Learning Objectives

- Apply the `cach_page`, `vary_on_headers`, and `vary_on_cookie` decorators to API views
- Wrap API view caching decorators with `@method_decorator`
- Vary on both headers and cookies to account for the various ways to authenticate with the API
- Add caching to generic views and viewsets

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 Caching

View Caching

In course two, module three, we looked at the caching framework in Django. As part of that, we discussed some view decorators:

- `cache_page()`: caches the response from a view for a certain number of seconds.
- `vary_on_headers()`: lets Django know that the response from the view can change based on headers, and so the cache key should take the given header into account.
- `vary_on_cookie`: a shortcut for `vary_on_headers("Cookie")`.

The good news is that these decorators can be applied to Django Rest Framework views and they'll work just the same as they do on regular Django views.

Applying them to class-based views is just a little different though. View decorators can't be applied directly to view methods – this applies to normal Django classed-based views too, not just DRF views.

Decorator functions must first be wrapped in the `django.utils.decorators.method_decorator` function. For the sake of a simple example, let's pretend we've written an API view for listing `Comment` objects, without making use of DRF's generic views. If we tried to cache the `get()` method's responses like this, it wouldn't work:

```
from django.views.decorators.cache import cache_page


class CommentListView(APIView):
    @cache_page(60)
    def get(self, request):
        comments = Comment.objects.all()
        serializer = CommentSerializer(
            comments, many=True, context={"request": request}
        )
        return Response(serializer.data)
```

Instead, we wrap the decorator in `method_decorator()`:

```
from django.utils.decorators import method_decorator
from django.views.decorators.cache import cache_page


class CommentListView(APIView):
    @method_decorator(cache_page(60))
    def get(self, request):
        comments = Comment.objects.all()
        serializer = CommentSerializer(
            comments, many=True, context={"request": request}
        )
        return Response(serializer.data)
```

▼ **CommentListView**

Aside: don’t implement the `CommentListView`, we’re putting it here just as an example.

This applies to the other decorators too. Let’s create a new action method on `PostViewSet` to demonstrate this. We’ll call it `mine()` (so its URL will be `/api/v1/posts/mine`). It will list only `Post` objects for which the current user is the `author`.

```
class PostViewSet(viewsets.ModelViewSet):
    # existing attributes omitted

    @action(methods=["get"], detail=False, name="Posts by the
        logged in user")
    def mine(self, request):
        if request.user.is_anonymous:
            raise PermissionDenied("You must be logged in to see
        which Posts are yours")
        posts = self.get_queryset().filter(author=request.user)
        serializer = PostSerializer(posts, many=True, context=
        {"request": request})
        return Response(serializer.data)
```

Now we want to cache the response from this view for five minutes, but the view’s response will differ based on the user. Remember with DRF there are two ways of knowing who the user is (at least, based on our configuration). Either the user will be identified with a cookie, if they’re using session authentication, or with the `Authorization` HTTP header, if they’re using token or basic authentication. This means the caching of the response needs to take into account both the `Cookie` and `Authorization` HTTP headers.

To do this, we'll use the `vary_on_headers("Authorization")` **and** `vary_on_cookie` decorators. Again, both wrapped in `method_decorator`. It would be implemented like this:

```
from django.utils.decorators import method_decorator
from django.views.decorators.cache import cache_page
from django.views.decorators.vary import vary_on_headers,
        vary_on_cookie


class PostViewSet(viewsets.ModelViewSet):
    permission_classes = [AuthorModifyOrReadOnly |
        IsAdminUserForObject]
    queryset = Post.objects.all()

    def get_serializer_class(self):
        if self.action in ("list", "create"):
            return PostSerializer
        return PostDetailSerializer

    @method_decorator(cache_page(300))
    @method_decorator(vary_on_headers("Authorization"))
    @method_decorator(vary_on_cookie)
    @action(methods=["get"], detail=False, name="Posts by the
        logged in user")
    def mine(self, request):
        if request.user.is_anonymous:
            raise PermissionDenied("You must be logged in to see
        which Posts are yours")
        posts = self.get_queryset().filter(author=request.user)
        serializer = PostSerializer(posts, many=True, context=
        {"request": request})
        return Response(serializer.data)
```

▼ Simplify the code

We could actually simplify this a little. Since `vary_on_headers()` accepts multiple header names as arguments we could do `@method_decorator(vary_on_headers("Authorization", "Cookie"))` instead of using the `vary_on_cookie` decorator. `vary_on_cookie` may be a little more explicit in its usage, the decision is up to you.

Caching Generic Views and Viewsets

Caching Generic Views and Viewsets

We've seen how to cache non-generic APIViews, and how to cache action methods on viewsets. But how can we cache the generic APIViews or viewsets when we don't implement the methods being called?

