# Django EL(Endless) Pagination Documentation

***Release 3.0.2***

**Oleksandr Shtalinberg and Francesco Banconi**

**Jun 13, 2017**

# Contents

This application provides Twitter- and Digg-style pagination, with multiple and lazy pagination and optional Ajax support. It is devoted to implementing web pagination in very few steps.

The **source code** for this app is hosted at https://github.com/shtalinberg/django-el-pagination.

*Getting started* is easy!

Contents:

CHAPTER 1

# Changelog

## Version 3.1.0

**Template changes:** link attribute rel="{{ querystring_key }}" replaced by data-el-querystring-key="{{ querystring_key }}"

**New feature**: Django 1.11 support. **New feature**:

added view for maintaining original functionality on page index out of range, but setting response code to 404 `PAGE_OUT_OF_RANGE_404` default *False* If True on page out of range, throw a 404 exception, otherwise display the first page

**Documentation: render_to_response deprecated in django 1.10** replaced to `return render(request, template, context)`

## Version 3.0.0

**New feature**: Django 1.10 support. New app Django EL(Endless) Pagination now supports Django from 1.8.x to 1.10

---

**New feature**: Travic CI support add tox and Travic CI config

---

**Documentation**: general clean up.

## Version 2.1.1

Bug-fix release

**Fix**: page_template decorator doesn't change template of ajax call

---

**Fix**: Fix syntax error in declaring variable in javascript

## Version 2.1.0

New name app: django-el-pagination

**New feature**: Django 1.8 and 1.9 support. New app Django EL(Endless) Pagination now supports Django from 1.4.x to 1.9

new jQuery plugin that can be found in `static/el-pagination/js/el-pagination.js`.

Support get the numbers of objects are normally display in per page

Usage:

```
{{ pages.per_page_number }}
```

add a class on chunk complete

Each time a chunk size is complete, the class `endless_chunk_complete` is added to the *show more* link,

## Version 2.0

**New feature**: Python 3 support.

Django Endless Pagination now supports both Python 2 and **Python 3**. Dropped support for Python 2.5. See *Getting started* for the new list of requirements.

---

**New feature**: the **JavaScript refactoring**.

This version introduces a re-designed Ajax support for pagination. Ajax can now be enabled using a brand new jQuery plugin that can be found in `static/el-pagination/js/el-pagination.js`.

Usage:

```
{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>$.endlessPaginate();</script>
{% endblock %}
```

The last line in the block above enables Ajax requests to retrieve new pages for each pagination in the page. That's basically the same as the old approach of loading the file `endless.js`. The new approach, however, is more jQuery-idiomatic, increases the flexibility of how objects can be paginated, implements some *new features* and also contains some bug fixes.

For backward compatibility, the application still includes the two JavaScript `endless.js` and `endless_on_scroll.js` files. However, please consider *migrating* as soon as possible: the old JavaScript files are deprecated, are no longer maintained, and don't provide the new JavaScript features. Also note that the old Javascript files will not work if jQuery >= 1.9 is used.

New features include ability to **paginate different objects with different options**, precisely **selecting what to bind**, ability to **register callbacks**, support for **pagination in chunks** and much more.

Please refer to the *JavaScript reference* for a detailed overview of the new features and for instructions on *how to migrate* from the old JavaScript files to the new one.

---

**New feature**: the *page_templates* decorator also accepts a sequence of `(template, key)` pairs, functioning as a dict mapping templates and keys (still present), e.g.:

```
from endless_pagination.decorators import page_templates

@page_templates((
    ('myapp/entries_page.html', None),
    ('myapp/other_entries_page.html', 'other_entries_page'),
))
def entry_index():
    ...
```

This also supports serving different paginated objects with the same template.

---

**New feature**: ability to provide nested context variables in the *paginate* and *lazy_paginate* template tags, e.g.:

```
{% paginate entries.all as myentries %}
```

The code above is basically equivalent to:

```
{% with entries.all as myentries %}
    {% paginate myentries %}
{% endwith %}
```

In this case, and only in this case, the *as* argument is mandatory, and a *TemplateSyntaxError* will be raised if the variable name is missing.

---

**New feature**: the page list object returned by the *get_pages* template tag has been improved adding the following new methods:

```
{# whether the page list contains more than one page #}
{{ pages.paginated }}

{# the 1-based index of the first item on the current page #}
{{ pages.current_start_index }}

{# the 1-based index of the last item on the current page #}
{{ pages.current_end_index }}

{# the total number of objects, across all pages #}
{{ pages.total_count }}

{# the first page represented as an arrow #}
{{ pages.first_as_arrow }}

{# the last page represented as an arrow #}
{{ pages.last_as_arrow }}
```

In the *arrow* representation, the page label defaults to << for the first page and to >> for the last one. As a consequence, the labels of the previous and next pages are now single brackets, respectively < and >. First and last

pages' labels can be customized using `settings.ENDLESS_PAGINATION_FIRST_LABEL` and `settings.ENDLESS_PAGINATION_LAST_LABEL`: see *Customization*.

---

**New feature**: The sequence returned by the callable `settings.ENDLESS_PAGINATION_PAGE_LIST_CALLABLE` can now contain two new values:

- *'first'*: will display the first page as an arrow;
- *'last'*: will display the last page as an arrow.

The *show_pages* template tag documentation describes how to customize Digg-style pagination defining your own page list callable.

When using the default Digg-style pagination (i.e. when `settings.ENDLESS_PAGINATION_PAGE_LIST_CALLABLE` is set to *None*), it is possible to enable first / last page arrows by setting the new flag `settings.ENDLESS_PAGINATION_DEFAULT_CALLABLE_ARROWS` to *True*.

---

**New feature**: `settings.ENDLESS_PAGINATION_PAGE_LIST_CALLABLE` can now be either a callable or a **dotted path** to a callable, e.g.:

```
ENDLESS_PAGINATION_PAGE_LIST_CALLABLE = 'path.to.callable'
```

In addition to the default, `endless_pagination.utils.get_page_numbers`, an alternative implementation is now available: `endless_pagination.utils.get_elastic_page_numbers`. It adapts its output to the number of pages, making it arguably more usable when there are many of them. To enable it, add the following line to your `settings.py`:

```
ENDLESS_PAGINATION_PAGE_LIST_CALLABLE = (
    'endless_pagination.utils.get_elastic_page_numbers')
```

---

**New feature**: ability to create a development and testing environment (see *Contributing*).

---

**New feature**: in addition to the ability to provide a customized pagination URL as a context variable, the *paginate* and *lazy_paginate* tags now support hardcoded pagination URL endpoints, e.g.:

```
{% paginate 20 entries with "/mypage/" %}
```

---

**New feature**: ability to specify negative indexes as values for the `starting from page` argument of the *paginate* template tag.

When changing the default page, it is now possible to reference the last page (or the second last page, and so on) by using negative indexes, e.g:

```
{% paginate entries starting from page -1 %}
```

See *Templatetags reference*.

