django-braces Documentation

Release 1.3.1

Kenneth Love and Chris Jones

February 28, 2014

Contents

You can view the code of our project or fork it and add your own mixins (please, send them back to us), on Github.

CHAPTER 1

Access Mixins

These mixins all control a user's access to a given view. Since they all extend the `AccessMixin`, the implement a common API that includes the following class attributes:

```
login_url = settings.LOGIN_URL
redirect_field_name = REDIRECT_FIELD_NAME
raise_exception = False
```

The `raise_exception` attribute will cause the view to raise a `PermissionDenied` exception if it is set to `True`, otherwise the view will redirect to the login view provided.

Contents

1.1 LoginRequiredMixin

This mixin is rather simple and is generally the first inherited class in any of our views. If we don't have an authenticated user there's no need to go any further. If you've used Django before you are probably familiar with the `login_required` decorator. All we are doing here is requiring a user to be authenticated to be able to get to this view.

While this doesn't look like much, it frees us up from having to manually overload the dispatch method on every single view that requires a user to be authenticated. If that's all that is needed on this view, we just saved 3 lines of code. Example usage below.

Note: As of version 1.0, the LoginRequiredMixin has been rewritten to behave like the rest of the `access` mixins. It now accepts `login_url`, `redirect_field_name` and `raise_exception`.

Note: This should be the left-most mixin of a view, except when combined with *CsrfExemptMixin* - which in that case should be the left-most mixin.

```
from django.views.generic import TemplateView

from braces.views import LoginRequiredMixin


class SomeSecretView(LoginRequiredMixin, TemplateView):
    template_name = "path/to/template.html"

    #optional
    login_url = "/signup/"
    redirect_field_name = "hollaback"
    raise_exception = True

    def get(self, request):
        return self.render_to_response({})
```

1.2 PermissionRequiredMixin

This mixin was originally written, I believe, by Daniel Sokolowski (code here), but we have updated it to eliminate an unneeded render if the permissions check fails.

Rather than overloading the dispatch method manually on every view that needs to check for the existence of a permission, we inherit this class and set the `permission_required` class attribute on our view. If you don't specify `permission_required` on your view, an `ImproperlyConfigured` exception is raised reminding you that you haven't set it.

The one limitation of this mixin is that it can **only** accept a single permission. If you need multiple permissions use `MultiplePermissionsRequiredMixin`.

In our normal use case for this mixin, `LoginRequiredMixin` comes first, then the `PermissionRequiredMixin`. If we don't have an authenticated user, there is no sense in checking for any permissions.

Note: If you are using Django's built in auth system, `superusers` automatically have all permissions in your system.

```
from braces.views import LoginRequiredMixin, PermissionRequiredMixin


class SomeProtectedView(LoginRequiredMixin, PermissionRequiredMixin, TemplateView):
    permission_required = "auth.change_user"
    template_name = "path/to/template.html"
```

1.3 MultiplePermissionsRequiredMixin

The multiple permissions required view mixin is a more powerful version of the `PermissionRequiredMixin`. This view mixin can handle multiple permissions by setting the mandatory `permissions` attribute as a dict with the keys `any` and/or `all` to a list/tuple of permissions. The `all` key requires the request.user to have all of the specified permissions. The `any` key requires the request.user to have at least ONE of the specified permissions. If you only need to check a single permission, the `PermissionRequiredMixin` is all you need.

Note: If you are using Django's built in auth system, `superusers` automatically have all permissions in your system.

```
from braces.views import LoginRequiredMixin, MultiplePermissionsRequiredMixin


class SomeProtectedView(LoginRequiredMixin,
                        MultiplePermissionsRequiredMixin,
                        TemplateView):

    #required
    permissions = {
        "all": ("blog.add_post", "blog.change_post"),
        "any": ("blog.delete_post", "user.change_user")
    }
```

1.4 GroupRequiredMixin

New in version 1.2.

The group required view mixin ensures that the requesting user is in the group or groups specified. This view mixin can handle multiple groups by setting the mandatory `group_required` attribute as a list or tuple.

Note: The mixin assumes you're using Django's default Group model and that your user model provides `groups` as a ManyToMany relationship. If this **is not** the case, you'll need to override `check_membership` in the mixin to handle your custom set up.

