
Django-flash Documentation

Release 1.7

Destaquenet Technology Solutions

January 01, 2010

CONTENTS

1	Documentation contents	3
1.1	Installation	3
1.2	Configuration	3
1.3	Using Django-Flash	6
1.4	Creating a custom flash storage backend	9
1.5	Creating a custom serialization codec	10
1.6	Django-Flash overview	11
1.7	Getting Involved	17
1.8	Changelog	18
2	Indices and tables	21
	Module Index	23
	Index	25

Django-Flash is a simple Django extension that provides support for [Rails](#)-like *flash* messages.

The *flash* is a temporary storage mechanism that looks like a Python dictionary, so you can store values associated with keys and later retrieve them. It has one special property: by default, values stored into the *flash* during the processing of a request will be available during the processing of the immediately following request. Once that second request has been processed, those values are removed automatically from the storage.

This is an open source project licenced under the terms of The [BSD License](#) and sponsored by [Destaquenet Technology Solutions](#), a brazilian software development and consultancy startup.

See Also:

[PDF version](#) of this documentation.

DOCUMENTATION CONTENTS

1.1 Installation

There are several ways to download and install Django-Flash:

Via PyPI

Execute the following command line to download and install the latest stable version from [CheeseShop](#):

```
$ easy_install -U django-flash
```

Follow [these instructions](#) to install SetupTools if you don't have it already.

Via GitHub

If you are a [Git](#) user and want to take a closer look at the project's source code, you would rather clone our [public repository](#) instead:

```
$ git clone git://github.com/danielm/django-flash.git
$ cd django-flash
$ python setup.py install
```

Zip file/tarball

Django-Flash is also available for download as [compressed archives](#) (either zip and tgz). After un-zip/untar the archive, execute the following command to install Django-Flash:

```
$ python setup.py install
```

Manually

To add Django-Flash to your project as a bundled library, just add the `djangoflash` directory into your project along with the other apps.

1.2 Configuration

In order to plug Django-Flash to your project, open your project's `settings.py` file and do the following changes:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    'djangoflash.context_processors.flash',
)
```

```
MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'djangoflash.middleware.FlashMiddleware',
)
```

That's all the required configuration.

Warning: The `djangoflash.middleware.FlashMiddleware` class must be declared after the `SessionMiddleware` class.

1.2.1 Django-Flash and requests to media files

Django itself doesn't serve static (media) files, such as images, style sheets, or video. It leaves that job to whichever web server you choose. But, *during development*, you can use the `django.views.static.serve()` view to serve media files.

The problem with it is that, as a regular view, requests to `django.views.static.serve()` trigger the installed middlewares. And since the *flash* gets updated by *a middleware*, messages might be removed from the *flash* by accident if the response causes the web browser to issue requests to fetch static files.

To make Django-Flash work well with the `django.views.static.serve()` view, you can add the setting `FLASH_IGNORE_MEDIA` to your project's `settings.py` file:

```
FLASH_IGNORE_MEDIA = True # Optional. Default: DEBUG
```

Set the `FLASH_IGNORE_MEDIA` setting to `True`, and Django-Flash won't remove any message from the *flash* if the request URL resolves to `django.views.static.serve()`. Otherwise, every request will trigger the `djangoflash.middleware.FlashMiddleware` as usual.

Note: This setting is optional; its default value is `DEBUG`. So, if you adjust the `DEBUG` setting according to the environment in which the application runs (as you *should*), you don't have to worry about this setting at all, things will just work.

1.2.2 Flash storage backends

Since *version 1.5*, Django-Flash supports custom flash storage backends.

By default, Django-Flash provides two built-in storage backends:

- `djangoflash.storage.session` – Session-based storage (default);
- `djangoflash.storage.cookie` – Cookie-based storage;

See Also:

Creating a custom flash storage backend

Using the session-based storage

Django-Flash uses the *session-based storage* by default, so you don't need to do anything else to use it.

Although you are not required to do so, you can add the following setting to your project's `settings.py` file to make it clear about what flash storage backend is being used:

```
FLASH_STORAGE = 'session' # Optional
```

This storage backend *doesn't* rely on codecs to serialize and de-serialize the flash data; it lets Django handle this.

Using the cookie-based storage

If you want to use the *cookie-based storage* instead the default one, then add the following setting to the `settings.py` file:

```
FLASH_STORAGE = 'cookie'
```

Since cookies will be used to store the contents of the flash scope, Django-Flash doesn't require you to add the `SessionMiddleware` class to the `MIDDLEWARE_CLASSES` section of your project's settings anymore.

This storage backend relies on codecs to serialize and de-serialize the flash data.

1.2.3 Flash serialization codecs

Since *version 1.7*, Django-Flash supports custom flash serialization codecs.

By default, Django-Flash provides three built-in codecs:

- `djangoflash.codec.json_impl` – JSON-based codec (default);
- `djangoflash.codec.json_zlib_impl` – JSON/zlib-based codec;
- `djangoflash.codec.pickle_impl` – Pickle-based codec;

See Also:

Creating a custom serialization codec

Using the JSON-based codec implementation

For security reasons, Django-flash uses the *JSON-based codec implementation* by default, so you don't need to do anything else to use it.

Although you are not required to do so, you can add the following setting to your project's `settings.py` file to make it clear about what codec implementation is being used:

```
FLASH_CODEC = 'json' # Optional
```

There's also an *alternative version* of this codec that uses the `zlib` module to reduce the encoded flash footprint. This is particularly useful when the flash storage backend in use (such as the *cookie-based storage*) cannot handle the amount of data in the *flash*:

```
FLASH_CODEC = 'json_zlib'
```

Using the Pickle-based codec implementation

If you want to use the *Pickle-based codec implementation* instead the default one, then add the following setting to the `settings.py` file:

```
FLASH_CODEC = 'pickle'
```

Warning: The use of this codec is not recommended since the [Pickle documentation](#) itself clearly states that it's not intended to be secure against erroneous or maliciously constructed data.

1.3 Using Django-Flash

Once plugged to your project, Django-Flash automatically adds a `flash` attribute to the `django.http.HttpRequest` objects received by your views. This property points to a `djangoflash.models.FlashScope` instance, which supports most if not all operations provided by a simple Python dict.

Here goes some examples on how to manipulate this scope from a view:

```
def my_view(request):
    request.flash['key'] = 'value' # Puts a string to the flash scope
    request.flash.put(key='value') # Alternative syntax that does the same as above
    'key' in request.flash        # Checks if the object is available at the flash scope
    request.flash['key']         # Gets an object from the flash scope
    del request.flash['key']     # Removes an object from the flash scope
```

To see the list of all methods available to you, take a look at the `djangoflash.models.FlashScope` documentation.

Although this example uses only *string* values, you are free to use, both as keys and values, any object that can be serialized by flash serialization codec in use.

1.3.1 Using the flash

You can use the *flash* the same way you use a plain dict since their interface are very similar:

```
def my_view(request):
    request.flash['key'] = 'value'           # Store a value
    request.flash['key'] = 'another value'  # Replace a value
    del request.flash['key']                # Remove a value
    for key, value in request.flash.items(): # And so on...
        print '%s - %s' % (key, value)
```

The *flash* also allows you to easily store several values under the same key. To do this just use the `djangoflash.models.FlashScope.add()` method:

```
def my_view(request):
    print 'key' in request.flash           # Output: False
    request.flash.add('key', 'one')
    request.flash.add('key', 'two')
    print request.flash['key']             # Output: ['one', 'two']
```

```
request.flash['key'] = 'one'
request.flash.add('key', 'two')
request.flash.add('key', 'three')
print request.flash['key']           # Output: ['one', 'two', 'three']
```

1.3.2 Flash-scoped objects: the default lifecycle

First let's see a basic example of how Django-Flash controls the lifecycle of flash-scoped objects. Consider the following views:

```
# URL: http://server/app/first
def first_view(request):
    request.flash['message'] = 'My message'
    return HttpResponseRedirect(reverse(second_view))

# URL: http://server/app/second
def second_view(request):
    print request.flash['message']           # Output: My message
    request.flash['another_message'] = 'Something'
    return HttpResponseRedirect(reverse(third_view))

