django-popupcrud Documentation

Release 0.12.0

Hari Mahadevan

Jun 16, 2020

Contents:

1 Quickstart **1**

2 Reference **5**
- 2.1 Classes . . . 5
- 2.2 Template Tags . . . 13

3 How-tos **15**
- 3.1 Create CRUD views for a model . . . 15
- 3.2 Control access using permissions . . . 16
- 3.3 Create a model object from its FK select box in another form . . . 17
- 3.4 Use Select2 instead of native Select widget . . . 17
- 3.5 Providing your own templates . . . 18
- 3.6 Use the formset feature . . . 19

4 Demo Project **21**

5 Settings **23**

Python Module Index **25**

Index **27**

CHAPTER 1

Quickstart

1. Install `django-popupcrud` using pip:

   ```
   pip install django-popucrud
   ```

   Or install it directly from the source repository:

   ```
   pip intall git+https://github.com/harikvpy/django-popupcrud.git
   ```

   Yet another way would be to clone this repository and install from the cloned root folder via `pip install -e .`.

2. Install the dependencies - `django-bootstrap3` and `django-pure-pagination`. Add the dependencies and `popupcrud` to `INSTALLED_APPS` in your project's `settings.py`:

   ```
   INSTALLED_APPS = [
       ...
       'bootstrap3',
       'pure_pagination',
       'popupcrud',
       ...
   ]
   ```

3. Let `PopupCrudViewSet` know of your base template file name. This defaults to `base.html`, but if your project uses a different base template filename, inform `PopupCrudViewSet` about it in `settings.py`:

   ```
   POPUPCRUD = {
       'base_template': 'mybase.html',
   }
   ```

   Include Bootstrap CSS & JS resources in this base template. If you were to use `django-bootstrap3` tags for these, your base template should look something like this:

   ```
   <head>
       {% bootstrap_css %}
       <script src="{% bootstrap_jquery_url %}" type="text/javascript" charset="utf-8
   ↪"></script>
   ```

(continues on next page)

(continued from previous page)

```
    {% bootstrap_javascript %}
    {% block extrahead %}{% endblock extrahead %}
</head>
```

Also, define a block named `extrahead` within the `<head>` element. `PopupCrudViewSet` views use a few custom CSS styles to show column sorting options and sort priority. These styles are defined in `static/popupcrud/css/popupcrud.css` which is inserted into the `extrahead` block. If you don't declare this block, you will have to explicitly load the stylesheet into your base template.

4. In your app's `views.py`, create a `ViewSet` for each model for which you want to support CRUD operations.

   Models.py:

```
from django.db import models

class Author(models.Model):
    name = models.CharField("Name", max_length=128)
    penname = models.CharField("Pen Name", max_length=128)
    age = models.SmallIntegerField("Age", null=True, blank=True)

    class Meta:
        ordering = ('name',)
        verbose_name = "Author"
        verbose_name_plural = "Authors"

    def __str__(self):
        return self.name
```

   Views.py:

```
from django.core.urlresolvers import reverse_lazy, reverse
from popupcrud.views import PopupCrudViewSet

class AuthorViewSet(PopupCrudViewSet):
    model = Author
    fields = ('name', 'penname', 'age')
    list_display = ('name', 'penname', 'age')
    list_url = reverse_lazy("library:authors:list")
    new_url = reverse_lazy("library:authors:create")

    def get_edit_url(self, obj):
        return reverse("library:authors:update", kwargs={'pk': obj.pk})

    def get_delete_url(self, obj):
        return reverse("library:authos:delete", kwargs={'pk': obj.pk})
```

5. Wire up the CRUD views generated by the viewset to the URLconf:

```
urlpatterns= [
    url(r'^authors/', views.AuthorCrudViewset.urls()),
]
```

   This will register the following urls:

   - `authors/` - list view
   - `authors/create/` - create view
   - `authors/<pk>/` - detail view

- `authors/<pk>/update/` - update view
- `authors/<pk>/delete/` - delete view

The urls are registered under its own namespace, which defaults to the model's `verbose_name_plural` meta value.

6. Thats it! Your modern HTML popup based CRUD for your table is up and running.

   PopupCrudViewSet has many options to customize the fields displayed in list view, form used for create/update operations, permission control and more. Refer to the Reference and How-to sections of the documentation for more details.

CHAPTER 2

Reference

2.1 Classes

2.1.1 PopupCrudViewSet

class popupcrud.views.**PopupCrudViewSet**(**args*, ***kwargs*)

This is the base class from which you derive a class in your project for each model that you need to build CRUD views for.

model = None

The model to build CRUD views for. This is a required attribute.

new_url = None

URL to the create view for creating a new object. This is a required attribute.

list_display = ()

Lists the fields to be displayed in the list view columns. This attribute is modelled after ModelAdmin.list_display and supports model methods as as ViewSet methods much like ModelAdmin. This is a required attribute.

