drf-messages

Dan Yishai

Jan 01, 2022

INSTALLATION AND SETUP

Use Django's Messages Framework with Django Rest Framework project.

CHAPTER

ONE

FEATURES

- Persistent message storage in database
- Automatic cleanup
- DRF endpoint for accessing messages

1.1 Quick Start

1. Install using:

```
$ pip install drf-messages
```

2. Configure project `settings.py`:

```
INSTALLED_APPS = [
    # ...
    'django.contrib.messages',
    'rest_framework',
    'drf_messages',
    # ...
]

MESSAGE_STORAGE = "drf_messages.storage.DBStorage"
```

4. Configure routes at your project's `urls.py`

```
urlpatterns = [
    path('messages/', include('drf_messages.urls')),
    # ...
]
```

3. Run migrations using:

```
$ py manage.py migrate drf_messages
```

For more details visit the docs for installation: https://drf-messages.readthedocs.io/en/latest/installation/installation.html

1.1.1 Usage

You can list all your messages with:

```
$ curl -X GET "http://localhost/messages/"
```

Any unread messages will have `read_at` as `null`. If you have `django-filter` configured, you can also query "http://localhost/messages/?unread=true" to get only unread messages.

1.2 Installation

This is a **detailed** walk-through the *drf-messages* installation and setup process.
For easy and quick installation please refer to the *Quick Start* guide.

1.2.1 Getting it

You can get `drf-messages` by using pip:

```
$ pip install drf-messages
```

If you want to install it from source, grab the git repository and run setup.py:

```
$ git clone https://github.com/danyi1212/drf-messages.git
$ python setup.py install
```

1.2.2 Dependencies

Django Rest Framework is an **optional dependency** of this module. It is required only for the provided views and serializers.

Install using pip with:

```
$ pip install djangorestframework
```

If you are only planning to use the persistent storage and do not need the provided view, you can skip the installation of Django Rest Framework.

See also:

To install it properly visit the installation docs at https://www.django-rest-framework.org/#installation

It is also recommended to also install `django-filter` to enable the included list filters for the views. See more information in the *Rest API Views* docs.

Install using pip with:

```
$ pip install django-filter
```

To install properly, follow the installation docs at https://django-filter.readthedocs.io/en/stable/guide/install.html

1.2.3 Installing

First, you will need to add the `drf_message` application to the `INSTALLED_APPS` setting in you Django project `settings.py` file.

```
INSTALLED_APPS = [
    # ...
    'django.contrib.auth',
    'django.contrib.sessions',  # optional
    'django.contrib.messages',
    'rest_framework',  # optional
    'django_filters',  # optional
    'drf_messages',
    # ...
]
```

Note: Note that both of Django's `messages` and `auth` contrib apps are required for this module, and the `sessions` is recommended. Make sure to include them in your `INSTALLED_APPS` too.

It is suggested to verify that both `messages`, `auth` and `sessions` middlewares are installed. The configuration should look like so:

```
MIDDLEWARE = [
    # ...
    'django.contrib.sessions.middleware.SessionMiddleware',  # optional
    # ...
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    # ...
]
```

See also:

Read the Django's messages framework docs for more information at https://docs.djangoproject.com/en/3.1/ref/contrib/messages/

Warning: Only database-backed sessions are compatible with this module.

After installing the new app, you will need to **run migration** to create the new database tables:

```
$ py manage.py migrate drf_messages
```

If you have more apps with pending migrations, you may want to omit the `drf_messages` argument and run all pending migrations together.

Next, you will want to **configure** the messages storage to use the `DBStorage` class. This is done using the `MESSAGE_STORAGE` setting in your project's `settings.py` file:

```
MESSAGE_STORAGE = "drf_messages.storage.DBStorage"
```

The last configuration is the addition of the the **messages views** to the router. This is done by including the `drf_messages.urls` to the urlpatterns in your project's `urls.py`.

```
urlpatterns = [
    path('messages/', include('drf_messages.urls')),
]
```

The views can be added anywhere throughout your project, at any path that fits your desires.

Note: This part requires `djangorestframework` to be installed.

