Introduction to class-based views

Class-based views provide an alternative way to implement views as Python objects instead of functions. They do not replace function-based views, but have certain differences and advantages when compared to function-based views:

* Organization of code related to specific HTTP methods (**GET**, **POST**, etc.) can be addressed by separate methods instead of conditional branching.
* Object oriented techniques such as mixins (multiple inheritance) can be used to factor code into reusable components.

## Using class-based views

At its core, a class-based view allows you to respond to different HTTP request methods with different class instance methods, instead of with conditionally branching code inside a single view function.

So where the code to handle HTTP **GET** in a view function would look something like:

**from** **django.http** **import** HttpResponse

**def** my\_view(request):

**if** request.method == 'GET':

*# <view logic>*

**return** HttpResponse('result')

In a class-based view, this would become:

**from** **django.http** **import** HttpResponse

**from** **django.views** **import** View

**class** **MyView**(View):

**def** get(self, request):

*# <view logic>*

**return** HttpResponse('result')

**from** **django.urls** **import** path

**from** **myapp.views** **import** MyView

urlpatterns = [

path('about/', MyView.as\_view()),

]

## Handling forms with class-based views[¶](https://docs.djangoproject.com/en/2.1/topics/class-based-views/intro/#handling-forms-with-class-based-views)

A basic function-based view that handles forms may look something like this:

**from** **django.http** **import** HttpResponseRedirect

**from** **django.shortcuts** **import** render

**from** **.forms** **import** MyForm

**def** myview(request):

**if** request.method == "POST":

form = MyForm(request.POST)

**if** form.is\_valid():

*# <process form cleaned data>*

**return** HttpResponseRedirect('/success/')

**else**:

form = MyForm(initial={'key': 'value'})

**return** render(request, 'form\_template.html', {'form': form})

A similar class-based view might look like:

**from** **django.http** **import** HttpResponseRedirect

**from** **django.shortcuts** **import** render

**from** **django.views** **import** View

**from** **.forms** **import** MyForm

**class** **MyFormView**(View):

form\_class = MyForm

initial = {'key': 'value'}

template\_name = 'form\_template.html'

**def** get(self, request, \*args, \*\*kwargs):

form = self.form\_class(initial=self.initial)

**return** render(request, self.template\_name, {'form': form})

**def** post(self, request, \*args, \*\*kwargs):

form = self.form\_class(request.POST)

**if** form.is\_valid():

*# <process form cleaned data>*

**return** HttpResponseRedirect('/success/')

**return** render(request, self.template\_name, {'form': form})

Class-based views[**¶**](https://docs.djangoproject.com/en/2.1/topics/class-based-views/#class-based-views)

A view is a callable which takes a request and returns a response. This can be more than just a function, and Django provides an example of some classes which can be used as views. These allow you to structure your views and reuse code by harnessing inheritance and mixins. There are also some generic views for simple tasks which we’ll get to later, but you may want to design your own structure of reusable views which suits your use case. For full details, see the [class-based views reference documentation](https://docs.djangoproject.com/en/2.1/ref/class-based-views/).

* [Introduction to class-based views](https://docs.djangoproject.com/en/2.1/topics/class-based-views/intro/)
* [Built-in class-based generic views](https://docs.djangoproject.com/en/2.1/topics/class-based-views/generic-display/)
* [Form handling with class-based views](https://docs.djangoproject.com/en/2.1/topics/class-based-views/generic-editing/)
* [Using mixins with class-based views](https://docs.djangoproject.com/en/2.1/topics/class-based-views/mixins/)

Basic examples[**¶**](https://docs.djangoproject.com/en/2.1/topics/class-based-views/#basic-examples)

Django provides base view classes which will suit a wide range of applications. All views inherit from the [**View**](https://docs.djangoproject.com/en/2.1/ref/class-based-views/base/#django.views.generic.base.View) class, which handles linking the view in to the URLs, HTTP method dispatching and other simple features. [**RedirectView**](https://docs.djangoproject.com/en/2.1/ref/class-based-views/base/#django.views.generic.base.RedirectView) is for a simple HTTP redirect, and [**TemplateView**](https://docs.djangoproject.com/en/2.1/ref/class-based-views/base/#django.views.generic.base.TemplateView) extends the base class to make it also render a template.

