in the list to authenticate a request, before running the appropriate method for the class-based view based on the request.

We can specify just one class. However, it is very important to understand the behavior in case we have to use more than one class. The first class in the list that generates a successful authentication will be responsible for setting the values for the following two attributes for the `request` object:

- `user`: This attribute represents the user model instance. In our examples, we will work with an instance of the Django User class, specifically, the `django.contrib.auth.User` class.
- `auth`: This attribute provides additional authentication data required by the authentication scheme, such as an authentication token.

After a successful authentication, we will be able to use the `request.user` attribute within the different methods in our class-based views that receive the `request` parameter. This way, we will be able to retrieve additional information about the `user` that generated the request.

Learning about the authentication classes

The Django REST framework provides the following three authentication classes in the `rest_framework.authentication` module. All of them are subclasses of the `BaseAuthentication` class:

- `BasicAuthentication`: This class provides an HTTP basic authentication against a username and a password.
- `SessionAuthentication`: This class works with Django's session framework for authentication.
- `TokenAuthentication`: This class provides a simple token-based authentication. The request must include the token generated for a user as the value for the `Authorization` HTTP header key with the '`Token`' string as a prefix for the token.

Of course, in a production environment, we must make sure that the RESTful Web Service is only available over HTTPS, with the usage of the latest TLS versions. We shouldn't use an HTTP basic authentication or a simple token-based authentication over plain HTTP in a production environment.

The previous classes are included in the Django REST framework out of the box. There are many additional authentication classes provided by many third-party libraries. We will work with some of these libraries later in this chapter.

Make sure you quit Django's development server. Remember that you just need to press `Ctrl + C` in the terminal or go to the Command Prompt

window in which it is running. We have to edit the models and then execute migrations before starting Django's development server again.

We will make the necessary changes to combine HTTP basic authentication against a username and a password with Django's session framework for authentication. Hence, we will add the `BasicAuthentication` and `SessionAuthentication` classes in the global authentication classes list.

Open the `restful01/restful01/settings.py` file that declares the module-level variables that define the configuration of Django for the `restful01` project. We will make some changes to this Django settings file. Add the highlighted lines to the `REST_FRAMEWORK` dictionary. The following lines show the new declaration of the `REST_FRAMEWORK` dictionary. The code file for the sample is included in the `hillar_django_restful_08_01` folder in the `restful01/restful01/settings.py` file:

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
        'drones.custompagination.LimitOffsetPaginationWithUpperBound',
    'PAGE_SIZE': 4,
    'DEFAULT_FILTER_BACKENDS': (
        'django_filters.rest_framework.DjangoFilterBackend',
        'rest_framework.filters.OrderingFilter',
        'rest_framework.filters.SearchFilter',
    ),
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.BasicAuthentication',
        'rest_framework.authentication.SessionAuthentication',
    )
}
```

We added the `DEFAULT_AUTHENTICATION_CLASSES` settings key to the `REST_FRAMEWORK` dictionary. This new key specifies a global setting with a tuple of string whose values indicate the classes that we want to use for authentication: `BasicAuthentication` and `SessionAuthentication`.

Including security and permissions-related data to models

We want each drone to have an owner. Only an authenticated user will be able to create a drone and it will automatically become the owner of this new drone. We want only the owner of a drone to be able to update or delete the drone. Hence, an authenticated user that is also the owner of the drone will be able to execute `PATCH`, `PUT`, and `DELETE` methods on the drone resource that he owns.

Any authenticated user that isn't the owner of a specific drone resource will have read-only access to this drone. In addition, unauthenticated requests will also have read-only access to drones.

We will combine authentication with specific permissions. Permissions use the authentication information included in the `request.user` and `request.auth` attributes to determine whether the request should be granted or denied access. Permissions allow us to control which types of users will be granted or denied access to the different features, methods, resources, or resource collections of our RESTful Web Service.

We will use the permissions features in the Django REST framework to allow only authenticated users to create new drones and automatically become their owners. We will make the necessary changes in the models to make a drone have a user as its owner. We will take advantage of the out-of-the-box permission classes included in the framework combined with a customized permission class, to define the previously explained permission policies for the drones and their related HTTP verbs supported in our web service.

In this case, we will stay focused on security and permissions and we will leave throttling rules for the next chapters. Bear in mind that throttling rules also determine whether a specific request must be authorized or not. However, we will work on throttling rules later and we will combine them with authentication and permissions.

Open the `restful01/drones/models.py` file and replace the code that declares the `Drone` class with the following code. The new lines are highlighted in the code listing. The code file for the sample is included in the `hillar_django_restful_08_01` folder, in the `restful01/drones/models.py` file:

```
class Drone(models.Model):
    name = models.CharField(max_length=250, unique=True)
    drone_category = models.ForeignKey(
        DroneCategory,
        related_name='drones',
        on_delete=models.CASCADE)
    manufacturing_date = models.DateTimeField()
    has_it_competed = models.BooleanField(default=False)
    inserted_timestamp = models.DateTimeField(auto_now_add=True)
    owner = models.ForeignKey(
        'auth.User',
        related_name='drones',
        on_delete=models.CASCADE)

    class Meta:
        ordering = ('name',)

    def __str__(self):
        return self.name
```

The highlighted lines declare a new `owner` field for the `Drone` model. The new field uses the `django.db.models.ForeignKey` class to provide a many-to-one relationship to the `django.contrib.auth.user` model.

This `user` model persists the users for the Django authentication system. Now, we are using this authentication system for our RESTful Web Service. The '`drones`' value specified for the `related_name` argument creates a backward relation from the `user` to the `Drone` model. Remember that this value indicates the name to use for the relation from the related `user` object

back to a `Drone` object. This way, we will be able to access all the drones owned by a specific user.

Whenever we delete a `User`, we want all drones owned by this user to be deleted too, and therefore, we specified the `models.CASCADE` value for the `on_delete` argument.

Open the `restful01/drones/serializers.py` file and add the following code after the last line that declares the imports, before the declaration of the `DroneCategorySerializer` class. The code file for the sample is included in the `hillar_django_restful_08_01` folder, in the `restful01/drones/serializers.py` file:

```
from django.contrib.auth.models import User

class UserDroneSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Drone
        fields = (
            'url',
            'name')

class UserSerializer(serializers.HyperlinkedModelSerializer):
    drones = UserDroneSerializer(
        many=True,
        read_only=True)

    class Meta:
        model = User
        fields = (
            'url',
            'pk',
            'username',
            'drone')
```

We don't want to use the `DroneSerializer` serializer class for the drones related to a user because we want to serialize fewer fields, and therefore, we created the `UserDroneSerializer` class. This class is a subclass of the `HyperlinkedModelSerializer` class. This new serializer allows us to serialize the drones related to a `User`. The `UserDroneSerializer` class defines a `Meta` inner

class that declares the following two attributes:

- `model`: This attribute specifies the model related to the serializer, that is, the `Drone` class.
- `fields`: This attribute specifies a tuple of string whose values indicate the field names that we want to include in the serialization from the related model. We just want to include the URL and the drone's name, and therefore, the code includes '`url`' and '`name`' as members of the tuple.

The `UserSerializer` is a subclass of the `HyperlinkedModelSerializer` class. This new serializer class declares a `drones` attribute as an instance of the previously explained `UserDroneSerializer` class, with the `many` and `read_only` arguments equal to `True` because it is a one-to-many relationship and it is read-only. The code specifies the `drones` name that we specified as the string value for the `related_name` argument when we added the `owner` field as a `models.ForeignKey` instance in the `Drone` model. This way, the `drones` field will provide us with an array of URLs and names for each drone that belongs to the user.

Now, we will add an `owner` field to the existing `DroneSerializer` class. Open the `restful01/drones/serializers.py` file and replace the code that declares the `DroneSerializer` class with the following code. The new lines are highlighted in the code listing. The code file for the sample is included in the `hillar_django_restful_08_01` folder, in the `restful01/drones/serializers.py` file.

```
class DroneSerializer(serializers.HyperlinkedModelSerializer):  
    # Display the category name  
    drone_category = serializers.SlugRelatedField(queryset=DroneCategory.objects  
    # Display the owner's username (read-only)  
    owner = serializers.ReadOnlyField(source='owner.username')  
  
    class Meta:  
        model = Drone  
        fields = (  
            'url',
```

```
'name',
'drone_category',
'owner',
'manufacturing_date',
'has_it_competed',
'inserted_timestamp',)
```

The new version of the `DroneSerializer` class declares an `owner` attribute as an instance of `serializers.ReadOnlyField` with the `source` argument equal to `'owner.username'`. This way, the serializer will serialize the value for the `username` field of the related `django.contrib.auth.User` instance stored in the `owner` field.

The code uses the `ReadOnlyField` class because the owner is automatically populated when an authenticated user creates a new drone. It will be impossible to change the owner after a drone has been created with an HTTP `POST` method call. This way, the `owner` field will render the username that created the related drone. In addition, we added `'owner'` to the `fields` string tuple within the `Meta` inner class.

We made the necessary changes to the `Drone` model and its serializer (the `DroneSerializer` class) to make drones have owners.

Working with object-level permissions via customized permission classes

The `rest_framework.permissions.BasePermission` class is the base class from which all customized permission classes should inherit to work with the Django REST framework. We want to make sure that only a drone owner can update or delete an existing drone.

Go to the `restful01/drones` folder and create a new file named `custompermission.py`. Write the following code in this new file. The following lines show the code for this file that declares the new `IsCurrentUserOwnerOrReadOnly` class declared as a subclass of the `BasePermission` class. The code file for the sample is included in the `hillar_django_restful_08_01` folder in the `restful01/drones/custompermission.py` file:

```
from rest_framework import permissions

class IsCurrentUserOwnerOrReadOnly(permissions.BasePermission):
    def has_object_permission(self, request, view, obj):
        if request.method in permissions.SAFE_METHODS:
            # The method is a safe method
            return True
        else:
            # The method isn't a safe method
            # Only owners are granted permissions for unsafe methods
            return obj.owner == request.user
```

The previous lines declare the `IsCurrentUserOwnerOrReadOnly` class and override the `has_object_permission` method defined in the `BasePermission` superclass that returns a `bool` value indicating whether the permission should be granted or not.

The `permissions.SAFE_METHODS` tuple of string includes the three HTTP methods or verbs that are considered safe because they are read-only and they don't produce changes to the related resource or resource collection: '`GET`', '`HEAD`', and '`OPTIONS`'. The code in the `has_object_permission` method checks whether the HTTP verb specified in the `request.method` attribute is any of the three safe methods specified in `permission.SAFE_METHODS`. If this expression evaluates to `True`, the `has_object_permission` method returns `True` and grants permission to the request.

If the HTTP verb specified in the `request.method` attribute is not any of the three safe methods, the code returns `True` and grants permission only when the `owner` attribute of the received `obj` object (`obj.owner`) matches the user that originated the request (`request.user`). The user that originated the request will always be the authenticated user. This way, only the owner of the related resource will be granted permission for those requests that include HTTP verbs that aren't safe.

We will use the new `IsCurrentUserOwnerOrReadOnly` customized permission class to make sure that only the drone owners can make changes to an existing drone. We will combine this permission class with the `rest_framework.permissions.IsAuthenticatedOrReadOnly` one that only allows read-only access to resources when the request doesn't belong to an authenticated user. This way, whenever an anonymous user performs a request, he will only have read-only access to the resources.

Saving information about users that make requests

Whenever a user performs an HTTP `POST` request to the drone resource collection to create a new drone resource, we want to make the authenticated user that makes the request the owner of the new drone. In order to make this happen, we will override the `perform_create` method in the `DroneList` class declared in the `views.py` file.

Open the `restful01/drones/views.py` file and replace the code that declares the `DroneList` class with the following code. The new lines are highlighted in the code listing. The code file for the sample is included in the `hillar_django_restful_08_01` folder, in the `restful01/drones/views.py` file:

```
class DroneList(generics.ListCreateAPIView):
    queryset = Drone.objects.all()
    serializer_class = DroneSerializer
    name = 'drone-list'
    filter_fields = (
        'name',
        'drone_category',
        'manufacturing_date',
        'has_it_competed',
    )
    search_fields = (
        '^name',
    )
    ordering_fields = (
        'name',
        'manufacturing_date',
    )

    def perform_create(self, serializer):
        serializer.save(owner=self.request.user)
```

The `generics.ListCreateAPIView` class inherits from the `CreateModelMixin` class

and other classes. The `DroneList` class inherits the `perform_create` method from the `rest_framework.mixins.CreateModelMixin` class.

The code that overrides the `perform_create` method provides an additional `owner` field to the `create` method by setting a value for the `owner` argument in the call to the `serializer.save` method. The code sets the `owner` argument to the value of `self.request.user`, that is, to the authenticated user that is making the request. This way, whenever a new `Drone` is created and persisted, it will save the `user` associated to the request as its owner.

Setting permission policies

We will configure permission policies for the class-based views that work with the `Drone` model. We will override the value for the `permission_classes` class attribute for the `DroneDetail` and `DroneList` classes.

We will add the same lines of code in the two classes. We will include the `IsAuthenticatedOrReadOnly` class and our recently declared `IsCurrentUserOwnerOrReadOnly` permission class in the `permission_classes` tuple.

Open the `restful01/drones/views.py` file and add the following lines after the last line that declares the imports, before the declaration of the `DroneCategorySerializer` class:

```
from rest_framework import permissions
from drones import custompermission
```

Replace the code that declares the `DroneDetail` class with the following code in the same `views.py` file. The new lines are highlighted in the code listing. The code file for the sample is included in the `hillar_django_restful_08_01` folder, in the `restful01/drones/views.py` file:

```
class DroneDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Drone.objects.all()
    serializer_class = DroneSerializer
    name = 'drone-detail'
    permission_classes = (
        permissions.IsAuthenticatedOrReadOnly,
        custompermission.IsCurrentUserOwnerOrReadOnly,
    )
```

Replace the code that declares the `DroneList` class with the following code in the same `views.py` file. The new lines are highlighted in the code listing.

The code file for the sample is included in the `hillar_django_restful_08_01` folder, in the `restful01/drones/views.py` file:

```
class DroneList(generics.ListCreateAPIView):
    queryset = Drone.objects.all()
    serializer_class = DroneSerializer
    name = 'drone-list'
    filter_fields = (
        'name',
        'drone_category',
        'manufacturing_date',
        'has_it_competed',
    )
    search_fields = (
        '^name',
    )
    ordering_fields = (
        'name',
        'manufacturing_date',
    )
    permission_classes = (
        permissions.isAuthenticatedOrReadOnly,
        custompermission.IsCurrentUserOwnerOrReadOnly,
    )

    def perform_create(self, serializer):
        serializer.save(owner=self.request.user)
```

Creating the superuser for Django

Now, we will run the necessary command to create the `superuser` for Django that will allow us to authenticate our requests. We will create other users later.

Make sure you are in the `restful01` folder that includes the `manage.py` file in the activated virtual environment. Execute the following command that executes the `createsuperuser` subcommand for the `manage.py` script to allow us to create the `superuser`:

```
python manage.py createsuperuser
```

The command will ask you for the username you want to use for the `superuser`. Enter the desired username and press *Enter*. We will use `djangosuper` as the username for this example. You will see a line similar to the following one:

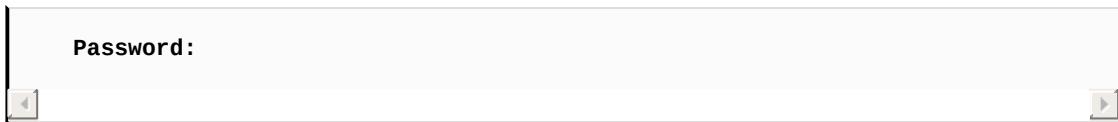
```
Username (leave blank to use 'gaston'):
```

Then, the command will ask you for the email address. Enter an email address and press *Enter*. You can enter `djangosuper@example.com`:

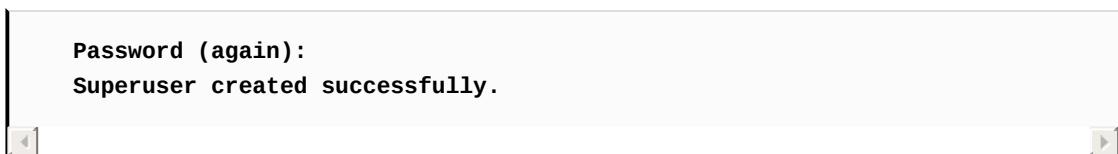
```
Email address:
```

Finally, the command will ask you for the password for the new `superuser`. Enter your desired password and press *Enter*. We will use `passwordforsuper` as an example in our tests. Of course, this password is not the best

example of a strong password. However, the password is easy to type and read in our tests:



The command will ask you to enter the password again. Enter it and press *Enter*. If both entered passwords match, the superuser will be created:



Our database has many rows in the `drones_drone` table. We added a new `owner` field for the `Drone` model and this required field will be added to the `drones_drone` table after we execute migrations. We have to assign a default owner for all the existing drones to make it possible to add this new required field without having to delete all these drones. We will use one of the features included in Django to solve the issue.

First, we have to know the `id` value for the superuser we have created to use it as the default owner for the existing drones. Then, we will use this value to let Django know which is the default owner for the existing drones.

We created the first user, and therefore, the `id` will be equal to `1`. However, we will check the procedure to determine the `id` value in case you create other users and you want to assign any other user as the default owner.

You can check the row in the `auth_user` table whose `username` field matches '`djangosuper`' in any tool that works with PostgreSQL. Another option is to run the following commands to retrieve the ID from the `auth_user` table for the row whose `username` is equal to '`djangosuper`'. In case you specified a different name, make sure you use the appropriate one. In addition, replace the `username` in the command with the `username` you used to create the PostgreSQL database and password with your chosen password for this database user. You specified this information when you executed

the steps explained in [chapter 6](#), *Working with Advanced Relationships and Serialization*, in the *Running migrations that generate relationships* section.

The command assumes that you are running PostgreSQL on the same computer in which you are executing the command:

