

django

REST

framework

Why REST API?

Before we get to the code, it's worth considering why you would want to build an API. If someone had explained these basic concepts to me before I started, I would have been so much better off.

A REST API is a standardized way to provide data to other applications. Those applications can then use the data however they want. Sometimes, APIs also offer a way for other applications to make changes to the data.



There are a few key options for a REST API request:

- GET – The most common option, returns some data from the API based on the endpoint you visit and any parameters you provide
- POST – Creates a new record that gets appended to the database
- PUT – Looks for a record at the given URI you provide. If it exists, update the existing record. If not, create a new record
- DELETE – Deletes the record at the given URI
- PATCH – Update individual fields of a record

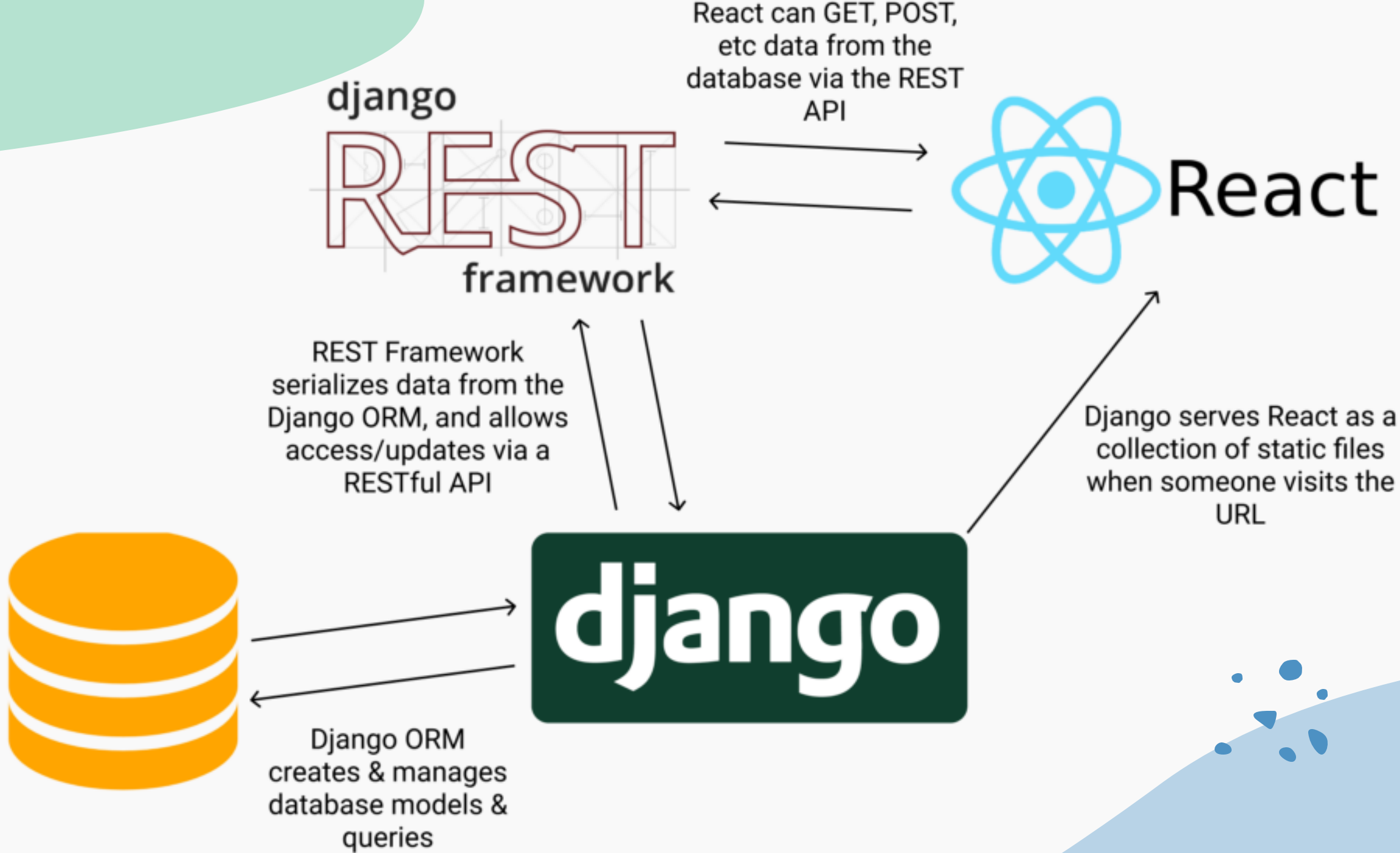


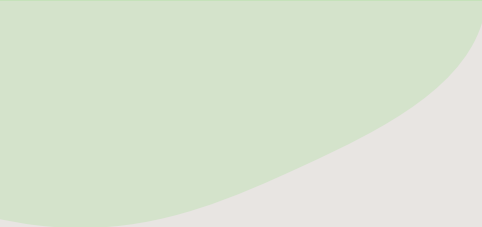
Why REST API?

Typically, an API is a window into a database. The API backend handles querying the database and formatting the response. What you receive is a static response, usually in JSON format, of whatever resource you requested.

REST APIs are so commonplace in software development, it's an essential skill for a developer to know how they work. APIs are how applications communicate with one another or even within themselves.

