Learning Objectives

- **Decorate a view function to become an API method**
- **Load the API in the DRF GUI**
- **Add ability to look at the JSON of API objects**
- **Contrast a class-based view with a function-based view**
- **Explain how generic views reduce the amount of code**
- **Create generic views for the `PostList` and `PostDetail` classes**

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.

View Functions and Response Objects

View functions and Response objects

Because most REST APIs follow the same pattern (`PUT` means update, `POST` means create, etc), Django Rest Framework provides a lot of helper classes to let you write less code.

We'll work out way towards them, but we're going to start at a lower level. DRF also provides a decorator to mark view functions as API methods. It also gives us HTTP response classes that have more options for rendering our serialized data.

The decorator `rest_framework.decorators.api_view` can be applied to view functions, and does a few helpful things for us.

- It automatically rejects unsupported methods by returning a response with a `405` status code. We no longer have to return `HttpResponseNotAllowed` ourselves.
- It parses the request data, including handling errors. Not only do we not have to parse JSON ourselves but this means other content types can be handled too.
- It handles the DRF `Response` object, returning different responses based on what the client wants.

Adding it to a view function is easy. Just import it and decorate the function, passing in a list of acceptable HTTP methods. Here it is on our Blango API views.

```
from rest_framework.decorators import api_view


@api_view(["GET", "POST"])
def post_list(request):
    ...

@api_view(["GET", "PUT", "DELETE"])
def post_detail(request, pk):
    ...
```

The decorator also adds another attribute to the request that the view functions receive: `data`. This is deserialized data ready to be passed to the serializer. We can use it in our create and update functions. For example, to update a `Post`:

```
elif request.method == "PUT":
    serializer = PostSerializer(post, data=request.data)
```

This means that we no longer have to manually perform JSON deserialization – the decorator takes care of this for us, including parsing or generating other formats for which we've added support – we'll come back to this later.

The last change we can make to utilize the `api_view` decorator is to start using `rest_framework.response.Response` objects instead of `JsonResponse`. For example

```
return Response(PostSerializer(post).data)
```

or

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

We can use this to replace all the different types of Django response classes we've been using. However this means we should also replace the use of the `get_object_or_404` shortcut and return a `Response` with status code `404` instead:

```
try:
    post = Post.objects.get(pk=pk)
except Post.DoesNotExist:
    return Response(status=HTTPStatus.NOT_FOUND)
```

We'll see why we should make this change soon.

Try it out

We can make all these changes to our Blango API methods. Since this is going to be temporary, we won't go through the changes in much detail. Instead, you can simply replace the entire content of `api_views.py` with this:

```
from http import HTTPStatus

from django.urls import reverse
from rest_framework.decorators import api_view
from rest_framework.response import Response

from blog.api.serializers import PostSerializer
from blog.models import Post


@api_view(["GET", "POST"])
def post_list(request):
    if request.method == "GET":
        posts = Post.objects.all()
        return Response({"data": PostSerializer(posts,
        many=True).data})
    elif request.method == "POST":
        serializer = PostSerializer(data=request.data)
        if serializer.is_valid():
            post = serializer.save()
            return Response(
                status=HTTPStatus.CREATED,
                headers={"Location": reverse("api_post_detail",
        args=(post.pk,))},
            )
        return Response(serializer.errors,
        status=HTTPStatus.BAD_REQUEST)


@api_view(["GET", "PUT", "DELETE"])
def post_detail(request, pk):
    try:
        post = Post.objects.get(pk=pk)
    except Post.DoesNotExist:
        return Response(status=HTTPStatus.NOT_FOUND)

    if request.method == "GET":
        return Response(PostSerializer(post).data)
    elif request.method == "PUT":
        serializer = PostSerializer(post, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(status=HTTPStatus.NO_CONTENT)
        return Response(serializer.errors,
        status=HTTPStatus.BAD_REQUEST)
    elif request.method == "DELETE":
        post.delete()
        return Response(status=HTTPStatus.NO_CONTENT)
```

Note that we've changed `serializer.is_valid()` to not raise an exception on failure. Instead, if it returns false, we'll return a `Response` containing an error dictionary.

▼ **Status Module**

Another module to point out is `rest_framework.status`, which contains HTTP status codes. So, instead of using `HTTPStatus.NO_CONTENT` we could change to `status.HTTP_204_NO_CONTENT`. `rest_framework.status` attributes can be easier to understand at a glance as they contain the status code as well as the name. Apart from that, they're no different, so we leave them as is.

