Modern Django—Part 2: REST APIs, Apps, and Django REST Framework

Introduction

The previous technical portions of the guide can be found here:

- Part 0: Introduction and Initial Setup
- Part 1: Project Refactor and Meeting the Django Settings API

After completing those sections, we now have a solid base waiting for additions! This portion begins with a crash course in HTTP. Following that, a clarification on the concept of "applications" in Django. Finally the great wall of text will end with a brief introduction to Django REST Framework.

After the concepts are explained, we will meet some of Django REST Framework's key functions. They will be used to build an API application to perform functions on the built-in User model. Along the way, we will learn how to interact with REST APIs in the terminal.

Technology

The technology used in this section will include those associated to Django applications, basic HTTP and REST standards, Django REST Framework, and programs useful for working with APIs in the terminal.

- Django Applications: "The term **application** describes a Python package that provides some set of features. Applications may be reused in various projects" (source). A Django project consists of a number of Applications that can be reused in other projects. More on Django Applications below.
- HTTP: Stands for Hypertext Transfer Protocol. "HTTP functions as a request–response protocol in the client–server computing model" (source).
- HTTP methods/verbs: Methods defined by HTTP standards that "indicate the desired action to be performed on the identified resource" (source). The basic methods are GET and POST. A number of others exists such as DELETE, PUT, PATCH, etc. A specific URL can have multiple functions based on the method passed to it in the HTTP request.
- RESTful: Stands for Representational State Transfer. "REST-compliant Web services allow requesting systems to access and manipulate textual representations of Web resources using a uniform and predefined set of stateless operations" (source). What this means is that REST applications have a standard set/structure of HTTP resources that provide data to the clients from the server based on the HTTP method used.
- Django REST Framework: "Django REST framework is a powerful and flexible toolkit for building Web APIs" (source). We will use the Django REST Framework application to easily integrate a REST API into existing Django functionality!

Walkthrough

1. Crash Course in HTTP and REST

Without delving into a complete understanding of networking, I will explain HTTP (Hypertext Transfer Protocol).

1-A. How the Internet Works in Layers

There are 4 large layers in the Internet Protocol Suite. They include:

1. Application Layer
2. Transport Layer
3. Internet Layer
4. Link Layer

HTTP (Hypertext Transfer Protocol) is the structure for holding "data." It is at the top most layer, the Application Layer. It is moved around the internet via the Transport Layer. Common Transport Layer technologies include TCP and UDP. TCP/UDP packets wrap HTTP and are moved along via the Internet and Link Layers. A broader understanding of how networking and the Internet works can be found by reviewing the 7 Layer OSI Model.

1-B. URLS and Resources

HTTP is typically accessed and used through the form of an HTTP URI. A URI is a Uniform Resource Identifier. A URL (Uniform Resource Locator) is a specific type of URI used to identify a web address. So, when we talk about a link such as `https://www.instagram.com` we are talking about an HTTP resource. That is, an HTTP URI, taking the form of an HTTP URL.

We use URLs to define HTTP resources that are available to us. They should be self-explanatory! Such as `https://www.instagram.com/accounts/login/` describing a resource of logging in a user.

1-C. HTTP Requests and Responses

HTTP resources are provided via **requests** and **responses**. An HTTP **request** says "give me this" or "do something" to a server. The server will do a number of processes based on the data from the **request**, determining which functions should be called and what data should be returned. Once those processes are completed, the server sends back an HTTP **response**. That response says "I did this" or "take this <resource>" with success/error messaging.

1.4 HTTP Methods

When an HTTP request is made, it is given a specific method that defines the action it will take on a resource. A particular resource available by a URL can take HTTP requests with different types of methods, and may perform different functions based on the method specified.

The most important HTTP methods to know are `GET` and `POST`.

- `GET`: used to retrieve data! Get a homepage, get images on Instagram
- `POST`: send data to the server! Such as login, post a picture on Instagram

All other HTTP methods are pretty self explanatory. [You can read more here.]

