django-braces Documentation

Release 1.11.0

Kenneth Love and Chris Jones

Apr 18, 2017

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 many of them extend the `AccessMixin`, the following are common attributes:

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

The `raise_exception` attribute allows for these scenarios, in case a permission is denied:

- `False` (default): redirects to the provided login view.
- `True`: raises a `PermissionDenied` exception.
- A subclass of `Exception`: raises this exception.
- A callable: gets called with the `request` argument. The function has to return a `HttpResponse` or `StreamingHttpResponse` (Django 1.5+), otherwise a `PermissionDenied` exception gets raised.

This gets done in `handle_no_permission`, which can be overridden itself.

Contents

LoginRequiredMixin

This mixin is rather simple and is generally the first inherited class in any view. If you 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. This mixin replicates the decorator's functionality.

> **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({})
```

An optional class attribute of `redirect_unauthenticated_users` can be set to `True` if you are using another `access` mixin with `raise_exception` set to `True`. This will redirect to the login page if the user is not authenticated, but raises an exception if they are but do not have the required access defined by the other mixins. This defaults

to `False`.

PermissionRequiredMixin

This mixin was originally written by Daniel Sokolowski (code here), but this version eliminates 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, use this mixin and set the `permission_required` class attribute on your 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 normal use of this mixin, *LoginRequiredMixin* comes first, then the `PermissionRequiredMixin`. If the user isn't an authenticated user, there is no point 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 django.views.generic import TemplateView

from braces import views


class SomeProtectedView(views.LoginRequiredMixin,
                        views.PermissionRequiredMixin,
                        TemplateView):

    permission_required = "auth.change_user"
    template_name = "path/to/template.html"
```

The `PermissionRequiredMixin` also offers a `check_permissions` method that should be overridden if you need custom permissions checking.

MultiplePermissionsRequiredMixin

The `MultiplePermissionsRequiredMixin` 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 or 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 a better choice.

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

```
from django.views.generic import TemplateView

from braces import views
```

```
class SomeProtectedView(views.LoginRequiredMixin,
                        views.MultiplePermissionsRequiredMixin,
                        TemplateView):

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

The `MultiplePermissionsRequiredMixin` also offers a `check_permissions` method that should be overridden if you need custom permissions checking.

GroupRequiredMixin

New in version 1.2.

The `GroupRequiredMixin` 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.

Standard Django Usage

```
from django.views.generic import TemplateView

from braces.views import GroupRequiredMixin


class SomeProtectedView(GroupRequiredMixin, TemplateView):

    #required
    group_required = u"editors"
```

Multiple Groups Possible Usage

```
from django.views.generic import TemplateView

from braces.views import GroupRequiredMixin


class SomeProtectedView(GroupRequiredMixin, TemplateView):

    #required
    group_required = [u"editors", u"admins"]
```

Custom Group Usage

```
from django.views.generic import TemplateView

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
```

Dynamically Build Groups

```
from django.views.generic import TemplateView

from braces.views import GroupRequiredMixin


class SomeProtectedView(GroupRequiredMixin, TemplateView):
    def get_group_required(self):
        # Get group or groups however you wish
        group = 'secret_group'
        return group
```

UserPassesTestMixin

New in version 1.3.0.

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 a specific domain).

```
from django.views.generic import TemplateView

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(u"mydomain.com"))
```

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 django.views.generic import TemplateView

from braces import views


class SomeSuperuserView(views.LoginRequiredMixin,
                        views.SuperuserRequiredMixin,
                        TemplateView):

    template_name = u"path/to/template.html"
```

AnonymousRequiredMixin

New in version 1.4.0.

Mixin that will redirect authenticated users to a different view. The default redirect is to Django's settings.LOGIN_REDIRECT_URL.