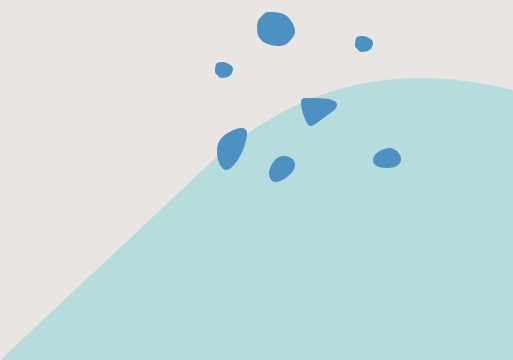






For example, in web development, many applications rely on REST APIs to allow the front end to talk to the back end. If you're deploying a React application atop Django, for instance, you'll need an API to allow React to consume information from the database.

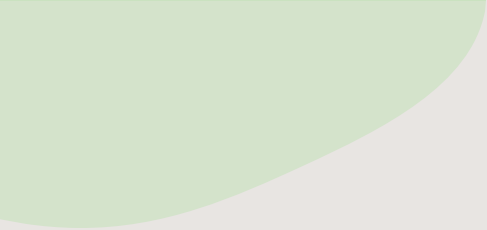
The process of querying and converting tabular database values into JSON or another format is called **serialization**. When you're creating an API, correct serialization of data is the major challenge.



Why Django REST Framework?

- Typically, an API is a window into a database. The API backend handles querying the database and formatting the response. What you receive is a static response, usually in JSON format, of whatever resource you requested.
- REST APIs are so commonplace in software development, it's an essential skill for a developer to know how they work. APIs are how applications communicate with one another or even within themselves.

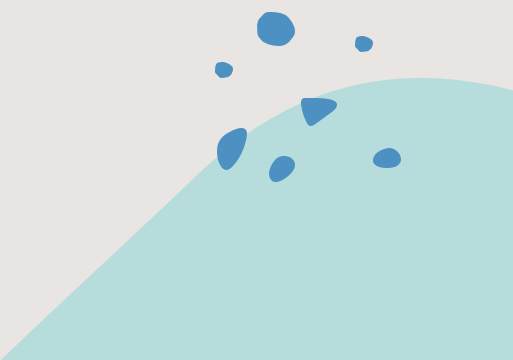




Think of the Django ORM like a librarian, pulling the information you need for you, so you don't have to go get it yourself.

As a developer, this frees you up to worry about the business logic of your application and forget about the low level implementation details. Django ORM handles all that for you.

The Django REST Framework, then, plays nicely with the Django ORM that's already doing all the heavy lifting of querying the database. Just a few lines of code using Django REST Framework, and you can serialize your database models to REST-ful formats.



To-do list to create a REST API in Django

Okay, so based on what we know, what are the steps to creating a REST API?

- Set up Django
- Create a model in the database that the Django ORM will manage
- Set up the Django REST Framework
- Serialize the model from step 2
- Create the URI endpoints to view the serialized data

If it seems simple, that's because it is. Let's get to it!

Set up Django

Install Django

- Create new project
- Install Django

```
$ pip install django
```

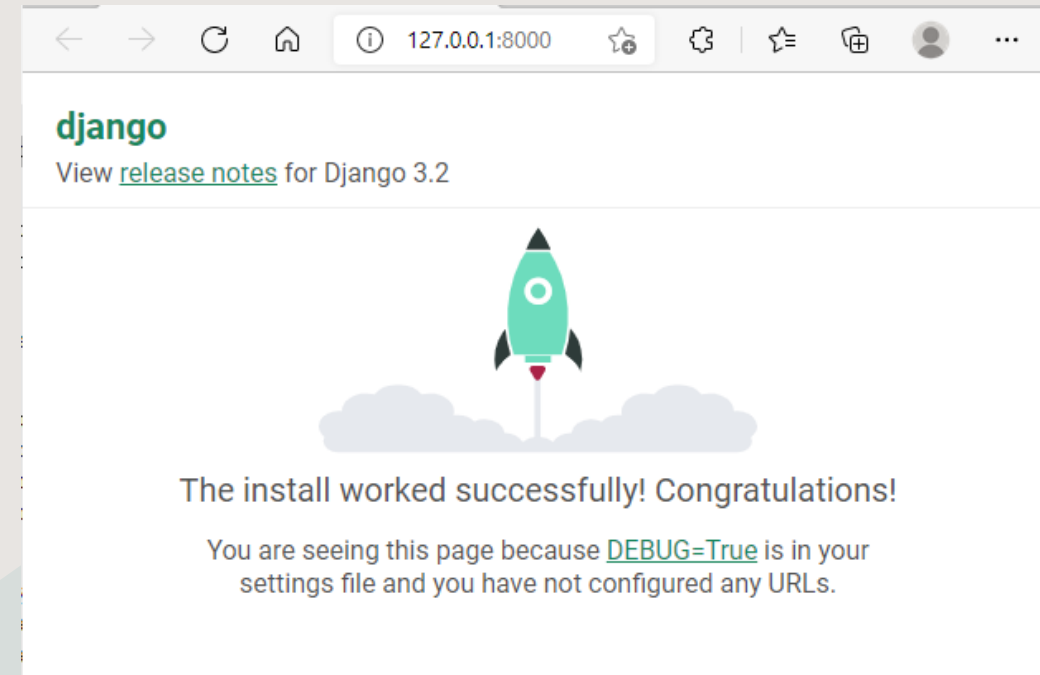
- Next, let's start a new Django project:

```
$ django-admin startproject mysite  
$ cd mysite/
```

- Let's make sure it works. Test run the Django server:

```
$ python manage.py runserver
```

- Go to localhost:8000 and you should see the Django welcome screen!