Now let's try out our API. Most of it hasn't changed. If you try to `POST` or `PUT` a `Post` object with errors, you'll get a detailed error response back. For example, if we remove the `summary` field:

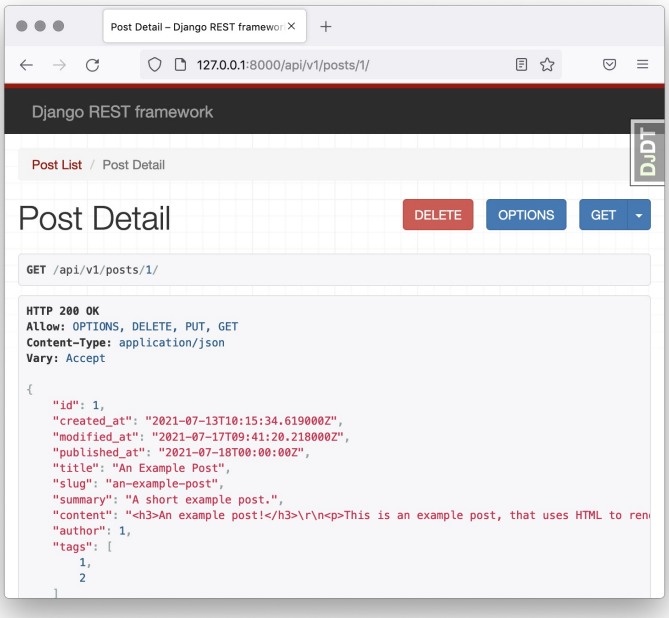
View Blog

```
{
    "summary": [
        "This field is required."
    ]
}
```

Or, if we make a `PUT` request to a view that doesn't support it:

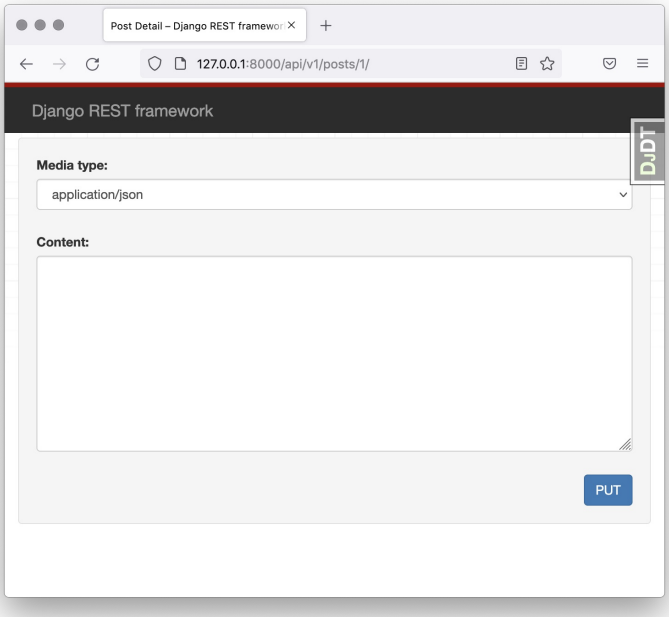
```
{
    "detail": "Method \"PUT\" not allowed."
}
```

The biggest change though, is if you load up the API in a web browser. Django Rest Framework provides its own browser-based GUI to let you work with the API. Here's what the `Post` detail view looks like, for example.



post detail

While it doesn't offer as many features as a tool like Postman, it makes it easy for anyone to be able to quickly browse our API, and make requests that a browser normally can't – like `DELETE`. If you scroll down the page, you'll also see a form for `PUT`ing JSON.



json put

The DRF web GUI is able to understand the `api_view` decorator and only show the form because the view allows `PUT` requests. If you load the `Post`

list URL, you'll notice there's no `DELETE` button at the top because it's not supported.

We'll now look at a small shortcut that DRF provides to make it easier to request different content.

Optional Format Suffixes

Optional format suffixes

If you tested the API with both Postman and a browser, you'll notice that you get different content (a webpage vs. JSON). This is done by DRF examining the request `Accept` header, the header that specifies what type of content it will accept. But DRF allows us to make a couple of changes to be able to control the content type using a file extension. This means we could request JSON data for a `Post` with a URL like `http://127.0.0.1:8000/api/v1/posts/5.json`, even from a browser.

The function that enables this is `rest_framework.urlpatterns.format_suffix_patterns`. It is passed a list of URL patterns, and adds extra patterns with a suffix. We would use it like this in Blango (in `api_urls.py`):

```
from rest_framework.urlpatterns import format_suffix_patterns

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


urlpatterns = format_suffix_patterns(urlpatterns)
```

Notice that the trailing `/` on the detail URL has been removed – otherwise we'd end up having URLs like `/api/v1/posts/5/.json`.

