

---

# handy Documentation

*Release 0.7*

**Alexander Schepanovski**

**Dec 31, 2017**



---

## Contents

---

<b>1</b>	<b>View decorators</b>	<b>3</b>
<b>2</b>	<b>Shortcuts</b>	<b>7</b>
<b>3</b>	<b>Ajax wrap up</b>	<b>9</b>
<b>4</b>	<b>Mail utilities</b>	<b>11</b>
<b>5</b>	<b>Form widgets</b>	<b>13</b>
<b>6</b>	<b>Model fields</b>	<b>15</b>
<b>7</b>	<b>Middleware</b>	<b>17</b>
<b>8</b>	<b>Database</b>	<b>19</b>
<b>9</b>	<b>Utilities</b>	<b>21</b>



Handy is a collection of different tools, shortcuts, decorators, form and model fields to make your django life easier.

Contents:



---

## View decorators

---

`@render_to([template])`

Decorator for Django views that sends returned dict to `render_to_response()` function. If `template` is not specified then it is guessed from module and view names.

---

**Note:** If view doesn't return dict then decorator does nothing – handy when you need conditionally return redirect.

---

There are a number of keys in view result dictionary which have special meaning:

**TEMPLATE** override template used to render view

**STATUS** set HTTP status code other than 200

**CONTENT\_TYPE** set *Content-Type* of view response

Most common usage:

```
from handy.decorators import render_to

# in module smth.views
@render_to()
def foo(request):
    return {'bar': Bar.objects.all()}

# equals to
def foo(request):
    return render_to_response('smth/foo.html',
                              {'bar': Bar.objects.all()},
                              context_instance=RequestContext(request))
```

`@paginate(name, ip)`

Paginates any queryset or sequence value returned in `name` key in response dict. Also, populates `page` key in response dict if not already used. It passes through if result is not `HttpResponse` or value is not a sequence, so you can return redirects and `None` conveniently:

```

from handy.decorators import render_to, paginate

@render_to()
@paginate('series', 10)
def search(request):
    q = request.GET.get('q')
    return {
        'columns': get_series_columns(),
        'series': search_series_qs(q) if q else None,
    }

```

Here is a Bootstrap and Jinja2 snippet for pagination (just drop all the parentheses to convert it to Django):

```

{% if page and page.has_other_pages() %}
<nav>
  <ul class="pagination">
    {% if page.has_previous() %}
    <li>
      <a href="?p={{ page.previous_page_number() }}" aria-label="Previous">
        <span aria-hidden="true">&laquo;</span>
      </a>
    </li>
    {% endif %}
    {% for p in page.paginator.page_range %}
    <li{% if p == page.number %} class="active"{% endif %}>
      <a href="?p={{ p }}">{{ p }}</a>
    </li>
    {% endfor %}
    {% if page.has_next() %}
    <li>
      <a href="?p={{ page.next_page_number() }}" aria-label="Next">
        <span aria-hidden="true">&raquo;</span>
      </a>
    </li>
    {% endif %}
  </ul>
</nav>
{% endif %}

```

**@render\_to\_json** (*ensure\_ascii=True, default=\_json\_default*)

Serializes view result to json and wraps into `HttpResponse`. Arguments are forwarded to `json.dumps()`.

An example of ajax action handler:

```

from handy.decorators import render_to_json

@render_to_json()
def enable_post(request):
    if not request.user.is_authenticated():
        return {'success': False, 'error': 'login_required'}

    try:
        post = Post.objects.get(pk=request.GET['id'])
    except Post.DoesNotExist:
        return {'success': False, 'error': 'no_post'}

    post.enabled = True
    post.save()

```



```
return {'success': True}
```

Or a JSON datasource:

```
@render_to_json()
def posts_by_tag(request, tag=None):
    posts = Post.object.values().filter(tag=tag)
    return {'success': True, 'data': list(posts)}
```