# URL: http://server/app/third
def third_view(request):
    print request.flash['another_message']  # Output: Something
    print 'message' in request.flash       # Output: False
    return HttpResponseRedirect(reverse(fourth_view))

# URL: http://server/app/fourth
def fourth_view(request):
    return HttpResponse(...)
```

Let's say that we have opened our web browser and issued a request to <http://server/app/first>. When `first_view()` executes, it stores an object inside the *flash* under the key `message`. The last line returns a HTTP Redirect, which makes our web browser fire a GET request to <http://server/app/second>.

When `second_view()` executes, it prints the content of the flash-scoped object under the key `message`, which was stored in the previous request by `first_view()`. The next line of code stores another object inside the *flash* under the key `another_message`. Again, the last line returns a HTTP Redirect, which makes our web browser fire a GET request to <http://server/app/third>.

When `third_view()` executes, the flash-scoped object under the key `another_message`, which was stored in the previous request by `second_view()`, is available for use. But, at the same time, the flash-scoped object added by `first_view()` was automatically removed.

See Also:

Django-Flash overview

1.3.3 Managing flash lifecycle

By default, all objects stored inside the *flash* survives until the *very next* request, being automatically removed after that. Unfortunately, this default behavior might not be enough in some situations.

Preventing flash-scoped objects from being removed

We can prevent flash-scoped objects from being removed by using the `djangoflash.models.FlashScope.keep()` method:

```
def first_view(request):
    request.flash['message'] = 'Operation succeeded!'
    return HttpResponseRedirect(reverse(second_view))

def second_view(request):
    print request.flash['message']           # Output: Operation succeeded!
    request.flash.keep('message')
    return HttpResponseRedirect(reverse(third_view))

def third_view(request):
    print request.flash['message']           # Output: Operation succeeded!
    return HttpResponseRedirect(reverse(fourth_view))

def fourth_view(request):
    print 'message' in request.flash         # Output: False
    return HttpResponse(...)
```

If you want to keep *all* flash-scoped objects, just call the `djangoflash.models.FlashScope.keep()` method with no arguments:

```
def second_view(request):
    request.flash.keep()
    return HttpResponseRedirect(reverse(third_view))
```

A more declarative way to keep values is also supported through the `djangoflash.decorators.keep_messages()` decorator:

```
from djangoflash.decorators import keep_messages

# Keeps the entire flash...
@keep_messages
def second_view(request):
    return HttpResponseRedirect(reverse(third_view))

# ...or specific messages
@keep_messages('message', 'another_message')
def second_view(request):
    return HttpResponseRedirect(reverse(third_view))
```

Adding an immediate flash-scoped object

It's sometimes convenient to store an object inside the *flash* and use it on the *current* request only.

This can be done by using the `djangoflash.models.FlashScope.now` attribute:

```
def first_view(request):
    request.flash.now['message'] = 'My message'
    request.flash.now(message='My message')           # Alternative syntax
    print request.flash['message']                     # Output: My message
    return HttpResponseRedirect(reverse(second_view))
```

```
def second_view(request):
    print 'message' in request.flash           # Output: False
```

1.3.4 Accessing flash-scoped objects from view templates

We already know how to access the *flash* from views. But what about the view templates?

See Also:

`djangoflash.context_processors` module.

It's just as easy:

```
<html>
<head>
  <title>My template</title>
</head>
<body>
  {% if flash.message %}
    <!-- There's a flash-scoped object under the 'message' key -->

    <div class="flash_message">
      <p>{{ flash.message }}</p>
    </div>
  {% endif %}
</body>
</html>
```

It's also possible to iterate over all flash-scoped objects using the `{% for %}` tag if you want to:

```
<html>
<head>
  <title>My template</title>
</head>
<body>
  {% if flash %}
    <!-- There's one or more flash-scoped objects -->