1.3 Access Messages

This module keeps the **same interface** of the vanilla Django's messages framework, and can be used just like shown in the Django docs at https://docs.djangoproject.com/en/3.1/ref/contrib/messages/.

1.3.1 Create a new message

Creating a new message is as straight forward as this:

```
from django.contrib import messages

messages.info(request, 'Hello world!')
```

In this example, an information message is created for that `request` object.

Warning: In case the request has **no user or session** provided, the message will be stored in memory for **temporary storage** available through that request processing. Messages added that are not used until the response will not be available anymore.

Extra tags can be attached to the message, as a string or list of strings. For example:

```
from django.contrib import messages

messages.info(request, 'Hello world!', extra_tags="debug")
messages.info(request, 'Hello world!', extra_tags=["debug", "test"])
```

Those extra tags will be save with the message and can be used for filtering, rendering or any other use you can think of.

About the levels

The django's messages framework uses **configurable level architecture** similar to that of the Python logging module. Each message provide a integer level that is represented by a tag, and can be used to group messages by type, filtered or to be rendered differently.

This module keeps the same architecture.

The django's messages framework provides a default of 5 levels. This can be configured using the `MESSAGE_TAGS` setting. For example:

```
MESSAGE_TAGS = {
    10: 'debug',
    20: 'info',
    25: 'success',
    30: 'warning',
    40: 'error',
    50: 'critical',
}
```

Note:

Modifying this setting will affect the `MessageSerializer` and alter the Rest API schema.
Additionally, the Messages admin will list the new tags as select options for level.

When doing so, you will need to **manually** create a new message (the default shortcuts will not suit you). It is advised to store any custom level in a **constant variable** for a more readable code. For example:

```
from django.contrib import messages

CRITICAL = 50

messages.add_message(request, CRITICAL, 'Hello world!')
```

Using those message levels, you can set a **minimum level**, so that any message with a lower level will be **ignored** and will not be saved.

Configuring minimum level can be done at the project level via the `MESSAGE_LEVEL` setting. For example:

```
MESSAGE_LEVEL = 20
# (this is the actual default value)
```

Alternatively, it can be configured per request:

```
from django.contrib import messages

# Change the messages level to ensure the debug message is added.
messages.set_level(request, messages.DEBUG)
messages.debug(request, 'Test message...')

# In another request, record only messages with a level of WARNING and higher
messages.set_level(request, messages.WARNING)
messages.success(request, 'Your profile was updated.') # ignored
messages.warning(request, 'Your account is about to expire.') # recorded

# Set the messages level back to default.
messages.set_level(request, None)
```

See also:

From the django docs https://docs.djangoproject.com/en/3.1/ref/contrib/messages/#changing-the-minimum-recorded-level-per-request

1.3.2 Reading messages

Sometimes it is useful to **access and read** the messages directly in your code.

Accessing the messages can be performed exactly with the **same interface** as the default Django messages framework, but with some extra flairs.

The vanilla ways to access the messages is inside templates:

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

Another classic way is iterating over the messages storage:

```
from django.contrib.messages import get_messages

storage = get_messages(request)
for message in storage:
    print(message)
```

Note: When using the traditional interface specified above, all messages will be **marked as read** immediately.

The storage object behaves almost like any other collection. You can get message at a **specific index**, **slice** it, check its **length**, etc. See more in the reference for *Storage*.

```
from django.contrib.messages import get_messages

storage = get_messages(request)
first_five_messages = storage[:5]
if storage:
    message = storage[0]
```

Alternatively, this module provides a **QuerySet access** to the messages.

It includes **extra information** in the messages, like `created`, `read_at` and `view` to specify the creation time, when read (or null if unread), and the view who submitted the message respectively. Using the QuerySet you will have all it's features like filtering, aggregations, etc.

This can be access through the storage, for example:

```
from django.contrib.messages import get_messages

storage = get_messages(request)
queryset = storage.get_queryset()  # all messages
unread_queryset = storage.get_unread_queryset()  # unread messages only
```

Warning: When using the queryset interface, it is important to **mark as seem** all queried messages after use.