So you have four possible values that can be used in list_display:

- A field of the model
- A callable that accepts one parameter for the model instance.
- A string representing an attribute on ViewSet class.
- A string representing an attribute on the model

See ModelAdmin.list_display documentation for examples.

A note about `list_display` fields with respect to how it differs from `ModelAdmin`'s `list_display`.

In `ModelAdmin`, if a field specified in `list_display` is not a database field, it can be set as a sortable field by setting the method's `admin_order_field` attribute to the relevant database field that can be used as the sort field. In `PopupCrudViewSet`, this attribute is named `order_Field`.

`fields = ()`
A list of names of fields. This is interpreted the same as the Meta.fields attribute of ModelForm. This is a required attribute.

`form_class = None`
The form class to instantiate for Create and Update views. This is optional and if not specified a ModelForm using the values of fields attribute will be instantiated. An optional attribute, if specified, overrides fields attribute value.

`list_url = None`
The url where the list view is rooted. This will be used as the success_url attribute value for the individual CRUD views. This is a required attribute.

`paginate_by = 10`
Number of entries per page in list view. Defaults to 10. Setting this to None will disable pagination. This is an optional attribute.

`list_permission_required = ()`
List of permission names for the list view. Permission names are of the same format as what is specified in `permission_required()` decorator. Defaults to no permissions, meaning no permission is required.

Depracated. Use *permissions_required* dictionary instead.

`create_permission_required = ()`
List of permission names for the create view. Defaults to no permissions, meaning no permission is required.

Depracated. Use *permissions_required* dictionary instead.

`detail_permission_required = ()`
List of permission names for the detail view. Defaults to no permissions, meaning no permission is required.

Depracated. Use *permissions_required* dictionary instead.

`update_permission_required = ()`
List of permission names for the update view. Defaults to no permissions, meaning no permission is required.

Depracated. Use *permissions_required* dictionary instead.

`delete_permission_required = ()`
List of permission names for the delete view. Defaults to no permissions, meaning no permission is required.

Depracated. Use *permissions_required* dictionary instead.

`permissions_required = {}`
Permissions table for the various CRUD views. Use this instead of list_permission_required, create_permission_required, etc. Entries in this table are indexed by the CRUD view code and its required permissions tuple. CRUD view codes are: 'list', 'create', 'detail', 'update' & 'delete'.

Note that if both the legacy *<crud-op>_permission_required* and *permissons_required* are specified, *permissions_required* setting value takes effect.

For example you can specify:

```
permissions_required = {
    'list': ('library.list_author',),
    'create': ('library.list_author',),
    'detail': ('library.view_author',),
    'update': ('library.update_author',),
    'delete': ('library.delete_author',)
}
```

list_template = None
The template file to use for list view. If not specified, defaults to the internal template.

related_object_popups = {}
A table that maps foreign keys to its target model's `PopupCrudViewSet.create()` view url. This would result in the select box for the foreign key to display a 'New {model}' link at its bottom, which the user can click to add a new {model} object from another popup. The newly created {model} object will be added to the select's options and set as its selected option.

Defaults to empty dict, meaning creation of target model objects, for the foreign keys of a model, from a popup is disabled.

page_title = ''
Page title for the list view page.

legacy_crud = False
Enables legacy CRUD views where each of the Create, Detail, Update & Delete views are performed from their own dedicated web views like Django admin (hence the term `legacy_crud` :-)).

This property can accept either a boolean value, which in turn enables/ disables the legacy mode for all the CRUD views or it can accept a dict of CRUD operation codes and its corresponding legacy mode specified as boolean value.

This dict looks like:

```
legacy_crud = {
    'create': False,
    'detail': False,
    'update': False,
    'delete': False
}
```

So by setting `legacy_crud[detail] = True`, you can enable legacy style crud for the detail view whereas the rest of the CRUD operations are performed from a modal popup.

In other words, `legacy_crud` boolean value results in a dict that consists of `True` or `False` values for all its keys, as the case may be.

This defaults to `False`, which translates into a dict consisting of `False` values for all its keys.

login_url = None
Same as `django.contrib.auth.mixins.AccessMixin login_url`, but applicable for all CRUD views.

raise_exception = False
Same as `django.contrib.auth.mixins.AccessMixin raise_exception`, but applicable for all CRUD views.

empty_list_icon = None
Icon to be displayed above the empty list state message. Defaults to None, which displays no icon. To specify an icon, set this property to the CSS class of the required icon.

For example to use the glyphicon-book icon, set this property to:

```
empty_list_icon = 'glyphicon glyphicon-book'
```

Icons displayed are enlarged to 5 times the standard font size.

empty_list_message = 'No records found.'
Message to be displayed when list view contains no records, that is, empty list state. Defaults to 'No records found'.

