Editing Django models in the front end Documentation

Release 1.0.0

Mario Orlandi

Jan 27, 2019

Contents:

1 Topics **3**
- 1.1 Basic modals . . . 3
- 1.2 Basic modals with Django . . . 6
- 1.3 Modals with simple content . . . 10
- 1.4 Form validation in the modal . . . 14
- 1.5 Creating and updating a Django Model in the front-end . . . 19
- 1.6 Creating and updating a Django Model in the front-end (optimized) . . . 22
- 1.7 A fully generic solution for Django models editing in the front-end . . . 24
- 1.8 Front-end generic helpers to work with any Model . . . 26
- 1.9 Possible enhancements . . . 35
- 1.10 References . . . 37
- 1.11 Contributing . . . 37

2 Sample code **39**

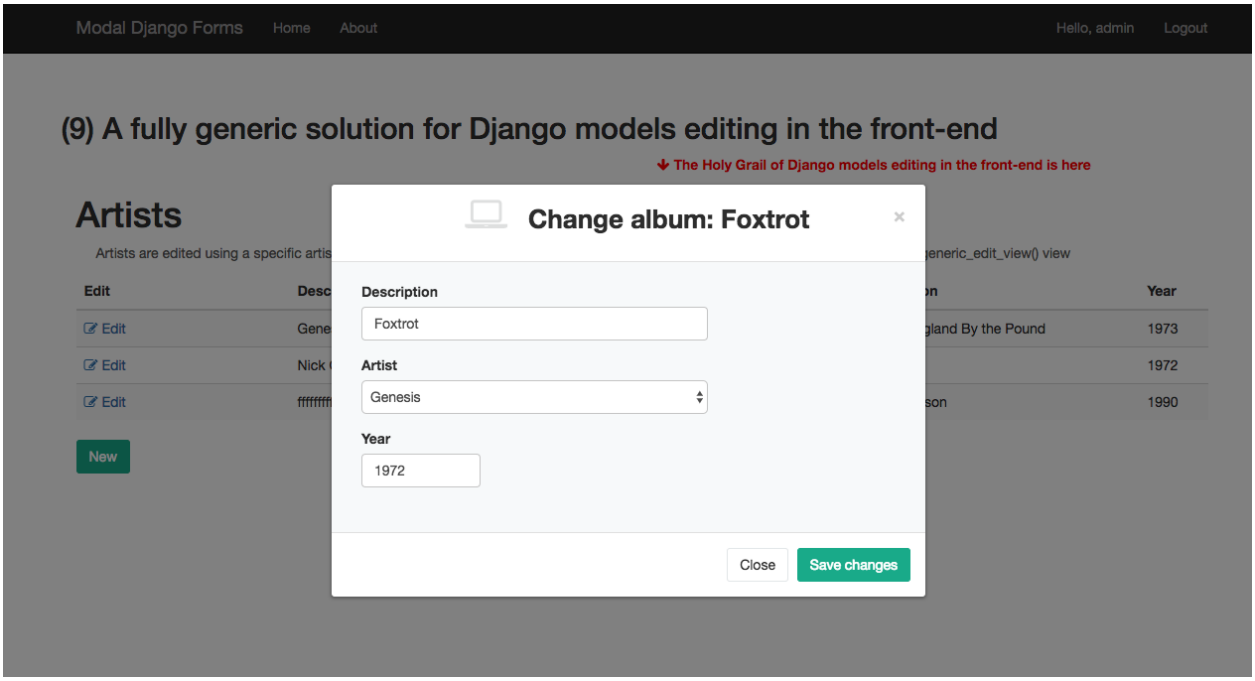
3 Search docs **41**

I try to take advantage of the powerful Django admin in all my web projects, at least in the beginning.

However, as the project evolves and the frontend improves, the usage of the admin tends to be more and more residual.

Adding editing capabilities to the frontend in a modern user interface requires the usage of modal forms, which, to be honest, have always puzzled me for some reason.

This project is not a reusable Django package, but rather a collection of techniques and examples used to cope with modal popups, form submission and validation via ajax, and best practices to organize the code in an effective way to minimize repetitions.



CHAPTER 1

Topics

1.1 Basic modals

1.1.1 HTML popup windows do not exist

There is no such thing as a poup windows in the HTML world.

You have to create the illusion of it stylizing a fragment of the main HTML page, and hiding and showing it as required.

Isn't this cheating ?

1.1.2 A basic modal box with pure Javascript

w3schools.com supplies an example; here is the code:

https://www.w3schools.com/howto/tryit.asp?filename=tryhow_css_modal2

Isn't this too much fuss for such a simple task ?

Well, actually it's not that bad.

Here is how it works:

- a semi-transparent and initially hidden "modal" element covers the whole html page, thus providing a backdrop
- a nested "modal content" element has been given the style to look as a popup window
- you can show or hide the modal by playing with it's display CSS attribute

```
<script language="javascript">

    $(document).ready(function() {

        // Get the modal
        var modal = $('#my-modal');
```

(continues on next page)

Fig. 1: w3school modal example

(continued from previous page)

```
        // Open the modal
        var button = $('#button-open-modal');
        button.on('click', function(event) {
            modal.css('display', 'block');
        })

        // Close the modal
        var close_button = $('.close');
        close_button.on('click', function(event) {
            modal.css('display', 'none');
        })

        // When the user clicks anywhere outside of the modal, close it
        $(window).on('click', function(event) {
            if (event.target.id == modal.attr('id')) {
                modal.css('display', 'none');
            }
        });

    });

</script>
```

Note: Check sample code at: (1) A basic modal box with jQuery

1.1.3 A modal which returns a value

How can we collect a value from the user in the modal window, and return it to the main page ?

We have access to any javascript functions available (after all, we're living in the same HTML page), so we can call any helper just before closing the modal.

```
function close_popup(modal) {
    var value = modal.find('.my-modal-body input').val();
    save_text_value(value);
    modal.hide();
}

function save_text_value(value) {
    if (value) {
        $('#result-wrapper').show();
        $('#result').text(value);
    }
    else {
        $('#result-wrapper').hide();
    }
}
```

Note: Check sample code at: (2) A basic modal box which returns a value

Always remember to clean the input box every time before showing the modal box, as this will be reused again and again . . .

```
function open_popup(modal) {
    var input = modal.find('.my-modal-body input');
    input.val('');
    modal.show();
    input.focus();
}
```

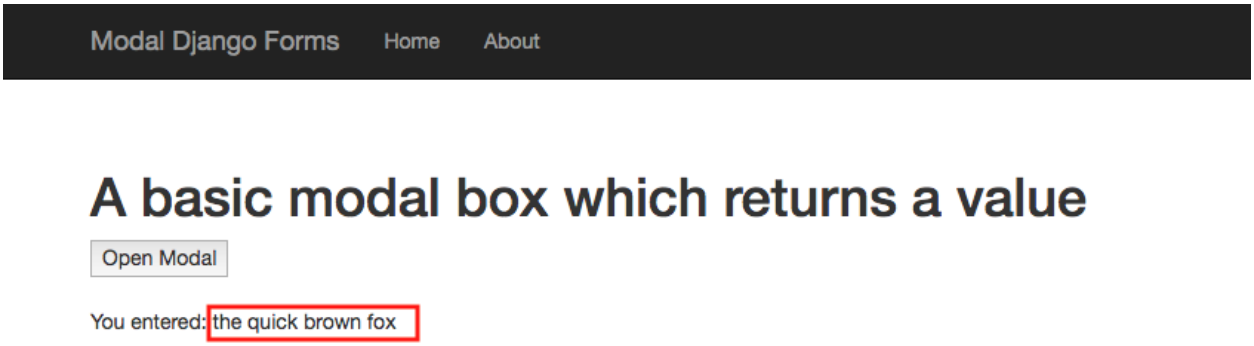
1.1.4 Bootstrap 3 modal plugin

Bootstrap 3 provides a specific plugin to handle modals:

https://getbootstrap.com/docs/3.3/javascript/#modals

You can ask for a larger or smaller dialog specifing either 'modal-lg' or 'modal-sm' class.

The pluging fires some specific events during the modal life cycle:



https://getbootstrap.com/docs/3.3/javascript/#modals-events

Note: Check sample code at: (3) A basic modal box with Bootstrap 3

1.2 Basic modals with Django

1.2.1 Purpose

Spostando la nostra attenzione su un sito dinamico basato su Django, i nostri obiettivi principali diventano:

- disporre di una dialog box da usare come "contenitore" per l'interazione con l'utente, e il cui layout sia coerente con la grafica del front-end
- il contenuto della dialog e il ciclo di vita dell'interazione con l'utente viene invece definito e gestito "lato server"
- la dialog viene chiusa una volta che l'utente completato (o annullato) l'operazione

1.2.2 Display the empty dialog

Il layout di ciascuna dialog box (quindi l'intera finestra a meno del contenuto) viene descritto in un template, e il rendering grafico viene determinato da un unico foglio di stile comune a tutte le finestre (file "modals.css").