```
psql --username=username --dbname=drones --command="SELECT id FROM auth_user WHERE username = 'djangosuper';"
```

The following lines show the output with the value for the `id` field: 1:

```
id  
----  
1  
(1 row)
```

Now, run the following Python script to generate the migrations that will allow us to synchronize the database with the new field we added to the `Drone` model:

```
python manage.py makemigrations drones
```

Django will explain to us that we cannot add a non-nullable field without a default and will ask us to select an option with the following message:

```
You are trying to add a non-nullable field 'owner' to drone without a default; we can't do that (the database needs something to populate existing rows).  
Please select a fix:  
1) Provide a one-off default now (will be set on all existing rows with a null value for this column)  
2) Quit, and let me add a default in models.py  
Select an option:
```

Enter `1` and press *Enter*. This way, we will select the first option to provide the one-off default that will be set on all the existing `drones_drone` rows.

Django will ask us to provide the default value we want to set for the `owner` field of the `drones_drone` table:

```
Please enter the default value now, as valid Python
The datetime and django.utils.timezone modules are available, so
you can do e.g. timezone.now
Type 'exit' to exit this prompt
>>>
```

Enter the value for the previously retrieved `id: 1`. Then, press *Enter*. The following lines show the output generated after running the previous command:

```
Migrations for 'drones':
  drones/migrations/0003_drone_owner.py
    - Add field owner to drone
```

The output indicates that the `restful01/drones/migrations/0003_drone_owner.py` file includes the code to add the field named `owner` to the `drone` table. The following lines show the code for this file that was automatically generated by Django. The code file for the sample is included in the `hillar_django_restful_08_01` folder, in the `restful01/drones/migrations/0003_drone_owner.py` file:

```
# -*- coding: utf-8 -*-
# Generated by Django 1.11.5 on 2017-11-09 22:04
from __future__ import unicode_literals
from django.conf import settings
from django.db import migrations, models
import django.db.models.deletion


class Migration(migrations.Migration):


    dependencies = [
        migrations.swappable_dependency(settings.AUTH_USER_MODEL),
```

```
        ('drones', '0002_auto_20171104_0246'),
    ]

operations = [
    migrations.AddField(
        model_name='drone',
        name='owner',
        field=models.ForeignKey(default=1, on_delete django.db.models.deleti
        preserve_default=False,
    ),
]
```

The code declares the `Migration` class as a subclass of the `django.db.migrations.Migration` class. The `Migration` class defines an `operations` list with a `migrations.AddField` instance that will add the `owner` field to the table related to the `drone` model.

Now, run the following Python script to apply all the generated migrations and execute the changes in the database tables:

```
python manage.py migrate
```

The following lines show the output generated after running the previous command:

```
Operations to perform:
Apply all migrations: admin, auth, contenttypes, drones, sessions
Running migrations:
  Applying drones.0003_drone_owner... OK
```

After we run the previous command, we will have a new `owner_id` field in the `drones_drone` table in the PostgreSQL database. The existing rows in the `drones_drone` table will use the default value we instructed Django to use for the new `owner_id` field: 1. This way, the superuser named '`djangosuper`' will be the owner for all the existing drones.

We can use the PostgreSQL command line or any other application that

allows us to easily check the contents of the PostgreSQL database to browse the `drones_drone` table that Django updated.

The following screenshot shows the new structure for the `drones_drone` table at the left-hand side and all its rows at the right-hand side:

The screenshot shows the pgAdmin interface with the following details:

- Left Panel (Schema Browser):** Shows the `drones` schema with its tables: `auth_group`, `auth_group_permissions`, `auth_permission`, `auth_user`, `auth_user_groups`, `auth_user_user_permissions`, `django_admin_log`, `django_content_type`, `django_migrations`, `django_session`, `drones_competition`, and `drones_drone`. The `drones_drone` table is expanded, showing columns: `id`, `name`, `manufacturing_date`, `has_it_competed`, `inserted_timestamp`, `drone_category_id`, and `owner_id`.
- Top Right Panel (Query Editor):** Untitled 1.sql tab with the query: `SELECT * FROM drones_drone LIMIT 100;`
- Bottom Right Panel (Results):** A table displaying 11 rows of data from the `drones_drone` table.

	<code>id</code>	<code>name</code>	<code>manufacturing_date</code>	<code>has_it_competed</code>	<code>inserted_timestamp</code>	<code>drone_category_id</code>	<code>owner_id</code>
1	1	WonderDrone	2017-07-19 23:02:00.716312-03	False	2017-11-02 22:58:49.135737-03	1	1
2	2	Atom	2017-08-17 23:02:00.716312-03	False	2017-11-02 22:59:31.108031-03	1	1
3	3	Need for Speed	2017-01-19 23:02:00.716312-03	False	2017-11-06 17:25:29.936153-03	1	1
4	4	Eclipse	2017-02-17 23:02:00.716312-03	False	2017-11-06 17:25:29.909965-03	2	1
5	5	Gossamer Albatross	2017-03-19 23:02:00.716312-03	False	2017-11-06 17:25:30.127661-03	1	1
6	6	Dassault Falcon 7X	2017-04-17 23:02:00.716312-03	False	2017-11-06 17:25:30.357127-03	2	1
7	7	Gulfstream I	2017-05-19 23:02:00.716312-03	False	2017-11-06 17:25:30.584031-03	1	1
8	8	RV-3	2017-06-17 23:02:00.716312-03	False	2017-11-06 17:25:30.819695-03	2	1
9	9	Dusty	2017-07-19 23:02:00.716312-03	False	2017-11-06 17:25:31.049833-03	1	1
10	10	Ripslinger	2017-08-17 23:02:00.716312-03	False	2017-11-06 17:25:31.279172-03	2	1
11	11	Skipper	2017-09-19 23:02:00.716312-03	False	2017-11-06 17:25:31.511881-03	1	1

Creating a user for Django

Now, we will use Django's interactive shell to create a new user for Django. Run the following command to launch Django's interactive shell. Make sure you are within the `restful01` folder in the terminal, Command Prompt, or Windows Powershell window in which you have the virtual environment activated:

```
python manage.py shell
```

You will notice that a line that says (InteractiveConsole) is displayed after the usual lines that introduce your default Python interactive shell. Enter the following code in the shell to create another user that is not a superuser. We will use this user and the superuser to test our changes in the permissions policies. The code file for the sample is included in the `hillar_django_restful_08_01` folder, in the `scripts/create_user.py` file. You can replace `user01` with your desired username, `user01@example.com` with the email and `user01password` with the password you want to use for this user. Notice that we will be using these credentials in the following sections. Make sure you always replace the credentials with your own credentials:

```
from django.contrib.auth.models import User  
  
user = User.objects.create_user('user01', 'user01@example.com', 'user01password')  
user.save()
```

Finally, enter the following command to quit the interactive console:

```
quit()
```

You can achieve the same goal by pressing *Ctrl + D*. Now, we have a new user for Django named `user01`.

Making authenticated requests

Now, we can launch Django's development server to compose and send authenticated HTTP requests to understand how the configured authentication classes, combined with the permission policies, work. Execute any of the following two commands based on your needs to access the API in other devices or computers connected to your LAN. Remember that we analyzed the difference between them in [Chapter 3, *Creating API Views*](#), in the *Launching Django's development server* section:

```
python manage.py runserver  
python manage.py runserver 0.0.0.0:8000
```

After we run any of the previous commands, the development server will start listening at port 8000.

We will compose and send an HTTP `POST` request without authentication credentials to try to create a new drone:

```
http POST :8000/drones/ name="Python Drone" drone_category="Quadcopter" manufact
```

The following is the equivalent `curl` command:

```
curl -iX POST -H "Content-Type: application/json" -d  
'{"name": "Python Drone", "drone_category": "Quadcopter",  
"manufacturing_date": "2017-07-16T02:03:00.716312Z",  
"has_it_competed": "false"}' localhost:8000/drones/
```

We will receive an `HTTP 401 Unauthorized` status code in the response header

and a detail message indicating that we didn't provide authentication credentials in the JSON body. The following lines show a sample response:

```
HTTP/1.0 401 Unauthorized
Allow: GET, POST, HEAD, OPTIONS
Content-Length: 58
Content-Type: application/json
Date: Tue, 19 Dec 2017 19:52:44 GMT
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Accept, Cookie
WWW-Authenticate: Basic realm="api"
X-Frame-Options: SAMEORIGIN

{
    "detail": "Authentication credentials were not provided."
}
```

After the changes we made, if we want to create a new drone, that is, to make an HTTP POST request to /drones/, we need to provide authentication credentials by using HTTP authentication. Now, we will compose and send an HTTP request to create a new drone with authentication credentials, that is, with the superuser name and his password. Remember to replace `djangosuper` with the name you used for the superuser and `passwordforsuper` with the password you configured for this user:

```
http -a "djangosuper":"passwordforsuper" POST :8000/drones/ name="Python Drone"
```

The following is the equivalent `curl` command:

```
curl --user "djangosuper":"passwordforsuper" -iX POST -H "Content-Type: application/json" -d '{"name": "Python Drone", "drone_category": "Quadcopter", "manufacturing_date": "2017-07-16T02:03:00.716312Z", "has_it_competed": "false"}' localhost:8000/drones/
```

The new `Drone` with the superuser named `djangosuper` as its owner has been successfully created and persisted in the database because the request was

authenticated. As a result of the request, we will receive an `HTTP 201 Created` status code in the response header and the recently persisted `drone` serialized to JSON in the response body. The following lines show an example response for the HTTP request, with the new `drone` object in the JSON response body. Notice that the JSON response body includes the `owner` key and the username that created the drone as its value: `djangosuper`:

```
HTTP/1.0 201 Created
Allow: GET, POST, HEAD, OPTIONS
Content-Length: 219
Content-Type: application/json
Date: Fri, 10 Nov 2017 02:55:07 GMT
Location: http://localhost:8000/drones/12
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
    "drone_category": "Quadcopter",
    "has_it_competed": false,
    "inserted_timestamp": "2017-11-10T02:55:07.361574Z",
    "manufacturing_date": "2017-07-16T02:03:00.716312Z",
    "name": "Python Drone",
    "owner": "djangosuper",
    "url": "http://localhost:8000/drones/12"
}
```

Now, we will try to update the `has_it_competed` field value for the previously created drone with an HTTP `PATCH` request. However, we will use the other user we created in Django to authenticate this HTTP `PATCH` request. This user isn't the owner of the drone, and therefore, the request shouldn't succeed.

Replace `user01` and `user01password` in the next command with the name and password you configured for this user. In addition, replace `12` with the ID generated for the previously created drone in your configuration:

```
http -a "user01":"user01password" PATCH :8000/drones/12 has_it_competed=true
```

The following is the equivalent `curl` command:

```
curl --user "user01":"user01password" -iX PATCH -H "Content-Type: application/json" -d '{ "name": "drone12", "owner": "djangosuper", "lat": 37.7749, "lon": -122.4194 }' http://127.0.0.1:8000/drones/12
```

We will receive an `HTTP 403 Forbidden` status code in the response header and a detail message indicating that we do not have permission to perform the action in the JSON body. The owner for the drone we want to update is `djangosuper` and the authentication credentials for this request use a different user: `user01`. Hence, the operation is rejected by the `has_object_permission` method in the `isCurrentUserOwnerOrReadOnly` customized permission class we created. The following lines show a sample response:

```
HTTP/1.0 403 Forbidden
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Length: 63
Content-Type: application/json
Date: Fri, 10 Nov 2017 03:34:43 GMT
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
    "detail": "You do not have permission to perform this action."
}
```

The user that isn't the drone's owner cannot make changes to the drone. However, he must be able to have read-only access to the drone. Hence, we must be able to compose and retrieve the previous drone details with an `HTTP GET` request with the same authentication credentials. It will work because `GET` is one of the safe methods and a user that is not the owner is allowed to read the resource. Replace `user01` and `user01password` in the next command with the name and password you configured for this user. In addition, replace `12` with the ID generated for the previously created drone in your configuration:

```
http -a "user01":"user01password" GET :8000/drones/12
```



The following is the equivalent `curl` command:

```
curl --user "user01":"user01password" -iX GET  
localhost:8000/drones/12
```



The response will return an `HTTP 200 OK` status code in the header and the requested `drone` serialized to JSON in the response body.

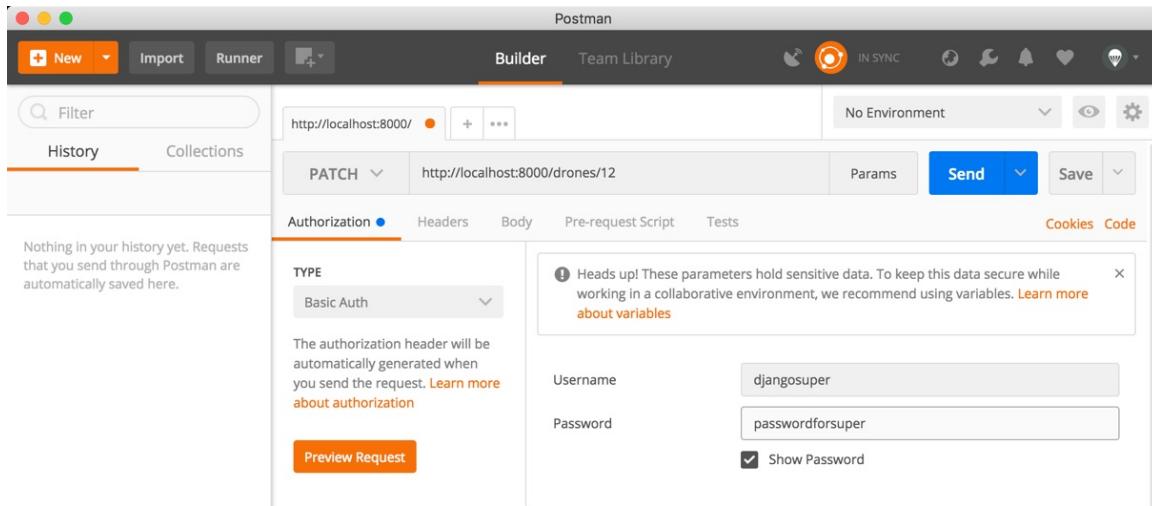
Making authenticated HTTP PATCH requests with Postman

Now, we will use one of the GUI tools we installed in *Chapter 1, Installing the Required Software and Tools*, specifically, Postman. We will use this GUI tool to compose and send an HTTP `PATCH` request with the appropriate authentication credentials to the web service. In the previous chapters, whenever we worked with Postman, we didn't specify authentication credentials.

We will use the Builder tab in Postman to compose and send an HTTP `PATCH` request to update the `has_it_competed` field for the previously created drone. Follow these steps:

1. In case you made previous requests with Postman, click on the plus (+) button at the right-hand side of the tab that displayed the previous request. This way, you will create a new tab.
2. Select PATCH in the drop-down menu at the left-hand side of the Enter request URL textbox.
3. Enter `http://localhost:8000/drones/12` in that textbox at the right-hand side of the drop-down. Replace `12` with the ID generated for the previously created drone in your configuration.
4. Click the Authorization tab below the textbox.
5. Select Basic Auth in the TYPE drop-down.
6. Enter the name you used to create `djangosuper` in the Username textbox.
7. Enter the password you used instead of `passwordforsuper` for this user in the Password textbox. The following screenshot shows the

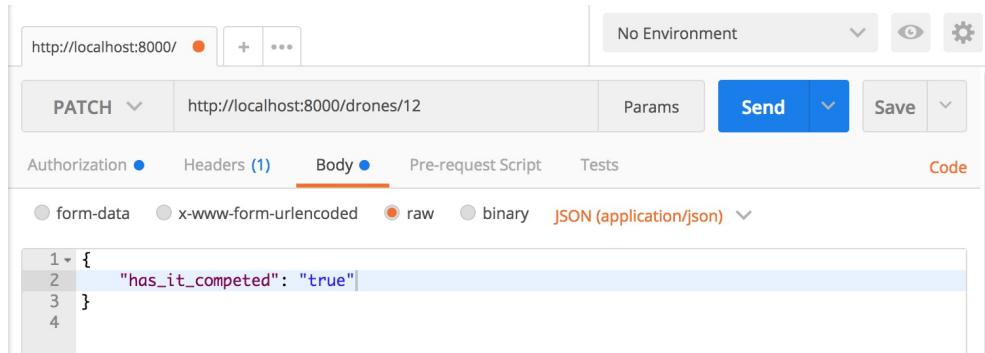
basic authentication configured in Postman for the HTTP PATCH request:



8. Click Body at the right-hand side of the Authorization and Headers tabs, within the panel that composes the request.
9. Activate the raw radio button and select JSON (application/json) in the drop-down at the right-hand side of the binary radio button. Postman will automatically add a Content-type = application/json header, and therefore, you will notice the Headers tab will be renamed to Headers (1), indicating to us that there is one key/value pair specified for the request headers.
10. Enter the following lines in the textbox below the radio buttons, within the Body tab:

```
{  
  "has_it_competed": "true"  
}
```

The following screenshot shows the request body in Postman:



We followed the necessary steps to create an HTTP `PATCH` request with a JSON body that specifies the necessary key/value pairs to update the value for the `was_included_in_home` field of an existing drone, with the necessary HTTP authentication credentials. Click Send and Postman will display the following information:

- Status: 200 OK
- Time: The time it took for the request to be processed
- Size: The approximate response size (sum of body size plus headers size)
- Body: The response body with the recently updated drone formatted as JSON with syntax highlighting

The following screenshot shows the JSON response body in Postman for the HTTP `PATCH` request. In this case, the request updated the existing drone because we authenticated the request with the user that is the drone's owner:

Body Cookies Headers (7) Test Results Status: 200 OK Time: 311 ms Size: 486 B

Pretty Raw Preview JSON 🔍

```
1 {  
2   "url": "http://localhost:8000/drones/12",  
3   "name": "Python Drone",  
4   "drone_category": "Quadcopter",  
5   "owner": "djangosuper",  
6   "manufacturing_date": "2017-07-16T02:03:00.716312Z",  
7   "has_it_competed": false,  
8   "inserted_timestamp": "2017-11-10T02:55:07.361574Z"  
9 }
```

Browsing the secured API with the required authentication

We want the browsable API to display the log in and log out views. In order to make this possible, we have to add a line in the `urls.py` file in the `restful01/restful01` folder, specifically, in the `restful01/restful01/urls.py` file. The file defines the root URL configurations and we want to include the URL patterns provided by the Django REST framework that provide the log in and log out views.

The following lines show the new code for the `restful01/restful01/urls.py` file. The new line is highlighted. The code file for the sample is included in the `hillar_django_restful_08_01` folder, in the `restful01/restful01/urls.py` file:

```
from django.conf.urls import url, include

urlpatterns = [
    url(r'^$', include('drones.urls')),
    url(r'^api-auth/', include('rest_framework.urls'))
]
```

Open a web browser and go to `http://localhost:8000/`. Replace localhost by the IP of the computer that is running Django's development server in case you use another computer or device to run the browser. The browsable API will compose and send a `GET` request to `/` and will display the results of its execution, that is, the Api Root. You will notice there is a Log in hyperlink at the upper-right corner.

Click or tap Log in and the browser will display the Django REST framework login page. Enter the name you used to create `djangosuper` in the Username textbox and the password you used instead of `passwordforsuper` for this user in the Password textbox. Then, click Log in.

Now, you will be logged in as `djangosuper` and all the requests you compose and send through the browsable API will use this user. You will be redirected again to the Api Root and you will notice the Log in hyperlink is replaced with the username (`djangosuper`) and a drop-down menu that allows you to log out. The following screenshot shows the Api Root after we are logged in as `djangosuper`:

The screenshot shows the Django REST framework browsable API interface. At the top, there is a dark header bar with the text "Django REST framework" on the left and "djangosuper" with a dropdown arrow on the right. Below the header, the title "Api Root" is displayed. To the right of the title are two buttons: "OPTIONS" and "GET" with a dropdown arrow. Underneath the title, there is a section labeled "GET /". Inside this section, the response status is "HTTP 200 OK" and the response headers are listed: "Allow: GET, HEAD, OPTIONS", "Content-Type: application/json", and "Vary: Accept". The response body is a JSON object with the following structure:

```
{  
    "drone-categories": "http://localhost:8000/drone-categories/",  
    "drones": "http://localhost:8000/drones/",  
    "pilots": "http://localhost:8000/pilots/",  
    "competitions": "http://localhost:8000/competitions/"  
}
```

Click or tap on the username that is logged in (`djangosuper`) and select Log Out from the drop-down menu. We will log in as a different user.

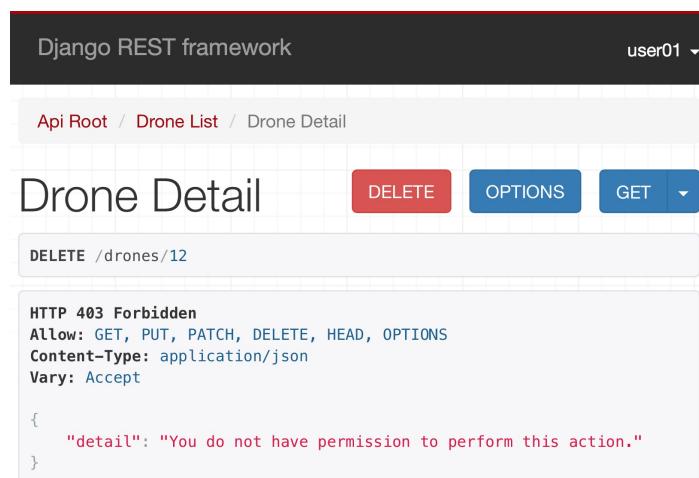
Click or tap Log in and the browser will display the Django REST framework login page. Enter the name you used to create `user01` in the Username textbox and the password you used instead of `user01password` for this user in the Password textbox. Then, click Log in.

Now, you will be logged in as `user01` and all the requests you compose and send through the browsable API will use this user. You will be redirected again to the Api Root and you will notice the Log in hyperlink is replaced with the username (`user01`).

Go to `http://localhost:8000/drones/12`. Replace `12` with the ID generated for the previously created drone in your configuration. The browsable API will render the web page with the results for the `GET` request to `localhost:8000/drones/12`.

Click or tap the OPTIONS button and the browsable API will render the results of the HTTP `OPTIONS` request to `http://localhost:8000/drones/12` and will include the DELETE button at the right-hand side of the Drone Detail title.

Click or tap DELETE. The web browser will display a confirmation modal. Click or tap the DELETE button in the modal. As a result of the HTTP `DELETE` request, the web browser will display an `HTTP 403 Forbidden` status code in the response header and a detail message indicating that we do not have permission to perform the action in the JSON body. The owner for the drone we want to delete is `djangosuper` and the authentication credentials for this request use a different user, specifically, `user01`. Hence, the operation is rejected by the `has_object_permission` method in the `IsCurrentUserOwnerOrReadOnly` class. The following screenshot shows a sample response for the HTTP `DELETE` request:



The browsable API makes it easy to compose and send authenticated requests to our RESTful Web Service.

Working with token-based authentication

Now, we will make changes to use token-based authentication to retrieve, update, or delete pilots. Only those users that have a token will be able to make these operations with pilots. Hence, we will setup a specific authentication for pilots. It will still be possible to see the pilot's name rendered in unauthenticated requests.

The token-based authentication requires a new model named `Token`. Make sure you quit the Django's development server. Remember that you just need to press `Ctrl + C` in the terminal or command prompt window in which it is running.

Of course, in a production environment, we must make sure that the RESTful Web Service is only available over HTTPS, with the usage of the latest TLS versions. We shouldn't use a token-based authentication over plain HTTP in a production environment.

Open the `restful01/restful01/settings.py` file that declares module-level variables that define the configuration of Django for the `restful01` project. Locate the lines that assign a strings list to `INSTALLED_APPS` to declare the installed apps. Add the following string to the `INSTALLED_APPS` strings list and save the changes to the `settings.py` file:

```
'rest_framework.authtoken'
```

The following lines show the new code that declares the `INSTALLED_APPS` strings list with the added line highlighted and with comments to understand what each added string means. The code file for the sample is

included in the `hillar_django_restful_08_02` folder in the `restful01/restful01/settings.py` file:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Django REST framework
    'rest_framework',
    # Drones application
    'drones.apps.DronesConfig',
    # Django Filters,
    'django_filters',
    # Token authentication
    'rest_framework.authtoken',
]
```

This way, we have added the `rest_framework.authtoken` application to our Django project named `restful01`.

Now, run the following Python script to execute all migrations required for the recently added `authtoken` application and apply the changes in the underlying database tables. This way, we will install the app:

```
python manage.py migrate
```

The following lines show the output generated after running the previous command. Notice that the order in which the migrations are executed can differ in your development computer:

```
Operations to perform:
  Apply all migrations: admin, auth, authtoken, contenttypes,
  drones, sessions
  Running migrations:
    Applying authtoken.0001_initial... OK
    Applying authtoken.0002_auto_20160226_1747... OK
```



After we run the previous command, we will have a new `authtoken_token` table in the PostgreSQL database. This table will persist the generated tokens and has a foreign key to the `auth_user` table.

We will configure authentication and permission policies for the class-based views that work with the `Pilot` model. We will override the values for the `authentication_classes` and `permission_classes` class attributes for the `PilotDetail` and `PilotList` classes.

We will add the same lines of code in the two classes. We will include the `TokenAuthentication` authentication class in the `authentication_classes` tuple, and the `IsAuthenticated` permission class in the `permission_classes` tuple.

Open the `restful01/drones/views.py` file and add the following lines after the last line that declares the imports, before the declaration of the `DroneCategorySerializer` class. The code file for the sample is included in the `hillar_django_restful_08_02` folder, in the `restful01/drones/views.py` file:

```
from rest_framework.permissions import IsAuthenticated
from rest_framework.authentication import TokenAuthentication
```

Replace the code that declares the `PilotDetail` class with the following code in the same `views.py` file. The new lines are highlighted in the code listing. The code file for the sample is included in the `hillar_django_restful_08_02` folder, in the `restful01/drones/views.py` file:

```
class PilotDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Pilot.objects.all()
    serializer_class = PilotSerializer
    name = 'pilot-detail'
    authentication_classes = (
        TokenAuthentication,
    )
    permission_classes = (
        IsAuthenticated,
    )
```

Replace the code that declares the `PilotList` class with the following code in the same `views.py` file. The new lines are highlighted in the code listing. The code file for the sample is included in the `hillar_django_restful_08_02` folder, in the `restful01/drones/views.py` file:

```
class PilotList(generics.ListCreateAPIView):
    queryset = Pilot.objects.all()
    serializer_class = PilotSerializer
    name = 'pilot-list'
    filter_fields = (
        'name',
        'gender',
        'races_count',
    )
    search_fields = (
        '^name',
    )
    ordering_fields = (
        'name',
        'races_count'
    )
    authentication_classes = (
        TokenAuthentication,
    )
    permission_classes = (
        IsAuthenticated,
    )
```



Generating and using tokens

Now, we will launch our default Python interactive shell in our virtual environment and make all the Django project modules available to write code that will generate a token for an existing user. We will do this to understand how the token generation works.

Run the following command to launch the interactive shell. Make sure you are within the `restful01` folder in the terminal, Command Prompt, or Windows Powershell:

```
python manage.py shell
```

You will notice that a line that says (InteractiveConsole) is displayed after the usual lines that introduce your default Python interactive shell. Enter the following code in the Python interactive shell to import all the things we will need to retrieve a `User` instance and generate a new token. The code file for the sample is included in the `hillar_django_restful_08_02` folder, in the `restful01/tokens_test_01.py` file.

```
from rest_framework.authtoken.models import Token
from django.contrib.auth.models import User
```

Enter the following code to retrieve an instance of the `User` model whose username matches "user01" and create a new `Token` instance related to this user. The last line prints the value for the `key` attribute for the generated `Token` instance saved in the `token` variable. Replace `user01` in the next lines with the name you configured for this user. The code file for the sample is included in the `hillar_django_restful_08_02` folder, in the `restful01/tokens_test_01.py` file:

```
# Replace user01 with the name you configured for this user
user = User.objects.get(username="user01")
token = Token.objects.create(user=user)
print(token.key)
```

The following line shows a sample output from the previous code with the string value for `token.key`. Copy the output generated when running the code because we will use this token to authenticate requests. Notice that the token generated in your system will be different:

```
ebebe08f5d7fe5997f9ed1761923ec5d3e461dc3
```

Finally, enter the following command to quit the interactive console:

```
quit()
```

Now, we have a token for the Django user named `user01`.

Now, we can launch Django's development server to compose and send HTTP requests to retrieve pilots to understand how the configured token authentication class combined with the permission policies work. Execute any of the following two commands based on your needs to access the API in other devices or computers connected to your LAN. Remember that we analyzed the difference between them in [chapter 3, *Creating API Views*](#), in the *Launching Django's development server* section:

```
python manage.py runserver
python manage.py runserver 0.0.0.0:8000
```

After we run any of the previous commands, the development server will start listening at port 8000.

We will compose and send an HTTP `GET` request without authentication credentials to try to retrieve the first page of the `pilots` collection:

```
http :8000/pilots/
```

The following is the equivalent `curl` command:

```
curl -iX GET localhost:8000/pilots/
```

We will receive an `HTTP 401 Unauthorized` status code in the response header and a detail message indicating that we didn't provide authentication credentials in the JSON body. In addition, the value for the `WWW-Authenticate` header specifies the authentication method that must be applied to access the resource collection: `Token`. The following lines show a sample response:

```
HTTP/1.0 401 Unauthorized
Allow: GET, POST, HEAD, OPTIONS
Content-Length: 58
Content-Type: application/json
Date: Sat, 18 Nov 2017 02:28:31 GMT
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Accept
WWW-Authenticate: Token
X-Frame-Options: SAMEORIGIN

{
    "detail": "Authentication credentials were not provided."
}
```

After the changes we made, if we want to retrieve the collection of pilots, that is, to make an `HTTP GET` request to `/pilots/`, we need to provide authentication credentials by using the token-based authentication. Now, we will compose and send an `HTTP` request to retrieve the collection of pilots with authentication credentials, that is, with the token. Remember to replace `PASTE-TOKEN-HERE` with the previously generated token:

```
http :8000/pilots/ "Authorization: Token PASTE-TOKEN-HERE"
```

The following is the equivalent `curl` command:

```
curl -iX GET http://localhost:8000/pilots/ -H "Authorization: Token  
PASTE-TOKEN-HERE"
```

As a result of the request, we will receive an `HTTP 200 OK` status code in the response header and the first page of the pilots collection serialized to JSON in the response body. The following screenshot shows the first lines of a sample response for the request with the appropriate token:

```
(0) Gastons-MacBook-Pro:restful01 gaston$ http :8000/pilots/ 'Authorization: Token ebebe08f5d7fe5997f9ed1761923ec5d3e461dc3'  
HTTP/1.0 200 OK  
Allow: GET, POST, HEAD, OPTIONS  
Content-Length: 1570  
Content-Type: application/json  
Date: Sat, 18 Nov 2017 02:56:07 GMT  
Server: WSGIServer/0.2 CPython/3.6.2  
Vary: Accept  
X-Frame-Options: SAMEORIGIN  
  
{"count": 2,  
 "next": null,  
 "previous": null,  
 "results": [  
     {"  
         "competitions": [  
             {"  
                 "distance_achievement_date": "2017-10-21T06:02:23.776594Z",  
                 "distance_in_feet": 2800,  
                 "drone": {  
                     "drone_category": "Quadcopter",  
                     "has_it_competed": false,  
                     "inserted_timestamp": "2017-11-03T01:58:49.135737Z",  
                     "manufacturing_date": "2017-07-20T02:02:00.716312Z",  
                     "name": "WonderDrone",  
                     "owner": "djangosuper",  
                     "url": "http://localhost:8000/drones/1"  
                 },  
                 "pk": 2,  
                 "url": "http://localhost:8000/competitions/2"  
             },  
             {"  
                 "distance_achievement_date": "2017-10-20T05:03:20.776594Z",  
                 "distance_in_feet": 800,  
                 "drone": {  
                     "drone_category": "Quadcopter",  
                     "has_it_competed": false,  
                     "inserted_timestamp": "2017-11-03T01:58:49.135737Z",  
                     "manufacturing_date": "2017-07-20T02:02:00.716312Z",  
                     "name": "WonderDrone",  
                     "owner": "djangosuper",  
                     "url": "http://localhost:8000/drones/2"  
             }  
         ]  
     }  
 ]  
 }
```

The token-based authentication provided with the Django REST framework is very simple and it requires customization to make it production ready. Tokens never expire and there is no setting to specify the default expiration time for a token.

Test your knowledge

Let's see whether you can answer the following questions correctly.

1. The `permissions.SAFE_METHODS` tuple of string includes the following HTTP methods or verbs that are considered safe:
 1. 'GET', 'HEAD', and 'OPTIONS'
 2. 'POST', 'PATCH', and 'OPTIONS'
 3. 'GET', 'PUT', and 'OPTIONS'
2. Which of the following settings key in the `REST_FRAMEWORK` dictionary specifies the global setting with a tuple of string whose values indicate the classes that we want to use for authentication?
 1. 'GLOBAL_AUTHENTICATION_CLASSES'
 2. 'DEFAULT_AUTHENTICATION_CLASSES'
 3. 'REST_FRAMEWORK_AUTHENTICATION_CLASSES'
3. Which of the following is the model that persists a Django user?
 1. `Django.contrib.auth.DjangoUser`
 2. `Django.contrib.auth.User`
 3. `Django.rest-framework.User`
4. Which of the following classes is the base class from which all

customized permission classes should inherit to work with the Django REST framework?

1. `Django.contrib.auth.MainPermission`
 2. `rest_framework.permissions.MainPermission`
 3. `rest_framework.permissions.BasePermission`
5. In order to configure permission policies for a class-based view, which of the following class attributes do we have to override?
1. `permission_classes`
 2. `permission_policies_classes`
 3. `rest_framework_permission_classes`

The rights answers are included in the [Appendix](#), *Solutions*.

Summary

In this chapter, we learned the differences between authentication and permissions in Django, the Django REST framework, and RESTful Web Services. We analyzed the authentication classes included in the Django REST framework out of the box.

We followed the necessary steps to include security and permissions-related data to models. We worked with object-level permissions via customized permission classes and we saved information about users that make requests. We understood that there are three HTTP methods or verbs that are considered safe.

We configured permission policies for the class-based views that worked with the `Drone` model. Then, we created a superuser and another user for Django to compose and send authenticated requests and to understand how the permission policies we configured were working.