1.4.1 Standard Django Usage

```
from braces.views import GroupRequiredMixin


class SomeProtectedView(GroupRequiredMixin, TemplateView):

    #required
    group_required = u'editors'
```

1.4.2 Custom Group Usage

```
from braces.views import GroupRequiredMixin


class SomeProtectedView(GroupRequiredMixin, TemplateView):

    #required
    group_required = u'editors'

    def check_membership(self, group):
        ...
        # Check some other system for group membership
        if user_in_group:
            return True
        else:
            return False
```

1.5 UserPassesTestMixin

New in version dev.

Mixin that reimplements the *user_passes_test* decorator. This is helpful for much more complicated cases than checking if user *is_superuser* (for example if their email is from specific a domain).

```
from braces.views import UserPassesTestMixin

class SomeUserPassView(UserPassesTestMixin, TemplateView):
    def test_func(self, user):
        return (user.is_staff and not user.is_superuser
                and user.email.endswith("mydomain.com"))
```

1.6 SuperuserRequiredMixin

Another permission-based mixin. This is specifically for requiring a user to be a superuser. Comes in handy for tools that only privileged users should have access to.

```
from braces.views import LoginRequiredMixin, SuperuserRequiredMixin


class SomeSuperuserView(LoginRequiredMixin, SuperuserRequiredMixin, TemplateView):
    template_name = "path/to/template.html"
```

1.7 StaffuserRequiredMixin

Similar to `SuperuserRequiredMixin`, this mixin allows you to require a user with `is_staff` set to True.

```
from braces.views import LoginRequiredMixin, StaffuserRequiredMixin


class SomeStaffuserView(LoginRequiredMixin, StaffuserRequiredMixin, TemplateView):
    template_name = "path/to/template.html"
```

CHAPTER 2

Form Mixins

All of these mixins, with one exception, modify how forms are handled within views. The `UserKwargModelFormMixin` is a mixin for use in forms to auto-pop a `user` kwarg.

Contents

2.1 CsrfExemptMixin

If you have Django's *CSRF protection* middleware enabled you can exempt views using the *csrf_exempt* decorator. This mixin exempts POST requests from the CSRF protection middleware without requiring that you decorate the `dispatch` method.

Note: This should always be the left-most mixin of a view.

```
from django.views.generic import UpdateView

from braces.views import LoginRequiredMixin, CsrfExemptMixin

from profiles.models import Profile
```

```
class UpdateProfileView(LoginRequiredMixin, CsrfExemptMixin, UpdateView):
    model = Profile
```

2.2 UserFormKwargsMixin

In one of our client's CMS, we have a lot of form-based views that require a user to be passed in for permission-based form tools. For example, only superusers can delete or disable certain objects. To custom tailor the form for users, we have to pass that user instance into the form and based on their permission level, change certain fields or add specific options within the forms `__init__` method.

This mixin automates the process of overloading the `get_form_kwargs` (this method is available in any generic view which handles a form) method and stuffs the user instance into the form kwargs. We can then pop the user off in the form and do with it what we need. **Always** remember to pop the user from the kwargs before calling `super` on your form, otherwise the form gets an unexpected keyword argument and everything blows up.

2.2.1 Usage

```
from django.views.generic import CreateView

from braces.views import LoginRequiredMixin, UserFormKwargsMixin
from next.example import UserForm


class SomeSecretView(LoginRequiredMixin, UserFormKwargsMixin,
    TemplateView):

    form_class = UserForm
    model = User
    template_name = "path/to/template.html"
```

This obviously pairs very nicely with the following `Form` mixin.

2.3 UserKwargModelFormMixin

The `UserKwargModelFormMixin` is a form mixin to go along with our *UserFormKwargsMixin*. This becomes the first inherited class of our forms that receive the user keyword argument. With this mixin, we have automated the popping off of the keyword argument in our form and no longer have to do it manually on every form that works this way. While this may be overkill for a weekend project, for us, it speeds up adding new features.