Simple usage in your URLconf[**¶**](https://docs.djangoproject.com/en/2.1/topics/class-based-views/#simple-usage-in-your-urlconf)

The simplest way to use generic views is to create them directly in your URLconf. If you’re only changing a few simple attributes on a class-based view, you can simply pass them into the [**as\_view()**](https://docs.djangoproject.com/en/2.1/ref/class-based-views/base/#django.views.generic.base.View.as_view) method call itself:

**from** **django.urls** **import** path

**from** **django.views.generic** **import** TemplateView

urlpatterns = [

path('about/', TemplateView.as\_view(template\_name="about.html")),

]

Any arguments passed to [**as\_view()**](https://docs.djangoproject.com/en/2.1/ref/class-based-views/base/#django.views.generic.base.View.as_view) will override attributes set on the class. In this example, we set **template\_name** on the **TemplateView**. A similar overriding pattern can be used for the **url** attribute on [**RedirectView**](https://docs.djangoproject.com/en/2.1/ref/class-based-views/base/#django.views.generic.base.RedirectView).

Subclassing generic views[**¶**](https://docs.djangoproject.com/en/2.1/topics/class-based-views/#subclassing-generic-views)

The second, more powerful way to use generic views is to inherit from an existing view and override attributes (such as the **template\_name**) or methods (such as **get\_context\_data**) in your subclass to provide new values or methods. Consider, for example, a view that just displays one template, **about.html**. Django has a generic view to do this -[**TemplateView**](https://docs.djangoproject.com/en/2.1/ref/class-based-views/base/#django.views.generic.base.TemplateView) - so we can just subclass it, and override the template name:

*# some\_app/views.py*

**from** **django.views.generic** **import** TemplateView

**class** **AboutView**(TemplateView):

template\_name = "about.html"

Then we just need to add this new view into our URLconf. [**TemplateView**](https://docs.djangoproject.com/en/2.1/ref/class-based-views/base/#django.views.generic.base.TemplateView) is a class, not a function, so we point the URL to the [**as\_view()**](https://docs.djangoproject.com/en/2.1/ref/class-based-views/base/#django.views.generic.base.View.as_view) class method instead, which provides a function-like entry to class-based views:

*# urls.py*

**from** **django.urls** **import** path

**from** **some\_app.views** **import** AboutView

urlpatterns = [

path('about/', AboutView.as\_view()),

]

For more information on how to use the built in generic views, consult the next topic on [generic class-based views](https://docs.djangoproject.com/en/2.1/topics/class-based-views/generic-display/).

**Supporting other HTTP methods**[**¶**](https://docs.djangoproject.com/en/2.1/topics/class-based-views/#supporting-other-http-methods)

Suppose somebody wants to access our book library over HTTP using the views as an API. The API client would connect every now and then and download book data for the books published since last visit. But if no new books appeared since then, it is a waste of CPU time and bandwidth to fetch the books from the database, render a full response and send it to the client. It might be preferable to ask the API when the most recent book was published.

We map the URL to book list view in the URLconf:

**from** **django.urls** **import** path

**from** **books.views** **import** BookListView

urlpatterns = [

path('books/', BookListView.as\_view()),

]

And the view:

**from** **django.http** **import** HttpResponse

**from** **django.views.generic** **import** ListView

**from** **books.models** **import** Book

**class** **BookListView**(ListView):

model = Book

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

last\_book = self.get\_queryset().latest('publication\_date')

response = HttpResponse('')

*# RFC 1123 date format*

response['Last-Modified'] = last\_book.publication\_date.strftime('%a, **%d** %b %Y %H:%M:%S GMT')

**return** response

## Handling forms with class-based views[¶](https://docs.djangoproject.com/en/2.1/topics/class-based-views/intro/#handling-forms-with-class-based-views)

A basic function-based view that handles forms may look something like this:

**from** **django.http** **import** HttpResponseRedirect

**from** **django.shortcuts** **import** render

**from** **.forms** **import** MyForm

**def** myview(request):

**if** request.method == "POST":

form = MyForm(request.POST)

**if** form.is\_valid():

*# <process form cleaned data>*

**return** HttpResponseRedirect('/success/')

**else**:

form = MyForm(initial={'key': 'value'})

