Learning Objectives

- **Explain the importance of the `JsonResponse` and `HttpResponse` classes**
- **Transform a model into data that is serializable**
- **List all `Post` objects in a view**
- **List all `Post` details in a view**
- **Create URL paths for the new views**

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the `blango` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

First-Party Django REST API

Intro and Motivation

If you've read the title of this course you'll know that we'll be working with a third-party library named Django Rest Framework, or **DRF** for short. We'll look at that and start using it in the next module, but in order to reinforce the REST concepts we'll see what can be built using just built-in Django functionality. And, after doing so, you'll see why DRF is so useful.

What we will develop will be very primitive, you'll be able to list all `Post` objects, get the full detail for a single `Post` object, create a new `Post` object, update an existing `Post` object, and finally delete a `Post` object. So while it will handle all our use cases, it won't do much in the way of validation, error handling or even authentication.

Building this simple API will help cement the REST API concepts without having to learn all about Django Rest Framework. It will also give us an API to work with when using Postman, a program for testing HTTP APIs.

Django Response Classes

The Django Views that we will implement will mostly not be anything new, but we will make use of a few different `HttpResponse` classes so we can be specific about what the responses mean. You may not have used these before.

The first response class we'll introduce is `JsonResponse`. It automatically encodes whatever data it's been passed into a JSON string. It will also set the `Content-Type` header of the response to `application/json`. To return a JSON response from your view, you'd do something like this:

```
return JsonResponse({"pk": post.pk, "summary": post.summary})
```

By default, for security reasons, `JsonResponse` only allows dictionaries to be passed as the response body. It will raise a `TypeError` if other types are passed. You can override this behavior and allow serialization of anything by passing the parameter `safe=False`. For example:

```
return JsonResponse([1, 2, 3], safe=False)
```

This is to prevent against a vulnerability in earlier versions of Javascript in older browsers. You can read about the vulnerability here. This vulnerability has been known and fixed for a long time (FireFox fixed it in 2007, for example), and the Django documentation actually says that you should be fine to specify `safe=False` for newer browsers. Django, being secure by default, defaults to `safe=True` though. We'll use dictionaries for each `JsonResponse` we use so we don't have to worry about the `safe` parameter at all.

We'll also use the standard `HttpResponse` class, which will allow us to return empty responses with status codes to nominate what they mean. For example, the status code `204` means *No Content*, that is, the body of the response is expected to be empty. This indicates that the request was accepted but doesn't return any data. To make the code easier to read, rather than using numeric literals we'll use the `HTTPStatus` enum from the `http` module (a built-in Python library). We return a "204" response like this:

```
return HttpResponse(status=HTTPStatus.NO_CONTENT)
```

In the case of our API, we'll return this when a `Post` is updated (`PUT` request) or deleted (`DELETE` request).

The other special status code we'll use is `201` which means *Created*. We'll return this when a `Post` is created (`POST` request to posts list view). The response will also include the HTTP header `Location` which contains the URL of the newly created `Post`, so that the end user knows how to retrieve it. We'll return one like this:

```
return HttpResponse(
    status=HTTPStatus.CREATED,
    headers={"Location": reverse("api_post_detail", args=
        (post.pk,))},
)
```

The last response class we will use is `HttpResponseNotAllowed`, which sends a `405` status code. We'll return this if a request is made using a method that's not supported. For example, trying to `PUT` or `DELTE` a list of `Post` objects. This class should be passed a list of HTTP methods that *are* acceptable. We use it like this (in the `Post` list view):

```
return HttpResponseNotAllowed(["GET", "POST"])
```

CSRF Exemption

When you've worked with Django forms before, you know that you have to include the `{% csrf_token %}` template tag to add a CSRF token to the form, otherwise the request will be rejected.

▼ **CSRF**
CSRF stands for Cross Site Request Forgery, and is an attack that could occur with a malicious website adding a form to trigger undesired actions on a target website. To refresh yourself on how Django prevents this with the CSRF token, check the Cross Site Request Forgery protection documentation.

