phileo Documentation

Release 1.3.1

Pinax

August 19, 2015

Contents

A liking app

CHAPTER 1

Development

The source repository can be found at https://github.com/pinax/phileo

1.1 Contents

1.1.1 ChangeLog

1.2

- `like_text_off` and `css_class_off` are passed into widget even if `can_like` is False.
- `PHILEO_LIKABLE_MODELS` entries now take an optional extra value `allowed` whose value should be a callable taking `user` and `obj` and returning True or False depending on whether the user is allowed to like that particular object

1.1.1

- Fixed regression causing error when widget displayed while unauth'd

1.1

- Fixed urls.py deprecation warnings
- Fixed unicode string
- Added support for custom User models
- Documentation updates

1.0

- Added an admin.py

0.6

- Added a *phileo_widget_brief* to display a brief widget template (*phileo/_widget_brief.html*)

0.5

- Added a *who_likes* template tag that returns a list of *Like* objects for given object

0.4.1

- Made the link in the default widget template a bootstrap button

0.4

- Fixed isinstance check to check `models.Model` instead of `models.base.ModelBase`
- Added permission checking
- Added rendering of HTML in the ajax response to liking
- Got rid of all the js/css cruft; up to site owner now but ships with bootstrap/bootstrap-ajax enabled templates
- Updated use of `datetime.datetime.now` to `timezone.now`

Backward Incompatibilities

- Added an auth_backend to check permissions, you can just add the *phileo.auth_backends.PermCheckBackend* and do nothing else, or you can implement you own backend checking the *phileo.can_like* permission against the object and user according to your own business logic.
- No more `phileo_css`, `phileo_js`, or `phileo_widget_js` tags.
- `PHILEO_LIKABLE_MODELS` has changed from a `list` to a `dict`
- `phileo_widget` optional parameters have been removed and instead put into per model settings

0.3

- Renamed *likes_css* and *likes_widget* to *phileo_css* and *phileo_widget*
- Turned the JavaScript code in to a jQuery plugin, removed most of the initialization code from the individual widget templates to a external JavaScript file, and added a {% phileo_js %} tag to load this plugin.
- Each like button gets a unique ID, so multiple like buttons can appear on a single page
- The like form works without JavaScript.
- Likeable models need to be added to *PHILEO_LIKABLE_MODELS* setting. This prevents users from liking anything and everything, which could potentially lead to security problems (eg. liking entries in permission tables, and thus seeing their content; liking administrative users and thus getting their username).
- Added request objects to both *object_liked* and *object_unliked* signals.

Backward Incompatibilities

- pretty much all the template tags have been renamed and work slightly differently

0.2

- made it easier to get rolling with a like widget using default markup and javascript
- added returning the like counts for an object when it is liked or unliked so that the widget (either your own or using the one that ships with phileo) can update via AJAX

Backward Incompatibilities

- removed *likes_ajax* and *likes_form* template tags so if you were using them and had written custom overrides in _ajax.js and _form.html you'll need to plan your upgrade accordingly.
- changed the url pattern, *phileo_like_toggle*, for likes to not require the user pk, instead, the view handling the POST to this url, uses request.user.
- changed the ajax returned by the *like_toggle* view so that it now just returns a single element: {"likes_count": <some-number>}

0.1

- initial release

1.1.2 Requirements

The view to handle the like toggling conforms to an ajax response that eldarion-ajax understands. Furthermore, the templates that ship with this project will work seemlessly with eldarion-ajax. All you have to do is include the eldarion-ajax in your base template somewhere like:

```
{% load staticfiles %}
<script src="{% static "js/eldarion-ajax.min.js" %}"></script>
```

This of course is optional. You can roll your own javascript handling as the view also returns data in addition to rendered HTML. Furthermore, if you don't want ajax at all the view will handle a regular POST and perform a redirect.

1.1.3 Installation

- To install phileo:

```
pip install phileo
```

- Add `'phileo'` to your `INSTALLED_APPS` setting:

```
INSTALLED_APPS = (
    # other apps
    "phileo",
)
```

- Add the models that you want to be likeable to `PHILEO_LIKABLE_MODELS`:

```
PHILEO_LIKABLE_MODELS = {
    "app.Model": {}  # can override default config settings for each model here
}
```

- Add 'phileo.auth_backends.CanLikeBackend' to your AUTHENTICATION_BACKENDS (or use your own custom version checking against the phileo.can_like permission):

```
AUTHENTICATION_BACKENDS = [
  ...
  "phileo.auth_backends.CanLikeBackend",
  ...
]
```

- Lastly you will want to add *phileo.urls* to your urls definition:

```
...
url(r"^likes/", include("phileo.urls")),
...
```