**return** render(request, 'form\_template.html', {'form': form})

A similar class-based view might look like:

**from** **django.http** **import** HttpResponseRedirect

**from** **django.shortcuts** **import** render

**from** **django.views** **import** View

**from** **.forms** **import** MyForm

**class** **MyFormView**(View):

form\_class = MyForm

initial = {'key': 'value'}

template\_name = 'form\_template.html'

**def** get(self, request, \*args, \*\*kwargs):

form = self.form\_class(initial=self.initial)

**return** render(request, self.template\_name, {'form': form})

**def** post(self, request, \*args, \*\*kwargs):

form = self.form\_class(request.POST)

**if** form.is\_valid():

*# <process form cleaned data>*

**return** HttpResponseRedirect('/success/')

**return** render(request, self.template\_name, {'form': form})

## Handling forms with class-based views[¶](https://docs.djangoproject.com/en/2.1/topics/class-based-views/intro/#handling-forms-with-class-based-views)

A basic function-based view that handles forms may look something like this:

**from** **django.http** **import** HttpResponseRedirect

**from** **django.shortcuts** **import** render

**from** **.forms** **import** MyForm

**def** myview(request):

**if** request.method == "POST":

form = MyForm(request.POST)

**if** form.is\_valid():

*# <process form cleaned data>*

**return** HttpResponseRedirect('/success/')

**else**:

form = MyForm(initial={'key': 'value'})

**return** render(request, 'form\_template.html', {'form': form})

A similar class-based view might look like:

**from** **django.http** **import** HttpResponseRedirect

**from** **django.shortcuts** **import** render

**from** **django.views** **import** View

**from** **.forms** **import** MyForm

**class** **MyFormView**(View):

form\_class = MyForm

initial = {'key': 'value'}

template\_name = 'form\_template.html'

**def** get(self, request, \*args, \*\*kwargs):

form = self.form\_class(initial=self.initial)

**return** render(request, self.template\_name, {'form': form})

**def** post(self, request, \*args, \*\*kwargs):

form = self.form\_class(request.POST)

**if** form.is\_valid():

*# <process form cleaned data>*

**return** HttpResponseRedirect('/success/')

**return** render(request, self.template\_name, {'form': form})

Built-in class-based generic views[**¶**](https://docs.djangoproject.com/en/2.1/topics/class-based-views/generic-display/#built-in-class-based-generic-views)

Writing Web applications can be monotonous, because we repeat certain patterns again and again. Django tries to take away some of that monotony at the model and template layers, but Web developers also experience this boredom at the view level.

Django’s *generic views* were developed to ease that pain. They take certain common idioms and patterns found in view development and abstract them so that you can quickly write common views of data without having to write too much code.

We can recognize certain common tasks, like displaying a list of objects, and write code that displays a list of *any* object. Then the model in question can be passed as an extra argument to the URLconf.

Django ships with generic views to do the following:

* Display list and detail pages for a single object. If we were creating an application to manage conferences then a **TalkListView** and a **RegisteredUserListView** would be examples of list views. A single talk page is an example of what we call a “detail” view.
* Present date-based objects in year/month/day archive pages, associated detail, and “latest” pages.
* Allow users to create, update, and delete objects – with or without authorization.

Taken together, these views provide easy interfaces to perform the most common tasks developers encounter.

Extending generic views[**¶**](https://docs.djangoproject.com/en/2.1/topics/class-based-views/generic-display/#extending-generic-views)

There’s no question that using generic views can speed up development substantially. In most projects, however, there comes a moment when the generic views no longer suffice. Indeed, the most common question asked by new Django developers is how to make generic views handle a wider array of situations.

This is one of the reasons generic views were redesigned for the 1.3 release - previously, they were just view functions with a bewildering array of options; now, rather than passing in a large amount of configuration in the URLconf, the recommended way to extend generic views is to subclass them, and override their attributes or methods.

That said, generic views will have a limit. If you find you’re struggling to implement your view as a subclass of a generic view, then you may find it more effective to write just the code you need, using your own class-based or functional views.

More examples of generic views are available in some third party applications, or you could write your own as needed.