Note: Check sample code at: (4) A generic empty modal for Django" illustra diverse possibilita'

Nel caso piu' semplice, ci limitiamo a visualizare la dialog prevista dal template:

```
<a href=""
   onclick="openModalDialog(event, '#modal_generic'); return false;">
    <i class="fa fa-keyboard-o"></i> Open generic modal (no contents, no␣
↪customizations)
</a>
```

Fig. 2: w3school modal example

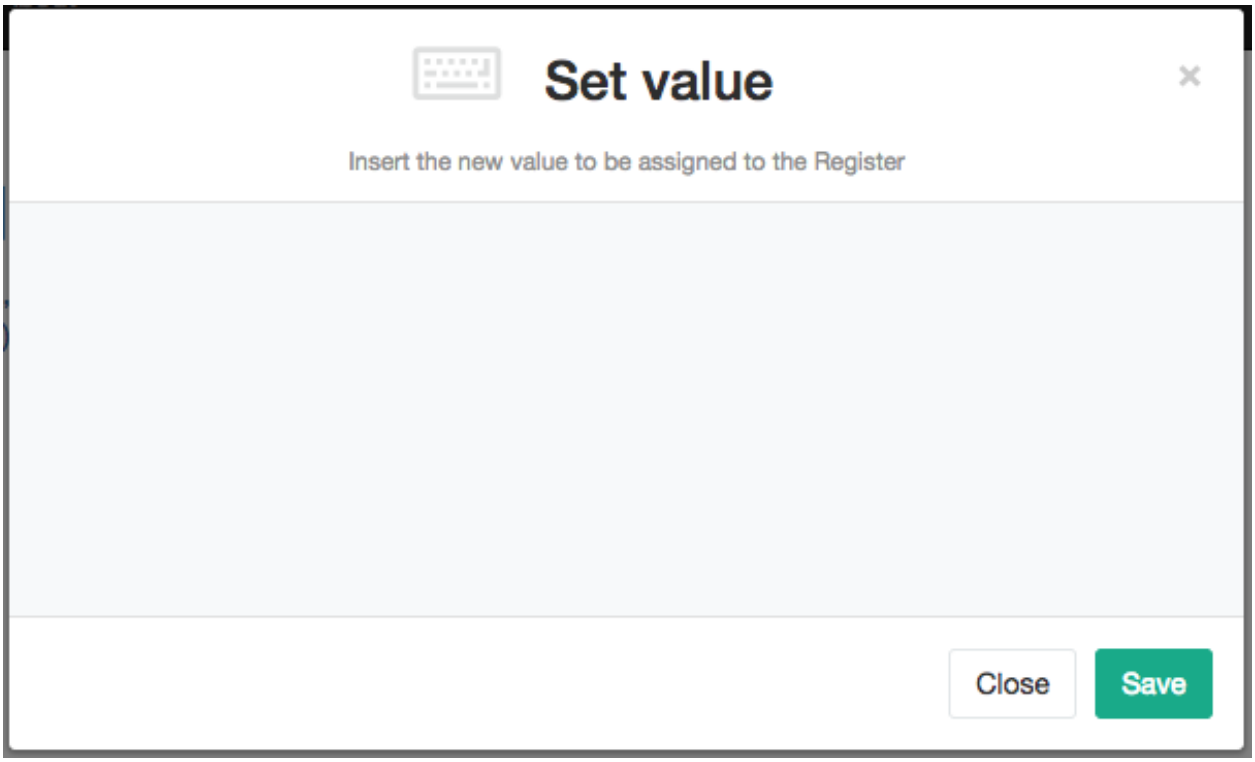
Questo e' sufficiente nei casi in cui il template contenga gia' tutti gli elementi richiesti; ci sono pero' buone possibilita' che un'unica "generica" dialog sia riutilizzabile in diverse circostanze (o forse ovunque) pur di fornire un minimo di informazioni accessorie:

```
<a href=""
   data-dialog-class="modal-lg"
   data-title="Set value"
   data-subtitle="Insert the new value to be assigned to the Register"
   data-icon="fa-keyboard-o"
   data-button-save-label="Save"
   onclick="openModalDialog(event, '#modal_generic'); return false;">
    <i class="fa fa-keyboard-o" style="pointer-events: none;"></i> Open generic modal␣
↪(no contents)
</a>
```

In entrambi i casi si fa' riperimento a un semplice javascript helper, che provvede ad aggiornare gli attributi della dialog prima di visualizzarla, dopo avere reperito i dettagli dall'elemento che l'ha invocata; il vantaggio di questo approccio e' che possiamo definire questi dettagli nel template della pagina principale, e quindi utilizzandone il contesto:

```
<script language="javascript">

    function initModalDialog(event, modal_element) {
        /*
```

(continues on next page)



(continued from previous page)

```
            You can customize the modal layout specifing optional "data" attributes
            in the element (either <a> or <button>) which triggered the event;
            "modal_element" identifies the modal HTML element.

            Sample call:

            <a href=""
               data-title="Set value"
               data-subtitle="Insert the new value to be assigned to the Register"
               data-dialog-class="modal-lg"
               data-icon="fa-keyboard-o"
               data-button-save-label="Save"
               onclick="openModalDialog(event, '#modal_generic'); return false;">
                <i class="fa fa-keyboard-o" style="pointer-events: none;"></i> Open␣
↪generic modal (no contents)
            </a>
        */
        var modal = $(modal_element);
        var target = $(event.target);

        var title = target.data('title') || '';
        var subtitle = target.data('subtitle') || '';
        // either "modal-lg" or "modal-sm" or nothing
        var dialog_class = (target.data('dialog-class') || '') + ' modal-dialog';
        var icon_class = (target.data('icon') || 'fa-laptop') + ' fa modal-icon';
        var button_save_label = target.data('button-save-label') || 'Save changes';

        modal.find('.modal-dialog').attr('class', dialog_class);
        modal.find('.modal-title').text(title);
```

(continues on next page)

(continued from previous page)

```
        modal.find('.modal-subtitle').text(subtitle);
        modal.find('.modal-header .title-wrapper i').attr('class', icon_class);
        modal.find('.modal-footer .btn-save').text(button_save_label);
        modal.find('.modal-body').html('');

        // Annotate with target (just in case)
        modal.data('target', target);

        return modal;
    }

    function openModalDialog(event, modal_element) {
        var modal = initModalDialog(event, modal_element);
        modal.modal('show');
    }

</script>
```

1.2.3 Make the modal draggable

To have the modal draggable, you can specify the "draggable" class:

```
<div class="modal draggable" id="modal_generic" tabindex="-1" role="dialog" aria-
↪hidden="true">
    <div class="modal-dialog">
      ...
```

and add this statement at the end of initModalDialog():

```
if (modal.hasClass('draggable')) {
    modal.find('.modal-dialog').draggable({
        handle: '.modal-header'
    });
}
```

Warning: draggable() requires the inclusion of jQuery UI

It's usefull to give a clue to the user adding this style:

```
.modal.draggable .modal-header {
    cursor: move;
}
```

1.2.4 Organizzazione dei files

Per convenienza, tutti i templates relativi alle dialog (quello generico e le eventuali varianti specializzate) vengono memorizzate in un unico folder:

templates/frontent/modals

e automaticamente incluse nel template "base.html":

```
{% block modals %}
    {% include 'frontend/modals/generic.html' %}
    {% include 'frontend/modals/dialog1.html' %}
    {% include 'frontend/modals/dialog2.html' %}
    ...
{% endblock modals %}
```

Questo significa che tutte le modal dialogs saranno disponibili in qualunque pagina, anche quando non richieste; trattandosi di elementi non visibili della pagina, non ci sono particolari controindicazioni; nel caso, il template specifico puo' eventulmente ridefinire il blocco *{% block modals %}* ed includere i soli template effettivamente necessari.

