django-libs Documentation

Release 0.1

Martin Brochhaus

Sep 27, 2017

Contents

django-libs provides useful tools and helpers that can be used in almost all Django applications. By putting all these into one central package we hope to ease our maintenance work across multiple Django projects and to reduce tedious (and erroneous) copy and paste orgies.

Contents:

CHAPTER 1

Context Processors

analytics

Most projects have the Google Analytics tracking code in their base template. If you like to put that code into a partial template or even re-use your whole base template between projects, then it would be a good idea to set the analytics code in your `local_settings.py` and add it to your template context using this context processor.

Add the processor to your list of context processors:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    'django.contrib.auth.context_processors.auth',
    'django.core.context_processors.debug',
    'django.core.context_processors.i18n',
    'django.core.context_processors.media',
    'django.core.context_processors.static',
    'django.core.context_processors.tz',
    'django.contrib.messages.context_processors.messages',
    'django.core.context_processors.request',
    'django_libs.context_processors.analytics',
)
```

Use it in your template:

```
<script>
    var _gaq=[['_setAccount','{{ ANALYTICS_TRACKING_ID }}'],['_trackPageview']];
    (function(d,t){var g=d.createElement(t),s=d.getElementsByTagName(t)[0];
    g.src=('https:'==location.protocol?'//ssl':'//www')+'.google-analytics.com/ga.js';
    s.parentNode.insertBefore(g,s)}(document,'script'));
</script>
```

CHAPTER 2

Decorators

lockfile

Very useful for custom admin commands. If your command is scheduled every minute but might take longer than one minute, it would be good to prevent execution of the command if the preceeding execution is still running. Lockfiles come in handy here.

Note: This decorator requires the `lockfile` package to be installed. Either add it to your requirements if not already in or to get the latest version from pypi do:

```
pip install lockfile
```

You should create a setting `LOCKFILE_PATH` which points to `/home/username/tmp/`.

Usage:

```
from django_libs.decorators import lockfile
...

LOCKFILE = os.path.join(
    settings.LOCKFILE_PATH, 'command_name')

class Command(BaseCommand):

    @lockfile(LOCKFILE)
    def handle(self, *args, **kwargs):
        ...
```

CHAPTER 3

Factories

HvadFactoryMixin

Writing factories for models under django-hvad is a bit hard because for each object you also have to create a translation object. This mixin takes care of this. Simply inherit from this mixin and write your factory as if it was a normal model, but make sure to add a `language_code` field with the default language you would like to use:

```
import factory
from django_libs.tests.factories import HvadFactoryMixin

class NewsEntryFactory(HvadFactoryMixin, factory.DjangoModelFactory):
    FACTORY_FOR = NewsEntry

    language_code = 'en'  # This is important
    title = factory.Sequence(lambda x: 'A title {0}'.format(x))
    slug = factory.Sequence(lambda x: 'a-title-{0}'.format(x))
    is_published = True
```

SimpleTranslationMixin

Writing factories for models under simple-translation is a bit annoying because you always have to override the `_prepare` method in order to create the translation object for the main object that you want to create.

By using this mixin, we suggest the following pattern to ease the process of writing factories for translateable objects:

```
import factory
from django_libs.tests.factories import SimpleTranslationMixin

class PersonFactory(SimpleTranslationMixin, factory.Factory):
    """Factory for ``Person`` objects."""
    FACTORY_FOR = Person
```

```
    @staticmethod
    def _get_translation_factory_and_field():
        # 'person' is the name of the ForeignKey field on the translation
        # model
        return (PersonTranslationFactory, 'person')

class PersonTranslationFactory(factory.Factory):
    """Factory for ``PersonTranslation`` objects."""
    FACTORY_FOR = PersonTranslation

    first_name = 'First name'
    person = factory.SubFactory(PersonFactory)
    language = 'en'
```

This allows you to use the `PersonTranslationFactory` which will create a `Person` as well via the sub factory but sometimes it is just more intuitive to use the `Person` factory (because you want to test something on the `Person` model), so now you can do that as well and you will still get a translation object created in the background. This is often important because simple-translation relies on the fact that there is ALWAYS at least one translation available.

UserFactory

We use https://github.com/rbarrois/factory_boy to create fixtures for our tests. Since Django comes with a User model that is needed by almost all tests a `UserFactory` is useful for all Django projects.

Usage

Let's assume you want to write a view test and your view requires an authenticated user. You can create the user using the `UserFactory` like so:

```
from django.test import TestCase
from django_libs.tests.factories import UserFactory

class MyViewTest(TestCase):
    def setUp(self):
        self.user = UserFactory()

    def test_view(self):
        self.client.login(username=self.user.email, password='test123')
        resp = self.client.get('/')
        self.assertEqual(resp.status_code, 200)
```

UploadedImageFactory

Are you also tired of having to deal with images in upload form tests? Well here's help! With the `UploadedImageFactory` you can create a `SimpleUploadedFile` with just one line of code.

Example:

..code-block:: python

    # Say your form has an image field from django import forms

    MyImageForm(forms.Form): avatar = forms.ImageField(...) ... # other fields

```
# Then you want to test this, so in your test case you do from django.test import TestCase
from django_libs.tests.factories import UploadedImageFactory
from ..forms import MyForm
class MyImageFormTestCase(TestCase):
    def test_form(self):  files = {'avatar': UploadedImageFactory()} data = {
            ... # other data
        } form = MyForm(data=data, files=files) self.assertTrue(form.is_valid())
```

CHAPTER 4

Forms

PlaceholderForm

Simple form mixin, which uses the field label as a placeholder attribute. E.g.:

first_name = forms.CharField(label=_('Name'))

will be rendered as:

<input id="id_first_name" name="first_name" placeholder="Name" type="text">

StripTagsFormMixin

A mixin that allows to mark certain fields to not allow any HTML tags.

Then the form gets initiated and data is given, all HTML tags will be stripped away from that data.

Usage:

```
class MyForm(StripTagsFormMixin, forms.Form):
    text = forms.CharField(max_length=1000)