---

**Documentation**: general clean up.

---

**Documentation**: added a *Contributing* page. Have a look!

---

**Documentation**: included a comprehensive *JavaScript reference*.

---

**Fix**: `endless_pagination.views.AjaxListView` no longer subclasses `django.views.generic.list.ListView`. Instead, the base objects and mixins composing the final view are now defined by this app.

This change eliminates the ambiguity of having two separate pagination machineries in place: the Django Endless Pagination one and the built-in Django `ListView` one.

---

**Fix**: the *using* argument of *paginate* and *lazy_paginate* template tags now correctly handles querystring keys containing dashes, e.g.:

```
{% lazy_paginate entries using "entries-page" %}
```

---

**Fix**: replaced namespace `endless_pagination.paginator` with `endless_pagination.paginators`: the module contains more than one paginator classes.

---

**Fix**: in some corner cases, loading `endless_pagination.models` raised an *ImproperlyConfigured* error while trying to pre-load the templates.

---

**Fix**: replaced doctests with proper unittests. Improved the code coverage as a consequence. Also introduced integration tests exercising JavaScript, based on Selenium.

---

**Fix**: overall code lint and clean up.

## Version 1.1

**New feature**: now it is possible to set the bottom margin used for pagination on scroll (default is 1 pixel).

For example, if you want the pagination on scroll to be activated when 20 pixels remain until the end of the page:

```
<script src="http://code.jquery.com/jquery-latest.js"></script>
<script src="{{ STATIC_URL }}endless_pagination/js/endless.js"></script>
<script src="{{ STATIC_URL }}endless_pagination/js/endless_on_scroll.js"></script>

{# add the lines below #}
<script type="text/javascript" charset="utf-8">
    var endless_on_scroll_margin = 20;
</script>
```

---

**New feature**: added ability to avoid Ajax requests when multiple pagination is used.

A template for multiple pagination with Ajax support may look like this (see *Multiple paginations in the same page*):

```
{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}endless_pagination/js/endless.js"></script>
{% endblock %}

<h2>Entries:</h2>
<div class="endless_page_template">
    {% include "myapp/entries_page.html" %}
</div>

<h2>Other entries:</h2>
<div class="endless_page_template">
    {% include "myapp/other_entries_page.html" %}
</div>
```

But what if you need Ajax pagination for *entries* but not for *other entries*? You will only have to add a class named `endless_page_skip` to the page container element, e.g.:

```
<h2>Other entries:</h2>
<div class="endless_page_template endless_page_skip">
    {% include "myapp/other_entries_page.html" %}
</div>
```

---

**New feature**: implemented a class-based generic view allowing Ajax pagination of a list of objects (usually a query-set).

Intended as a substitution of *django.views.generic.ListView*, it recreates the behaviour of the *page_template* decorator.

For a complete explanation, see *Generic views*.

---

**Fix**: the `page_template` and `page_templates` decorators no longer hide the original view name and docstring (*update_wrapper*).

---

**Fix**: pagination on scroll now works on Firefox >= 4.

---

**Fix**: tests are now compatible with Django 1.3.

CHAPTER 2

# Getting started

## Requirements

| | |
|---|---|
| Python | >= 2.7 (or Python 3) |
| Django | >= 1.8 |
| jQuery | >= 1.7 |

## Installation

The Git repository can be cloned with this command:

```
git clone https://github.com/shtalinberg/django-el-pagination.git
```

The `el_pagination` package, included in the distribution, should be placed on the `PYTHONPATH`.

Otherwise you can just `easy_install -Z django-el-pagination` or `pip install django-el-pagination`.

## Settings

Add the request context processor to your *settings.py*, e.g.:

```
from django.conf.global_settings import TEMPLATES

TEMPLATES[0]['OPTIONS']['context_processors'].insert(0, 'django.core.context_
↪processors.request')
```

or just adding it to the context_processors manually like so:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates'), ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                '...',
                '...',
                '...',
                '...',
                'django.template.context_processors.request', ## For EL-pagination
            ],
        },
    },
]
```

Add `'el_pagination'` to the `INSTALLED_APPS` to your *settings.py*.

See the *Customization* section for other settings.

## Quickstart

Given a template like this:

```
{% for entry in entries %}
    {# your code to show the entry #}
{% endfor %}
```

you can use Digg-style pagination to display objects just by adding:

```
{% load el_pagination_tags %}

{% paginate entries %}
{% for entry in entries %}
    {# your code to show the entry #}
{% endfor %}
{% show_pages %}
```

Done.

This is just a basic example. To continue exploring all the Django Endless Pagination features, have a look at *Twitter-style Pagination* or *Digg-style pagination*.

CHAPTER 3

# Twitter-style Pagination

Assuming the developer wants Twitter-style pagination of entries of a blog post, in *views.py* we have class-based:

```
from el_pagination.views import AjaxListView

class EntryListView(AjaxListView):
    context_object_name = "entry_list"
    template_name = "myapp/entry_list.html"

    def get_queryset(self):
        return Entry.objects.all()
```

or fuction-based:

```
def entry_index(request, template='myapp/entry_list.html'):
    context = {
        'entry_list': Entry.objects.all(),
    }
    return render(request, template, context)
```

In *myapp/entry_list.html*:

```
<h2>Entries:</h2>
{% for entry in entry_list %}
    {# your code to show the entry #}
{% endfor %}
```

## Split the template

The response to an Ajax request should not return the entire template, but only the portion of the page to be updated or added. So it is convenient to extract from the template the part containing the entries, and use it to render the context if the request is Ajax. The main template will include the extracted part, so it is convenient to put the page template name in the context.

*views.py* class-based becomes:

```
from el_pagination.views import AjaxListView

class EntryListView(AjaxListView):
    context_object_name = "entry_list"
    template_name = "myapp/entry_list.html"
    page_template='myapp/entry_list_page.html'

    def get_queryset(self):
        return Entry.objects.all()
```

or fuction-based:

```
def entry_list(request,
    template='myapp/entry_list.html',
    page_template='myapp/entry_list_page.html'):
    context = {
        'entry_list': Entry.objects.all(),
        'page_template': page_template,
    }
    if request.is_ajax():
        template = page_template
    return render(request, template, context)
```

See *below* how to obtain the same result **just decorating the view**.

*myapp/entry_list.html* becomes:

```
<h2>Entries:</h2>
{% include page_template %}
```

*myapp/entry_list_page.html* becomes:

```
{% for entry in entry_list %}
    {# your code to show the entry #}
{% endfor %}
```

## A shortcut for ajaxed views

A good practice in writing views is to allow other developers to inject the template name and extra data, so that they are added to the context. This allows the view to be easily reused. Let's resume the original view with extra context injection:

*views.py*:

```
def entry_index(request,
        template='myapp/entry_list.html', extra_context=None):
    context = {
        'entry_list': Entry.objects.all(),
    }
    if extra_context is not None:
        context.update(extra_context)
    return render(request, template, context)
```

Splitting templates and putting the Ajax template name in the context is easily achievable by using an included decorator.

*views.py* becomes:

```
from el_pagination.decorators import page_template

@page_template('myapp/entry_list_page.html')  # just add this decorator
def entry_list(request,
        template='myapp/entry_list.html', extra_context=None):
    context = {
        'entry_list': Entry.objects.all(),
    }
    if extra_context is not None:
        context.update(extra_context)
    return render(request, template, context)
```

## Paginating objects

All that's left is changing the page template and loading the *endless templatetags*, the jQuery library and the jQuery plugin `el-pagination.js` included in the distribution under `/static/el-pagination/js/`.

*myapp/entry_list.html* becomes:

```
<h2>Entries:</h2>
{% include page_template %}

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>$.endlessPaginate();</script>
{% endblock %}
```

*myapp/entry_list_page.html* becomes:

```
{% load el_pagination_tags %}

{% paginate entry_list %}
{% for entry in entry_list %}
    {# your code to show the entry #}
{% endfor %}
{% show_more %}
```

The *paginate* template tag takes care of customizing the given queryset and the current template context. In the context of a Twitter-style pagination the *paginate* tag is often replaced by the *lazy_paginate* one, which offers, more or less, the same functionalities and allows for reducing database access: see *Lazy pagination*.

The *show_more* one displays the link to navigate to the next page.

You might want to glance at the *JavaScript reference* for a detailed explanation of how to integrate JavaScript and Ajax features in Django Endless Pagination.

## Pagination on scroll

If you want new items to load when the user scroll down the browser page, you can use the *pagination on scroll* feature: just set the *paginateOnScroll* option of *$.endlessPaginate()* to *true*, e.g.:

```
<h2>Entries:</h2>
{% include page_template %}

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>$.endlessPaginate({paginateOnScroll: true});</script>
{% endblock %}
```

That's all. See the *Templatetags reference* to improve the use of included templatetags.

It is possible to set the bottom margin used for *pagination on scroll* (default is 1 pixel). For example, if you want the pagination on scroll to be activated when 20 pixels remain to the end of the page:

```
<h2>Entries:</h2>
{% include page_template %}

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>
        $.endlessPaginate({
            paginateOnScroll: true,
            paginateOnScrollMargin: 20
        });
    </script>
{% endblock %}
```

Again, see the *JavaScript reference*.

## On scroll pagination using chunks

Sometimes, when using on scroll pagination, you may want to still display the *show more* link after each *N* pages. In Django Endless Pagination this is called *chunk size*. For instance, a chunk size of 5 means that a *show more* link is displayed after page 5 is loaded, then after page 10, then after page 15 and so on. Activating *chunks* is straightforward, just use the *paginateOnScrollChunkSize* option:

```
{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>
        $.endlessPaginate({
            paginateOnScroll: true,
            paginateOnScrollChunkSize: 5
        });
    </script>
{% endblock %}
```

## Before version 2.0

Django Endless Pagination v2.0 introduces a redesigned Ajax support for pagination. As seen above, Ajax can now be enabled using a brand new jQuery plugin that can be found in `static/el-pagination/js/el-pagination.js`.

For backward compatibility, the application still includes the two JavaScript files `el-pagination-endless.js` and `el-pagination_on_scroll.js` that were used before, so that it is still possible to use code like this:

```
<script src="http://code.jquery.com/jquery-latest.js"></script>
{# Deprecated. #}
<script src="{{ STATIC_URL }}el-pagination/js/el-pagination-endless.js"></script>
```

To enable pagination on scroll, the code was the following:

```
<script src="http://code.jquery.com/jquery-latest.js"></script>
{# Deprecated. #}
<script src="{{ STATIC_URL }}el-pagination/js/el-pagination-endless.js"></script>
<script src="{{ STATIC_URL }}el-pagination/js/el-pagination_on_scroll.js"></script>
```

However, please consider *migrating* as soon as possible: the old JavaScript files are deprecated, are no longer maintained, and don't provide the new JavaScript features. Also note that the old Javascript files will not work if jQuery >= 1.9 is used.

Please refer to the *JavaScript reference* for a detailed overview of the new features and for instructions on *how to migrate* from the old JavaScript files to the new one.

CHAPTER 4

# Digg-style pagination

Digg-style pagination of queryset objects is really easy to implement. If Ajax pagination is not needed, all you have to do is modifying the template, e.g.:

```
{% load el_pagination_tags %}

{% paginate entries %}
{% for entry in entries %}
    {# your code to show the entry #}
{% endfor %}
{% show_pages %}
```

That's it! As seen, the *paginate* template tag takes care of customizing the given queryset and the current template context. The *show_pages* one displays the page links allowing for navigation to other pages.

## Page by page

If you only want to display previous and next links (in a page-by-page pagination) you have to use the lower level *get_pages* template tag, e.g.:

```
{% load el_pagination_tags %}

{% paginate entries %}
{% for entry in entries %}
    {# your code to show the entry #}
{% endfor %}
{% get_pages %}
{{ pages.previous }} {{ pages.next }}
```

*Customization* explains how to customize the arrows that go to previous and next pages.

## Showing indexes

The *get_pages* template tag adds to the current template context a `pages` variable containing several methods that can be used to fully customize how the page links are displayed. For example, assume you want to show the indexes of the entries in the current page, followed by the total number of entries:

```
{% load el_pagination_tags %}

{% paginate entries %}{% get_pages %}
{% for entry in entries %}
    {# your code to show the entry #}
{% endfor %}

Showing entries
{{ pages.current_start_index }}-{{ pages.current_end_index }} of
{{ pages.total_count }}.
{# Just print pages to render the Digg-style pagination. #}
{{ pages }}
```

## Number of pages

You can use `{{ pages|length }}` to retrieve and display the pages count. A common use case is to change the layout or display additional info based on whether the page list contains more than one page. This can be achieved checking `{% if pages|length > 1 %}`, or, in a more convenient way, using `{{ pages.paginated }}`. For example, assume you want to change the layout, or display some info, only if the page list contains more than one page, i.e. the results are actually paginated:

```
{% load el_pagination_tags %}

{% paginate entries %}
{% for entry in entries %}
    {# your code to show the entry #}
{% endfor %}
{% get_pages %}
{% if pages.paginated %}
    Some info/layout to display only if the available
    objects span multiple pages...
    {{ pages }}
{% endif %}
```

Again, for a full overview of the *get_pages* and all the other template tags, see the *Templatetags reference*.

## Adding Ajax

The view is exactly the same as the one used in *Twitter-style Pagination*:

```
from el_pagination.decorators import page_template

@page_template('myapp/entry_index_page.html')  # just add this decorator
def entry_index(
        request, template='myapp/entry_index.html', extra_context=None):
    context = {
```

