djem Documentation

Release 0.9.0a6

Alex Church

Nov 20, 2023

CONTENTS

Djem is a collection of useful stuff for Django projects.

CHAPTER

ONE

REQUIREMENTS

- Django 2.2+
- pytz is required to make use of *models.TimeZoneField*, *forms.TimeZoneField* and *TimeZoneHelper*.

CHAPTER

TWO

INSTALLATION

Install the latest stable version from PyPI:

```
pip install djem
```

CHAPTER

THREE

FEATURES

3.1 Object-Level Permissions

Django's permissions framework has the foundation for, but no implementation of, object-level permissions (OLP). The permissions it supports apply to *all* records of a particular model - hence they are "model-level" permissions (MLP). For example, using the standard Django "polls" application to illustrate, you can use the Django permissions framework to determine if any given user can change `Question`s, but not to determine if they can change a given `Question` in particular.

Djem provides a very simple implementation of an OLP system, described in the following sections:

3.1.1 Introduction

Djem's object-level permission (OLP) system provides following features:

- As with model-level permissions, an object-level permission may be granted to a user based on the `User` object itself, or based on a `Group` to which the user belongs.
- Permissions are only checked at the object level if the user has the model-level permission. That is, a user must be able to change `Question`s in general if they are to be granted permission to change a particular `Question`.
- Methods on the object itself grant or deny the permission, based on self-contained logic. The database is not required to store links between users/groups and individual model objects.

Note: Djem's OLP system is simply an extension of the default Django permissions system. As such, it expects the `User` model to have the same attributes and methods that power that system. If you are using a custom user model (as is recommended), it should either inherit from `AbstractUser` or include the `PermissionsMixin` to be compatible. See the Django documentation for custom user models and permissions.

Enabling

Basic usage of object-level permissions is enabled simply by including the custom *djem.auth.ObjectPermissionsBackend* authentication backend in the `AUTHENTICATION_BACKENDS` setting:

```
AUTHENTICATION_BACKENDS = [
    'django.contrib.auth.backends.ModelBackend',
    'djem.auth.ObjectPermissionsBackend'
]
```

See the Django documentation on authentication backends for more information.

More advanced features require additional steps. These are outlined in *Advanced Features*.

Supported permissions

Any existing permission can be used with the OLP system, though it may not make sense for all of them. For example, Django provides default "add" permissions for all models. It doesn't make sense for adding to involve object-level permissions, as no object would yet exist on which to *check* for an "add" permission. That being said, the OLP system contains no logic preventing you from using *any* permission at the object level.

If the default Django-provided permissions ("add", "change" and "delete") aren't enough, you can add custom permissions via the `permissions` attribute of a model's inner `Meta` class, as per the Django documentation.

Any permissions added this way are automatically supported by the OLP system. You just need to define the necessary methods on the model class, *as described below*. And remember: a user must have the standard, model-level permission before object-level permissions will even be checked.

Defining permissions

Object-level permissions are ultimately determined by specially-named methods on the object in question. These are the object-level *access methods*. The two types of access methods are:

- `_user_can_<permission_name>(self, user)`: Grant/deny permission based on the given `User` instance by returning `True` or `False`, respectively.
- `_group_can_<permission_name>(self, groups)`: Grant/deny permission based on the given `Group` queryset by returning `True` or `False`, respectively.

For the Django default "change" permission on the `polls.Question` model, the method names would be: `_user_can_change_question()` and `_group_can_change_question()`.

When defining custom permissions, the permission name used in the method names must be the same as that provided in the `permissions` attribute of the model's `Meta` class. If *either* of the methods returns `True`, the user is granted the permission.

The following example demonstrates how to define a model that uses object-level permissions for a custom permission. It uses a modified version of the `Question` model created in the Django tutorial that only allows voting by explicitly defined users.

```
from django.conf import settings
from django.db import models

class Question(models.Model):

    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')
    allowed_voters = models.ManyToManyField(settings.AUTH_USER_MODEL)

    def _user_can_vote_on_question(self, user):

        return self.allowed_voters.filter(pk=user.pk).exists()

    class Meta:
        permissions = (('vote_on_question', 'Can vote on question'),)
```

Note: The *ObjectPermissionsBackend* handles calling these methods when necessary - they should never need to be called manually. See *Checking Permissions*.

Note: These object-level access methods can raise `PermissionDenied` and it will be treated as if they returned `False`. Regardless of whether the user-based or group-based check raises the exception, the other could still grant the permission.

Permissions default open

An important concept in Djem's OLP system is that permissions default *open* at the object level. That is to say, unless explicit logic is given to dictate how an object-level permission should be granted/denied, it is assumed to be granted. As such, an OLP check on an object with no defined object-level access methods is equivalent to a model-level check for the same permission.

This makes the system interchangeable with the default Django permissions system. Common code can check permissions at the object level and will be unaffected if no object-level access control exists for a given model - it doesn't need to pick and choose whether to use object-level or model-level permission checking.

3.1.2 Checking Permissions

The main ways of using the OLP system to check a user's permissions on a specific object are:

- the `permission_required` decorator for function-based views or `PermissionRequiredMixin` mixin for class-based views
- the `ifperm` and `ifnotperm` template tags
- the `has_perm()` method on a `User` instance

All of these approaches use the standard Django permissions system and rely on the custom *ObjectPermissionsBackend* to call the appropriate object-level access methods. In the examples below, each permission check will result in `_user_can_<permission_name>` being called and provided the `User` instance involved in the check, and `_group_can_<permission_name>` being called and provided with a queryset of all `Groups` to which that user belongs. Either method can return `True` to grant the user permission.

Warning: The object on which a permission is checked *is not verified*. That is, you could check the `polls.vote_on_question` permission on an instance of *any* random model and no warning would be given (checking validity would add unnecessary overhead to such a common operation). This is important because, if the instance provided does not define the appropriate object-level access methods, the permission is assumed to be *granted* at the object level, since *permissions default open*.

Many of the below examples draw on the sample `polls.Question` model introduced in the *documentation on defining access methods*:

```
from django.conf import settings
from django.db import models

class Question(models.Model):
```

(continues on next page)

(continued from previous page)

```
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')
    allowed_voters = models.ManyToManyField(settings.AUTH_USER_MODEL)

    def _user_can_vote_on_question(self, user):

        return self.allowed_voters.filter(pk=user.pk).exists()

    class Meta:
        permissions = (('vote_on_question', 'Can vote on question'),)
```

Protecting views

Protecting views that should only be accessed by users with certain object-level permissions is supported by Djem's extensions of the standard Django `permission_required` decorator (for function-based views) and `PermissionRequiredMixin` mixin (for class-based views).

See the Django documentation for the decorator and the mixin for the basic functionality these helpers provide.

Checking an object-level permission involves querying for an instance of the model the permission is for. If such an instance cannot be found, a `Http404` exception is raised.

New in version 0.5: The *permission_required()* decorator and the *PermissionRequiredMixin* class-based view mixin.

Basic usage

Usage of Djem's *permission_required()* and *PermissionRequiredMixin* is very similar to the originals except that specifying an object-level permission is done using a tuple of two strings: the first naming the permission, the second naming the view function keyword argument that contains the primary key of the object to test.

Model-level permissions can still be checked by specifying a plain string as per usual.

A mixture of multiple model-level and object-level permissions is also fully supported. In this case, permissions are checked in the order they are listed, and a user must pass every check in order to access the view.

```
from django.views import View
from djem.auth import PermissionRequiredMixin, permission_required

# Check a model-level permission on a function-based view
@permission_required('polls.view_questions')
def cast_vote(request, question):
    ...

# Check an object-level permission on a function-based view
@permission_required(('polls.vote_on_question', 'question'))
def cast_vote(request, question):
    ...

# Check a mixture of permissions on a function-based view
@permission_required('polls.view_questions', ('polls.vote_on_question', 'question'))
def cast_vote(request, question):
```

(continues on next page)

(continued from previous page)

```
    ...

# Check a model-level permission on a class-based view
class CastVote(PermissionRequiredMixin, View):

    permission_required = 'polls.view_questions'
    ...

# Check an object-level permission on a class-based view
class CastVote(PermissionRequiredMixin, View):

    permission_required = [('polls.vote_on_question', 'question')]
    ...

# Check a mixture of permissions on a class-based view
class CastVote(PermissionRequiredMixin, View):

    permission_required = ['polls.view_questions', ('polls.vote_on_question', 'question
↪')]
    ...
```

Note: When specifying a single object-level permission using the `permission_required` attribute of *PermissionRequiredMixin*, it must be given as an item of a sequence (e.g. a list). While a single model-level permission can be provided as either a plain string *or* a single-item sequence (a feature inherited from Django's own mixin class), because an object-level permission is defined in a tuple - which is itself an sequence - it would be treated as two model-level permissions (and would be invalid).

`PermissionRequiredMixin` and the URLconf

Like all attributes of class-based views, the `permission_required` attribute added by *PermissionRequiredMixin* can be specified/overridden in the URLconf:

```
from django.conf.urls import url
from .views import CastVote

urlpatterns = [
    url(
        r'^question/(?P<question>\d+)/votes/cast/$',
        CastVote.as_view(permission_required=['polls.view_questions', ('polls.vote_on_
↪question', 'question')]),
        name='cast-vote'
    )
]
```

Controlling denied access

As with the originals, `login_url` and `raise_exception` are fully supported - as arguments to the *permission_required()* decorator or as attributes of a class inheriting from *PermissionRequiredMixin*. These allow additional control over what happens when a user is denied access to a view protected by the decorator/mixin. See the Django documentation for permission_required for more info on how these features work.

Djem extends this control slightly with the *DJEM_DEFAULT_403* setting. This setting can be used to control the *default value* of `raise_exception`. Django's default is to NOT raise the `PermissionDenied` exception, preferring to redirect to the login view, but allowing you to override this behaviour per-view using `raise_exception`. Setting *DJEM_DEFAULT_403* to `True` allows you to configure all protected views to raise the `PermissionDenied` exception by default, while still allowing per-view customisation with `raise_exception`.

Argument replacement

When using either *permission_required()* or *PermissionRequiredMixin* to protect a view using object-level permissions, and the user passes all permission checks, any arguments named in an OLP two-tuple will be replaced with the appropriate instances.

These instances will already have been queried in order to check the user's permissions against them, so they are injected into the view's keyword arguments, replacing those that specified the primary key used in the queries. This allows the view to access such instances without needing to query for them again.

In the "cast vote" view examples used above, the view accepts a `question` keyword argument. This argument is named as the source of the primary key of a `Question` record, and used to check the user's `polls.vote_on_question` permission against that specific `Question`. While the view was originally passed the *primary key* of a `Question` (as controlled by the URLconf), this is used and replaced as part of the permissions check, and the view sees a `Question` *instance*.

Checking in templates

Checking object-level permissions in a Django template can be done using the *ifperm* and *ifnotperm* template tags. These are block tags whose content is displayed if the permissions check passes. For *ifperm*, it passes if the user *has* the permission. For *ifnotperm*, it passes if the user *does not* have the permission. Each tag supports an `else` block, whose content is displayed if the permissions check fails.

Each tag must be passed a user instance, the name of the permission to check and the object to check it on.

```
{% load djem %}
...
{% ifperm user 'polls.vote_on_question' question_obj %}
    <a href="{% url 'vote' question_obj.pk %}">Vote Now</a>
{% else %}
    You do not have permission to vote on this question.
{% endifperm %}
...
```

```
{% load djem %}
...
{% ifnotperm user 'polls.vote_on_question' question_obj %}
    You do not have permission to vote on this question.
{% else %}
    <a href="{% url 'vote' question_obj.pk %}">Vote Now</a>
```

(continues on next page)

(continued from previous page)

```
{% endifnotperm %}
...
```

Checking via `User` instances

The `has_perm()` method provided by Django's `User` model (or by `PermissionsMixin` if using a custom user model) accepts an optional `obj` argument. Django does nothing with it by default, but passing it will invoke Djem's OLP system. Thus it can be used to check a user's object-level permissions on a given object.

Continuing with the modified `Question` model defined above:

```
>>> user = User.objects.get(username='alice')
>>> question = Question.objects.filter(voters=user).first()
>>> user.has_perm('polls.vote_on_question', question)
True

>>> question = Question.objects.exclude(voters=user).first()
>>> user.has_perm('polls.vote_on_question', question)
False
```

See `has_perm()` documentation for User and PermissionsMixin.

Other `PermissionsMixin` methods

The OLP system is fully compatible with Django's `PermissionsMixin`, meaning it supports more than just the `has_perm()` method. Other supported methods include:

- `has_perms()`: For checking multiple permissions against a particular object at once.
- `get_all_permissions()`: To obtain a list of all permissions accessible to the user, either directly or via their groups, with all necessary object-level logic applied.
- `get_group_permissions()`: To obtain a list of all permissions accessible to the user via their groups only, with all necessary object-level logic applied.

While `has_perms()` is a simple extension of `has_perm()` to allow checking multiple permissions at once, some care should be taken with `get_all_permissions()` and `get_group_permissions()`.

Firstly, depending on the number of permissions your project uses, the amount that have object-level access methods defined, and the complexity of the logic used by those access methods, obtaining a list of available permissions could involve a lot of processing (compared to testing one at a time).

More subtly, `get_group_permissions()` can potentially list permissions that would not actually be granted to the user via a standard permissions checking. This is a side-effect of the fact that *object-level permissions default open*. If a user-based object-level access method denied a certain permission, and no group-based access method was defined, a normal permissions check would return `False`, on account of the user-based check. But a group-only check, such as performed by `get_group_permissions()` would *grant* the permission, due to there being no object-level access method to indicate otherwise.

While not accessible via `PermissionsMixin`, *`ObjectPermissionsBackend`* also contains a `get_user_permissions()` method which suffers from the same side-effect due to ignoring group-based access methods.

Caching

Like `ModelBackend` does for model-level permissions, *`ObjectPermissionsBackend`* caches object-level permissions on the `User` object after the first time they are checked. Unlike `ModelBackend`, the user's entire set of object-level permissions are not determined and cached on this first access, only the specific permission being tested, for the specific object given.

This caching system has the same advantages and disadvantages as that used at the model level. Multiple checks of the same permission (on the same object) in the same request will only need to execute the (possibly expensive) logic in your object-level access methods once. However, that means that if something changes within the request that would alter the state of a permission, and that permission has already been checked, the `User` object will not immediately reflect the new state of the permission. Exactly what *might* affect the state of a permission depends entirely upon the logic implemented in the `_user_can_<permission_name>()`/`_group_can_<permission_name>()` methods, so this is something to be aware of both while writing these methods and while using them.

Clearing the cache is possible by querying for a new instance of the `User` or, depending on how your user model is configured, *using the cache-clearing helper method*.

3.1.3 Advanced Features

Basic use of Djem's OLP system is a simple drop-in extension of Django's own permissions system, enabled by *`ObjectPermissionsBackend`*. If your user model, no matter how it is defined, is compatible with Django's default permissions system, it will be compatible with the OLP system as well.

However, more advanced features are available that require a higher level of configuration. Specifically, they *require* a custom user model - they will not be available if simply making use of Django's included `auth.User` model. Django recommends using a custom user model anyway (for new projects, at least), even if it doesn't actually customise anything.

To enable these advanced features, described below, your custom user model must include the *`OLPMixin`*.