Set up Django

Create API app

We could build our application with the folder structure the way it is right now. However, best practice is to separate your Django project into separate apps when you build something new.

So, let's create a new app for our API:

```
$ python manage.py startapp myapi
```

Set up Django

Register the myapi app with the mysite project

We need to tell Django to recognize this new app that we just created. The steps we do later won't work if Django doesn't know about myapi.

So, we edit `mysite/settings.py`:

```
INSTALLED_APPS = [  
    'myapi.apps.MyapiConfig',  
    ... # Leave all the other INSTALLED_APPS  
]
```

Set up Django

Migrate the database

Remember how I said Django allows you to define database models using Python?

Whenever we create or make changes to a model, we need to tell Django to migrate those changes to the database. The Django ORM then writes all the SQL `CREATE TABLE` commands for us.

It turns out that Django comes with a few models already built in. We need to migrate those built in models to our database.

(For those of you thinking, “We didn’t create a database!” You’re right. But Django will create a simple SQLite database for us if we don’t specify differently. And SQLite is awesome!)

So, let’s migrate those initial models:

```
$ python manage.py migrate
```

Set up Django

Create Super User

One more thing before we move on.

We're about to create some models. It would be nice if we had access to Django's pretty admin interface when we want to review the data in our database.

To do so, we'll need login credentials. So, let's make ourselves the owners and administrators of this project. THE ALL-POWERFUL SUPERUSER!!!

```
$ python manage.py createsuperuser
```

```
(venv) C:\Users\MarizzaMil\PycharmProjects\django-rest\mysite>python manage.py createsuperuser
Username (leave blank to use 'marizzamil'): admin
Email address: admin@gmail.com
Password:
Password (again):
This password is too short. It must contain at least 8 characters.
This password is too common.
This password is entirely numeric.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
```

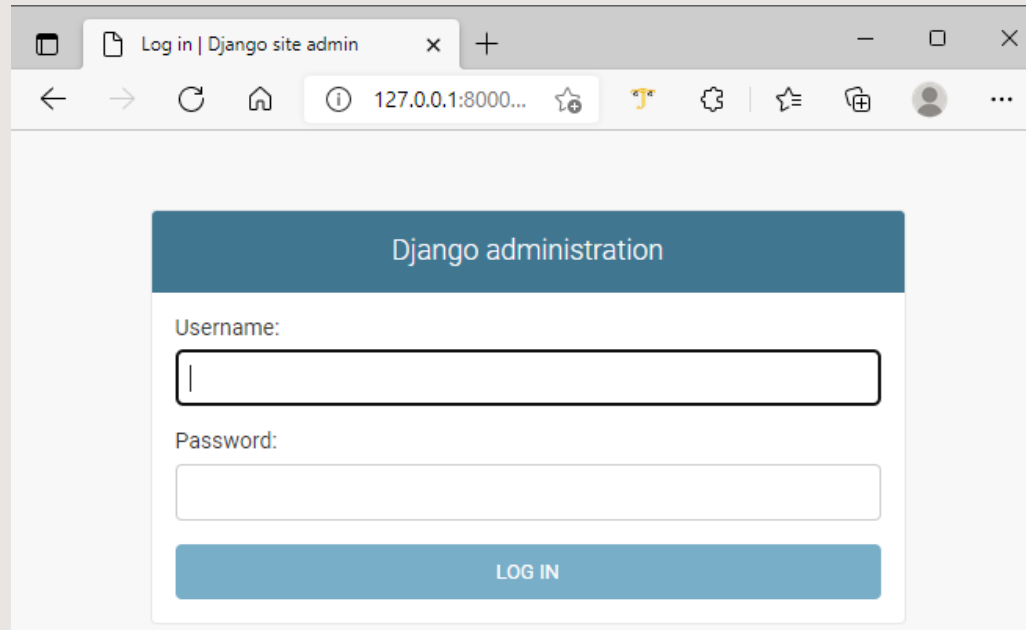
Set up Django

Create Super User

Let's verify that it works. Start up the Django server:

```
$ python manage.py runserver
```