Generic views of objects[**¶**](https://docs.djangoproject.com/en/2.1/topics/class-based-views/generic-display/#generic-views-of-objects)

[**TemplateView**](https://docs.djangoproject.com/en/2.1/ref/class-based-views/base/#django.views.generic.base.TemplateView) certainly is useful, but Django’s generic views really shine when it comes to presenting views of your database content. Because it’s such a common task, Django comes with a handful of built-in generic views that make generating list and detail views of objects incredibly easy.

Let’s start by looking at some examples of showing a list of objects or an individual object.

We’ll be using these models:

*# models.py*

**from** **django.db** **import** models

**class** **Publisher**(models.Model):

name = models.CharField(max\_length=30)

address = models.CharField(max\_length=50)

city = models.CharField(max\_length=60)

state\_province = models.CharField(max\_length=30)

country = models.CharField(max\_length=50)

website = models.URLField()

**class** **Meta**:

ordering = ["-name"]

**def** \_\_str\_\_(self):

**return** self.name

**class** **Author**(models.Model):

salutation = models.CharField(max\_length=10)

name = models.CharField(max\_length=200)

email = models.EmailField()

headshot = models.ImageField(upload\_to='author\_headshots')

**def** \_\_str\_\_(self):

**return** self.name

**class** **Book**(models.Model):

title = models.CharField(max\_length=100)

authors = models.ManyToManyField('Author')

publisher = models.ForeignKey(Publisher, on\_delete=models.CASCADE)

publication\_date = models.DateField()

Now we need to define a view:

*# views.py*

**from** **django.views.generic** **import** ListView

**from** **books.models** **import** Publisher

**class** **PublisherList**(ListView):

model = Publisher

Finally hook that view into your urls:

*# urls.py*

**from** **django.urls** **import** path

**from** **books.views** **import** PublisherList

urlpatterns = [

path('publishers/', PublisherList.as\_view()),

]

That’s all the Python code we need to write. We still need to write a template, however. We could explicitly tell the view which template to use by adding a **template\_name** attribute to the view, but in the absence of an explicit template Django will infer one from the object’s name. In this case, the inferred template will be **"books/publisher\_list.html"** – the “books” part comes from the name of the app that defines the model, while the “publisher” bit is just the lowercased version of the model’s name.

**Note**

Thus, when (for example) the **APP\_DIRS** option of a **DjangoTemplates** backend is set to True in [**TEMPLATES**](https://docs.djangoproject.com/en/2.1/ref/settings/#std:setting-TEMPLATES), a template location could be: /path/to/project/books/templates/books/publisher\_list.html

This template will be rendered against a context containing a variable called **object\_list** that contains all the publisher objects. A very simple template might look like the following:

{% **extends** "base.html" %}

{% **block** content %}

<**h2**>Publishers</**h2**>

<**ul**>

{% **for** publisher **in** object\_list %}

<**li**>{{ publisher.name }}</**li**>

{% **endfor** %}

</**ul**>

{% **endblock** %}

That’s really all there is to it. All the cool features of generic views come from changing the attributes set on the generic view. The [generic views reference](https://docs.djangoproject.com/en/2.1/ref/class-based-views/) documents all the generic views and their options in detail; the rest of this document will consider some of the common ways you might customize and extend generic views.

**Making “friendly” template contexts**[**¶**](https://docs.djangoproject.com/en/2.1/topics/class-based-views/generic-display/#making-friendly-template-contexts)

You might have noticed that our sample publisher list template stores all the publishers in a variable named **object\_list**. While this works just fine, it isn’t all that “friendly” to template authors: they have to “just know” that they’re dealing with publishers here.

Well, if you’re dealing with a model object, this is already done for you. When you are dealing with an object or queryset, Django is able to populate the context using the lower cased version of the model class’ name. This is provided in addition to the default **object\_list** entry, but contains exactly the same data, i.e. **publisher\_list**.

If this still isn’t a good match, you can manually set the name of the context variable. The **context\_object\_name**attribute on a generic view specifies the context variable to use:

*# views.py*

**from** **django.views.generic** **import** ListView

**from** **books.models** **import** Publisher

**class** **PublisherList**(ListView):

model = Publisher

context\_object\_name = 'my\_favorite\_publishers'

Providing a useful **context\_object\_name** is always a good idea. Your coworkers who design templates will thank you.