If not looking to actually customise anything, a custom user model incorporating *`OLPMixin`* is as simple as:

```
from django.contrib.auth.models import AbstractUser

from djem.models import OLPMixin


class User(OLPMixin, AbstractUser):

    pass
```

If looking to customise the user model more heavily (for example, using an email address instead of a username as the user's identification token), use something like the following:

```
from django.contrib.auth.models import AbstractBaseUser, PermissionsMixin

from djem.models import OLPMixin


class User(OLPMixin, AbstractBaseUser, PermissionsMixin):

    ...
```

New in version 0.7: *`OLPMixin`*

Important: *OLPMixin* must be listed *ahead* of `AbstractUser`/`PermissionsMixin` in order for it to work correctly.

Superusers

The Django permissions system automatically grants any and all permissions to a `User` instance with the `is_superuser` flag set to `True`. By default, this is how the OLP system operates as well: no object-level access methods are executed, the superuser is simply granted the permission.

There are, however, situations in which this is not desirable. For example, you may want to define a model that does not grant the "delete" permission to anyone but the user that created it, no matter how "super" the user is. It would be trivial to configure the model to achieve this for a standard user, but a superuser would bypass any custom object-level access methods and be granted the permission anyway.

Djem provides a means of forcing superusers to be subject to the same OLP logic as regular users. They are still implicitly granted all permissions at the model level, but any object-level access methods *will* be executed and *can* deny the user permission.

In order to enable this feature, two things are required:

- A custom user model including *OLPMixin*, as described above.
- The *DJEM_UNIVERSAL_OLP* setting set to `True`.

With these two requirements met, object-level permissions will be applied "universally", including for superusers.

Note: Enabling this feature will cause superusers to be subject to the OLP logic for *all* permissions that define some. If your project contains permissions which should still be granted to superusers regardless of the additional checks that standard users are subject to, the relevant access method can include a simple guard clause:

```
def _user_can_vote_on_question(self, user):

    if user.is_superuser:
        return True

    # Do custom logic
    ...
```

Clearing the permission cache

As described in *Caching*, the results of object-level permission checks are cached, which has the downside of the results potentially getting out-of-date if elements of the state used to determine the permission are changed.

By default, the only way to clear this cache is to re-query for a new user instance. This is particularly annoying if needing to replace the user instance on the `request` object. *OLPMixin* provides a *clear_perm_cache()* method, which, as the name suggests, clears the permissions cache on the user instance.

In addition to clearing the OLP cache, *clear_perm_cache()* also clears Django's model-level permissions caches, for good measure.

Automatically logging permission checks

OLPMixin leverages instance-based logging to support automatically logging all permission checks made via its overridden *has_perm()* method - both model-level and object-level.

Read the *documentation for the instance-based logging functionality* provided by *Loggable* for an introduction to the system. *OLPMixin* inherits from *Loggable*, and thus offers all the same features, in addition to those specific to permissions.

There are multiple levels of automatic permission logging available, controlled via the *DJEM_PERM_LOG_VERBOSITY* setting:

- `0`: No automatic logging
- `1`: Logs are automatically created for each permission check, with minimal automatic entries
- `2`: Logs are automatically created for each permission check, with more informative automatic entries

Using a setting above `0` configures *has_perm()* to create an appropriately-named log and populate it with automated entries as appropriate (based on the verbosity level chosen). In addition to the automated entries, having a suitable log already created and active provides a simpler experience if utilising logging in object-level access methods. Revisiting the "delete_product" access method described in the *instance-based logging examples*, enabling automatic logging allows for a simpler method definition:

```
class Product(models.Model):

    code = models.CharField(max_length=20)
    name = models.CharField(max_length=100)
    active = models.BooleanField(default=True)
    supplier = models.ForeignKey(Supplier, on_delete=models.PROTECT)

    def _user_can_delete_product(self, user):

        if self.active:
            user.log('Cannot delete active product lines')
            return False
        elif get_quantity_in_stock(self):
            user.log('Cannot delete products with stock on hand')
            return False
        elif not user.has_perm('inventory.manage_supplier', self.supplier):
            inner_log = user.get_last_log(raw=True)
            user.log(*inner_log)
            return False

        user.log('Product can be deleted')
        return True
```

Log output

Using the "delete_product" permission from the above `Product` model as a reference, the output of a permission check with a *DJEM_PERM_LOG_VERBOSITY* setting of `1` might look something like:

```
Model-level Result: Granted

Cannot delete active product lines

RESULT: Permission Denied
```

And with a *DJEM_PERM_LOG_VERBOSITY* of `2`:

```
Permission: inventory.delete_product
User: user.name (54)
Object: PROD123 (1375)

Model-level Result: Granted

Cannot delete active product lines

RESULT: Permission Denied
```

Tags

Automatically generated log entries *utilise tagging*, and are given the `'auto'` tag. This allows them to be easily identified and filtered out if desired. Reproducing the above high-verbosity log output, highlighting which lines are tagged, gives:

```
[tag:auto] Permission: inventory.delete_product
[tag:auto] User: user.name (54)
[tag:auto] Object: PROD123 (1375)
[tag:auto]
[tag:auto] Model-level Result: Granted
[tag:auto]
Cannot delete active product lines
[tag:auto]
[tag:auto] RESULT: Permission Denied
```

Log names

Retrieving automatically generated permission logs via *get_log()* requires knowing their name. As long as you know the name of the permission that was checked, and the primary key of the object it was checked against (where applicable), the name of the log can be easily determined:

For model-level permission checks: `auto-<permission_name>` (e.g. `auto-inventory.delete_product`)

For object-level permission checks: `auto-<permission_name>-<object_id>` (e.g. `auto-inventory.delete_product-1375`)

3.2 Instance-based Logging

The *Loggable* mixin provides a series of methods for creating, storing, and retrieving "instance-based" logs - that is, logs that are specific to an instance of a class incorporating *Loggable*. These logs can be used for anything - recording events, detailing the steps taken by a complex process, etc. Each log is created with its own name, and can be later retrieved from the instance using that name. The logs themselves are maintained as Python lists - each item in the list being a separate entry, or line, in the log. This allows them to be easily appended to over time.

When a new log is created, it becomes the "active" log for that instance - the one to which new log entries are appended. There can only be one active log on an instance at a time. If another log is active when a new log is created, it gets pushed back in the queue, and will become active again when the new log is finished. Logs must be explicitly declared "finished" in order to both re-activate previous logs and also to enable retrieving the log (unfinished logs cannot be retrieved).

The logs are stored internally on the instance. They will persist for the lifetime of that instance. No support for more persistent storage of these logs is included, but *it is possible*.

The methods available are:

- *start_log()*: Create a new log with a given name, making it the active log.
- *log()*: Add a new entry to the active log.
- *end_log()*: Mark the active log as finished.
- *discard_log()*: Remove the active log.
- *get_log()*: Retrieve a log with the given name.
- *get_last_log()*: Retrieve the most recently finished log.

By default, *get_log()* and *get_last_log()* retrieve logs as strings, but copies of the internal lists can be retrieved by passing `raw=True` to either method.

OLPMixin, used to provide advanced features to Djem's *object-level permissions system*, inherits from *Loggable*. Keeping user-based logs of permission checks is the primary use of instance-based logging. The examples in this documentation use user-based logging in object-level permission access methods to illustrate the supported features of *Loggable*. While *OLPMixin* provides support for *automatically logging permission checks*, these examples assume that feature is disabled and demonstrate the basic functionality of the system.

Consider a `Product` model where a user may not be allowed to delete a product that is currently an actively-sold product line or if the product currently has any stock on hand. These restrictions can be imposed by an object-level access method, and a log can allow auditing why the permission was not granted:

```
from django.db import models

from .utils import get_quantity_in_stock


class Product(models.Model):

    code = models.CharField(max_length=20)
    name = models.CharField(max_length=100)
    active = models.BooleanField(default=True)

    def _user_can_delete_product(self, user):

        user.start_log('delete-product-{0}'.format(self.pk))

```

(continues on next page)

(continued from previous page)

```
        if self.active:
            user.log('Cannot delete active product lines')
            allowed = False
        elif get_quantity_in_stock(self):
            user.log('Cannot delete products with stock on hand')
            allowed = False
        else:
            user.log('Product can be deleted')
            allowed = True

        user.end_log()

        return allowed
```

3.2.1 Nested logs

As mentioned, if a log is already active and a new log is created, the new log becomes active and the previous log will be *reactivated* when the new log is finished. This allows for logs to be nested without worrying about interfering with other logs.

Continuing the above `Product` example, deletion may also require that the user has permission to manage the inventory from the supplier in question. Thus, checking the "delete_product" permission can trigger a "manage_supplier" permission check. If both permissions perform logging, those logs are nested, but neither need to do anything special to handle it:

```
from django.conf import settings
from django.db import models

from .inventory import get_quantity_in_stock


class Supplier(models.Model):

    name = models.CharField(max_length=100)
    managers = models.ManyToManyField(settings.AUTH_USER_MODEL)

    def _user_can_manage_supplier(self, user):

        user.start_log('manage-supplier-{0}'.format(self.pk))

        if not self.managers.filter(pk=user.pk).exists():
            user.log('You do not have permission to manage the inventory of {0}'.
↪format(self.name))
            allowed = False
        else:
            allowed = True

        user.end_log()

        return allowed
```

(continues on next page)

(continued from previous page)

```
    class Meta:
        permissions = (
            ('manage_supplier', "Can manage inventory from this supplier")
        )


class Product(models.Model):

    code = models.CharField(max_length=20)
    name = models.CharField(max_length=100)
    active = models.BooleanField(default=True)
    supplier = models.ForeignKey(Supplier, on_delete=models.PROTECT)

    def _user_can_delete_product(self, user):

        user.start_log('delete-product-{0}'.format(self.pk))

        if self.active:
            user.log('Cannot delete active product lines')
            allowed = False
        elif get_quantity_in_stock(self):
            user.log('Cannot delete products with stock on hand')
            allowed = False
        elif not user.has_perm('inventory.manage_supplier', self.supplier):
            user.log('You do not have permission to manage the inventory of {0}'.
↪format(self.supplier.name))
            allowed = False
        else:
            user.log('Product can be deleted')
            allowed = True

        user.end_log()

        return allowed
```

Of course, sometimes it *is* useful to explicitly handle nested logs. Instead of duplicating the same log entry in both the "manage_supplier" and "delete_product" access methods, "delete_product" can simply incorporate the log generated by "manage_supplier" into its own, using *get_last_log()*:

```
def _user_can_delete_product(self, user):

    ...
    elif not user.has_perm('inventory.manage_supplier', self.supplier):
        inner_log = user.get_last_log(raw=True)
        user.log(*inner_log)
        allowed = False
    ...
```

3.2.2 Tagging log entries

In some cases, the *entire* log may not be relevant. For example, you may want to display a concise reason for a permission check failure, while ignoring any additional log entries (such as those added when using *automatic permission check logging*). In such cases, it is possible to tag individual log entries and then retrieve only those entries with a given tag.

Tagging can be done by passing the `tag` argument to *`log()`*. The named tag will be applied to all lines added with the same call. The lines with the tag can then be retrieved using *`get_log()`* or *`get_last_log()`*, using the `tags` argument. E.g.:

```
def some_process(user):

    user.start_log('some-process')
    user.log('Starting the process')

    result = do_something()
    if not result:
        user.log('No result!', tag='error')

    user.log('Ending the process')
    user.end_log()

    return result
```

After calling the above function, the log can be retrieved using:

```
# Full log
>>> user.get_log('some-process')
'Starting the process\nNo result!\nEnding the process'

# Only the error
>>> user.get_log('some-process', tags=['error'])
'No result!'
```

Note that if multiple tags are passed to *`get_log()`* or *`get_last_log()`*, lines with *any* of the given tags will be returned.

3.2.3 Duplicate names

Creating a new log with the same name as another *unfinished* log is not possible - it will raise a `ValueError`. Therefore, it is important to use sufficiently informative names. The above nested logging example uses two logs named after the permission they are checking and the object they are checking it on. If they were both to use the same generic name, e.g. `'perm-log'`, attempting to call one from within the other would fail.

However, it is perfectly valid to re-use a name after the log has been finished. Doing so will overwrite the previous log when the new one is finished. This allows the same process to be run multiple times within the lifetime of the *`Loggable`* instance, without causing issues. But it is important to note that only the latest version of the log will be kept.

3.2.4 Persistence

As noted above, logs are stored on the *Loggable* instance itself. This can influence the types of classes that it makes sense to incorporate *Loggable* into. It may not be useful to use the mixin on classes that generate short-lived or difficult-to-access instances.

A custom user model is a good choice to add *Loggable* to (or, even better, *OLPMixin*). In the typical scenario, a user instance is accessible on every incoming request, can easily (and is often required to) be passed around among various function/method calls, and persists for the entire request-response cycle.

If there is a need for a more persistent storage mechanism for these logs, there are a number of ways that can be achieved. One possible solution is to override a suitable method from *Loggable*. The following example uses a simple model to store logs made on a custom `User` model once they have been completed:

```
from django.contrib.auth.models import AbstractUser
from django.db import models

from djem.models import Loggable


class UserLog(models.Model):

    name = models.CharField(max_length=100)
    log = models.TextField()


class User(Loggable, AbstractUser):

    def end_log(self):

        name, log = super(User, self).end_log()

        UserLog.objects.create(
            name=name,
            log='\n'.join(log)
        )

        return name, log
```

3.2.5 Thread safety

Despite being able to be *safely nested*, instance-based logging is not thread safe. Due to the way the "active" log is maintained, multiple threads sharing the same *Loggable* instance can easily contaminate each other's logs. In addition, if they run the same process (that uses instance-based logging), they may error out due to being unable to have *multiple unfinished logs with the same name*.

Multiple threads using *separate* instances of *Loggable* will only be problematic if implementing some kind of *persistent storage* mechanism for the logs, depending on the nature of the chosen mechanism.

When using *Loggable* (or *OLPMixin*) on a custom user model, as part of a standard request-response cycle, thread safety is not a concern as each request uses a separate user instance.

3.3 Models

Djem provides a series of custom classes to support common model-related functionality, including models, model managers and model fields.

3.3.1 Auditable

The *Auditable* class is designed as a mixin for Django models, providing:

- Standard user and datetime fields: `user_created`, `user_modified`, `date_created`, `date_modified`.
- Support for *ensuring these fields remain accurate* as records are updated over time.
- Support for *ownership checking*.
- A custom manager/queryset to assist with maintaining accuracy and checking ownership.

Changed in version 0.7: The *Auditable* mixin was previously called `CommonInfoMixin`. The old name is still available for backwards compatibility, but is considered deprecated.

Warning: Using *Auditable* can break code that automatically calls methods such as the model's *save()* method, or the queryset's *update()* method. See *Maintaining accuracy* for a description of the caveats of `Auditable`, and workarounds.

Usage

To make use of *Auditable*, simply include it among your model's parent classes. It should be listed ahead of `models.Model`:

```
from django.db import models
from djem.models import Auditable

class ExampleModel(Auditable, models.Model):

    name = models.CharField(max_length=64)
```

Default values

The `date_created` and `date_modified` fields will default to `django.utils.timezone.now()` at the moment the instance is initially saved.

The `user_created` and `user_modified` fields will require a `User` instance in order to populate their values. However, they do not need to be populated manually. Djem provides various mechanisms to both make it easy to populate these fields automatically, and to ensure they are populated any time a record is updated. See *Maintaining accuracy*.

If any of the fields *are* populated manually, those values will take precedence.