Then we need to add an extra argument to each of our views: `format`. This can default to `None`. We don't have to actually do anything for it, the `api_view` decorator takes care of it, but it must be present.

For Blango, our view function signatures change to:

Open api_views.py

```
def post_list(request, format=None):
```

and

```
def post_detail(request, pk, format=None):
```

Try it out

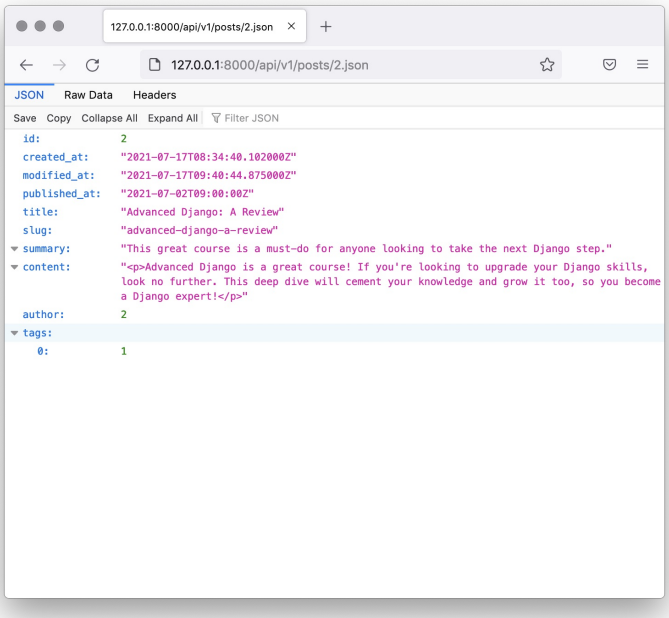
Make the above change to your own `api_urls.py` (add the import, wrap `urlpatterns` and remove the trailing `/` on the second URL pattern).

```
from rest_framework.urlpatterns import format_suffix_patterns

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


urlpatterns = format_suffix_patterns(urlpatterns)
```

Then, add the `format=None` parameter to each API view in `api_views.py`.

Now you can try it out in a browser. If you visit, say, `/api/v1/posts/2` you'll get the DRF web GUI. But, if you visit `/api/v1/posts/2.json`, you'll get a JSON response. **Note,** Codio's browser does not format JSON like the screenshot. However, the structure should be the same.

View Blog



json in browser

Next up we'll look at how we can further simplify the code with class-based views.

APIView

APIView

The `rest_framework.views.APIView` class is a base class you can inherit from to define API views as class-based views. Using it doesn't offer many advantages over the function views we've already seen, but you might like to use it if you prefer class-based views. We'll see where class-based views have genuine advantages in the next sub-section.

For now, we won't dive deeply into the `APIView`, instead we'll just show how to use it to replace our Blango `Post` detail view. You don't have to try this out, as we'll be moving on to generic views soon.

```
class PostDetail(APIView):
    @staticmethod
    def get_post(pk):
        return get_object_or_404(Post, pk=pk)

    def get(self, request, pk, format=None):
        post = self.get_post(pk)
        return Response(PostSerializer(post).data)

    def put(self, request, pk, format=None):
        post = self.get_post(pk)
        serializer = PostSerializer(post, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(status=HTTPStatus.NO_CONTENT)
        return Response(serializer.errors,
        status=HTTPStatus.BAD_REQUEST)

    def delete(self, request, pk, format=None):
        post = self.get_post(pk)
        post.delete()
        return Response(status=HTTPStatus.NO_CONTENT)
```

Essentially, instead of branching inside the view function, we "branch" by having dedicated methods that are called for each request method type (just like a regular Django class-based view). But, by inheriting from `APIView`, we'll also have the same benefits as afforded to us by the `api_view` decorator.

But, where we really start to reduce the amount of code we need to write, is with DRF's generic views.

Generic Views and Mixins

Generic views and mixins

DRF uses the mixin pattern to add functionality to generic views. If you've never used mixins before, it means building your class and inheriting from multiple parent classes. Each parent class might provide just a single method that's useful to our class. Let's look at this in the context of our Blango API `Post` list view, and then afterward we'll explain what's taking place.

```
from rest_framework import mixins
from rest_framework import generics


class PostList(mixins.ListModelMixin, mixins.CreateModelMixin,
        generics.GenericAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer

    def get(self, request, *args, **kwargs):
        return self.list(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        return self.create(request, *args, **kwargs)
```

The "main" parent class is `generics.GenericAPIView`, which has defaults assigned for attributes that its subclass methods might refer to, and we haven't defined on our class. The mixin classes are `mixins.ListModelMixin` and `mixins.CreateModelMixin`. How do these work?