Altri files utilizzati:

- *static/frontend/css/modals.css*: stili comuni a tutte le dialogs
- *static/frontend/js/modals.js*: javascript helpers pertinenti alla gestione delle dialogs

1.3 Modals with simple content

Possiamo riempire il contenuto della dialog invocando via Ajax una vista che restituisca l'opportuno frammento HTML:

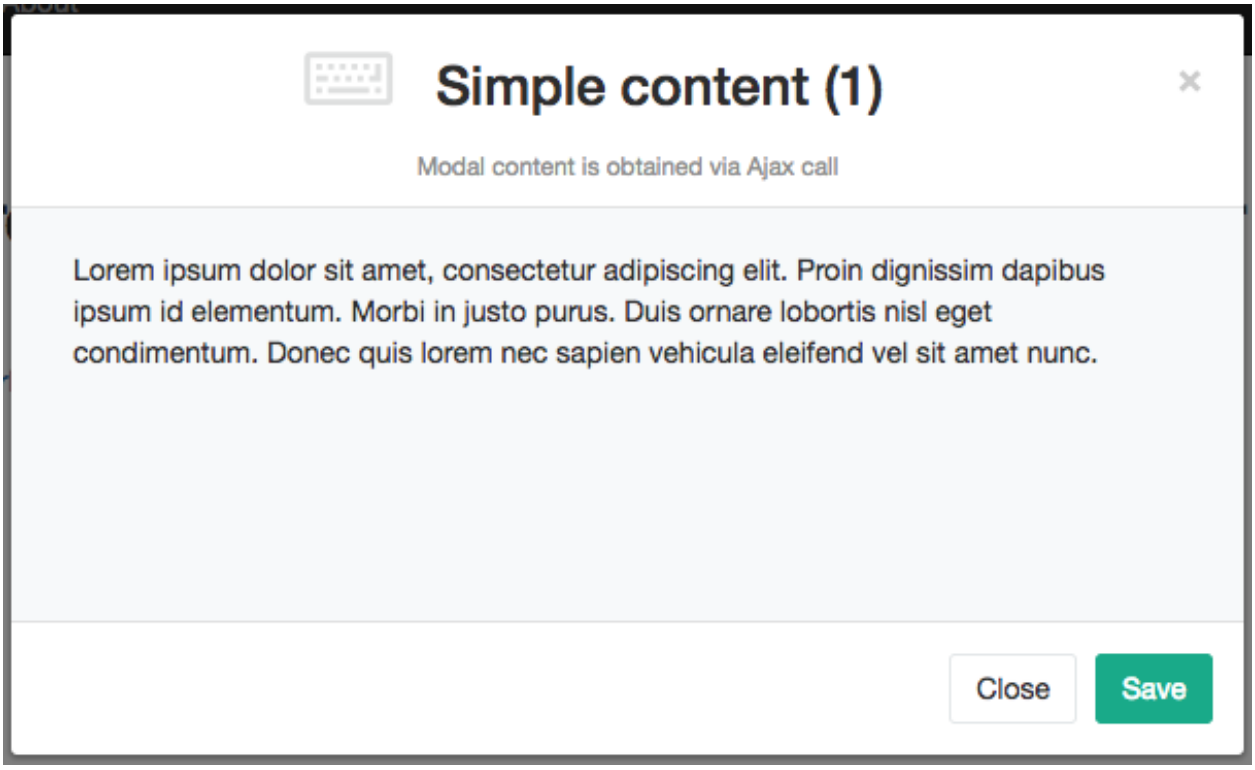
```
<a href=""
   data-action="{% url 'frontend:simple-content' %}"
   data-title="Simple content"
   data-subtitle="Modal content is obtained via Ajax call"
   data-icon="fa-keyboard-o"
   data-button-save-label="Save"
   onclick="openMyModal(event); return false;">
    <i class="fa fa-keyboard-o"></i> Modal with simple content
</a>
```

```
<script language="javascript">

    function openMyModal(event) {
        var modal = initModalDialog(event, '#modal_generic');
        var url = $(event.target).data('action');
        modal.find('.modal-body').load(url, function() {
            modal.modal('show');
        });
    }

</script>
```

```
def simple_content(request):
    return HttpResponse('Lorem ipsum dolor sit amet, consectetur adipiscing elit.␣
↪Proin dignissim dapibus ipsum id elementum. Morbi in justo purus. Duis ornare␣
↪lobortis nisl eget condimentum. Donec quis lorem nec sapien vehicula eleifend vel␣
↪sit amet nunc.')
```

Si osservi come abbiamo specificato l'url della view remota nell'attributo "data-action" del trigger.

Questa soluzione e' preferibile rispetto all'utilizzo di *href* per evitare che, in caso di errori javascript il link, il link invochi direttamente la view, compromettendo quindi l'intera pagina o inserendo nel corpo della dialog contenuti inappropriati.



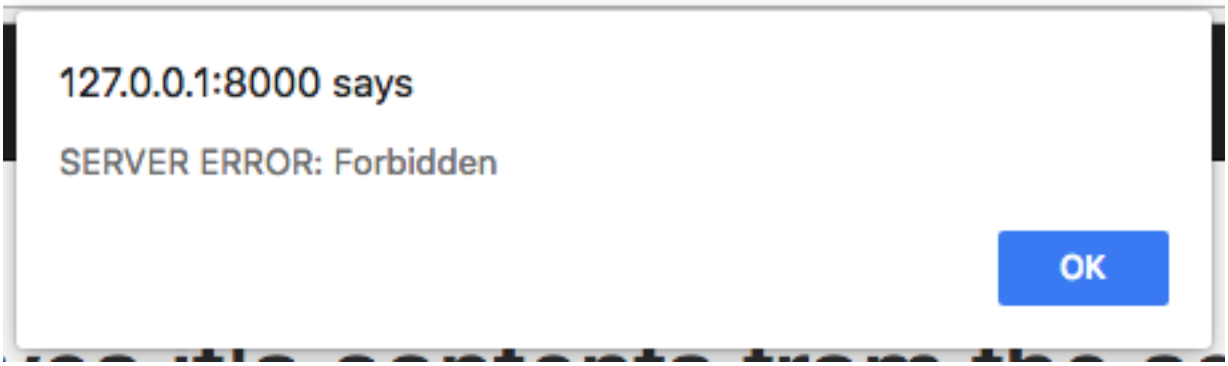
Un limite del codice precedente e' che non siamo in grado di rilevare eventuali errori del server, nel qual caso la dialog verrebbe comunque aperta (con contenuto vuoto).

Il problema viene facilmente superato invocando direttamente $.ajax() anziche' lo shortcut load().

La soluzione e' leggermente piu' verbose, ma consente un controllo piu' accurrato:

```
<script language="javascript">

    function openMyModal(event) {
        var modal = initModalDialog(event, '#modal_generic');
        var url = $(event.target).data('action');
        $.ajax({
            type: "GET",
            url: url
        }).done(function(data, textStatus, jqXHR) {
            modal.find('.modal-body').html(data);
            modal.modal('show');
        }).fail(function(jqXHR, textStatus, errorThrown) {
            alert("SERVER ERROR: " + errorThrown);
        });
    }

</script>
```

```
def simple_content_forbidden(request):
    raise PermissionDenied
```



1.3.1 More flexible server side processing

A volte puo' essere utile riutilizzare la stessa view per fornire, a seconda delle circostanze, una dialog modale oppure una pagina standalone.

La soluzione proposta prevede l'utilizzo di templates differenziati nei due casi:

```
def simple_content2(request):

    # Either render only the modal content, or a full standalone page
    if request.is_ajax():
        template_name = 'frontend/includes/simple_content2_inner.html'
    else:
        template_name = 'frontend/includes/simple_content2.html'

    return render(request, template_name, {
    })
```

dove il template "inner" fornisce il contenuto:

```
<div class="row">
    <div class="col-sm-4">
        {% lorem 1 p random %}
    </div>
    <div class="col-sm-4">
        {% lorem 1 p random %}
    </div>
    <div class="col-sm-4">
        {% lorem 1 p random %}
    </div>
</div>
```

mentre il template "esterno" si limita a includerlo nel contesto piu' completo previsto dal frontend:

```
{% extends "base.html" %}
{% load static staticfiles i18n %}

{% block content %}
{% include 'frontend/includes/simple_content2_inner.html' %}
{% endblock content %}
```