Empty list state rendering can be customized further by overriding `popupcrud/empty_list.html` template in your own project.

breadcrumbs = []
List of breadcrumbs that will be added to ViewSet views' context, allowing you build a breadcrumb hierarchy that reflects the ViewSet's location in the site.

Note that for `legacy_crud` views, system would add the `list view` url to the breadcrumbs list.

breadcrumbs_context_variable = 'breadcrumbs'
The template context variable name that will be initialized with the value of `breadcrumbs` property. You can enumerate this variable in your base template to build a breadcrumbs list that reflects the hierarchy of the page.

item_actions = []

- **its title**
  - its icon css such as `glyphicon glyphicon-ok`
  - its action handler, which is the name of the CrudViewSet method to be called when user selects the action. This method has the following signature:

```
def action_handler(self, request, item):
    # action processing

    return (True, "Action completed")
```

The return value from the action handler is a 2-tuple that consists of a boolean success indicator and a message. The message is displayed to the user when the action is completed.

Also see `get_item_actions()` documentation below.

modal_sizes = {}
Allows specifying the size of the modal windows used for the CRUD operations. Value is a dictionary, where the key names indicate the modal whose size you want to adjust. Allowed values for these are: `create_update`, `delete` & `detail`. Value for the keys indicate the size of the modal and has to be one of: `{small|normal|large}`.

Defaults to:

```
modal_sizes = {
    'create_update': 'normal',
    'delete': 'normal',
    'detail': 'normal'
}
```

You only need to specify the modal whose size you want to adjust. So if you want to adjust the size of the `create_update` modal to large while leaving the rest to their default size, you may specify modal_sizes as:

```
modal_sizes = {
    'create_update': 'large',
}
```

context_object_name = None
If specified, the name of the context variable that will be used to assign the object instance value. By default the context variable `object` will be assigned the object instance. If `context_object_name` is specified, that too will be assigned the object instance.

pk_url_kwarg = 'pk'
Same as SingleObjectMixin.pk_url_kwarg, which is the name of the URLConf keyword argument that contains the primary key. By default, `pk_url_kwarg` is *pk*'.

slug_field = 'slug'
Same as SingleObjectMixin.slug_field, which is the name of the field on the model that contains the slug. By default, `slug_field` is `slug`.

To use `slug_field` as the key to access object instances (for detail, update & delete views), set `pk_url_kwarg = None` in the ViewSet class and initialize `slug_field` and `slug_url_kwarg` to the relevant slug field's name & its corresponding URLconf parameter name respectively.

slug_url_kwarg = 'slug'
Same as SingleObjectMixin.slug_url_kwarg, which is the name of the URLConf keyword argument that contains the slug. By default, `slug_url_kwarg` is `slug`.

classmethod list(***initkwargs*)
Returns the list view that can be specified as the second argument to url() in urls.py.

classmethod create(***initkwargs*)
Returns the create view that can be specified as the second argument to url() in urls.py.

classmethod detail(***initkwargs*)
Returns the create view that can be specified as the second argument to url() in urls.py.

classmethod update(***initkwargs*)
Returns the update view that can be specified as the second argument to url() in urls.py.

classmethod delete(***initkwargs*)
Returns the delete view that can be specified as the second argument to url() in urls.py.

get_new_url()
Returns the URL to create a new model object. Returning None would disable the new object creation feature and will hide the `New {model}` button.

You may override this to dynamically determine if new object creation ought to be allowed. Default implementation returns the value of `ViewSet.new_url`.

get_detail_url(*obj*)
Override this returning the URL where `PopupCrudViewSet.detail()` is placed in the URL namespace such that ViewSet can generate the appropriate href to display item detail in list view.

When this hyperlink is clicked, a popup containing the object's detail will be shown. By default this popup only shows the object's string representation. To show additional information in this popup, implement `<object>_detail.html` in your project, typically in the app's template folder. If this file exists, it will be used to render the object detail popup. True to Django's `DetailView` convention, you may use the `{{ object }}` template variable in the template file to access the object and its properties.

Default implementations returns None, which results in object detail popup being disabled.

get_edit_url(*obj*)
Override this returning the URL where PopupCrudViewSet.update() is placed in the URL namespace such that ViewSet can generate the appropriate href to the item edit hyperlink in list view.

If None is returned, link to edit the specified item won't be shown in the object row.

get_delete_url(*obj*)
Override this returning the URL where PopupCrudViewSet.delete() is placed in the URL namespace such that ViewSet can generate the appropriate href to the item delete hyperlink in list view.

If None is returned, link to delete the specified item won't be shown in the object row.

get_obj_name(*obj*)
Return the name of the object that will be displayed in item action prompts for confirmation. Defaults to `str(obj)`, ie., the string representation of the object. Override this to provide the user with additional object details. The returned string may contain embedded HTML tags.