Maintaining accuracy

The fields provided by *Auditable* are designed to be automatically populated whenever necessary. And in the case of `date_modified` and `user_modified`, it is necessary to update them whenever a record is updated.

For the date fields, this is easy to accomplish. For the user fields, it requires something extra - knowledge of the user doing the creating/updating.

Various means exist to provide this:

Calling `save()` on the instance

The *Auditable.save()* method is overridden to require a `User` instance as the first argument. This allows the method to populate `user_created` when a new instance is being created, and keep `user_modified` up to date as changes are made.

```
>>> alice = User.objects.get(username='alice')
>>> bob = User.objects.get(username='bob')
>>> obj = ExampleModel(name='Awesome Example')
>>> obj.user_created
None
>>> obj.save(alice)
>>> obj.user_created.username
"alice"
>>> obj.user_modified.username
"alice"
>>> obj.save(bob)
>>> obj.user_created.username
"alice"
>>> obj.user_modified.username
"bob"
```

Note: These fields will be updated even if the *save()* method is passed a sequence of `update_fields` that does not include it (see Django documentation for update_fields). They will simply be appended to the list.

Using the queryset

Like *Auditable.save()*, various methods on *AuditableQuerySet* are also overridden to require an additional argument providing a user model instance. Again, this allows the methods to set or update the user-based fields as necessary. These methods include *create()*, *get_or_create()*, *update()*, and *update_or_create()*.

The following demonstrates the use of the *create()* and *update()* methods:

```
>>> alice = User.objects.get(username='alice')
>>> bob = User.objects.get(username='bob')
>>> obj = ExampleModel.objects.create(alice, name='Good Example')
>>> obj.user_created.username
'alice'
>>> obj.user_modified.username
'alice'
>>> ExampleModel.objects.filter(name='Good Example').update(bob, name='Great Example')
```

(continues on next page)

(continued from previous page)

```
>>> obj.refresh_from_db()
>>> obj.user_created.username
'alice'
>>> obj.user_modified.username
'bob'
```

New in version 0.7: The *create()*, *get_or_create()*, and *update_or_create()* queryset methods.

Using forms

The `ModelForm` is core to any Django web application. For compatibility with *Auditable* (i.e. ensuring a `user` argument is passed to the *Auditable.save()* method), Djem provides *AuditableForm* and *UserSavable*.

AuditableForm is a simple wrapper around `ModelForm`, and is designed to be used as a replacement to it for forms based on `Auditable` models. It takes a `User` instance as a constructor argument, giving it a known user to pass to the model's `save()` method when the form is saved.

```
# forms.py
from djem.forms import AuditableForm

class ExampleForm(AuditableForm):

    class Meta:
        model = ExampleModel
        fields = ['name']

# views.py
def create_example(request):
    #...
    form = ExampleForm(request.POST, user=request.user)
    if form.is_valid():
        form.save()
    #...
```

Alternatively, *UserSavable* is a mixin for `ModelForm`, and is designed to be used for forms that already accept and store a known user. Unlike *AuditableForm*, it does not modify the constructor, but does assume a `self.user` attribute is available to pass to the model's `save()` method when the form is saved.

```
# forms.py
from django import forms
from djem.forms import UserSavable

class ExampleForm(UserSavable, forms.ModelForm):

    def __init__(self, *args, user=None, **kwargs):

        self.user = user

        super().__init__(*args, **kwargs)

    class Meta:
        model = ExampleModel
```

(continues on next page)

(continued from previous page)

```
        fields = ['name']

# views.py
def create_example(request):
    #...
    form = ExampleForm(request.POST, user=request.user)
    if form.is_valid():
        form.save()
    #...
```

Changed in version 0.7: *AuditableForm* was previously called `CommonInfoForm`. The old name is still available for backwards compatibility, but is considered deprecated.

New in version 0.7: *UserSavable*

Caveats and workarounds

Obviously any code that calls a model's `save()` method or various queryset methods (such as `create()` or `update()`) will need to be altered to pass a user argument for models that incorporate *Auditable*. This may not always be possible for third party code. *AuditableForm*/*UserSavable* solve this problem for one common occurrence, by providing wrappers around Django's `ModelForm.save()` method, but there are plenty of others.

If it is not feasible to customise code that calls these methods, it *is* possible to disable the requirement of the `user` argument. This can be done by setting *DJEM_AUDITABLE_REQUIRE_USER_ON_SAVE* to `False` in `settings.py`:

```
DJEM_AUDITABLE_REQUIRE_USER_ON_SAVE = False
```

This allows the use of `Auditable` and all related functionality without the strict requirement of passing the `user` argument to methods that save/update the record. If passed, it will still be used as described above, but not providing it will not raise an exception. Of course, the methods won't automatically populate the appropriate fields, either. This means that `user_created` and `user_modified` will need to be manually populated when creating, and `user_modified` will need to be manually populated when updating.

Changed in version 0.7: *DJEM_AUDITABLE_REQUIRE_USER_ON_SAVE* was previously called `DJEM_COMMON_INFO_REQUIRE_USER_ON_SAVE`. The old setting is still available for backwards compatibility, but is considered deprecated.

Warning: Setting *DJEM_AUDITABLE_REQUIRE_USER_ON_SAVE* to `False` reduces the accuracy of the `user_modified` field, as it cannot be guaranteed that the user that made a change was recorded.

Note: As the accuracy of the `user_modified` field is often irrelevant in tests, setting *DJEM_AUDITABLE_REQUIRE_USER_ON_SAVE* to `False` using override_settings() can help make updating model instances in tests a bit easier.

E.g.

```
from django.test import TestCase, override_settings

# For the whole TestCase:

@override_settings(DJEM_AUDITABLE_REQUIRE_USER_ON_SAVE=False)
```

(continues on next page)

(continued from previous page)

```
class ExampleTestCase(TestCase):
    # ...

# For specific tests:

class ExampleTestCase(TestCase):

    @override_settings(DJEM_AUDITABLE_REQUIRE_USER_ON_SAVE=False)
    def test_something(self):
        # ...
```

An additional caveat is that there may not always be a known user when a change is being made to a `Auditable` record, e.g. during a system-triggered background process. Situations such as these may be solved by setting *DJEM_AUDITABLE_REQUIRE_USER_ON_SAVE* as described above, and taking responsibility for keeping `user_modified` up to date when necessary, or by creating a "system" user that can be passed in during these operations.

Ownership checking

Auditable also adds support for *ownership checking*. The *owned_by()* method can be called on an model instance to check if the instance is owned by the given user. The user can be provided either as a `User` instance or as the primary key of a `User` record.

```
>>> alice = User.objects.get(username='alice')
>>> bob = User.objects.get(username='bob')
>>> obj = ExampleModel(name='Awesome Example')
>>> obj.save(alice)
>>> obj.owned_by(alice)
True
>>> obj.owned_by(bob)
False
```

Ownership checking is also available via a `Auditable` model's manager and queryset. The queryset's *owned_by()* method also accepts a user as a `User` instance or as the primary key of a `User` record. It returns a queryset filtered to records where the `user_created` field matches the given user.

```
>>> ExampleModel.objects.owned_by(alice)
[<ExampleModel: Awesome Example>]
>>> ExampleModel.objects.owned_by(bob)
[]
>>> ExampleModel.objects.filter(name__contains='Great').owned_by(alice)
[]
```

3.3.2 Archivable

The *Archivable* class is designed as a mixin for Django models, providing:

- An `is_archived` Boolean field, defaulting to `False`.
- A custom manager/queryset with shortcuts for filtering on the `is_archived` field.
- Support for easily *archiving and unarchiving* an instance.

Changed in version 0.7: The *Archivable* mixin was previously called `ArchivableMixin`. The old name is still available for backwards compatibility, but is considered deprecated.

Usage

To make use of *Archivable*, simply include it among your model's parent classes. It should be listed ahead of `models.Model`:

```
from django.db import models
from djem.models import Archivable

class ExampleModel(Archivable, models.Model):

    name = models.CharField(max_length=64)
```

Archiving and unarchiving

Instances of *Archivable* have the *archive()* and *unarchive()* methods. These set the `is_archived` flag of the instance to `True` or `False`, respectively, and save the instance. Any arguments provided to them are passed through to their internal calls to `save()`.

```
>>> obj = ExampleModel(name='Awesome Example')
>>> obj.save()
>>> ExampleModel.objects.get(name='Awesome Example').is_archived
False
>>> obj.archive()
>>> ExampleModel.objects.get(name='Awesome Example').is_archived
True
>>> obj.unarchive()
>>> ExampleModel.objects.get(name='Awesome Example').is_archived
False
```

Changed in version 0.7: Previous versions of Djem also provided `archive()` and `unarchive()` methods on *ArchivableQuerySet*. These were removed due to the overhead they added to combining the functionality of *ArchivableQuerySet* and *AuditableQuerySet*, and because the naming was too similar after the introduction of the *archived()* and *unarchived()* methods. Using the `update()` method and passing `is_archived` is more explicit and safer.

Filtering shortcuts

Archivable uses the custom *ArchivableQuerySet*, which provides the *archived()* and *unarchived()* methods. These methods filter the queryset to records with `is_archived=True` or `is_archived=False`, respectively. They are accessible both at the manager and queryset level.

```
>>> ExampleModel(name='Example1', is_archived=True).save()
>>> ExampleModel(name='Example2', is_archived=False).save()
>>> ExampleModel.objects.count()
2
>>> ExampleModel.objects.unarchived().count()
1
>>> ExampleModel.objects.filter(name='Example1').unarchived().count()
0
>>> ExampleModel.objects.archived().count()
1
>>> ExampleModel.objects.filter(name='Example2').archived().count()
0
```

Changed in version 0.7: Previous versions of Djem used three different managers - `objects`, `live` and `archived` - to provide access to querysets with various preset filters on the `is_archived` field. This didn't allow for applying the filters if a queryset was obtained by other means (e.g. via a `ManyToManyField` related manager) and made it more difficult to extend the manager (if the model inheriting from `Archivable` needed its own custom manager/queryset).

3.3.3 Versionable

The *Versionable* class is designed as a mixin for Django models, providing a `version` field that is automatically incremented on every save.

Changed in version 0.7: The *Versionable* mixin was previously called `VersioningMixin`. The old name is still available for backwards compatibility, but is considered deprecated.

Usage

To make use of *Versionable*, simply include it among your model's parent classes. It should be listed ahead of `models.Model`:

```
from django.db import models
from djem.models import Versionable

class ExampleModel(Versionable, models.Model):

    name = models.CharField(max_length=64)
```

Incrementing `version`

Incrementation of the `version` field is done atomically, through the use of a Django `F()` expression, to avoid possible race conditions. See Django documentation for F() expressions.

To ensure the `version` field is always kept current, *`Versionable`* overrides the *`save()`* method and the *`update()`* method of its custom queryset. The `update_or_create()` method, while not overridden, will also increment the version appropriately.

Note: The `version` field will be updated even if the `save()` method is passed a sequence of `update_fields` that does not include it (see Django documentation for update_fields). It will simply be appended to the list.

Warning: Once an instance is saved and the `F()` expression is used to increment the version, the `version` field will become a Django `Expression` instance. At this point, it is no longer accessible as an integer. For the same reason an `F()` expression is used to perform the incrementation (race conditions), the new version cannot be retrieved from the database after the save and used to replace the `Expression` value. There is the possibility the version retrieved will not be the one that matches the rest of the values on the model. The only way to regain a usable `version` field after saving a model instance is requerying for the whole instance. Attempting to access the `version` field after it has been incremented will raise a *`Versionable.AmbiguousVersionError`* exception.

Note: Even though directly accessing the `version` field is not possible after it has been atomically incremented, subsequent saves of the same instance will continue to correctly increment it.

3.3.4 Mixing Mixins

A model can include any combination of the above mixins. However, since they all use custom managers/querysets to provide additional functionality unique to them, a model using multiple mixins will also need to specify its own manager/queryset that incorporates the functionality of each. The custom querysets have been designed to make this as simple as possible, without any additional customisation necessary. Each mixin supports a custom *`as_manager()`* method that makes it even easier, as demonstrated below. This functionality is provided by the *`MixableQuerySet`* mixin, which is also available for use in your own custom queryset classes.

For a ready-made combination of all three mixins (*Auditable*, *Archivable* and *Versionable*), see *StaticAbstract*.

The following is an example of a model using both the *Auditable* and *Archivable* mixins:

```
from django.db import models
from djem.models import Archivable, Auditable
from djem.managers import ArchivableQuerySet, AuditableQuerySet

class ExampleModel(Auditable, Archivable, models.Model):

    name = models.CharField(max_length=64)

    objects = AuditableQuerySet.as_manager(ArchivableQuerySet)
```

New in version 0.7: *`MixableQuerySet`* and its custom *`as_manager()`* method. Previously, mixin querysets had to be combined manually.

3.3.5 StaticAbstract

StaticAbstract is a combination of *Auditable*, *Archivable* and *Versionable*. It is designed as an abstract base class for models, rather than a mixin itself. It includes all the fields and functionality offered by each of the mixins, including:

- *Maintaining the accuracy* of `date_modified` and `user_modified` as changes are made.
- Automatically and *atomically incrementing* `version` as changes are made.
- Allowing *archiving and unarchiving*.
- Providing *ownership checking*.

Usage

To make use of *StaticAbstract*, simply inherit from it:

```
from django.db import models
from djem.models import StaticAbstract

class ExampleModel(StaticAbstract):

    name = models.CharField(max_length=64)
```

3.3.6 TimeZoneField

TimeZoneField is a model field that stores timezone name strings ('Australia/Sydney', 'US/Eastern', etc) in the database and provides access to *TimeZoneHelper* instances for the stored timezones, as *explained below*.

In forms, a *TimeZoneField* is represented by a `TypedChoiceField`, and rendered using a `Select` widget by default.

Note: Use of *TimeZoneField* requires pytz to be installed. It will raise an exception during instantiation if `pytz` is not available.

Note: Use of *TimeZoneField* only makes sense if USE_TZ is True.

Usage

TimeZoneField is used just like any model field. The following demonstrates adding a `time_zone` field to a custom `User` model.

```
from django.contrib.auth.models import AbstractBaseUser
from djem.models import TimeZoneField

class User(AbstractBaseUser):
    ...
    time_zone = TimeZoneField()
```

Accessing the `time_zone` field on a `User` instance yields a *TimeZoneHelper* instance, which provides some helpers for dealing with times in local timezones, as *explained below*.

```
>>> user = User.objects.get(timezone='Australia/Sydney')
>>> user.timezone
<TimeZoneHelper: Australia/Sydney>
```

Available Timezones

TimeZoneField is a reasonably light wrapper around a `CharField`, providing a default value for the `choices` argument. The default choices are taken from pytz.common_timezones.

These choices can be modified in the same way as any other `CharField`. However, they need to be valid timezone name strings as per the Olson tz database, used by pytz.

For example, using a very limited set of timezones:

```
from django.contrib.auth.models import AbstractBaseUser
from djem.models import TimeZoneField

class User(AbstractBaseUser):
    ...
    time_zone = TimeZoneField(choices=(
        ('Australia/Brisbane'),
        ('Australia/Sydney'),
        ('Australia/Melbourne')
    ))
```

TimeZoneHelper

TimeZoneHelper is a simple helper class that provides shortcuts for getting the current date and the current datetime for a known local timezone.