For higher order tool see *Ajax wrap up*

#### **@last\_modified**

Adds Last-Modified header with current time to view response. Meaned to be used with CommonMiddleware and caching to produce 403 Not Modified responses:

```
from django.views.decorators.cache import cache_page
from handy.decorators import last_modified

@cache_page(60 * 15)
@last_modified
def my_view(request):
    ...
```



**paginate** (*request, queryset, ip*)

Paginates *queryset* and returns a `Page` object. Current page number is extracted from `request.GET['p']` if exists and coerced to available pages:

```
from handy.shortcuts import paginate

def search(request, ...):
    items = Item.objects.filter(...)
    page = paginate(request, items, 10)
    # ...
```

See also `@paginate()` decorator.



## CHAPTER 3

---

### Ajax wrap up

---

An example of ajax handler and JSON datasource:

```
from handy.ajax import ajax

@ajax
@ajax.login_required
@ajax.catch(Post.DoesNotExist)
def enable_post(request):
    post = Post.objects.get(pk=request.GET['id'])

    if post.author != request.user:
        # sends {"success": false, "error": "permission_denied"}
        raise ajax.error('permission_denied')

    post.enabled = True
    post.save()
    # sends {"success": true, "data": null} on successful return

@ajax
def posts_by_tag(request, tag=None):
    # sends {"success": true, "data": [{...}, {...}, ...]}
    return Post.object.values().filter(tag=tag)
```

See also `@render_to_json()`



**render\_to\_email** (*email*, *template* [, *data*] [, *request*] [, *from\_email*] [, *attachment* ])

Renders *template* with context constructed with help of *request* and filled with *data*, then sends it to *email*. An email template could contain email headers:

```
from handy.mail import render_to_email

def approve(...):
    article = ...
    render_to_email(article.author.email, 'approved.html', {'article': article})
```

in *approved.html*:

```
Content-Type: text/html
Subject: Your article «{{ article.title }}» approved

Hello, {{ article.author.username }}!  
<br><br>
....
```

**mail\_admins** (*subject*, *message*=" ", *trace*=True)

Send an email to admins, optionally appends stack trace to message. Handy when you want get an exception email but still serve user request.





**class SimpleWidget**

Just renders a html snippet given to constructor. Can contain `%(name) s` and `%(value) s`.

**class CommaSeparatedInput**

A text field for editing multiple values as comma separated list. Comes handy with `IntegerArrayField`, `BigIntegerArrayField` and `StringArrayField`.

**class MultilineInput**

A textarea for editing multivalue fields. Most usefull with `StringArrayField`.



Here come some custom model fields mostly designed to work with PostgreSQL.

**class IntegerArrayField**

**class BigIntegerArrayField**

**class StringArrayField** (*max\_length=None*)

An arrays of integers, big integers or strings. Most useful to store different array fields to store array of values or choices:

```
DAYS = zip(range(7), 'Sun Mon Tue Wed Thu Fri Sat'.split())

class Company(models.Model):
    phones = StringArrayField('Phone numbers', blank=True, default=lambda: [])
    workdays = IntegerArrayField('Work days', choices=DAYS)

company = Company(phones=['234-5016', '516-2314'], workdays=[1,2,3,4])
company.save()
```

In model form phones field would be represented as *CommaSeparatedInput* and workdays as multiple checkboxes:

```
class CompanyForm(forms.ModelForm):
    class Meta:
        model = Company # No additional magic needed
```

**class JSONField** (*pickle=False*)

A field for storing arbitrary jsonifiable data. Set *pickle* to *True* to pickle anything non-jsonifiable.

**class PickleField**

A field for storing arbitrary picklable data.

**class AdditionalAutoField**

Additional autoincremented field which is not primary key.