    {% for key, value in flash.items %}
      <div class="flash_{{ key }}">
        <p>{{ value }}</p>
      </div>
    {% endfor %}
  {% endif %}
</body>
</html>
```

1.4 Creating a custom flash storage backend

Since *version 1.5*, Django-Flash supports custom flash storage backends.

By default, Django-flash provides two built-in storage backends:

- `djangoflash.storage.session` – Session-based storage (default);
- `djangoflash.storage.cookie` – Cookie-based storage;

The good news is that you can create your own storage backend if the existing ones are getting in your way. To do so, the first thing you need to do is create a Python module with a class called `FlashStorageClass`:

```
# Let's suppose this module is called 'myproj.djangoflash.custom'

# You can use the serialization codec configured by the user
from.djangoflash.codec import codec

class FlashStorageClass(object):
    def __is_flash_stored(self, request):
        # This method checks whether the flash is already stored
        pass

    def set(self, flash, request, response):
        if flash:
            # Store the flash
            pass
        elif self.__is_flash_stored(request):
            # Flash is null or empty, so remove the already stored flash
            pass

    def get(self, request):
        if self.__is_flash_stored(request):
            # Return the stored flash
            pass
```

Then, to use your custom flash storage backend, add the following setting to your project's `settings.py` file:

```
FLASH_STORAGE = 'myproj.djangoflash.custom' # Path to module
```

See Also:

Configuration

1.5 Creating a custom serialization codec

Since *version 1.7*, Django-Flash supports custom flash serialization codecs.

By default, Django-Flash provides three built-in codecs:

- `djangoflash.codec.json_impl` – JSON-based codec (default);
- `djangoflash.codec.json_zlib_impl` – JSON/zlib-based codec;
- `djangoflash.codec.pickle_impl` – Pickle-based codec;

The good news is that you can create your own codec if the existing ones are getting in your way. To do so, the first thing you need to do is create a Python module with a class called `CodecClass`:

```
# Let's suppose this module is called 'myproj.djangoflash.custom'

from.djangoflash.codec import BaseCodec

class CodecClass(BaseCodec):
    def __init__(self):
        BaseCodec.__init__(self)

    def encode(self, flash):
```

```

    pass

    def decode(self, encoded_flash):
        pass

```

Note that custom codecs must extend the `djangoflash.codec.BaseCodec` class direct or indirectly.

Finally, to use your custom codec, add the following setting to your project's `settings.py` file:

```
FLASH_CODEC = 'myproj.djangoflash.custom' # Path to module
```

See Also:

Configuration

1.6 Django-Flash overview

1.6.1 djangoflash.models — Django-Flash model

This module provides the `FlashScope` class, which provides a simple way to pass temporary objects between views.

FlashScope Class

```
class FlashScope(data=None)
```

Bases: `object`

The purpose of this class is to implement the *flash*, which is a temporary storage mechanism that looks like a Python dictionary, so you can store values associated with keys and later retrieve them.

It has one special property: by default, values stored into the *flash* during the processing of a request will be available during the processing of the immediately following request. Once that second request has been processed, those values are removed automatically from the storage.

The following operations are supported by `FlashScope` instances:

len(f)

Returns the number of items in the flash *f*.

f[key]

Returns the item of *f* with key *key*. Raises a `KeyError` if *key* is not in the flash *f*.

f[key] = value

Sets *f[key]* to *value*.

del f[key]

Removes *f[key]* from *f*. Raises a `KeyError` if *key* is not in the map.

key in f

Returns `True` if *f* has a key *key*, else `False`.

key not in f

Equivalent to `not key in f`.

f.now[key] = value

Sets *f[key]* to *value* and marks it as *used*.

f.now(items)**

Puts the given *items* into *f* and marks them as *used*.

add (*key*, *value*)
Appends a value to a key in this flash.

clear ()
Removes all items from this flash.

discard (**keys*)
Marks the entire current flash or a single value as *used*, so when the next request hit the server, those values will be automatically removed from this flash by `FlashMiddleware`.

get (*key*, *default=None*)
Gets the value under the given *key*. If the *key* is not found, *default* is returned instead.

has_key (*key*)
Returns `True` if there's a value under the given *key*. Deprecated since version 1.4.2: `has_key()` is deprecated in favor of `key in f`.

items ()
Returns the list of items as tuples (*key*, *value*).

iteritems ()
Returns an iterator over the (*key*, *value*) items.

iterkeys ()
Returns an iterator over the keys.

itervalues ()
Returns an iterator over the values.

keep (**keys*)
Prevents specific values from being removed on the next request. If this method is called with no args, the entire flash is preserved.

keys ()
Returns the list of keys.

pop (*key*, *default=None*)
Removes the specified *key* and returns the corresponding value. If *key* is not found, *default* is returned instead.

put (***kwargs*)
Puts one or more values into this flash.

put_immediate (*key*, *value*)
Puts a value inside this flash and marks it as *used*.

to_dict ()
Exports this flash to a `dict`.

update ()
Mark for removal entries that were kept, and delete unkept ones.

Note: This method is called automatically by `django.flash.middleware.FlashMiddleware` when a HTTP request hits the server, so never call this method yourself, unless you have a very good reason to do so.

values ()
Returns the list of values.

See Also:

[Django-Flash overview](#)

1.6.2 `djangoflash.middleware` — Django-Flash middleware

This module provides the `FlashMiddleware` class, which manages the *flash* whenever a HTTP request hits the server.

To plug this middleware to your Django project, edit your project's `settings.py` file as follows:

```
MIDDLEWARE_CLASSES = (
    'djangoflash.middleware.FlashMiddleware',
)
```

FlashMiddleware Class

```
class FlashMiddleware()
```

Bases: `object`

This middleware uses the flash storage backend specified by the project's `settings.py` file in order to store and retrieve `djangoflash.models.FlashScope` objects, being also responsible for expiring old flash-scoped objects.

Note: This class is designed to be used by the Django framework itself.

process_request (*request*)

This method is called by the Django framework when a *request* hits the server.

process_response (*request*, *response*)

This method is called by the Django framework when a *response* is sent back to the user.

See Also:

Django-Flash overview

1.6.3 `djangoflash.context_processors` — Django-Flash context processors

This module provides the context processor that exposes `djangoflash.models.FlashScope` objects to view templates.

To plug this context processor to your Django project, edit your project's `settings.py` file as follows:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    'djangoflash.context_processors.flash',
)
```

Doing this, the view templates will be able to access the *flash* contents using the `flash` context variable.

Warning: Your views should use the `RequestContext` class to render the templates, otherwise the `flash` variable (along with *all* other variables provided by other context processors) won't be available to them. Please read the [Django docs](#) for further instructions.

flash (*request*)

This context processor gets the `FlashScope` object from the current *request* and adds it to the template context:

```
<html>
  <head></head>
  <body>
    request.flash['message'] = {{ flash.message }}
```

```
</body>
</html>
```

See Also:

Django-Flash overview

1.6.4 `djangoflash.decorators` — Django-Flash decorators

This module provides decorators to simplify common tasks.

keep_messages (**keys*)

Prevents specific values from being removed during the processing of the decorated view. If this decorator is used with no args, the entire flash is preserved.

See Also:

Django-Flash overview

1.6.5 `djangoflash.storage` — Flash storage backends

This package provides some built-in flash storage backends used to persist the *flash* contents across requests.

get_storage (*module*)

Creates and returns the flash storage backend defined in the given module path (ex: "myapp.mypackage.mymodule"). The argument can also be an alias to a built-in storage backend, such as "session" or "cookie".

Built-in flash storage backends

`djangoflash.storage.session` — Session-based flash storage

This module provides a session-based flash storage backend.

Since this backend relies on the user's session, you need to include the `SessionMiddleware` class to the `MIDDLEWARE_CLASSES` section of your project's `settings.py` file:

```
MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'djangoflash.middleware.FlashMiddleware',
)
```

See Also:

Configuration

FlashStorageClass Class

```
class FlashStorageClass ()
```

Bases: object

Session-based flash storage backend.

get (*request*)

Returns `FlashScope` object stored in the session.

set (*flash, request, response*)
Stores the given `FlashScope` object in the session.

See Also:

Django-Flash overview

djangoflash.storage.cookie — Cookie-based flash storage

This module provides a cookie-based flash storage backend.

Warning: The actual `FlashScope` object is sent back to the user in a cookie. Although some encryption is performed to help spot when the flash data is modified by third-parties, this backend should be avoided when sensitive information is stored in the *flash*.