Assuming a `User` model with a `time_zone` field, as shown above:

```
>>> aus_user = User.objects.get(timezone='Australia/Sydney')
>>> aus_user.timezone.name
'Australia/Sydney'
>>> aus_user.timezone.now()
datetime.datetime(2016, 6, 21, 9, 47, 4, 29965, tzinfo=<DstTzInfo 'Australia/Sydney'␣
↪AEST+10:00:00 STD>)
>>> aus_user.timezone.today()
datetime.date(2016, 6, 21)

>>> us_user = User.objects.get(timezone='US/Eastern')
>>> us_user.timezone.name
'US/Eastern'
>>> us_user.timezone.now()
datetime.datetime(2016, 6, 20, 19, 47, 4, 32814, tzinfo=<DstTzInfo 'US/Eastern' EDT-1␣
↪day, 20:00:00 DST>)
>>> us_user.timezone.today()
datetime.date(2016, 6, 20)
```

Warning: Be careful when dealing with local times. Django recommends you "use UTC in the code and use local time only when interacting with end users", with the conversion from UTC to local time usually only being performed in templates. And the pytz documentation notes "The preferred way of dealing with times is to always work in UTC, converting to localtime only when generating output to be read by humans". See the Django timezone documentation and the pytz documentation.

3.4 Rendering Forms

New in version 0.6.

Djem provides some simple template tags to assist in rendering form fields in templates.

While Django provides numerous ways to customise the way it renders form fields, they are often inadequate. The options for rendering all of a form's fields as a series of <tr>, <li>, or <p> tags, are limited to providing fairly basic field/label pairs, with little opportunity for customisation. And rendering fields manually often results in repeating boilerplate code, e.g. specifying a wrapping element, the field, its label, its error list, etc.

The *form_field* and *checkbox* template tags use a fully customisable template to render each individual field. The template, found at djem/form_field.html, can be altered per-project using Django's standard template overriding mechanism. By default, it provides:

- A wrapping element with the following features:
  - By default, the element is a , but this can be modified using the *DJEM_FORM_FIELD_TAG* setting.
  - The element is given the following CSS classes:
    - form-field
    - Any classes defined by the form. See below examples.
    - Any classes passed to the template tag. See below examples.
  - The element is given additional HTML attributes based on keyword arguments passed to the template tag. See below examples.
- The field's label (for fields rendered with form_field, see below examples).
- The field's error list (when applicable). The error list is the Django default <ul class="errorlist">.
- The field itself.
- The specified checkbox label (for fields rendered with checkbox, see below examples). This <label> element will have the check-label CSS class.
- A with the CSS class form-field__help containing the field's help text, if any.

The following usage examples are based off a basic user registration form:

```
from django import forms

class RegistrationForm(forms.Form):
    first_name = forms.CharField(label='Your first name')
    last_name = forms.CharField(label='Your last name')
    email = forms.EmailField(label='Your email address', help_text="Don't worry, we won
↪'t share your email address with anyone.")
    terms = forms.BooleanField(label='Terms of Service')
```

In the below template snippets, an instance of this form is contained in a context variable called registration_form.

3.4.1 form_field

The *form_field* tag has a single required argument: the form field to be rendered.

```
{% load djem %}
...
{% form_field registration_form.first_name %}
{% form_field registration_form.last_name %}
{% form_field registration_form.email %}
{% form_field registration_form.terms %}
...
```

Rendering this template would result in the following HTML:

```
<div class="form-field">
    <label for="id_first_name">Your first name:</label>
    <input type="text" name="first_name" id="id_first_name" />
</div>
<div class="form-field">
    <label for="id_last_name">Your last name:</label>
    <input type="text" name="last_name" id="id_last_name" />
</div>
<div class="form-field">
    <label for="id_email">Your email address:</label>
    <input type="email" name="email" id="id_email" />
    <div class="form-field__help">Don't worry, we won't share your email address with␣
↪anyone.</div>
</div>
<div class="form-field">
    <label for="id_terms">Terms of Service:</label>
    <input type="checkbox" name="terms" id="id_terms" />
</div>
```

This doesn't look too different from what `{{ registration_form.as_p }}` would generate, but note:

- The `<div>` wrapper around each field can be customised without even overriding the template, using the *DJEM_FORM_FIELD_TAG* setting.
- The `form-field` class provides an easy CSS styling target.
- Help text is rendered automatically when defined, with its own easily targetted `field-field__help` class for styling.
- The entire block rendered for each field can be completely customised by overriding the `djem/form_field.html` template, allowing for custom and consistent rendering of form fields across your project.

The *form_field* tag has more advanced uses, too. But first, the "terms" checkbox rendered above could be nicer, and that's what the *checkbox* tag is for.

3.4.2 checkbox

The *checkbox* tag uses the same `djem/form_field.html` template as *form_field*, but it renders labels differently:

- it includes the <label> element *after* the field itself, not before
- it gives the <label> element the `check-label` CSS class, allowing it to be styled independently of regular labels

Also, unlike *form_field*, *checkbox* is a *block tag*. It uses the content between its start and end tags as the label for the field. This has one important benefit: we can include HTML in the label text. Updating the previous example to use the *checkbox* tag for the "terms" field:

```
{% load djem %}
...
{% checkbox registration_form.terms %}
    I agree to the <a href="{% url 'terms' %}" target="_blank">Terms of Service</a>.
{% endcheckbox %}
...
```

The rendered HTML for this field then becomes:

```
<div class="form-field">
    <input type="checkbox" name="terms" id="id_terms" />
    <label class="check-label" for="id_terms">I agree to the <a href="{% url 'terms' %}"␣
↪target="_blank">Terms of Service</a>.</label>
</div>
```

That's a bit more user-friendly.

If no content is entered between the start and end tags, the field's default label text is used. In this case, the <label> element will still be included after the field itself, instead of before, and will still receive the `check-label` class.

Note: Don't go too crazy with HTML in your label text. It is still rendered inside a <label> element, so should only contain markup that is valid within <label>.

Note: *checkbox* is not strictly limited to actual checkbox inputs. You could, if for some reason it was appropriate, use it for any form field.

3.4.3 Extra CSS classes

Both *form_field* and *checkbox* can add additional field-specific CSS classes to the wrapping element. These classes can come from two places: the `Form` class definition, or passing them into the template tag itself.

A `Form` class can specify additional CSS classes per field in several ways. The `error_css_class` and `required_css_class` attributes can be used to automatically apply additional classes to fields that are required or contain errors, respectively. Also, custom classes can be applied to specific fields using their `css_classes()` method.

Adapting the registration form example to add a CSS class to highlight required fields is simple:

```
from django import forms

class RegistrationForm(forms.Form):
```

(continues on next page)

(continued from previous page)

```
    required_css_class = 'form-field--required'

    first_name = forms.CharField(label='Your first name')
    last_name = forms.CharField(label='Your last name')
    email = forms.EmailField(label='Your email address', help_text="Don't worry, we won
↪'t share your email address with anyone.")
    terms = forms.BooleanField(label='Terms of Service')
```

Without changing the template, this generates the following HTML:

```
<div class="form-field  form-field--required">
    <label for="id_first_name">Your first name:</label>
    <input type="text" name="first_name" id="id_first_name" />
</div>
<div class="form-field  form-field--required">
    <label for="id_last_name">Your last name:</label>
    <input type="text" name="last_name" id="id_last_name" />
</div>
<div class="form-field  form-field--required">
    <label for="id_email">Your email address:</label>
    <input type="email" name="email" id="id_email" />
    <div class="form-field__help">Don't worry, we won't share your email address with␣
↪anyone.</div>
</div>
<div class="form-field  form-field--required">
    <label for="id_terms">Terms of Service:</label>
    <input type="checkbox" name="terms" id="id_terms" />
</div>
```

The other way to apply custom CSS classes to a field is to pass them directly into the template tag. For example, in order to display the `first_name` and `last_name` fields side-by-side, simply pass in a CSS class that can style them accordingly:

```
{% load djem %}
...
{% form_field registration_form.first_name 'one-half' %}
{% form_field registration_form.last_name 'one-half' %}
{% form_field registration_form.email %}
{% form_field registration_form.terms %}
...
```

```
<div class="form-field  form-field--required  one-half">
    <label for="id_first_name">Your first name:</label>
    <input type="text" name="first_name" id="id_first_name" />
</div>
<div class="form-field  form-field--required  one-half">
    <label for="id_last_name">Your last name:</label>
    <input type="text" name="last_name" id="id_last_name" />
</div>
<div class="form-field  form-field--required">
    <label for="id_email">Your email address:</label>
```

(continues on next page)

(continued from previous page)

```
    <input type="email" name="email" id="id_email" />
    <div class="form-field__help">Don't worry, we won't share your email address with␣
↪anyone.</div>
</div>
<div class="form-field  form-field--required">
    <label for="id_terms">Terms of Service:</label>
    <input type="checkbox" name="terms" id="id_terms" />
</div>
```

3.4.4 Extra HTML attributes

form_field and *checkbox* also support adding custom HTML attributes to the wrapping element. The attribute name and value can be provided to the template tag as keyword arguments. Since attribute names can contain dashes, which are invalid in Python keyword argument names, any underscores in the argument name will be converted into dashes to form the HTML attribute name.

This feature is probably most useful for attaching `data-*` attributes. For example, if interacting with a particular field should trigger an AJAX lookup of some description, the URL to use for that request could be stored in a `data-url` attribute:

```
{% load djem %}
...
{% url 'verify-email' as verify_email_url %}
{% form_field registration_form.email data_url=verify_email_url %}
...
```

```
<div class="form-field  form-field--required" data-url="/accounts/email/verify/">
    <label for="id_email">Your email address:</label>
    <input type="email" name="email" id="id_email" />
    <div class="form-field__help">Don't worry, we won't share your email address with␣
↪anyone.</div>
</div>
```

Something like the above could be used, for example, to keep hardcoded values (such as URLs) out of external JavaScript files.

3.4.5 Rendering in bulk

If you don't need per-field customisation, such as additional CSS classes, extra HTML attributes, or custom checkbox labels, you can use *form_field* in a loop:

```
{% load djem %}
...
{% for field in form.hidden_fields %}
    {{ field }}
{% endfor %}

{% for field in form.visible_fields %}
    {% form_field field %}
{% endfor %}
...
```

Note: Note that, while *form_field* will work for hidden fields, it is largely useless. Most of the features it provides are unnecessary for hidden fields.

3.5 Pagination

Djem provides some simple helpers to remove common boilerplate in fetching and rendering pages of results among some list of items.

3.5.1 In the view

Django's `Page` object provides a number of helpful methods and attributes that aid in rendering a paged result list, in addition to providing a slice of the result list itself. However, obtaining a `Page` instance can involve a bit of boilerplate, particularly in validating the page number and handling various possible invalid states. Moreover, it needs to be repeated in every view that uses pagination.

Djem's *get_page()* removes this boilerplate. It is a basic wrapper around a Django `Paginator` object that primarily provides some sanity checking of the given page number.

Internally, it constructs a `Paginator` instance and uses its `page()` method to retrieve a `Page`.

See the Django pagination documentation for details on the `Paginator` and `Page` objects provided by Django.

New in version 0.5.

Usage

get_page() has three required arguments:

number
: The page number to retrieve, as a 1-based index. The given value is validated in the following ways:

  - If it is not an integer, `Paginator` raises a `PageNotAnInteger` exception. `get_page()` catches this and returns the first page of results instead.
  - If it is less than 1, `Paginator` raises an `EmptyPage` exception. `get_page()` catches this and returns the first page of results instead.
  - If it is over the maximum number of pages for the given item sequence, `Paginator` also raises an `EmptyPage` exception. `get_page()` catches this and returns the last page of results instead.

object_list
: The sequence of items from which to retrieve the specified page. As per the `Paginator` object itself, the sequence can be a list, tuple, Django `QuerySet`, or any other sliceable object with a `count()` or `__len__()` method.

per_page
: The number of results to be included in each page. While required by default, a project-wide default value can be set that removes the need to specify `per_page` on every call. See *Controlling page length* below.

While `get_page()` only returns a `Page` instance, the `Paginator` that created it is accessible through the `Page.paginator` attribute. Adapting Django's example:

```
>>> from djem.pagination import get_page
>>> objects = ['john', 'paul', 'george', 'ringo']
>>> page1 = get_page(1, objects, per_page=2)

>>> page1.paginator.count
4
>>> page1.paginator.num_pages
2
>>> page1.paginator.page_range
range(1, 3)

>>> page1
<Page 1 of 2>
>>> page1.object_list
['john', 'paul']
>>> page1.has_next()
True
>>> page1.has_previous()
False

>>> get_page('spam', objects, per_page=2)
<Page 1 of 2>

>>> get_page(-1, objects, per_page=2)
<Page 1 of 2>

>>> get_page(9999, objects, per_page=2)
<Page 2 of 2>
```

`get_page()` also accepts all remaining keyword arguments of the `Paginator` constructor, which are passed through to the `Paginator` instance created internally. For example, using the `orphans` argument with the above example:

```
>>> get_page(1, objects, per_page=3)
<Page 1 of 2>

>>> get_page(1, objects, per_page=3, orphans=1)
<Page 1 of 1>
```

`get_page()` can still raise an `EmptyPage` exception if `allow_empty_first_page=False` is given and the `object_list` is empty:

```
>>> from djem.pagination import get_page
>>> objects = []
>>> page1 = get_page(1, objects, per_page=20, allow_empty_first_page=False)
Traceback (most recent call last):
...
EmptyPage: That page contains no results
```

Note: *`get_page()`* differs from the `Paginator.get_page()` method introduced in Django 2.0 in two ways:

- The default page length support *described below*.
- The behaviour of an out-of-range page number: For numbers less than 1, `Paginator.get_page()` returns the *last page* while `get_page()` returns the *first* page.

Controlling page length

In a lot of cases, sites use a standard page length for multiple (sometimes numerous) paginated lists they display. Djem reduces the need to specify this value in repeated calls to *get_page()* by providing a setting that defines this standard page length: *DJEM_DEFAULT_PAGE_LENGTH*.

If added to your `settings.py` file, `get_page()` will use the defined page length in any call that does not explicitly pass the `per_page` argument. Providing the argument will override the default on a call-by-call basis.

Not providing the `per_page` argument when `DJEM_DEFAULT_PAGE_LENGTH` is **not** defined in `settings.py` will result in an exception.

3.5.2 In the template

Rendering a block of links to control pagination of a result list (first, last, next, previous, etc) can also involve boilerplate code and repetition. The *paginate* template tag allows one-line generation of such pagination links. Simply pass it the same Django `Page` instance used to render the list itself and it will render appropriate page navigation links. This allows quick and easy rendering of a consistent paged-list navigation block site wide.

New in version 0.6.

For example, where `user_list` is a `Page` instance:

```
{% load djem %}
...
{% for user in user_list %}
    {{ user.name }}
{% endfor %}
{% paginate user_list %}
...
```

The structure of the navigation block that is rendered is controlled by the `djem/pagination.html` template. Djem ships with a default template, but (as per any template provided by a Django app) this can be overridden by a specific project. See the Django documentation for overriding templates.

By default, the block will be rendered as a HTML ordered list (<ol>) with the following items:

- A link to the first page, labelled "First" (hidden if displaying the first page)
- A link to the previous page, labelled "Previous" (hidden if displaying the first page)
- "Page X of Y", where X is the current page number and Y is the total number of pages
- A link to the next page, labelled "Next" (hidden if displaying the last page)
- A link to the last page, labelled "Last" (hidden if displaying the last page)

The links are defined simply as "?page=n", where n is the relevant page number.