We used command-line tools and GUI tools to compose and send authenticated requests. Then, we browsed the secured RESTful Web Service with the browsable API feature. Finally, we worked with a simple token-based authentication provided by the Django REST framework to understand another way of authenticating requests.

Now that we have improved our RESTful Web Service with authentication and permission policies, it is time to combine these policies with throttling rules and versioning. We will cover these topics in the next chapter.

Applying Throttling Rules and Versioning Management

In this chapter, we will work with throttling rules to limit the usage of our RESTful Web Service. We don't want to process requests until our RESTful Web Service runs out of resources, and therefore, we will analyze the importance of throttling rules. We will take advantage of the features included in the Django REST framework to manage different versions of our web service. We will gain an understanding of:

- Understanding the importance of throttling rules
- Learning the purpose of the different throttling classes in the Django REST framework
- Configuring throttling policies in the Django REST framework
- Running tests to check that throttling policies work as expected
- Understanding versioning classes
- Configuring the versioning scheme
- Running tests to check that versioning works as expected

Understanding the importance of throttling rules

In [Chapter 8, Securing the API with Authentication and Permissions](#), we made sure that some requests were authenticated before processing them. We took advantage of many authentication schemes to identify the user that originated the request. Throttling rules also determine whether the request must be authorized or not. We will work with them in combination with authentication.

So far, we haven't established any limits on the usage of our RESTful Web Service. As a result of this configuration, both unauthenticated and authenticated users can compose and send as many requests as they want to. The only thing we have limited is the resultset size throughout the configuration of the pagination features available in the Django REST framework. Hence, large results sets are split into individual pages of data. However, a user might compose and send thousands of requests to be processed with any kind of limitation. Of course, the servers or virtual machines that run our RESTful Web Services or the underlying database can be overloaded by the huge amount of requests because we don't have limits.

Throttles control the rate of requests that users can make to our RESTful Web Service. The Django REST framework makes it easy to configure throttling rules. We will use throttling rules to configure the following limitations to the usage of our RESTful Web Service:

- A maximum of 3 requests per hour for unauthenticated users
- A maximum of 10 requests per hour for authenticated users
- A maximum of 20 requests per hour for the drones related views

- A maximum of 15 requests per hour for the pilots related views

Learning the purpose of the different throttling classes in the Django REST framework

The Django REST framework provides three throttling classes in the `rest_framework.throttling` module. All of them are subclasses of the `SimpleRateThrottle` class which inherits from the `BaseThrottle` class.

The three classes allow us to specify throttling rules that indicate the maximum number of requests in a specific period of time and within a determined scope. Each class is responsible for computing and validating the maximum number of requests per period. The classes provide different mechanisms to determine the previous request information to specify the scope by comparing it with the new request. The Django REST framework stores the required data to analyze each throttling rule in the cache. Thus, the classes override the inherited `get_cache_key` method that determines the scope that will be used for computing and validating.

The following are the three throttling classes:

- `AnonRateThrottle`: This class limits the rate of requests that an anonymous user can make, and therefore, its rules apply to unauthenticated users. The unique cache key is the IP address of the incoming request. Hence, all the requests originated in the same IP address will accumulate the total number of requests for this IP.
- `UserRateThrottle`: This class limits the rate of requests that a specific user can make and applies to both authenticated and non-

authenticated users. Obviously, when the requests are authenticated, the authenticated user ID is the unique cache key. When the requests are unauthenticated and come from anonymous users, the unique cache key is the IP address of the incoming request.

- `ScopedRateThrottle`: This class is useful whenever we have to restrict access to specific features of our RESTful Web Service with different rates. The class uses the value assigned to the `throttle_scope` attribute to limit requests to the parts that are identified with the same value.

The previous classes are included in the Django REST framework out of the box. There are many additional throttling classes provided by many third-party libraries.

Make sure you quit the Django's development server. Remember that you just need to press *Ctrl + C* in the terminal or Command Prompt window in which it is running. We will make the necessary changes to combine the different authentication mechanisms we set up in the previous chapter with the application of throttling rules. Hence, we will add the `AnonRateThrottle` and `UserRateThrottle` classes in the global throttling classes list.

The value for the `DEFAULT_THROTTLE_CLASSES` settings key specifies a global setting with a tuple of string whose values indicate the default classes that we want to use for throttling rules. We will specify the `AnonRateThrottle` and `UserRateThrottle` classes.

The `DEFAULT_THROTTLE_RATES` settings key specifies a dictionary with the default throttle rates. The next list specifies the keys, the values that we will assign and their meaning:

- 'anon': We will specify '3/hour' as the value for this key, which

means we want a maximum of 3 requests per hour for anonymous users. The `AnonRateThrottle` class will apply this throttling rule.

- 'user': We will specify '`10/hour`' as the value for this key, which means we want a maximum of 10 requests per hour for authenticated users. The `UserRateThrottle` class will apply this throttling rule.
- 'drones': We will specify '`20/hour`' as the value for this key, which means we want a maximum of 20 requests per hour for the drones-related views. The `ScopedRateThrottle` class will apply this throttling rule.
- 'pilots': We will specify '`15/hour`' as the value for this key, which means we want a maximum of 15 requests per hour for the drones-related views. The `ScopedRateThrottle` class will apply this throttling rule.

The maximum rate value for each key is a string that specifies the number of requests per period with the following format: '`number_of_requests/period`', where `period` can be any of the following:

- `d: day`
- `day: day`
- `h: hour`
- `hour: hour`
- `m: minute`
- `min: minute`
- `s: second`

- sec: second

In this case, we will always work with a maximum number of requests per hour, and therefore, the values will use /hour after the maximum number of requests.

Open the `restful01/restful01/settings.py` file that declares module-level variables that define the configuration of Django for the `restful01` project. We will make some changes to this Django settings file. Add the highlighted lines to the `REST_FRAMEWORK` dictionary. The following lines show the new declaration of the `REST_FRAMEWORK` dictionary. The code file for the sample is included in the `hillar_django_restful_09_01` folder in the `restful01/restful01/settings.py` file:

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
        'drones.custompagination.LimitOffsetPaginationWithUpperBound',
    'PAGE_SIZE': 4,
    'DEFAULT_FILTER_BACKENDS': (
        'django_filters.rest_framework.DjangoFilterBackend',
        'rest_framework.filters.OrderingFilter',
        'rest_framework.filters.SearchFilter',
    ),
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.BasicAuthentication',
        'rest_framework.authentication.SessionAuthentication',
    ),
    'DEFAULT_THROTTLE_CLASSES': (
        'rest_framework.throttling.AnonRateThrottle',
        'rest_framework.throttling.UserRateThrottle',
    ),
    'DEFAULT_THROTTLE_RATES': {
        'anon': '3/hour',
        'user': '10/hour',
        'drones': '20/hour',
        'pilots': '15/hour',
    }
}
```

We added values for the `DEFAULT_THROTTLE_CLASSES` and the `DEFAULT_THROTTLE_RATES` settings keys to configure the default throttling classes and the desired

rates.

Configuring throttling policies in the Django REST framework

Now, we will configure throttling policies for the class-based views related to drones: `DroneList` and `DroneDetail`. We will override the values for the following class attributes for the class-based views:

- `throttle_classes`: This class attribute specifies a tuple with the names of the classes that will manage throttling rules for the class. In this case, we will specify the `ScopedRateThrottle` class as the only member of the tuple.
- `throttle_scope`: This class attribute specifies the throttle scope name that the `ScopedRateThrottle` class will use to accumulate the number of requests and limit the rate of requests.

This way, we will make these class-based views work with the `ScopedRateThrottle` class and we will configure the throttle scope that this class will consider for each of the class based views related to drones.

Open the `restful01/drones/views.py` file and add the following lines after the last line that declares the imports, before the declaration of the `DroneCategoryList` class:

```
from rest_framework.throttling import ScopedRateThrottle
```

Replace the code that declares the `DroneDetail` class with the following code in the same `views.py` file. The new lines are highlighted in the code listing. The code file for the sample is included in the `hillar_django_restful_09_01`

folder, in the `restful01/drones/views.py` file:

```
class DroneDetail(generics.RetrieveUpdateDestroyAPIView):
    throttle_scope = 'drones'
    throttle_classes = (ScopedRateThrottle,)
    queryset = Drone.objects.all()
    serializer_class = DroneSerializer
    name = 'drone-detail'
    permission_classes = (
        permissions.IsAuthenticatedOrReadOnly,
        custompermission.IsCurrentUserOwnerOrReadOnly,
    )
```

Replace the code that declares the `DroneList` class with the following code in the same `views.py` file. The new lines are highlighted in the code listing. The code file for the sample is included in the `hillar_django_restful_09_01` folder, in the `restful01/drones/views.py` file:

```
class DroneList(generics.ListCreateAPIView):
    throttle_scope = 'drones'
    throttle_classes = (ScopedRateThrottle,)
    queryset = Drone.objects.all()
    serializer_class = DroneSerializer
    name = 'drone-list'
    filter_fields = (
        'name',
        'drone_category',
        'manufacturing_date',
        'has_it_competed',
    )
    search_fields = (
        '^name',
    )
    ordering_fields = (
        'name',
        'manufacturing_date',
    )
    permission_classes = (
        permissions.IsAuthenticatedOrReadOnly,
        custompermission.IsCurrentUserOwnerOrReadOnly,
    )

    def perform_create(self, serializer):
        serializer.save(owner=self.request.user)
```

We added the same lines in the two classes. We assigned 'drones' to the `throttle_scope` class attribute and we included `ScopedRateThrottle` in the tuple that defines the value for `throttle_classes`. This way, the two class-based views will use the settings specified for the 'drones' scope and the `ScopeRateThrottle` class for throttling. We added the 'drones' key to the `DEFAULT_THROTTLE_RATES` key in the `REST_FRAMEWORK` dictionary, and therefore, the 'drones' scope is configured to serve a maximum of 20 requests per hour.

Now, we will configure throttling policies for the class-based views related to pilots: `PilotList` and `PilotDetail`. We will also override the values for the `throttle_scope` and `throttle_classes` class attributes.

Replace the code that declares the `PilotDetail` class with the following code in the same `views.py` file. The new lines are highlighted in the code listing. The code file for the sample is included in the `hillar_django_restful_09_01` folder, in the `restful01/drones/views.py` file:

```
class PilotDetail(generics.RetrieveUpdateDestroyAPIView):
    throttle_scope = 'pilots'
    throttle_classes = (ScopedRateThrottle,)
    queryset = Pilot.objects.all()
    serializer_class = PilotSerializer
    name = 'pilot-detail'
    authentication_classes = (
        TokenAuthentication,
    )
    permission_classes = (
        IsAuthenticated,
    )
```

Replace the code that declares the `PilotList` class with the following code in the same `views.py` file. The new lines are highlighted in the code listing. The code file for the sample is included in the `hillar_django_restful_09_01` folder, in the `restful01/drones/views.py` file:

```
class PilotList(generics.ListCreateAPIView):
    throttle_scope = 'pilots'
```

```
throttle_classes = (ScopedRateThrottle,)
queryset = Pilot.objects.all()
serializer_class = PilotSerializer
name = 'pilot-list'
filter_fields = (
    'name',
    'gender',
    'races_count',
)
search_fields = (
    '^name',
)
ordering_fields = (
    'name',
    'races_count'
)
authentication_classes = (
    TokenAuthentication,
)
permission_classes = (
    IsAuthenticated,
)
```

We added the same lines in the two classes. We assigned 'pilots' to the `throttle_scope` class attribute and we included `ScopedRateThrottle` in the tuple that defines the value for `throttle_classes`. This way, the two class-based views will use the settings specified for the 'pilots' scope and the `ScopeRateThrottle` class for throttling. We added the 'pilots' key to the `DEFAULT_THROTTLE_RATES` key in the `REST_FRAMEWORK` dictionary, and therefore, the 'drones' scope is configured to serve a maximum of 15 requests per hour.

All the class-based views we have edited won't take into account the global settings that applied the default classes that we use for throttling: `AnonRateThrottle` and `UserRateThrottle`. These class-based views will use the configuration we have specified for them.

Running tests to check that throttling policies work as expected

Before Django runs the main body of a class-based view, it performs the checks for each throttle class specified in the throttle classes settings. In the drones and pilots-related views, we wrote code that overrides the default settings.

If a single throttle check fails, the code will raise a `Throttled` exception and Django won't execute the main body of the view. The cache is responsible for storing previous request information for throttling checking.

Now, we can launch Django's development server to compose and send HTTP requests to understand how the configured throttling rules, combined with all the previous configurations, work. Execute any of the following two commands based on your needs to access the API in other devices or computers connected to your LAN. Remember that we analyzed the difference between them in [chapter 3, *Creating API Views*](#), in the *Launching Django's development server* section.

```
python manage.py runserver
python manage.py runserver 0.0.0.0:8000
```

After we run any of the previous commands, the development server will start listening at port 8000.

Now, we will compose and send the following HTTP `GET` request without authentication credentials to retrieve the first page of the competitions four times:

```
http :8000/competitions/
```

We can also use the features of the shell in macOS or Linux to run the previous command four times with just a single line with a bash shell. The command is compatible with a Cygwin terminal in Windows. We must take into account that we will see all the results one after the other and we will have to scroll to understand what happened with each execution:

```
for i in {1..4}; do http :8000/competitions/; done;
```

The following line allows you to run the command four times with a single line in Windows PowerShell:

```
1..4 | foreach { http :8000/competitions/ }
```

The following is the equivalent curl command that we must execute four times:

```
curl -iX GET localhost:8000/competitions/
```

The following is the equivalent curl command that is executed four times with a single line in a bash shell in a macOS or Linux, or a Cygwin terminal in Windows:

```
for i in {1..4}; do curl -iX GET localhost:8000/competitions/; done;
```

The following is the equivalent curl command that is executed four times with a single line in Windows PowerShell:

```
1..4 | foreach { curl -iX GET localhost:8000/competitions/ }
```

The Django REST framework won't process the request number 4. The `AnonRateThrottle` class is configured as one of the default throttle classes and its throttle settings specify a maximum of 3 requests per hour. Hence, we will receive an `HTTP 429 Too many requests` status code in the response header and a message indicating that the request was throttled and the time in which the server will be able to process an additional request. The value for the `Retry-After` key in the response header provides the number of seconds that we must wait until the next request: 2347. The following lines show a sample response. Notice that the number of seconds might be different in your configuration:

```
HTTP/1.0 429 Too Many Requests
Allow: GET, POST, HEAD, OPTIONS
Content-Length: 71
Content-Type: application/json
Date: Thu, 30 Nov 2017 03:07:28 GMT
Retry-After: 2347
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
    "detail": "Request was throttled. Expected available in 2347 seconds."
}
```

Now, we will compose and send the following `HTTP GET` request with authentication credentials to retrieve the first page of the competitions four times. We will use the superuser we created in the previous chapter. Remember to replace `djangosuper` with the name you used for the superuser and `passwordforsuper` with the password you configured for this user as shown here:

```
http -a "djangosuper":"passwordforsuper" :8000/competitions/
```

In a Linux, macOS or a Cygwin terminal, we can run the previous command four times with the following single line:

```
for i in {1..4}; do http -a "djangosuper":"passwordforsuper" :8000/competiti
```

The following line allows you to run the command four times with a single line in Windows PowerShell.