Warning: Although in general user agents' cookie support should have no fixed limits, according to [RFC-2965](#), section 5.3, all implementations must support at least 4096 bytes per cookie. So be careful about the amount of data you store in the *flash* when using this storage backend.

FlashStorageClass Class

class `FlashStorageClass` ()

Bases: `object`

Cookie-based flash storage backend.

get (*request*)
Returns `FlashScope` object stored in a cookie.

set (*flash, request, response*)
Stores the given `FlashScope` object in a cookie.

See Also:

Django-Flash overview

See Also:

Django-Flash overview

1.6.6 djangoflash.codec — Flash serialization codecs

This package provides some built-in flash serialization codecs.

get_codec (*module*)

Creates and returns the codec defined in the given module path (ex: `"myapp.mypackage.mymodule"`). The argument can also be an alias to a built-in codec, such as `"json"`, `"json_zlib"` or `"pickle"`.

BaseCodec Class

class `BaseCodec` ()

Bases: `object`

Base codec implementation. All codec implementations must extend this class.

decode (*encoded_flash*)
Empty implementation that raises `NotImplementedError`.

decode_signed (*encoded_flash*)

Restores the *flash* object from the given encoded-and-signed data.

encode (*flash*)

Empty implementation that raises `NotImplementedError`.

encode_and_sign (*flash*)

Returns an encoded-and-signed version of the given *flash*.

Built-in serialization codecs

`djangoflash.codec.json_impl` — JSON-based codec implementation

This module provides a JSON-based codec implementation.

CodecClass Class

class CodecClass ()

Bases: `djangoflash.codec.BaseCodec`

JSON-based codec implementation.

decode (*encoded_flash*)

Restores the *flash* from the given JSON string.

encode (*flash*)

Encodes the given *flash* as a JSON string.

See Also:

[Django-Flash overview](#)

`djangoflash.codec.json_zlib_impl` — JSON/zlib-based codec implementation

This module provides a JSON-based codec implementation that uses the `zlib` module to reduce the encoded flash footprint.

CodecClass Class

class CodecClass ()

Bases: `djangoflash.codec.json_impl.CodecClass`

JSON/zlib-based codec implementation.

decode (*encoded_flash*)

Restores the *flash* from the given zlib compressed JSON string.

encode (*flash*)

Encodes the given *flash* as a zlib compressed JSON string.

See Also:

[Django-Flash overview](#)

`djangoflash.codec.pickle_impl` — Pickle-based codec implementation

This module provides a Pickle-based codec implementation.

Warning: The use of this codec is not recommended since the [Pickle documentation](#) itself clearly states that it's not intended to be secure against erroneous or maliciously constructed data.

CodecClass Class

`class CodecClass ()`

Bases: `djangoflash.codec.BaseCodec`

Pickle-based codec implementation.

decode (*encoded_flash*)

Restores the *flash* from the given Pickle dump string.

encode (*flash*)

Encodes the given *flash* as a Pickle dump string.

See Also:

[Django-Flash overview](#)

See Also:

[Django-Flash overview](#)

1.7 Getting Involved

As with any open source project, there are several ways you can help:

- Report bugs, feature requests and other issues in the [issue tracking system](#);
- Submit patches to reported issues (both those you find, or that others have filed);
- Help with the documentation by pointing out areas that are lacking or unclear, and if you are so inclined, submitting patches to correct it;
- Improve the overall project quality by suggesting refactorings and improving the test cases. A great way to learn – and in turn give value back to the community – is to review someone else's code. So, we invite you to review ours;
- Create and share packages to make it even easier to distribute Django-Flash to other users of your favourite Distribution or Operating System;
- Write about Django-Flash in your blog or personal web site. Let your friends know about this project.

Your participation is much appreciated. Keep up with Django-Flash development on [Github](#).

1.7.1 How do I join the team?

Django-Flash is a very mature project and it's probably not going to get lots of new features. But those that the developers notice participating to a high extent will be invited to join the team as a committer.

