django-braces Documentation

Release 1.1.0

Kenneth Love and Chris Jones

July 26, 2013

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

ONE

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`.

```
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({})
```

CHAPTER

TWO

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.

```
from django.views.generic import UpdateView

from braces.views import LoginRequiredMixin, CsrfExemptMixin

from profiles.models import Profile


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

CHAPTER

THREE

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"

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

CHAPTER

FOUR

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")
    }

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

CHAPTER

FIVE

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"

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

CHAPTER

SIX

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"

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

CHAPTER

SEVEN

USERFORMKWARGSMIXIN

In our clients 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. Example 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.

CHAPTER

EIGHT

USERKWARGMODELFORMMIXIN

The `UserKwargModelFormMixin` is a new form mixin we just implemented this week 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. Example 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"]
```

CHAPTER

NINE

SUCCESSURLREDIRECTLISTMIXIN

The `SuccessURLRedirectListMixin` is a bit more tailored to how we handle CRUD within our CMS. Our 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"
    ...
```

CHAPTER

TEN

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.

10.1 Static Example

```
from braces.views import SetHeadlineMixin


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

10.2 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.

CHAPTER

ELEVEN

CREATEANDREDIRECTTOEDITVIEW

Mostly used for CRUD, where you're going to create an object and then move direct to the update view for that object. Your URL for the update view has to accept a PK for the object. This `mixin` extends from *CreateView*.

> **Warning:** This mixin is pending deprecation and will be removed in a future release.

```
# urls.py
...
url(r"^users/create/$", UserCreateView.as_view(), name="cms_users_create"),
url(r"^users/edit/(?P<pk>\d+)/$", UserUpdateView.as_view(), name="cms_users_update"),
...

# views.py
from braces.views import CreateAndRedirectToEditView


class UserCreateView(CreateAndRedirectToEditView):
    model = User
    ...
```

CHAPTER

TWELVE

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"
```

CHAPTER

THIRTEEN

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"
```

CHAPTER

FOURTEEN

JSONRESPONSEMIXIN

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'
```

CHAPTER

FIFTEEN

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)
```

CHAPTER

SIXTEEN

ORDERABLELISTMIXIN

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

CHAPTER

SEVENTEEN

INDICES AND TABLES

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