```
1..4 | foreach { http -a "djangosuper":"passwordforsuper" :8000/competitions
```

The following is the equivalent curl command that we must execute four times:

```
curl --user 'djangosuper':'passwordforsuper' -iX GET localhost:8000/competit
```

The following is the equivalent curl command that we can execute four times in a Linux, macOS or a Cygwin terminal with a single line:

```
for i in {1..4}; do curl --user "djangosuper":"passwordforsuper" -iX GET loc
```

The following is the equivalent curl command that is executed four times with a single line in Windows PowerShell:

```
1..4 | foreach { curl --user "djangosuper":"passwordforsuper" -iX GET localh
```

In this case, Django will process the request number 4 because we have composed and sent 4 authenticated requests with the same user. The `UserRateThrottle` class is configured as one of the default throttle classes and its throttle settings specify 10 requests per hour. We still have 6 requests before we accumulate the maximum number of requests per hour.

If we compose and send the same request 7 times more, we will accumulate 11 requests and we will receive an HTTP `429 Too many requests` status code in the response header, a message indicating that the request was throttled and the time in which the server will be able to process an additional request after the last execution.

Now, we will compose and send the following HTTP `GET` request without authentication credentials to retrieve the first page of the drones collection 20 times:

```
http :8000/drones/
```

We can use the features of the shell in macOS or Linux to run the previous command 20 times with just a single line with a bash shell. The command is compatible with a Cygwin terminal in Windows:

```
for i in {1..20}; do http :8000/drones/; done;
```

The following line allows you to run the command 20 times with a single line in Windows PowerShell:

```
1..21 | foreach { http :8000/drones/ }
```

The following is the equivalent curl command that we must execute 20 times:

```
curl -iX GET localhost:8000/drones/
```

The following is the equivalent curl command that is executed 20 times with a single line in a bash shell in macOS or Linux, or a Cygwin terminal in Windows:

```
for i in {1..21}; do curl -iX GET localhost:8000/drones/; done;
```

The following is the equivalent curl command that is executed 20 times with a single line in Windows PowerShell:

```
1..20 | foreach { curl -iX GET localhost:8000/drones/ }
```

The Django REST framework will process the 20 requests. The `DroneList` class has its `throttle_scope` class attribute set to `'drones'` and uses the `ScopedRateThrottle` class to accumulate the requests in the specified scope. The `'drones'` scope is configured to accept a maximum of 20 requests per hour, and therefore, if we make another request with the same non-authenticated user and this request accumulates in the same scope, the request will be throttled.

Now, we will compose and send an HTTP `GET` request to retrieve the details for a drone. Make sure you replace `1` for any existing drone ID value that was listed in the results for the previous requests:

```
http :8000/drones/1
```

The following is the equivalent curl command:

```
curl -iX GET localhost:8000/drones/1
```

The Django REST framework won't process this request. The request ends up routed to the `DroneDetail` class. The `DroneDetail` class has its `throttle_scope` class attribute set to `'drones'` and uses the `ScopedRateThrottle` class to accumulate the requests in the specified scope. Thus, both the `DroneList` and the `DroneDetail` class accumulate in the same scope. The new request from the same non-authenticated user becomes the request number 21 for the `'drones'` scope that is configured to accept a maximum of 20 requests per

hour, and therefore, we will receive an HTTP 429 Too many requests status code in the response header and a message indicating that the request was throttled and the time in which the server will be able to process an additional request. The value for the `Retry-After` key in the response header provides the number of seconds that we must wait until the next request: 3138. The following lines show a sample response. Notice that the number of seconds might be different in your configuration:

```
HTTP/1.0 429 Too Many Requests
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Length: 71
Content-Type: application/json
Date: Mon, 04 Dec 2017 03:55:14 GMT
Retry-After: 3138
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
    "detail": "Request was throttled. Expected available in 3138 seconds."
}
```

Throttling rules are extremely important to make sure that users don't abuse our RESTful Web Service and that we keep control of the resources that are being used to process incoming requests. We should never put a RESTful Web Service in production without a clear configuration for throttling rules.

Understanding versioning classes

Sometimes, we have to keep many different versions of a RESTful Web Service alive at the same time. For example, we might need to have version 1 and version 2 of our RESTful Web Service accepting and processing requests. There are many versioning schemes that make it possible to serve many versions of a web service.

The Django REST framework provides five classes in the `rest_framework.versioning` module. All of them are subclasses of the `BaseVersioning` class. The five classes allow us to work with a specific versioning scheme.

We can use one of these classes in combination with changes in the URL configurations and other pieces of code to support the selected versioning scheme. Each class is responsible for determining the version based on the implemented schema and to make sure that the specified version number is a valid one based on the allowed version settings. The classes provide different mechanisms to determine the version number. The following are the five versioning classes:

- `AcceptHeaderVersioning`: This class configures a versioning scheme that requires each request to specify the desired version as an additional value of the media type specified as a value for the `Accept` key in the header. For example, if a request specifies '`application/json; version=1.2`' as the value for the `Accept` key in the header, the `AcceptHeaderVersioning` class will set the `request.version` attribute to '`1.2`'. This scheme is known as media type versioning, content negotiation versioning or accept header versioning.

- `HostNameVersioning`: This class configures a versioning scheme that requires each request to specify the desired version as a value included in the hostname in the URL. For example, if a request specifies `v2.myrestfulservice.com/drones/` as the URL, it means that the request wants to work with version number 2 of the RESTful Web Service. This scheme is known as hostname versioning or domain versioning.
- `URLPathVersioning`: This class configures a versioning scheme that requires each request to specify the desired version as a value included in the URL path. For example, if a request specifies `v2/myrestfulservice.com/drones/` as the URL, it means that the request wants to work with version number 2 of the RESTful Web Service. The class requires us to work with a `version` URL keyword argument. This scheme is known as URI versioning or URL path versioning.
- `NamespaceVersioning`: This class configures the versioning scheme explained for the `URLPathVersioning` class. The only difference compared with this other class is that the configuration in the Django REST framework application is different. In this case, it is necessary to use URL namespaces.
- `QueryParameterVersioning`: This class configures a versioning scheme that requires each request to specify the desired version as a query parameter. For example, if a request specifies `myrestfulservice.com/?version=1.2`, the `QueryParameterVersioning` class will set the `request.version` attribute to '`1.2`'. This scheme is known as query parameter versioning or request parameter versioning.

The previous classes are included in the Django REST framework out of

the box. It is also possible to code our own customized versioning scheme. Each versioning scheme has its advantages and trade-offs. In this case, we will work with the `NamespaceVersioning` class to provide a new version of the RESTful Web Service with a minor change compared to the first version. However, it is necessary to analyze carefully whether you really need to use any versioning scheme. Then, you need to figure out which is the most appropriate one based on your specific needs. Of course, if possible, we should always avoid any versioning scheme because they add complexity to our RESTful Web Service.

Configuring a versioning scheme

Let's imagine we have to serve the following two versions of our RESTful Web Service:

- **Version 1:** The version we have developed so far. However, we want to make sure that the clients understand that they are working with version 1, and therefore, we want to include a reference to the version number in the URL for each HTTP request.
- **Version 2:** This version has to allow clients to reference the `drones` resource collection with the `vehicles` name instead of `drones`. In addition, the drone categories resource collection must be accessed with the `vehicle-categories` name instead of `drone-categories`. We also want to make sure that the clients understand that they are working with version 2, and therefore, we want to include a reference to the version number in the URL for each HTTP request.

The difference between the second and the first version will be minimal because we want to keep the example simple. In this case, we will take advantage of the previously explained `NamespaceVersioning` class to configure a URL path versioning scheme.

Make sure you quit the Django's development server. Remember that you just need to press *Ctrl + C* in the terminal or command prompt window in

which it is running.

We will make the necessary changes to configure the usage of the `NameSpaceVersioning` class as the default versioning class for our RESTful Web Service. Open the `restful01/restful01/settings.py` file that declares module-level variables that define the configuration of Django for the `restful01` project. We will make some changes to this Django settings file. Add the highlighted lines to the `REST_FRAMEWORK` dictionary. The following lines show the new declaration of the `REST_FRAMEWORK` dictionary. The code file for the sample is included in the `hillar_django_restful_09_02` folder in the `restful01/restful01/settings.py` file:

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
        'drones.custompagination.LimitOffsetPaginationWithUpperBound',
    'PAGE_SIZE': 4,
    'DEFAULT_FILTER_BACKENDS': (
        'django_filters.rest_framework.DjangoFilterBackend',
        'rest_framework.filters.OrderingFilter',
        'rest_framework.filters.SearchFilter',
    ),
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.BasicAuthentication',
        'rest_framework.authentication.SessionAuthentication',
    ),
    'DEFAULT_THROTTLE_CLASSES': (
        'rest_framework.throttling.AnonRateThrottle',
        'rest_framework.throttling.UserRateThrottle',
    ),
    'DEFAULT_THROTTLE_RATES': {
        'anon': '3/hour',
        'user': '10/hour',
        'drones': '20/hour',
        'pilots': '15/hour',
    }
    'DEFAULT_VERSIONING_CLASS':
        'rest_framework.versioning.NamespaceVersioning',
}
```

We added a value for the `DEFAULT_VERSIONING_CLASS` settings key to configure the default versioning class that we want to use. As happened whenever we added values for settings keys, the new configuration will be applied to

all the views as a global setting that we are able to override if necessary in specific classes.

Create a new sub-folder named `v2` within the `restful01/drones` folder (`restful01\drones` in Windows). This new folder will be the baseline for the specific code required for version 2 of our RESTful Web Service.

Go to the recently created `restful01/drones/v2` folder and create a new file named `views.py`. Write the following code in this new file. The following lines show the code for this file that creates the new `ApiRootVersion2` class declared as a subclass of the `generics.GenericAPIView` class. The code file for the sample is included in the `hillar_django_restful_09_02` folder in the `restful01/drones/v2/views.py` file.

```
from rest_framework import generics
from rest_framework.response import Response
from rest_framework.reverse import reverse
from drones import views

class ApiRootVersion2(generics.GenericAPIView):
    name = 'api-root'
    def get(self, request, *args, **kwargs):
        return Response({
            'vehicle-categories': reverse(views.DroneCategoryList.name, request=request),
            'vehicles': reverse(views.DroneList.name, request=request),
            'pilots': reverse(views.PilotList.name, request=request),
            'competitions': reverse(views.CompetitionList.name, request=request)
        })
```

The `ApiRootVersion2` class is a subclass of the `rest_framework.generics.GenericAPIView` class and declares the `get` method. As we learned in [chapter 6, Working with Advanced Relationships and Serialization](#), the `GenericAPIView` class is the base class for all the generic views we have been working with. We will make the Django REST framework use this class instead of the `ApiRoot` class when the requests work with version 2.

The `ApiRootVersion2` class defines the `get` method that returns a `Response` object

with key/value pairs of strings that provide a descriptive name for the view and its URL, generated with the `rest_framework.reverse.reverse` function. This URL resolver function returns a fully qualified URL for the view.

Whenever we call the `reverse` function, we include the `request` value for the `request` argument. It is very important to do this in order to make sure that the `NamespaceVersioning` class can work as expected to configure the versioning scheme.

In this case, the response defines keys named '`vehicle-categories`' and '`vehicles`' instead of the '`drone-cagories`' and '`drones`' keys that are included in the `views.py` file, in the `ApiRoot` class that will be used for version 1.

Now, go to the recently created `restful01/drones/v2` folder and create a new file named `urls.py`. Write the following code in this new file. The following lines show the code for this file that declares the `urlpatterns` array. The lines that are different compared to the first version are highlighted. The code file for the sample is included in the `hillar_django_restful_09_02` folder in the `restful01/drones/v2/urls.py` file.

```
from django.conf.urls import url
from drones import views
from drones.v2 import views as views_v2

urlpatterns = [
    url(r'^vehicle-categories/$',
        views.DroneCategoryList.as_view(),
        name=views.DroneCategoryList.name),
    url(r'^vehicle-categories/(?P<pk>[0-9]+)$',
        views.DroneCategoryDetail.as_view(),
        name=views.DroneCategoryDetail.name),
    url(r'^vehicles/$',
        views.DroneList.as_view(),
        name=views.DroneList.name),
    url(r'^vehicles/(?P<pk>[0-9]+)$',
        views.DroneDetail.as_view(),
        name=views.DroneDetail.name),
    url(r'^pilots/$',
        views.PilotList.as_view(),
        name=views.PilotList.name),
    url(r'^pilots/(?P<pk>[0-9]+)$',
        views.PilotDetail.as_view(),
```

```
        name=views.PilotDetail.name),
    url(r'^competitions/$',
        views.CompetitionList.as_view(),
        name=views.CompetitionList.name),
    url(r'^competitions/(?P<pk>[0-9]+)$',
        views.CompetitionDetail.as_view(),
        name=views.CompetitionDetail.name),
    url(r'^$', views_v2.ApiRootVersion2.as_view(),
        name=views_v2.ApiRootVersion2.name),
]
```

The previous code defines the URL patterns that specify the regular expressions that have to be matched in the request to run a specific method for a class-based view defined in the original version of the `views.py` file.

We want version 2 to use `vehicle-categories` and `vehicles` instead of `drone-categories` and `drones`. However, we won't make changes in the serializer, and therefore, we will only change the URL that the clients must use to make requests related to drone categories and drones.

Now, we have to replace the code in the `urls.py` file in the `restful01/restful01` folder, specifically, the `restful01/restful01/urls.py` file. The file defines the root URL configurations, and therefore, we must include the URL patterns for the two versions declared in the `restful01/drones/urls.py` and in the `restful01/drones/v2/urls.py`. The following lines show the new code for the `restful01/restful01/urls.py` file. The code file for the sample is included in the `hillar_django_restful_09_02` folder, in the `restful01/restful01/urls.py` file.

```
from django.conf.urls import url, include

urlpatterns = [
    url(r'^v1/', include('drones.urls', namespace='v1')),
    url(r'^v1/api-auth/', include('rest_framework.urls', namespace='rest_framework')),
    url(r'^v2/', include('drones.v2.urls', namespace='v2')),
    url(r'^v2/api-auth/', include('rest_framework.urls', namespace='rest_framework'))
]
```

Whenever a URL starts with `v1/`, the url patterns defined for the previous version will be used and the `namespace` will be set to '`v1`'. Whenever a URL

starts with `v2/`, the url patterns defined for version 2 will be used and the namespace will be set to '`v2`'. We want the browsable API to display the log in and log out views for the two versions, and therefore, we included the necessary code to include the definitions included in `rest_framework.urls` for each of the versions, with different namespaces. This way, we will be able to easily test the two versions with the browsable API and the configured authentication.

Running tests to check that versioning works as expected

Now, we can launch Django's development server to compose and send HTTP requests to understand how the configured versioning scheme works. Execute any of the following two commands based on your needs to access the API in other devices or computers connected to your LAN. Remember that we analyzed the difference between them in [Chapter 3, Creating API Views](#), in the *Launching Django's development server* section.

```
python manage.py runserver  
python manage.py runserver 0.0.0.0:8000
```

After we run any of the previous commands, the development server will start listening at port 8000.

Now, we will compose and send an HTTP `GET` request to retrieve the first page of the drone categories by working with the first version of our RESTful Web Service:

```
http :8000/v1/drone-categories/
```

The following is the equivalent curl command:

```
curl -iX GET localhost:8000/v1/drone-categories/
```

The previous commands will compose and send the following HTTP request: `GET http://localhost:8000/v1/drone-categories/`. The request URL starts

with `v1/` after the domain and the port number (`http://localhost:8000/`), and therefore, it will match the '`^v1/`' regular expression and will test the regular expressions defined in the `restful01/drones/urls.py` file and will work with a namespace equal to '`v1`'. Then, the URL without the version prefix (`'v1/'`) will match the '`drone-categories/$'` regular expression and run the `get` method for the `views.DroneCategoryList` class-based view.

The `NamespaceVersioning` class makes sure that the rendered URLs include the appropriate version prefix in the response. The following lines show a sample response for the HTTP request, with the first and only page of drone categories. Notice that the URLs for the drones list for each category include the version prefix. In addition, the value of the `url` key for each drone category includes the version prefix.

```
HTTP/1.0 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Length: 670
Content-Type: application/json
Date: Sun, 03 Dec 2017 19:34:13 GMT
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
    "count": 2,
    "next": null,
    "previous": null,
    "results": [
        {
            "drones": [
                "http://localhost:8000/v1/drones/6",
                "http://localhost:8000/v1/drones/4",
                "http://localhost:8000/v1/drones/8",
                "http://localhost:8000/v1/drones/10"
            ],
            "name": "Octocopter",
            "pk": 2,
            "url": "http://localhost:8000/v1/drone-categories/2"
        },
        {
            "drones": [
                "http://localhost:8000/v1/drones/2",
                "http://localhost:8000/v1/drones/9",
                "http://localhost:8000/v1/drones/5",
                "http://localhost:8000/v1/drones/3"
            ],
            "name": "Hexacopter",
            "pk": 1,
            "url": "http://localhost:8000/v1/drone-categories/1"
        }
    ]
}
```

```
        "http://localhost:8000/v1/drones/7",
        "http://localhost:8000/v1/drones/3",
        "http://localhost:8000/v1/drones/12",
        "http://localhost:8000/v1/drones/11",
        "http://localhost:8000/v1/drones/1"
    ],
    "name": "Quadcopter",
    "pk": 1,
    "url": "http://localhost:8000/v1/drone-categories/1"
}
]
```

Now, we will compose and send an HTTP `GET` request to retrieve the first page of the vehicle categories by working with the second version of our RESTful Web Service:

```
http :8000/v2/vehicle-categories/
```

The following is the equivalent curl command:

```
curl -iX GET localhost:8000/v2/vehicle-categories/
```

The previous commands will compose and send the following HTTP request: `GET http://localhost:8000/v2/vehicle-categories/`. The request URL starts with `v2/` after the domain and the port number (`http://localhost:8000/`), and therefore, it will match the '`^v2/`' regular expression and will test the regular expressions defined in the `restful01/drones/v2/urls.py` file and will work with a namespace equal to '`v2`'. Then, the URL without the version prefix (`'v2/'`) will match the '`vehicle-categories/$'` regular expression and run the `get` method for the `views.DroneCategoryList` class-based view.