To show how an HTTP resource can take different types of HTTP request + method combinations, we will use an example URL from Instagram:

`https://www.instagram.com/media/<media-id>/likes`

[Documentation for this API can be found here.]

This URL allows us to perform operations upon a user's photo or video likes. By making different HTTP requests to this URL different operations can be done:

- Requesting with `GET`: will return a list of all the likes on the media
- Requesting with `POST`: will have the current user like the media

1.5 Status Codes

After a request has been processed, the server will attempt to send an HTTP response back to the client. The beginning of the response will contain a status code for the outcome of the request.

Wikipedia states that status codes are broken up into five different major groups (source). These groups and a common example are below:

1. Informational 1XX: Probably won't encounter 1XX's often
2. Successful 2XX: 200 OK (getting a page)
3. Redirection 3XX: 304 Not Modified (cached CSS has not changed)
4. Client Error 4XX: 404 Not Found (page does not exist)
5. Server Error 5XX: 500 Internal Server Error (generic error)

When using the `manage.py runserver` command you will see requests, methods, responses, and status codes being show in the terminal! Two GET request/responses are shown below. One has an internal server error denoted by the 500 error and one returned 200 OK for the root URL.

```
[14/Mar/2017 00:49:24] "GET /admin/ HTTP/1.1" 500
[14/Mar/2017 00:49:34] "GET / HTTP/1.1" 200 1767
```

1.6 RESTful API

The HTTP resource from Instagram used earlier, is part of their RESTful API!

`https://www.instagram.com/media/<media-id>/likes`

As described above, REST stands for Representational State Transfer. RESTful APIs are a structured set of HTTP resources. Those resources describe functions/database models and perform manipulation upon those structures via a set of standard HTTP requests.

These standards specify how URLs should be structured and what functions should happen based on a given HTTP methods. RESTful APIs should be structured such that all resources represent a data model. Methods supplied to the resource should be self explanatory. `GET`ing a resource should return all of values of that model, such as `users/` returning all users in the database. If say a Primary Key value (association in the database) is supplied, it should return only that user's data

`users/1/`. Again, using the same resource with `POST` would require the necessary data to create a new user. `POST` on `users/1/` would represent the ability to update the data for user 1.** There are a number of other specifics that we will go over when writing the API!

We should structure all of our URLs to map to our database models to make it as RESTful as possible! We should **NOT** create resources that provide random arbitrary functions by requesting them.

** In a standard workflow we would use `PUT` or `PATCH` for these types of function. However we are talking in basic examples here.

1.7 Summary

For the time being, the above should hold you over as we create a base API. However in the future, mastering HTTP concepts will be essential for writing custom APIs and custom response handlers on the front end.

---

2. Intro to Django Applications

Back to Django! It is now important to understand the concept of Django applications before we create another one.

> "But we haven't even used `manage.py startapp` ?"—A naive developer

As described by the documentation, Django applications are "a Python package that provides some set of features", reusable, and are "a combination of models, views, templates, …, URLs, etc".

Read that again.

> "*a Python package that provides some set of features*"

The takeaway from that statement is: We have already made a Django application! It is the files included in the `config` directory. Any Python package that provides some set of features is an application. A "project" is an application. It is important to decouple the idea that a Django project and a Django application. They are vastly different things. This concept is described in the documentation as:

> "a Django application is just a set of code that interacts with various parts of the framework."

With that in mind, a *project* in Django is, in itself, just a different structured reusable application. All applications we create should be reusable! That means if we create an API application we should be able to clone that code into any other "project" application we like and it should just work! This also works in the reverse! We can have a "project" application (such as the one we have created) one that any other correctly configured application could work with!

A "project" application acts as the ringmaster and gatekeeper for Django's libraries to interact with functions of the third party/handmade applications. These interactions are made possibly by the `AppConfig` class (documentation, source). A settings file knows of other applications and their

functions through the use of `AppConfig`, which is typically found in the default `app.py`. By supplying a directory in `INSTALLED_APPS`, it will first look into that directory's `app.py` for a configuration. (The `AppConfig` could be supplied in another file, however we will assume the base structure given by `manage.py startapp` command).