2.3.1 Usage

```
from braces.forms import UserKwargModelFormMixin


class UserForm(UserKwargModelFormMixin, forms.ModelForm):
    class Meta:
        model = User

    def __init__(self, *args, **kwargs):
        super(UserForm, self).__init__(*args, **kwargs)
        if not self.user.is_superuser:
            del self.fields["group"]
```

2.4 SuccessURLRedirectListMixin

The `SuccessURLRedirectListMixin` is a bit more tailored to how we have handled CRUD within the CMSes we've built. One CMS's workflow, by design, redirects the user to the `ListView` for whatever model they are working with, whether they are creating a new instance, editing an existing one or deleting one. Rather than having to override `get_success_url` on every view, we simply use this mixin and pass it a reversible route name. Example:

```
# urls.py
url(r"^users/$", UserListView.as_view(), name="cms_users_list"),

# views.py
from braces.views import (LoginRequiredMixin, PermissionRequiredMixin,
    SuccessURLRedirectListMixin)


class UserCreateView(LoginRequiredMixin, PermissionRequiredMixin,
    SuccessURLRedirectListMixin, CreateView):

    form_class = UserForm
    model = User
    permission_required = "auth.add_user"
    success_list_url = "cms_users_list"
    ...
```

2.5 FormValidMessageMixin

New in version 1.2.

The `FormValidMessageMixin` allows you to to *statically* or *programmatically* set a message to be returned using Django's messages framework when the form is valid. The returned message is controlled by the `form_valid_message` property which can either be set on the view or returned by the `get_form_valid_message` method. The message is not processed until the end of the `form_valid` method.

Warning: This mixin requires the Django messages app to be enabled.

Note: This mixin is designed for use with Django's generic form class-based views, e.g. `FormView`, `CreateView`, `UpdateView`

2.5.1 Static Example

```
from django.views.generic import CreateView

from braces.views import FormValidMessageMixin


class BlogPostCreateView(FormValidMessageMixin, CreateView):
    form_class = PostForm
    model = Post
    form_valid_message = 'Blog post created!'
```

2.5.2 Dynamic Example

```
from django.views.generic import CreateView

from braces.views import FormValidMessageMixin


class BlogPostCreateView(FormValidMessageMixin, CreateView):
    form_class = PostForm
    model = Post

    def get_form_valid_message(self):
        return '{0} created!'.format(self.object.title)
```

2.6 FormInvalidMessageMixin

New in version 1.2.

The `FormInvalidMessageMixin` allows you to to *statically* or *programmatically* set a message to be returned using Django's messages framework when the form is invalid. The returned message is controlled by the `form_invalid_message` property which can either be set on the view or returned by the `get_form_invalid_message` method. The message is not processed until the end of the `form_invalid` method.

Warning: This mixin requires the Django messages app to be enabled.

Note: This mixin is designed for use with Django's generic form class-based views, e.g. `FormView`, `CreateView`, `UpdateView`

2.6.1 Static Example

```
from django.views.generic import CreateView

from braces.views import FormInvalidMessageMixin


class BlogPostCreateView(FormInvalidMessageMixin, CreateView):
    form_class = PostForm
    model = Post
    form_invalid_message = 'Oh snap, something went wrong!'
```

2.6.2 Dynamic Example

```python
from django.views.generic import CreateView

from braces.views import FormInvalidMessageMixin


class BlogPostCreateView(FormInvalidMessageMixin, CreateView):
    form_class = PostForm
    model = Post

    def get_form_invalid_message(self):
        return 'Some custom message'
```

2.7 FormMessagesMixin

New in version 1.2.

`FormMessagesMixin` is a convenience mixin which combines *FormValidMessageMixin* and *FormInvalidMessageMixin* since we commonly provide messages for both states (`form_valid`, `form_invalid`).

Warning: This mixin requires the Django messages app to be enabled.

2.7.1 Static & Dynamic Example

```python
from django.views.generic import CreateView

from braces.views import FormMessagesMixin


class BlogPostCreateView(FormMessagesMixin, CreateView):
    form_class = PostForm
    form_invalid_message = 'Something went wrong, post was not saved'
    model = Post

    def get_form_valid_message(self):
        return '{0} created!'.format(self.object.title)
```

CHAPTER 3

Other Mixins

These mixins handle other random bits of Django's views, like controlling output, controlling content types, or setting values in the context.