As happened with the previous request, the `NamespaceVersioning` class makes sure that the rendered URLs include the appropriate version prefix in the response. The following lines show a sample response for the HTTP request, with the first and only page of vehicle categories. We haven't made changes to the serializer in the new version, and therefore, each

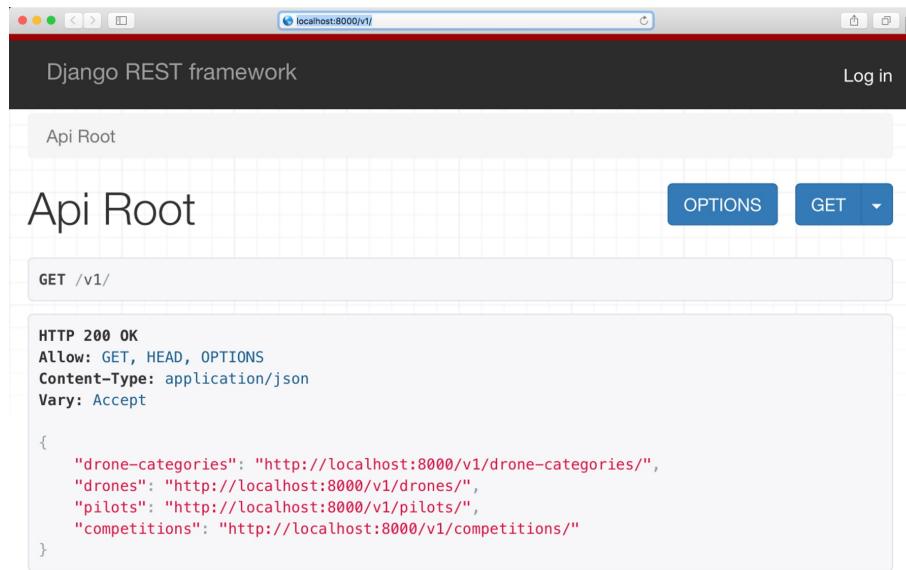
category will render a list named `drones`. However, the URLs for the drones list for each category include the version prefix and they use the appropriate URL with a `vehicle` in the URL instead of a drone. In addition, the value of the `url` key for each vehicle category includes the version prefix.

```
HTTP/1.0 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Length: 698
Content-Type: application/json
Date: Sun, 03 Dec 2017 19:34:29 GMT
Server: WSGIServer/0.2 CPython/3.6.2
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

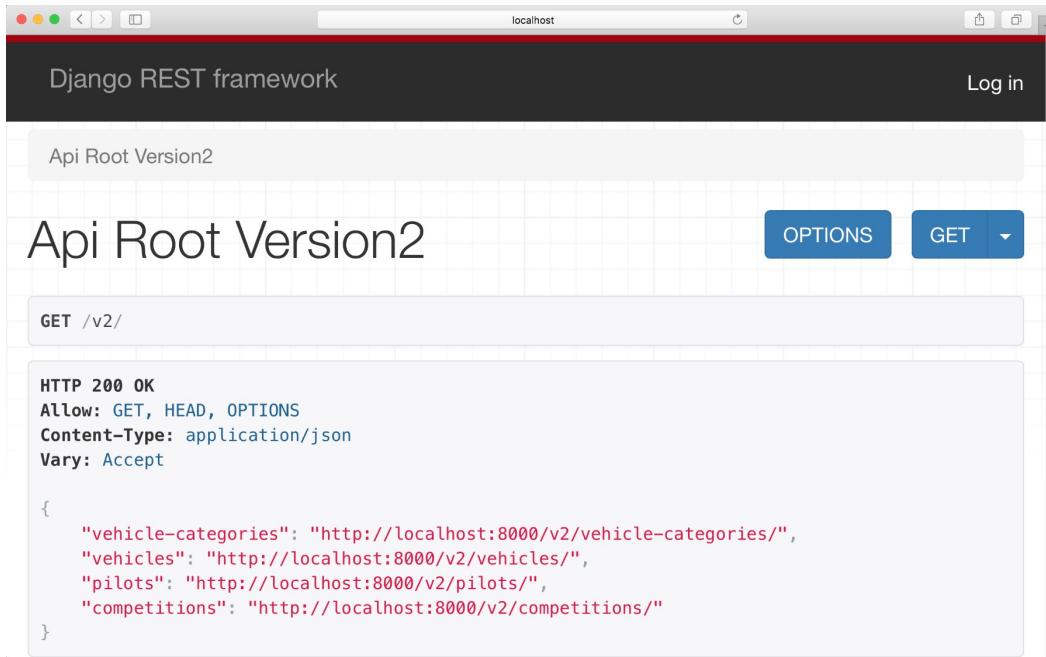
{
    "count": 2,
    "next": null,
    "previous": null,
    "results": [
        {
            "drones": [
                "http://localhost:8000/v2/vehicles/6",
                "http://localhost:8000/v2/vehicles/4",
                "http://localhost:8000/v2/vehicles/8",
                "http://localhost:8000/v2/vehicles/10"
            ],
            "name": "Octocopter",
            "pk": 2,
            "url": "http://localhost:8000/v2/vehicle-categories/2"
        },
        {
            "drones": [
                "http://localhost:8000/v2/vehicles/2",
                "http://localhost:8000/v2/vehicles/9",
                "http://localhost:8000/v2/vehicles/5",
                "http://localhost:8000/v2/vehicles/7",
                "http://localhost:8000/v2/vehicles/3",
                "http://localhost:8000/v2/vehicles/12",
                "http://localhost:8000/v2/vehicles/11",
                "http://localhost:8000/v2/vehicles/1"
            ],
            "name": "Quadcopter",
            "pk": 1,
            "url": "http://localhost:8000/v2/vehicle-categories/1"
        }
    ]
}
```



Open a web browser and enter `http://localhost:8000/v1`. The browser will compose and send a `GET` request to `/v1` with `text/html` as the desired content type and the returned HTML web page will be rendered. The request will end up executing the `get` method defined in the `ApiRoot` class within the `restful01/drones/views.py` file. The following screenshot shows the rendered web page with the resource description: Api Root. The Api Root for the first version uses the appropriate URLs for version 1, and therefore, all the URLs start with `http://localhost:8000/v1/`.



Now, go to `http://localhost:8000/v2`. The browser will compose and send a `GET` request to `/v2` with `text/html` as the desired content type and the returned HTML web page will be rendered. The request will end up executing the `get` method defined in the `ApiRootVersion2` class within the `restful01/drones/v2/views.py` file. The following screenshot shows the rendered web page with the resource description: Api Root Version2. The Api Root for the first version uses the appropriate URLs for version 2, and therefore, all the URLs start with `http://localhost:8000/v2/`. You can check the differences with the Api Root rendered for version 1.



This new version of the Api Root renders the following hyperlinks:

- <http://localhost:8000/v2/vehicle-categories/>: The collection of vehicle categories
- <http://localhost:8000/v2/vehicles/>: The collection of vehicles
- <http://localhost:8000/v2/pilots/>: The collection of pilots
- <http://localhost:8000/v2/competitions/>: The collection of competitions

We can use all the features provided by the browsable API with the two versions we have configured.

Developing and maintaining multiple versions of a RESTful Web Service is an extremely complex task that requires a lot of planning. We must take into account the different versioning schemes that the Django REST framework provides out of the box to make our job simpler. However, it is always very important to avoid making things more complex than necessary. We should keep any versioning

scheme as simple as possible and we must make sure that we continue to provide RESTful Web Services with easily identifiable resources and resource collections in the URLs.

Test your knowledge

Let's see whether you can answer the following questions correctly:

1. The `rest_framework.throttling.UserRateThrottle` class:
 1. Limits the rate of requests that a specific user can make and applies to *both authenticated and non-authenticated users*
 2. Limits the rate of requests that a specific user can make and applies *only to authenticated users*
 3. Limits the rate of requests that a specific user can make and applies *only to non-authenticated users*
2. Which of the following settings key in the `REST_FRAMEWORK` dictionary specifies the global setting with a tuple of string whose values indicate the classes that we want to use for throttling rules:
 1. `'DEFAULT_THROTTLE_CLASSES'`
 2. `'GLOBAL_THROTTLE_CLASSES'`
 3. `'REST_FRAMEWORK_THROTTLE_CLASSES'`
3. Which of the following settings key in the `REST_FRAMEWORK` dictionary specifies a dictionary with the default throttle rates:
 1. `'GLOBAL_THROTTLE_RATES'`

2. '`DEFAULT_THROTTLE_RATES`'
 3. '`REST_FRAMEWORK_THROTTLE_RATES`'
4. The `rest_framework.throttling.ScopedRateThrottle` class:
1. Limits the rate of requests that an anonymous user can make
 2. Limits the rate of requests that a specific user can make
 3. Limits the rate of requests for specific parts of the RESTful Web Service identified with the value assigned to the `throttle_scope` property
5. The `rest_framework.versioning.NamespaceVersioning` class configures a versioning scheme known as:
1. Query parameter versioning or request parameter versioning
 2. Media type versioning, content negotiation versioning or accept header versioning
 3. URI versioning or URL path versioning

The rights answers are included in the [Appendix](#), *Solutions*.

Summary

In this chapter, we understood the importance of throttling rules and how we can combine them with authentication and permissions in Django, the Django REST framework and RESTful Web Services. We analyzed the throttling classes included in the Django REST framework out of the box.

We followed the necessary steps to configure many throttling policies in the Django REST framework. We worked with global and scope-related settings. Then, we used command-line tools to compose and send many requests to test how the throttling rules worked.

We understood versioning classes and we configured a URL path versioning scheme to allow us to work with two versions of our RESTful Web Service. We used command-line tools and the browsable API to understand the differences between the two versions.

Now that we can combine throttling rules, authentication and permission policies with versioning schemes, it is time to explore other features offered by the Django REST framework and third-party packages to improve our RESTful Web Service and automate tests. We will cover these topics in the next chapter.

Automating Tests

In this chapter, we will add some automated testing to our RESTful Web Services and we will execute the tests within a test database. We won't cover all the tests that we should write for our complex RESTful Web Service. However, we will follow the first steps and we will gain an understanding of:

- Getting ready for unit testing with pytest
- Writing unit tests for a RESTful Web Service
- Discovering and running unit tests with pytest
- Writing new unit tests to improve tests' code coverage
- Running new unit tests

Getting ready for unit testing with pytest

So far, we have been writing code to add features to our RESTful Web Service and configuring many settings for the Django REST framework. We used command-line and GUI tools to understand how all the pieces worked together and to check the results of diverse HTTP requests. Now, we will write unit tests that will allow us to make sure that our RESTful Web Service will work as expected. Before we can start writing unit tests, it is necessary to install many additional packages in our virtual environment. Make sure you quit Django's development server. Remember that you just need to press *Ctrl + C* in the terminal or go to the Command Prompt window in which it is running. First, we will make some changes to work with a single version of our RESTful Web Service.

This way, it will be easier to focus on tests for a single version in our examples. Replace the code in the `urls.py` file in the `restful01/restful01` folder, specifically, the `restful01/restful01/urls.py` file. The file defines the root URL configurations, and therefore, we want to include only the URL patterns for the first version of our web service. The code file for the sample is included in the `hillar_django_restful_10_01` folder, in the `restful01/restful01/urls.py` file:

```
from django.conf.urls import url, include

urlpatterns = [
    url(r'^$', include('drones.urls')),
    url(r'^api-auth/', include('rest_framework.urls'))
]
```

We will install the following Python packages in our virtual environment:

- `pytest`: This is a very popular Python unit test framework that makes testing easy and reduces boilerplate code
- `pytest-django`: This `pytest` plugin allows us to easily use and configure the features provided by `pytest` in our Django tests

Notice that we won't be working with Django's `manage.pytest` command. We will work with `pytest` instead. However, in case you don't want to work with `pytest`, most of the things you will learn can be easily adapted to any other test framework. In fact, the code is compatible with `nose` in case you decide to use the most common, yet a bit outdated, configuration for testing with the Django REST framework. Nowadays, `pytest` is the preferred unit test framework for Python.

Run the following command to install the `pytest` package:

```
pip install pytest
```

The last lines for the output will indicate that the `pytest` package and its required packages have been successfully installed:

```
Installing collected packages: attrs, pluggy, six, py, pytest  Running setup.py
```

We just need to run the following command to install the `pytest-django` package:

```
pip install pytest-django
```

The last lines for the output will indicate that the `pytest-django` package has been successfully installed:

```
Installing collected packages: pytest-django
Successfully installed pytest-django-3.1.2
```

Now, go to the `restful01` folder that contains the `manage.py` file and create a new file named `pytest.ini`. Write the following code in this new file. The following lines show the code for this file that specifies the Django settings module (`restful01.settings`) and the pattern that `pytest` will use to locate the Python files, the declare tests. The code file for the sample is included in the `hillar_django_restful_10_01` folder in the `restful01/pytest.ini` file:

```
[pytest]
DJANGO_SETTINGS_MODULE = restful01.settings
python_files = tests.py test_*.py *_tests.py
```

Whenever we execute `pytest` to run tests, the test runner will check the following to find test definitions:

- Python files named `tests.py`
- Python files whose name starts with the `test_` prefix
- Python files whose name ends with the `_tests` suffix

In [Chapter 9, Applying Throttling Rules and Versioning Management](#), we configured throttling rules for our RESTful Web Service. We want to run our tests considering the throttling rules. In fact, we should write tests to make sure that the throttling rules are working OK. We will be running requests many times, and therefore, the low values we used for the throttling rules might complicate running all the requests required by our tests. Hence, we will increase the values for the throttling rules to simplify our testing samples. Open the `restful01/restful01/settings.py` file that declares module-level variables that define the configuration of Django for the `restful01` project. We will make some changes to this Django settings file. Replace the code for the highlighted lines included in the `REST_FRAMEWORK`

dictionary. The following lines show the new declaration of the `REST_FRAMEWORK` dictionary. The code file for the sample is included in the `hillar_django_restful_10_01` folder in the `restful01/restful01/settings.py` file:

```
REST_FRAMEWORK = {  
    'DEFAULT_PAGINATION_CLASS':  
        'drones.custompagination.LimitOffsetPaginationWithUpperBound',  
    'PAGE_SIZE': 4,  
    'DEFAULT_FILTER_BACKENDS': (  
        'django_filters.rest_framework.DjangoFilterBackend',  
        'rest_framework.filters.OrderingFilter',  
        'rest_framework.filters.SearchFilter',  
    ),  
    'DEFAULT_AUTHENTICATION_CLASSES': (  
        'rest_framework.authentication.BasicAuthentication',  
        'rest_framework.authentication.SessionAuthentication',  
    ),  
    'DEFAULT_THROTTLE_CLASSES': (  
        'rest_framework.throttling.AnonRateThrottle',  
        'rest_framework.throttling.UserRateThrottle',  
    ),  
    'DEFAULT_THROTTLE_RATES': {  
        'anon': '300/hour',  
        'user': '100/hour',  
        'drones': '200/hour',  
        'pilots': '150/hour',  
    }  
}
```

We increased the number of requests per hour that we can execute in each of the throttling rates configurations. This way, we will be able to run our tests without issues.

In this case, we are using the same settings file for our tests in order to avoid running additional steps and repeating test settings. However, in most cases, we would create a specific Django configuration file for testing.

Writing unit tests for a RESTful Web Service

Now, we will write our first round of unit tests related to the drone category class based views: `DroneCategoryList` and `DroneCategoryDetail`. Open the existing `restful01/drones/tests.py` file and replace the existing code with the following lines that declare many `import` statements and the `DroneCategoryTests` class. The code file for the sample is included in the `hillar_django_restful_10_01` folder in the `restful01/drones/tests.py` file:

```
from django.utils.http import urlencode
from django.core.urlresolvers import reverse
from rest_framework import status
from rest_framework.test import APITestCase
from drones.models import DroneCategory
from drones import views

class DroneCategoryTests(APITestCase):
    def post_drone_category(self, name):
        url = reverse(views.DroneCategoryList.name)
        data = {'name': name}
        response = self.client.post(url, data, format='json')
        return response

    def test_post_and_get_drone_category(self):
        """
        Ensure we can create a new DroneCategory and then retrieve it
        """
        new_drone_category_name = 'Hexacopter'
        response = self.post_drone_category(new_drone_category_name)
        print("PK {}".format(DroneCategory.objects.get().pk))
        assert response.status_code == status.HTTP_201_CREATED
        assert DroneCategory.objects.count() == 1
        assert DroneCategory.objects.get().name == new_drone_category_name
```

The `DroneCategoryTests` class is a subclass of the `rest_framework.test.APITestCase`

superclass and declares the `post_drone_category` method that receives the desired `name` for the new drone category as an argument.

This method builds the URL and the data dictionary to compose and send an HTTP `POST` request to the view associated with the `views.DroneCategoryList.name name` (`dronecategory-list`) and returns the response generated by this request.

The code uses the `self.client` attribute to access the `APIClient` instance that allows us to easily compose and send HTTP requests for testing our RESTful Web Service that uses the Django REST framework. For this test, the code calls the `post` method with the built `url`, the `data` dictionary, and the desired format for the data: '`json`'.

Many test methods will call the `post_drone_category` method to create a new drone category and then compose and send other HTTP requests to the RESTful Web Service. For example, we will need a drone category to post a new drone.

The `test_post_and_get_drone_category` method tests whether we can create a new `DroneCategory` and then retrieve it. The method calls the `post_drone_category` method and then calls `assert` many times to check for the following expected results:

1. The `status_code` attribute for the response is equal to HTTP 201 Created (`status.HTTP_201_CREATED`)
2. The total number of `DroneCategory` objects retrieved from the database is 1
3. The value of the `name` attribute for the retrieved `DroneCategory` object is equal to the `new_drone_category_name` variable passed as a parameter to the `post_drone_category` method

The previously coded tests make sure that we can create a new drone category with the RESTful Web Service, it is persisted in the database, and

the serializer does its job as expected. The drone category is a very simple entity because it just has a primary key and a name. Now, we will add more test methods that will allow us to cover more scenarios related to drone categories.

Add the `test_post_existing_drone_category_name` method to the recently created `DroneCategoryTests` class in the `restful01/drones/tests.py` file. The code file for the sample is included in the `hillar_django_restful_10_01` folder in the `restful01/drones/tests.py` file:

```
def test_post_existing_drone_category_name(self):
    """
    Ensure we cannot create a DroneCategory with an existing name
    """
    url = reverse(views.DroneCategoryList.name)
    new_drone_category_name = 'Duplicated Copter'
    data = {'name': new_drone_category_name}
    response1 = self.post_drone_category(new_drone_category_name)
    assert response1.status_code == status.HTTP_201_CREATED
    response2 = self.post_drone_category(new_drone_category_name)
    print(response2)
    assert response2.status_code == status.HTTP_400_BAD_REQUEST
```

The new method tests whether the unique constraint for the drone category name works as expected and doesn't make it possible for us to create two drone categories with the same name. The second time we compose and send an `HTTP POST` request with a duplicate drone name, we must receive an `HTTP 400 Bad Request` status code (`status.HTTP_400_BAD_REQUEST`).