After every access, you will probably want to **mark those messages as read** in order to allow them to be cleared from the database.

This can be done manually like so:

```
from django.contrib.messages import get_messages

storage = get_messages(request)
queryset = storage.get_unread_queryset()
# do something with the messages...
queryset.mark_read()
```

Alternatively, you can use the `with` operator on the storage to mark all messages as read on block exit. For example:

```
from django.contrib.messages import get_messages

storage = get_messages(request)
with get_messages(request) as storage:
    queryset = storage.get_unread_queryset()
    # do something with the messages...
```

Note: When **no session** is available in the request, the messages are saved in a **temporary storage** in memory and can be accessed **only throughout the same request/response process**.

In this scenario, **only legacy interface** is available. That means all queryset related features, such as `get_queryset()`, `get_unread_queryset()`, `mark_read()` and the `with` operator will not do practically anything.

1.3.3 Deleting messages

When using a persistent message storage, it is important to implement procedure for **clearing out** old messages.

When using sessions, messages get cleared automatically only when the **appropriate session is deleted** from database due to user logout or `clearsessions` command.

This behavior is not affected by the `MESSAGES_USE_SESSIONS` setting. As long as there is a session provided with the request, all the messages will be cleared when the session is cleared.

Note: Make sure to regularly run the `clearsessions` command to delete any expired session and clear stale messages. See more at the django docs https://docs.djangoproject.com/en/3.1/topics/http/sessions/#clearing-the-session-store

If you are **not using Session Authentication**, it is advised to setup a manual message clearing procedure, such as a scheduled deletion of all read messages created before a certain time.

Additionally, you may want to configure the `MESSAGE_DELETE_READ` setting to `True` at your project's `settings.py` file. This setting will cause any read message to be **deleted just after the request is done processing**.

Another way is to **delete messages manually** in you code. This can be done using the QuerySet interface to messages:

```
from django.contrib.messages import get_messages

storage = get_messages(request)
queryset = storage.get_queryset()
# delete only messages that have already been read
queryset.filter(read_at__isnull=False).delete()
```

Note: Make sure not to delete unread messages before the user gets a chance or getting them…

1.4 Rest API Views

Provided with this app is a DRF ViewSet that provides clients to access their messages.

Through those endpoints, clients can **list** all of their messages (read and unread), **retrieve** a single message, and **delete** a message.

1.4.1 Endpoints

list GET - List all messages for this context. (`drf_messages:messages-list`)

```
$ curl -X GET "http://127.0.0.1/messages/"
```

peek GET - Get summary of unread messages for this context. (`drf_messages:messages-peek`)

```
$ curl -X GET "http://127.0.0.1/messages/peek/"
```

retrieve GET - Retrieve specific message from this context. (`drf_messages:messages-detail`)

```
$ curl -X GET "http://127.0.0.1/messages/{id}/"
```

delete DELETE - Delete a specific message from this context. (`drf_messages:messages-detail`)

```
$ curl -X DELETE "http://127.0.0.1/messages/{id}/"
```

Note: By default, clients are **not allowed** to delete messages that are unread. You can change this behavior by setting the `MESSAGES_ALLOW_DELETE_UNREAD` to `True` in your project's settings.

1.4.2 List Filters

This module includes a predefined `django-filter` FilterSet. To use it, simply install `django-filter` in your project and the filters will be added automatically.

Install using pip with:

```
$ pip install django-filter
```

Then add 'django_filters' to your INSTALLED_APPS:

```
INSTALLED_APPS = [
    # ...
    'django_filters',
    # ...
]
```

See also:

To install properly, follow the installation docs at https://django-filter.readthedocs.io/en/stable/guide/install.html

The filters included are:

unread Boolean Filter *(true/false)*, show new messages, and vice versa.

level_tag Text Filter, minimum message level to show (similar to Python logging handler level).

level Integer Filter, show messages filtered by level (with integer lookups).

extra_tags Text Filter, messages with specific extra tag (with text lookups).

view Text Filter, messages from specific view.

read_before/after Date & Time Filter, message read between date and time range.

created_before/after Date & Time Filter, message created between date and time range.