Static Examples

```
from django.views.generic import TemplateView

from braces.views import AnonymousRequiredMixin


class SomeView(AnonymousRequiredMixin, TemplateView):
    authenticated_redirect_url = u"/send/away/"
```

```
from django.core.urlresolvers import reverse_lazy
from django.views.generic import TemplateView

from braces.views import AnonymousRequiredMixin


class SomeLazyView(AnonymousRequiredMixin, TemplateView):
    authenticated_redirect_url = reverse_lazy(u"view_url")
```

Dynamic Example

```
from django.views.generic import TemplateView

from braces.views import AnonymousRequiredMixin


class SomeView(AnonymousRequiredMixin, TemplateView):
    """ Redirect based on user level """
```

```
    def get_authenticated_redirect_url(self):
        if self.request.user.is_superuser:
            return u"/admin/"
        return u"/somewhere/else/"
```

StaffuserRequiredMixin

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

```
from django.views.generic import TemplateView

from braces import views


class SomeStaffuserView(views.LoginRequiredMixin,
                        views.StaffuserRequiredMixin,
                        TemplateView):

    template_name = u"path/to/template.html"
```

SSLRequiredMixin

New in version 1.8.0.

Simple view mixin that requires the incoming request to be secure by checking Django's *request.is_secure()* method. By default the mixin will return a permanent (301) redirect to the https verison of the current url. Optionally you can set *raise_exception=True* and a 404 will be raised.

Standard Django Usage

```
from django.views.generic import TemplateView

from braces.views import SSLRequiredMixin


class SomeSecureView(SSLRequiredMixin, TemplateView):
    """ Redirects from http -> https """
    template_name = "path/to/template.html"
```

Standard Django Usage

```
from django.views.generic import TemplateView

from braces.views import SSLRequiredMixin


class SomeSecureView(SSLRequiredMixin, TemplateView):
    """ http request would raise 404. https renders view """
```

```
    raise_exception = True
    template_name = "path/to/template.html"
```

RecentLoginRequiredMixin

New in version 1.8.0.

This mixin requires a user to have logged in within a certain number of seconds. This is to prevent stale sessions or to create a session time-out, as is often used for financial applications and the like. This mixin includes the functionality of *LoginRequiredMixin*, so you don't need to use both on the same view.

```
from django.views.generic import TemplateView

from braces.views import RecentLoginRequiredMixin


class SomeSecretView(RecentLoginRequiredMixin, TemplateView):
    max_last_login_delta = 600  # Require a login within the last 10 minutes
    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

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 mixin should always be the left-most plugin.

```
from django.views.generic import UpdateView

from braces.views import LoginRequiredMixin, CsrfExemptMixin

from profiles.models import Profile


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

UserFormKwargsMixin

A common pattern in Django is to have forms that are customized to a user. To custom tailor the form for users, you have to pass that user instance into the form and, based on their permission level or other details, 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. The user can then be `pop()`ped off in the form. **Always** remember to pop the user from the kwargs before calling `super()` on your form, otherwise the form will get an unexpected keyword argument.

Usage

```
from django.views.generic import CreateView

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


class SomeSecretView(LoginRequiredMixin, UserFormKwargsMixin, CreateView):
    form_class = UserForm
    model = User
    template_name = "path/to/template.html"
```

This obviously pairs very nicely with the following mixin.

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, the `pop()`ping of the `user` is automated and no longer has to be done manually on every form that needs this behavior.

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"]
```

SuccessURLRedirectListMixin

The `SuccessURLRedirectListMixin` is a bit more tailored to how CRUD is often handled within CMSes. Many CMSes, by design, redirect 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, use this mixin and pass it a reversible route name. Example:

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

# views.py
from django.views import CreateView

from braces import views


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

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

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`

Static Example

```
from django.utils.translation import ugettext_lazy as _
from django.views.generic import CreateView

from braces.views import FormValidMessageMixin


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

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 u"{0} created!".format(self.object.title)
```

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`

Static Example

```
from django.utils.translation import ugettext_lazy
from django.views.generic import CreateView

from braces.views import FormInvalidMessageMixin


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

Dynamic Example

```
from django.utils.translation import ugettext_lazy as _
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 _(u"Some custom message")
```

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.

Static & Dynamic Example

```
from django.utils.translation import ugettext_lazy as _
from django.views.generic import CreateView

from braces.views import FormMessagesMixin


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

```
    def get_form_valid_message(self):
        return u"{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

SetHeadlineMixin

The `SetHeadlineMixin` allows you to *statically* or *programmatically* set the headline of any of your views. Ideally, you'll write as few templates as possible, so a mixin like this helps you 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:

Static Example

```
from django.utils.translation import ugettext_lazy as _
from django.views import TemplateView

from braces.views import SetHeadlineMixin


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

Dynamic Example