And then navigate to localhost:8000/admin

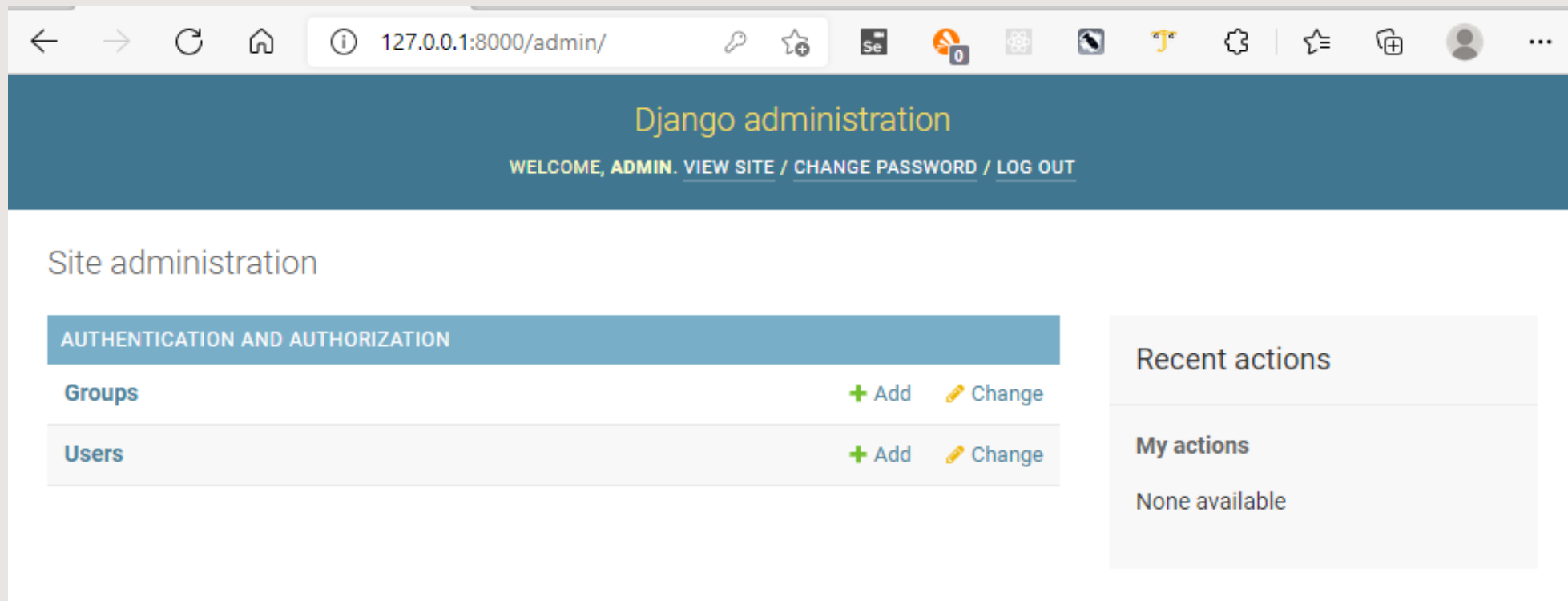


The screenshot shows a web browser window with the title "Log in | Django site admin". The address bar displays "127.0.0.1:8000...". The main content area shows the "Django administration" login form. The form has a dark blue header with the text "Django administration". Below the header, there are two input fields: "Username:" and "Password:". The "Username:" field is a text input with a cursor. The "Password:" field is a password input. At the bottom of the form is a blue button labeled "LOG IN".

Set up Django

Create Super User

Log in with your superuser credentials, and you should see the admin dashboard:



Create a model in the database that Django ORM will manage

Let's make our first model!

We'll build it in `myapi/models.py`, so open up that file.

myapi/models.py

Let's make a database of superheroes! Each hero has a name and an alias that they go by in normal life. We'll start there with our model:

```
from django.db import models

class Hero(models.Model):
    name = models.CharField(max_length=60)
    alias = models.CharField(max_length=60)

    def __str__(self):
        return self.name
```

name and alias are character fields where we can store strings. The `__str__` method just tells Django what to print when it needs to print out an instance of the Hero model.



Create a model in the database that Django ORM will manage

Make migrations

Remember, whenever we define or change a model, we need to tell Django to migrate those changes.

```
$ python manage.py makemigrations
```

```
$ python manage.py migrate
```



Create a model in the database that Django ORM will manage

Register Hero with the admin site

Remember that awesome admin site that comes out of the box with Django? It doesn't know the Hero model exists, but with two lines of code, we can tell it about Hero. Open `myapi/admin.py` and make it look like this:

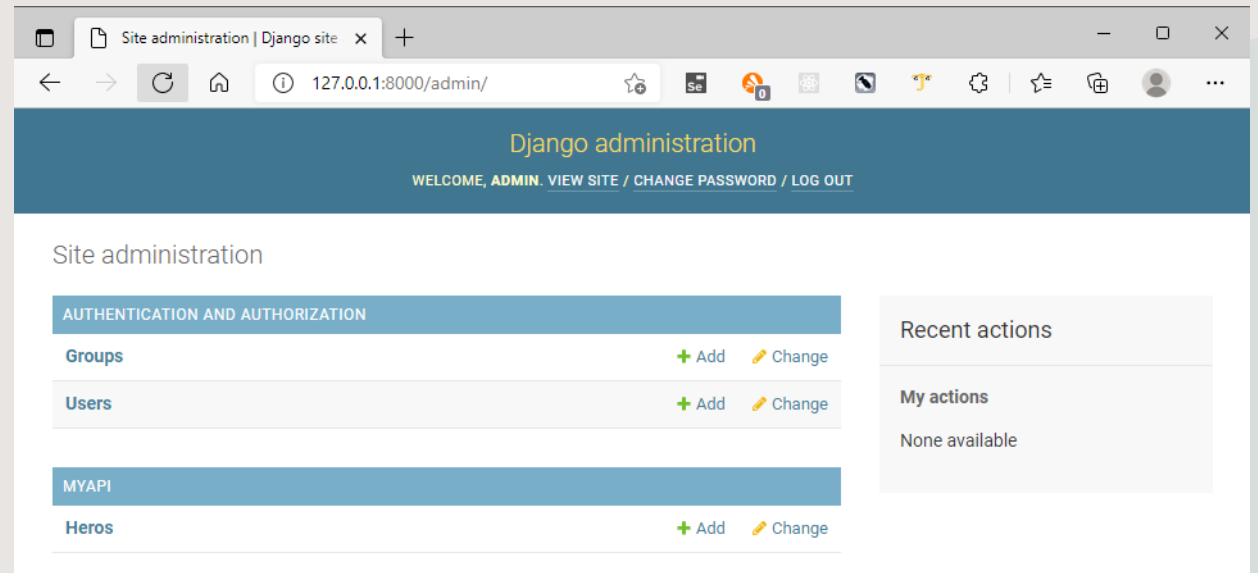
```
from django.contrib import admin
from .models import Hero

admin.site.register(Hero)
```