1.4.3 Customize the views

Those views does not specify any permission classes, authentication classes, filters, pagination, versioning or any other optional extension.

Warning: Users can access only **their messages** or the messages of their **current session** when `MESSAGES_USE_SESSIONS` is configured to `True`. Unauthenticated users can practically access the endpoints, but will always receive an empty list or error 404.

Providing a permission class like `IsAuthenticated` is a good practice, but is not mandatory.

There are mainly two ways to customize those settings, configuring default settings for DRF or creating a custom ViewSet of your own.

Configuration of defaults for your views is done using the `REST_FRAMEWORK` setting in your project's `settings.py` file. For example:

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.SessionAuthentication',
        'rest_framework.authentication.TokenAuthentication',
    ),
    'DEFAULT_FILTER_BACKENDS': (
        'django_filters.rest_framework.DjangoFilterBackend',
        'rest_framework.filters.SearchFilter',
        'rest_framework.filters.OrderingFilter',
    ),
    'DEFAULT_VERSIONING_CLASS': 'rest_framework.versioning.NamespaceVersioning',
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
```

(continues on next page)

(continued from previous page)

```
    ),
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.LimitOffsetPagination',
    'PAGE_SIZE': 10,
}
```

Note: Note that `django_filters` is included in this example, and needs to be installed before use.

See also:

See more in the Django Rest Framework docs https://www.django-rest-framework.org/api-guide/settings/

Alternatively, you can create your oen version of the `MessagesViewSet` and use it instead.

First at your `views.py` create a new ViewSet that extends the `MessagesViewSet` class.

```
from rest_framework.pagination import LimitOffsetPagination
from rest_framework.permissions import IsAuthenticated
from django_filters.rest_framework import DjangoFilterBackend

from drf_messages.views import MessagesViewSet


class MyMessagesViewSet(MessagesViewSet):
    permission_classes = (IsAuthenticated,)
    pagination_class = (LimitOffsetPagination,)
    filter_backends = (DjangoFilterBackend,)
```

Then at your `urls.py` create a router, register your custom view, and attach it to the `urlpatterns`. For example:

```
from rest_framework.routers import DefaultRouter

from myapp.views import MyMessagesViewSet

router = DefaultRouter()
router.register("messages", MyMessagesViewSet, "messages")


app_name = "myapp"
urlpatterns = [
    *router.urls,
]
```

1.5 Settings

1.5.1 MESSAGES_USE_SESSIONS

Type `bool`; Default to `False`; Not Required.
Use session context to query messages.

Query messages for current session only. When is set to `True`, only messages created from the **same session** will be shown.

By default (`False`), messages are queried only by the **authenticated user**. That means the user can see all their messages from **all sessions**.

Relating messages to session is different according to your configured Session Engine. For the most part, the `session_key` string is used to filter the query. When is available, the `Session` model object is used as *ForeignKey* and is also used to filter the query.

Note: When using a session engine that works with db `Session` model, you unlock extra functionality that **automatically clears out messages** after user logout or clearsessions command.

Tested session engines:

- `django.contrib.sessions.backends.db` (uses db)
- `django.contrib.sessions.backends.file`
- `django.contrib.sessions.backends.cache`
- `django.contrib.sessions.backends.cached_db` (uses db)
- `django.contrib.sessions.backends.signed_cookies`
- `redis_sessions.session` (django-redis-sessions)

1.5.2 MESSAGES_ALLOW_DELETE_UNREAD

Type `bool`; Default to `False`; Not Required.
Allow users to delete unread message of their own.

By default, users are **forbidden to delete** any unread messages through the Rest API endpoint. Setting this value to `True` will allow users to delete messages haven't already been read.

Note: Users (or rather sessions) have access only to **their messages** only and cannot delete messages that are not their own regardless of this setting.

1.5.3 MESSAGES_DELETE_READ

Type `bool`; Default to `False`; Not Required.
Automatically delete read messages.

When this setting is set to `True` each message will be **delete just after it is read**. This behavior is useful to minimize storage space used by the messages.

When set to `False`, messages will be deleted either manually or when the appropriate session is cleared.

1.6 Storage

Storage can be access using `django.contrib.message.get_messages` method. For example:

```
from django.contrib.messages import get_messages

storage = get_messages(request)
```

1.6.1 Attributes

level Minimum message level. Can be modified to specify custom minimum message level. For example:

```
from django.contrib import messages

messages.set_level(request, messages.DEBUG)
# is the same as
storage = messages.get_messages(request)
storage.level = messages.DEBUG
```

used Indicates the storage was used. You may set to `False` to avoid the message deletion procedure.