```
        'entries': Entry.objects.all(),
    }
    if extra_context is not None:
        context.update(extra_context)
    return render(request, template, context)
```

As seen before in *Twitter-style Pagination*, you have to *split the templates*, separating the main one from the fragment representing the single page. However, this time a container for the page template is also required and, by default, must be an element having a class named *endless_page_template*.

*myapp/entry_index.html* becomes:

```
<h2>Entries:</h2>
<div class="endless_page_template">
    {% include page_template %}
</div>

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>$.endlessPaginate();</script>
{% endblock %}
```

*myapp/entry_index_page.html* becomes:

```
{% load el_pagination_tags %}

{% paginate entries %}
{% for entry in entries %}
    {# your code to show the entry #}
{% endfor %}
{% show_pages %}
```

Done.

It is possible to manually *override the container selector* used by *$.endlessPaginate()* to update the page contents. This can be easily achieved by customizing the *pageSelector* option of *$.endlessPaginate()*, e.g.:

```
<h2>Entries:</h2>
<div id="entries">
    {% include page_template %}
</div>

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>$.endlessPaginate({pageSelector: 'div#entries'});</script>
{% endblock %}
```

See the *JavaScript reference* for a detailed explanation of how to integrate JavaScript and Ajax features in Django Endless Pagination.

CHAPTER 5

# Multiple paginations in the same page

Sometimes it is necessary to show different types of paginated objects in the same page. In this case we have to associate a different querystring key to every pagination.

Normally, the key used is the one specified in `settings.ENDLESS_PAGINATION_PAGE_LABEL` (see *Customization*), but in the case of multiple pagination the application provides a simple way to override the settings.

If you do not need Ajax, the only file you need to edit is the template. Here is an example with 2 different paginations (*entries* and *other_entries*) in the same page, but there is no limit to the number of different paginations in a page:

```
{% load el_pagination_tags %}

{% paginate entries %}
{% for entry in entries %}
    {# your code to show the entry #}
{% endfor %}
{% show_pages %}

{# "other_entries_page" is the new querystring key #}
{% paginate other_entries using "other_entries_page" %}
{% for entry in other_entries %}
    {# your code to show the entry #}
{% endfor %}
{% show_pages %}
```

The `using` argument of the *paginate* template tag allows you to choose the name of the querystring key used to track the page number. If not specified the system falls back to `settings.EL_PAGINATION_PAGE_LABEL`.

In the example above, the url *http://example.com?page=2&other_entries_page=3* requests the second page of *entries* and the third page of *other_entries*.

The name of the querystring key can also be dinamically passed in the template context, e.g.:

```
{# page_variable is not surrounded by quotes #}
{% paginate other_entries using page_variable %}
```

You can use any style of pagination: *show_pages*, *get_pages*, *show_more* etc... (see *Templatetags reference*).

## Adding Ajax for multiple pagination

Obviously each pagination needs a template for the page contents. Remember to box each page in a div with a class called *endless_page_template*, or to specify the container selector passing an option to *$.endlessPaginate()* as seen in *Digg-style pagination and Ajax*.

*myapp/entry_index.html*:

```
<h2>Entries:</h2>
<div class="endless_page_template">
    {% include "myapp/entries_page.html" %}
</div>

<h2>Other entries:</h2>
<div class="endless_page_template">
    {% include "myapp/other_entries_page.html" %}
</div>

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>$.endlessPaginate();</script>
{% endblock %}
```

See the *JavaScript reference* for further details on how to use the included jQuery plugin.

*myapp/entries_page.html*:

```
{% load el_pagination_tags %}

{% paginate entries %}
{% for entry in entries %}
    {# your code to show the entry #}
{% endfor %}
{% show_pages %}
```

*myapp/other_entries_page.html*:

```
{% load el_pagination_tags %}

{% paginate other_entries using other_entries_page %}
{% for entry in other_entries %}
    {# your code to show the entry #}
{% endfor %}
{% show_pages %}
```

As seen *before*, the decorator `page_template` simplifies the management of Ajax requests in views. You must, however, map different paginations to different page templates.

You can chain decorator calls relating a template to the associated querystring key, e.g.:

```
from endless_pagination.decorators import page_template

@page_template('myapp/entries_page.html')
@page_template('myapp/other_entries_page.html', key='other_entries_page')
def entry_index(
        request, template='myapp/entry_index.html', extra_context=None):
```

```
    context = {
        'entries': Entry.objects.all(),
        'other_entries': OtherEntry.objects.all(),
    }
    if extra_context is not None:
        context.update(extra_context)
    return render_to_response(
        template, context, context_instance=RequestContext(request))
```

As seen in previous examples, if you do not specify the *key* kwarg in the decorator, then the page template is associated to the querystring key defined in the settings. You can use the `page_templates` (note the trailing *s*) decorator in substitution of a decorator chain when you need multiple Ajax paginations. The previous example can be written as:

```
from endless_pagination.decorators import page_templates

@page_templates({
    'myapp/entries_page.html': None,
    'myapp/other_entries_page.html': 'other_entries_page',
})
def entry_index():
    ...
```

As seen, a dict object is passed to the `page_templates` decorator, mapping templates to querystring keys. Alternatively, you can also pass a sequence of `(template, key)` pairs, e.g.:

```
from endless_pagination.decorators import page_templates

@page_templates((
    ('myapp/entries_page.html', None),
    ('myapp/other_entries_page.html', 'other_entries_page'),
))
def entry_index():
    ...
```

This also supports serving different paginated objects with the same template.

## Manually selecting what to bind

What if you need Ajax pagination only for *entries* and not for *other entries*? You can do this in a straightforward way using jQuery selectors, e.g.:

```
{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>$('#entries').endlessPaginate();</script>
{% endblock %}
```

The call to *$('#entries').endlessPaginate()* applies Ajax pagination starting from the DOM node with id *entries* and to all sub-nodes. This means that *other entries* are left intact. Of course you can use any selector supported by jQuery.

Refer to the *JavaScript reference* for an explanation of other features like calling *$.endlessPaginate()* multiple times in order to customize the behavior of each pagination in a multiple pagination view.

CHAPTER 6

# Lazy pagination

Usually pagination requires hitting the database to get the total number of items to display. Lazy pagination avoids this *select count* query and results in a faster page load, with a disadvantage: you won't know the total number of pages in advance.

For this reason it is better to use lazy pagination in conjunction with *Twitter-style Pagination* (e.g. using the *show_more* template tag).

In order to switch to lazy pagination you have to use the *lazy_paginate* template tag instead of the *paginate* one, e.g.:

```
{% load el_pagination_tags %}

{% lazy_paginate entries %}
{% for entry in entries %}
    {# your code to show the entry #}
{% endfor %}
{% show_more %}
```

The *lazy_paginate* tag can take all the args of the *paginate* one, with one exception: negative indexes can not be passed to the `starting from page` argument.

CHAPTER 7

# Different number of items on the first page

Sometimes you might want to show on the first page a different number of items than on subsequent pages (e.g. in a movie detail page you want to show 4 images of the movie as a reminder, making the user click to see the next 20 images). To achieve this, use the *paginate* or *lazy_paginate* tags with comma separated *first page* and *per page* arguments, e.g.:

```
{% load el_pagination_tags %}

{% lazy_paginate 4,20 entries %}
{% for entry in entries %}
    {# your code to show the entry #}
{% endfor %}
{% show_more %}
```

This code will display 4 entries on the first page and 20 entries on the other pages.

Of course the *first page* and *per page* arguments can be passed as template variables, e.g.:

```
{% lazy_paginate first_page,per_page entries %}
```

CHAPTER 8

# Getting the current page number

## In the template

You can get and display the current page number in the template using the *show_current_number* template tag, e.g.:

```
{% show_current_number %}
```

This call will display the current page number, but you can also insert the value in the context as a template variable:

```
{% show_current_number as page_number %}
{{ page_number }}
```

See the *show_current_number* refrence for more information on accepted arguments.

## In the view

If you need to get the current page number in the view, you can use an utility function called `get_page_number_from_request`, e.g.:

```
from el_pagination import utils

page = utils.get_page_number_from_request(request)
```

If you are using *multiple pagination*, or you have changed the default querystring for pagination, you can pass the querystring key as an optional argument:

```
page = utils.get_page_number_from_request(request, querystring_key=mykey)
```

If the page number is not present in the request, by default *1* is returned. You can change this behaviour using:

```
page = utils.get_page_number_from_request(request, default=3)
```

CHAPTER 9

# Templatetags reference

## paginate

Usage:

```
{% paginate entries %}
```

After this call, the *entries* variable in the template context is replaced by only the entries of the current page.

You can also keep your *entries* original variable (usually a queryset) and add to the context another name that refers to entries of the current page, e.g.:

```
{% paginate entries as page_entries %}
```

The *as* argument is also useful when a nested context variable is provided as queryset. In this case, and only in this case, the resulting variable name is mandatory, e.g.:

```
{% paginate entries.all as entries %}
```

The number of paginated entries is taken from settings, but you can override the default locally, e.g.:

```
{% paginate 20 entries %}
```

Of course you can mix it all:

```
{% paginate 20 entries as paginated_entries %}
```

By default, the first page is displayed the first time you load the page, but you can change this, e.g.:

```
{% paginate entries starting from page 3 %}
```

When changing the default page, it is also possible to reference the last page (or the second last page, and so on) by using negative indexes, e.g:

```
{% paginate entries starting from page -1 %}
```

This can be also achieved using a template variable that was passed to the context, e.g.:

```
{% paginate entries starting from page page_number %}
```

If the passed page number does not exist, the first page is displayed. Note that negative indexes are specific to the `{% paginate %}` tag: this feature cannot be used when contents are lazy paginated (see *lazy_paginate* below).

If you have multiple paginations in the same page, you can change the querydict key for the single pagination, e.g.:

```
{% paginate entries using article_page %}
```

In this case *article_page* is intended to be a context variable, but you can hardcode the key using quotes, e.g.:

```
{% paginate entries using 'articles_at_page' %}
```

Again, you can mix it all (the order of arguments is important):

```
{% paginate 20 entries starting from page 3 using page_key as paginated_entries %}
```

Additionally you can pass a path to be used for the pagination:

```
{% paginate 20 entries using page_key with pagination_url as paginated_entries %}
```

This way you can easily create views acting as API endpoints, and point your Ajax calls to that API. In this case *pagination_url* is considered a context variable, but it is also possible to hardcode the URL, e.g.:

```
{% paginate 20 entries with "/mypage/" %}
```

If you want the first page to contain a different number of items than subsequent pages, you can separate the two values with a comma, e.g. if you want 3 items on the first page and 10 on other pages:

```
{% paginate 3,10 entries %}
```

You must use this tag before calling the *show_more*, *get_pages* or *show_pages* ones.

## lazy_paginate

Paginate objects without hitting the database with a *select count* query. Usually pagination requires hitting the database to get the total number of items to display. Lazy pagination avoids this *select count* query and results in a faster page load, with a disadvantage: you won't know the total number of pages in advance.

Use this in the same way as *paginate* tag when you are not interested in the total number of pages.

The `lazy_paginate` tag can take all the args of the `paginate` one, with one exception: negative indexes can not be passed to the `starting from page` argument.

## show_more

Show the link to get the next page in a *Twitter-style Pagination*. Usage:

```
{% show_more %}
```

Alternatively you can override the label passed to the default template:

```
{% show_more "even more" %}
```

You can override the loading text too:

```
{% show_more "even more" "working" %}
```

Must be called after *paginate* or *lazy_paginate*.

## get_pages

Usage:

```
{% get_pages %}
```

This is mostly used for *Digg-style pagination*.

This call inserts in the template context a *pages* variable, as a sequence of page links. You can use *pages* in different ways:

- just print *pages* and you will get Digg-style pagination displayed:

```
{{ pages }}
```

- display pages count:

```
{{ pages|length }}
```

- display numbers of objects in per page:

```
{{ pages.per_page_number }}
```

- check if the page list contains more than one page:

```
{{ pages.paginated }}
{# the following is equivalent #}
{{ pages|length > 1 }}
```

- get a specific page:

```
{# the current selected page #}
{{ pages.current }}

{# the first page #}
{{ pages.first }}

{# the last page #}
{{ pages.last }}

{# the previous page (or nothing if you are on first page) #}
{{ pages.previous }}

{# the next page (or nothing if you are in last page) #}
{{ pages.next }}

{# the third page #}
```

```
{{ pages.3 }}
{# this means page.1 is the same as page.first #}

{# the 1-based index of the first item on the current page #}
{{ pages.current_start_index }}

{# the 1-based index of the last item on the current page #}
{{ pages.current_end_index }}

{# the total number of objects, across all pages #}
{{ pages.total_count }}

{# the first page represented as an arrow #}
{{ pages.first_as_arrow }}

{# the last page represented as an arrow #}
{{ pages.last_as_arrow }}
```

- iterate over *pages* to get all pages:

```
{% for page in pages %}
    {# display page link #}
    {{ page }}

    {# the page url (beginning with "?") #}
    {{ page.url }}

    {# the page path #}
    {{ page.path }}

    {# the page number #}
    {{ page.number }}

    {# a string representing the page (commonly the page number) #}
    {{ page.label }}

    {# check if the page is the current one #}
    {{ page.is_current }}

    {# check if the page is the first one #}
    {{ page.is_first }}

    {# check if the page is the last one #}
    {{ page.is_last }}
{% endfor %}
```

You can change the variable name, e.g.:

```
{% get_pages as page_links %}
```

This must be called after *paginate* or *lazy_paginate*.

## show_pages

Show page links. Usage:

```
{% show_pages %}
```

It is just a shortcut for:

```
{% get_pages %}
{{ pages }}
```

You can set `EL_PAGINATION_PAGE_LIST_CALLABLE` in your *settings.py* to a callable used to customize the pages that are displayed. `EL_PAGINATION_PAGE_LIST_CALLABLE` can also be a dotted path representing a callable, e.g.:

```
EL_PAGINATION_PAGE_LIST_CALLABLE = 'path.to.callable'
```

The callable takes the current page number and the total number of pages, and must return a sequence of page numbers that will be displayed.

The sequence can contain other values:

- *'previous'*: will display the previous page in that position;
- *'next'*: will display the next page in that position;
- *'first'*: will display the first page as an arrow;
- *'last'*: will display the last page as an arrow;
- *None*: a separator will be displayed in that position.

Here is an example of a custom callable that displays the previous page, then the first page, then a separator, then the current page, and finally the last page:

```
def get_page_numbers(current_page, num_pages):
    return ('previous', 1, None, current_page, 'last')
```

If `EL_PAGINATION_PAGE_LIST_CALLABLE` is *None* the internal callable `endless_pagination.utils.get_page_numbers` is used, generating a Digg-style pagination.

An alternative implementation is available: `endless_pagination.utils.get_elastic_page_numbers`: it adapts its output to the number of pages, making it arguably more usable when there are many of them.

This must be called after *paginate* or *lazy_paginate*.

## show_current_number

Show the current page number, or insert it in the context.

This tag can for example be useful to change the page title according to the current page number.

To just show current page number:

```
{% show_current_number %}
```

If you use multiple paginations in the same page, you can get the page number for a specific pagination using the querystring key, e.g.:

```
{% show_current_number using mykey %}
```

The default page when no querystring is specified is 1. If you changed it in the *paginate* template tag, you have to call `show_current_number` according to your choice, e.g.:

```
{% show_current_number starting from page 3 %}
```

This can be also achieved using a template variable you passed to the context, e.g.:

```
{% show_current_number starting from page page_number %}
```

You can of course mix it all (the order of arguments is important):

```
{% show_current_number starting from page 3 using mykey %}
```

If you want to insert the current page number in the context, without actually displaying it in the template, use the *as* argument, i.e.:

```
{% show_current_number as page_number %}
{% show_current_number starting from page 3 using mykey as page_number %}
```

CHAPTER 10

# JavaScript reference

For each type of pagination it is possible to enable Ajax so that the requested page is loaded using an asynchronous request to the server. This is especially important for *Twitter-style Pagination* and *endless pagination on scroll*, but *Digg-style pagination* can also take advantage of this technique.

## Activating Ajax support

Ajax support is activated linking jQuery and the `el-pagination.js` file included in this app. It is then possible to use the *$.endlessPaginate()* jQuery plugin to enable Ajax pagination, e.g.:

```
<h2>Entries:</h2>
<div class="endless_page_template">
    {% include page_template %}
</div>

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>$.endlessPaginate();</script>
{% endblock %}
```

This example assumes that you *separated the fragment* containing the single page (*page_tempate*) from the main template (the code snipper above). More on this in *Twitter-style Pagination* and *Digg-style pagination*.

The *$.endlessPaginate()* call activates Ajax for each pagination present in the page.

## Pagination on scroll

If you want new items to load when the user scrolls down the browser page, you can use the **pagination on scroll** feature: just set the *paginateOnScroll* option of *$.endlessPaginate()* to *true*, e.g.:

```
<h2>Entries:</h2>
<div class="endless_page_template">
    {% include page_template %}
</div>

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>$.endlessPaginate({paginateOnScroll: true});</script>
{% endblock %}
```

That's all. See the *Templatetags reference* page to improve usage of the included templatetags.

It is possible to set the **bottom margin** used for pagination on scroll (default is 1 pixel). For example, if you want the pagination on scroll to be activated when 20 pixels remain to the end of the page:

```
<h2>Entries:</h2>
<div class="endless_page_template">
    {% include page_template %}
</div>

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>
        $.endlessPaginate({
            paginateOnScroll: true,
            paginateOnScrollMargin: 200
        });
    </script>
{% endblock %}
```

## Attaching callbacks

It is possible to customize the behavior of JavaScript pagination by attaching callbacks to *$.endlessPaginate()*, called when the following events are fired:

- *onClick*: the user clicks on a page link;
- *onCompleted*: the new page is fully loaded and inserted in the DOM.

The context of both callbacks is the clicked link fragment: in other words, inside the callbacks, *this* will be the HTML fragment representing the clicked link, e.g.:

```
<h2>Entries:</h2>
<div class="endless_page_template">
    {% include page_template %}
</div>

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>
```

```
        $.endlessPaginate({
            onClick: function() {
                console.log('Label:', $(this).text());
            }
        });
    </script>
{% endblock %}
```

Both callbacks also receive a *context* argument containing information about the requested page:

- *context.url*: the requested URL;
- *context.key*: the querystring key used to retrieve the requested contents.

If the *onClick* callback returns *false*, the pagination process is stopped, the Ajax request is not performed and the *onCompleted* callback never called.

The *onCompleted* callbacks also receives a second argument: the data returned by the server. Basically this is the HTML fragment representing the new requested page.

To wrap it up, here is an example showing the callbacks' signatures:

```
<h2>Entries:</h2>
<div class="endless_page_template">
    {% include page_template %}
</div>

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>
        $.endlessPaginate({
            onClick: function(context) {
                console.log('Label:', $(this).text());
                console.log('URL:', context.url);
                console.log('Querystring key:', context.key);
                if (forbidden) {  // to be defined...
                    return false;
                }
            },
            onCompleted: function(context, fragment) {
                console.log('Label:', $(this).text());
                console.log('URL:', context.url);
                console.log('Querystring key:', context.key);
                console.log('Fragment:', fragment);
            }
        });
    </script>
{% endblock %}
```

## Manually selecting what to bind

As seen above, *$.endlessPaginate()* enables Ajax support for each pagination in the page. But assuming you are using *Multiple paginations in the same page*, e.g.:

```
<h2>Entries:</h2>
<div id="entries" class="endless_page_template">
    {% include "myapp/entries_page.html" %}
</div>

<h2>Other entries:</h2>
<div id="other-entries" class="endless_page_template">
    {% include "myapp/other_entries_page.html" %}
</div>

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>$.endlessPaginate();</script>
{% endblock %}
```

What if you need Ajax pagination only for *entries* and not for *other entries*? You can do this in a straightforward way using jQuery selectors, e.g.:

```
{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>$('#entries').endlessPaginate();</script>
{% endblock %}
```

The call to *$('#entries').endlessPaginate()* applies Ajax pagination starting from the DOM node with id *entries* and to all sub-nodes. This means that *other entries* are left intact. Of course you can use any selector supported by jQuery.

At this point, you might have already guessed that *$.endlessPaginate()* is just an alias for *$('body').endlessPaginate()*.

## Customize each pagination

You can also call *$.endlessPaginate()* multiple times if you want to customize the behavior of each pagination. E.g. if you need to register a callback for *entries* but not for *other entries*:

```
<h2>Entries:</h2>
<div id="entries" class="endless_page_template">
    {% include "myapp/entries_page.html" %}
</div>

<h2>Other entries:</h2>
<div id="other-entries" class="endless_page_template">
    {% include "myapp/other_entries_page.html" %}
</div>

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>
        $('#entries').endlessPaginate({
            onCompleted: function(data) {
                console.log('New entries loaded.');
            }
```

```
        });
        $('#other-entries').endlessPaginate();
    </script>
{% endblock %}
```

## Selectors

Each time *$.endlessPaginate()* is used, several JavaScript selectors are used to select DOM nodes. Here is a list of them all:

- containerSelector: '.endless_container' (Twitter-style pagination container selector);
- loadingSelector: '.endless_loading' - (Twitter-style pagination loading selector);
- moreSelector: 'a.endless_more' - (Twitter-style pagination link selector);
- pageSelector: '.endless_page_template' (Digg-style pagination page template selector);
- pagesSelector: 'a.endless_page_link' (Digg-style pagination link selector).

An example can better explain the meaning of the selectors above. Assume you have a Digg-style pagination like the following:

```
<h2>Entries:</h2>
<div id="entries" class="endless_page_template">
    {% include "myapp/entries_page.html" %}
</div>

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>
        $('#entries').endlessPaginate();
    </script>
{% endblock %}
```

Here the `#entries` node is selected and Digg-style pagination is applied. Digg-style needs to know which DOM node will be updated with new contents, and in this case it's the same node we selected, because we added the *endless_page_template* class to that node, and *.endless_page_template* is the selector used by default. However, the following example is equivalent and does not involve adding another class to the container:

```
<h2>Entries:</h2>
<div id="entries">
    {% include "myapp/entries_page.html" %}
</div>

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>
        $('#entries').endlessPaginate({
            pageSelector: '#entries'
        });
    </script>
{% endblock %}
```

## On scroll pagination using chunks

Sometimes, when using on scroll pagination, you may want to still display the *show more* link after each *N* pages. In Django Endless Pagination this is called *chunk size*. For instance, a chunk size of 5 means that a *show more* link is displayed after page 5 is loaded, then after page 10, then after page 15 and so on. Activating this functionality is straightforward, just use the *paginateOnScrollChunkSize* option:

```
{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>
        $.endlessPaginate({
            paginateOnScroll: true,
            paginateOnScrollChunkSize: 5
        });
    </script>
{% endblock %}
```

Each time a chunk size is complete, the class `endless_chunk_complete` is added to the *show more* link, so you still have a way to distinguish between the implicit click done by the scroll event and a real click on the button.

## Migrate from version 1.1 to 2.1

Django Endless Pagination v2.0 introduces changes in how Ajax pagination is handled by JavaScript. These changes are discussed in this document and in the *Changelog*.

The JavaScript code now lives in a file named `el-pagination.js`. For backward compatibility, the application still includes the two JavaScript files `el-pagination-endless.js` and `el-pagination_on_scroll.js`. However, please consider migrating as soon as possible: the old JavaScript files are deprecated, are no longer maintained, and don't provide the new JavaScript features.

Instructions on how to migrate from the old version to the new one follow.

### Basic migration

Before:

```
<h2>Entries:</h2>
{% include page_template %}

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination-endless.js"></script>
{% endblock %}
```

Now:

```
<h2>Entries:</h2>
{% include page_template %}

{% block js %}
    {{ block.super }}
```

```
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>$.endlessPaginate();</script>
{% endblock %}
```

## Pagination on scroll

Before:

```
<h2>Entries:</h2>
{% include page_template %}

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination-endless.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination_on_scroll.js"></
↪script>
{% endblock %}
```

Now:

```
<h2>Entries:</h2>
{% include page_template %}

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>
        $.endlessPaginate({paginateOnScroll: true});
    </script>
{% endblock %}
```

## Pagination on scroll with customized bottom margin

Before:

```
<h2>Entries:</h2>
{% include page_template %}

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination-endless.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination_on_scroll.js"></
↪script>
    <script>
        var endless_on_scroll_margin = 200;
    </script>
{% endblock %}
```

Now:

```
<h2>Entries:</h2>
{% include page_template %}

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>
        $.endlessPaginate({
            paginateOnScroll: true,
            paginateOnScrollMargin: 200
        });
    </script>
{% endblock %}
```

## Avoid enabling Ajax on one or more paginations

Before:

```
<h2>Other entries:</h2>
<div class="endless_page_template endless_page_skip">
    {% include "myapp/other_entries_page.html" %}
</div>

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination-endless.js"></script>
{% endblock %}
```

Now:

```
<h2>Other entries:</h2>
<div class="endless_page_template endless_page_skip">
    {% include "myapp/other_entries_page.html" %}
</div>

{% block js %}
    {{ block.super }}
    <script src="http://code.jquery.com/jquery-latest.js"></script>
    <script src="{{ STATIC_URL }}el-pagination/js/el-pagination.js"></script>
    <script>$('not:(.endless_page_skip)').endlessPaginate();</script>
{% endblock %}
```

In this last example, activating Ajax just where you want might be preferred over excluding nodes.

CHAPTER 11

# Generic views

This application provides a customized class-based view, similar to *django.views.generic.ListView*, that allows Ajax pagination of a list of objects (usually a queryset).

## AjaxListView reference

**class** el_pagination.views.**AjaxListView**(*django.views.generic.ListView*)

A class based view, similar to *django.views.generic.ListView*, that allows Ajax pagination of a list of objects.

You can use this class based view in place of *ListView* in order to recreate the behaviour of the *page_template* decorator.

For instance, assume you have this code (taken from Django docs):

```
from django.conf.urls import url
from django.views.generic import ListView
from books.models import Publisher

urlpatterns = [
    url(r'^publishers/$', ListView.as_view(model=Publisher)),
]
```

You want to Ajax paginate publishers, so, as seen, you need to switch the template if the request is Ajax and put the page template into the context as a variable named *page_template*.

This is straightforward, you only need to replace the view class, e.g.:

```
from django.conf.urls import *
from books.models import Publisher

from el_pagination.views import AjaxListView

urlpatterns = [
    url(r'^publishers/$', AjaxListView.as_view(model=Publisher)),
]
```

**key**
the querystring key used for the current pagination (default: *settings.EL_PAGINATION_PAGE_LABEL*)

**page_template**
the template used for the paginated objects

**page_template_suffix**
the template suffix used for autogenerated page_template name (when not given, default='_page')

**get_context_data** (*self*, ***kwargs*)
Adds the *page_template* variable in the context.

If the *page_template* is not given as a kwarg of the *as_view* method then it is invented using app label, model name (obviously if the list is a queryset), *self.template_name_suffix* and *self.page_template_suffix*.

For instance, if the list is a queryset of *blog.Entry*, the template will be *myapp/publisher_list_page.html*.

**get_template_names** (*self*)
Switch the templates for Ajax requests.

**get_page_template** (*self*, ***kwargs*)
Only called if *page_template* is not given as a kwarg of *self.as_view*.

## Generic view example

If the developer wants pagination of publishers, in *views.py* we have code class-based:

```
from django.views.generic import ListView

class EntryListView(ListView)
    model = Publisher
    template_name = "myapp/publisher_list.html"
    context_object_name = "publisher_list"
```

or fuction-based:

```
def entry_index(request, template='myapp/publisher_list.html'):
    context = {
        'publisher_list': Entry.objects.all(),
    }
    return render(request, template, context)
```

In *myapp/publisher_list.html*:

```
<h2>Entries:</h2>
{% for entry in publisher_list %}
    {# your code to show the entry #}
{% endfor %}
```

This is just a basic example. To continue exploring more AjaxListView examples, have a look at *Twitter-style Pagination*

CHAPTER 12

# Customization

## Settings

You can customize the application using `settings.py`.

| Name | Default | Description |
|---|---|---|
| EL_PAGINATION_PER_PAGE | 10 | How many objects are normally displayed in a page (overwriteable by templatetag). |
| EL_PAGINATION_PAGE_LABEL | 'page' | The querystring key of the page number (e.g. http://example.com?page=2). |
| EL_PAGINATION_ORPHANS | 0 | See Django *Paginator* definition of orphans. |
| EL_PAGINATION_LOADING | 'loading' | If you use the default show_more template, here you can customize the content of the loader hidden element. HTML is safe here, e.g. you can show your pretty animated GIF EL_PAGINATION_LOADING = """<img src="/static/img/loader .gif" alt="loading" />""". |
| EL_PAGINATION_PREVIOUS_LABEL | '<' | Default label for the *previous* page link. |
| EL_PAGINATION_NEXT_LABEL | '>' | Default label for the *next* page link. |
| EL_PAGINATION_FIRST_LABEL | '<<' | Default label for the *first* page link. |
| EL_PAGINATION_LAST_LABEL | '>>' | Default label for the *last* page link. |
| EL_PAGINATION_ADD_NOFOLLOW | *False* | Set to *True* if your SEO alchemist wants search engines not to follow pagination links. |
| EL_PAGINATION_PAGE_LINK_CALLABLE | *None* | Callable (or dotted path to a callable) that returns pages to be displayed. If *None*, a default callable is used; that produces *Digg-style pagination*. The applicationt provides also a callable producing elastic pagination: EL_pagination.utils.get_elastic_page_numbers. It adapts its output to the number of pages, making it arguably more usable when there are many of them. See *Templatetags reference* for information about writing custom callables. |
| EL_PAGINATION_DEFAULT_CALLABLE_EXTREMES | 3 | Default number of *extremes* displayed when *Digg-style pagination* is used with the default callable. |
| EL_PAGINATION_DEFAULT_CALLABLE_AROUNDS | 2 | Default number of *arounds* displayed when *Digg-style pagination* is used with the default callable. |
| EL_PAGINATION_DEFAULT_CALLABLE_ARROWS | *False* | Whether or not the first and last pages arrows are displayed when *Digg-style pagination* is used with the default callable. |
| EL_PAGINATION_TEMPLATE_VARNAME | 'template' | Template variable name used by the page_template decorator. You can change this value if you are going to decorate generic views using a different variable name for the template (e.g. template_name). |
| PAGE_OUT_OF_RANGE_404 | *False* | If True on page out of range, throw a 404 exception, otherwise display the first page. There is a view that maintains the original functionality but sets the 404 status code found in el_pagination\views.py |

## Templates and CSS

You can override the default template for show_more templatetag following some rules:

- *more* link is shown only if the variable querystring is not False;
- the container (most external html element) class is *endless_container*;
- the *more* link and the loader hidden element live inside the container;
- the *more* link class is *endless_more*;
- the *more* link data-el-querystring-key attribute is {{ querystring_key }};
- the loader hidden element class is *endless_loading*.

CHAPTER 13

# Contributing

Here are the steps needed to set up a development and testing environment.

**WARNING**

This app use *git flow* for branching strategy and release management.

Please, change code and submit all pull requests into branch *develop*

## Creating a development environment

The development environment is created in a virtualenv. The environment creation requires the *make* and *virtualenv* programs to be installed.

To install *make* under Debian/Ubuntu:

```
$ sudo apt-get install build-essential
```

Under Mac OS/X, *make* is available as part of XCode.

To install virtualenv:

```
$ sudo pip install virtualenv
```

At this point, from the root of this branch, run the command:

```
$ make
```

This command will create a `.venv` directory in the branch root, ignored by DVCSes, containing the development virtual environment with all the dependencies.

## Testing the application

To install *xvfb* (for integration tests) under Debian/Ubuntu:

```
$ sudo apt-get install xvfb
```

If you are on CentOS and using yum, it's:

```
$ yum install xorg-X11-server-Xvfb
```

Run the tests:

```
$ make test
```

The command above also runs all the available integration tests. They use Selenium and require Firefox to be installed. To avoid executing integration tests, define the environment variable SKIP_SELENIUM, e.g.:

```
$ make test SKIP_SELENIUM=1
```

Integration tests are excluded by default when using Python 3. The test suite requires Python >= 2.6.1.

Run the tests and lint/pep8 checks:

```
$ make check
```

Again, to exclude integration tests:

```
$ make check SKIP_SELENIUM=1
```

## Debugging

Run the Django shell (Python interpreter):

```
$ make shell
```

Run the Django development server for manual testing:

```
$ make server
```

After executing the command above, it is possible to navigate the testing project going to <http://localhost:8000>.

See all the available make targets, including info on how to create a Python 3 development environment:

```
$ make help
```

Thanks for contributing, and have fun!

CHAPTER 14

# Source code and contacts

## Repository and bugs

The **source code** for this app is hosted on https://github.com/shtalinberg/django-el-pagination.

To file **bugs and requests**, please use https://github.com/shtalinberg/django-el-pagination/issues.

## Contacts

Oleksandr Shtalinberg

- Email: `o.shtalinberg at gmail.com`

Francesco Banconi

- Email: `frankban at gmail.com`
- IRC: `frankban@freenode`

CHAPTER 15

# Thanks

This application was initially inspired by the excellent tool *django-pagination* (see https://github.com/ericflo/django-pagination).

Thanks to Jannis Leidel for improving the application with some new features, and for contributing the German translation.

And thanks to Nicola 'tekNico' Larosa for reviewing the documentation and for implementing the elastic pagination feature.

# Python Module Index

## e

# Index