Contents

3.1 SetHeadlineMixin

The `SetHeadlineMixin` is a newer edition to our client's CMS. It allows us to *statically* or *programmatically* set the headline of any of our views. We like to write as few templates as possible, so a mixin like this helps us reuse generic templates. Its usage is amazingly straightforward and works much like Django's built-in `get_queryset` method. This mixin has two ways of being used.

3.1.1 Static Example

```python
from braces.views import SetHeadlineMixin


class HeadlineView(SetHeadlineMixin, TemplateView):
    headline = "This is our headline"
    template_name = "path/to/template.html"
```

3.1.2 Dynamic Example

```
from datetime import date

from braces.views import SetHeadlineMixin


class HeadlineView(SetHeadlineMixin, TemplateView):
    template_name = "path/to/template.html"

    def get_headline(self):
        return u"This is our headline for %s" % date.today().isoformat()
```

In both usages, in the template, just print out `{{ headline }}` to show the generated headline.

3.2 SelectRelatedMixin

A simple mixin which allows you to specify a list or tuple of foreign key fields to perform a select_related on. See Django's docs for more information on select_related.

```
# views.py
from django.views.generic import DetailView

from braces.views import SelectRelatedMixin

from profiles.models import Profile


class UserProfileView(SelectRelatedMixin, DetailView):
    model = Profile
    select_related = ["user"]
    template_name = "profiles/detail.html"
```

3.3 PrefetchRelatedMixin

A simple mixin which allows you to specify a list or tuple of reverse foreign key or ManyToMany fields to perform a prefetch_related on. See Django's docs for more information on prefetch_related.

```
# views.py
from django.contrib.auth.models import User
from django.views.generic import DetailView

from braces.views import PrefetchRelatedMixin


class UserView(PrefetchRelatedMixin, DetailView):
    model = User
    prefetch_related = ["post_set"]  # where the Post model has an FK to the User model as an author
    template_name = "users/detail.html"
```

3.4 JSONResponseMixin

Changed in version 1.1: `render_json_response` now accepts a `status_code` keyword argument. `json_dumps_kwargs` class-attribute and `get_json_dumps_kwargs` method to provide arguments to the `json.dumps()` method.

A simple mixin to handle very simple serialization as a response to the browser.

```
# views.py
from django.views.generic import DetailView

from braces.views import JSONResponseMixin

class UserProfileAJAXView(JSONResponseMixin, DetailView):
    model = Profile
    json_dumps_kwargs = {'indent': 2}

    def get(self, request, *args, **kwargs):
        self.object = self.get_object()

        context_dict = {
            'name': self.object.user.name,
            'location': self.object.location
        }

        return self.render_json_response(context_dict)
```

You can additionally use the *AjaxResponseMixin*

```
# views.py
from braces.views import AjaxResponseMixin

class UserProfileView(JSONResponseMixin, AjaxResponseMixin, DetailView):
    model = Profile

    def get_ajax(self, request, *args, **kwargs):
        return self.render_json_object_response(self.get_object())
```

The *JSONResponseMixin* provides a class-level variable to control the response type as well. By default it is *application/json*, but you can override that by providing the *content_type* variable a different value or, programmatically, by overriding the *get_content_type()* method.

```
from braces.views import JSONResponseMixin

class UserProfileAJAXView(JSONResponseMixin, DetailView):
    content_type = 'application/javascript'
    model = Profile

    def get(self, request, *args, **kwargs):
        self.object = self.get_object()

        context_dict = {
            'name': self.object.user.name,
            'location': self.object.location
        }

        return self.render_json_response(context_dict)

    def get_content_type(self):
        # Shown just for illustrative purposes
        return 'application/javascript'
```

3.5 JsonRequestResponseMixin

New in version 1.3.

A mixin that attempts to parse request as JSON. If request is properly formatted, the json is saved to self.request_json as a Python object. request_json will be None for imparsible requests.

To catch requests that aren't JSON-formatted, set the class attribute `require_json` to True.

Override the class attribute `error_response_dict` to customize the default error message.

It extends *JSONResponseMixin*, so those utilities are available as well.

