

## When and how to use Django ListView

### When to use ListView?

Django provides several class based generic views to accomplish common tasks. One among them is ListView.

Most basic class based generic view is TemplateView. We wrote about it in our [last post](#).

ListView **should be used** when you want to present a list of objects in a html page.

ListView **shouldn't be used** when your page has forms and does creation or update of objects. FormView, CreateView and UpdateView are more suitable for working with forms, creation or updation of objects.

TemplateView can achieve everything which ListView can, but ListView has an advantage of avoiding a lot of boilerplate code which would be needed with TemplateView.

Let's write a view using base view **View** and then modify it to use TemplateView and then to use ListView. ListView would help us avoid several lines of code and would also provide better separation of concern.

### Vanilla View

Assume there is a model called Book which looks like:

```
class Book(models.Model):  
    name = models.CharField(max_length=100)  
    author_name = models.CharField(max_length=100)
```

# Agiliq

```
class BookListView(View):  
  
    def get(self, request, *args, **kwargs):  
        books = Book.objects.all()  
        context = {'books': books}  
        return render(request, "book-list.html", context=context)
```

book-list.html looks like the following:

## By subclassing TemplateView

```
class BookListView(TemplateView):  
    template_name = 'book-list.html'  
  
    def get_context_data(self, *args, **kwargs):  
        context = super(BookListView, self).get_context_data(*args, **kwargs)  
        context['books'] = Book.objects.all()  
        return context
```

As discussed in last post on TemplateView, we didn't have to provide a `get()` implementation and didn't have to bother with `render()` while using TemplateView. All that was taken care of by TemplateView.

We only had to provide a `get_context_data()` implementation to add context to the template.

## By subclassing ListView

```
from django.views.generic.list import ListView  
  
class BookListView(ListView):  
    template_name = 'book-list.html'  
    queryset = Book.objects.all()  
    context_object_name = 'books'
```

# Agiliq

We can also add filtering in `ListView.queryset`.

```
class BookListView(ListView):  
    template_name = 'book-list.html'  
    queryset = Book.objects.filter(name='A Feast for Crows')  
    context_object_name = 'books'
```

Had we wanted pagination, we would have had to add several lines of code in `TemplateView` or vanilla `View` implementation. `ListView` provides pagination for free, we don't have to add pagination code.

Pagination can be added to `ListView` subclasses by setting a variable `paginate_by`

```
class BookListView(ListView):  
    template_name = 'book-list.html'  
    queryset = Book.objects.all()  
    context_object_name = 'books'  
    paginate_by = 10
```

After this `/books-list/?page=1` will return first 10 books. `/books-list/?page=2` will return next 10 books and so on.

## Further configuring ListView

If your list page's queryset doesn't need any filtering, and works with `.all()` on your model, then you can provide a `model` attribute on the `BookListView` instead of providing `queryset`.

```
class BookListView(ListView):  
    template_name = 'book-list.html'  
    model = Book  
    context_object_name = 'books'  
    paginate_by = 10
```

You can add ordering to your queryset by adding `ordering` attribute on `View`. Suppose you want the books to be ordered in the page by their created date descending. You can do:

# Agiliq

```
model = Book
context_object_name = 'books'
paginate_by = 10
ordering = ['-created']
```

Since ordering is a list, so you can order by multiple attributes.

In case you want to filter the queryset differently for different web requests, then you can skip adding `model` or `queryset` on the list view and instead provide a `get_queryset()` implementation.

```
class BookListView(ListView):
    template_name = 'book-list.html'
    context_object_name = 'books'
    paginate_by = 10
    ordering = ['-created']

    def get_queryset(self):
        return Book.objects.filter(created_by=self.request.user)
```

You can avoid `template_name` attribute too. The default behaviour of ListView is to use a template with name `<app-label>/<model-name>_list.html`. You can change your BookListView to look like:

```
class BookListView(ListView):
    model = Book
    context_object_name = 'books'
    paginate_by = 10
```

But then your template code should be in `books/book_list.html`. This assumes that your Book model is in app `books`. In case Book model is in, say `entitites` app, then template code should be in `entities/book_list.html`.

You can avoid `context_object_name` too. The default behaviour of ListView is to populate the template with context name `object_list`. You can change your BookListView to look like:

# Agiliq

```
paginate_by = 10
```

In such case your template code should change to:

Essentially a ListView helps you avoid boilerplate code like:

- providing a GET() implementation.
- Creating queryset with ordering.
- providing an encapsulated pagination code. Had we written pagination code in vanilla view, it would have easily added more 10 lines of code.
- providing the template with a sane context.
- creating and returning a HttpResponse() or a subclass of HttpResponse() object.

The **must** requirement for a ListView is, it must be provided with a `model` or `queryset` or a `get_queryset()` implementation. Every other piece has a sane default.

## Our other posts on generic class views

- [TemplateView](#)
- [DetailView](#)
- [FormView](#)
- [CreateView](#)

Thank you for reading the Agiliq blog. This article was written by Akshar on Dec 29, 2017 in [django](#).

You can [subscribe](#) ✉ to our [blog](#).

We love building amazing apps for web and mobile for our clients. If you are looking for development help, [contact us today](#) ✉.

Would you like to download 10+ free Django and Python books? [Get them here](#)



# Agiliq

Comments for this thread are now closed

×

0 Comments

The Agiliq Blog

1

Login ▾

---

♥ Recommend

2

🐦 Tweet

📌 Share


Sort by Oldest ▾

This discussion has been closed.

ALSO ON THE AGILIQ BLOG

Profiling Django Middlewares - Agiliq Blog | Django web app development

4 years ago


zealfire

— Very informative post!

Avatar

Understanding Django Middlewares - Agiliq Blog | Django web app ...

4 years ago



Ashish Verma

— you're right!


Avatar

The output is like:

Middleware executed

Building Chrome Extensions - Agiliq Blog | Django web app development

5 years ago



Techy Ashutosh

— I like the way of coding, but


Avatar

i am confused from where i put this code? Can

you help me Karambir?

AngularJS injectors internals - Agiliq Blog | Django web app development

2 years ago



Palani Suresh

— Thanks for the valuable

Avatar

information to be share with us.This is very

helpful to me,understanding the concepts of

✉ Subscribe

📄

Add Disqus to your siteAdd DisqusAdd

🔒

Disqus' Privacy PolicyPrivacy PolicyPrivacy Policy

# Agiliq

Building Amazing Apps. © 2010-2018, Agiliq  
All rights reserved.

- About Us
- Blog
- Books
- Newsletter

# Agiliq

---

Phone us: **+919949997612**