Add the `test_filter_drone_category_by_name` method to the `DroneCategoryTests` class in the `restful01/drones/tests.py` file. The code file for the sample is included in the `hillar_django_restful_10_01` folder in the `restful01/drones/tests.py` file:

```
def test_filter_drone_category_by_name(self):
    """
    Ensure we can filter a drone category by name
    """
    drone_category_name1 = 'Hexacopter'
```

```
self.post_drone_category(drone_category_name1)
drone_caregory_name2 = 'Octocopter'
self.post_drone_category(drone_caregory_name2)
filter_by_name = { 'name' : drone_category_name1 }
url = '{0}?{1}'.format(
    reverse(views.DroneCategoryList.name),
    urlencode(filter_by_name))
print(url)
response = self.client.get(url, format='json')
print(response)
assert response.status_code == status.HTTP_200_OK
# Make sure we receive only one element in the response
assert response.data['count'] == 1
assert response.data['results'][0]['name'] ==
drone_category_name1
```

The new method tests whether we can filter a drone category by name, and therefore, checks the usage of the filter field we have configured for the `DroneCategoryList` class-based view. The code creates two drone categories and then calls the `django.utils.http.urlencode` function to build an encoded URL from the `filter_by_name` dictionary. This dictionary includes the field name as a key and the desired string for the field as a value. In this case, `drone_category_name1` is equal to `'Hexacopter'`, and therefore, the encoded URL saved in the `url` variable will be `'name=Hexacopter'`.

After the call to `self.client.get` with the built URL to retrieve the filtered list of drone categories, the method verifies the data included in the response JSON body by inspecting the `data` attribute for the response. The second line that calls `assert` checks whether the value for `count` is equal to `1` and the next lines verify whether the `name` key for the first element in the `results` array is equal to the value hold in the `drone_category_name1` variable. The code is easy to read and understand.

Add the `test_get_drone_categories_collection` method to the `DroneCategoryTests` class in the `restful01/drones/tests.py` file. The code file for the sample is included in the `hillar_django_restful_10_01` folder in the `restful01/drones/tests.py` file:

```
def test_get_drone_categories_collection(self):
    """
```

```
Ensure we can retrieve the drone categories collection
"""
new_drone_category_name = 'Super Copter'
self.post_drone_category(new_drone_category_name)
url = reverse(views.DroneCategoryList.name)
response = self.client.get(url, format='json')
assert response.status_code == status.HTTP_200_OK
# Make sure we receive only one element in the response
assert response.data['count'] == 1
assert response.data['results'][0]['name'] ==
new_drone_category_name
```

The method tests whether we can retrieve the drone categories collection. First, the code creates a new drone category and then makes an `HTTP GET` request to retrieve the drones collection. The lines that call `assert` check that the results include the only created and persisted drone and that its name is equal to the name used for the call to the `POST` method to create the new drone category.

Add the `test_update_drone_category` method to the `DroneCategoryTests` class in the `restful01/drones/tests.py` file. The code file for the sample is included in the `hillar_django_restful_10_01` folder in the `restful01/drones/tests.py` file:

```
def test_update_drone_category(self):
    """
    Ensure we can update a single field for a drone category
    """
    drone_category_name = 'Category Initial Name'
    response = self.post_drone_category(drone_category_name)
    url = reverse(
        views.DroneCategoryDetail.name,
        None,
        {response.data['pk']})
    updated_drone_category_name = 'Updated Name'
    data = {'name': updated_drone_category_name}
    patch_response = self.client.patch(url, data, format='json')
    assert patch_response.status_code == status.HTTP_200_OK
    assert patch_response.data['name'] ==
updated_drone_category_name
```

The new method tests whether we can update a single field for a drone

category. First, the code creates a new drone category and then makes an HTTP `PATCH` request to update the name field for the previously persisted drone category. The lines that call `assert` check that the returned status code is `HTTP_200_OK` and that the value of the `name` key in the response body is equal to the new name that we specified in the HTTP `PATCH` request.

Add the `test_get_drone_category` method to the `DroneCategoryTests` class in the `restful01/drones/tests.py` file. The code file for the sample is included in the `hillar_django_restful_10_01` folder in the `restful01/drones/tests.py` file:

```
def test_get_drone_category(self):
    """
    Ensure we can get a single drone category by id
    """
    drone_category_name = 'Easy to retrieve'
    response = self.post_drone_category(drone_category_name)
    url = reverse(
        views.DroneCategoryDetail.name,
        None,
        {response.data['pk']})
    get_response = self.client.get(url, format='json')
    assert get_response.status_code == status.HTTP_200_OK
    assert get_response.data['name'] == drone_category_name
```

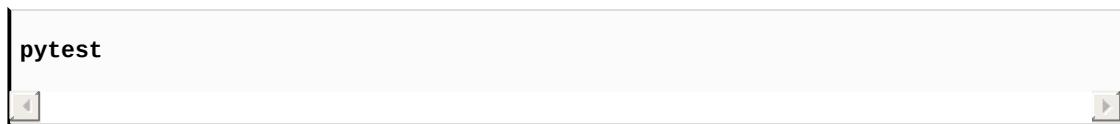
The new method tests whether we can retrieve a single category with an HTTP `GET` request. First, the code creates a new drone category and then makes an HTTP `GET` request to retrieve the previously persisted drone category. The lines that call `assert` check that the returned status code is `HTTP_200_OK` and that the value of the `name` key in the response body is equal to the name that we specified in the HTTP `POST` request that created the drone category.

Each test method that requires a specific condition in the database must execute all the necessary code to generate the required data. For example, in order to update the name for an existing drone category, it was necessary to create a new drone category before making the HTTP `PATCH` request to update it. Pytest and the Django REST framework will execute each test method without data from the previously executed test methods in the database, that is, each test will run with a database cleansed of data

from the previous tests.

Discovering and running unit tests with pytest

Now, go to the `restful01` folder that contains the `manage.py` file, with the virtual environment activated, and run the following command:



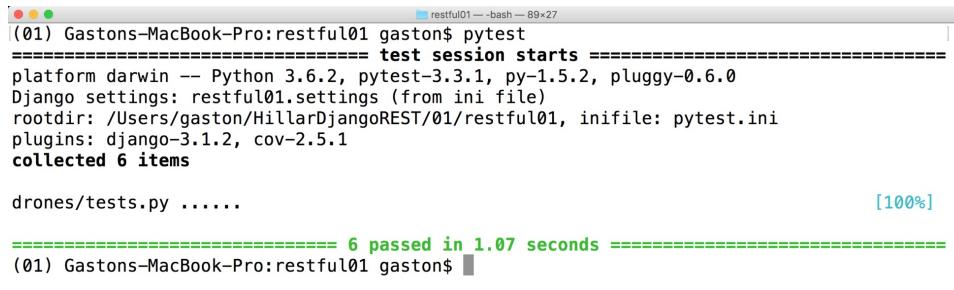
The `pytest` command and the Django REST framework will perform the following actions:

1. Create a clean test database name `test_drones`.
2. Run all the migrations required for the database.
3. Discover the tests that have to be executed based on the settings specified in the `pytest.ini` file.
4. Run all the methods whose name starts with the `test_` prefix in the `DroneCategoryTests` class and display the results. We declared this class in the `tests.py` file and it matches the pattern specified for the `python_files` setting in the `pytest.ini` file.
5. Drop the test database named `test_drones`.

It is very important to know that the tests won't make changes to the database we have been using when working with our RESTful Web Service. Notice that the test database name is `test_drones` and the database name that we have been using with Django's development server is `drones`.

The following screenshot shows a sample output generated by the `pytest`

command:



```
(01) Gastons-MacBook-Pro:restful01 gaston$ pytest
=====
platform darwin -- Python 3.6.2, pytest-3.3.1, py-1.5.2, pluggy-0.6.0
Django settings: restful01.settings (from ini file)
rootdir: /Users/gaston/HillarDjangoREST/01/restful01, infile: pytest.ini
plugins: django-3.1.2, cov-2.5.1
collected 6 items

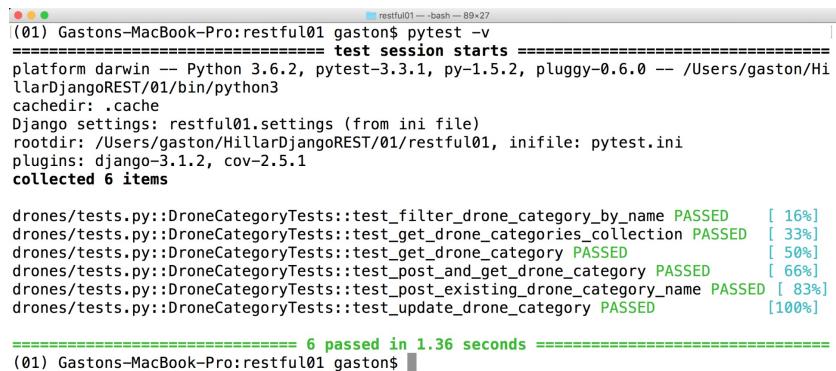
drones/tests.py ......

=====
6 passed in 1.07 seconds
(01) Gastons-MacBook-Pro:restful01 gaston$
```

The output indicated that the test runner collected and executed six tests and all of them passed. However, the output didn't show the names of the tests that passed. Hence, we will run `pytest` again with the `-v` option to increase verbosity. Run the following command:

```
pytest -v
```

The following screenshot shows a sample output generated by the `pytest` command with the increased verbosity:



```
(01) Gastons-MacBook-Pro:restful01 gaston$ pytest -v
=====
platform darwin -- Python 3.6.2, pytest-3.3.1, py-1.5.2, pluggy-0.6.0 -- /Users/gaston/HillarDjangoREST/01/bin/python3
cachedir: .cache
Django settings: restful01.settings (from ini file)
rootdir: /Users/gaston/HillarDjangoREST/01/restful01, infile: pytest.ini
plugins: django-3.1.2, cov-2.5.1
collected 6 items

drones/tests.py::DroneCategoryTests::test_filter_drone_category_by_name PASSED [ 16%]
drones/tests.py::DroneCategoryTests::test_get_drone_categories_collection PASSED [ 33%]
drones/tests.py::DroneCategoryTests::test_get_drone_category PASSED [ 50%]
drones/tests.py::DroneCategoryTests::test_post_and_get_drone_category PASSED [ 66%]
drones/tests.py::DroneCategoryTests::test_post_existing_drone_category_name PASSED [ 83%]
drones/tests.py::DroneCategoryTests::test_update_drone_category PASSED [100%]

=====
6 passed in 1.36 seconds
(01) Gastons-MacBook-Pro:restful01 gaston$
```

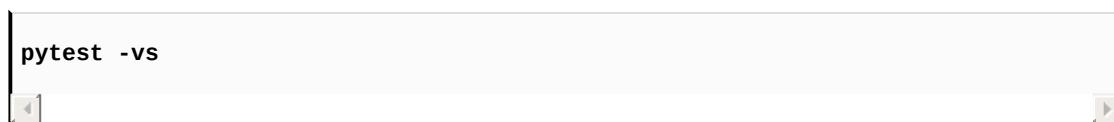
We enabled verbose mode, and therefore, the new output displayed the full test names. Pytest displays the following information for each discovered and executed test: the Python file that defines it, the class name, and the method, such as the following line:

```
drones/tests.py::DroneCategoryTests::test_filter_drone_category_by_name PASSED [
```

The line indicates that the `test_filter_drone_category_by_name` test method declared in the `DroneCategoryTests` class, within the `drones/tests.py` module has been executed, passed, and its execution represents 16% of the discovered tests.

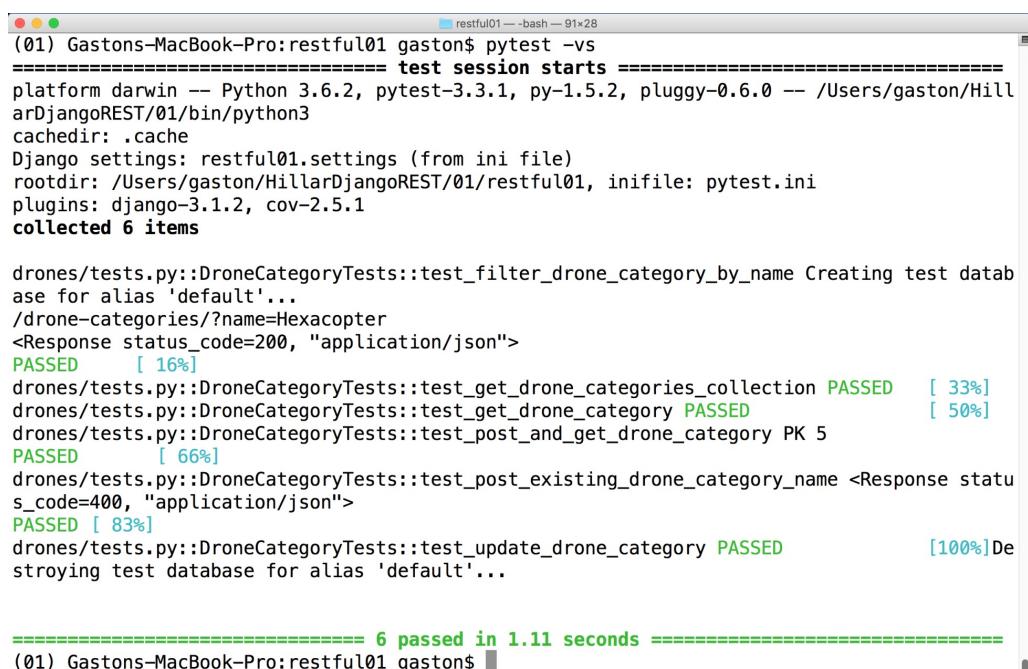
The verbose mode makes it possible to know the specific tests that have been executed.

Some of the test methods include calls to the `print` function. By default, `pytest` captures both the `stdout` and `stderr` and only shows the captured content for the tests that fail. Sometimes, it is useful for us to see the results of calls to the `print` function while `pytest` runs the tests. We will run `pytest` again with `-s` option combined with the `-v` option to disable capturing and increase verbosity. Notice that the `-s` option is a shortcut that is equivalent to the `-capture=no` option. Run the following command:



```
pytest -vs
```

The following screenshot shows a sample output for the previous command:



```
(01) Gastons-MacBook-Pro:restful01 gaston$ pytest -vs
=====
platform darwin -- Python 3.6.2, pytest-3.3.1, py-1.5.2, pluggy-0.6.0 -- /Users/gaston/HillarDjangoREST/01/bin/python3
cachedir: .cache
Django settings: restful01.settings (from ini file)
rootdir: /Users/gaston/HillarDjangoREST/01/restful01, inifile: pytest.ini
plugins: django-3.1.2, cov-2.5.1
collected 6 items

drones/tests.py::DroneCategoryTests::test_filter_drone_category_by_name Creating test database for alias 'default'...
/.../drone-categories/?name=Hexacopter
<Response status_code=200, "application/json">
PASSED [ 16%]
drones/tests.py::DroneCategoryTests::test_get_drone_categories_collection PASSED [ 33%]
drones/tests.py::DroneCategoryTests::test_get_drone_category PASSED [ 50%]
drones/tests.py::DroneCategoryTests::test_post_and_get_drone_category PK 5
PASSED [ 66%]
drones/tests.py::DroneCategoryTests::test_post_existing_drone_category_name <Response status_code=400, "application/json">
PASSED [ 83%]
drones/tests.py::DroneCategoryTests::test_update_drone_category PASSED [100%] Destroying test database for alias 'default'...

=====
6 passed in 1.11 seconds
(01) Gastons-MacBook-Pro:restful01 gaston$
```

The new output displayed the results of each call to the `print` function. In addition, we will notice that there are two messages displayed that are printed by Django, one line before the first test runs and another line after the last test finishes its execution:

```
Creating test database for alias 'default'...Destroying test database for alias
```

These messages indicate that Django created the test database before running the first test and drops the database after all the tests have been executed.

The `test_filter_drone_category_by_name` test method declared in the `DroneCategoryTests` class has the following two calls to the `print` function:

```
url = '{0}?{1}'.format(  
    reverse(views.DroneCategoryList.name),  
    urlencode(filter_by_name))  
print(url)  
response = self.client.get(url, format='json')  
print(response)
```

The previous output shows the results of the two calls to the `print` function. First, the tests output display the value of the `url` variable with the composed URL and then the output shows the response of the call to `self.client.get` as a string:

```
drones/tests.py::DroneCategoryTests::test_filter_drone_category_by_name Creating
```

In this case, the output is clear. However, as you might notice in the previous screenshot, the output generated by the other `print` statements is shown at the right-hand side of the test method name that was executed and it is not so clear. Hence, whenever we want to provide helpful output for tests, it is always a good idea to make sure we start with a new line (`'\n'`) and provide some context for the output we are displaying.