This is as much based on personality and ability to work with other developers and the community as it is with proven technical ability. Being unhelpful to other users, or obviously looking to become a committer for bragging rights and nothing else is frowned upon, as is asking to be made a committer without having contributed sufficiently to be invited.

1.7.2 Contact information

Author Daniel Fernandes Martins <daniel@destaquenet.com>

Company Destaquenet Technology Solutions

1.8 Changelog

Like any other piece of software, Django-Flash is evolving at each release. Here you can track our progress:

Version 1.7 (October 25, 2009)

- Added support for custom flash serialization codecs;
- Three built-in codec implementations: JSON, JSON/zlib and Pickle;
- Module `djangoflash.storage.base` removed;

Version 1.6.3 (October 07, 2009)

- Using the `DEBUG` setting as the default value of `FLASH_IGNORE_MEDIA`;

Version 1.6.2 (September 18, 2009)

- Done some work to avoid the loss of messages when the `CommonMiddleware` returns a `HttpResponseRedirect` due to a missing trailing slash;

Version 1.6.1 (August 19, 2009)

- Now the middleware checks if the request resolves to `django.views.static.serve()` instead of relying on the `MEDIA_URL` setting;

Version 1.6 (August 13, 2009)

- Fixed a bug in which messages are prematurely removed from the flash when they are replaced using `flash.now` in some circumstances;
- Added the `FLASH_IGNORE_MEDIA` setting to let the user choose whether requests to static files should be ignored;

Version 1.5.3 (July 22, 2009)

- Fixed a bug in the middleware which causes flash data to be discarded after requests to static files;

Version 1.5.2 (July 15, 2009)

- Added a `djangoflash.decorators.keep_messages()` decorator for keeping flash messages;
- New `AUTHORS` file;

Version 1.5.1 (June 26, 2009)

- Added a method `djangoflash.models.FlashScope.add()` that simplifies the storage of multiple values under the same key;

Version 1.5 (June 24, 2006)

- License changed from LGPL to BSD to give users more freedom;
- Added support for custom flash storage backends;
- Added a cookie-based flash storage;
- Default session-based storage was factored out to an independent class;
- Added a few more sanity checks;

Version 1.4.4 (*June 09, 2009*)

- Fixed a critical bug in the middleware;

Version 1.4.3 (*June 08, 2009*)

- Added a few more sanity checks;

Version 1.4.2 (*February 13, 2009*)

- Deprecating method `djangoflash.models.FlashScope.has_key()`;
- Documentation improvements;
- Internals refactoring;

Version 1.4.1 (*February 06, 2009*)

- Immediate values (`djangoflash.models.FlashScope.now`) can be manipulated using a dict-like syntax;
- Unit test improvements;
- Documentation improvements;

Version 1.4 (*February 05, 2009*)

- **Notice:** *breaks backwards compatibility*;
- Now Django-Flash works pretty much like the original [Ruby on Rails](#)' flash;
- Several code optimizations;
- Several improvements on the test suite;

Version 1.3.5 (*February 03, 2009*)

- Several documentation improvements;
- Improvements on source code comments and unit tests;

Version 1.3.4 (*February 01, 2009*)

- Added [Sphinx](#)-based documentation;
- Source code changed to improve the [Pylint](#) score;
- `djangoflash` module now have a `__version__` property, which is very useful when you need to know what version of the Django-Flash is installed in your machine;

Version 1.3.3 (*January 31, 2009*)

- *Critical Bug Fixed:* Django-Flash creates several useless session entries when the cookie support in user's browser is disabled;
- Small improvements on unit tests;

Version 1.3.2 (*December 07, 2008*)

- Small fixes;

Version 1.3.1 (*December 07, 2008*)

- Added some sanity checks;

Version 1.3 (*December 07, 2008*)

- **Notice:** *breaks backwards compatibility*;

- Django-Flash now controls the expiration of flash-scoped values individually, which means that only expired values are removed from the session (and not the whole flash context);
- Unit testing code was completely rewritten and now a real Django application is used in integration tests;
- Huge source code review to make it easier to read and to assure the use of Python conventions;
- Project renamed to **Django-Flash** (it was previously called **djangoflash**, without the hyphen);

Version 1.2 (November 01, 2008)