Now in a moment, we will create another application using the `startapp` command (source). The structure of an application created with `startapp` command will be:

```
app
├── __init__.py
├── admin.py            # Add this app's models to the Admin app
├── apps.py             # Entrypoint to this app
├── migrations          # Set of database changes made by models
│   ├── __init__.py
├── models.py           # Database relations used by this app
├── tests.py            # Tests to run on this app
└── views.py            # Functions to provide model/custom data
```

Most Django applications will be built out to include a `urls.py`, migration files, and directories such as `templates` and `static`.

From here we will add to our database tables as we make additions to `models.py`. Data created by `models.py` classes will be manipulated and moved using HTTP response via `views.py`. `urls.py` will handle routing HTTP request to their appropriate view.

We will see the structure of our application change as we add in more files that fit Django defaults/best practices or third party applications.

3. Intro to Django REST Framework

Now that we understand the basics of Django Applications, we will take a brief look at Django REST Framework (DRF). DRF is a Third Party Application that we will use in conjunction with our own application to create a REST API. It provides us with many features that connect deeply with Django's existing structures, helping us create RESTful HTTP resources that correspond to the models we create! We will use DRF to:

- to handle creation of URLs
- perform data manipulation with generic views and ViewSets
- authenticate users
- handle request permissions
- provide a nice document for our API

We will see Django REST Framework's features up close and explain each aspect, in depth, as we begin working with it!

Done with the big reading! Onto adding the API application.

4. Addition of Non Configuration based Applications

Keeping inline with the larger structure of our project, we will create a directory at the root called `project`. Inside of this directory, we will add our other applications. Create the directory then move into it:

```
# Commands
mkdir project
cd project/
```

We will then use the `startapp` command to create an application inside of that directory called `api`.

```
# Command
python ../manage.py startapp api
```

Our `api` directory should reflect the following:

```
api/
├── __init__.py
├── admin.py
├── apps.py
├── migrations
│   └── __init__.py
├── models.py
├── tests.py
└── views.py
```

Now that the application has been created, we will create a `urls.py`. First create the file:

```
# Command
touch project/api/urls.py
```

We will then add the defaults needed for a URL file so Django doesn't get angry:

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

urlpatterns = [
]
```

Now we must ensure `config` is aware of our new app by adding the appropriate changes to `config/settings/base.py` and `config/urls.py`. In `base.py`, add the `api` app to `LOCAL_APPS`.

```
LOCAL_APPS = (
    'project.api',
)
```

It should be picked up automatically based on `api.app.AppConfig`!

Now we can add `api`'s URLs to the base URLs, `config/urls.py`. We will make the `api` URLs function as the root level, as this is intended to be an API server after-all.

Make your `config/urls.py` reflect the following additions:

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

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^', include('project.api.urls')),
]
```

We have added `django.conf.urls.include` and used include `api/urls.py`.

6. Installing and Configuring Django REST Framework

Now that the structure has been created, we need to install Django REST Framework, add its configuration to `config/settings/base.py`, and make appropriate changes.

6–1. Install Django REST Framework

In `requirements/base.txt` make the addition:

```
djangorestframework==3.6.2
```

Then install the dependency with our usual command:

```
# Command
pip install -Ur requirements/local.txt

# Expected Output
...
Successfully installed djangorestframework-3.6.2
```

6–2. Configure Django REST Framework

Inside of `config/settings/base.py` we must add Django REST Framework to `INSTALLED_APPS` and add the configurations.

In the `THIRD_PARTY_APPS` add `rest_framework`:

```
THIRD_PARTY_APPS = (
    'rest_framework',
)
```

At the bottom of our `base.py` file make these additions:

```
REST_FRAMEWORK = {
}
```

This is the place where we will add `REST_FRAMEWORK` configurations. For now we will leave it empty, allowing the defaults to do their work behind the scenes.