For example, you might want to display the balance due from a customer when confirming user action to delete the customer record.

get_permission_required(*op*)
Return the permission required for the CRUD operation specified in op. Default implementation returns the value of one `{list|create|detail|update|delete}_permission_required` class attributes. Overriding this allows you to return dynamically computed permissions.

Parameters **op** – The CRUD operation code. One of `{'list'|'create'|'detail'|'update'|'delete'}`.

Return type

The `permission_required` tuple for the specified operation. Determined by looking up the given `op` from the table:

```
permission_table = {
    'list': self.list_permission_required,
    'create': self.create_permission_required,
    'detail': self.detail_permission_required,
    'update': self.update_permission_required,
    'delete': self.delete_permission_required
}
```

get_page_title(*view*, *obj=None*)
Returns page title for the CRUD view. Parameter *view* specifies the CRUD view whose page title is being queried and is one of *create*, *detail*, *update*, *delete* or *list*.

For *detail*, *update* & *delete* views, the model instance is is passed as second argument. For the rest of the views, this is set to *None*.

classmethod urls(*namespace=None*, *views=('create', 'update', 'delete', 'detail')*)
Returns the CRUD urls for the viewset that can be added to the URLconf. The URLs returned can be controlled by the `views` parameter which is tuple of strings specifying the CRUD operations URLs to be returned. This defaults to all the CRUD operations: create, read(detail), update & delete (List view URL is added by default).

This method can be seen as a wrapper to calling the individual view generator methods, `list()`, `detail()`, `create()`, `update()` & `delete()`, to register them with the URLconf.

The urls for CRUD actions involving a single object (detail, update & delete) are by default composed using `pk` as URLConf keyword argument. However, if `pk_url_kwarg` is set to None and `slug_field` and `slug_url_kwarg` are initialized, it will be based as the field used to locate the individual object and URLConf keyword argument respectively.

Parameters

- **namespace** – The namespace under which the CRUD urls are registered. Defaults to the value of `<model>.Meta.verbose_name_plural` (in lowercase and in English).
- **views** – A tuple of strings representing the CRUD views whose URL patterns are to be registered. Defaults to `('create', 'update', 'delete', 'detail')`, that is all the CRUD operations for the model.

Return type A collection of URLs, packaged using `django.conf.urls.include()`, that can be used as argument 2 to `url()` (see example below).

Example The following pattern registers all the CRUD urls for model Book (in app `library`), generated by BooksCrudViewSet:

```
urlpatterns += [
    url(r'^books/', BooksCrudViewSet.urls())
]
```

This allows us to refer to individual CRUD operation url as:

```
reverse("library:books:list")
reverse("library:books:create")
reverse("library:books:detail", kwargs={'pk': book.pk})
reverse("library:books:update", kwargs={'pk': book.pk})
reverse("library:books:delete", kwargs={'pk': book.pk})
```

popups
Provides a normalized dict of crud view types to use for the viewset depending on client.legacy_crud setting.

Computes this dict only once per object as an optimization.

get_empty_list_icon()
Determine the icon used to display empty list state.

Returns the value of `empty_list_icon` property by default.

get_empty_list_message()
Determine the message used to display empty table state.

Returns the value of `empty_list_message` property by default.

get_breadcrumbs()
Returns the value of `ViewSet.breadcrumbs` property. You can use this method to return breadcrumbs that contain runtime computed values.

get_queryset(*qs*)
Called by ListView allowing ViewSet to do further filtering of the queryset, if necessary. By default returns the queryset argument unchanged.

Parameters **qs** – Queryset that is used for rendering ListView content.

Return type A valid Django queryset.

get_form_kwargs()
For Create and Update views, this method allows passing custom arguments to the form class constructor. The return value from this method is combined with the default form constructor `**kwargs` before it is passed to the form class' `__init__()` routine's `**kwargs`.

Since Django CBVs use kwargs `initial` & `instance`, be careful when using these, unless of course, you want to override the objects provided by these keys.

get_formset_class()
Return the formset class to the child model of this parent model allowing create/edit of multiple objects of the child model.

Typically, though not required, the return value will be a class generated by the django formset factory function inlineformset_factory().

By default, this method returns None, which indicates that the model Create/Edit forms do not have a formset for a child model.

get_formset()
Returns the formset object instantiated from the class returned by the get_formset_class() method.

By default returns None indicating no formset is associated with the model.

get_item_actions(*obj*)
Determine the custom actions for the given model object that is displayed after the standard Edit & Delete actions in list view.

Parameters **obj** – The row object for which actions are being queried.

Return type A list of action 3-tuple (as explained in `item_actions`) objects relevant for the given object. If no actions are to be presented for the object, an empty list(`[]`) can be returned.

Default implementation returns the value of `item_actions` class variable.