Note: To allow public access to your view, you'll need to use the `csrf_exempt` decorator or *CsrfExemptMixin*.

```
from django.views.generic import View

from braces.views import CsrfExemptMixin, JsonRequestResponseMixin

class SomeView(CsrfExemptMixin, JsonRequestResponseMixin, View):
    require_json = True

    def post(self, request, *args, **kwargs):
        try:
            burrito = self.request_json['burrito']
            toppings = self.request_json['toppings']
        except:
            error_dict = {'message':
                'your order must include a burrito AND toppings'}
            return self.render_bad_request_response(error_dict)
        place_order(burrito, toppings)
        return self.render_json_response(
            {'message': 'Your order has been placed!'})
```

3.6 AjaxResponseMixin

A mixin to allow you to provide alternative methods for handling AJAX requests.

To control AJAX-specific behavior, override `get_ajax`, `post_ajax`, `put_ajax`, or `delete_ajax`. All four methods take `request`, `*args`, and `**kwargs` like the standard view methods.

```
# views.py
from django.views.generic import View

from braces.views import AjaxResponseMixin, JSONResponseMixin

class SomeView(JSONResponseMixin, AjaxResponseMixin, View):
    def get_ajax(self, request, *args, **kwargs):
        json_dict = {
            'name': "Benny's Burritos",
            'location': "New York, NY"
        }
        return self.render_json_response(json_dict)
```

3.7 OrderableListMixin

New in version 1.1.

A mixin to allow easy ordering of your queryset basing on the GET parameters. Works with *ListView*.

To use it, define columns that the data can be order by as well as the default column to order by in your view. This can be done either by simply setting the class attributes...

```
# views.py
class OrderableListView(OrderableListMixin, ListView):
    model = Article
    orderable_columns = ('id', 'title',)
    orderable_columns_default = 'id'
```

...or by using similarly name methods to set the ordering constraints more dynamically:

```
# views.py
class OrderableListView(OrderableListMixin, ListView):
    model = Article

    def get_orderable_columns(self):
        # return an iterable
        return ('id', 'title', )

    def get_orderable_columns_default(self):
        # return a string
        return 'id'
```

The `orderable_columns` restriction is here in order to stop your users from launching inefficient queries, like ordering by binary columns.

`OrderableListMixin` will order your queryset basing on following GET params:

- `order_by`: column name, e.g. `'title'`
- `ordering`: *'asc'* (default) or `'desc'`

Example url: *http://127.0.0.1:8000/articles/?order_by=title&ordering=asc*

3.8 CanonicalSlugDetailMixin

New in version 1.3.

A mixin that enforces a canonical slug in the url. Works with `DetailView`.

If a urlpattern takes a object's pk and slug as arguments and the slug url argument does not equal the object's canonical slug, this mixin will redirect to the url containing the canonical slug.

To use it, the urlpattern must accept both a `pk` and `slug` argument in its regex:

```
# urls.py
urlpatterns = patterns('',
    url(r'^article/(?P<pk>\d+)-(?P<slug>[-\w]+)$')
    ArticleView.as_view(),
    'view_article'
)
```

Then create a standard DetailView that inherits this mixin:

```
class ArticleView(CanonicalSlugDetailMixin, DetailView):
    model = Article
```

Now, given an Article object with `{pk: 1, slug: 'hello-world'}`, the url *http://127.0.0.1:8000/article/1-goodbye-moon* will redirect to *http://127.0.0.1:8000/article/1-hello-world* with the HTTP status code 301 Moved Permanently. Any other non-canonical slug, not just 'goodbye-moon', will trigger the redirect as well.

Control the canonical slug by either implementing the method `get_canonical_slug()` on the model class:

```
class Article(models.Model):
    blog = models.ForeignKey('Blog')
    slug = models.SlugField()

    def get_canonical_slug(self):
      return "{}-{}".format(self.blog.get_canonical_slug(), self.slug)
```

Or by overriding the `get_canonical_slug()` method on the view:

```
class ArticleView(CanonicalSlugDetailMixin, DetailView):
    model = Article

    def get_canonical_slug():
        import codecs
        return codecs.encode(self.get_object().slug, 'rot_13')
```

Given the same Article as before, this will generate urls of *http://127.0.0.1:8000/article/1-my-blog-hello-world* and *http://127.0.0.1:8000/article/1-uryyb-jbeyq*, respectively.

CHAPTER 4

Indices and tables

- *genindex*
- *modindex*
- *search*