django-stored-messages Documentation

Release 1.3.1

evonove

March 02, 2016

Contents

Django contrib.messages on steroids!

The app integrates smoothly with Django's messages framework (*django.contrib.messages*), but users can decide which messages have to be stored on the database backend and kept available over sessions.

CHAPTER 1

Features

- Seamless integration with `django.contrib.messages`
- All the features are in a mixin you can attach to your existing storage
- Stored messages are archived in the database or in a Redis instance
- Users can configure which message levels have to be persisted
- REST api to retrieve and mark messages as read (needs `djangorestframework` being installed)
- Signalling api to perform actions in response to messages activity

CHAPTER 2

Compatibility table

- Python 2.7, 3.4
- Django 1.4, 1.5, 1.6, 1.7, 1.8
- Django Rest Framework 2.4.x, 3.1.x (only if you want to use REST endpoints)

CHAPTER 3

Contents

3.1 Installation

At the command line:

```
$ easy_install django-stored-messages
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv django-stored-messages
$ pip install django-stored-messages
```

Add *stored_messages* to the list of installed apps. You also have to enable *django.contrib.messages* framework for using stored messages:

```
INSTALLED_APPS = (
    # ...
    'django.contrib.messages',
    'stored_messages',
)

MIDDLEWARE_CLASSES = (
    # ...
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
)

TEMPLATE_CONTEXT_PROCESSORS = (
    # ...
    'django.contrib.messages.context_processors.messages'
)
```

Specify which is the storage class for messages, django-stored-messages provides a convenient default which adds persistent messages to the *storage.fallback.FallbackStorage* class from Django:

```
MESSAGE_STORAGE = 'stored_messages.storage.PersistentStorage'
```

3.2 Usage

3.2.1 Using django.contrib.messages api

Which messages are stored?

django-stored-messages integrates smoothly with *django.contrib.messages* so you can keep on adding flash messages together with stored ones. But how does django-stored-messages know which messages have to be persisted and which not? This is completely up to the user, who can configure the desired behaviour through the *STORE_LEVELS* settings. This setting is a list containing the message levels (both provided by Django or custom) which have to be persisted. For example:

```
'STORE_LEVELS': (
    INFO,
    ERROR,
),
```

tells django-stored-messages to persist messages of level *INFO* and *ERROR*, both provided by Django. django-stored-messages provides a set of message levels for convenience that can be used to store message without setting anything and letting Django levels to behave normally:

- STORED_DEBUG,
- STORED_INFO,
- STORED_SUCCESS,
- STORED_WARNING,
- STORED_ERROR,

How do I retrieve stored messages?

Premise: stored messages have a status which can be *read* or *unread*. Using the Django api for displaying messages, it will show *unread* messages together with Django "regular" messages. For example, in a template:

```
{% if messages %}
<ul class="messages">
    {% for message in messages %}
    <li{% if message.tags %} class="{{ message.tags }}"{% endif %}>{{ message }}</li>
    {% endfor %}
</ul>
{% endif %}
```

Please notice that displaying stored messages, just like regular messages, will expire them: this means regular messages are removed from their storages (cookies or the session) and stored messages will be marked as *read* (they'll be still in the database, though). If this is not the desired behaviour, and you want to keep messages *unread* even after displaying them, set the *used* parameter in the storage instance as *False*:

```
storage = messages.get_messages(request)
for message in storage:
    do_something_with(message)
storage.used = False
```

3.2.2 Using django-stored-messages api

There are situations in which one can leverage the fact that messages are stored in the database and use them beyond the intentions of *django.contrib.messages.api*. For example one could:

- send a message without having access to a request object
- send a message to multiple users
- manually mark a message *read* instead of doing this automatically iterating the storage

django-stored-messages provides an additional api containing some utility methods useful in such cases.