- **Notice:** *breaks backwards compatibility*;
- Improvements on the test comments;
- Now the flash scope works pretty much like a `dict`, although still there's no value-based expiration (the whole flash scope expires at the end of the request);

Version 1.1 (November 01, 2008)

- Now using `SetupTools` to make the project easier to distribute;

Version 1.0 (October 22, 2008)

- First (very simple) version;

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

MODULE INDEX

D

- `djangoflash.codec`, 15
- `djangoflash.codec.json_impl`, 16
- `djangoflash.codec.json_zlib_impl`, 16
- `djangoflash.codec.pickle_impl`, 17
- `djangoflash.context_processors`, 13
- `djangoflash.decorators`, 14
- `djangoflash.middleware`, 13
- `djangoflash.models`, 11
- `djangoflash.storage`, 14
- `djangoflash.storage.cookie`, 15
- `djangoflash.storage.session`, 14

INDEX

A

add() (django.flash.models.FlashScope method), 11

B

BaseCodec (class in django.flash.codec), 15

C

clear() (django.flash.models.FlashScope method), 12

CodecClass (class in django.flash.codec.json_impl), 16

CodecClass (class in django.flash.codec.json_zlib_impl), 16

CodecClass (class in django.flash.codec.pickle_impl), 17

D

decode() (django.flash.codec.BaseCodec method), 15

decode() (django.flash.codec.json_impl.CodecClass method), 16

decode() (django.flash.codec.json_zlib_impl.CodecClass method), 16

decode() (django.flash.codec.pickle_impl.CodecClass method), 17

decode_signed() (django.flash.codec.BaseCodec method), 15

discard() (django.flash.models.FlashScope method), 12

django.flash.codec (module), 15

django.flash.codec.json_impl (module), 16

django.flash.codec.json_zlib_impl (module), 16

django.flash.codec.pickle_impl (module), 17

django.flash.context_processors (module), 13

django.flash.decorators (module), 14

django.flash.middleware (module), 13

django.flash.models (module), 11

django.flash.storage (module), 14

django.flash.storage.cookie (module), 15

django.flash.storage.session (module), 14

E

encode() (django.flash.codec.BaseCodec method), 16

encode() (django.flash.codec.json_impl.CodecClass method), 16

encode() (django.flash.codec.json_zlib_impl.CodecClass method), 16

encode() (django.flash.codec.pickle_impl.CodecClass method), 17

encode_and_sign() (django.flash.codec.BaseCodec method), 16

F

flash() (in module django.flash.context_processors), 13

FlashMiddleware (class in django.flash.middleware), 13

FlashScope (class in django.flash.models), 11

FlashStorageClass (class in django.flash.storage.cookie), 15

FlashStorageClass (class in django.flash.storage.session), 14

G

get() (django.flash.models.FlashScope method), 12

get() (django.flash.storage.cookie.FlashStorageClass method), 15

get() (django.flash.storage.session.FlashStorageClass method), 14

get_codec() (in module django.flash.codec), 15

get_storage() (in module django.flash.storage), 14

H

has_key() (django.flash.models.FlashScope method), 12

I

items() (django.flash.models.FlashScope method), 12

iteritems() (django.flash.models.FlashScope method), 12

iterkeys() (django.flash.models.FlashScope method), 12

itervalues() (django.flash.models.FlashScope method), 12

K

keep() (django.flash.models.FlashScope method), 12

keep_messages() (in module django.flash.decorators), 14

keys() (django.flash.models.FlashScope method), 12

P

pop() (django.flash.models.FlashScope method), 12

`process_request()` (django-flash.middleware.FlashMiddleware method), 13

`process_response()` (django-flash.middleware.FlashMiddleware method), 13

`put()` (django-flash.models.FlashScope method), 12

`put_immediate()` (django-flash.models.FlashScope method), 12

S

`set()` (django-flash.storage.cookie.FlashStorageClass method), 15

`set()` (django-flash.storage.session.FlashStorageClass method), 14

T

`to_dict()` (django-flash.models.FlashScope method), 12

U

`update()` (django-flash.models.FlashScope method), 12

V

`values()` (django-flash.models.FlashScope method), 12