Note: Check sample code at: (5) Modal with simple content

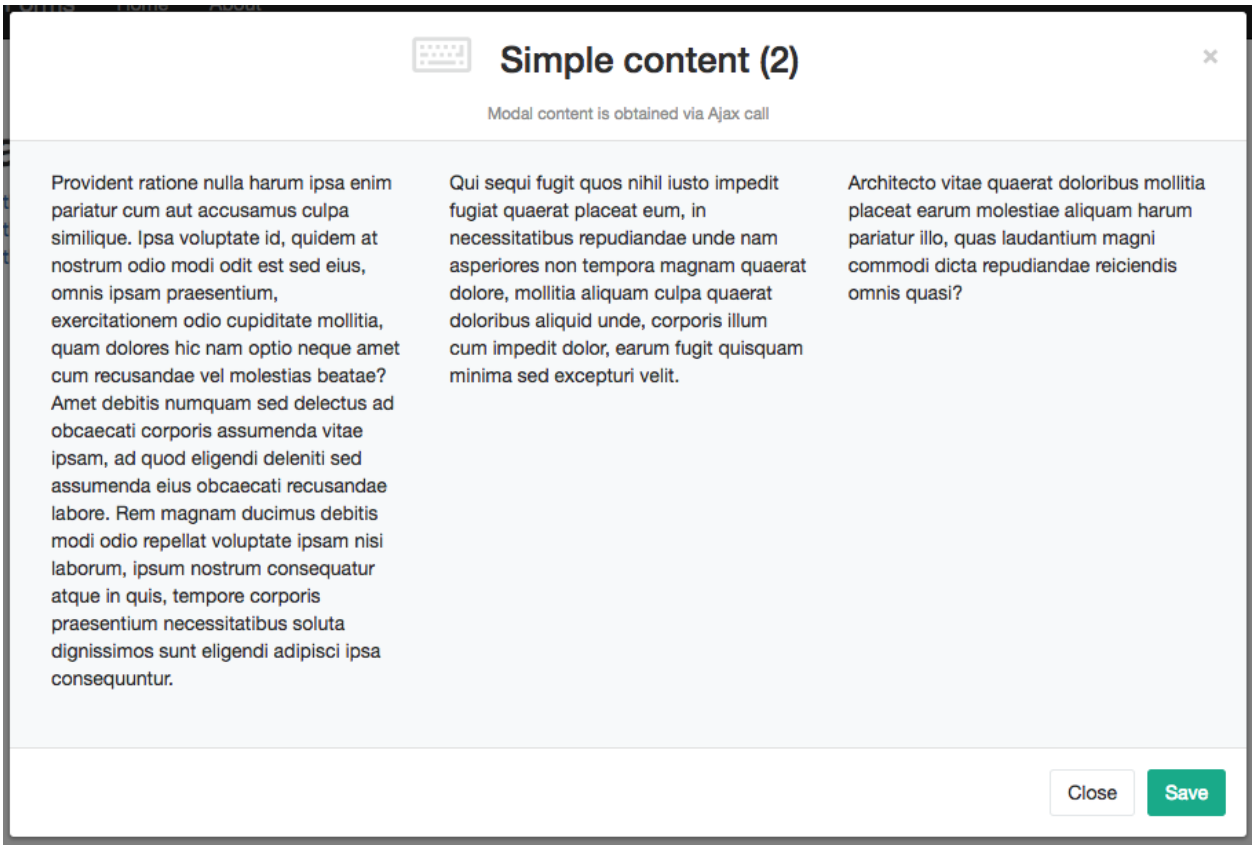


Fig. 3: Modal dialog

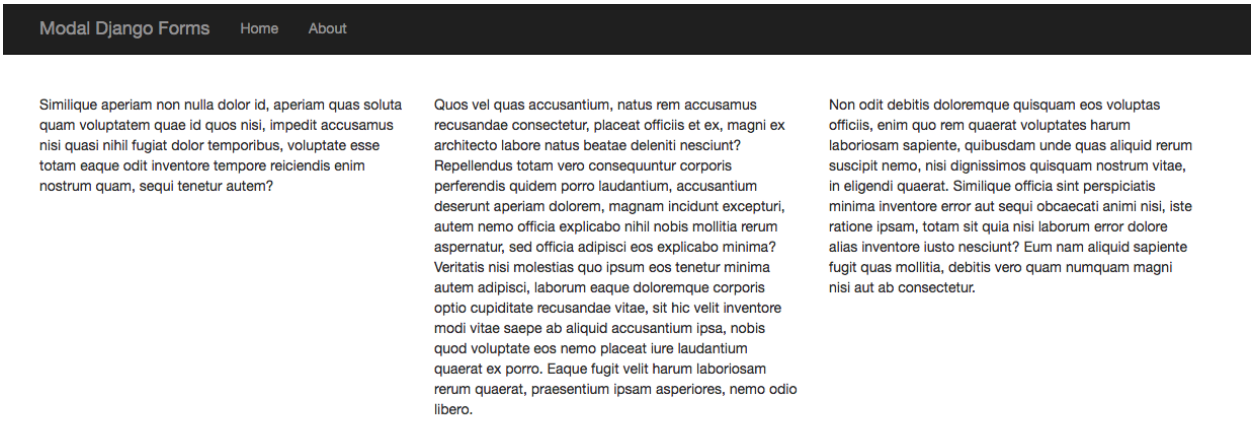


Fig. 4: Same content in a standalone page

1.4 Form validation in the modal

We've successfully injected data retrieved from the server in our modals, but did not really interact with the user yet.

When the modal body contains a form, things start to become interesting and tricky.

1.4.1 Handling form submission

First and foremost, we need to **prevent the form from performing its default submit**.

If not, after submission we'll be redirected to the form action, outside the context of the dialog.

We'll do this binding to the form's submit event, where we'll serialize the form's content and sent it to the view for validation via an Ajax call.

Then, upon a successufull response from the server, **we'll need to further investigate the HTML received**:

- if it contains any field error, the form did not validate successfully, so we update the modal body with the new form and its errors
- otherwise, user interaction is completed, and we can finally close the modal

We'll obtain all this (and more) with a javacript helper function **formAjaxSubmit()** which I'll explain later in details.

```
<script language="javascript">

    function openMyModal(event) {
        var modal = initModalDialog(event, '#modal_generic');
        var url = $(event.target).data('action');
        $.ajax({
            type: "GET",
            url: url
        }).done(function(data, textStatus, jqXHR) {
            modal.find('.modal-body').html(data);
            modal.modal('show');
            formAjaxSubmit(modal, url, null, null);
        }).fail(function(jqXHR, textStatus, errorThrown) {
            alert("SERVER ERROR: " + errorThrown);
        });
    }

</script>
```

Again, the very same view can also be used to render a standalone page:

1.4.2 The formAjaxSubmit() helper

I based my work on the inspiring ideas presented in this brilliant article:

Use Django's Class-Based Views with Bootstrap Modals

Here's the full code:

```
<script language="javascript">

    function formAjaxSubmit(modal, action, cbAfterLoad, cbAfterSuccess) {
        var form = modal.find('.modal-body form');
        var header = $(modal).find('.modal-header');
```

(continues on next page)