We still need to define `get()` and `post()` methods to respond the `GET` and `POST` HTTP requests, respectively. However, we've reduced the amount of code in the method bodies to a single line.

In `get()`, we're calling the `list()` method, which is provided by the `ListModelMixin` class. It executes the `QuerySet` that's in the `queryset` attribute (remember, `QuerySet` objects are evaluated lazily, so this won't actually hit the database until we need it). It then passes the results of the `QuerySet` to the serializer defined by the `serializer_class` attribute, then creates a `Response` and returns it.

The `post()` method just calls the `create()` method, from `CreateModelMixin`. This will pass the `data` from the `request` into the serializer (again, from `serializer_class`), then call its `save()` method which will create the `Post`.

Here's what our `post_detail` function would change to:

```
class PostDetail(
    mixins.RetrieveModelMixin,
    mixins.UpdateModelMixin,
    mixins.DestroyModelMixin,
    generics.GenericAPIView,
):
    queryset = Post.objects.all()
    serializer_class = PostSerializer

    def get(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)

    def put(self, request, *args, **kwargs):
        return self.update(request, *args, **kwargs)

    def delete(self, request, *args, **kwargs):
        return self.destroy(request, *args, **kwargs)
```

`retrieve()` is provided by `RetrieveModelMixin`, `update()` from `UpdateModelMixin` and `destroy()` by `DestroyModelMixin`. To fetch a single `Post`, the `queryset` will be filtered based on the `**kwargs` provided. Since we're passing in `pk` from the URL pattern, the `Post` object will be loaded by filtering on `pk`.

You can see how this reduces the amount of code we need to write significantly, thanks to the fact that REST follows a pattern that DRF provides.

But, we won't bother implementing these views in Blango, because we can make our views even shorter.

Advanced Generic Views

Advanced generic views

Since these mixins are used together so often, DRF provides base classes that combine them. We can make our `PostList` view even shorter by just inheriting from `generics.ListCreateAPIView`. This view includes all the HTTP methods already implemented. So, the entirety of our `PostList` view can be converted to:

```
class PostList(generics.ListCreateAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
```

That's it. The `get()` method that calls `list()` is defined on the `ListCreateAPIView` itself, so we don't even need to define that.

Similarly, we can use the `generics.RetrieveUpdateDestroyAPIView` base view to shorten our `PostDetail` view, to this:

```
class PostDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
```

Again, there's no need to implement `get()`, `put()` or `delete()` methods as they're already provided by `RetrieveUpdateDestroyAPIView`.

▼ **More information**
More information on Generic Views is available at the official documentation page.

Let's finally implement class-based views in Blango to replace our view functions.

Try it out

We'll also consolidate all our API related code into the `api` directory inside the `blog` app. Create a new file called `views.py` inside this directory, and implement the class-based views inside. When you're done the content of this file should be like this:

Open api/views.py

```
from rest_framework import generics

from blog.api.serializers import PostSerializer
from blog.models import Post


class PostList(generics.ListCreateAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer


class PostDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
```

Next, create `urls.py` also inside the `api` directory. You can put this content inside:

Open api/urls.py

```
from django.urls import path
from rest_framework.urlpatterns import format_suffix_patterns

from blog.api.views import PostList, PostDetail

urlpatterns = [
    path("posts/", PostList.as_view(), name="api_post_list"),
    path("posts/<int:pk>", PostDetail.as_view(),
        name="api_post_detail"),
]

urlpatterns = format_suffix_patterns(urlpatterns)
```

Open the main `urls.py` file (inside the `blango` directory) and change the `api/v1/` rule to include your new `urls.py` file instead:

Open urls.py

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

Finally, we can delete the faithful files we've been using before: `blog/api_views.py` and `blog/api_urls.py`.

You can now test out the new implementation of the API. You should notice that it mostly behaves as our previous implementation. One difference is that when we `POST` a new `Post`, a full `Post` object is returned, instead of a `201` response with a `Location` header. As we mentioned in the REST introduction, you could choose to implement object creation in either way, and DRF chooses to return the full object. We can then look at the `id` of the object to determine its URL. Both responses are valid.

View Blog

If you try hitting the API endpoints in a browser, you should notice you'll also be able to `POST` data using a form that DRF generates, instead of having to write out JSON. Because we're using these generic views, DRF is able to more closely relate the view and model (via the serializer) and generate a form for you.

That's all we're going to cover for Django Rest Framework views. In the next module, we'll look at improving our API with authentication.

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 DRF views"
```

- Push to GitHub:

```
git push
```