**Adding extra context**[**¶**](https://docs.djangoproject.com/en/2.1/topics/class-based-views/generic-display/#adding-extra-context)

Often you simply need to present some extra information beyond that provided by the generic view. For example, think of showing a list of all the books on each publisher detail page. The [**DetailView**](https://docs.djangoproject.com/en/2.1/ref/class-based-views/generic-display/#django.views.generic.detail.DetailView) generic view provides the publisher to the context, but how do we get additional information in that template?

The answer is to subclass [**DetailView**](https://docs.djangoproject.com/en/2.1/ref/class-based-views/generic-display/#django.views.generic.detail.DetailView) and provide your own implementation of the **get\_context\_data** method. The default implementation simply adds the object being displayed to the template, but you can override it to send more:

**from** **django.views.generic** **import** DetailView

**from** **books.models** **import** Book, Publisher

**class** **PublisherDetail**(DetailView):

model = Publisher

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

*# Call the base implementation first to get a context*

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

*# Add in a QuerySet of all the books*

context['book\_list'] = Book.objects.all()

**return** context

**Note**

Generally, **get\_context\_data** will merge the context data of all parent classes with those of the current class. To preserve this behavior in your own classes where you want to alter the context, you should be sure to call **get\_context\_data** on the super class. When no two classes try to define the same key, this will give the expected results. However if any class attempts to override a key after parent classes have set it (after the call to super), any children of that class will also need to explicitly set it after super if they want to be sure to override all parents. If you’re having trouble, review the method resolution order of your view.

Another consideration is that the context data from class-based generic views will override data provided by context processors; see [**get\_context\_data()**](https://docs.djangoproject.com/en/2.1/ref/class-based-views/mixins-single-object/#django.views.generic.detail.SingleObjectMixin.get_context_data) for an example.

**Viewing subsets of objects**[**¶**](https://docs.djangoproject.com/en/2.1/topics/class-based-views/generic-display/#viewing-subsets-of-objects)

Now let’s take a closer look at the **model** argument we’ve been using all along. The **model** argument, which specifies the database model that the view will operate upon, is available on all the generic views that operate on a single object or a collection of objects. However, the **model** argument is not the only way to specify the objects that the view will operate upon – you can also specify the list of objects using the **queryset** argument:

**from** **django.views.generic** **import** DetailView

**from** **books.models** **import** Publisher

**class** **PublisherDetail**(DetailView):

context\_object\_name = 'publisher'

queryset = Publisher.objects.all()

Specifying **model = Publisher** is really just shorthand for saying **queryset = Publisher.objects.all()**. However, by using **queryset** to define a filtered list of objects you can be more specific about the objects that will be visible in the view (see [Making queries](https://docs.djangoproject.com/en/2.1/topics/db/queries/) for more information about [**QuerySet**](https://docs.djangoproject.com/en/2.1/ref/models/querysets/#django.db.models.query.QuerySet) objects, and see the [class-based views reference](https://docs.djangoproject.com/en/2.1/ref/class-based-views/) for the complete details).

To pick a simple example, we might want to order a list of books by publication date, with the most recent first:

**from** **django.views.generic** **import** ListView

**from** **books.models** **import** Book

**class** **BookList**(ListView):

queryset = Book.objects.order\_by('-publication\_date')

context\_object\_name = 'book\_list'

That’s a pretty simple example, but it illustrates the idea nicely. Of course, you’ll usually want to do more than just reorder objects. If you want to present a list of books by a particular publisher, you can use the same technique:

**from** **django.views.generic** **import** ListView

**from** **books.models** **import** Book

**class** **AcmeBookList**(ListView):

context\_object\_name = 'book\_list'

queryset = Book.objects.filter(publisher\_\_name='ACME Publishing')

template\_name = 'books/acme\_list.html'

Notice that along with a filtered **queryset**, we’re also using a custom template name. If we didn’t, the generic view would use the same template as the “vanilla” object list, which might not be what we want.

Also notice that this isn’t a very elegant way of doing publisher-specific books. If we want to add another publisher page, we’d need another handful of lines in the URLconf, and more than a few publishers would get unreasonable. We’ll deal with this problem in the next section.