```
from django.contrib import messages

storage = messages.get_messages(request)
for message in storage:
    print(message)

storage.used = False
```

See also:

More information in the django's messages framework docs https://docs.djangoproject.com/en/3.1/ref/contrib/messages/#expiration-of-messages

1.6.2 Python Access

Storage object provide many syntax based access that is similar to other **Python collections**. All access via those interfaces is for **unread messages**, and some of them also **mark them as read**.

Example	Read	Explanation
`[m for m in storage]`		Retrieve messages lazily
`storage[1]`		Get specific message from storage
`storage[:5]`		Get subset of messages (slice)
`if storage:`		Check if there are unread messages
`len(storage)`		Get count if unread messages
`Message() in storage`		Check if a message exists and is unread
`"Message Body" in storage`		Check if an unread message with that text exists
`messages.INFO in storage`		Check if an unread message with that level exists
`str(storage)`		Get all messages, divided by comma
`repr(storage)`		Get all messages, divided by comma

Warning: When **iterating** over storage, the marking of messages as read is done after iteration is over. When iteration is complete **all unread messages will be marked as read**, whether they were returned during iteration or not.

1.6.3 Methods

get_queryset() Get queryset of all messages for that request.

get_unread_queryset() Get queryset of unread messages for that request.

add(level, message, extra_args) Add a new message to the storage.

update(response) Perform deleting procedure manually.

1.7 Models

1.7.1 Message

Fields:

id Integer, ID.

session Session, related sessions.Session object.

message String (up to 1024), the actual text of the message.

level Integer, describing the type of the message.

extra_tags.all List, all related drf_messages.MessageTag objects.

view String (up to 64), the view where the message was submitted from.

read_at Date (with time), when the message was read (or null).

created Date (with time), when the message was crated

Properties:

level_tag String, describing the level of the message

Methods:

add_tag Add extra tag (or multiple tags)

mark_read Mark message as read now

get_django_message Parse message to django message object (`django.contrib.messages.storage.base.Message`)

1.7.2 MessageTag

Fields:

id Integer, ID.

message Message, related drf_messages.Message object.

text String (up to 128), custom tags for the message.

1.7.3 MessageManager

Accessed via `Message.objects`.

Methods:

create_message(request, message, level, extra_tags) Create a new message in database.

create_user_message(request, message, level, extra_tags) Create a new message in database for a user.

with_context(request) QuerySet of messages filtered to a request context.

1.7.4 MessageQuerySet

Accessed via `Message.objects.with_context(request)`. Alternative access via `messages.get_messages(request).get_queryset()`.

Methods:

mark_read() Mark messages as read now.

1.8 Change Log

1.8.1 1.1.1

Release date: 9 Dec, 2021

- **ADDED** Support for all session engines (not only DB engine) See docs for **:docs:`settings_reference`**

> **Warning:** This version **requires migration** after upgrade from older version

1.8.2 1.1.0

Release date: 2 Dec, 2021

- **ADDED** Support for Django 3.2
- **NEW** Methods for Message model. See docs for *Models*
- **NEW** Collection-like interface for Storage objects. See docs for *Storage*
- **NEW** CI/CD for testing, linting and deploying
- **BUG FIX** Messages incorrectly marked read after using list endpoint
- **BUG FIX** Errors when querying messages from request without a session

1.8.3 1.0.1

Release date: 24 Feb. 2021

- **NEW** Peak messages endpoint

1.8.4 1.0.0

Release date: 13 Feb. 2021

- First published version