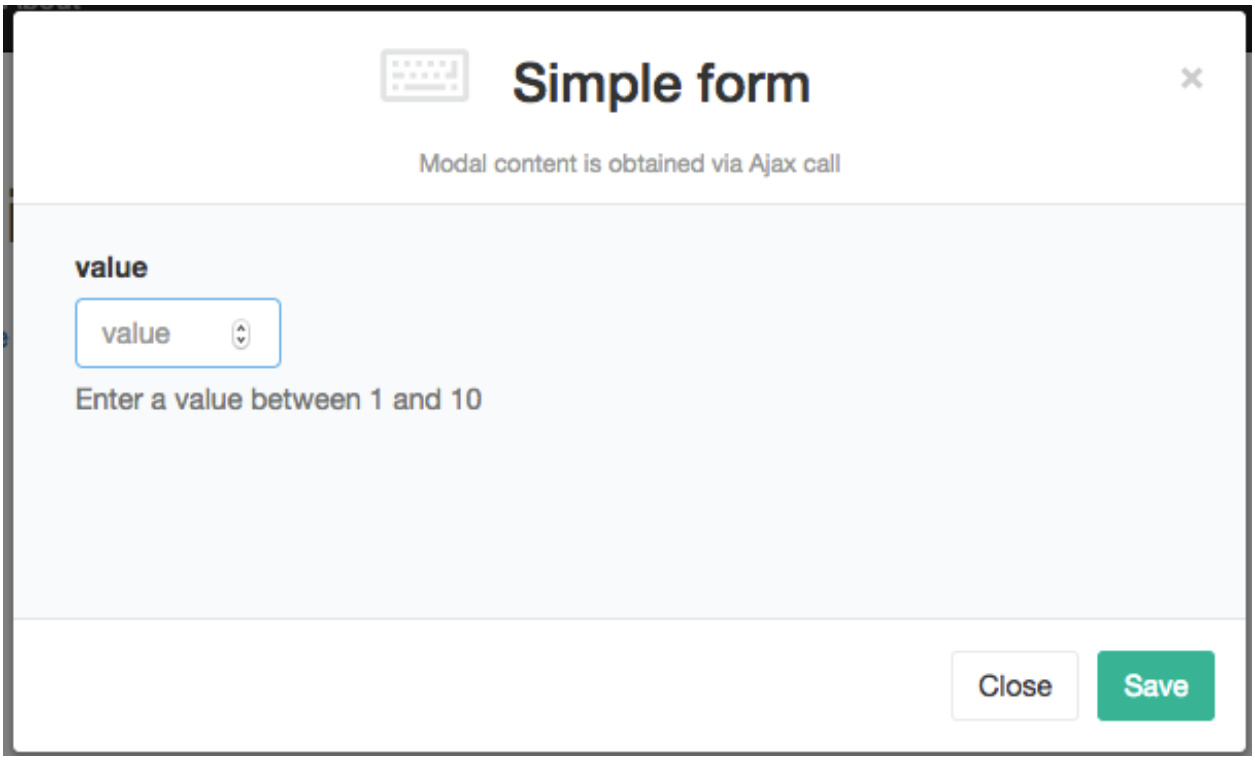


Fig. 5: A form in the modal dialog

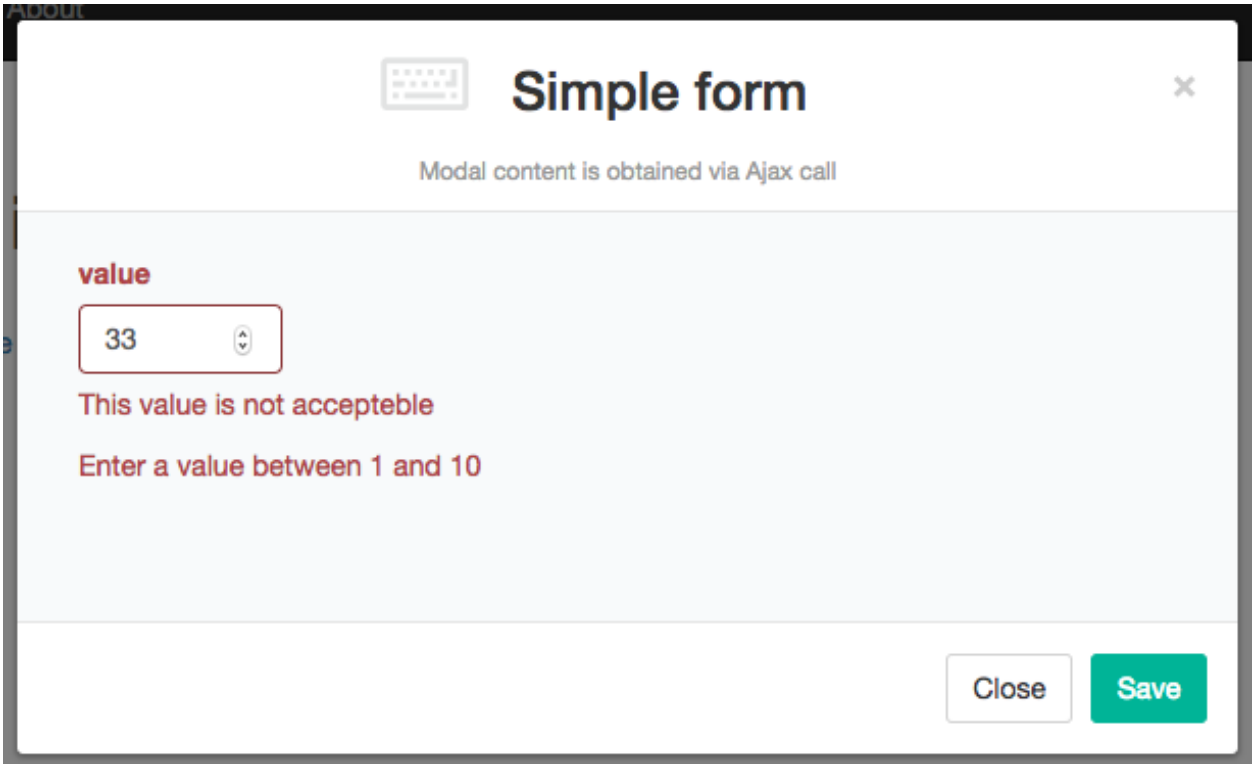
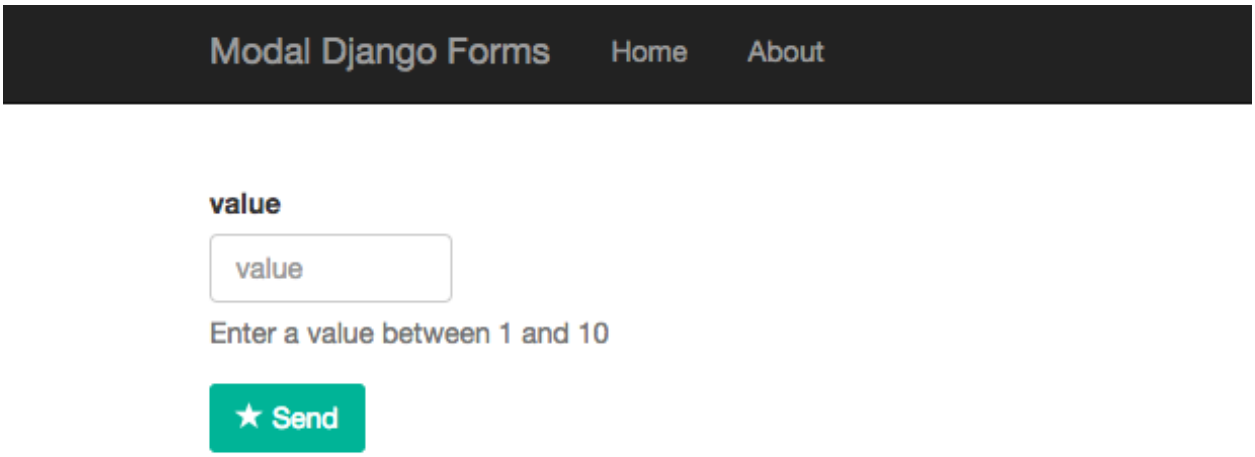


Fig. 6: While the form does not validate, we keep the dialog open



(continued from previous page)

```
        // use footer save button, if available
        var btn_save = modal.find('.modal-footer .btn-save');
        if (btn_save) {
            modal.find('.modal-body form .form-submit-row').hide();
            btn_save.off().on('click', function(event) {
                modal.find('.modal-body form').submit();
            });
        }
        if (cbAfterLoad) { cbAfterLoad(modal); }

        // Give focus to first visible form field
        modal.find('form input:visible').first().focus();

        // bind to the form's submit event
        $(form).on('submit', function(event) {

            // prevent the form from performing its default submit action
            event.preventDefault();
            header.addClass('loading');

            var url = $(this).attr('action') || action;

            // serialize the form's content and send via an AJAX call
            // using the form's defined action and method
            $.ajax({
                type: $(this).attr('method'),
                url: url,
                data: $(this).serialize(),
                success: function(xhr, ajaxOptions, thrownError) {

                    // If the server sends back a successful response,
                    // we need to further check the HTML received

                    // update the modal body with the new form
```

(continues on next page)

(continued from previous page)

```
                    $(modal).find('.modal-body').html(xhr);

                    // If xhr contains any field errors,
                    // the form did not validate successfully,
                    // so we keep it open for further editing
                    if ($(xhr).find('.has-error').length > 0) {
                        formAjaxSubmit(modal, url, cbAfterLoad, cbAfterSuccess);
                    } else {
                        // otherwise, we've done and can close the modal
                        $(modal).modal('hide');
                        if (cbAfterSuccess) { cbAfterSuccess(modal); }
                    }
                },
                error: function(xhr, ajaxOptions, thrownError) {
                    console.log('SERVER ERROR: ' + thrownError);
                },
                complete: function() {
                    header.removeClass('loading');
                }
            });
        });
    }

</script>
```

As anticipated, the most important action is to hijack form submission:

```
// bind to the form's submit event
$(form).on('submit', function(event) {

    // prevent the form from performing its default submit action
    event.preventDefault();
    header.addClass('loading');

    var url = $(this).attr('action') || action;

    // serialize the form's content and sent via an AJAX call
    // using the form's defined action and method
    $.ajax({
        type: $(this).attr('method'),
        url: url,
        data: $(this).serialize(),
        ...
```

If the form specifies an action, we use it as the end-point of the ajax call; if not (which is the most common case), we're using the same view for both rendering and form processing, and we can reuse the original url instead:

```
var url = $(this).attr('action') || action;
```

Secondly, we need to detect any form errors after submission; see the "success" callback after the Ajax call for details.