```
from datetime import date

from django.views import TemplateView

from braces.views import SetHeadlineMixin


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

    def get_headline(self):
        return u"This is our headline for {0}".format(date.today().isoformat())
```

For both usages, the context now contains a `headline` key with your headline.

StaticContextMixin

New in version 1.4.

The `StaticContextMixin` allows you to easily set static context data by using the `static_context` property.

Note: While it's possible to override the `StaticContextMixin.get_static_context method`, it's not very practical. If you have a need to override a method for dynamic context data it's best to override the standard `get_context_data` method of Django's generic class-based views.

View Example

```
# views.py

from django.views import TemplateView

from braces.views import StaticContextMixin


class ContextTemplateView(StaticContextMixin, TemplateView):
    static_context = {u"nav_home": True}
```

URL Example

```
# urls.py

urlpatterns = patterns(
    '',
    url(ur"^$",
        ContextTemplateView.as_view(
            template_name=u"index.html",
            static_context={u"nav_home": True}
        ),
        name=u"index")
)
```

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 = [u"user"]
    template_name = u"profiles/detail.html"
```

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 = [u"post_set"]  # where the Post model has an FK to the User␣
↪model as an author.
    template_name = u"users/detail.html"
```

JSONResponseMixin

Changed in version 1.1: `render_json_response` now accepts a `status` 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 = {u"indent": 2}

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

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

        return self.render_json_response(context_dict)
```

You can additionally use the *AjaxResponseMixin*

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

from braces import views


class UserProfileView(views.JSONResponseMixin,
                      views.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 django.views import DetailView

from braces.views import JSONResponseMixin


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

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

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

        return self.render_json_response(context_dict)

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

The *JSONResponseMixin* provides another class-level variable *json_encoder_class* to use a custom json encoder with *json.dumps*. By default it is *django.core.serializers.json.DjangoJsonEncoder*

```
from django.core.serializers.json import DjangoJSONEncoder

from braces.views import JSONResponseMixin


class SetJSONEncoder(DjangoJSONEncoder):
    """
    A custom JSONEncoder extending `DjangoJSONEncoder` to handle serialization
    of `set`.
    """
    def default(self, obj):
        if isinstance(obj, set):
            return list(obj)
        return super(DjangoJSONEncoder, self).default(obj)


class GetSetDataView(JSONResponseMixin, View):
    json_encoder_class = SetJSONEncoder

    def get(self, request, *args, **kwargs):
        numbers_set = set(range(10))
        data = {'numbers': numbers_set}
        return self.render_json_response(data)
```

JsonRequestResponseMixin

New in version 1.3.

A mixin that attempts to parse the request as JSON. If the 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 import views

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

    def post(self, request, *args, **kwargs):
        try:
            burrito = self.request_json[u"burrito"]
            toppings = self.request_json[u"toppings"]
        except KeyError:
            error_dict = {u"message":
               u"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(
            {u"message": u"Your order has been placed!"})
```

AjaxResponseMixin

This mixin provides hooks for altenate processing of AJAX requests based on HTTP verb.

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 import views

class SomeView(views.JSONResponseMixin, views.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)
```

Note: This mixin is only useful if you need to have behavior in your view fork based on `request.is_ajax()`.

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 ordered 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
from django.views import ListView

from braces.views import OrderableListMixin


class OrderableListView(OrderableListMixin, ListView):
    model = Article
    orderable_columns = (u"id", u"title",)
    orderable_columns_default = u"id"
```

Or by using similarly-named methods to set the ordering constraints more dynamically:

```
# views.py
from django.views import ListView

from braces.views import OrderableListMixin


class OrderableListView(OrderableListMixin, ListView):
    model = Article

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

    def get_orderable_columns_default(self):
        # return a string
        return u"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*

You can also override the default ordering from `"asc"` to `"desc"` by setting the `"ordering_default"` in your view class.

```
# views.py
from django.views import ListView

from braces.views import OrderableListMixin


class OrderableListView(OrderableListMixin, ListView):
    model = Article
    orderable_columns = (u"id", u"title",)
    orderable_columns_default = u"id"
    ordering_default = u"desc"
```

This will reverse the order of list objects if no query param is given.