You will have to implement whichever method you want to cache, and have it just behave as a "pass-through" to the super class's method. For example, in our UserDetail view, we're inheriting from RetrieveAPIView which implements the get method. So, we can just implement this method, and add caching to it (let's say, five minutes):

```
class UserDetail(generics.RetrieveAPIView):
    # existing methods omitted

    @method_decorator(cache_page(300))
    def get(self, *args, **kwargs):
        return super(UserDetail, self).get(*args, *kwargs)
```

Adding caching to viewsets is similar, except the decorator(s) need to be added to the built-in action methods. Remember from the last course, these methods are: `list()`, `create()`, `retrieve()`, `update()`, `partial_update()` and `destroy()`.

Unless you had a very good reason, you wouldn't want to add caching to the methods that alter data, because then they wouldn't do anything. That means to add caching to your viewset, you just need to implement `list()` and `retrieve()`. They can be added as pass-through methods that just call the super class.

Let's see how to do this on the viewsets in Blango (`PostViewSet` and `TagViewSet`). We want the following caching rules:

- The list of `Posts` should be cached for two minutes, however when fetching a `Post` detail we should get the latest data from the database.
- We don't expect `Tag` objects to change very often, so we will cache both the list and detail views for five minutes.

Here's the new `list()` method on `PostViewSet`:

```
class PostViewSet(viewsets.ModelViewSet):
    # existing methods omitted

    @method_decorator(cache_page(120))
    def list(self, *args, **kwargs):
        return super(PostViewSet, self).list(*args, **kwargs)
```

And here are the two new methods, `list()` and `retrieve()` on `TagViewSet`:

```
class TagViewSet(viewsets.ModelViewSet):
    # exiting methods omitted

    @method_decorator(cache_page(300))
    def list(self, *args, **kwargs):
        return super(TagViewSet, self).list(*args, **kwargs)

    @method_decorator(cache_page(300))
    def retrieve(self, request, *args, **kwargs):
        return super(TagViewSet, self).retrieve(*args, **kwargs)
```

Now you can implement these new methods and caches in Blango.

Try It Out

Try It Out

You'll be making changes just to the `blog/api/views.py` file. Open it up, then add these imports:

```
from django.utils.decorators import method_decorator
from django.views.decorators.cache import cache_page
from django.views.decorators.vary import vary_on_headers,
        vary_on_cookie

from rest_framework.exceptions import PermissionDenied
```

Next, the `mine()` method on `PostViewSet`. It's decorated with the `action()` function so that DRF adds a URL to point to it. We're setting `detail` to `False` so that the route is added to the list endpoint (`/posts/mine`) instead of to a detail endpoint (`/posts/<pk>/mine`).

We also add the caching and vary decorators, so that the response is cached individually for each user, whether they access by session authentication (`Cookie` header) or `Authorization` header.

Add this method and decorators to `PostViewSet`:

```
    @method_decorator(cache_page(300))
    @method_decorator(vary_on_headers("Authorization"))
    @method_decorator(vary_on_cookie)
    @action(methods=["get"], detail=False, name="Posts by the
        logged in user")
    def mine(self, request):
        if request.user.is_anonymous:
            raise PermissionDenied("You must be logged in to see
        which Posts are yours")
        posts = self.get_queryset().filter(author=request.user)
        serializer = PostSerializer(posts, many=True, context=
        {"request": request})
        return Response(serializer.data)
```

We also want to cache the list of `Posts` for two minutes, which means overriding the `list()` view. Implement this `list()` view on `PostViewSet`:

```
    @method_decorator(cache_page(120))
    def list(self, *args, **kwargs):
        return super(PostViewSet, self).list(*args, **kwargs)
```

Now on to `UserDetail`. Since this is a view, and not a viewset, we want to override and cache on the view methods. In our case, just `get()`. Add this to `UserDetail`:

```
    @method_decorator(cache_page(300))
    def get(self, *args, **kwargs):
        return super(UserDetail, self).get(*args, *kwargs)
```

Finally, `TagViewSet`. We'll add caching to both the `list()` and `retrieve()` methods. Add both these methods to `TagViewSet`:

```
    @method_decorator(cache_page(300))
    def list(self, *args, **kwargs):
        return super(TagViewSet, self).list(*args, **kwargs)

    @method_decorator(cache_page(300))
    def retrieve(self, *args, **kwargs):
        return super(TagViewSet, self).retrieve(*args, **kwargs)
```

Now start the Django Dev Server and try it out, but note that the DRF GUI based responses aren't cached. If you create and retrieve an object using the browser, you'll see your change straight away. To test if caching is working, you'll need to make a request using Postman. Try fetching the list of `Posts`, then making a change or creating a new one (you can make the new `Post` through the DRF GUI if you prefer). Then fetch the list of `Posts` again (using Postman). It should not include your changes. Wait two minutes, then fetch again, and you'll see the changes.

View Blog

In the next section, we're going to look at applying throttling to Django Rest Framework requests.

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 caching"
```

- Push to GitHub:

```
git push
```