Finally, we also take care of the submit button embedded in the form. While it's useful and necessary for the rendering of a standalone page, it's rather disturbing in the modal dialog:

Here's the relevant code:

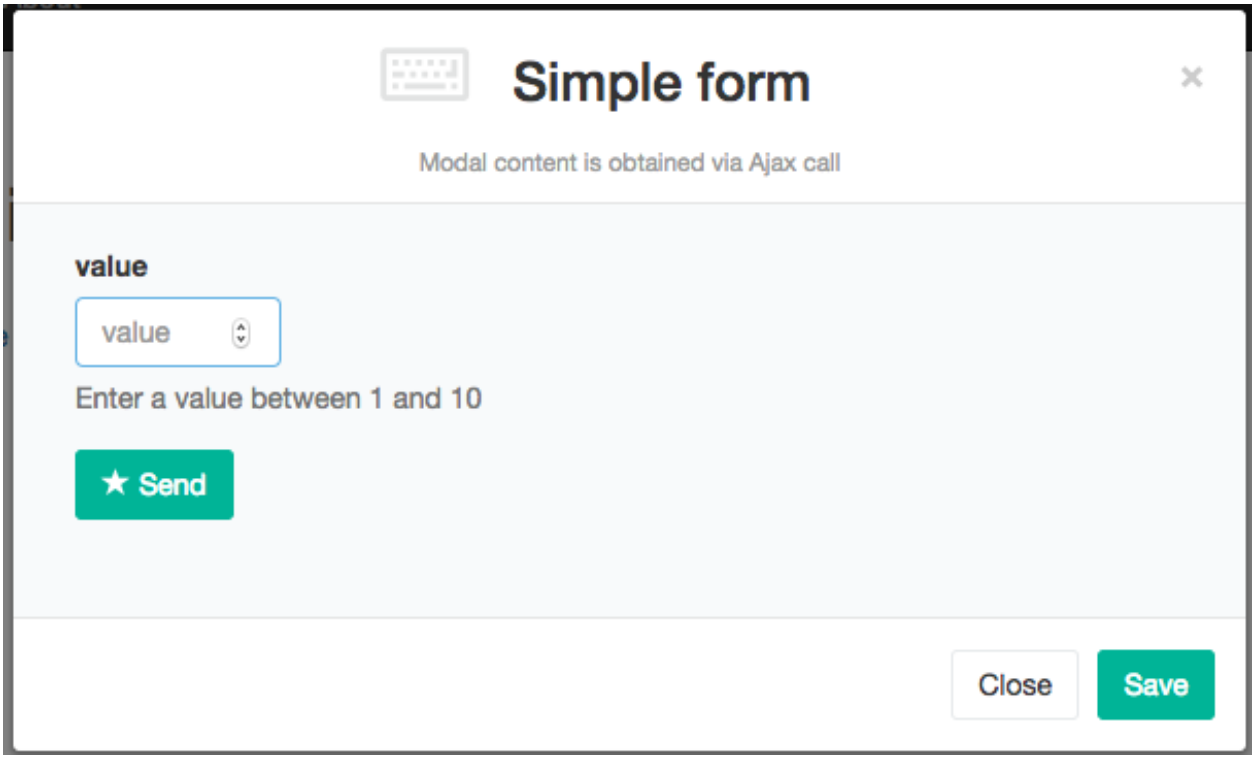


Fig. 7: Can we hide the "Send" button and use the "Save" button from the footer instead ?

```
// use footer save button, if available
var btn_save = modal.find('.modal-footer .btn-save');
if (btn_save) {
    modal.find('.modal-body form .form-submit-row').hide();
    btn_save.off().on('click', function(event) {
        modal.find('.modal-body form').submit();
    });
}
```

One last detail: during content loading, we add a "loading" class to the dialog header, to make a spinner icon visible until we're ready to either update or close the modal.

1.4.3 Optional callbacks supported by formAjaxSubmit()

cbAfterLoad called every time new content has been loaded; you can use it to bind form controls when required

cbAfterSuccess called after successfull submission; at this point the modal has been closed, but the bounded form might still contain useful informations that you can grab for later inspection

Sample usage:

```
...
formAjaxSubmit(modal, url, afterModalLoad, afterModalSuccess);
...

function afterModalLoad(modal) {
```

(continues on next page)

(continued from previous page)

```
    console.log('modal %o loaded', modal);
}

function afterModalSuccess(modal) {
    console.log('modal %o succeeded', modal);
}
```

Note: Check sample code at: (6) Form validation in the modal

Warning: In the sample project, a sleep of 1 sec has been included in the view (POST) to simulate a more complex elaboration which might occur in real situations

1.5 Creating and updating a Django Model in the front-end

We can now apply what we've built so far to edit a specific Django model from the front-end.

Note: Check sample code at: (7) Creating and updating a Django Model in the front-end

1.5.1 Creating a new model

This is the view:

```
@login_required
def artist_create(request):

    if not request.user.has_perm('backend.add_artist'):
        raise PermissionDenied

    # Either render only the modal content, or a full standalone page
    if request.is_ajax():
        template_name = 'frontend/includes/generic_form_inner.html'
    else:
        template_name = 'frontend/includes/generic_form.html'

    object = None
    if request.method == 'POST':
        form = ArtistCreateForm(data=request.POST)
        if form.is_valid():
            object = form.save()
            if not request.is_ajax():
                # reload the page
                next = request.META['PATH_INFO']
                return HttpResponseRedirect(next)
            # if is_ajax(), we just return the validated form, so the modal will close
    else:
        form = ArtistCreateForm()
```

(continues on next page)

(continued from previous page)

```
    return render(request, template_name, {
        'form': form,
        'object': object,
    })
```

and this is the form:

```
class ArtistCreateForm(forms.ModelForm):

    class Meta:
        model = Artist
        fields = [
            'description',
            'notes',
        ]
```

Note that we're using a generic template called *frontend/includes/generic_form_inner.html*;

Chances are we'll reuse it unmodified for other Models as well.

```
{% load i18n bootstrap3 %}

<div class="row">
    <div class="col-sm-8">

        <form method="post" class="form" novalidate>
            {% csrf_token %}
            {% bootstrap_form form %}
            <input type="hidden" name="object_id" value="{{ object.id|default:'' }}">
            {% buttons %}
                <div class="form-submit-row">
                    <button type="submit" class="btn btn-primary">
                        {% bootstrap_icon "star" %} {% trans 'Send' %}
                    </button>
                </div>
            {% endbuttons %}
        </form>
    </div>
</div>
```

On successful creation, we might want to update the user interface; in the example, for simplicity, we just reload the entire page, but before doing that we also display with an alert the new object id retrieved from the hidden field 'object_id' of the form; this could be conveniently used for in-place page updating.

```
<script language="javascript">

    function afterModalCreateSuccess(modal) {
        var object_id = modal.find('input[name=object_id]').val();
        alert('New artist created: id=' + object_id);
        location.reload(true);
    }

</script>
```

1.5.2 Updating an existing object

We treat the update of an existing object in a similar fashion, but binding the form to the specific database record.

The view:

```
@login_required
def artist_update(request, pk):

    if not request.user.has_perm('backend.change_artist'):
        raise PermissionDenied

    # Either render only the modal content, or a full standalone page
    if request.is_ajax():
        template_name = 'frontend/includes/generic_form_inner.html'
    else:
        template_name = 'frontend/includes/generic_form.html'

    object = get_object_by_uuid_or_404(Artist, pk)
    if request.method == 'POST':
        form = ArtistUpdateForm(instance=object, data=request.POST)
        if form.is_valid():
            form.save()
            if not request.is_ajax():
                # reload the page
                next = request.META['PATH_INFO']
                return HttpResponseRedirect(next)
            # if is_ajax(), we just return the validated form, so the modal will close
    else:
        form = ArtistUpdateForm(instance=object)

    return render(request, template_name, {
        'object': object,
        'form': form,
    })
```

and the form:

```
class ArtistUpdateForm(forms.ModelForm):

    class Meta:
        model = Artist
        fields = [
            'description',
            'notes',
        ]
```

Finally, here's the object id retrival after successful completion:

```
<script language="javascript">

    function afterModalUpdateSuccess(modal) {
        var object_id = modal.find('input[name=object_id]').val();
        alert('Artist updated: id=' + object_id);
        location.reload(true);
    }

</script>
```

1.5.3 Possible optimizations

In the code above, we can detect at list three redundancies:

- the two model forms are identical
- the two views are similar
- and, last but not least, we might try to generalize the views for reuse with any Django model

We'll investigate all these opportunities later on; nonetheless, it's nice to have a simple snippet available for copy and paste to be used as a starting point anytime a specific customization is in order.

1.6 Creating and updating a Django Model in the front-end (optimized)

Let's start our optimizations by removing some redundancies.