Front-end Example Usage

If you're using bootstrap you could create a template like the following:

```
<div class="table-responsive">
    <table class="table table-striped table-bordered">
        <tr>
            <th><a class="order-by-column" data-column="id" href="#">ID</a></th>
            <th><a class="order-by-column" data-column="title" href="#">Title</a></th>
        </tr>
        {% for object in object_list %}
            <tr>
                <td>{{ object.id }}</td>
                <td>{{ object.title }}</td>
            </tr>
        {% endfor %}
    </table>
</div>

<script>
function setupOrderedColumns(order_by, orderin) {

    $('.order-by-column').each(function() {

        var $el = $(this),
            column_name = $el.data('column'),
            href = location.href,
            next_order = 'asc',
            has_query_string = (href.indexOf('?') !== -1),
            order_by_param,
            ordering_param;

        if (order_by === column_name) {
            $el.addClass('current');
            $el.addClass(ordering);
            $el.append('<span class="caret"></span>');
            if (ordering === 'asc') {
                $el.addClass('dropup');
                next_order = 'desc';
            }
        }

        order_by_param = "order_by=" + column_name;
        ordering_param = "ordering=" + next_order;
```

```
        if (!has_query_string) {
            href = '?' + order_by_param + '&' + ordering_param;
        } else {
            if (href.match(/ordering=(asc|desc)/)) {
                href = href.replace(/ordering=(asc|desc)/, ordering_param);
            } else {
                href += '&' + ordering_param;
            }

            if (href.match(/order_by=[_\w]+/)) {
                href = href.replace(/order_by=([_\w]+)/, order_by_param);
            } else {
                href += '&' + order_by_param;
            }

        }

        $el.attr('href', href);

    });
}
setupOrderedColumns('{{ order_by }}', '{{ ordering }}');
</script>
```

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 "{0}-{1}".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.

MessageMixin

New in version 1.4.

A mixin that adds a `messages` attribute on the view which acts as a wrapper to `django.contrib.messages` and passes the `request` object automatically.

> **Warning:** If you're using Django 1.4, then the `message` attribute is only available after the base view's `dispatch` method has been called (so our second example would not work for instance).

```
from django.views.generic import TemplateView

from braces.views import MessageMixin


class MyView(MessageMixin, TemplateView):
    """
    This view will add a debug message which can then be displayed
    in the template.
    """
    template_name = "my_template.html"

    def get(self, request, *args, **kwargs):
        self.messages.debug("This is a debug message.")
        return super(MyView, self).get(request, *args, **kwargs)
```

```
from django.contrib import messages
from django.views.generic import TemplateView

from braces.views import MessageMixin


class OnlyWarningView(MessageMixin, TemplateView):
    """
    This view will only show messages that have a level
    above `warning`.
```

```
    """
    template_name = "my_template.html"

    def dispatch(self, request, *args, **kwargs):
        self.messages.set_level(messages.WARNING)
        return super(OnlyWarningView, self).dispatch(request, *args, **kwargs)
```

AllVerbsMixin

New in version 1.4.

This mixin allows you to specify a single method that will response to all HTTP verbs, making a class-based view behave much like a function-based view.

```
from django.views import TemplateView

from braces.views import AllVerbsMixin


class JustShowItView(AllVerbsMixin, TemplateView):
    template_name = "just/show_it.html"

    def all(self, request, *args, **kwargs):
        return super(JustShowItView, self).get(request, *args, **kwargs)
```

If you need to change the name of the method called, provide a new value to the `all_handler` attribute (default is `'all'`)

HeaderMixin

New in version 1.11.

This mixin allows you to add arbitrary HTTP header to a response. Static headers can be defined in the `headers` attribute of the view.

```
from django.views import TemplateView

from braces.views import HeaderMixin


class StaticHeadersView(HeaderMixin, TemplateView):
    template_name = "some/headers.html"
    headers = {
        'X-Header-Sample': 'some value',
        'X-Some-Number': 42
    }
```

If you need to set the headers dynamically, e.g depending on some request information, override the `get_headers` method instead.

```
from django.views import TemplateView

from braces.views import HeaderMixin
```

```
class EchoHeadersView(HeaderMixin, TemplateView):
    template_name = "some/headers.html"

    def get_headers(self, request):
        """
        Echo back request headers with ``X-Request-`` prefix.
        """
        for key, value in request.META.items():
            yield "X-Request-{}".format(key), value
```

View our Changelog

Want to contribute?

CHAPTER 4

Indices and tables

- genindex
- modindex
- search