Now run the Django server:

```
$ python manage.py runserver
```

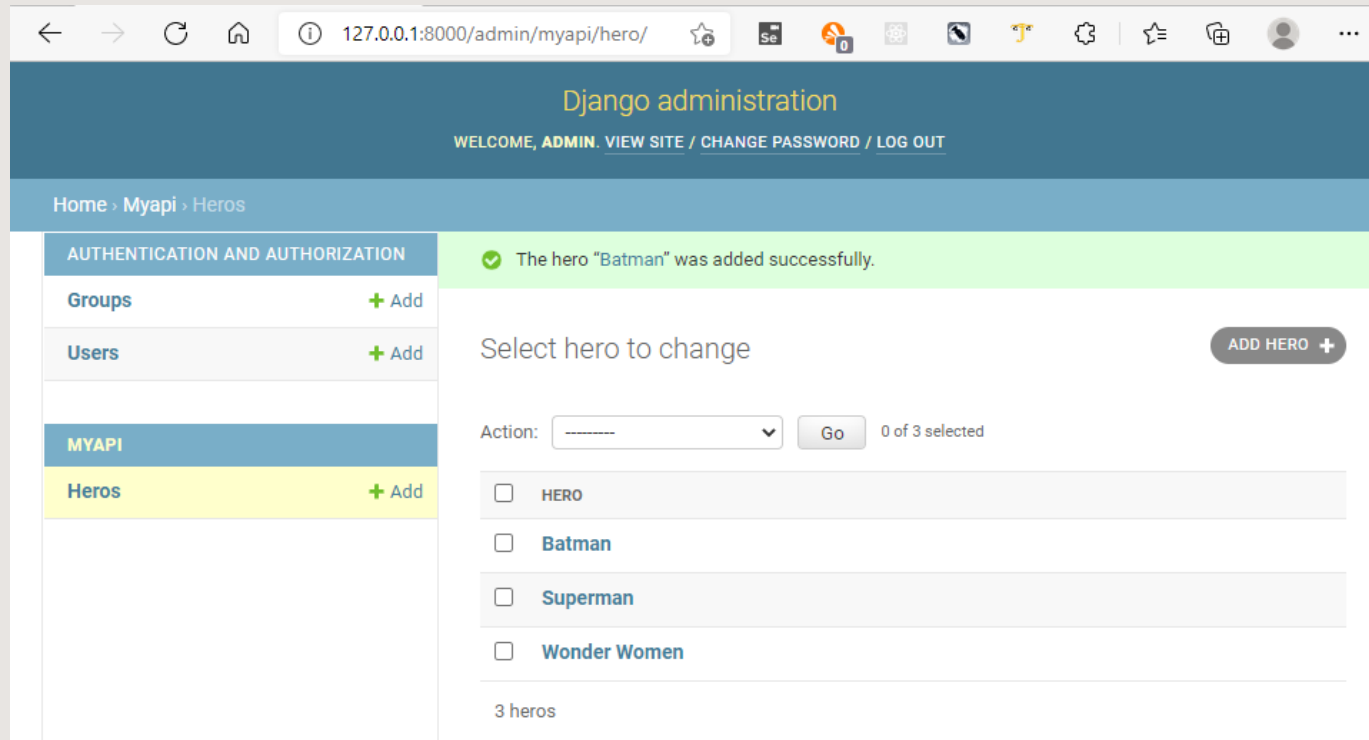
And visit localhost:8000/admin



Create a model in the database that Django ORM will manage

Create some new heroes

While we're on the admin site, might as well create a few heroes to play around with in our application.
Click "Add." Then, make your heroes!



Set up Django REST Framework

Okay, time to start thinking about our heroes API. We need to serialize the data from our database via endpoints.

To do that, we'll need Django REST Framework, so let's get that installed.

```
$ pip install djangorestframework
```

Now, tell Django that we installed the REST Framework in mysite/settings.py:

```
INSTALLED_APPS = [  
    # All your installed apps stay the same  
    ...  
    'rest_framework',  
]
```

Serialize the Hero model

Now we're starting to get into some new waters. We need to tell REST Framework about our Hero model and how it should serialize the data.

Remember, serialization is the process of converting a Model to JSON. Using a serializer, we can specify what fields should be present in the JSON representation of the model.

The serializer will turn our heroes into a JSON representation so the API user can parse them, even if they're not using Python. In turn, when a user POSTs JSON data to our API, the serializer will convert that JSON to a Hero model for us to save or validate.