Note: Check sample code at: (8) Editing a Django Model in the front-end, using a common basecode for creation and updating

---

Sharing a single view for both creating a new specific Model and updating an existing one is now straitforward; see *artist_edit()* belows:

```
################################################################################
# A single "edit" view to either create a new Artist or update an existing one

def artist_edit(request, pk=None):

    # Retrieve object
    if pk is None:
        # "Add" mode
        object = None
        required_permission = 'backend.add_artist'
    else:
        # Change mode
        object = get_object_by_uuid_or_404(Artist, pk)
        required_permission = 'backend.change_artist'

    # Check user permissions
    if not request.user.is_authenticated or not request.user.has_perm(required_
↪permission):
        raise PermissionDenied

    # Either render only the modal content, or a full standalone page
    if request.is_ajax():
        template_name = 'frontend/includes/generic_form_inner.html'
    else:
        template_name = 'frontend/includes/generic_form.html'

    if request.method == 'POST':
        form = ArtistEditForm(instance=object, data=request.POST)
        if form.is_valid():
            object = form.save()
            if not request.is_ajax():
```

(continues on next page)

(continued from previous page)

```
                # reload the page
                if pk is None:
                    message = 'The object "%s" was added successfully.' % object
                else:
                    message = 'The object "%s" was changed successfully.' % object
                messages.success(request, message)
                next = request.META['PATH_INFO']
                return HttpResponseRedirect(next)
            # if is_ajax(), we just return the validated form, so the modal will close
    else:
        form = ArtistEditForm(instance=object)

    return render(request, template_name, {
        'object': object,
        'form': form,
    })
```

When "pk" is None, we act in *add* mode, otherwise we retrieve the corresponding object to *change* it.

Both "add" and "change" URL patterns point to the same view, but the first one doesn't capture anything from the URL and the default value of None will be used for *pk*.

```
urlpatterns = [
    ...
    path('artist/add/', views.artist_edit, name="artist-add"),
    path('artist/<uuid:pk>/change/', views.artist_edit, name="artist-change"),
    ...
]
```

We also share a common form:

```
class ArtistEditForm(forms.ModelForm):
    """
    To be used for both creation and update
    """

    class Meta:
        model = Artist
        fields = [
            'description',
            'notes',
        ]
```

The javascript handler which opens the dialog can be refactored in a completely generic way, with no reference to the specific Model in use (and is also reusable for any dialog which submits an arbitrary form):

```
<script language="javascript">

    function openModalDialogWithForm(event, modal, cbAfterLoad, cbAfterSuccess) {
        // If "modal" is a selector, initialize a modal object,
        // otherwise just use it
        if ($.type(modal) == 'string') {
            modal = initModalDialog(event, modal);
        }

        var url = $(event.target).data('action');
        if (!url) {
```

(continues on next page)

(continued from previous page)

```
            console.log('ERROR: openModalDialogWithForm() could not retrieve action␣
↪from event');
            return;
        }

        $.ajax({
            type: 'GET',
            url: url
        }).done(function(data, textStatus, jqXHR) {
            modal.find('.modal-body').html(data);
            modal.modal('show');
            formAjaxSubmit(modal, url, cbAfterLoad, cbAfterSuccess);
        }).fail(function(jqXHR, textStatus, errorThrown) {
            alert('SERVER ERROR: ' + errorThrown);
        });
    }

</script>
```

so I moved it from the template to "modals.js". It can be invoked directly from there, or copied to any local template for further customization.

1.7 A fully generic solution for Django models editing in the front-end

We're really very close to **the Holy Grail of Django models editing in the front-end**.

Can we really do it all with a single generic view ?

Yes sir !

Note: Check sample code at: (9) A fully generic solution for Django models editing in the front-end

```
################################################################################
# A fully generic "edit" view to either create a new object or update an existing one;
# works with any Django model

def generic_edit_view(request, model_form_class, pk=None):

    model_class = model_form_class._meta.model
    app_label = model_class._meta.app_label
    model_name = model_class._meta.model_name
    model_verbose_name = model_class._meta.verbose_name.capitalize()

    # Retrieve object
    if pk is None:
        # "Add" mode
        object = None
        required_permission = '%s.add_%s' % (app_label, model_name)
    else:
        # Change mode
        object = get_object_by_uuid_or_404(model_class, pk)
        required_permission = '%s.change_%s' % (app_label, model_name)
```

(continues on next page)

(continued from previous page)

```
    # Check user permissions
    if not request.user.is_authenticated or not request.user.has_perm(required_
↪permission):
        raise PermissionDenied

    # Either render only the modal content, or a full standalone page
    if request.is_ajax():
        template_name = 'frontend/includes/generic_form_inner.html'
    else:
        template_name = 'frontend/includes/generic_form.html'

    if request.method == 'POST':
        form = model_form_class(instance=object, data=request.POST)
        if form.is_valid():
            object = form.save()
            if not request.is_ajax():
                # reload the page
                if pk is None:
                    message = 'The %s "%s" was added successfully.' % (model_verbose_
↪name, object)
                else:
                    message = 'The %s "%s" was changed successfully.' % (model_
↪verbose_name, object)
                messages.success(request, message)
                next = request.META['PATH_INFO']
                return HttpResponseRedirect(next)
            # if is_ajax(), we just return the validated form, so the modal will close
    else:
        form = model_form_class(instance=object)

    return render(request, template_name, {
        'object': object,
        'form': form,
    })
```

Adding an appropriate ModelForm in the URL pattern is all what we need; from that, the view will deduce the Model and other related details.

```
urlpatterns = [
    ...
    path('album/add/',
        views.generic_edit_view,
        {'model_form_class': forms.AlbumEditForm},
        name="album-add"),
    path('album/<uuid:pk>/change/',
        views.generic_edit_view,
        {'model_form_class': forms.AlbumEditForm},
        name="album-change"),
    ...
]
```

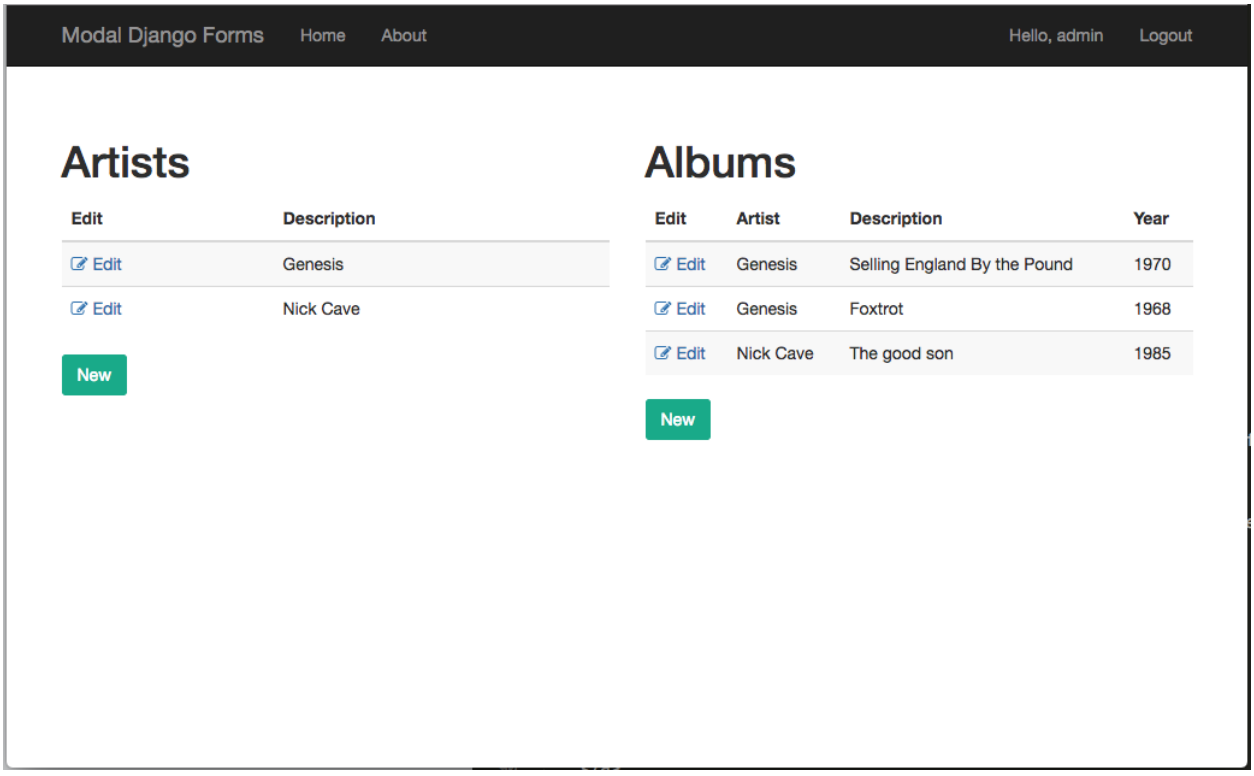
In the page template, we bind the links to the views as follows:

```
<!-- Change -->
<a href=""
   data-action="{% url 'frontend:album-change' row.id %}"
```