To alter the labels, change the format of the links (e.g. the name of the `GET` param), or completely change which links are displayed (e.g. adding links to individual page numbers), the `djem/pagination.html` will need to be overridden. However, this is not necessary to simply style the default navigation block. The following CSS classes are available by default:

- The <ol> element has the `pagination` class
- The <li> element containing the first page link has the `pagination__first` class

- The <li> element containing the previous page link has the `pagination__previous` class
- The <li> element containing the next page link has the `pagination__next` class
- The <li> element containing the last page link has the `pagination__last` class

The following CSS renders the <ol> and its items horizontally, removes their indentation and bullets, and adds double-angle quotation marks as glyphs to the "first" and "last" links (« and »):

```
.pagination {
    list-style: none;
    margin-left: 0;
    text-align: center;
}

.pagination > li {
    display: inline-block;
    margin-left: 0;
}

.pagination__first > a:before{
    content:"\00AB" "\00A0";
}

.pagination__last > a:after{
    content:"\00A0" "\00BB";
}
```

3.6 AJAX

3.6.1 Checking authentication

Django's built in `login_required()` decorator is not well suited to AJAX requests. If the user making the request is not authenticated, it issues a redirect to the configured login view. This is not particularly useful to an AJAX client.

Djem provides a simple alternate decorator, *ajax_login_required()*, to use on views that are the target of AJAX requests. Instead of issuing a redirect if the user is not authenticated, it returns a `HttpForbiddenResponse`.

New in version 0.7: The *ajax_login_required()* decorator.

3.6.2 Responding to AJAX requests

Django provides `JsonResponse` to aid in responding to data-centric AJAX requests (as opposed to those that return rendered HTML to be injected directly into the page). Djem provides a simple extension of `JsonResponse` that adds some additional features: *AjaxResponse*.

AjaxResponse automatically adds any messages in the Django messages framework store to the response body.

This allows views that are called via AJAX and return JSON-encoded data to still make use of the messages framework and have those messages automatically embedded in the response and removed from the message store.

The messages are added to the parent `JsonResponse`'s `data` dictionary using the "messages" key. The messages themselves are dictionaries containing the following:

- message: The message string.

- tags: A string of the tags applied to the message, space separated.

If there are no messages to add, the "messages" key will not be added to the `data` dictionary at all (i.e. it will not be added as an empty list).

Using *AjaxResponse* differs from `JsonResponse` in the following ways:

- The first positional argument should be a Django `HttpRequest` instance. This argument is required - it is used to retrieve messages from the message framework store.
- The `data` argument is optional. If provided, it must always be a `dict` instance, and messages will be added to this dictionary. If not provided, messages will be added to a new `dict` instance, passing it to the parent `JsonResponse`. Using the `safe` argument of `JsonResponse` to JSON-encode other types is not supported (see the documentation for the `safe` argument of `JsonResponse`).
- The optional argument `success` can be set to add a "success" attribute to the `data` dictionary. The "success" attribute will always be added as a boolean value, regardless of what was passed to the `success` argument (though it will not be added at all if nothing was passed). As with messages, this will be added to the `data` dictionary passed to the parent `JsonResponse` regardless of whether one was provided to *AjaxResponse* itself or not.

With the exception of `safe`, as noted above, *AjaxResponse* accepts and supports all arguments of `JsonResponse`.

Usage

This simple example demonstrates how *AjaxResponse* can be used with the messages framework:

```
from django.contrib import messages
from djem import AjaxResponse

def my_view(request):

    # do something...

    messages.success(request, 'Did something!')

    return AjaxResponse(request)
```

This will give a JSON-encoded response body that looks like this:

```
{
    "messages":[{
        "message":"Did something!",
        "tags":"success"
    }]
}
```

The following is a more complete example, based on the "polls" application created in the Django tutorial. This view records a vote on a given `Choice`:

```
from django.contrib import messages
from djem import AjaxResponse

from polls.models import Choice

def vote_on_question(request, choice_id):

```

(continues on next page)

(continued from previous page)

```
    try:
        choice = Choice.objects.get(pk=choice_id)
    except Choice.DoesNotExist:
        messages.error(request, 'Invalid choice.')
        return AjaxResponse(request, success=False)

    choice.votes += 1
    choice.save()

    messages.success(request, 'Vote recorded.')

    return AjaxResponse(request, {'votes': choice.votes}, success=True)
```

Note: This example is for illustrative purposes only. It does not represent a good way to increment a counter in the database - such an operation should be performed atomically.

3.6.3 CSRF

As Django notes in its own documentation, adding the CSRF token to AJAX POST requests can be done on each individual request, or you can use JavaScript framework features to add it to all outgoing POST requests, using the `X-CSRFToken` header.

Djem's *`csrfify_ajax`* template tag does exactly that in a single line:

```
{% load djem %}
...
{% csrfify_ajax %}
...
```

The tag injects a `<script>` tag containing library-specific code to add `X-CSRFToken` headers to all outgoing requests that require it.

New in version 0.6: The *`csrfify_ajax`* template tag.

Note: As with Django's standard `{% csrf_token %}` tag, to use `{% csrfify_ajax %}` the view must use `RequestContext` to render the template, e.g. using the `render()` shortcut function.

Note: As the `<script>` tag rendered by the tag contains library-specific code, it needs to be included *after* the library itself.

Library support

By default, Djem ships with support for jQuery, but it is simple to add additional libraries. Extra libraries may be included by default in future releases.

The <script> tag, and its contents, that *csrfify_ajax* renders is stored in a template under the `djem/csrfify_ajax/` directory, named after the library. E.g. the included jQuery template is at `djem/csrfify_ajax/jquery.html`. As with any Django app templates, you can override those that Djem includes or add your own by including your own `djem/csrfify_ajax/` directory somewhere on your configured template path. See the Django documentation for overriding templates.

By providing your own template, you can use *csrfify_ajax* with any library of your choosing. To specify which library template the tag should use, simply provide the name of the template (without the `.html`) as an argument:

```
{% load djem %}
...
{% csrfify_ajax 'some_other_lib' %}
...
```

A default value of `'jquery'` is used when no argument is provided.

These templates have access to the CSRF token via the `{{ csrf_token }}` template variable.

3.6.4 XSS

Django's template system automatically escapes all template variables by default. This helps prevent user-generated content from messing up the rendered HTML, or worse, exposing your users to a security vulnerability. This kind of injection is known as cross site scripting or XSS.

If including user-generated content in the JSON body of an AJAX response (that is, not returning rendered HTML), these autoescaping protections do not apply. If that content is then injected directly into the page when the response is received, similar XSS vulnerabilities exist.

You should be wary of XSS threats when returning data in this way. Either ensure all returned data has been properly escaped on the server side, or do not inject it as HTML (e.g. use jQuery's .text() method instead of .html())

Djem's *AjaxResponse* class helps mitigate one potential vector for XSS - messages added via the Django messages framework. All messages included in the response body are escaped, unless marked as safe.

```
# View
from django.contrib import messages
from django.template.defaultfilters import mark_safe
from djem.ajax import AjaxResponse

def view(request):

    some_user_input = '<strong>BAD</strong> user input'

    messages.error(request, 'Invalid input: {0}.'.format(some_user_input))
    messages.info(request, mark_safe('This is a message with <em>safe</em> HTML.'))

    return AjaxResponse(request)

# Response Body
{
    "messages": [{
```

(continues on next page)

(continued from previous page)

```
        "message": "Invalid input: &lt;strong&gt;BAD&lt;/strong&gt; user input.",
        "tags": "error"
    }, {
        "message": "This is a message with <em>safe</em> HTML.",
        "tags": "info"
    }]
}
```

3.6.5 The messages framework

As noted above, the *AjaxResponse* class can automatically include messages stored in Django's builtin messages framework. But whether or not you are using *AjaxResponse*, the messages framework is not perfectly suited for use in AJAX requests.

It is designed to make it simple to store one-time notification messages throughout the lifecycle of a request, then display them together at some later point, typically when rendering a template. Importantly, it offers persistent storage of these messages, allowing them to be retrieved and displayed on a *later* request. This is particularly useful when issuing browser redirects to display subsequent pages, and needing the messages from the current request to be available after the redirect.

As Django notes in its documentation, this persistence between requests can cause issues if multiple requests are issued in parallel. The wrong request can read from the persistent messages store, display the messages in the wrong context, and prevent the messages being shown where they were intended. When dealing with full requests for entire pages, this situation only occurs when a user is issuing requests from multiple browser tabs/windows at the same time, and thus is only a minor problem. But on a site using AJAX requests, the risk increases. Depending on the nature of the site and the AJAX requests it issues, the chances of multiple requests running simultaneously can increase dramatically, making the problem of requests stealing messages from one another much worse.

One solution is to avoid using the messages framework in views used by AJAX requests. After all, an AJAX request doesn't require the same redirect hopping that might otherwise be required - any messages that it generates can be returned immediately, making the persistent storage offered by the framework unnecessary. But some views can be used by both AJAX and non-AJAX requests, plus a consistent method of handling message passing regardless of the view would be nice.

Djem provides a *MessageMiddleware* class that acts as a drop-in replacement for Django's own. Simply replace the `django.contrib.messages.middleware.MessageMiddleware` string in your `MIDDLEWARE` setting with `djem.middleware.MessageMiddleware`.

```
# before
MIDDLEWARE = [
    ...
    'django.contrib.messages.middleware.MessageMiddleware'
    ...
]

# after
MIDDLEWARE = [
    ...
    'djem.middleware.MessageMiddleware'
    ...
]
```

Djem's *MessageMiddleware* is nearly identical to Django's, and usage of the messages framework itself is completely unchanged. The middleware has only one minor difference: it disables persistent message storage for AJAX requests.

On standard requests, the storage backend configured via the `MESSAGE_STORAGE` setting is still used as per usual. But on AJAX requests, a separate storage backend is used - one that keeps the messages in memory only, making them inaccessible to simultaneous requests.

New in version 0.6: *`MessageMiddleware`*

Important: *`MessageMiddleware`* uses the `HttpRequest` object's `is_ajax()` method to differentiate between AJAX and non-AJAX requests. Your `XMLHttpRequest` call must use the appropriate headers in order to be correctly detected. Most modern JavaScript libraries do so.

Note: Using a memory-only storage backend for messages in AJAX requests also makes them unavailable to subsequent requests. If using Djem's *`MessageMiddleware`*, be sure to read the messages from the storage as part of the same request and include them in the response. *`AjaxResponse`* does this automatically.

Note: Using Djem's *`MessageMiddleware`* doesn't change any of the other requirements for using the messages framework. For instance, `django.contrib.messages` still needs to be listed in the `INSTALLED_APPS` setting and, if using `SessionStorage`, `SessionMiddleware` still needs to be listed before `MessageMiddleware` in the `MIDDLEWARE` setting.

3.7 Testing

Djem includes some utilities that make writing certain types of tests easier. They are used internally within Djem's own test suite, but may well be useful elsewhere.

3.7.1 Test-only models

New in version 0.7.

It is often the case that tests are required for code that deals with models generically, and concrete example models to use in those tests may not exist. Django does not (yet) explicitly support defining models purely for testing. That is, models that do not get picked up by `manage.py makemigrations` or otherwise affect the live database, but only exist in the isolated test environment. But, thanks to the discussion on Django ticket #7835, the community has developed several workarounds.

Djem includes an adaption of one of the cleanest solutions (at time of writing, at least), provided by Simon Charette: *`setup_test_app()`*. It requires the use of a `tests` package, rather than just a `test.py` module, but otherwise all that is necessary is a simple function call.

Assuming a Django app with the name `myapp`, simply place the following in `myapp.tests.__init__`:

```
from djem.utils.tests import setup_test_app

setup_test_app(__package__)
```

This will configure Django with an extra app, named `myapp_tests`, and any models defined in `myapp.tests.models` will get picked up by the test database creation routine and added to the test database as part of this temporary app. These models can then be freely used in the test suite just like a regular model, but will not affect the live database.

If the default app label (created by appending `_tests` to the containing app's own app label) is not suitable, it can be provided explicitly:

```
from djem.utils.tests import setup_test_app

setup_test_app(__package__, 'mytests')
```

Warning: If the tests package, `myapp.tests` in the above examples, is imported by anything that forms part of the standard Django runtime environment (such as within a `models.py` module, or anything imported *by* a `models.py` module), the test-only models will **not** remain isolated to the test suite. It should not be necessary to import anything from within the test suite anywhere else in the project.

3.7.2 Enhanced request factory

New in version 0.6.

The *MessagingRequestFactory* is an extension of Django's `RequestFactory` helper for tests. It enables the use of the messages framework within the generated request.

This is typically not possible with `RequestFactory`, since it does not execute middleware. While the test client offers full middleware support, there are numerous situations in which `RequestFactory` is a preferable method of generating test requests.

MessagingRequestFactory does not add full middleware support, but does prepare the generated requests such that the messages framework can be used. It does not use the standard message storage backend (as per the `MESSAGE_STORAGE` setting), but rather a memory-only backend that does not involve the use of sessions, cookies or any other means of persistent storage of the messages.

This means that messages need to be read in the same request they were added, or they will be lost. A subsequent request will not be able to access them as they typically would, e.g. after a redirect. Despite this limitation, it is sufficient in many testing scenarios - whether or not a message was added as part of a request can be tested, even if that message is inaccessible to a later request:

```
from django.contrib.auth.models import User
from django.contrib import messages
from django.test import TestCase

from djem.utils.tests import MessagingRequestFactory

# Views expected to set messages
from .views import MyView, my_view


class SimpleTest(TestCase):

    def setUp(self):

        self.factory = MessagingRequestFactory()
        self.user = User.objects.create_user(
            username='test.user', email='test@...', password='top_secret'
        )

    def test_details(self):

        request = self.factory.get('/customer/details')
```

(continues on next page)

(continued from previous page)

```
        # Recall that middleware are not supported. You can simulate a
        # logged-in user by setting request.user manually.
        request.user = self.user

        # Test my_view() as if it were deployed at /customer/details
        response = my_view(request)

        # Use this syntax for class-based views
        response = MyView.as_view()(request)

        self.assertEqual(response.status_code, 200)

        # Test the expected message was set
        message_list = list(messages.get_messages(request))
        self.assertEqual(len(message_list), 1)
        self.assertEqual(message_list[0].message, 'An error occurred.')
```

3.7.3 Rendering string-based templates

New in version 0.6.

TemplateRendererMixin is a mixin for `TestCase` classes whose tests render templates from strings (as opposed to rendering them from files), using the Django template engine. This can be helpful, for example, when testing templatetags. Short template snippets can be rendered to test the tag under a variety of scenarios without requiring separate template files for each.

The mixin adds a *render_template()* method to the `TestCase`. This method takes the template to be rendered, as a string, and a template context dictionary as arguments. It returns the rendered template.

```
from django.test import TestCase

from djem.utils.tests import TemplateRendererMixin


class SomeTestCase(TemplateRendererMixin, TestCase):

    def test_something(self):

        template_string = (
            '{% if something %}'
            '    <p>'
            '        RENDER THIS'
            '    </p>'
            '    <p>'
            '        AND THIS'
            '    </p>'
            '{% endif %}'
        )

        output = self.render_template(template_string, {
            'something': True
```

(continues on next page)

(continued from previous page)

```
        })

        self.assertEqual(output, '<p> RENDER THIS </p><p> AND THIS </p>')
```

The output is stripped of all leading and trailing whitespace. Optionally, remaining whitespace will be "flattened" if the `flatten` argument is `True`, which is the default. Flattening removes all whitespace from between HTML tags and compresses all other whitespace down to a single space.