    STRIP_TAGS_FIELDS = ['text', ]

    def __init__(self, *args, **kwargs):
        super(MyForm, self).__init__(*args, **kwargs)
        self.strip_tags()
```

1. Inherit from *StripTagsFormMixin*
2. Add *STRIP_TAGS_FIELDS* attribute to your form class
3. Override *__init__* and call *self.strip_tags()* after your super call

CHAPTER 5

JavaScript

getcookie.js

Provides the function `getCookie` which allows to retrieves values from the cookie. This is especially useful when you want to do a POST request via AJAX.

Usage:

```
<script src="{{ STATIC_URL }}django_libs/js/getcookie.js"></script>
<script>
    var data = [{name: 'csrfmiddlewaretoken', value: getCookie('csrftoken')}];
    $.post(
        '/some/url/'
        ,data
        ,function(data) {
            if (data == 'success') {
                // do something
            }
        }
    );
</script>
```

modals.js

Provides functions to easily get or post requests that should be shown in a Twitter Bootstrap modal. In order to use this:

1. Make sure that you are using bootstrap-modal
2. Make sure that you are using the AjaxRedirectMiddleware
3. Add * * at the end of your `base.html`

Now you could place a button somewhere in your code and use the `onclick` event to open the modal. You can pass in the URL that serves the modal's template and extra context that should be sent in the request as GET data:

```
<a href="#" onclick="getModal('/ajax-url/', {next: '/profile/'}); return false;">Open␣
↪Modal</a>
```

In your modal you might have a form with a submit button. You can now trigger the POST request like so:

```
// This is how your modal template should look like
<form id="formID" method="post" action="/ajax-url/">
    {% csrf_token %}
    <div class="modal-body">
        <fieldset>
            {% for field in form %}
                {% include "partials/form_field.html" %}
            {% endfor %}
        </fieldset>
        <input type="hidden" name="next" value="{% if next %}{{ next }}{% endif %}" />
    </div>
    <div class="modal-footer">
        <input type="button" onclick="postModal('/ajax-url/', $('#formID')); return␣
↪false;" value="Submit">
    </div>
</form>
```

CHAPTER 6

Template Tags

add_form_widget_attr

Adds widget attributes to a bound form field.

This is helpful if you would like to add a certain class to all your forms (i.e. *form-control* to all form fields when you are using Bootstrap):

```
{% load libs_tags %}
{% for field in form.fields %}
    {% add_form_widget_attr field 'class' 'form-control' as field_ %}
    {{ field_ }}
{% endfor %}
```

The tag will check if the attr already exists and only append your value. If you would like to replace existing attrs, set *replace=1*:

```
{% add_form_widget_attr field 'class' 'form-control' replace=1 as field_ %}
```

block_anyfilter

Turns any template filter into a blocktag.

Usage:

```
{% load libs_tags %}
{% block_anyfilter django.template.defaultfilters.truncatewords_html 15 %}
    // Something complex that generates html output
{% endblockanyfilter %}
```

This is useful when you are working with django-cms' *render_placeholder* tag, for example. That tag is unfortunately not an assignment tag, therefore you can't really do anything with the output. Imagine you want to show a list of latest news and for each news a little excerpt based on the content placeholder. That placeholder could contain anything,

like images and *h1* tags but you really just want to show the first ten words without images or any styles. Now you can do this:

{% block_anyfilter django.template.defaultfilters.truncatewords_html 15 %}

{% block_anyfilter django.template.defaultfilters.striptags %} {% render_placeholder news_entry.content %}

{% endblockanyfilter %}

{% endblockanyfilter %}

block_truncatewords_html

Allows to truncate any block of content. Calls Django's `truncatewords_html` internally.

This is useful when rendering other tags that generate content, such as django-cms' `render_placeholder` tag, which is not available as an assignment tag:

```
{% load libs_tags %}
{% block_truncatewords_html 15 %}
    {% render_placeholder object.placeholder %}
{% endblocktruncatewordshtml %}
```

The first parameter is the number of words you would like to truncate after.

calculate_dimensions

`calculate_dimensions` is a way to auto-correct thumbnail dimensions depending on the images format. The required args are am image instance, the length of the long image side and finally the length of the short image side.

Usage Example with easy_thumbnails:

```
{% load libs_tags thumbnail %}
{% calculate_dimensions image 320 240 as dimensions%}
<img src="{% thumbnail image dimensions %}" />
```

It then ouputs `320x240` if the image is landscape and `240x320` if the image is portait.

call

`call` is an assignemnt tag that allows you to call any method of any object with args and kwargs, because you do it in Python all the time and you hate not to be able to do it in Django templates.

Usage:

```
{% load libs_tags %}
{% call myobj 'mymethod' myvar foobar=myvar2 as result %}
{% call myobj 'mydict' 'mykey' as result %}
{% call myobj 'myattribute' as result %}
{{ result }}
```

concatenate

Concatenates the given strings.

Usage:

```
{% load libs_tags %}
{% concatenate "foo" "bar" as new_string %}
{% concatenate "foo" "bar" divider="_" as another_string %}
```

The above would result in the strings “foobar” and “foo_bar”.

exclude

`exclude` is a filter tag that allows you to exclude one queryset from another.

Usage:

```
{% load libs_tags %}
{% for clean_obj in qs|exclude:dirty_qs %}
    {{ clean_obj }}
{% endfor %}
```

get_content_type

`get_content_type` is a simple template filter to return the content type of an object or to return one of the content type’s fields.

This might be very useful if you want to e.g. call a URL, which needs a content object as a keyword argument.

In order to use it, just import the tag library and set the tag:

```
{% load libs_tags %}
<a href="{% url "review_content_object" content_type=user|get_content_type:'model'␣
↪object_id=user.pk %}">Review this user!</a>
```

As you can see, you can provide a field argument to return the relevant content type’s field.

get_form_field_type

Returns the widget type of the given form field.

This can be helpful if you want to render form fields in your own way (i.e. following Bootstrap standards).

Usage:

```
{% load libs_tags %}
{% for field in form %}
    {% get_form_field_type field as field_type %}
    {% if "CheckboxInput" in field_type %}
        <div class="checkbox">
            <label>
                // render input here
```

```
            </label>
        </div>
    {% else %}
        {{ field }}
    {% endif %}
{% endfor %}
```

get_range

`get_range` behaves just like Python's `range` function and allows you to iterate over ranges in your templates:

```
{% load libs_tags %}
{% for item in 5|get_range %}
    Item number {{ item }}
{% endfor %}
```

You can also calculate the difference between your value and a max value. This is useful if you want to fill up empty space with items so that the total amount of items is always `max_num`:

```
{% load libs_tags %}
{% for item in object_list.count|get_range %}
    // render the actual items
{% endfor %}
{% for item in object_list.count|get_range:10 %}
    // render the placeholder items to fill up the space
{% endfor %}
```

get_range_around

Returns a range of numbers around the given number.

This is useful for pagination, where you might want to show something like this:

```
<< < ... 4 5 (6) 7 8 .. > >>
```

In this example *6* would be the current page and we show 2 items left and right of that page.

Usage:

```
{% load libs_tags %}
{% get_range_around page_obj.paginator.num_pages page_obj.number 2 as pages %}
```

The parameters are:

1. range_amount: Number of total items in your range (1 indexed)
2. The item around which the result should be centered (1 indexed)
3. Number of items to show left and right from the current item.

get_verbose

`get_verbose` is a simple template tag to provide the verbose name of an object's specific field.

This can be useful when you are creating a `DetailView` for an object where, for some reason you don't want to use a ModelForm. Instead of using the `{% trans %}` tag to create your labels and headlines that are related to the object's fields, you can now obey the DRY principle and re-use the translations that you have already done on the model's field's `verbose_name` attributes.

In order to use it, just import the tag library and set the tag:

```
{% load libs_tags %}
<ul>
    <li>
        <span>{{ news|get_verbose:"date" }}</span>
    </li>
    <li>
        <span>{{ news|get_verbose:"title" }}</span>
    </li>
</ul>
```

get_query_params

Allows to change (or add) one of the URL get parameter while keeping all the others.

Usage:

```
{% load libs_tags %}
{% get_query_params request "page" page_obj.next_page_number as query %}
<a href="?{{ query }}">Next</a>
```

You can also pass in several pairs of keys and values:

```
{% get_query_params request "page" 1 "foobar" 2 as query %}
```

You often need this when you have a paginated set of objects with filters.

Your url would look something like `/?region=1&gender=m`. Your paginator needs to create links with `&page=2` in them but you must keep the filter values when switching pages.

If you want to remove a special parameter, you can do that by setting it's value to `!remove`:

```
{% get_query_params request "page" 1 "foobar" "!remove" as query %}
```

is_context_variable

Checks if a given variable name is already part of the template context.

This is useful if you have an expensive templatetag which might or might not have been called in a parent template and you also need it in some child templates.

You cannot just check for `{% if variable_name %}` because that would equal to `False` in all cases:

- if the variable does not exist
- if the variable exists but is `None`
- if the variable exists but is `False`
- if the variable exists but is 0

This tag allows you to do something like this:

```
{% is_context_variable 'variable_name' as variable_exists %}
{% if not variable_exists %}
    {% expensive_templatetag as variable_name %}
{% endif %}
{{ variable_name }}
```

load_context

`load_context` allows you to load any python module and add all it's attributes to the current template's context. This is very useful for the RapidPrototypingView, for example. You would be able to create the template without having any view providing a useful context (because the view might not exist, yet). But as a template designer you might already know that the view will definitely return a list of objects and that list will be called `objects` and each object will have a `name` attribute.

Here is how you would use it:

- create a file `yourproject/context/__init__.py`
- create a file `yourproject/context/home.py`. A good convention would be to name these context modules just like you would name your templates.

Now create the context that you would like to use in your `home.html` template:

```
# in object_list.py:
objects = [
    {'name': 'Object 1', },
    {'name': 'Object 2', },
]
```

Now create your template:

```
# in home.html
{% load libs_tags %}
{% load_context "myproject.context.home" %}

{% for object in objects %}
    <h1>{{ object.name }}</h1>
{% endfor %}
```

This should allow your designers to create templates long before the developers have finished the views.

navactive

`navactive` is a simple template tag to provide the string `active` if the current URL is in the desired url path.

In order to use it, just import the tag library and set the tag, e.g. as a css class:

```
{% load libs_tags %}
<ul class="nav">
    <li class="{% navactive request "/news/" exact=1 %}">
        <a href="{% url "news_list" %}">{% trans "News" %}</a>
    </li>
    <li class="{% navactive request "/news/" %}">
```

```
        <a href="{% url "news_detail" pk=latest.pk %}">{% trans "Latest News Entry" %}
↪</a>
    </li>
</ul>
```

render_analytics_code

`render_analytics_code` is an inclusion tag to render Google's analytics script code.

Usage:

```
{% load libs_tags %}
{% render_analytics_code %}
```

or (if you don't want to use the `anonymizeIp` setting):

```
{% load libs_tags %}
...
<head>
...
{% render_analytics_code False %}
</head>
```

If you would like to override the template used by the tag, please use `django_libs/analytics.html`.

render_analytics2_code

The same as `render_analytics_code` but uses the new syntax and always uses anonymize IP.

Usage:

```
{% load libs_tags %}
...
<head>
...
{% render_analytics2_code %}
</head>
```

save

`save` allows you to save any variable to the context. This can be useful when you have a template where different sections are rendered depending on complex conditions. If you want to render *<hr />* tags between those sections, it can be quite difficult to figure out when to render the divider and when not.

Usage:

```
{% load libs_tags %}
...
{% if complex_condition1 %}
    // Render block 1
    {% save "NEEDS_HR" 1 %}
```

```
{% endif %}

{% if complex_condition2 %}
    {% if NEEDS_HR %}
        <hr />
        {% save "NEEDS_HR" 0 %}
    {% endif %}
    // Render block 2
    {% save "NEEDS_HR" 1 %}
{% endif %}
```

When you have to render lots of divicers, the above example can become more elegant when you replace the *if NEEDS_HR* block with:

```
{% include "django_libs/partials/dynamic_hr.html" %}
```

sum

Adds the given value to the total value currently held in *key*.

Use the multiplier if you want to turn a positive value into a negative and actually substract from the current total sum.

Usage:

```
{% sum "MY_TOTAL" 42 -1 %}
{{ MY_TOTAL }}
```

set_context

NOTE: It turns out that this implementation only saves to the current template's context. If you use this in a sub-template, it will not be available in the parent template. Use our `save` tag for manipulating the global RequestContext.

`set_context` allows you to put any variable into the context. This can be useful when you are creating prototype templates where you don't have the full template context, yet but you already know that certain variables will be available later:

```
{% load libs_tags %}
{% set_context '/dummy-url/' as contact_url %}
{% blocktrans with contact_url=contact_url %}
Please don't hesitate to <a href="{{ contact_url }}">contact us</a>.
{% endblocktrans %}
```

verbatim

`verbatim` is a tag to render x-tmpl templates in Django templates without losing the code structure.

Usage:

```
{% load libs_tags %}
{% verbatim %}
{% if test1 %}
```

```
    {% test1 %}
{% endif %}
{{ test2 }}
{% endverbatim %}
```

The output will be:

```
{% if test1 %}
    {% test1 %}
{% endif %}
{{ test2 }}
```

CHAPTER 7

Loaders

This module provides a few simple utility functions for loading classes from strings like `myproject.models.FooBar`.

load_member_from_setting

Use this function to load a member from a setting:

```
# in your settings.py:
FOOBAR_CLASS = 'myproject.models.FooBar'

# anywhere in your code:
from django_libs.loaders import load_member_from_setting
cls = load_member_form_setting('FOOBAR_CLASS')
```

If you are using the reusable app settings pattern, you can hand in an optional parameter which should be the `app_settings` module where you define your app's setting's default values:

```
# in your app_settings.py:
from django.conf import settings

FOOBAR_CLASS = getattr(settings, 'APPNAME_FOOBAR_CLASS', 'appname.models.FooBar')

# anywhere in your code:
from appname import app_settings
from django_libs.loaders import load_member_from_setting

cls = load_member_from_setting('FOOBAR_CLASS', app_settings)
```

load_member

This function is used by `load_member_from_setting` internally. Use this if you already have the FQN string of the member you would like to load:

```
# anywhere in your code:
from django_libs.loaders import load_member

cls = load_member('myproject.models.FooBar')
```

split_fqn

This function is used by `load_member` internally. Use this if you want to get the left and right side of a fully qualified name:

```
# anywhere in your code:
from django_libs.loaders import split_fqn

modulename, classname = split_fqn('myproject.models.FooBar')
```

In this example, `modulename` would be `myproject.models` and `classname` would be `FooBar`.

If you need it more dynamically, you can also pass in a function that returns a fqn string:

```
# anywhere in your code
from django_libs.loaders import split_fqn

def function_that_returns_fqn_string():
    return 'myproject.models.FooBar'

modulename, classname = split_fqn(function_that_returns_fqn_string)
```

CHAPTER 8

Management Commands

cleanup_mailer_messagelog

If you want to delete old message logs of the `mailer` app simple use:

```
./manage.py cleanup_mailer_messagelog
```

You can also add the command to your cronjobs:

```
0 4 * * 4 $HOME/bin/django-cleanup-mailer-messagelog.sh > $HOME/mylogs/cron/django-cleanup-mailer-messagelog.log 2>&1
```

Logs younger than 122 days (~4 months) will be ignored, logs older than 122 days will be deleted.

CHAPTER 9

Middlewares

AjaxRedirectMiddleware

When calling a view from an AJAX call and when that view returns a redirect, jQuery changes the status code to 200. This means, in your success callback you will not be able to determine, if the view returned to 200 or a redirect.

Interestingly, there is a workaround: If we return some made up status code, jQuery will not change it.

This middleware makes sure that, if there was a redirect and if it was an AJAX call, the return code will be set to `278`.

In order to use this middleware, add it to your `MIDDLEWARE_CLASSES` setting:

```
MIDDLEWARE_CLASSES = [
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    ...
    'django_libs.middleware.AjaxRedirectMiddleware',
]
```

In your jQuery script you can now react to redirects:

```
$.post(url, data, function(data, textStatus, jqXHR) {
    if (jqXHR.status == 278) {
        window.location.href = jqXHR.getResponseHeader("Location");
    } else {
        $("#" + container).replaceWith(data);
    }
});
```

When you are using this middleware, it means that Redirects will no longer be executed on the server and your AJAX function has to call the redirect URL manually. If you really want to get the HTML that the last view in the redirect chain would return, you can disable this middleware for some requests by adding *ajax_redirect_passthrough* parameter to your data payload. When this parameter is given, the middleware will be skipped:

```
<form method="post" action=".">
    <input type="hidden" name="ajax_redirect_passthrough" value="1" />
    ...
</form>
```

CustomBrokenLinkEmailsMiddleware

Use this instead of the default *BrokenLinkEmailsMiddleware* in order to see the current user in the email body. Use this with Django 1.6. Earlier versions should use the *CustomCommonMiddleware* instead.

CustomCommonMiddleware

Use this instead of the default *CommonMiddleware* in order to see the current user in the email body. Use this with Django 1.5. Later versions should use the *CustomBrokenLinkEmailsMiddleware* instead.

CustomSentry404CatchMiddleware

Make sure to add *user-agents* to your `requirements.txt`.

Use this instead of the middleware provided by raven. It allows you to define a list of regex strings in the setting `RAVEN_IGNORABLE_USER_AGENTS`. This way you can stop reporting for nasty spiders crawling your site and testing all kinds of weird non-existant URLs.

This middleware also uses user-agents to parse the user agent string and determine the device. If the device family equals `Spider`, the request will be ignored. If you don't want to block all spiders like this, set `RAVEN_IGNORE_SPIDERS = False`.

ErrorMiddleware

Add this middleware if you would like to see the user's email address in the traceback that is sent to you when a 500 error happens.

In order to use this middleware, add it to your `MIDDLEWARE_CLASSES` setting:

```
MIDDLEWARE_CLASSES = [
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    ...
    'django_libs.middleware.ErrorMiddleware',
]
```

SSLMiddleware

Add this middleware as the first middleware in your stack to forward all requests to *http://yoursite.com* to *https://yoursite.com*. Define exceptions via the setting *NO_SSL_URLS* - these requests will be served without HTTPS.

CHAPTER 10

Models Mixins

SimpleTranslationMixin

We ended up adding a `get_translation` method to all models that have a corresponding translation model, so this mixin will add common methods needed by models under simple-translation.

You can use this by inheriting the class:

```
from django.db import models
from django_libs.models_mixins import SimpleTranslationMixin

class MyModel(SimpleTranslationMixin, models.Model):
    pass

class MyModelTranslation(models.Model):
    name = models.CharField(max_length=256)

    # needed by simple-translation
    my_model = models.ForeignKey(MyModel)
    language = models.CharField(max_length=16)
```

get_translation

The method takes an optional parameter `language` if you want to get a specific translation, otherwise it will return the translation for the currently active language:

```
myobject = MyModel.objects.get(pk=1)
trans = myobject.get_translation()
print trans.name
```

SimpleTranslationPublishedManager

When your model `MyModel` has a corresponding `MyModelTranslation` model, like above, you can use the `SimpleTranslationPublishedManager` to filter for published versions of `MyModel`.

To set it up simply add the manager to your model:

```
from django_libs.models_mixins import SimpleTranslationPublishedManager

class MyModel(models.Model):

    # some fields

    objects = SimpleTranslationPublishedManager()


class MyModelTranslation(models.Model):

    # these fields are required
    is_published = models.BooleanField()
    language = models.CharField(max_length=16)
```

If you want to alter the required fields, because your models differ from our assumed structure or you want to build it into your own custom manager, you can inherit the `SimpleTranslationPublishedManager` like so:

```
from django_libs.models_mixins import SimpleTranslationPublishedManager

class MyModelManager(SimpleTranslationPublishedManager)
    # set these values, if your translation model or the fields differ from
    # the defaults
    published_field = `mymodeltranslation__published_it_is`
    language_field = `mymodeltranslation__language_used`
```

and then add the new `MyModelManager` to `MyModel` like we did above.

Usage

When you set it up correctly, you can use the manager like so:

```
MyModel.objects.published(request)
```

Note, that you need to pass a request instance, so the manager can fetch the language.

CHAPTER 11

Storage support

Amazon S3

If you want to store your media files in an Amazon S3 bucket we provide some helpful files for you.

First of all, setup your Amazon stuff. This article will help you out:

> http://martinbrochhaus.com/s3.html

Then install `django-storages` (http://django-storages.readthedocs.org/) and `boto` (https://github.com/boto/boto). Add the following code to your `local_settings.py`:

```
USE_S3 = False
AWS_ACCESS_KEY = 'XXXX'
AWS_SECRET_ACCESS_KEY = 'XXXX'
AWS_STORAGE_BUCKET_NAME = 'bucketname'
AWS_QUERYSTRING_AUTH = False
S3_URL = 'https://%s.s3.amazonaws.com' % AWS_STORAGE_BUCKET_NAME

if USE_S3:
    DEFAULT_FILE_STORAGE = 'django_libs.s3.MediaRootS3BotoStorage'
    THUMBNAIL_DEFAULT_STORAGE = DEFAULT_FILE_STORAGE
    MEDIA_URL = S3_URL + '/media/'

    # Add this line, if you're using ``django-compressor``
    COMPRESS_STORAGE = 'django_libs.s3.CompressorS3BotoStorage'

MEDIA_ROOT = os.path.join(PROJECT_ROOT, '../..',  'media')
STATIC_ROOT = os.path.join(PROJECT_ROOT, '../..', 'static')
```

Test the upload. If you get a `NoAuthHandlerFound` exception, add the following lines to `$HOME/.boto`:

```
[Credentials]
aws_access_key_id = XXXX
aws_secret_access_key = XXXX
```

If you're using `django-compressor` add the following settings:

```
COMPRESS_PARSER = 'compressor.parser.HtmlParser'
COMPRESS_CSS_FILTERS = [
    'django_libs.compress_filters.S3CssAbsoluteFilter',
]
COMPRESS_ENABLED = True
```

Make sure to run `./manage.py compress --force` on every deployment. Also check:

```
http://martinbrochhaus.com/compressor.html
```

CHAPTER 12

Test Email Backend

EmailBackend

`EmailBackend` is a simple Email backend, that sends all emails to a defined address, no matter what the recipient really is.

This is useful in times of development & testing to prevent mass mails to example.com or existing addresses and to review all email communication.

In order to use it, set this in your local_settings.py:

```
EMAIL_BACKEND = 'django_libs.test_email_backend.EmailBackend'
TEST_EMAIL_BACKEND_RECIPIENTS = (
    ('Name', 'email@gmail.com'),
)
```

If you're using django-mailer don't forget to add:

```
MAILER_EMAIL_BACKEND = 'django_libs.test_email_backend.EmailBackend'
```

WhitelistEmailBackend

`WhitelistEmailBackend` provides more control over what can be sent where.

To use it, first define the `EMAIL_BACKEND_WHITELIST` setting::

```
EMAIL_BACKEND_WHITELIST = [r'.*@example.com']
```

This setting holds regex patterns which define, which emails may be sent and which are being discarded. The above example will allow every email adress from the `example.com` domain to be delivered.

If you still want to receive all the discarded emails, you can additionally define TEST_EMAIL_BACKEND_RECIPIENTS like above and set EMAIL_BACKEND_REROUTE_BLACKLIST to True:

```
EMAIL_BACKEND_REROUTE_BLACKLIST = True
TEST_EMAIL_BACKEND_RECIPIENTS = (
    ('Name', 'email@gmail.com'),
)
```

With this setup, all recipients, that match one of the whitelisted email patterns will be sent to the correct recipient, but in case it didn't match, the recipients will be replaced with the ones from the TEST_EMAIL_BACKEND_RECIPIENTS setting.

CHAPTER 13

Test Mixins

Usage

In order to use the `ViewTestMixin` you need to import it and implement a few methods on your test case. A typical test case looks like this:

```
from django.test import TestCase
from django_libs.tests.mixins import ViewTestMixin
from your_invoice_app.tests.factories import InvoiceFactory

class InvoiceDetailViewTestCase(ViewTestMixin, TestCase):
    """Tests for the ``InvoiceDetailView`` generic class based view."""
    def setUp(self):
        self.invoice = InvoiceFactory()
        self.user = self.invoice.user

    def get_view_name(self):
        return 'invoice_detail'

    def get_view_kwargs(self):
        return {'pk': self.invoice.pk}

    def test_view(self):
        self.should_redirect_to_login_when_anonymous()
        self.should_be_callable_when_authenticated(self.user)
        # your own tests here
```

For a slightly longer explanation on why the test looks like this, please read on...

Tutorial

It is a good idea to write a test that calls your view before you actually write the view. And when you are at it, you might just as well test if a view that is protected by `login_required`, actually *does* require the user to be logged

in. Walking down that road, you might also just as well try to call the view and manipulate the URL so that this user tries to access another user's objects. And so on, and so forth...

Fact is: You will be calling `self.client.get` and `self.client.post` a lot in your integration tests (don't confuse these tests with your unit tests).

Let's assume that you have defined your `urls.py` like this:

```
...
url(r'^invoice/(?P<pk>\d+)/', InvoiceDetailView.as_view(), name='invoice_detail'),
...
```

In order to test such a view, you would create an `integration_tests/views_tests.py` file and create a test case for this view:

```
from django.test import TestCase

class InvoiceDetailViewTestCase(TestCase):
    def test_view(self):
        resp = self.client.get('/invoice/1/')
```

Writing the test this way is flawed because if you ever change that URL your test will fail. It would be much better to use the view name instead:

```
from django.core.urlresolvers import reverse
...
class InvoiceDetailViewTestCase(TestCase):
    def test_view(self):
        resp = self.client.get(reverse('invoice_detail'))
```

If your view is just slightly complex, you will have to call `self.client.get` several times and it is probably not a good idea to repeat the string `invoice_detail` over and over again, because that might change as well. So let's centralize the view name:

```
class InvoiceDetailViewTestCase(TestCase):
    def get_view_name(self):
        return 'invoice_detail'

    def test_view(self):
        resp = self.client.get(reverse(self.get_view_name()))
```

The code above was simplified. The `reverse` calls would fail because the view actually needs some kwargs. A proper call would look like this:

```
invoice = InvoiceFactory()
resp = self.client.get(reverse(self.get_view_name(), kwargs={
    'pk': invoice.pk}))
```

This can get annoying when you need to call the view many times because most of the time you might call the view with the same kwargs. So let's centralize the kwargs as well:

```
class InvoiceDetailViewTestCase(TestCase):
    def setUp(self):
        self.invoice = InvoiceFactory()

    def get_view_name(self):
        ...
```

```
    def get_view_kwargs(self):
        return {'pk': self.invoice.pk}

    def test_view(self):
        resp = self.client.get(reverse(self.get_view_name(),
            self.get_view_kwargs()))
```

This is much better. Someone who looks at your test, can easily identify the view name and the expected view kwargs that are needed to get a positive response from the view. When writing tests you don't have to think about the view name or about constructing the view kwargs any more, which will speed up your workflow.

But this is still an awful lot of code to type. Which is why we created the ViewTestMixin:

```
class InvoiceDetailViewTestCase(ViewTestMixin, TestCase):
    def setUp(self):
        ...

    def get_view_name(self):
        ...

    def get_view_kwargs(self):
        ...

    def test_view(self):
        resp = self.client.get(self.get_url())
```

Now we have got it down to a one-liner to call `self.client.get` in a future proof and maintainable way. After writing a few hundred tests with this approach new patterns emerge. You will want to test almost all views if they are accessible by anonymous or the opposite: If they are *not* accessible by anonymous but by a logged in user.

For this reason the `ViewTestMixin` provides a few convenience methods:

```
class InvoiceDetailViewTestCase(ViewTestMixin, TestCase):
    ...
    def test_view(self):
        user = UserFactory()
        self.should_redirect_to_login_view_when_anonymous()
        self.should_be_callable_when_authenticated(user)
```

If your view expectes some data payload (either POST or GET data), then you can set `self.data_payload` in your test. If all your tests need the same data, you can override the `get_data_payload()` method:

```
class InvoiceDetailViewTestCase(ViewTestMixin, TestCase):
    ...
    def get_data_payload(self):
        # If you stick to this implementation, you can still change the
        # data payload for ``some`` of your tests.
        if hasattr(self, 'data_payload'):
            return self.data_payload
        return {'foo': 'bar', }

    def test_view(self):
        user = UserFactory()
        self.should_redirect_to_login_view_when_anonymous()

        # Now your view will be called with the given data payload
        self.should_be_callable_when_authenticated(user)
```

```
        self.data_payload = {'foobar': 'barfoo'}
        # Now you have changed the standard payload to be returned by
        # ``get_data_payload``
        self.should_be_callable_when_authenticated(user)
```

"is_callable" and "is_not_callable"

If a view becomes more complex, you might end up with rather many assertions for many different situations. If you take all these cases into account when testing, which you probably should, you will write a lot of:

```
def test_view(self):
    # case 1
    resp = self.client.get(self.get_url())
    self.assertEqual(resp.status_code, 200, msg=(
        'If this then that, because foo is bar.'))
    # case 2
    resp = ...
    self.assertEqual(...)
    # case 3
    ...
```

`is_callable` and `is_not_callable` let you quickly assign different values to customize your actual assertion case in one method call. `is_callable` by default makes an assertion on status code 200. `is_not_callable` defaults to an assertion on status code 404.

Warning: Note if you used previous versions, that `is_callable` will only default to 200 in the future. It's best to use `and_redirects_to` for a redirect assertion or if you only want to make sure to get the right code set `status_code` to 302.

Also the `code` parameter changed into `status_code`.

They can still be used, but you will get annoying warnings. So, you might as well change it right away.

Argument	Definition
`method`	String that defines if either 'post' or 'get' is used.
`data`	dictionary with GET data payload or POST data. If not provided it calls `self.get_data_payload()` instead.
`kwargs`	dictionary to overwrite view kwargs. If not provided, it calls `self.get_view_kwargs()` instead.
`user`	Assign a user instance to log this user in first. As in `self.should_be_callable_when_authenticated()` the password is expected to be 'test123'.
`anonymous`	If this is assigned `True`, the user is logged out before the assertion. So basically you test with an anonymous user. Default is `False`.
`and_redirects_to`	If set, it performs an `assertRedirects` assertion. Note that, of course this will overwrite the `status_code` to 302.
`status_code`	If set, it overrides the status code, the assertion is made with.
`ajax`	If `True` it will automatically set `HTTP_X_REQUESTED_WITH='XMLHttpRequest'` to simulate an ajax call. Defaults to `False`.

You can also define no arguments to test according to your current situation. Then still, it is a handy shortcut.

Further methods are:

- should_be_callable_when_anonymous
- should_be_callable_when_has_correct_permissions

Have a look at the docstrings in the code for further explanations: https://github.com/bitmazk/django-libs/blob/master/django_libs/tests/mixins.py

CHAPTER 14

Testrunner

We like to combine django-nose (https://github.com/jbalogh/django-nose) and django-coverage (https://github.com/kmike/django-coverage) to create a custom testrunner that uses Nose for test file discovery and generates a coverage report on each test run.

The reason why we use nose is that we can easily exclude folders containing test files, for example slow integration tests.

Installation

Create a `test_settings.py` in your project with the following code:

```
from myproject.settings import *
from django_libs.settings.test_settings import *
```

We assume that you are using the new project layout that comes with Django 1.4 where your `settings.py` lives inside the `myproject` folder.

In order for this to work you should split up your `INSTALLED_APPS` setting into several lists. This allows us to tell coverage to ignore all external apps in the coverage report because we don't run tests of external apps:

```
INTERNAL_APPS = [
    'myproject',
    'foobar',
]

EXTERNAL_APPS = [
    'cms',
    'shop',
]

DJANGO_APPS = [
    ...
]
```

```
INSTALLED_APPS = DJANGO_APPS + EXTERNAL_APPS + INTERNAL_APPS
```

You will probably want to add `coverage/` to your `.gitignore` file.

Usage

Run your tests with your new `test_settings.py`:

```
./manage.py test -v 2 --traceback --failfast --settings=myproject.test_settings
```

Or if you want to exclude integration tests:

```
./manage.py test -v 2 --traceback --failfast --settings=myproject.test_settings --
↪exclude='integration_tests'
```

You will probably want to write a Fabric task `fab test` to make this call a bit more convenient.

CHAPTER 15

Utils

conditional_decorator

Allows you to decorate a function based on a condition.

This can be useful if you want to require login for a view only if a certain setting is set:

```
from django.conf import settings
from django.contrib.auth.decorators import login_required
from django.utils.decorators import method_decorator
from django_libs.utils import conditional_decorator
class MyView(ListView):
    @conditional_decorator(method_decorator(login_required), settings.LOGIN_REQUIRED)
    def dispatch(self, request, *args, **kwargs):
        return super(MyView, self).dispatch(request, *args, **kwargs)
```

create_random_string

Returns a random string. By default it returns 7 unambiguous capital letters and numbers, without any repetitions:

```
from django_libs.utils import create_random_string

result = create_random_string()
```

Will return something like `CH178AS`. You can set a length, characters to use and you can allow repetitions:

```
result = create_random_string(length=3, chars='abc123', repetitions=True)
```

html_to_plain_text

Converts html code into formatted plain text.

Use it to e.g. provide an additional plain text email.

Just feed it with some html:

```
from django_libs.utils import html_to_plain_text

html = (
    """
    <html>
            <head></head>
            <body>
                <ul>
                    <li>List element</li>
                    <li>List element</li>
                    <li>List element</li>
                </ul>
            </body>
        </html>
    """
)
plain_text = html_to_plain_text(html)
```

This will result in:

```
* List element
* List element
* List element
```

You can also feed the parser with a file:

```
from django_libs.utils import html_to_plain_text

with open('test_app/templates/html_email.html', 'rb') as file:
    plain_text = html_to_plain_text(file)
```

You can customize this parser by overriding its settings:

HTML2PLAINTEXT_IGNORED_ELEMENTS

Default: ['html', 'head', 'style', 'meta', 'title', 'img']

Put any tags in here, which should be ignored in the converting process.

HTML2PLAINTEXT_NEWLINE_BEFORE_ELEMENTS

Default: ['h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'div', 'p', 'li']

Put any tags in here, which should get a linebreak in front of their content.

HTML2PLAINTEXT_NEWLINE_AFTER_ELEMENTS

Default: ['h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'div', 'p', 'td']

Put any tags in here, which should get a linebreak at the end of their content.

HTML2PLAINTEXT_STROKE_BEFORE_ELEMENTS

Default: ['tr']

Put any tags in here, which should get a stroke in front of their content.

HTML2PLAINTEXT_STROKE_AFTER_ELEMENTS

Default: ['tr']

Put any tags in here, which should get a stroke at the end of their content.

HTML2PLAINTEXT_STROKE_TEXT

Default: '————————————————n'

You can override the appearance of a stroke.

CHAPTER 16

Utils Email

send_email

`send_email` sends html emails via django-mailer based on templates for subject and body.

Argument	Definition
`request`	The current request instance.
`extra_context`	A dictionary of items that should be added to the templates' contexts
`subject_template`	A string representing the path to the template of of the email's subject.
`body_template`	A string representing the path to the template of the email's body.
`from_email`	String that represents the sender of the email.
`recipients`	A tuple of recipients.

In order to use it, include the following code:

```
send_email(
    request={},
    extra_context={'Foo': bar},
    subject_template='email/notification_subject.html',
    body_template='email/notification_body.html',
    from_email=('Name', 'email@gmail.com'),
    recipients=[self.user.email, ]
)
```

CHAPTER 17

Views

Http404TestView & Http500TestView

Warning: These views are deprecated. Use the `RapidPrototypingView` instead.

Simple template views that use the `404.html` and `500.html` template. Just create this template in your project's *templates'* folder and add the views to your `urls.py`:

```
from django_libs.views import Http404TestView, Http500TestView
urlpatterns += patterns(
    '',
    url(r'^404/$', Http404TestView.as_view()),
    url(r'^500/$', Http500TestView.as_view()),
    ...
)
```

HybridView

You often need to display a different home page for authenticated users. For example Facebook shows a login page when you visit their site but when you are logged in it shows your stream under the same URL.

This `HybridView` does the same thing. Here is how you use it in your `urls.py`:

```
from django_libs.views import HybridView
from myapp.views import View1
from myapp2.views import func_based_view

authed_view = View1.as_view(template_name='foo.html')
anonymous_view = func_based_view
anonymous_view_kwargs = {'template_name': 'bar.html', }

urlpatterns += patterns(
    '',
```

```
    ...
    url(r'^$',
        HybridView.as_view(
            authed_view=authed_view, anonymous_view=anonymous_view,
            anonymous_view_kwargs=anonymous_view_kwargs
        ),
    name='home',
)
```

PaginatedCommentAJAXView

Provides a simple solution to display comments from the Django comment framework for any object. It's paginated and because it uses ajax, there's no need to reload the page every time you want to change a page.

Hook up the view in your urls::

```
from django_libs.views import PaginatedCommentAJAXView

urlpatterns += patterns(
    '',
    ...
    url(r'^comments/$', PaginatedCommentAJAXView.as_view(),
        name='libs_comment_ajax'),
)
```

Add the comment scripts. E.g. in your `base.html` do::

```
{% load static %}

<script type="text/javascript" src="{% static "django_libs/js/comments.js" %}"></
↪script>
```

Add the markup to the template, that contains the object, you want to display comments for::

```
<div data-id="ajaxComments" data-ctype="mymodel" data-object-pk="{{ object.pk }}"␣
↪data-comments-url="{% url "libs_comment_ajax" %}"></div>
```

- `data-id=ajaxComments` indicates to the scripts, that inside this div is where to render the comment list template.
- `data-ctype` is the content type name of the object. E.g. 'user' for `auth.User`.
- `data-object-pk` is most obiously the object's primary key.
- `data-comments-url` is the url you've hooked up the view.

To customize the template take a look at `django_libs/templates/django_libs/partials/ajax_comments.html`.

Also you can choose the amount of comments per page via the setting `COMMENTS_PAGINATE_BY`::

```
COMMENTS_PAGINATE_BY = 10  # default
```

There you go. All done.

RapidPrototypingView

This view allows you to render any template even when there is no URL hooked up and no view implemented. This allows your designers to quickly start writing HTML templates even before your developers have created views for those templates.

In order to use this view, hook it up in your `urls.py`:

```
from django_libs.views import RapidPrototypingView
urlpatterns += patterns(
    '',
    url(r'^prototype/(?P<template_path>.*)$',
        RapidPrototypingView.as_view(),
        name='prototype')
    ...
)
```

Now you can call any template by adding it's path to the URL of the view:

```
localhost:8000/prototype/404.html
localhost:8000/prototype/cms/partials/main_menu.html
```

Check out the `load_context` templatetag which allos you to create fake context variables for your template.

UpdateSessionAJAXView

This view allows you to update any session variables in an AJAX post.

In order to use this view, hook it up in your `urls.py`:

```
from django_libs.views import UpdateSessionAJAXView
urlpatterns += patterns(
    '',
    url(r'^update-session/$', UpdateSessionAJAXView.as_view(),
        name='update_session'),
    ...
)
```

Now you can call it by using `session_name` and `session_value`:

```
<script src="{% static "django_libs/js/getcookie.js" %}"></script>
<script>
    var data = [
        {name: 'csrfmiddlewaretoken', value: getCookie('csrftoken')}
        ,{name: 'session_name', value: 'foo'}
        ,{name: 'session_value', value: 'bar'}
    ];
    $.post(
        '/update-session/'
        ,data
    );
</script>
```

CHAPTER 18

Views Mixins

AccessMixin

Use this mixin if you want to allow users of your app to decide if the views of your app should be accessible to anonymous users or only to authenticated users:

```
form django_libs.views_mixins import AccessMixin

class YourView(AccessMixin, TemplateView):
    access_mixin_setting_name = 'YOURAPP_ALLOW_ANONYMOUS'

    # your view code here
```

Given the above example, users of your app would have to set `YOURAPP_ALLOW_ANONYMOUS` to `True` or `False`.

AjaxResponseMixin

Use this with views that can be called normally or from an AJAX call. Usually, when you call a view normally, you will have `{% extends "base.html" %}` at the beginning of the view's template. However, when you call the same view from an AJAX call you just want to update a partial region of your page, therefore the view needs to return that partial template only.

You can use this by inheriting the class:

```
from django_libs.views_mixins import AjaxResponseMixin

class MyView(AjaxResponseMixin, CreateView):
    ajax_template_prefix = 'partials/ajax_'
```

The attribute `ajax_template_prefix` defaults to `ajax_`. If you would like to store your app's ajax templates in a different way, for example in a subfolder called `partials`, you can override that attribute in your class.

DetailViewWithPostAction

This view enhances the class-based generic detail view with even more generic post actions. In order to use it, import it like all the other generic class based views and view mixins.

- Create a Mixin or View which inherits from this action mixin.
- Be sure to add a general `get_success_url()` function or custom success functions for each post action.
- Create your post actions
- Make sure to add this action names to the name attribute of an input field.

Basic usage in a html template:

```
<form method="post" action=".">
    {% csrf_token %}
    <input name="post_verify" type="submit" value="Verify" />
</form>
```

Usage in a views.py:

```
from django_libs.views_mixins import DetailViewWithPostAction

class NewsDetailBase(DetailViewWithPostAction):
    def post_verify(self):
        self.object.is_verified = True
        self.object.save()

    def post_reject(self):
        self.object.is_verified = False
        self.object.save()


class NewsEntryDetailView(NewsDetailBase):
    model = NewsEntry

def get_success_url(self):
    return reverse('newsentry_detail', kwargs={'pk': self.object.pk})

    def post_verify(self):
        super(NewsEntryDetailView, self).post_verify()
        ctx_dict = {
            'verified': True,
            'entry': self.object,
        }
        self.send_mail_to_editor(ctx_dict)

    def get_success_url_post_reject(self):
        return reverse('newsentry_list')
```

JSONResponseMixin

You can find out more about the `JSONResponseMixin` in the official Django docs: https://docs.djangoproject.com/en/dev/topics/class-based-views/#more-than-just-html

In order to use it, just import it like all the other generic class based views and view mixins:

```
from django.views.generic import View
from django_libs.views_mixins import JSONResponseMixin

class MyAPIView(JSONResponseMixin, View):
    pass
```

CHAPTER 19

Indices and tables

- genindex
- modindex
- search