**Note**

If you get a 404 when requesting **/books/acme/**, check to ensure you actually have a Publisher with the name ‘ACME Publishing’. Generic views have an **allow\_empty** parameter for this case. See the [class-based-views reference](https://docs.djangoproject.com/en/2.1/ref/class-based-views/) for more details.

**Dynamic filtering**[**¶**](https://docs.djangoproject.com/en/2.1/topics/class-based-views/generic-display/#dynamic-filtering)

Another common need is to filter down the objects given in a list page by some key in the URL. Earlier we hard-coded the publisher’s name in the URLconf, but what if we wanted to write a view that displayed all the books by some arbitrary publisher?

Handily, the **ListView** has a [**get\_queryset()**](https://docs.djangoproject.com/en/2.1/ref/class-based-views/mixins-multiple-object/#django.views.generic.list.MultipleObjectMixin.get_queryset) method we can override. Previously, it has just been returning the value of the **queryset** attribute, but now we can add more logic.

The key part to making this work is that when class-based views are called, various useful things are stored on **self**; as well as the request (**self.request**) this includes the positional (**self.args**) and name-based (**self.kwargs**) arguments captured according to the URLconf.

Here, we have a URLconf with a single captured group:

*# urls.py*

**from** **django.urls** **import** path

**from** **books.views** **import** PublisherBookList

urlpatterns = [

path('books/<publisher>/', PublisherBookList.as\_view()),

]

Next, we’ll write the **PublisherBookList** view itself:

*# views.py*

**from** **django.shortcuts** **import** get\_object\_or\_404

**from** **django.views.generic** **import** ListView

**from** **books.models** **import** Book, Publisher

**class** **PublisherBookList**(ListView):

template\_name = 'books/books\_by\_publisher.html'

**def** get\_queryset(self):

self.publisher = get\_object\_or\_404(Publisher, name=self.kwargs['publisher'])

**return** Book.objects.filter(publisher=self.publisher)

As you can see, it’s quite easy to add more logic to the queryset selection; if we wanted, we could use **self.request.user** to filter using the current user, or other more complex logic.

We can also add the publisher into the context at the same time, so we can use it in the template:

*# ...*

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

*# Call the base implementation first to get a context*

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

*# Add in the publisher*

context['publisher'] = self.publisher

**return** context

**Performing extra work**[**¶**](https://docs.djangoproject.com/en/2.1/topics/class-based-views/generic-display/#performing-extra-work)

The last common pattern we’ll look at involves doing some extra work before or after calling the generic view.

Imagine we had a **last\_accessed** field on our **Author** model that we were using to keep track of the last time anybody looked at that author:

*# models.py*

**from** **django.db** **import** models

**class** **Author**(models.Model):

salutation = models.CharField(max\_length=10)

name = models.CharField(max\_length=200)

email = models.EmailField()

headshot = models.ImageField(upload\_to='author\_headshots')

last\_accessed = models.DateTimeField()

The generic **DetailView** class, of course, wouldn’t know anything about this field, but once again we could easily write a custom view to keep that field updated.

First, we’d need to add an author detail bit in the URLconf to point to a custom view:

**from** **django.urls** **import** path

**from** **books.views** **import** AuthorDetailView

urlpatterns = [

*#...*

path('authors/<int:pk>/', AuthorDetailView.as\_view(), name='author-detail'),

]

Then we’d write our new view – **get\_object** is the method that retrieves the object – so we simply override it and wrap the call:

**from** **django.utils** **import** timezone

**from** **django.views.generic** **import** DetailView

**from** **books.models** **import** Author

**class** **AuthorDetailView**(DetailView):

queryset = Author.objects.all()

**def** get\_object(self):

obj = super().get\_object()

*# Record the last accessed date*

obj.last\_accessed = timezone.now()

obj.save()

**return** obj

**Note**

The URLconf here uses the named group **pk** - this name is the default name that **DetailView** uses to find the value of the primary key used to filter the queryset.

If you want to call the group something else, you can set **pk\_url\_kwarg** on the view. More details can be found in the reference for [**DetailView**](https://docs.djangoproject.com/en/2.1/ref/class-based-views/generic-display/#django.views.generic.detail.DetailView)

Bottom of Form