Flattening makes comparing rendered template output easier. This is demonstrated in the above example. The following example shows the difference when `flatten` is given as `False`:

```
def test_something(self):

    template_string = (
        '{% if something %}'
        '    <p>'
        '        RENDER THIS'
        '    </p>'
        '    <p>'
        '        AND THIS'
        '    </p>'
        '{% endif %}'
    )

    output = self.render_template(template_string, {
        'something': True
    }, flatten=False)

    self.assertEqual(output, '<p>        RENDER THIS    </p>    <p>        AND THIS    </
↪p>')
```

render_template() can optionally accept a `request` argument, which should be a `HttpRequest` instance if given. This enables it to be used to test templates that require being rendered with a `RequestContext`. For example, it could be combined with the Django request factory:

```
from django.http import HttpResponse
from django.test import RequestFactory, TestCase

from djem.utils.tests import TemplateRendererMixin


class SomeTestCase(TemplateRendererMixin, TestCase):

    def test_something(self):

        def view(r):

            template_string = '...'

            output = self.render_template(template_string, {}, r)

            return HttpResponse(output)
```

(continues on next page)

(continued from previous page)

```
        request = RequestFactory().get('/test/')
        response = view(request)

        self.assertContains(response, '...', status_code=200)
```

If the `TestCase` has a `user` attribute, e.g. defined in `setUp()` to be available to all tests, a "user" variable will be added to the template context. This is done automatically unless `request` is provided.

```
from django.contrib.auth.models import User
from django.test import TestCase

from djem.utils.tests import TemplateRendererMixin


class SomeTestCase(TemplateRendererMixin, TestCase):

    def setUp(self):

        self.user = self.user = User.objects.create_user(
            username='test.user', email='test@...', password='top_secret'
        )

    def test_something(self):

        template_string = '{{ user.username }}'

        output = self.render_template(template_string, {})

        self.assertEqual(output, 'test.user')
```

3.8 Utils

3.8.1 UNDEFINED

New in version 0.7.

The *UNDEFINED* constant is designed for use in argument default values.

Sometimes it is necessary to know whether *any* value was passed into a function or not, including values traditionally used as argument defaults (such as `None`, `False`, etc). Any given value could have an explicit meaning, with some default behaviour only performed if *nothing* was passed in.

Take the following contrived example:

```
def make_thing(name, label=None):

    if not label:
        label = name

    ...
```

The `make_thing` function takes a required `name` argument and an optional `label` argument. The label defaults to the name unless overridden with something more meaningful/verbose. But what if the caller would like their thing to have *no label*? And what if `None`, `False` or an empty string could all indicate that? The function needs to differentiate those values from *nothing* being provided.

This example could be rewritten using *UNDEFINED* in place of `None` as the argument default, allowing the caller to explicitly pass `None` to indicate that the label should not be given a default value:

```
from djem import UNDEFINED

def make_thing(name, label=UNDEFINED):

    if label is UNDEFINED:
        label = name

    ...
```

UNDEFINED is "falsey", so can also be used in more generic conditional statements:

```
from djem import UNDEFINED

value = UNDEFINED

if value:
    print('truthy')
else:
    print('falsey')

# output: 'falsey'
```

3.9 API Reference

3.9.1 Model Mixins and QuerySets

Mixins

Loggable

New in version 0.7.

class djem.models.**Loggable**(**args*, ***kwargs*)

A mixin for creating, storing, and retrieving logs on an instance. Named logs are stored internally on the `Loggable` instance and persist for the lifetime of the object. A single log is "active" at any given time and can be freely appended to while it is.

Adds *instance-based logging* support to any model.

start_log(*name*)

Start a new log with the given `name`. The new log becomes the current "active" log. Queue any previous active log so that it can be reactivated when the new log is either finished or discarded.

Parameters

name – The name of the log.

`end_log()`

End the currently active log and return a `(name, log)` tuple, where `name` is the name of the log that was ended and `log` is a list of the entries that have been added to the log. Reactivate the previous log, if any.

The returned list will be a *copy* of the one used to store the log internally, allowing it to be safely manipulated without affecting the original log.

A log must be ended in order to be retrieved.

Returns
A `(name, log)` tuple.

`discard_log()`

Discard the currently active log. Reactivate the previous log, if any.

`log(*lines, tag=None)`

Append to the currently active log. Each given argument will be added as a separate line to the log. If `tag` is specified, each added line will be tagged with the given value.

Parameters

- **`lines`** – Individual lines to add to the log.
- **`tag`** – A tag to apply to each line.

`get_log(name, tags=None, raw=False)`

Return the named log, as a string. The log must have been ended (via `end_log()`) in order to retrieve it.

Return a raw list of lines in the log if `raw=True`. In this case, the returned list will be a *copy* of the one used to store the log internally, allowing it to be safely manipulated without affecting the original log.

Whether returning a string or a list, use `tags` to filter the included lines to just those with at least one of the given tags.

Parameters

- **`name`** – The name of the log to retrieve.
- **`tags`** – An iterable of tags to filter the log by.
- **`raw`** – `True` to return the log as a list. Returned as a string by default.

Returns
The log, either as a string or a list.

`get_last_log(tags=None, raw=False)`

Return the most recently finished log, as a string.

Return a raw list of lines in the log if `raw=True`. In this case, the returned list will be a *copy* of the one used to store the log internally, allowing it to be safely manipulated without affecting the original log.

Whether returning a string or a list, use `tags` to filter the included lines to just those with at least one of the given tags.

Parameters

- **`tags`** – An iterable of tags to filter the log by.
- **`raw`** – `True` to return the log as a list. Returned as a string by default.

Returns
The log, either as a string or a list.

OLPMixin

New in version 0.7.

class djem.models.**OLPMixin**(**args*, ***kwargs*)

A companion to Django's `PermissionsMixin` that enables additional advanced features of the object-level permission system. It is not necessary to use this mixin in order to use object-level permissions, it just provides additional functionality (such as logging permission checks, optionally allowing superusers to be restricted by object-level conditions, etc).

Inherits instance-based logging functionality from *Loggable*. For more information on the available features, see *Advanced Features*.

has_perm(*perm*, *obj=None*)

A replacement for the default `has_perm()` method defined by Django's `PermissionsMixin`.

In conjunction with the *DJEM_UNIVERSAL_OLP* setting, this version can force superusers to be subject to the same object-level permissions checks as regular users.

In conjunction with the *DJEM_PERM_LOG_VERBOSITY*, an automatic log of all permission checks can be kept, using *instance-based logging*.

clear_perm_cache()

Clear the object-level and model-level permissions caches on the user instance.

Auditable

Changed in version 0.7: Renamed from `CommonInfoMixin`. The old name is still available for backwards compatibility, but is considered deprecated.

class djem.models.**Auditable**

`Auditable` is a model mixin class that provides:

- Standard user and datetime fields: `user_created`, `user_modified`, `date_created`, `date_modified`.
- An overridden `objects` Manager that provides access to the custom *AuditableQuerySet*.
- Support for *Ownership checking* on an instance and via *AuditableQuerySet*.

save(*user=None*, **args*, ***kwargs*)

Overridden to ensure the `user_modified` and `date_modified` fields are always updated. The `user` argument is required and must be passed a `User` instance, unless the *DJEM_AUDITABLE_REQUIRE_USER_ON_SAVE* setting is `False`.

owned_by(*user*)

Return `True` if `user_created` matches the given user, otherwise return `False`. The user can be given as an id or a `User` instance.

See also:

AuditableQuerySet
The custom QuerySet used by `Auditable`.

AuditableForm
A `ModelForm` subclass that supports `Auditable` models.

UserSavable
A `ModelForm` mixin to add support for `Auditable` models, for forms that already have a known user.

Archivable

Changed in version 0.7: Renamed from `ArchivableMixin`. The old name is still available for backwards compatibility, but is considered deprecated.

class djem.models.**Archivable**

`Archivable` is a model mixin class that provides:

- An `is_archived` Boolean field, defaulting to `False`.
- An overridden `objects` Manager that provides access to the custom *ArchivableQuerySet*.
- Methods for *archiving and unarchiving*

archive(**args*, ***kwargs*)

Archive this record.

Accepts all arguments of the `save` method, as it saves the instance after setting the `is_archived` flag. It saves using the `update_fields` keyword argument, containing the `is_archived` field, whether it was provided to this method or not. If provided, it is extended, not replaced.

unarchive(**args*, ***kwargs*)

Unarchive this record.

Accepts all arguments of the `save` method, as it saves the instance after setting the `is_archived` flag. It saves using the `update_fields` keyword argument, containing the `is_archived` field, whether it was provided to this method or not. If provided, it is extended, not replaced.

See also:

ArchivableQuerySet
The custom QuerySet used by `Archivable`.

Versionable

Changed in version 0.7: Renamed from `VersioningMixin`. The old name is still available for backwards compatibility, but is considered deprecated.

class djem.models.**Versionable**

`Versionable` is a model mixin class that provides:

- A `version` field that is automatically incremented on every save.
- An overridden `objects` Manager that provides access to the custom *VersionableQuerySet*.

save(**args*, ***kwargs*)

Overridden to ensure the `version` field is always updated.

exception AmbiguousVersionError

A subclass of *ModelAmbiguousVersionError* specific to the *Versionable* class. Raised when attempting to access the `version` field after it has been atomically incremented.

See also:

VersionableQuerySet
The custom QuerySet used by `Versionable`.

QuerySets

MixableQuerySet

New in version 0.7.

class djem.models.**MixableQuerySet**

A mixin for `QuerySet` classes that simply provides an enhanced *as_manager()* method that can be used to combine the queryset class with any number of other queryset classes automatically.

classmethod as_manager(**other_querysets*)

Similar to the `as_manager` classmethod available on regular Django queryset classes, this returns an instance of `Manager` with a copy of the queryset's methods. However, it also accepts *other* queryset classes as arguments and includes *their* methods in the created `Manager` also. This allows easily creating combinations of this queryset class with other custom queryset classes, without needing to manually create an extra class to do the grouping.

This is only useful where the querysets being combined do not contain conflicting methods. Method inheritance is supported (i.e. multiple querysets can contain the same method and they will be resolved in normal method resolution order), but depending on the logic of those methods, they may not be compatible. In such cases, an extra class resolving any incompatibilities is still required.

AuditableQuerySet

Changed in version 0.7: Renamed from `CommonInfoQuerySet`. The old name is still available for backwards compatibility, but is considered deprecated.

class djem.models.**AuditableQuerySet**(*model=None*, *query=None*, *using=None*, *hints=None*)

Provides custom functionality pertaining to the fields provided by *Auditable*.

as_manager()

See *MixableQuerySet.as_manager()*.

create(*_user=None*, ***kwargs*)

Overridden to ensure a user is provided to the `save()` call on the model instance. The `_user` argument (named to reduce potential conflicts with model field names) is the user instance to pass through. It is required unless the *DJEM_AUDITABLE_REQUIRE_USER_ON_SAVE* setting is `False`.

New in version 0.7.

get_or_create(*defaults=None*, *_user=None*, ***kwargs*)

Overridden to ensure a user is provided to the `save()` call on the model instance, if a record needs to be created. The `_user` argument (named to reduce potential conflicts with model field names) is the user instance to pass through. It is required unless the *DJEM_AUDITABLE_REQUIRE_USER_ON_SAVE* setting is `False`.

New in version 0.7.

update(*_user=None*, ***kwargs*)

Overridden to ensure the `user_modified` and `date_modified` fields are always updated. The `_user` argument (named to reduce potential conflicts with model field names) is the user instance to update `user_modified` with. It is required unless the *DJEM_AUDITABLE_REQUIRE_USER_ON_SAVE* setting is `False`.

update_or_create(*defaults=None*, *_user=None*, ***kwargs*)

Overridden to ensure a user is provided to the `save()` call on the model instance, whether the record id being created or updated. The `_user` argument (named to reduce potential conflicts with model field names) is the user instance to pass through. It is required unless the *DJEM_AUDITABLE_REQUIRE_USER_ON_SAVE* setting is `False`.

New in version 0.7.

owned_by(*user*)

Return a queryset of records "owned" by the given user, as per the `user_created` field. `user` can be a `User` instance or an id.

ArchivableQuerySet

class djem.models.**ArchivableQuerySet**(*model=None*, *query=None*, *using=None*, *hints=None*)

Provides custom functionality pertaining to the `is_archived` field provided by *Archivable*.

as_manager()

See *MixableQuerySet.as_manager()*.

archived()

Filter the queryset to archived records (`is_archived=True`).

New in version 0.7.

unarchived()

Filter the queryset to unarchived records (`is_archived=False`).

New in version 0.7.

VersionableQuerySet

Changed in version 0.7: Renamed from `VersioningQuerySet`. The old name is still available for backwards compatibility, but is considered deprecated.

class djem.models.**VersionableQuerySet**(*model=None*, *query=None*, *using=None*, *hints=None*)

Provides custom functionality pertaining to the `version` field provided by *Versionable*.

as_manager()

See *MixableQuerySet.as_manager()*.

update(***kwargs*)

Overridden to ensure the `version` field is always updated.

StaticAbstract

class djem.models.**StaticAbstract**

`StaticAbstract` is a combination of *Auditable*, *Archivable* and *Versionable*. It is designed as an abstract base class for models, rather than a mixin itself. It includes all the fields and functionality offered by each of the mixins.

3.9.2 Model Fields

TimeZoneField

class djem.models.**TimeZoneField**(***kwargs*)

`TimeZoneField` is a model field that stores timezone name strings ('Australia/Sydney', 'US/Eastern', etc) in the database and provides access to *`TimeZoneHelper`* instances for the stored timezones. `TimeZoneField` will only store valid timezone strings, or a null value if `null=True`. It will not store arbitrary strings, including the empty string.

The default form field is a `TypedChoiceField` with a `Select` widget.

`TimeZoneField` provides default values for the following constructor arguments:

choices

Defaults to a list of 2-tuples containing the timezones provided by pytz.common_timezones. Both items of each 2-tuple simply contain the timezone name. This is equivalent to:

```
choices = [(tz, tz) for tz in pytz.common_timezones]
```

If passing in a custom list of choices, it must match this format. The default value is stored on `TimeZoneField` in the `CHOICES` constant.

max_length

Defaults to 63. This default value is stored on `TimeZoneField` in the `MAX_LENGTH` constant.

Example, using `TimeZoneField` on a custom User model:

```
# models.py
from django.contrib.auth.models import AbstractBaseUser
from djem.models import TimeZoneField

class CustomUser(AbstractBaseUser):
    ...
    time_zone = TimeZoneField()
```

```
>>> user = CustomUser.objects.filter(timezone='Australia/Sydney').first()
>>> user.timezone
<TimeZoneHelper: Australia/Sydney>
```

Note: Use of `TimeZoneField` requires pytz to be installed. It will raise an exception during instantiation if `pytz` is not available.

Note: Use of `TimeZoneField` only makes sense if USE_TZ is True.

See also:

The *`djem.forms.TimeZoneField`* form field.

3.9.3 Form Classes and Mixins

UserSavable

New in version 0.7.

class djem.forms.**UserSavable**

A mixin for a Django `ModelForm` that adds support for models using the *Auditable* model mixin. It handles providing a user instance to the model's `save()` method when the form's own `save()` method is called, as is required by *Auditable*.