`stored_messages.api.`**`add_message_for`**(*users*, *level*, *message_text*, *extra_tags=''*, *date=None*, *url=None*, *fail_silently=False*)

Send a message to a list of users without passing through *django.contrib.messages*

Parameters

- **`users`** – an iterable containing the recipients of the messages
- **`level`** – message level
- **`message_text`** – the string containing the message
- **`extra_tags`** – like the Django api, a string containing extra tags for the message
- **`date`** – a date, different than the default timezone.now
- **`url`** – an optional url
- **`fail_silently`** – not used at the moment

`stored_messages.api.`**`broadcast_message`**(*level*, *message_text*, *extra_tags=''*, *date=None*, *url=None*, *fail_silently=False*)

Send a message to all users aka broadcast.

Parameters

- **`level`** – message level
- **`message_text`** – the string containing the message
- **`extra_tags`** – like the Django api, a string containing extra tags for the message
- **`date`** – a date, different than the default timezone.now
- **`url`** – an optional url
- **`fail_silently`** – not used at the moment

`stored_messages.api.`**`mark_read`**(*user*, *message*)

Mark message instance as read for user. Returns True if the message was *unread* and thus actually marked as *read* or False in case it is already *read* or it does not exist at all.

Parameters

- **`user`** – user instance for the recipient
- **`message`** – a Message instance to mark as read

`stored_messages.api.`**`mark_all_read`**(*user*)

Mark all message instances for a user as read.

Parameters **`user`** – user instance for the recipient

3.3 Storage Backends

With version 1.0, the concept of *Storage Backend* was introduced to let developers choose how messages are persisted. Django Stored Messages provides a pool of backends out of the box and developers can extend the app providing their own implementation of a *Storage Backend*.

STORAGE_BACKEND settings parameter contains a string representing the backend class to use. If not specified, it defaults to the default backend.

Here follows a list of supported backends.

3.3.1 Default backend: Django ORM

```
'STORAGE_BACKEND': 'stored_messages.backends.DefaultBackend'
```

This is the default backend, it stores messages on the configured database using plain old Django models; it doesn't need any additional configuration.

3.3.2 Redis backend

```
'STORAGE_BACKEND': 'stored_messages.backends.redis'
```

Users' inbox and archives are persisted on a Redis instance. Keys are in the form *user:<userid>:notifications* *user:<userid>:archive* and values are lists. This backend needs the *REDIS_URL* settings to be specified, for example:

```
STORED_MESSAGES={
    'REDIS_URL': 'redis://username:password@localhost:6379/0',
}
```

3.3.3 Implementing your own backend

Custom backends should derive from *StoredMessagesBackend* class and implement all the methods:

class `stored_messages.backends.base.`**`StoredMessagesBackend`**

`archive_list`(*user*)
Retrieve all the messages in *user*'s archive.

Params: user: Django User instance

Return: An iterable containing *Message* instances

`archive_store`(*users*, *msg_instance*)
Store a *Message* instance in the archive for a list of users.

Params: users: a list or iterable containing Django User instances msg_instance: Message instance to persist in archive

Return: None

Raise: MessageTypeNotSupported if *msg_instance* cannot be managed by current backend

`can_handle`(*msg_instance*)
Determine if this backend can handle messages of the same type of *msg_instance*.

Params: *msg_instance*: *Message* instance

Return: True if type is correct, False otherwise

create_message(*level*, *msg_text*, *extra_tags*, *date=None*)
Create and return a *Message* instance. Instance types depend on backends implementation.

Params: *level*: message level (see django.contrib.messages) *msg_text*: what you think it is *extra_tags*: see django.contrib.messages *date*: a DateTime (optional)

Return: *Message* instance

expired_messages_cleanup()
Remove messages that have been expired.

Params: None

Return: None

inbox_delete(*user*, *msg_id*)
Remove a *Message* instance from *user*'s inbox.

Params: user: Django User instance msg_id: Message identifier

Return: None

Raise: MessageDoesNotExist if msg_id was not found

inbox_get(*user*, *msg_id*)
Retrieve a *Message* instance from *user*'s inbox.

