Pagination¶

Django provides high-level and low-level ways to help you manage paginated data – that is, data that's split across several pages, with "Previous/Next" links.

The Paginator class¶

Under the hood, all methods of pagination use the Paginator class. It does all the heavy lifting of actually splitting a QuerySet into Page objects.

Example¶

Give Paginator a list of objects, plus the number of items you'd like to have on each page, and it gives you methods for accessing the items for each page:

```
>>> from django.core.paginator import Paginator
>>> objects = ['john', 'paul', 'george', 'ringo']
>>> p = Paginator(objects, 2)

>>> p.count
4
```

```
>>> p.num_pages
2
>>> type(p.page_range)
<class 'range_iterator'>
>>> p.page_range
range(1, 3)

>>> page1 = p.page(1)
>>> page1
<Page 1 of 2>
>>> page1.object_list
['john', 'paul']

>>> page2 = p.page(2)
>>> page2.object_list
['george', 'ringo']
>>> page2.has_next()
False
>>> page2.has_previous()
True
>>> page2.has_other_pages()
True
>>> page2.next_page_number()
Traceback (most recent call last):
...
EmptyPage: That page contains no results
```

```
>>> page2.previous_page_number()
1
>>> page2.start_index() # The 1-based index of the first item on this page
3
>>> page2.end_index() # The 1-based index of the last item on this page
4

>>> p.page(0)
Traceback (most recent call last):
...
EmptyPage: That page number is less than 1
>>> p.page(3)
Traceback (most recent call last):
...
EmptyPage: That page contains no results
```

Note

Note that you can give Paginator a list/tuple, a Django QuerySet, or any other object with a count() or __len__() method. When determining the number of objects contained in the passed object, Paginator will first try calling count(), then fallback to using len() if the passed object has no count() method. This allows objects such as

Django's QuerySet to use a more efficient count() method when available.

Paginating a ListView¶

django.views.generic.list.ListView provides a builtin way to paginate the displayed list. You can do this by adding a paginate_by attribute to your view class, for example:

```
from django.views.generic import ListView

from myapp.models import Contact

class ContactListView(ListView):
    paginate_by = 2
    model = Contact
```

This limits the number of objects per page and adds a paginator and page_obj to the context. To allow your users to navigate between pages, add links to the next and previous page, in your template like this:

```
{% for contact in page_obj %}
   {# Each "contact" is a Contact model object. #}
   {{ contact.full_name|upper }}<br>
   ...
{% endfor %}
```

```
<div class="pagination">
   <span class="step-links">
      {% if page_obj.has_previous %}
         <a href="?page=1">&laquo; first</a>
         <a                                    href="?page={{
page_obj.previous_page_number }}">previous</a>
      {% endif %}

      <span class="current">
         Page {{ page_obj.number }} of {{
page_obj.paginator.num_pages }}.
      </span>

      {% if page_obj.has_next %}
         <a                                    href="?page={{
page_obj.next_page_number }}">next</a>
```

```
        <a                                    href="?page={{
page_obj.paginator.num_pages }}">last &raquo;</a>
      {% endif %}
   </span>
</div>
```

Using Paginator in a view function¶

Here's an example using Paginator in a view function to paginate a queryset:

```
from django.core.paginator import Paginator
from django.shortcuts import render

from myapp.models import Contact

def listing(request):
   contact_list = Contact.objects.all()
   paginator = Paginator(contact_list, 25) # Show 25
contacts per page.

   page_number = request.GET.get('page')
   page_obj = paginator.get_page(page_number)
   return render(request, 'list.html', {'page_obj':
page_obj})
```

In the template list.html, you can include navigation between pages in the same way as in the template for the ListView above.

Logging
Security in Django

Additional Information