This mixin *assumes* the presence of a `self.user` attribute that the `save()` method can use. It is designed for use by forms that already accept and store a known user, e.g. as a constructor argument.

For a form that provides the same customisation of the `save()` method and includes `user` as a constructor argument, see *AuditableForm*.

AuditableForm

Changed in version 0.7: Renamed from `CommonInfoForm`. The old name is still available for backwards compatibility, but is considered deprecated.

class djem.forms.**AuditableForm**(**args*, *user=None*, ***kwargs*)

A Django `ModelForm` that is customised to support models using the *Auditable* model mixin. It handles providing a user instance to the model's `save()` method when the form's own `save()` method is called, as is required by *Auditable*. It also adds a `user` keyword argument to the constructor so the `save()` method has a known user to work with. The `user` argument is required if the field is bound, otherwise it is optional. The given user is stored in the *user* instance attribute. Subclasses may choose to use the known user for their own purposes.

For a mixin that provides the same customisation of the `save()` method without the extra constructor argument (e.g. for use by forms that already accept and store a known user), see *UserSavable*.

user

The user model instance provided to the constructor on instantiation. May be `None` on unbound forms.

3.9.4 Form Fields

TimeZoneField

class djem.forms.**TimeZoneField**(***kwargs*)

A `TypedChoiceField` with defaults applied for `coerce` and `choices`.

- Default widget: `Select`.
- Empty value: `''` (the empty string), by default.
- Normalizes to: a *TimeZoneHelper* instance.
- Validates that the given value exists in the list of choices and can be coerced.
- Error message keys: `required`, `invalid_choice`.

coerce

Defaults to a function that accepts a timezone name string ('Australia/Sydney', 'US/Eastern', etc) and returns a *TimeZoneHelper* instance for that timezone.

choices

Defaults to a list of 2-tuples containing the timezones provided by pytz.common_timezones. Both items of each 2-tuple simply contain the timezone name. This is equavalient to:

```
choices = [(tz, tz) for tz in pytz.common_timezones]
```

Note: Use of `TimeZoneField` requires pytz to be installed. If `pytz` is not available, the default `choices` list will be empty and no *TimeZoneHelper* objects will be able to be instantiated.

Note: Use of `TimeZoneField` only makes sense if USE_TZ is True.

See also:

The *djem.models.TimeZoneField* model field.

3.9.5 Auth

ObjectPermissionsBackend

class djem.auth.**ObjectPermissionsBackend**

A custom Django authentication backend providing support for object-level permissions.

Does not provide any additional mechanisms for authentication, only authorisation.

To use, list in the Django `AUTHENTICATION_BACKENDS` setting:

```
AUTHENTICATION_BACKENDS = [
    'django.contrib.auth.backends.ModelBackend',
    'djem.auth.ObjectPermissionsBackend'
]
```

permission_required

djem.auth.**permission_required**(**perms*, *login_url=None*, *raise_exception=settings.DJEM_DEFAULT_403*)

New in version 0.5.

A replacement for Django's own `permission_required` decorator that adds support for object-level permissions.

Object-level permissions are specified as a tuple of two strings: the first naming the permission, the second naming the view function keyword argument that contains the primary key of the object to test.

E.g. Specifying one standard (model-level) and one object-level permission:

```
from djem.auth import permission_required

@permission_required('polls.view_questions', ('polls.vote_on_question', 'question'))
def cast_vote(request, question):
    ...
```

Checking the permission involves querying for an instance of the model the permission is for, using the primary key specified in the named argument. If such an instance cannot be found, a `Http404` exception is raised. If an instance *is* found, and the user has the appropriate permission, the primary key argument is *replaced* with the instance. This allows the view access to the instance without needing to query for it again.

In the example above, `question` argument as seen by the view will be a `Question` instance, not the primary key as was originally passed to the function.

Behaviour of the `login_url` and `raise_exception` keyword arguments is as per the original, except that the default value for `raise_exception` can be specified with the *DJEM_DEFAULT_403* setting.

PermissionRequiredMixin

class djem.auth.**PermissionRequiredMixin**

New in version 0.5.

A replacement for Django's own `PermissionRequiredMixin` class-based view mixin that adds support for object-level permissions.

Object-level permissions are specified as a tuple of two strings: the first naming the permission, the second naming the view function keyword argument that contains the primary key of the object to test.

The permission/s can be specified as an attribute of the view class, or via the URLconf.

E.g. Specifying one standard (model-level) and one object-level permission as an attribute of the class:

```
# views.py
from django.views import View
from djem.auth import PermissionRequiredMixin

class CastVote(PermissionRequiredMixin, View):

    permission_required = ['polls.view_questions', ('polls.vote_on_question',
↪'question')]
    ...
```

E.g. Specifying one standard (model-level) and one object-level permission via the URLconf:

```
# urls.py
from django.conf.urls import url
from .views import CastVote

urlpatterns = [
    url(
        r'^question/(?P<question>\d+)/votes/cast/$',
        CastVote.as_view(permission_required=['polls.view_questions', ('polls.vote_
↪on_question', 'question')]),
        name='cast-vote'
    )
]
```

As with Django's version, the `permission_required` attribute can be specified as a single permission or a sequence of permissions. When specifying only a single permission, only model-level permissions (i.e. a string) are valid - object-level permissions (i.e. a two-tuple) are not. Object-level permissions must always be provided as an item of an iterable.

```
class CastVote(PermissionRequiredMixin, View):

    # Valid
    permission_required = 'polls.view_questions'

    # Valid
    permission_required = ['polls.view_questions', 'polls.vote_on_question']

    # Valid
    permission_required = ['polls.view_questions', ('polls.vote_on_question',
↪'question')]

    # INVALID
    permission_required = ('polls.vote_on_question', 'question')
```

Checking the permission involves querying for an instance of the model the permission is for, using the primary key specified in the named argument. If such an instance cannot be found, a `Http404` exception is raised. If an instance *is* found, and the user has the appropriate permission, the primary key argument is *replaced* with the instance. This allows the view access to the instance without needing to query for it again.

In the examples above, `question` argument as seen by the view will be a `Question` instance, not the primary key as was originally passed to the function.

Behaviour of the `login_url` and `raise_exception` attributes is as per the original, except that the default value for `raise_exception` can be specified with the *DJEM_DEFAULT_403* setting.

3.9.6 Pagination

get_page

djem.pagination.**get_page**(*number*, *object_list*, *per_page=None*, ***kwargs*)

New in version 0.5.

A simple wrapper around a Django `Paginator` that immediately invokes its `page()` method and returns a `Page` object.

`number` is the number of the page to retrieve, as a 1-based index. If the given value is not an integer, or it is less than 1, it is treated as 1. If it is greater than the total number of pages, it is treated as `Paginator.num_pages`.

`object_list` is the sequence of items from which to retrieve the specified page - as a list, tuple, `QuerySet` or any other sliceable object with a `count()` or `__len__()` method.

`per_page` is the number of results to be included in each page. Not required if *DJEM_DEFAULT_PAGE_LENGTH* has been defined.

All other keyword arguments of the `Paginator` constructor are also accepted and passed through to the `Paginator` instance created internally.

See also:

DJEM_DEFAULT_PAGE_LENGTH
: The setting for controlling the default value of the `per_page` argument.

paginate
: A templatetag for rendering a block of pagination links based on a given `Page` object.

3.9.7 Middleware

MessageMiddleware

class djem.middleware.**MessageMiddleware**

New in version 0.6.

Middleware that handles temporary messages, differentiating between those added as part of an AJAX request vs those that are not. Different storage backends are used for each: the default backend (configurable via `MESSAGE_STORAGE`) for standard requests and a custom memory-only backend for AJAX requests.

Using a memory-only backend, which does not offer any persistence between requests, prevents simultaneous AJAX requests interfering with each other's message stores, avoiding the documented caveat of Django's out-of-the-box messages framework.

This is a drop-in replacement for Django's own `MessageMiddleware`:

```
# before
MIDDLEWARE = [
    ...
    'django.contrib.messages.middleware.MessageMiddleware'
    ...
]

# after
MIDDLEWARE = [
    ...
    'djem.middleware.MessageMiddleware'
    ...
]
```

Important: `MessageMiddleware` uses the `HttpRequest` object's `is_ajax()` method to differentiate between AJAX and non-AJAX requests. An `XMLHttpRequest` call must use the appropriate headers in order to be correctly detected.

See also:

AjaxResponse
An extension of Django's `JsonResponse` that, among other things, will automatically include any messages that are in the message store as part of the response.

3.9.8 AJAX

ajax_login_required

New in version 0.7.

djem.ajax.**ajax_login_required**(*view_fn*)

Simple decorator that returns `HttpResponseForbidden` if the user is not authenticated, and executes the decorated function if they are. Designed for use by views accessed with AJAX requests, where redirecting to the login view is not useful.

AjaxResponse

class djem.ajax.**AjaxResponse**(*request*, *data=None*, *success=None*, ***kwargs*)

An extension of Django's `JsonResponse`, differing in the following ways:

- The `data` argument is optional. If provided, it must always be a `dict` instance. If not provided, a new `dict` instance will be created and used. Using the `safe` argument of `JsonResponse` to JSON-encode other types is not supported (see the documentation for the `safe` argument of `JsonResponse`).
- The first positional argument should be a Django `HttpRequest` instance, used to retrieve messages from the Django message framework store and add them to the `data` dictionary under the "messages" key. The messages are added as a list of dictionaries containing:
  - message: The message string.
  - tags: A string of the tags applied to the message, space separated.
- The optional argument `success` can be set to add a "success" key to the `data` dictionary. The "success" key will always be added as a boolean value, regardless of what was passed (though it will not be added at all if nothing was passed).

With the exception of `safe`, as noted above, `AjaxResponse` accepts and supports all arguments of `JsonResponse`.

To help prevent XSS vulnerabilities, the messages from the Django messages framework that are included in the response are automatically escaped. If a message should legitimately contain HTML, it can be marked as safe to prevent it being escaped.

See also:

MessageMiddleware
A replacement for Django's own `MessageMiddleware` that avoids simultaneous requests interfering with each other's message stores - an issue made more likely when making use of AJAX.

3.9.9 Template Tags

Djem provides several template tags as part of the `djem` tag library.

Many of the tags are designed to help keep things DRY and alleviate boilerplate code. The output of such tags is often dictated by a template, for which Djem will provide a default, but which can be overridden per-project using Django's standard template overriding mechanism.

ifperm

The `{% ifperm %}` block tag performs an object-level permission check and, if that check *passes*, renders contents of the block. To perform the permission check, it requires a `User` instance, the name of the permission and a model instance. The check will pass if the given user has the given permission on the given instance.

```
{% load djem %}
...
{% ifperm user 'polls.vote_on_question' question_obj %}
    <a href="{% url 'vote' question_obj.pk %}">Vote Now</a>
{% endifperm %}
...
```

The tag supports an `else` block, which will be rendered if the user does not have permission.

```
{% load djem %}
...
{% ifperm user 'polls.vote_on_question' question_obj %}
    <a href="{% url 'vote' question_obj.pk %}">Vote Now</a>
{% else %}
    You do not have permission to vote on this question.
{% endifperm %}
...
```

ifnotperm

The `{% ifnotperm %}` block tag performs an object-level permission check and, if that check *fails*, the contents of the block are rendered. To perform the permission check, it requires a `User` instance, the name of the permission and a model instance. The check will fail if the given user *does not* have the given permission on the given instance.

```
{% load djem %}
...
{% ifnotperm user 'polls.vote_on_question' question_obj %}
    You do not have permission to vote on this question.
{% endifnotperm %}
...
```

The tag supports an `else` block, which will be rendered if the user does have permission.

```
{% load djem %}
...
{% ifnotperm user 'polls.vote_on_question' question_obj %}
    You do not have permission to vote on this question.
{% else %}
    <a href="{% url 'vote' question_obj.pk %}">Vote Now</a>
{% endifnotperm %}
...
```

csrfify_ajax

New in version 0.6.

The `{% csrfify_ajax %}` template tag renders a HTML `<script>` tag containing JavaScript to configure the `X-CSRFToken` header on outgoing AJAX requests where necessary (e.g. POST requests). The JavaScript is library-specific, and is stored in templates under `djem/csrfify_ajax/`, e.g. `djem/csrfify_ajax/jquery.html`.

Support for jQuery is included by default. Additional libraries can be added by creating project-specific templates for them under the `djem/csrfify_ajax/` path and providing the name of the template as an argument to the tag. Such templates have access to the CSRF token via the `{{ csrf_token }}` template variable.

```
{% load djem %}

{# Uses jquery as default argument #}
{% csrfify_ajax %}

{# But it can be given explicitly #}
{% csrfify_ajax 'jquery' %}
```

(continues on next page)

(continued from previous page)

```
{# As can another library, provided a template exists to support it #}
{% csrfify_ajax 'some_other_lib' %}
```

paginate

New in version 0.6.

The `{% paginate %}` template tag renders the pagination links associated with a result list. Simply pass it the same Django `Page` instance used to render the list itself and it will render appropriate page navigation links.

For example, where `user_list` is a `Page` instance:

```
{% load djem %}
...
{% for user in user_list %}
    {{ user.name }}
{% endfor %}
{% paginate user_list %}
...
```

The structure of the navigation block that is rendered is controlled by the `djem/pagination.html` template.

See also:

get_page()
: A helper utility for retrieving a `Page` instance.

form_field

New in version 0.6.

The `{% form_field %}` template tag renders a form field in a consistent, customisable fashion. The structure of the output is controlled by the `djem/form_field.html` template. By default, it provides:

- A wrapping element with the following features:
  - By default, the element is a `<div>`, but this can be modified using the *DJEM_FORM_FIELD_TAG* setting.
  - The element is given the following CSS classes:
    - `form-field`
    - Any classes defined by the form, either declaratively or programatically.
    - Any classes passed to the template tag.
  - The element is given additional HTML attributes based on keyword arguments passed to the template tag.
- The field's label.
- The field's error list (when applicable). The error list is the Django default `<ul class="errorlist">`.
- The field itself.
- A `<div>` with the CSS class `form-field__help` containing the field's help text, if any.

```
{% load djem %}
...
{# Basic usage #}
{% form_field form.first_name %}

{# Adding extra CSS classes to the wrapper #}
{% form_field form.first_name 'one-half' %}
{% form_field form.last_name 'one-half' %}

{# Adding extra HTML attributes to the wrapper #}
{% url 'verify-email' as verify_email_url %}
{% form_field form.email data_url=verify_email_url %}
...
```

Note: Since attribute names can contain dashes, which are invalid in Python keyword argument names, any underscores in the argument name will be converted into dashes to form the HTML attribute name.

checkbox

New in version 0.6.

The `{% checkbox %}` template tag is very similar to the `{% form_field %}` tag. It likewise renders form fields, and the same `djem/form_field.html` template controls the output, but it is specifically designed for checkboxes. The differences lie in the way the field's `<label>` element is rendered:

- it is included *after* the field itself, not before
- it is given the `check-label` CSS class, allowing it to be styled independently of regular labels

Also, unlike `{% form_field %}`, `{% checkbox %}` is a *block tag*. It uses the content between its start and end tags as the label for the field. Specifically, this allows HTML to be included in the label text:

```
{% load djem %}
...
{% checkbox form.terms %}
    I agree to the <a href="{% url 'terms' %}" target="_blank">Terms of Service</a>.
{% endcheckbox %}
...
```

If no content is entered between the start and end tags, the field's default label text is used. In this case, the `<label>` element will still be included after the field itself, instead of before, and will still receive the `check-label` class.