At this point we have successfully added the ability to use Django REST Framework. However we have not built out any API.

Now we will add the base User endpoints defined by the Quickstart guide at the Django REST Framework website. These resources will allow Admin users to make changes to the User model. We will make more additions later (token authentication anyone?), but it is important to cover/show the ease of DRF first!

7. Serializers

We must first create some serializers. Serializers are defined by DRF as:

> Serializers allow complex data such as querysets and model instances to be converted to native Python datatypes that can then be easily rendered into `JSON`, `XML` or other content types. Serializers also provide deserialization, allowing parsed data to be converted back into complex types, after first validating the incoming data.
>
> The serializers in REST framework work very similarly to Django's `Form` and `ModelForm` classes. We provide a `Serializer` class which gives you a powerful, generic way to control the output of your responses, as well as a `ModelSerializer` class which provides a useful shortcut for creating serializers that deal with model instances and querysets.

With that we can break serializers down to two main uses:

1. **Get model data from the database in JSON**
2. **Use them like forms to validate data and create instances of a model**

There are different types of serializers, but we will focus our efforts today on HyperlinkedModelSerializers. To understand them, we must first look at ModelSerializers:

> **The `ModelSerializer` class is the same as a regular `Serializer` class, except that**:
>
> It will automatically generate a set of fields for you, based on the model.

> It will automatically generate validators for the serializer, such as unique_together validators.
>
> It includes simple default implementations of `.create()` and `.update()`.

HyperlinkedModelSerializers build on top of ModelSerializers by using a URL instead of primary key values to define relations. Thus when getting data back from a serializer, you will get a field `url` instead of `pk`.

All of those things can manually be done, however we should opt to use HyperlinkedModelSerializer 9 times out of 10 when dealing with Model manipulation. In more advanced cases, we may need to write an entire `Serializer` by hand.

Full documentation on DRF Serializers can be found here.

Now we will create our Serializer for the User model. First create `serializers.py`:

```
# Command
touch project/api/serializers.py
```

Inside of the file `project/api/serializers.py` we will add the class `UserSerializer`:

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

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

We have imported the User model and created a HyperlinkedModelSerializer that will get us the data associated with the model, as well as the URL to the instance.

That URL would go to that model's User Detail page. When that User is represented as a foreign key to other objects, the URL will be displayed instead of a simple PK.

8. Views

We will now create a View that allows a URL to get data from a serializer and return that data to a user/endpoint. To get there we need to understand Views, Class-Based Views, and ViewSets.

8–1. Views

First, one must understand what a View is. As defined by Django:

> A view function, or *view* for short, is simply a Python function that takes a Web request and returns a Web response.

Well it's a good thing we know all about HTTP requests and responses now!

Views "handle linking the URLs, HTTP method dispatching (GET, POST, etc)", among other things.

8–2. Class-Based Views

Most beginner Django developers end up writing a lot of views by hand. The idea of Class-Based Views is either unheard of (yes, it is not heavily documented that one should use them at the start) or frightening upon discovery. However a clear definition of what they are is provided by Classy Class-Based Views:

> "Django's class-based generic views provide abstract classes implementing common web development tasks. These are very powerful, and heavily-utilise Python's object orientation and multiple inheritance in order to be extensible. This means they're more than just a couple of generic shortcuts—they provide utilities which can be mixed into the much more complex views that you write yourself."

Classy Class-Based Views is probably the most helpful tool for working with Class-Based Views. Use it. SHARE IT. It shows you which attributes and methods are available to override. Familiarizing yourself with its documentation will help tremendously when deciphering issues you may have when overriding defaults.

I will not really be discussing any more topics related to Django's Class-Based Views, but I believe it is important to understand this concept of abstraction for our next step. Understanding and using Class-Based Views will be instrumental for those of you creating Django projects that are non API based / include heavy server side rendering.**

** If lots of discussion is generated from Class Based View, I may write a separate document for that topic.

Further Reading on Class-Based Views:

- The Django Documentation's Introduction to Class-Based Views

8–3. Django REST Framework Generic Views and ViewSets

Just as Django has Class-Based Views, Django REST Framework has their own. Again, many built-in functions required should not be rewritten. You can easily write a View by hand with DRF, or you can use the expansion to Generic Views provided. The DRF Generic Views are defined here.