Since this method is called once for each row item, you can customize the actions that is presented for each object. You can also altogether turn off all actions for an object by returning an empty list(`[]`).

invoke_action(*request*, *index*, *item*)
Invokes the custom action specified by the index.

Parameters: request - HttpRequest object index - the index of the action into get_item_actions() list item_or_list - the item upon which action is to be performed

Return: Action result as a 2-tuple: (bool, message)

Raises: Index error if the action index specified is outside the scope of the array returned by get_item_actions().

get_context_data(*kwargs*)
Called for every CRUD view's get_context_data() method, this method allows additional data to be added to the CRUD view's template context.

Parameters **kwargs** – The kwargs dictionary that is the usual argument to View.get_context_data()

Return type None

If you want to add context data for a specific CRUD view, you can achieve this by checking the view object's class type. For example the following code adds context data only for DetailView:

```
if isinstance(self.view, DetailView):
    obj = kwargs['object']
    kwargs['user_fullname'] = obj.user.first_name + ' ' + obj.user.last_name
```

2.1.2 RelatedFieldPopupFormWidget

class popupcrud.widgets.**RelatedFieldPopupFormWidget**(*widget*, *new_url*, **args*, ***kwargs*)
A modified version of django.admin's *RelatedFieldWidgetWrapper*, adds a **Create New** hyperlink to the bottom

of the select box of a related object field. This hyperlink will have CSS class *add-another* and its id set to *add_id_<field_name>* with its href set to *javascript:void(0);*.

The associated JavaScript *popupcrud/js/popupcrud.js*, binds a click handler to *.add-another*, which then activates the Bootstrap modal associated with the hyperlink. The modal body will be filled with the HTML response from an AJAX request to the hyperlink's *data-url* attribute value.

The JavaScript file is added to the form's media list automatically.

__init__ (*widget*, *new_url*, **args*, ***kwargs*)

Constructor takes the following required parameters:

Parameters

- **widget** – The underlying *Select* widget that this widget replaces.
- **url** – The url to load the HTML content to fill the assocaited modal body.

2.2 Template Tags

2.2.1 bsmodal

A tag to help creation of Bootstrap modal dialogs. You may use this tag as:

```
{% bsmodal dialogTitle dialogId [close_title_button={Yes|No}] %}
    <dialog content goes here>
{% endbsmodal %}
```

dialogTitle Required. The title of the modal window. This can be a template variable (created with `{% trans 'something' as var %}`) or a string literal.

dialogId Required. The id of the modal window specified as string literal.

close_title_button Optional. A flag indicating whether to show the modal window close button on the titlebar. Specify one of `Yes` or `No`.

size Optional. Dialog size hint. Acceptable values: `{small|normal|large}`. Defaults to `normal`.

This would create a hidden dialog with title `dialogTitle` and id `dialogId`. The content of the dialog body is to be written between the pair of tags `{% bsmodal %}` and `{% endbsmodal %}`.

The final rendered html fragment would look like this:

```
<div class="modal fade" tabindex="-1" role="dialog">
    <div class="modal-dialog modal-dialog-top" role="document">
        <div class="modal-content">
            <div class="modal-header">
                <button type="button" class="close" data-dismiss="modal"␣
↪aria-label="Close"><span aria-hidden="true">&times;</span></button>
                <h4 class="modal-title">{{dialogTitle}}</h4>
            </div>
            <div class="modal-body">
                <..content between bsmodal & endbsmodal tags..>
            </div>
        </div>
    </div>
</div>
```

The html template for the modal is stored in `popupcrud/modal.html`. So if you want to custom styling of the modal windows, you may define your own template in your projects `templates` folder.

Refer to Boostrap docs on modals for more information on how to show and hide the modal windows.

CHAPTER 3

How-tos

3.1 Create CRUD views for a model

Given a model in app named `library` (source code taken from the `demo` project) in project's repo:

```
# library/models.py
class Author(models.Model):
    name = models.CharField("Name", max_length=128)
    penname = models.CharField("Pen Name", max_length=128)
    age = models.SmallIntegerField("Age", null=True, blank=True)

    class Meta:
        ordering = ('name',)
        verbose_name = "Author"
        verbose_name_plural = "Authors"

    def __str__(self):
        return self.name
```

Declare a PopupCrudViewSet derived class in app's views.py:

```
# library/views.py
from popupcrud.views import PopupCrudViewSet

class AuthorViewSet(PopupCrudViewSet):
    model = Author
    fields = ('name', 'penname', 'age')
    list_display = ('name', 'penname', 'age')
    list_url = reverse_lazy("library:authors")
    new_url = reverse_lazy("library:new-author")

    def get_edit_url(self, obj):
        return reverse_lazy("library:edit-author", kwargs={'pk': obj.pk})
```

(continues on next page)

(continued from previous page)

```
    def get_delete_url(self, obj):
        return reverse_lazy("library:delete-author", kwargs={'pk': obj.pk})
```