Serialize the Hero model

To do so, let's create a new file – myapi/serializers.py

In this file, we need to:

- Import the Hero model
- Import the REST Framework serializer
- Create a new class that links the Hero with its serializer

Here's how:

```
from rest_framework import serializers
from .models import Hero

class HeroSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Hero
        fields = ('name', 'alias')
```

Display the data

Now, all that's left to do is wire up the URLs and views to display the data!

Views

Let's start with the view. We need to render the different heroes in JSON format.

To do so, we need to:

1. Query the database for all heroes
2. Pass that database queryset into the serializer we just created, so that it gets converted into JSON and rendered

In myapi/views.py:

```
from rest_framework import viewsets

from .serializers import HeroSerializer
from .models import Hero

class HeroViewSet(viewsets.ModelViewSet):
    queryset = Hero.objects.all().order_by('name')
    serializer_class = HeroSerializer
```

ModelViewSet is a special view that Django Rest Framework provides. It will handle GET and POST for Heroes without us having to do any more work.

Display the data

Site URLs

The last step is to point a URL at the viewset we just created.

In Django, URLs get resolved at the project level first. So there's a file in **mysite/** directory called **urls.py**.

Head over there. You'll see the URL for the admin site is already in there. Now, we just need to add a URL for our API. For now, let's just put our API at the index:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('myapi.urls')),
]
```

Display the data

API URLs

If you're paying attention and not just blindly copy-pasting, you'll notice that we included `'myapi.urls'`. That's a path to a file we haven't edited yet. And that's where Django is going to look next for instructions on how to route this URL.

So, let's go there next – `myapi/urls.py`:

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

router = routers.DefaultRouter()
router.register(r'heroes', views.HeroViewSet)

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

Display the data

Notice we added something called router that we imported from `rest_framework`.

The REST Framework router will make sure our requests end up at the right resource dynamically. If we add or delete items from the database, the URLs will update to match. Cool right?

A router works with a viewset (see `views.py` above) to dynamically route requests. In order for a router to work, it needs to point to a viewset, and in most cases, if you have a viewset you'll want a router to go with it.

So far, we've only added one model+serializer+viewset to the router – Heroes. But we can add more in the future repeating the same process above for different models!

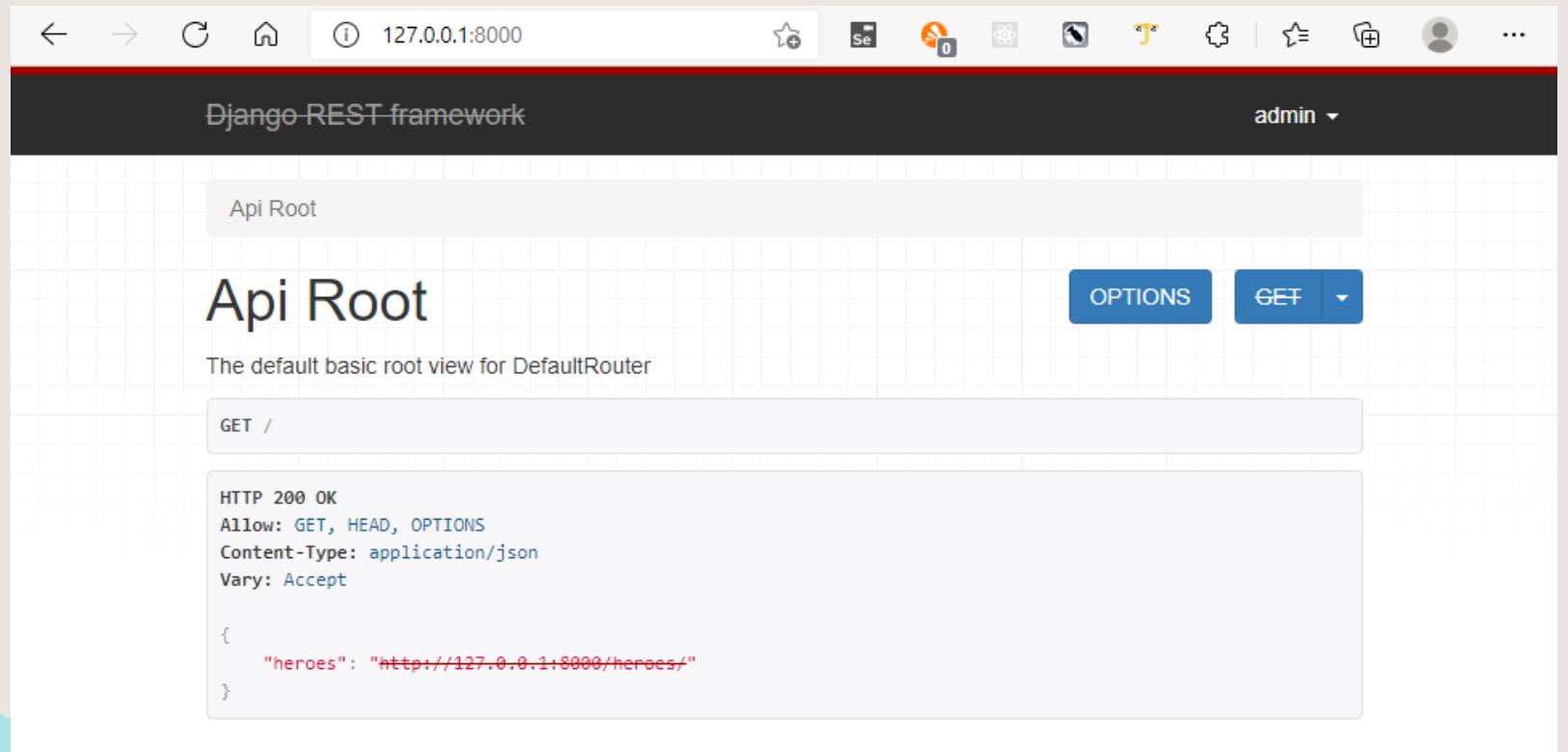
Of course, if you only want to use standard [DRF Views](#) instead of viewsets, then `urls.py` will look a little

Test it out!

Start up the Django server again:

```
$ python manage.py runserver
```

Now go to localhost:8000



Test it out!

Visit the endpoint via GET

If we click the link (Hyperlinks are good REST-ful design, btw), we see the heroes API results:

The screenshot shows a web browser window with the URL `127.0.0.1:8000/heroes/`. The page title is "Django REST framework" and the user is logged in as "admin". The breadcrumb trail is "Api Root / Hero List". The main heading is "Hero List" with "OPTIONS" and "GET" buttons. Below the heading, the method "GET /heroes/" is shown. The response details are:

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

The JSON response body is:

```
[
  {
    "name": "Batman",
    "alias": "Bruce Wayne"
  },
  {
    "name": "Superman",
    "alias": "Clark Kent"
  },
  {
    "name": "Wonder Women",
    "alias": "Diana Prince"
  }
]
```

At the bottom, there are tabs for "Raw data" and "HTML form". The "HTML form" tab is active, showing a form with two input fields: "Name" and "Alias". A "POST" button is located at the bottom right of the form.

Test it out!

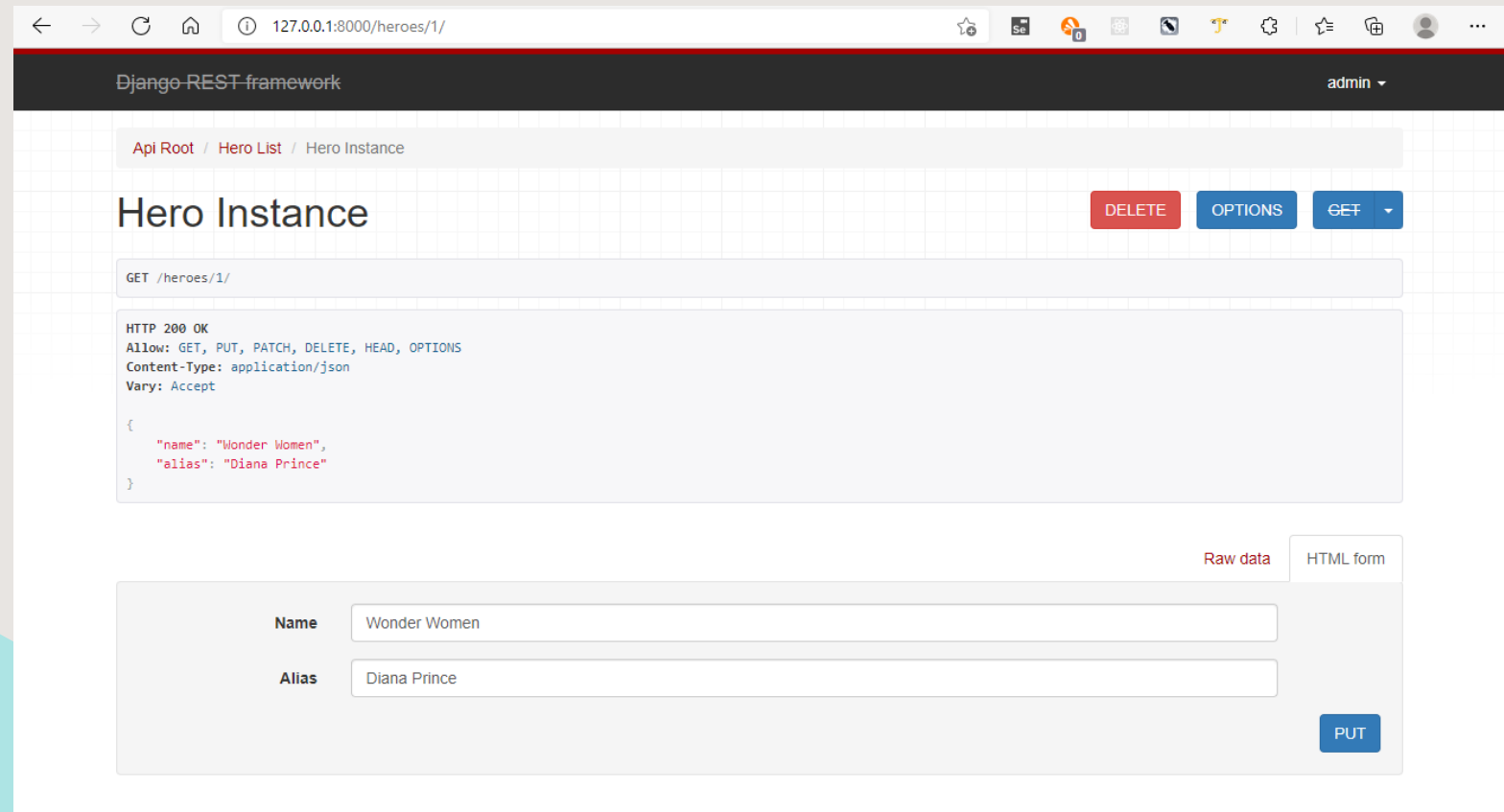
GET an Individual Hero

We can GET a single model instance using its ID.

Django REST Framework viewsets take care of this for us.

If you go to `127.0.0.1:8000/heroes/<id>/` where `<id>` is the ID of one of your Heroes models, you'll be able to see just that hero.

For example, <http://127.0.0.1:8000/heroes/1/> for me returns:



The screenshot shows a web browser window with the address bar set to `127.0.0.1:8000/heroes/1/`. The page title is "Django REST framework" and the user is logged in as "admin". The breadcrumb trail is "Api Root / Hero List / Hero Instance". The main heading is "Hero