Note: Don't go too crazy with HTML in the label text. It is still rendered inside a `<label>` element, so should only contain markup that is valid within `<label>`.

Note: `{% checkbox %}` is not strictly limited to actual checkbox inputs. You could, if for some reason it was appropriate, use it for any form field.

3.9.10 Utils

General Utils

UNDEFINED

djem.**UNDEFINED**

New in version 0.7.

The `UNDEFINED` constant is a falsey value designed for use in argument default values, for situations in which *any* value given (including traditional argument default values such as `None`) have an explicit meaning.

```
from djem import UNDEFINED

def make_thing(name, label=UNDEFINED):

    # None, the empty string, or False could indicate "no label" and need
    # to be distinguished from no value being provided
    if label is UNDEFINED:
        label = name

    ...
```

It can also be used in more generic conditional statements:

```
from djem import UNDEFINED

value = UNDEFINED

if value:
    print('truthy')
else:
    print('falsey')

# output: 'falsey'
```

Date/Time Utils

TimeZoneHelper

class djem.utils.dt.**TimeZoneHelper**(*tz*)

A simple helper class that provides shortcuts for getting the current date and the current datetime for a known local timezone.

`tz` should be a valid timezone name string (as accepted by the `pytz.timezone` function) or a `pytz tzinfo` instance (as returned by the `pytz.timezone` function).

tz

The `pytz tzinfo` instance representing the timezone used by this `TimeZoneHelper` instance.

name

The name of the timezone represented by this `TimeZoneHelper` instance, as a string. Equivalent to `tz.zone`, where `tz` is the *instance's tz attribute*.

now()

Return the current datetime in the local timezone.

today()

Return the current date in the local timezone.

> **Warning:** Be careful when dealing with local times. Django recommends you "use UTC in the code and use local time only when interacting with end users", with the conversion from UTC to local time usually only being performed in templates. And the pytz documentation notes "The preferred way of dealing with times is to always work in UTC, converting to localtime only when generating output to be read by humans". See the Django timezone documentation and the pytz documentation.

Test Utils

setup_test_app()

New in version 0.7.

djem.utils.tests.**setup_test_app**(*package*, *label=None*)

Setup a Django test app for the provided package to allow test-only models to be used.

This function should be called from `myapp.tests.__init__` like so:

```
setup_test_app(__package__)
```

This will create an app with the label "myapp_tests". If a specific app label is required, it can be provided explicitly:

```
setup_test_app(__package__, 'mytests')
```

Using either of the above, models can be placed in `myapp.tests.models` and be discovered and used just like regular models by the test suite. As long as `myapp.tests` is not imported by anything that forms part of the standard Django runtime environment, these models will not be picked up in that environment, and will be isolated to the test suite only.

This solution is adapted from Simon Charette's comment on Django ticket #7835.

MessagingRequestFactory

New in version 0.6.

class djem.utils.tests.**MessagingRequestFactory**

An extension of Django's `RequestFactory` helper for tests that enables the use of the messages framework within the generated request. It does not use the standard message storage backend (as per the `MESSAGE_STORAGE` setting), but rather a memory-only backend that does not involve the use of sessions, cookies or any other means of persistent storage of the messages. Thus, messages need to be read in the same request they were added, or they will be lost.

It is used in the same way as `RequestFactory`:

```
from django.contrib.auth.models import User
from django.contrib import messages
from django.test import TestCase

from djem.utils.tests import MessagingRequestFactory

# Views expected to set messages
from .views import MyView, my_view


class SimpleTest(TestCase):

    def setUp(self):

        self.factory = MessagingRequestFactory()
        self.user = User.objects.create_user(
            username='test.user', email='test@...', password='top_secret'
        )

    def test_details(self):

        request = self.factory.get('/customer/details')

        # Recall that middleware are not supported. You can simulate a
        # logged-in user by setting request.user manually.
        request.user = self.user

        # Test my_view() as if it were deployed at /customer/details
        response = my_view(request)

        # Use this syntax for class-based views
        response = MyView.as_view()(request)

        self.assertEqual(response.status_code, 200)

        # Test the expected message was set
        message_list = list(messages.get_messages(request))
        self.assertEqual(len(message_list), 1)
        self.assertEqual(message_list[0].message, 'An error occurred.')
```

TemplateRendererMixin

New in version 0.6.

class djem.utils.tests.**TemplateRendererMixin**

A mixin for `TestCase` classes whose tests render templates from strings (as opposed to rendering them from files), using the Django template engine.

render_template(*template_string*, *context*, *request=None*, *flatten=True*)

Render the given string as a template, with the given context and request (if provided).

If `request` is NOT provided, and a `self.user` attribute is available on the `TestCase`, a "user" variable will be automatically added to the context.

The rendered output will be stripped of any leading or trailing whitespace, and can optionally have excess whitespace "flattened" by passing the `flatten` argument as True (the default). Flattening removes ALL whitespace from between HTML tags and compresses all other whitespace down to a single space.

Parameters

- **template_string** – The string to render as a template.
- **context** – The context with which to render the template.
- **request** – The `HttpRequest` with which to render the template context.
- **flatten** – True to "flatten" the rendered output (default), False to return the output with all internal whitespace intact.

Returns

The rendered template output.

```
from django.contrib.auth.models import User
from django.test import TestCase

from djem.utils.tests import TemplateRendererMixin


class SomeTestCase(TemplateRendererMixin, TestCase):

    def setUp(self):

        self.user = self.user = User.objects.create_user(
            username='test.user', email='test@...', password='top_secret'
        )

    def test_something(self):

        template_string = (
            '{% if something %}'
            '    <p>'
            '        The user is: {{ user.username }}'
            '    </p>'
            '{% endif %}'
        )

        output = self.render_template(template_string, {
            'something': True
        })

        self.assertEqual(output, '<p> The user is: test.user </p>')
```

3.9.11 Settings

The following settings can be added to your project's `settings.py` file to customise the behaviour of Djem features.

DJEM_AUDITABLE_REQUIRE_USER_ON_SAVE

Changed in version 0.7: Renamed from `DJEM_COMMON_INFO_REQUIRE_USER_ON_SAVE`. The old setting is still available for backwards compatibility, but is considered deprecated.

Default: `True`

By default, the *save()* method of *Auditable* and various methods of *AuditableQuerySet* (e.g. *create()*, *update()*, etc) require being passed a user model instance, so they can automatically set the `user_created` and/or `user_modified` fields that `Auditable` provides. This behaviour can cause issues if you are using third party code that calls any of these methods, as it will not pass this required argument.

Djem provides *AuditableForm* and *UserSavable* to enable Django `ModelForms` to work with models making use of `Auditable`. But if you can't use similar wrappers around other third party code invoking these methods, this setting can help.

Setting `DJEM_AUDITABLE_REQUIRE_USER_ON_SAVE` to `False` removes the "required" nature of this additional user argument. It will still be accepted, and will still be used as per usual if it is provided. But if it is not provided, no exception will be raised, and the fields that would ordinarily be populated by it will simply be left alone.

Warning: The `user_created` and `user_modified` fields will still be required. When creating instances of models using `Auditable`, and a user instance is not passed to the *save()* method, these fields will need to be populated manually, or an `IntegrityError` will be raised.

Warning: Using this setting can reduce the accuracy of `user_modified`, as it is no longer guaranteed to be updated by any save/update to the model instance. It will be up to you to ensure that this field is updated when necessary.

DJEM_DEFAULT_403

New in version 0.5.

Default: `False`

Specifies the default behaviour of the *permission_required()* decorator and `djem.auth.PermissionRequired` class-based view mixin when a user does not have the specified permission/s. If `True`, the `PermissionDenied` exception will be raised, invoking the 403 handler. If `False`, the user will be redirected to the appropriate login url.

This affects default behaviour only - individual uses of *permission_required()* and `djem.auth.PermissionRequired` can customise it.

DJEM_DEFAULT_PAGE_LENGTH

New in version 0.5.

Default: `None`

The default page length to use for the *djem.pagination.get_page()* helper function. Adding this setting removes the need to provide a page length argument to every *djem.pagination.get_page()* call. See *Controlling page length* for more details.

DJEM_FORM_FIELD_TAG

New in version 0.6.

Default: `'div'`

The HTML tag to use for the wrapping element rendered around form fields when using the *form_field* or *checkbox* template tags.

DJEM_UNIVERSAL_OLP

New in version 0.7.

Default: `False`

In conjunction with a custom user model including *OLPMixin*, setting this to `True` enables support for forcing superusers to undergo the same object-level permissions checking that regular users do, allowing OLP logic to actually *deny* permissions to superusers where relevant.

DJEM_PERM_LOG_VERBOSITY

New in version 0.7.

Default: `0`

In conjunction with a custom user model including *OLPMixin*, this setting controls the level of *automatic permission logging* performed by *OLPMixin.has_perm()*:

- `0`: No automatic logging
- `1`: Logs are automatically created for each permission check, with minimal automatic entries
- `2`: Logs are automatically created for each permission check, with more informative automatic entries

In addition to the automatic entries, a value of `1` or `2` allow manual log entries to be added from within object-level access methods with no need to manually start/finish any logs.

3.9.12 Exceptions

The following custom exceptions are raised by Djem.

exception djem.exceptions.**ModelAmbiguousVersionError**

Raised when a model's `version` field contains an ambiguous value, such as after it has been atomically incremented.

3.10 Change Log

3.10.1 0.9.0 (unreleased)

- Improved customisability of pagination template
- Updated `UNDEFINED` to ensure it cannot be deep copied (it will always be the same instance)
- Updated the `Archivable.archive()` method to raise `ProtectedError` and `RestrictedError` as `delete()` would, when inbound foreign keys using `on_delete=models.PROTECT` or `on_delete=models.RESTRICT` are detected against unarchived records
- Updated `Loggable` to support tagging individual log entries and subsequently filtering retrieved log entries

3.10.2 0.8.0 (2022-12-12)

- Added compatibility with Django versions 4.0 and 4.1
- Dropped support for Django versions below 3.2
- Updated `Auditable` and `Versionable` to use `self._state.adding` to differentiate between records being added or updated
- Switched development environment from Vagrant to Docker

Note: As of this release, `TimeZoneField`/`TimeZoneHelper` still require `pytz`, despite the Django 4.0+ move to `zoneinfo`.

3.10.3 0.7.2 (2021-10-05)

- Removed public `djem.auth.get_user_log_verbosity()` function due to the apparent potential for causing cyclic imports, as observed in real world usage.

3.10.4 0.7.1 (2021-08-25)

- Fixed readthedocs configuration
- Fixed README badges

3.10.5 0.7 (2021-08-24)

A number of features are renamed. In all cases, the old names remain available for backwards compatibility, but are deprecated.

- Dropped support for Python 2 (minimum supported version is 3.6)
- Dropped support for Django versions below 2.2
- Renamed `CommonInfoMixin` and `CommonInfoQuerySet` to `Auditable` and `AuditableQuerySet`, respectively
- Renamed `ArchivableMixin` to `Archivable`
- Renamed `VersioningMixin` and `VersioningQuerySet` to `Versionable` and `VersionableQuerySet`, respectively
- Renamed `CommonInfoForm` to `AuditableForm`

- Renamed `DJEM_COMMON_INFO_USER_REQUIRED_ON_SAVE` setting to `DJEM_AUDITABLE_USER_REQUIRED_ON_SAVE`
- Added `UNDEFINED` constant
- Added `Loggable` for instance-based logging
- Added `OLPMixin` for custom user models, to support advanced OLP-related functionality
- Added `MixableQuerySet` mixin for custom `QuerySet` classes
- Added `DJEM_UNIVERSAL_OLP` setting
- Added `DJEM_PERM_LOG_VERBOSITY` setting
- Added `AuditableQuerySet` `create()`, `get_or_create()`, and `update_or_create()` methods
- Added `ArchivableQuerySet` `archived()` and `unarchived()` methods
- Added `ajax_login_required()` decorator
- Added `UserSavable` mixin for forms
- Removed default implementations of `_user_can_change_*()` and `_user_can_delete_*()` on `Auditable` - this was far too specific a use-case to be the default
- Removed `ArchivableQuerySet`'s `archive()` and `unarchive()` methods
- Removed `Archivable`'s `live` and `archived` Managers
- Removed explicit `Manager` classes for mixins
- Moved custom `QuerySet` classes for mixins into `djem.models.models`

3.10.6 0.6.4 (2018-12-06)

- Fixed setup.py to include `include_package_data=True`

3.10.7 0.6.3 (2018-12-06)

- Updated MANIFEST.in to include the templates directory

3.10.8 0.6.2 (2018-03-25)

- Fixed #2: Object level access now defaults open when no model method exists to define it explicitly

3.10.9 0.6.1 (2018-03-02)

- Updated PyPi details

3.10.10 0.6 (2018-03-02)

- Renamed project
- Added `csrfify_ajax` template tag
- Added `paginate` template tag
- Added `form_field` and `checkbox` template tags
- Added `MessageMiddleware`
- Added `MessagingRequestFactory`
- Added `TemplateRendererMixin`
- Updated `AjaxResponse` to allow message strings marked as safe to skip being escaped
- Moved `AjaxResponse` from `djem.misc.AjaxResponse` to `djem.ajax.AjaxResponse`. Also removed shortcut import `djem.AjaxResponse`.
- Moved `get_page()` from `djem.misc.get_page` to `djem.pagination.get_page`. Also removed shortcut import `djem.get_page`.

3.10.11 0.5 (unreleased)

Never released: project renaming took precedence. These features were released under 0.6, and the new project name, instead.

- Added replacements for `permission_required` decorator and `PermissionRequiredMixin` that support object-level permissions
- Added `get_page()` helper function
- Added `Table` helper class
- Added `M` and `Mon` helper classes for simple code performance debugging
- Added `mon()` decorator as a shortcut for monitoring a function
- Added inspection/prettyprint utilities for debugging
- Added extensible `Developer` class as a home for shortcuts to common user-based operations useful to developers

3.10.12 0.4.3 (2016-09-17)

- Added `authenticate()` method to `ObjectPermissionsBackend`, fixing a bug where it broke authentication if a user's credentials were not authenticated by earlier backends

3.10.13 0.4.2 (2016-06-21)

- Fixed missing commits under 0.4.1 tag

3.10.14 0.4.1 (2016-06-21)

- Fixed documentation build issues on `readthedocs.org`

3.10.15 0.4 (2016-06-21)

- Added `AjaxResponse`
- Added `GOODIES_COMMON_INFO_REQUIRE_USER_ON_SAVE` setting
- Added object-level permission support (`ObjectPermissionsBackend`, `ifperm` and `ifnotperm` template tags)
- Updated `CommonInfoMixin` to provide default object-level permissions for subclasses, based on ownership

3.10.16 0.3 (2016-03-19)

- Added `TimeZoneField`/`TimeZoneHelper`
- Cleaned code as per `isort` and `flake8`

3.10.17 Pre-0.3

- `CommonInfoMixin`, with associated manager and queryset
- `ArchivableMixin`, with associated manager and queryset
- `VersioningMixin`, with associated manager and queryset
- `StaticAbstract` parent model, with associated manager and queryset
- `CommonInfoForm` for ModelForms based on `CommonInfoMixin` models

CHAPTER

FOUR

LICENSE

Djem is released under the BSD license.

PYTHON MODULE INDEX

d

INDEX