We will use the Generic Views to write out more custom API functions! However most of our usecases will use **ViewSets**.

ViewSets are defined by DRF as:

> **"a type of class-based View, that does not provide any method handlers** such as `.get()` or `.post()`, and instead provides actions such as `.list()` and `.create()`."

This means many of the views we would have had to create with Generic Views are now bundled into one! Another great reason to use ViewSets is that, when combined with DRF Routers (more on that soon), our URLs can automatically be created!

The Class-Based Views, Viewsets, and Serialiers provided by DRF also have their own reference at Classy Django REST Framework.

8–4. Creating our User ViewSet

Now that we have gone over the concept of Views, it's time to create our User ViewSet. We will build the `UserViewSet` off of the `ModelViewSet` class.

From the documentation we can see that `ModelViewSet` includes the following functions: `.list()`, `.retrieve()`, `.create()`, `.update()`, `.partial_update()`, and `.destroy()`. So now we can perform these functions without needing to write them by hand! However, we must also override some of the attributes inherited from `GenericAPIView`, where `queryset` and `serializer_class` must be set. View the other attributes here.

Open up `project/api/views.py` and make these additions:

```
from django.contrib.auth.models import User
from rest_framework import viewsets
from project.api.serializers import UserSerializer

class UserViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows users to be viewed or edited.
    """
    queryset = User.objects.all()
    serializer_class = UserSerializer
```

In the above we have:

- Imported the `User` model, `viewsets` library, and our new `UserSerializer`
- Created a view, `UserViewSet` that inherits from `ModelViewSet`
- Overridden the `ModelViewSet` attributes of `queryset` and `serializer_class`

There are more attributes and functions one can override in `ModelViewSet`, however we must set `queryset` and `serializer_class` to those for the User model.

- `queryset`: the queryset that will return objects on `.list()`
- `serializer_class`: the serializer class that is used for the functions

9. URLs and DRF Routers

Now our User's serializer and views are complete. Time to create a router that will create URLs for us from a ViewSet!

Routers have two attributes that must be set, the `prefix` and `viewset`. The third option `base_name` can be set later. The `prefix` is a regular expression that will start our URLs. We will add the router for users to our `api/urls.py`, shown below:

```
from django.conf.urls import url, include
from rest_framework import routers
from project.api import views

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

urlpatterns = [
    url(r'^', include(router.urls)),
]
```

