# def form\_valid(self, form): ...

In the view, ensure that you don’t include created\_by in the list of fields to edit, and override [form\_valid()](https://docs.djangoproject.com/en/2.1/ref/class-based-views/mixins-editing/#django.views.generic.edit.ModelFormMixin.form_valid) to add the user:

from django.views.generic.edit import CreateView

from myapp.models import Author

class AuthorCreate(CreateView):

model = Author

fields = ['name']

def form\_valid(self, form):

form.instance.created\_by = self.request.user

return super().form\_valid(form)

Compare the following two classes: they are identical

In these particular examples, we had to override the form\_valid() method so as to set some extra fields such as the updated\_by and updated\_at .

class PostUpdateView(UpdateView):

...

success\_url = '/'

def form\_valid(self, form):

form.instance.updated\_by = self.request.user

form.instance.updated\_at = timezone.now()

return super(PostUpdateView, self).form\_valid(form)

[In second example](https://stackoverflow.com/questions/56029008/when-i-need-to-remove-superpostupdateview-self-form-validform-from-form), I am already saving the object and returning a redirect response, calling super(PostUpdateView, self).form\_valid(form) is just repeating the code I have already written. It is fine to remove it.

# 1st Example

class PostUpdateView(UpdateView):

...

def form\_valid(self, form):

post = form.save(commit=False)

post.updated\_by = self.request.user

post.updated\_at = timezone.now()

post.save()

return redirect('board:view\_topic',

pk=post.topic.board.pk, topic\_pk=post.topic.pk)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# 2nd Example

def form\_valid(self, form):

form.instance.author = self.request.user

return super().form\_valid(form)

# get\_object()

get\_object() is actually a method of django.views.generic.detail.SingleObjectMixin

What is mixin?

In object-oriented programming languages, a **mixin** is a class that contains methods for use by other classes without having to be the parent class of those other classes.

get\_object() is the method that gets the object for which the DetailView is meant for.

Essentially what the method does is it search the \*\*kwargs passed into the method for a 'slug' or a 'pk'. If it is able to find either, it basically does a Model.objects.get(slug=slug) (or pk=pk), if neither is found it throws an error.

get\_object(queryset=None)

Returns the single object that this view will display. If queryset is provided, that queryset will be used as the source of objects; otherwise, [get\_queryset()](https://docs.djangoproject.com/en/2.2/ref/class-based-views/mixins-single-object/#django.views.generic.detail.SingleObjectMixin.get_queryset) will be used. get\_object() looks for a [pk\_url\_kwarg](https://docs.djangoproject.com/en/2.2/ref/class-based-views/mixins-single-object/#django.views.generic.detail.SingleObjectMixin.pk_url_kwarg) argument in the arguments to the view; if this argument is found, this method performs a primary-key based lookup using that value. If this argument is not found, it looks for a [slug\_url\_kwarg](https://docs.djangoproject.com/en/2.2/ref/class-based-views/mixins-single-object/#django.views.generic.detail.SingleObjectMixin.slug_url_kwarg) argument, and performs a slug lookup using the [slug\_field](https://docs.djangoproject.com/en/2.2/ref/class-based-views/mixins-single-object/#django.views.generic.detail.SingleObjectMixin.slug_field).

When [query\_pk\_and\_slug](https://docs.djangoproject.com/en/2.2/ref/class-based-views/mixins-single-object/#django.views.generic.detail.SingleObjectMixin.query_pk_and_slug) is True, get\_object() will perform its lookup using both the primary key and the slug.

# 1st Example

class PostDetailView(DetailView):

...

    def get\_object(self):

        slug = self.kwargs.get('title\_slug')

        return get\_object\_or\_404(BlogPost, slug=slug)

# 2nd Example

class ArticleDetailView(DetailView):

...

# overriding the primary function to replace `pk` with `id`

def get\_object(self):

# you access the URL args with `self.args` and `self.kwargs` in views

id\_ = self.kwargs.get('id')

return get\_object\_or\_404(Article, id=id\_)

# get\_queryset()

Used by ListViews - it determines the **list of objects** that **you want to display**. By default it will just give you all for the model you specify. By overriding this method you decide what list of objects to show.

# 1st Example

class UserPostListView(ListView):

    # model = Post

    queryset = Post.objects.all()

    # look below function this is overridden

    paginate\_by = 5

    # If we don't filter it, it will show posts of all user.

    def get\_queryset(self):

         user = get\_object\_or\_404(User, username=self.kwargs.get('username'))

         return Post.objects.filter(author=user).order\_by('-date\_posted')

# 2nd Example

class PostListView(ListView):

model = Post

context\_object\_name = 'posts'

template\_name = 'topic\_posts.html'

paginate\_by = 2

def get\_queryset(self):

self.topic = get\_object\_or\_404(Topic,

board\_\_pk=self.kwargs.get('pk'),

pk=self.kwargs.get('topic\_pk'))

queryset = self.topic.posts.order\_by('created\_at')

return queryset

def get\_context\_data(self, \*\*kwargs):

self.topic.views += 1

self.topic.save()

kwargs['topic'] = self.topic

return super().get\_context\_data(\*\*kwargs)

# 2nd Example

class FilteredAuthorView(ListView):

template\_name = 'authors.html'

model = Author

def get\_queryset(self):

qs = super().get\_queryset() # original qs

# filter by a variable captured from url, for example

return qs.filter(name\_\_startswith=self.kwargs['name'])

# ****get\_context\_data()****

get\_context\_data is how we add stuff ***to the*** request context when extending a GCBV.

# 1st Example

class PostListView(ListView):

model = Topic

context\_object\_name = 'posts'

template\_name = 'board/view\_topic.html'

paginate\_by = 3

# board/<int:pk>/topic/<int:topic\_pk>/

def get\_queryset(self):

self.topic = get\_object\_or\_404(Topic,

board\_\_pk=self.kwargs.get('pk'),

pk=self.kwargs.get('topic\_pk'))

queryset = self.topic.posts.order\_by('created\_at')

return queryset

def get\_context\_data(self, \*\*kwargs):

self.topic.views += 1

self.topic.save()

kwargs['topic'] = self.topic

return super().get\_context\_data(\*\*kwargs)

# 2nd Example

class PostListView(ListView):

form = EmailSignupForm()

model = Post

template\_name = 'blog.html'

context\_object\_name = 'queryset'

paginate\_by = 1

def get\_context\_data(self, \*\*kwargs):

category\_count = get\_category\_count()

most\_recent = Post.objects.order\_by('-timestamp')[:3]

context = super().get\_context\_data(\*\*kwargs)

context['most\_recent'] = most\_recent

context['page\_request\_var'] = "page"

context['category\_count'] = category\_count

context['form'] = self.form

return context

# def save(self, \*args, \*\*kwargs): ...

from django.utils.text import slugify

class BlogPost(models.Model):

title = models.CharField(max\_length=255)

content = models.TextField()

slug = models.SlugField(blank=True)

image = models.ImageField(default='default.jpg',

upload\_to='posts/%Y/%m/%d')

def save(self, \*args, \*\*kwargs):

self.slug = slugify(self.title)

super(BlogPost, self).save(\*args, \*\*kwargs)