Wire up the individual CRUD views generated by the viewset to the app URL namespace in urls.py:

```
# library/urls.py
urlpatterns= [
    url(r'^authors/$', views.AuthorCrudViewset.list(), name='authors'),
    url(r'^authors/new/$', views.AuthorCrudViewset.create(), name='new-author'),
    url(r'^authors(?P<pk>\d+)/edit/$', views.AuthorCrudViewset.update(), name='edit-
↪author'),
    url(r'^authors(?P<pk>\d+)/delete/$', views.AuthorCrudViewset.delete(), name=
↪'delete-author'),
    ]
```

In the projects root urls.py:

```
# demo/urls.py
urlpatterns + [
    url(r'^library/', include('library.urls', namespace='library')),
]
```

3.2 Control access using permissions

In your CRUD ViewSet, declare the permissions required for each CRUD view as:

```
class AuthorViewSet(PopupCrudViewSet):
    model = Author
    ...
    list_permission_required = ('library.list_authors',)
    create_permission_required = ('library.add_author',)
    update_permission_required = ('library.change_author',)
    delete_permission_required = ('library.delete_author',)
```

However, if you want to determine the permission dynamically, override the `get_permission_required()` method and implement your custom permission logic:

```
class AuthorViewSet(PopupCrudViewSet):
    model = Author
    ...

    def get_permission_required(self, op):
        if op == 'create':
            # custom permission for creating new objects

        elif op == 'delete':
            # custom permission for updating existing objects
        else:
            return super(AuthorViewSet, self).get_permission_required(op)
```

3.3 Create a model object from its FK select box in another form

This allows user to create new instances of a model while they are working on a form which has a FK reference to the model for which PopupCrudViewSet views exist. This allows objects to be added seamlessly without the user switching context to another page to add the object and then coming back to work on the form.

To illustrate with an example:

```
from popupcrud.widgets import RelatedFieldPopupFormWidget

class AuthorRatingForm(forms.Form):
    author = forms.ModelChoiceField(queryset=Author.objects.all())
    rating = forms.ChoiceField(label="Rating", choices=(
        ('1', '1 Star'),
        ('2', '2 Stars'),
        ('3', '3 Stars'),
        ('4', '4 Stars')
    ))

    def __init__(self, *args, **kwargs):
        super(AuthorRatingForm, self).__init__(*args, **kwargs)
        author = self.fields['author']
        # Replace the default Select widget with PopupCrudViewSet's
        # RelatedFieldPopupFormWidget. Note the url argument to the widget.
        author.widget = RelatedFieldPopupFormWidget(
            widget=forms.Select(choices=author.choices),
            new_url=reverse_lazy("library:new-author"))


class AuthorRatingView(generic.FormView):
    form_class = AuthorRatingForm

    # rest of the View handling code as per Django norms
```

In the above form, the default widget for `author`, django.forms.widgets.Select has been replaced by `RelatedFieldPopupFormWidget`. Note the arguments to the widget constructor – it takes the underlying Select widget and a url to create a new instance of the model.

3.4 Use Select2 instead of native Select widget

Select2 is an advanced version the browser native Select box allowing users navigate through fairly large selection list using keystrokes. Select2 is excellently supported in Django through the thirdparty app django-select2. Replacing the native `django.forms.Select` control with equivalent `django_select2.forms.Select2Widget` widget is extremely easy:

```
from django_select2.forms import Select2Widget
from popupcrud.widgets import RelatedFieldPopupFormWidget

class AuthorRatingForm(forms.Form):
    author = forms.ModelChoiceField(queryset=Author.objects.all())
    rating = forms.ChoiceField(label="Rating", choices=(
        ('1', '1 Star'),
        ('2', '2 Stars'),
        ('3', '3 Stars'),
        ('4', '4 Stars')
```

(continues on next page)

(continued from previous page)

```
    ))

    def __init__(self, *args, **kwargs):
        super(AuthorRatingForm, self).__init__(*args, **kwargs)
        author = self.fields['author']
        # Replace the default Select widget with PopupCrudViewSet's
        # RelatedFieldPopupFormWidget. Note the url argument to the widget.
        author.widget = RelatedFieldPopupFormWidget(
            widget=forms.Select2Widget(choices=author.choices),
            new_url=reverse_lazy("library:new-author"))
```

Note how `Select2Widget` is essentially a drop in replacement for the native `django.forms.Select` widget. Consult `django-select2` docs for instructions on integrating it with your project.

3.5 Providing your own templates

Out of the box, `popupcrud` comes with its own templates for rendering all the CRUD views. For most use cases this ought to suffice. For the detail view, the default template just renders the object name in the popup. Typically, you might want to include additional information about an object in its detail view. To do this, implement `<model>_detail.html` in your app's template folder and this template will be used to display details about an object.