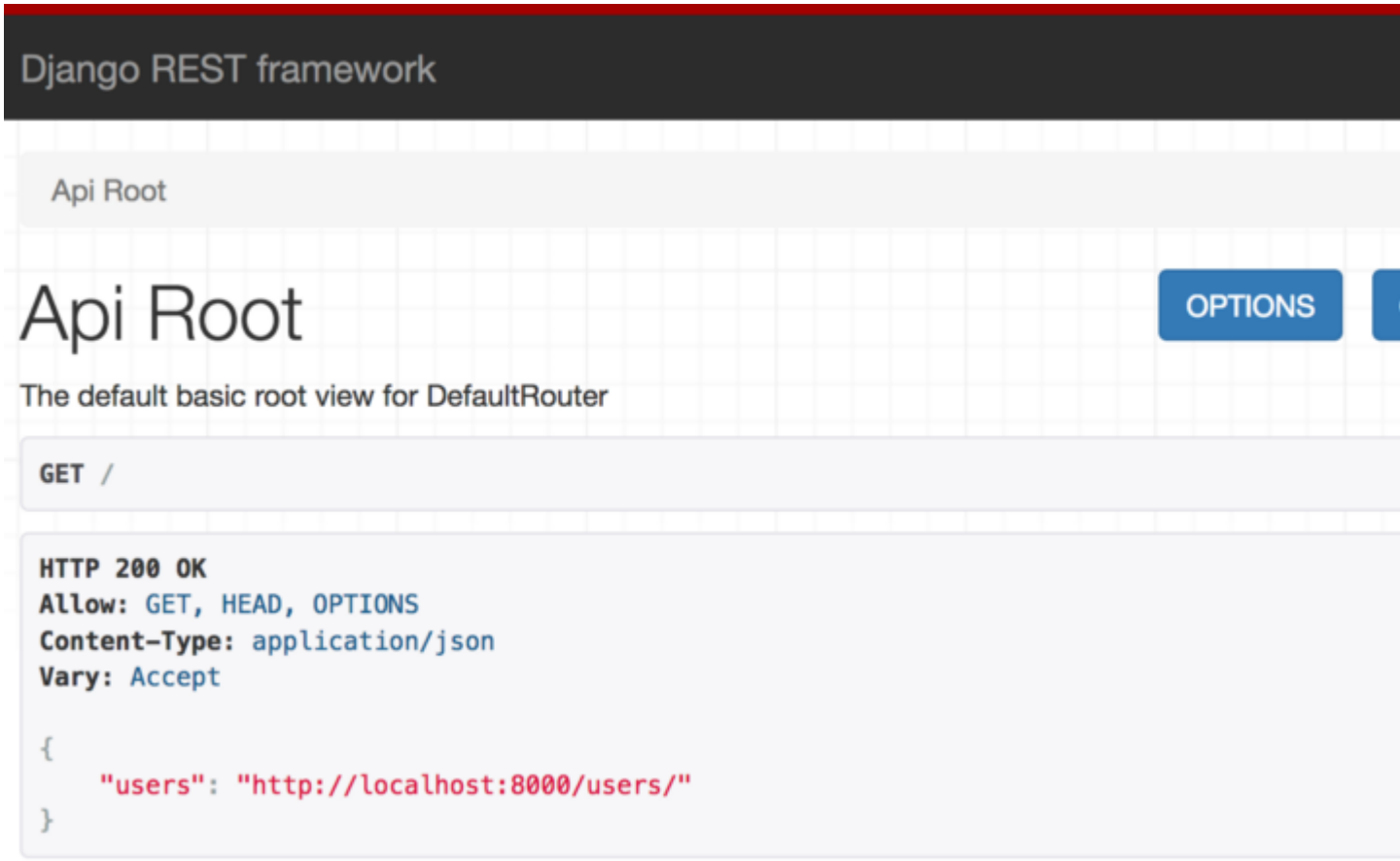
With that we should be able to manipulate User models in a RESTful way! Further reading on how to make URLs and general info on regular expressions.

10. Testing Your API!

We can test our API in two ways, via the terminal or through the Django REST Framework interface in the web browser. To begin testing, first run the Django server.

```
# Command
python manage.py runserver
```

Now we can move to the browse-able API Root in the browser! Navigate to http://localhost:8000/ and you should see this page!



You can click on the link for users, to be brought to the User List, and clicking on a user will bring you to a User Detail. The GUI allows you to create, delete, or update values on the User model or others. You can also use these functions through the terminal with tools like curl.

Using `curl`, we can make a GET request to `/users/` and have the data be formatted in the terminal as JSON:

```
curl -H 'Accept: application/json; indent=4' http://localhost:8000/users/
```

When a request has been made to `/users/` at the terminal or in the browser this happens: An HTTP request with a GET method has been made to request the resource `/users/`. The router has sent that request to the `UserViewSet`, which determined that it should return a list of all Users.

If another request was made with a set of data to the same URL it would be as a POST to create a new user!

Summary

In this section we have gone over a wide range of information; including the concepts of HTTP, REST, and Django Applications. We have then demonstrated how Django REST Framework exercises HTTP concepts with URLs and Views.

The GitHub for this project and all its sections can be found here.

As always, feedback, tips, tricks, and criticism are always welcome on the comments or by email.

Best and thank you,