1.1.4 Filters

likes_count

This simple returns the count of likes for a given object:

```
{{ obj|likes_count }}
```

1.1.5 Template Tags

who_likes

An assignment tag that fetches a list of likes for a given object:

```
{% who_likes car as car_likes %}

{% for like in car_likes %}
    <div class="like">{{ like.sender.get_full_name }} likes {{ car }}</div>
{% endfor %}
```

render_like

This renders a like, so that you can provide a list of likes. It combines well with *likes*.:

```
{% likes user as like_list %}
<ul>
    {% for like in like_list %}
        <li>{% render_like like %}</li>
    {% endfor %}
</ul>
```

The *render_like* tag looks in the following places for the template to render. Any of them can be overwritten as needed, allowing you to customize the rendering of the like on a per model and per application basis:

- phileo/app_name/model.html
- phileo/app_name/like.html
- phileo/_like.html

phileo_widget

This renders a fragment of html that will be what the user will click on to unlike or like objects. It only has two required parameters, which are the user and the object.:

```
{% phileo_widget user object %}
```

It renders "phileo/_widget.html" and can be overridden as desired.

phileo_widget_brief

Same, functionally, as *phileo_widget*, except that it renders "phileo/_widget_brief.html" instead.

liked

The "liked" template tag will decorate an iterable of objects given a particular user, with a "liked" boolean indicating whether or not the user likes each object in the iterable:

```
{% liked objects by request.user as varname %}
{% for obj in varname %
    <div>{% if obj.liked %}* {% endif %}{{ obj.title }}</div>
{% endfor %}
```

likes

The "likes" tag will fetch into a context variable a list of objects that the given user likes:

```
{% likes request.user "app.Model" as objs %}
{% for obj in objs %}
    <div>{{ obj }}</div>
{% endfor %}
```

1.1.6 Signals

Both of these signals are sent from the Like model in the view that processes the toggling of likes and unlikes.

phileo.signals.object_liked

This signal is sent immediately after the object is liked and provides the single kwarg of "like" which is the instance of the Like object that was created.

phileo.signals.object_unliked

This signal is sent immediately after the object is unliked and provides the single kwarg of "object" which is the objects that was just unliked.

1.1.7 Usage

In your settings

You need to add each model that you want to be likable to the *PHILEO_LIKABLE_MODELS* setting:

```
PHILEO_LIKABLE_MODELS = {
    "profiles.Profile": {},
    "videos.Video": {},
    "biblion.Post": {},
}
```

In the templates

Let's say you have a detail page for a blog post. First you will want to load the tags:

```
{% load phileo_tags %}
```

In the body where you want the liking widget to go, add:

```
{% phileo_widget request.user post %}
```

That's all you need to do to get the basics working.

1.1.8 AJAX

It's quite common to have this snippet already included in a site and there are a few different variations, but to avoid exempting CSRF checks for the POST, you'll want something like this included in your base template.

If you don't have this already (or something similar), considering creating a `ajax.js` file with the following contents:

```
$(document).ajaxSend(function(event, xhr, settings) {
    function getCookie(name) {
        var cookieValue = null;
        if (document.cookie && document.cookie != '') {
            var cookies = document.cookie.split(';');
            for (var i = 0; i < cookies.length; i++) {
                var cookie = jQuery.trim(cookies[i]);
                // Does this cookie string begin with the name we want?
                if (cookie.substring(0, name.length + 1) == (name + '=')) {
                    cookieValue = decodeURIComponent(cookie.substring(name.length + 1));
                    break;
                }
            }
        }
        return cookieValue;
    }
    function sameOrigin(url) {
        // url could be relative or scheme relative or absolute
        var host = document.location.host; // host + port
        var protocol = document.location.protocol;
        var sr_origin = '//' + host;
        var origin = protocol + sr_origin;
        // Allow absolute or scheme relative URLs to same origin
        return (url == origin || url.slice(0, origin.length + 1) == origin + '/') ||
            (url == sr_origin || url.slice(0, sr_origin.length + 1) == sr_origin + '/') ||
            // or any other URL that isn't scheme relative or absolute i.e relative.
```

```
            !(/^(\/\/|http:|https:).*/.test(url));
    }
    function safeMethod(method) {
        return (/^(GET|HEAD|OPTIONS|TRACE)$/.test(method));
    }

    if (!safeMethod(settings.type) && sameOrigin(settings.url)) {
        xhr.setRequestHeader("X-CSRFToken", getCookie('csrftoken'));
    }
});
```

And including it:

```
<script src="{% static "js/ajax.js" %}"></script>
```