Now, we will replace the line that calls the `print` function in the `test_post_and_get_drone_category` method for the `DroneCategoryTests` class in the `restful01/drones/tests.py` file. The code file for the sample is included in the `hillar_django_restful_10_02` folder in the `restful01/drones/tests.py` file. The replaced line is highlighted:

```
def test_post_and_get_drone_category(self):
    """
    Ensure we can create a new DroneCategory and then retrieve it
    """
    new_drone_category_name = 'Hexacopter'
    response = self.post_drone_category(new_drone_category_name)
    print("nPK {0}n".format(DroneCategory.objects.get().pk)
    assert response.status_code == status.HTTP_201_CREATED
    assert DroneCategory.objects.count() == 1
    assert DroneCategory.objects.get().name ==
    new_drone_category_name
```

Run the following command to execute pytest again with the `-s` and `-v` options combined:

```
pytest -vs
```

The following screenshot shows a sample output for the previous command:

```
(01) Gastons-MacBook-Pro:restful01 gaston$ pytest -v
=====
platform darwin -- Python 3.6.2, pytest-3.3.1, py-1.5.2, pluggy-0.6.0 -- /Users/gaston/HillarDjangoREST/01/bin/python3
cachedir: .cache
Django settings: restful01.settings (from ini file)
rootdir: /Users/gaston/HillarDjangoREST/01/restful01, infile: pytest.ini
plugins: django-3.1.2, cov-2.5.1
collected 6 items

drones/tests.py::DroneCategoryTests::test_filter_drone_category_by_name Creating test database for alias 'default'...
/drones-categories/?name=Hexacopter
<Response status_code=200, "application/json">
PASSED [ 16%]
drones/tests.py::DroneCategoryTests::test_get_drone_categories_collection PASSED [ 33%]
drones/tests.py::DroneCategoryTests::test_get_drone_category PASSED [ 50%]
drones/tests.py::DroneCategoryTests::test_post_and_get_drone_category
PK 5
PASSED [ 66%]
drones/tests.py::DroneCategoryTests::test_post_existing_drone_category_name <Response status_code=400, "application/json">
PASSED [ 83%]
drones/tests.py::DroneCategoryTests::test_update_drone_category PASSED [100%] Destroying test database for alias 'default'...

=====
6 passed in 1.10 seconds
(01) Gastons-MacBook-Pro:restful01 gaston$
```

The edits made in the call to the print statement that added a new line before and after the output made it easier to read the output. The generated output is highlighted in the previous screenshot. It is very important to take this formatting into account when working with pytest.

Writing new unit tests to improve the tests' code coverage

Our first round of unit tests was related to the drone category class-based views: `DroneCategoryList` and `DroneCategoryDetail`. Now, we will write a second round of unit tests related to the pilot class-based views: `PilotList` and `PilotDetail`. The new tests will be a bit more complex because we will have to work with authenticated requests.

In [*chapter 8, Securing the API with Authentication and Permissions*](#), we configured authentication and permission policies for the class-based views that work with the `Pilot` model. We overrode the values for the `authentication_classes` and `permission_classes` class attributes for the `PilotDetail` and `PilotList` classes. In order to create, read, update, or delete pilots, we have to provide an authentication token. Hence, we will write tests to make sure that an unauthenticated request cannot perform operations related to pilots. In addition, we want to make sure that an authenticated request with a token can create a new pilot and then retrieve it.

Open the `restful01/drones/tests.py` file and add the following lines after the last line that declares the imports, before the declaration of the `DroneCategoryTests` class:

```
from drones.models import Pilot
from rest_framework.authtoken.models import Token
from django.contrib.auth.models import User
```

Add the following code to the existing `restful01/drones/tests.py` file to create the new `PilotTests` class. The code file for the sample is included in the `hillar_django_restful_10_02` folder in the `restful01/drones/tests.py` file:

```

class PilotTests(APITestCase):
    def post_pilot(self, name, gender, races_count):
        url = reverse(views.PilotList.name)
        print(url)
        data = {
            'name': name,
            'gender': gender,
            'races_count': races_count,
        }
        response = self.client.post(url, data, format='json')
        return response

    def create_user_and_set_token_credentials(self):
        user = User.objects.create_user(
            'user01', 'user01@example.com', 'user01P4ssw0rD')
        token = Token.objects.create(user=user)
        self.client.credentials(
            HTTP_AUTHORIZATION='Token {}'.format(token.key))

    def test_post_and_get_pilot(self):
        """
        Ensure we can create a new Pilot and then retrieve it
        Ensure we cannot retrieve the persisted pilot without a token
        """
        self.create_user_and_set_token_credentials()
        new_pilot_name = 'Gaston'
        new_pilot_gender = Pilot.MALE
        new_pilot_races_count = 5
        response = self.post_pilot(
            new_pilot_name,
            new_pilot_gender,
            new_pilot_races_count)
        print("nPK {}".format(Pilot.objects.get().pk))
        assert response.status_code == status.HTTP_201_CREATED
        assert Pilot.objects.count() == 1
        saved_pilot = Pilot.objects.get()
        assert saved_pilot.name == new_pilot_name
        assert saved_pilot.gender == new_pilot_gender
        assert saved_pilot.races_count == new_pilot_races_count
        url = reverse(
            views.PilotDetail.name,
            None,
            {saved_pilot.pk})
        authorized_get_response = self.client.get(url, format='json')
        assert authorized_get_response.status_code ==
            status.HTTP_200_OK
        assert authorized_get_response.data['name'] == new_pilot_name
        # Clean up credentials
        self.client.credentials()

```

```
unauthorized_get_response = self.client.get(url, format='json')
assert unauthorized_get_response.status_code ==
status.HTTP_401_UNAUTHORIZED
```

The `PilotTests` class is a subclass of the `rest_framework.test.APITestCase` superclass and declares the `post_pilot` method that receives the desired `name` and `gender` for the new pilot as arguments.

This method builds the URL and the data dictionary to compose and send an HTTP `POST` request to the view associated with the `views.PilotList.name` name (`pilot-list`) and returns the response generated by this request.

Many test methods will call the `post_pilot` method to create a new pilot and then compose and send other HTTP requests to the RESTful Web Service. Notice that the `post_pilot` method doesn't configure authentication credentials, and therefore, we will be able to call this method for unauthenticated or authenticated users. We already know that unauthenticated users shouldn't be able to post a pilot, and a test will call this method without a token and make sure no pilot is persisted in the database.

The `create_user_and_set_token_credentials` method executes the following actions:

- Creates a Django user with a call to the `User.objects.create_user` method.
- Creates a token for the previously created Django user with a call to the `Token.objects.create` method.
- Includes the token generated for the Django user as the value for the `Authorization` HTTP header key with the '`Token`' string as a prefix for the token. The last line calls the `self.client.credentials` method to set the generated HTTP header as the value for the `HTTP_AUTHORIZATION` named argument.

Remember that the `self.client` attribute allows us to access the `APIClient` instance.

Whenever a test wants to perform an HTTP request with a token, the code will call the `create_user_and_set_token_credentials` method. In order to clean up the credentials configured for the `APIClient` instance saved in `self.client`, it is necessary to call the `self.client.credentials()` method without arguments.

The `test_post_and_get_pilot` method tests the following path:

1. We can create a new `Pilot` with an HTTP `POST` request that has an appropriate authentication token
2. We can retrieve the recently created `Pilot` with an HTTP `GET` request that has an appropriate authentication token
3. We cannot retrieve the recently created `Pilot` with an unauthenticated HTTP `GET` request

The code calls the `create_user_and_set_token_credentials` method and then calls the `post_pilot` method. Then, the code calls `assert` many times to check for the following expected results:

1. The `status_code` attribute for the response is equal to HTTP 201 Created (`status.HTTP_201_CREATED`)
2. The total number of `Pilot` objects retrieved from the database is 1
3. The value of the `name`, `gender`, and `races_count` attributes for the retrieved `Pilot` object is equal to the values passed as parameters to the `post_pilot` method

Then, the code calls the `self.client.get` with the built URL to retrieve the previously persisted pilot. This request will use the same credentials

applied to the HTTP POST request, and therefore, the new request is authenticated by a valid token. The method verifies the data included in the response JSON body by inspecting the `data` attribute for the response. The code calls `assert` twice to check for the following expected results:

1. The `status_code` attribute for the response is equal to HTTP 201 Created (`status.HTTP_201_CREATED`)
2. The value of the `name` key in the response body is equal to the `name` that we specified in the HTTP POST request

Then, the code calls the `self.client.credentials` method without arguments to clean up the credentials and calls the `self.client.get` method again with the same built URL, this time, without a token. Finally, the code calls `assert` to check that the `status_code` attribute for the response is equal to HTTP 401 Unauthorized (`status.HTTP_401_UNAUTHORIZED`).

The previously coded test makes sure that we can create a new pilot with the RESTful Web Service and the appropriate authentication requirement we configured, the pilot is persisted in the database, and the serializer does its job as expected. In addition, unauthenticated users aren't able to access a pilot.

Add the `test_try_to_post_pilot_without_token` method to the recently created `DroneCategoryTests` class in the `restful01/drones/tests.py` file. The code file for the sample is included in the `hillar_django_restful_10_02` folder in the `restful01/drones/tests.py` file:

```
def test_try_to_post_pilot_without_token(self):
    """
    Ensure we cannot create a pilot without a token
    """
    new_pilot_name = 'Unauthorized Pilot'
    new_pilot_gender = Pilot.MALE
    new_pilot_races_count = 5
    response = self.post_pilot(
        new_pilot_name,
        new_pilot_gender,
```

```
    new_pilot_races_count)
print(response)
print(Pilot.objects.count())
assert response.status_code == status.HTTP_401_UNAUTHORIZED
assert Pilot.objects.count() == 0
```

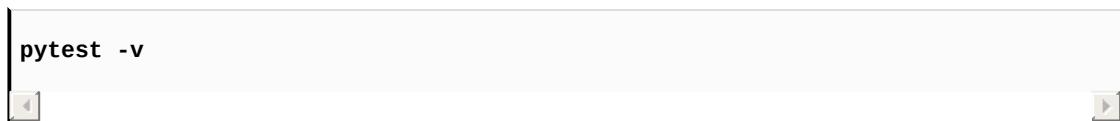
The new method tests that the combination of permission and authentication classes configured for the `PilotList` class doesn't make it possible for an unauthenticated HTTP `POST` request to create a pilot. The code calls the `post_pilot` method without configuring any credentials, and therefore the request runs without authentication. Then, the code calls `assert` twice to check for the following expected results:

1. The `status_code` attribute for the response is equal to HTTP 401 Unauthorized (`status.HTTP_401_UNAUTHORIZED`)
2. The total number of `Pilot` objects retrieved from the database is 0 because the received data to create a new pilot wasn't processed

We have increased the scenarios covered by our tests. We should write more tests related to pilots. However, with all the examples provided, you will have the necessary information to write all the tests required to make sure that each new version of a RESTful Web Service developed with Django and the Django REST framework works as expected.

Running unit tests again with pytest

Now, go to the `restful01` folder that contains the `manage.py` file, with the virtual environment activated, and run the following command to execute `pytest` again with the `-v` option to increase verbosity:



In this case, `pytest` will run all the methods whose name starts with the `test_` prefix in both the `DroneCategoryTests` and `PilotTests` classes and display the results.

The following screenshot shows a sample output generated for the new execution of the `pytest` command with the increased verbosity:

A screenshot of a terminal window on a Mac OS X system, showing the output of the `pytest -v` command. The output shows the test session starting, platform details (darwin, Python 3.6.2, etc.), settings (Django settings), root directory, and plugins. It then lists the collected items (8 items) and their individual test results, all of which passed. The final message indicates 8 passed tests in 0.94 seconds.

```
(01) Gastons-MacBook-Pro:restful01 gaston$ pytest -v
=====
platform darwin -- Python 3.6.2, pytest-3.3.1, py-1.5.2, pluggy-0.6.0 -- /Users/gaston/HillarDjangoREST/01/bin/python3
cachedir: .cache
Django settings: restful01.settings (from ini file)
rootdir: /Users/gaston/HillarDjangoREST/01/restful01, inifile: pytest.ini
plugins: xdist-1.20.1, forked-0.2, django-3.1.2, cov-2.5.1
collected 8 items

drones/tests.py::DroneCategoryTests::test_filter_drone_category_by_name PASSED [ 12%]
drones/tests.py::DroneCategoryTests::test_get_drone_categories_collection PASSED [ 25%]
drones/tests.py::DroneCategoryTests::test_get_drone_category PASSED [ 37%]
drones/tests.py::DroneCategoryTests::test_post_and_get_drone_category PASSED [ 50%]
drones/tests.py::DroneCategoryTests::test_post_existing_drone_category_name PASSED [ 62%]
drones/tests.py::DroneCategoryTests::test_update_drone_category PASSED [ 75%]
drones/tests.py::PilotTests::test_post_and_get_pilot PASSED [ 87%]
drones/tests.py::PilotTests::test_try_to_post_pilot_without_token PASSED [100%]

=====
8 passed in 0.94 seconds
(01) Gastons-MacBook-Pro:restful01 gaston$
```

We enabled verbose mode again, and therefore, the output displayed the full test names that the `test_post_and_get_pilot` and `test_try_to_post_pilot_without_token` test methods passed.

We should continue writing tests related to pilots, drone categories, drones, and competitions. It is extremely important that we cover all the scenarios for our RESTful Web Service. Automated tests will make it possible for us to make sure that each new version of our RESTful Web Service will work as expected after it is deployed to production.

We built RESTful Web Services with Django, the Django REST framework, and Python 3.6. We learned to design a RESTful Web Service from scratch, starting with the requirements, and to run some of the necessary tests to make sure our web service runs as expected. We learned to work with different command-line and GUI tools to make our development tests easy. We understood many features included in the Django REST framework and how to configure them.

Now, we are ready to create RESTful Web Services with Django and the Django REST framework. We will definitely need to dive deep into additional features, packages, and configurations. We definitely have a great baseline to develop our next RESTful Web Service with the most versatile programming language: Python.

Test your knowledge

Let's see whether you can answer the following questions correctly.

1. In a subclass of `APITestCase`, `self.client` is:
 1. The `APITestCase` instance that allows us to easily compose and send HTTP requests for testing
 2. The `APITestClient` instance that allows us to easily compose and send HTTP requests for testing
 3. The `APIClient` instance that allows us to easily compose and send HTTP requests for testing
2. Which of the following lines clean up the credentials of a method within a subclass of `APITestCase`?
 1. `self.client.credentials()`
 2. `self.client.clean_credentials()`
 3. `self.client.credentials = {}`
3. Which of the following methods for `self.client` in a method within a subclass of `APITestCase` allows us to make an HTTP POST request?
 1. `http_post`
 2. `make_http_post_request`

3. `post`
4. Which of the following methods for `self.client` in a method within a subclass of `APITestCase` allows us to make an HTTP GET request?
 1. `http_get`
 2. `make_http_get_request`
 3. `get`
5. Which of the following methods for `self.client` in a method within a subclass of `APITestCase` allows us to make an HTTP PATCH request?
 1. `http_patch`
 2. `make_http_patch_request`
 3. `patch`

The rights answers are included in the [Appendix](#), *Solutions*.

Summary

In this chapter, we learned to write unit tests for our RESTful Web Service. We installed the necessary packages and made the appropriate configurations to work with the modern and popular pytest unit test framework. Then, we wrote our first round of unit tests for the RESTful Web Service related to different scenarios with drone categories.

We worked with the different options for the pytest command to discover and run unit tests in the default mode, the increase verbosity mode, and the disable capture mode. We understood how to combine pytest with the testing classes provided by the Django REST framework.

Finally, we wrote additional unit tests for the RESTful Web Service related to different scenarios with pilots and the token authentication requirements for specific requests. We are able to continue adding tests for our RESTful Web Service with all the things we have learned.

Now, it is your turn. You can start developing RESTful Web Services with Django, Django REST framework, and Python 3.6.

Solutions

Chapter 1: Installing the Required Software and Tools

Questions	Answers
Q1	2
Q2	3
Q3	3
Q4	1
Q5	1

Chapter 2: Working with Models, Migrations, Serialization, and Deserialization

Questions	Answers
Q1	3
Q2	2
Q3	2
Q4	1
Q5	3

Chapter 3: Creating API Views

Questions	Answers
Q1	2
Q2	2
Q3	1
Q4	3
Q5	3

Chapter 4: Using Generalized Behavior from the APIView Class

Questions	Answers
Q1	2
Q2	2
Q3	1
Q4	3
Q5	3

Chapter 5: Understanding and Customizing the Browsable API Feature

Questions	Answers
Q1	3
Q2	1
Q3	2
Q4	1
Q5	2

Chapter 6: Working with Advanced Relationships and Serialization

Questions	Answers
Q1	1
Q2	2
Q3	3
Q4	2
Q5	2

Chapter 7: Using Constraints, Filtering, Searching, Ordering, and Pagination

Questions	Answers
Q1	3
Q2	2
Q3	1
Q4	3
Q5	2

Chapter 8: Securing the API with Authentication and Permissions

Questions	Answers
Q1	1
Q2	2
Q3	2
Q4	3
Q5	1

Chapter 9: Applying Throttling Rules and Versioning Management

Questions	Answers
Q1	1
Q2	1
Q3	2
Q4	3
Q5	3

Chapter 10: Automating Tests

Questions	Answers
Q1	3
Q2	1
Q3	3
Q4	3
Q5	3

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Flask: Building Python Web Services

Gareth Dwyer, Shalabh Aggarwal, Jack Stouffer

ISBN: 978-1-78728-822-5

- Build three web applications from the ground up using the powerful Python micro framework, Flask.
- Extend your applications to build advanced functionality, such as a user account
- control system using Flask-Login
- Learn about web application security and defend against common attacks, such as SQL injection and XSS
- Integrate with technologies like Redis, Sentry, MongoDB and so on
- Build applications with integrations to most of the login

mechanisms available
Don't just stop at development. Learn about deployment and post-deployment

- Use SQLAlchemy to programmatically query a database
- Develop a custom Flask extension



Building RESTful Python Web Services

Gastón C. Hillar

ISBN: 978-1-78862-015-4

- Develop RESTful APIs from scratch with Python with and without data sources
- Add authentication and permissions to a RESTful API built in Django framework
- Map URL patterns to request handlers and check how the API works

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!