Instance". To the right of the heading are three buttons: "DELETE" (red), "OPTIONS" (blue), and "GET" (blue with a dropdown arrow). Below the heading, the HTTP method and path are shown as "GET /heroes/1/". The response details are listed: "HTTP 200 OK", "Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS", "Content-Type: application/json", and "Vary: Accept". The response body is a JSON object:

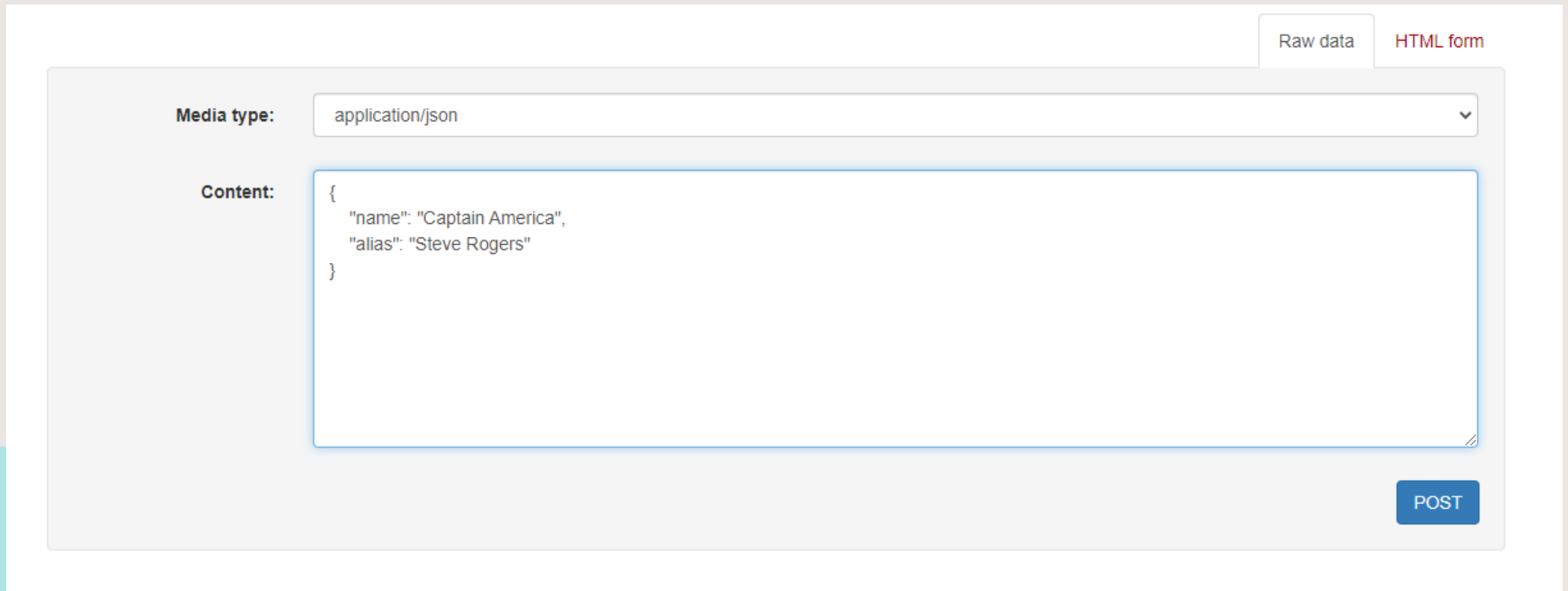
```
{  "name": "Wonder Women",  "alias": "Diana Prince"}
```

. At the bottom, there are two tabs: "Raw data" (selected) and "HTML form". The "HTML form" tab shows two input fields: "Name" with the value "Wonder Women" and "Alias" with the value "Diana Prince". A "PUT" button is located at the bottom right of the form.

Test it out!

Send a POST request

Our API also handles post requests. We can send JSON to the API and it will create a new record in the database.



The screenshot shows a REST client interface with two tabs at the top: "Raw data" and "HTML form". The "Raw data" tab is active. Below the tabs, there is a "Media type:" label and a dropdown menu showing "application/json". To the left of a large text area is a "Content:" label. The text area contains a JSON object:

```
{  "name": "Captain America",  "alias": "Steve Rogers"} 
```

 In the bottom right corner of the interface is a blue button labeled "POST".

Quick REST APIs with Django REST Framework

I hope you enjoyed this quick tutorial. Of course, APIs can get much more complicated with multiple models intersecting and endpoints with more complex queries.

That said, you're well on your way with this post. REST Framework handles complexity fairly well.

Have fun building some new APIs!