Django will reject `POST`, `PUT` and `DELETE` requests that don't include a CSRF token, unless the view is marked as not requiring one. This is done with the `django.views.decorators.csrf.csrf_exempt` decorator.

For example:

```
from django.views.decorators.csrf import csrf_exempt


@csrf_exempt
def post_list(request):
    …
```

Marking a view as `csrf_exempt` can be considered a security risk, but as we're specifically designing views for use with a REST API we would expect to have some other method of authenticating the user and requests, so CSRF protection is not really required.

Transforming `Post` Objects

Django models or `QuerySets` are not serializable by the `JsonResponse` view. In order for it to work, we'll need to manually extract the fields we want into a dictionary, which *can* be serialized. It's quite a simple function:

```
def post_to_dict(post):
    return {
        "pk": post.pk,
        "author_id": post.author_id,
        "created_at": post.created_at,
        "modified_at": post.modified_at,
        "published_at": post.published_at,
        "title": post.title,
        "slug": post.slug,
        "summary": post.summary,
        "content": post.content,
    }
```

When you see `post_to_dict()` used in the view code, it will refer to this implementation above.

Views

The Views 1: Post List View

We'll include the full implementation of the views below, but first we'll discuss their operation. The first view is to list existing `Post` objects or create a new one, called `post_list`. It will respond to `GET` or `POST` requests.

For a `GET`, it fetches all `Post` objects, transforms them to dictionaries with `post_to_dict` and then returns a `JsonResponse`:

```
posts = Post.objects.all()
posts_as_dict = [post_to_dict(p) for p in posts]
return JsonResponse({"data": posts_as_dict})
```

Since `JsonResponse` only allows dictionaries (unless we set `safe=False`) we'll wrap the `posts_as_dict` list in a dictionary, under the `data` key. This is actually similar to the structure that DRF will use anyway.

For a `POST` request, a new `Post` is created using values from the request body (JSON encoded dictionary).

```
post_data = json.loads(request.body)
post = Post.objects.create(**post_data)
return HttpResponse(
    status=HTTPStatus.CREATED,
    headers={"Location": reverse("api_post_detail", args=
        (post.pk,))},
)
```

If it's neither a `GET`, nor a `POST`, we'll return a `HttpResponseNotAllowed`, and include the allowed methods.

```
return HttpResponseNotAllowed(["GET", "POST"])
```

Next the other `Post` view.

The Views 2: Post Detail View

The other view we'll implement is `post_detail`. It will fetch a `Post` object using its primary key. Then, for a `GET` it will just encode it as a dictionary and return it with a `JsonResponse`:

```
return JsonResponse(post_to_dict(post))
```

For a `PUT`, we'll set the `Post` object's attributes using each of the values that were passed in the request body.

```
post_data = json.loads(request.body)
for field, value in post_data.items():
    setattr(post, field, value)
post.save()
return HttpResponse(status=HTTPStatus.NO_CONTENT)
```

We'll return a *204 No Content* response, which indicates that the change was successful. The client will already have the `Post` objects URL and so can choose to request the updated data again if it needs to; we don't need to pass the `Location` header.

Finally, for a `DELETE` request, we can call the `Post` object's `delete()` method, and return another *204 No Content* to indicate success.

```
post.delete()
return HttpResponse(status=HTTPStatus.NO_CONTENT)
```

Like with `post_list`, we default to returning `HttpResponseNotAllowed` if we get a request using a different method.

Now, you'll see the view code in its entirety and you can implement it in your version of Blango.

Try It Out

Try It Out

Before building our API code, let's do a tiny bit of housekeeping. The `slug` field on `Post` should be unique. Since we've been querying by this field in our `post_detail` view then it should probably have been unique from the start of the project. It hasn't really been an issue though since we're just been creating `Post` objects through the Django admin as quite a manual process. Going ahead though, when we start creating them through the API, we might want to repeat requests as we're debugging, and don't want a bunch of `Post` objects with the same slug. So the first thing we'll do is add the `unique=True` argument to the `SlugField` on the `Post` model, like this:

```
slug = models.SlugField(unique=True)
```

Then run the `makemigrations` and `migrate` management commands to apply the change to the database.