Should be used with *AdditionalAutoFieldManager*:

```
from handy.models import AdditionalAutoField, AdditionalAutoFieldManager

class MyModel(models.Model):
    num = AdditionalAutoField()

    _base_manager = AdditionalAutoFieldManager()
```

**class BigAutoField**

An `AutoField` but uses `bigint`. If `external_sequence` argument is set to true then sequence is not created with field.

### **class StripWhitespace**

A middleware that strips whitespace from html responses to make them smaller. Doesn't strip newlines in order to not break any embedded javascript.

Just add handy `.middleware.StripWhitespace` to your `MIDDLEWARE_CLASSES`.



---

A couple of low-level utilities for those who tired of manually creating cursors.

**fetch\_all** (*sql*, *params=()*, *server='default'*)  
Execute given sql and return all resulting rows.

**fetch\_row** (*sql*, *params=()*, *server='default'*)  
Execute given sql and return one row.

**fetch\_col** (*sql*, *params=()*, *server='default'*)  
Execute given sql and return list of values in first result column.

**fetch\_val** (*sql*, *params=()*, *server='default'*)  
Execute given sql and return single resulting value:

```
last_id = fetch_val('select max(id) from some_table')
```

**do\_sql** (*sql*, *params=()*, *server='default'*)  
Execute given sql with given params.

**fetch\_dicts** (*sql*, *params=()*, *server='default'*)  
Same as *fetch\_all()*, but returns results as dicts.

**fetch\_dict** (*sql*, *params=()*, *server='default'*)  
Same as *fetch\_row()*, but returns result as dict.

**fetch\_named** (*sql*, *params=()*, *server='default'*)  
Same as *fetch\_all()*, but returns *namedtuples*.

**fetch\_named\_row** (*sql*, *params=()*, *server='default'*)  
Same as *fetch\_row()*, but returns *namedtuple*.

**queryset\_iterator** (*queryset*, *chunksize=1000*)  
Iterate over a Django Queryset ordered by the primary key.

This method loads a maximum of *chunksize* (default: 1000) rows in it's memory at the same time while django normally would load all rows in it's memory. It also bypasses django queryset cache.

Note that ordered querysets not supported.

**queryset\_chunks** (*queryset*, *chunksize=1000*)

Returns iterator yielding chunks of django queryset. Takes care not to load everything at once.



## CHAPTER 9

---

### Utilities

---

**get\_or\_none** (*Model*, *\*\*conds*)

Gets instance of `Model` class from database or returns `None`.

**obj\_dump** (*obj*)

**get\_module\_attr** (*path*)

---

**Note:** functional tools, namely `@memoize` and `@cache`, moved to `fancy`

---



## A

AdditionalAutoField (built-in class), 15

## B

BigAutoField (built-in class), 16

BigIntegerArrayField (built-in class), 15

## C

CommaSeparatedInput (built-in class), 13

## D

do\_sql() (built-in function), 19

## F

fetch\_all() (built-in function), 19

fetch\_col() (built-in function), 19

fetch\_dict() (built-in function), 19

fetch\_dicts() (built-in function), 19

fetch\_named() (built-in function), 19

fetch\_named\_row() (built-in function), 19

fetch\_row() (built-in function), 19

fetch\_val() (built-in function), 19

## G

get\_module\_attr() (built-in function), 21

get\_or\_none() (built-in function), 21

## I

IntegerArrayField (built-in class), 15

## J

JSONField (built-in class), 15

## L

last\_modified() (built-in function), 5

## M

mail\_admins() (built-in function), 11

MultilineInput (built-in class), 13

## O

obj\_dump() (built-in function), 21

## P

paginate() (built-in function), 3, 7

PickleField (built-in class), 15

## Q

queryset\_chunks() (built-in function), 19

queryset\_iterator() (built-in function), 19

## R

render\_to() (built-in function), 3

render\_to\_email() (built-in function), 11

render\_to\_json() (built-in function), 4

## S

SimpleWidget (built-in class), 13

StringArrayField (built-in class), 15

StripWhitespace (built-in class), 17