Params: user: Django User instance msg_id: Message identifier

Return: A *Message* instance

Raise: MessageDoesNotExist if msg_id was not found

inbox_list(*user*)
Retrieve all the messages in *user*'s Inbox.

Params: *user*: Django User instance

Return: An iterable containing *Message* instances

inbox_purge(*user*)
Delete all the messages in *user*'s Inbox.

Params: user: Django User instance

Return: None

inbox_store(*users*, *msg_instance*)
Store a *Message* instance in the inbox for a list of users.

Params: users: a list or iterable containing Django User instances msg_instance: Message instance to persist in inbox

Return: None

Raise: MessageTypeNotSupported if *msg_instance* cannot be managed by current backend

3.4 Advanced Usage

3.4.1 Interact with stored messages through the REST api

When *Django REST framework* is available and installed, Stored Messages exposes a RESTful api which consists of the following endpoints:

- */inbox/* - method: *GET*: retrieve the list of unread messages for current logged in user.
- */inbox/{lookup}/* - method: *GET*: get the details for the message having *{lookup}* pk.
- */inbox/{lookup}/read/* - method: *POST*: mark the message having *{lookup}* pk as read.
- */mark_all_read/* - method: *POST*: mark all messages as read for current logged in user.

3.4.2 Writing a custom storage

All the functionalities for persisitng messages are implemented in the *StorageMixin* class. Such mixin can be derived together with one of the default storages provided by *django.contrib.messages* so that messages which types are configured to be persisted will be actually saved to the database and all the others will be passed to the default storage. The mixin could also be implemented together with a more specialized storage provided by the user and not necessarily one of those provided by Django.

3.4.3 Signals

A few hooks are available in *backends.signals*.

For inbox we raise the following signals:

- *inbox_stored*: a message has been stored, providing *user* and *message* as arguments
- *inbox_deleted*: a message has been deleted, providing *user* and *message_id* as arguments
- *inbox_purged*: the inbox has been purged, providing *user* as argument

For archive we raise the following signals:

- *archive_stored*: a message has been stored, providing *user* and *message* as arguments

3.5 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

3.5.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/evonove/django-stored-messages/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.

- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

django-stored-messages could always use more documentation, whether as part of the official django-stored-messages docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/evonove/django-stored-messages/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

3.5.2 Get Started!

Ready to contribute? Here's how to set up *django-stored-messages* for local development.

1. Fork the *django-stored-messages* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/django-stored-messages.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv django-stored-messages
$ cd django-stored-messages/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass the tests, including testing other Python versions with tox. **Remember to start a local instance of Redis before running the testsuite**:

```
$ python runtests.py
$ tox
```

To get tox, just pip install it into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

3.5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7 and 3.3. Check https://travis-ci.org/evonove/django-stored-messages/pull_requests and make sure that the tests pass for all supported Python versions.

3.6 Contributors

Massimiliano Pippi <masci@evonove.it> Federico Frenguelli <synasius@gmail.com> Fabrizio Buratta <fabrizio@moldiscovery.com> Marco Angelucci <tilde@autistici.org>

3.7 History

3.7.1 1.0.1 (2014-04-17)

- Major bug fixed on *inbox_get()* backend api
- Fixed InboxSerializer for redis backend messages
- Enhanced testsuite
- Added MessageDoesNotExist descriptions and return 404

3.7.2 1.0.0 (2014-04-01)

- New backend architecture with Redis support
- Support for broadcast messages

3.7.3 0.2.1 (2013-12-23)

- Added *stored_messages_count* template tag and tests

3.7.4 0.2.0 (2013-10-22)

- Added *stored_messages_archive* template tag
- Extended REST api

3.7.5 0.1.2 (2013-10-13)

- Added specific template tags for stored messages

3.7.6 0.1.1 (2013-10-10)

- Fixed setup.py

3.7.7 0.1.0 (2013-10-08)

- First release on PyPI.

S

A

B

C

E

I

M

S