▼ **Multiple Post objects**
Be aware that if you have multiple `Post` objects in your database that share a slug, you'll need to make them all unique before running `migrate`.

We'll create the views in a file called `api_views.py` inside the `blog` application, to keep them separate from our normal views. Create this file now.

Open api_views.py

Now, copy and paste this code into it:

```
import json
from http import HTTPStatus

from django.http import JsonResponse, HttpResponse,
        HttpResponseNotAllowed
from django.shortcuts import get_object_or_404
from django.urls import reverse
from django.views.decorators.csrf import csrf_exempt

from blog.models import Post
```

```python
def post_to_dict(post):
    return {
        "pk": post.pk,
        "author_id": post.author_id,
        "created_at": post.created_at,
        "modified_at": post.modified_at,
        "published_at": post.published_at,
        "title": post.title,
        "slug": post.slug,
        "summary": post.summary,
        "content": post.content,
    }


@csrf_exempt
def post_list(request):
    if request.method == "GET":
        posts = Post.objects.all()
        posts_as_dict = [post_to_dict(p) for p in posts]
        return JsonResponse({"data": posts_as_dict})
    elif request.method == "POST":
        post_data = json.loads(request.body)
        post = Post.objects.create(**post_data)
        return HttpResponse(
            status=HTTPStatus.CREATED,
            headers={"Location": reverse("api_post_detail",
        args=(post.pk,))},
        )

    return HttpResponseNotAllowed(["GET", "POST"])


@csrf_exempt
def post_detail(request, pk):
    post = get_object_or_404(Post, pk=pk)

    if request.method == "GET":
        return JsonResponse(post_to_dict(post))
    elif request.method == "PUT":
        post_data = json.loads(request.body)
        for field, value in post_data.items():
            setattr(post, field, value)
        post.save()
        return HttpResponse(status=HTTPStatus.NO_CONTENT)
    elif request.method == "DELETE":
        post.delete()
        return HttpResponse(status=HTTPStatus.NO_CONTENT)
```

```
    return HttpResponseNotAllowed(["GET", "PUT", "DELETE"])
```

We've explained the behavior of most of this already, all that's new is how we're branching the code based on the HTTP method from `request.method`.

Next we'll set up the URL routes. To keep our main `urls.py` file from getting too large, we'll create a new routing file also in the `blog` application; call it `api_urls.py`. Add this add as the content:

Open api_urls.py

```
from django.urls import path

from blog.api_views import post_list, post_detail

urlpatterns = [
    path("posts/", post_list, name="api_post_list"),
    path("posts/<int:pk>/", post_detail,
        name="api_post_detail"),
]
```

Finally we'll add route API requests to this file. Open the main `urls.py` and add this to `urlpatterns`:

Open urls.py

```
path("api/v1/", include("blog.api_urls")),
```

▼ **Versioning**

Note that we're versioning the URL (by adding the `v1` path component). This will allow us to implement changes to the API without breaking backwards-compatibility with older clients. While we won't be using it, Django Rest Framework has support for versioning which allows you to reuse the same view for different versions and alter the view's response based on a special version attribute that's available. Read the DRF versioning guide if you do end up having multiple versions of your real API.

Now start the Django development server if it's not already running, and navigate to the `Post` list API view, the path is `/api/v1/posts/`. You should see a list of `Post` objects in JSON format. The other request you can easily test in a browser is a `Post` detail `GET`. Try viewing something like `/api/v1/posts/1/`. You should see a single `Post` object in JSON format.

View Blog

That's about the limit of what we can easily test with a web browser. To make requests other than `GET` we will use the tool Postman. In the next section we'll download and set up Postman and use it to test our basic API.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish First-Party Django REST API"
```

- Push to GitHub:

```
git push
```