One point to highlight about templates is that since popupcrud can work in both legacy(like Django admin) and the more modern Web 2.0 modal dialog based modes, it needs two templates to render the content for the two modes. This is necessary as contents of a modal popup window should only contain details of the object without site-wide common elements such as headers and menu that is usually provided through a base template whereas the dedicated legacy crud page requires all the site-wide common artifacts. This problem exists for all CRUD views - create, update, delete and detail. Therefore, for consistency across different CRUD views, popupcurd uses a standard file naming convention to determine the template name to use for the given CRUD view mode.

This convention gives first priority to Django generic CRUD views' default template file name. If it's present it will be used for the CRUD view. However, if the view is to be rendered in a modal popup window, which should not have site-wide common artifacts, popupcrud appends `_inner` to the base template filename (the part before `.html`). So if you want to display details of a object of class `Book` in a modal popup, you have to implement the template file `book_detail_inner.html`. However, if you disable popups for the `detail` view, you have to implement `book_detail.html`. The difference between the two being that `*_inner.html` only renders the object's details whereas `book_detail.html` renders the object's details along with site-wide page common artifacts such as header, footers and/or sidebars.

One strategy is to provide both templates and organize them using the `{% include %}` tag. With this pattern, `book_detail.html` would look like this:

```
{% extends "base.html" %}
{% block content %}
{% include "book_detail_inner.html" %}
{% endblock content %}
```

The same pattern is applicable to other CRUD views as well where template files such as `book_form.html`, `confirm_book_delete.html` are looked for first before using popupcrud's own internal templates.

3.6 Use the formset feature

To add a formset to edit objects of a child model, override the `PopupCrudViewSet.get_formset_class()` method in your derived class returning the `BaseModelFormSet` class which will be used to render the formset along with the model form. Formsets are always rendered at the bottom of the model form.

To illustrate with an example, assume that we have a `Book` table with the following definition:

```
class Book(models.Model):
    title = models.CharField('Title', max_length=128)
    isbn = models.CharField('ISBN', max_length=12)
    author = models.ForeignKey(Author)

    class Meta:
        ordering = ('title',)
        verbose_name = "Book"
        verbose_name_plural = "Books"

    def __str__(self):
        return self.title
```

To allow the user to edit one or more `Book` objects while creating or editing a `Author` object, you just need to extend the `AuthorCrudViewset` in the previous example to:

```
from django import forms
from popupcrud.views import PopupCrudViewSet

class AuthorViewSet(PopupCrudViewSet):
    model = Author
    ...

    def get_formset_class(self):
        return forms.models.inlineformset_factory(
            Author,
            Book,
            fields=('title', 'isbn'),
            can_delete=True,
            extra=1)
```

Now when the modal for create or edit views will show a formset at the bottom with two fields – `Book.title` and `Book.isbn`. A button at the bottom of the formset allows additional formset rows to be added. Each formset row will also have a button at the right to delete the row.

The sample above uses the django formset factory function to dynamically build a formset class based on models parent-child relationship. You may also return a custom formset class that is derived from `BaseModelFormSet` with appropriate specializations to suit your requirements.

`BaseModelFormSet` base class requirement is due to `PopupCrudViewSet` invoking the `save()` method of the class to save formset data if all of them pass the field validation rules.

A note about formset feature. Since formset forms are rendered in a tabular format, and since the modal dialogs are not resizable, there is a limit to the number of formset form fields that can be specified before it becomes unusable for the user. To cater for this, `PopupCrudViewSet` now allows the modal sizes to be adjusted through the `modal_sizes` class attribute. This allows you to specify the appropriate modal size based on your form and formset field count & sizes. See *modal sizes*.

CHAPTER 4

Demo Project

The demo project in folder `demo` shows four usage scenarios of `PopupCrudViewSet`. To run the demo, issue the following commands from `demo` folder:

```
./manage migrate --settings demo.settings
./manage runserver --settings demo.settings
```

Homepage has links to the various views in the project that demonstrates different use cases. Each link has a brief description on the type of use case it demonstrates.

One of the forms in the demo `MultipleRelatedObjectForm`, shows how the advanced `Select2` can be used instead of the django's native *'Select'* widget. For this to work, you need to install `django-select2` in the virtual environment where `demo` is run.

CHAPTER 5

Settings

popupcrud.views.**POPUPCRUD_DEFAULTS**

django-popupcrud global settings are specified as the dict variable `POPUPCRUD` in settings.py.

`POPUPCRUD` currently supports the following settings with their default values:

- `base_template`: The prjoject base template from which all popupcrud templates should be derived.

  Defaults to `base.html`.

- `page_title_context_variable`: Name of the context variable whose value will be set as the title for the CRUD list view page. This title is specified as the value for the class attribute `ViewSet.page_title` or as the return value of `ViewSet.get_page_title()`.

  Defaults to `page_title`.

- `paginate_by`: Default number of rows per page for queryset pagination. This is the same as ListView.paginate_by.

  Defaults to 10.

Python Module Index

p

popupcrud.templatetags.bsmodal, 13

Index

Symbols

__init__() (*popupcrud.widgets.RelatedFieldPopupFormWidget method*), 13

B

breadcrumbs (*popupcrud.views.PopupCrudViewSet attribute*), 8
breadcrumbs_context_variable (*popupcrud.views.PopupCrudViewSet attribute*), 8

C

context_object_name (*popupcrud.views.PopupCrudViewSet attribute*), 9
create() (*popupcrud.views.PopupCrudViewSet class method*), 9
create_permission_required (*popupcrud.views.PopupCrudViewSet attribute*), 6

D

delete() (*popupcrud.views.PopupCrudViewSet class method*), 9
delete_permission_required (*popupcrud.views.PopupCrudViewSet attribute*), 6
detail() (*popupcrud.views.PopupCrudViewSet class method*), 9
detail_permission_required (*popupcrud.views.PopupCrudViewSet attribute*), 6

E

empty_list_icon (*popupcrud.views.PopupCrudViewSet attribute*), 7
empty_list_message (*popupcrud.views.PopupCrudViewSet attribute*), 8

F

fields (*popupcrud.views.PopupCrudViewSet attribute*), 6
form_class (*popupcrud.views.PopupCrudViewSet attribute*), 6

G

get_breadcrumbs() (*popupcrud.views.PopupCrudViewSet method*), 11
get_context_data() (*popupcrud.views.PopupCrudViewSet method*), 12
get_delete_url() (*popupcrud.views.PopupCrudViewSet method*), 10
get_detail_url() (*popupcrud.views.PopupCrudViewSet method*), 9
get_edit_url() (*popupcrud.views.PopupCrudViewSet method*), 9
get_empty_list_icon() (*popupcrud.views.PopupCrudViewSet method*), 11
get_empty_list_message() (*popupcrud.views.PopupCrudViewSet method*), 11
get_form_kwargs() (*popupcrud.views.PopupCrudViewSet method*), 11
get_formset() (*popupcrud.views.PopupCrudViewSet method*), 12
get_formset_class() (*popupcrud.views.PopupCrudViewSet method*), 11
get_item_actions() (*popupcrud.views.PopupCrudViewSet method*),

12
get_new_url() (popupcrud.views.PopupCrudViewSet method), 9
get_obj_name() (popupcrud.views.PopupCrudViewSet method), 10
get_page_title() (popupcrud.views.PopupCrudViewSet method), 10
get_permission_required() (popupcrud.views.PopupCrudViewSet method), 10
get_queryset() (popupcrud.views.PopupCrudViewSet method), 11

I

invoke_action() (popupcrud.views.PopupCrudViewSet method), 12
item_actions (popupcrud.views.PopupCrudViewSet attribute), 8

L

legacy_crud (popupcrud.views.PopupCrudViewSet attribute), 7
list() (popupcrud.views.PopupCrudViewSet class method), 9
list_display (popupcrud.views.PopupCrudViewSet attribute), 5
list_permission_required (popupcrud.views.PopupCrudViewSet attribute), 6
list_template (popupcrud.views.PopupCrudViewSet attribute), 7
list_url (popupcrud.views.PopupCrudViewSet attribute), 6
login_url (popupcrud.views.PopupCrudViewSet attribute), 7

M

modal_sizes (popupcrud.views.PopupCrudViewSet attribute), 8
model (popupcrud.views.PopupCrudViewSet attribute), 5

N

new_url (popupcrud.views.PopupCrudViewSet attribute), 5

P

page_title (popupcrud.views.PopupCrudViewSet attribute), 7
paginate_by (popupcrud.views.PopupCrudViewSet attribute), 6
permissions_required (popupcrud.views.PopupCrudViewSet attribute), 6
pk_url_kwarg (popupcrud.views.PopupCrudViewSet attribute), 9
popupcrud.templatetags.bsmodal (module), 13
POPUPCRUD_DEFAULTS (in module popupcrud.views), 23
PopupCrudViewSet (class in popupcrud.views), 5
popups (popupcrud.views.PopupCrudViewSet attribute), 11

R

raise_exception (popupcrud.views.PopupCrudViewSet attribute), 7
related_object_popups (popupcrud.views.PopupCrudViewSet attribute), 7
RelatedFieldPopupFormWidget (class in popupcrud.widgets), 12

S

slug_field (popupcrud.views.PopupCrudViewSet attribute), 9
slug_url_kwarg (popupcrud.views.PopupCrudViewSet attribute), 9

U

update() (popupcrud.views.PopupCrudViewSet class method), 9
update_permission_required (popupcrud.views.PopupCrudViewSet attribute), 6
urls() (popupcrud.views.PopupCrudViewSet class method), 10