(continues on next page)

(continued from previous page)

```
   onclick="openModalDialogWithForm(event, '#modal_generic', null,␣
↪afterObjectEditSuccess); return false;"
   data-title="Change album: {{ row }}"
>
    <i class="fa fa-edit"></i> Edit
</a>

...

<!-- Add -->
<button
    href=""
    data-action="{% url 'frontend:album-add' %}"
    data-title="New album"
    onclick="openModalDialogWithForm(event, '#modal_generic', null,␣
↪afterObjectEditSuccess); return false;"
    type="button"class="btn btn-primary">
    New
</button>
```

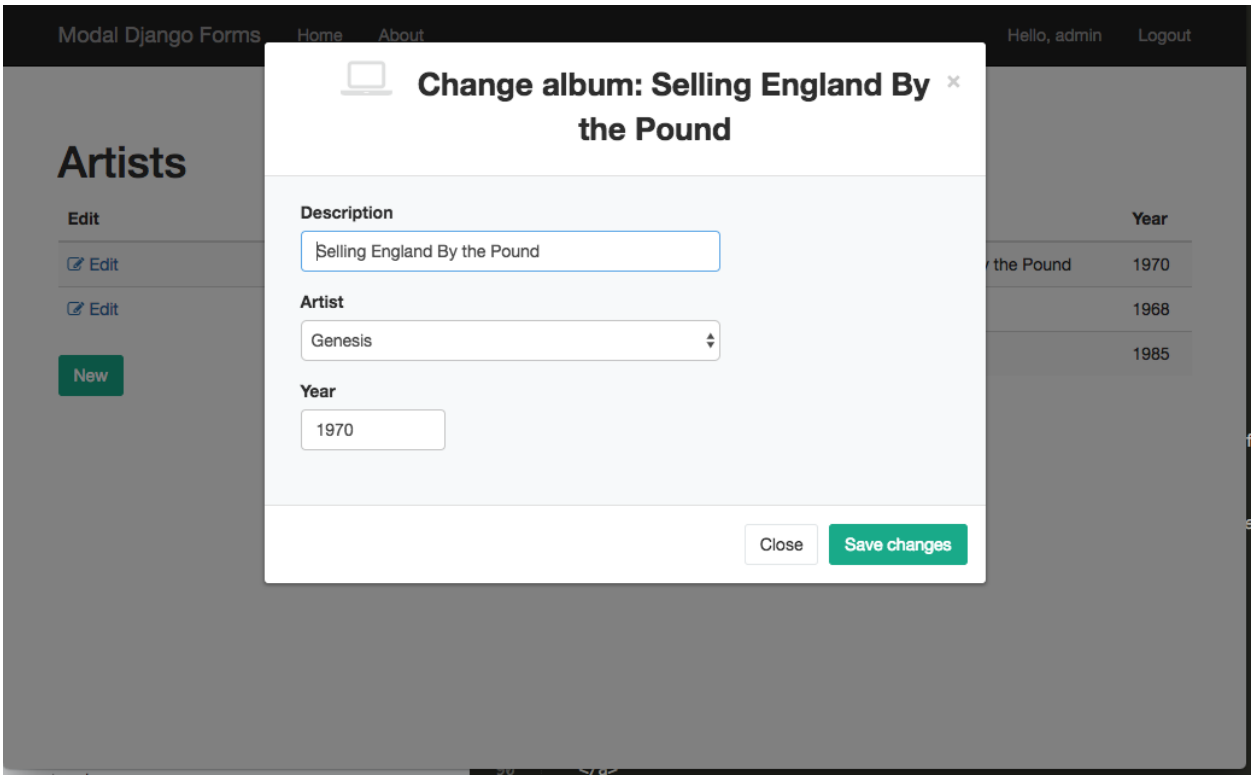


1.8 Front-end generic helpers to work with any Model

Abbiamo realizzato una generic_edit_view() che provvede a gestire l'editing di un oggetto arbitrario.

Tuttavia, l'introduzione nel front-end di nuovo Model ci costringe a scrivere alcune istruzioni piuttosto ripetitive:

- una ModelForm specifica



- due url per invocare generic_edit_view() nei due casi “add” e “change” selezionando un valore opportuno per il parametro “model_form_class”
- i templates in cui inserire i link richiesti, che saranno presumibilmente analoghi a quelli gia’ realizzati per altri Models.

Se il front-end prevede un numero limitato di Models, e’ ragionevole accettare queste ripetizioni; in caso contrario puo’ essere giustificato affrontare la complicazione di introdurre anche lato front-end soluzioni piu’ generiche.

Note: Check sample code at: (10) Front-end generic helpers to work with any Model

---

1.8.1 Generic end-points for changing and adding an object

Possiamo definire due end-point del tutto generici richiedento come paramteri **app_label** e **model_name** per individuare il Model richiesto; trattandosi di stringhe, possono essere inserite in un url:

file *frontend/urls.py*

```
urlpatterns = [
    ...
    # Edit any object
    path('object/<str:app_label>/<str:model_name>/add/', views.edit_object, name=
↪"object-add"),
    path('object/<str:app_label>/<str:model_name>/<uuid:pk>/change/', views.edit_
↪object, name="object-change"),
]
```

dove la nuova view edit_object() si incarica di fornire a generic_edit_view() la ModelForm opportuna:

```
from .forms import get_model_form_class

def edit_object(request, app_label, model_name, pk=None):
    model_form_class = get_model_form_class(app_label, model_name)
    return generic_edit_view(request, model_form_class, pk)
```

Il vero lavoro viene delegato all'utility get_model_form_class() che seleziona la prima ModelForm compatibile con il Model richiesto, oppure ne crea una al volo:

file *frontend/forms.py*

```
import sys
import inspect
from django import forms
from django.apps import apps


def get_model_form_class(app_label, model_name):

    model_form_class = None

    # List all ModelForms in this module
    model_forms = [
        klass
        for name, klass in inspect.getmembers(sys.modules[__name__])
        if inspect.isclass(klass) and issubclass(klass, forms.ModelForm)
    ]

    # Scan ModelForms until we find the right one
    for model_form in model_forms:
        model = model_form._meta.model
        if (model._meta.app_label, model._meta.model_name) == (app_label, model_name):
            return model_form

    # Failing that, build a suitable ModelForm on the fly
    model_class = apps.get_model(app_label, model_name)
    class _ObjectForm(forms.ModelForm):
        class Meta:
            model = model_class
            exclude = []
    return _ObjectForm
```

Il template utilizzato per il rendering della pagina puo' utilizzare i nuovi urls generici, nell'ipotesi che il context ricevuto sia stato annotato allo scopo con la variabile **model**:

```
{# change object #}
data-action="{% url 'frontend:object-change' model|app_label model|model_name object.
↪id %}"

{# add object #}
data-action="{% url 'frontend:object-add' model|app_label model|model_name %}"
```

Sfortunatamente per estrarre **app_label** e **model_name** e altre informazioni accessorie da **model** e' necessario predisporre alcuni semplici template_tags, poiche' l'attributo *_meta* del model non e' direttamente accessibile nel contesto del template:

file *frontend/template_tags/frontend_tags.py*

```
from django import template
from django.urls import reverse

register = template.Library()


@register.filter
def model_verbose_name(model):
    """
    Sample usage:
        {{model|model_name}}
    """
    return model._meta.verbose_name


@register.filter
def model_verbose_name_plural(model):
    """
    Sample usage:
        {{model|model_name}}
    """
    return model._meta.verbose_name_plural


@register.filter
def model_name(model):
    """
    Sample usage:
        {{model|model_name}}
    """
    return model._meta.model_name


@register.filter
def app_label(model):
    """
    Sample usage:
        {{model|app_label}}
    """
    return model._meta.app_label
```

Per maggiore leggibilita' possiamo anche introdurre ulteriori filtri che forniscono direttamente il link "canonico":

```
@register.filter
def add_model_url(model):
    """
    Given a model, return the "canonical" url for adding a new object:

        <a href="{{model|add_model_url}}">add a new object</a>
    """
    return reverse('frontend:object-add', args=(model._meta.app_label, model._meta.
↪model_name))


@register.filter
def change_object_url(object):
    """
```

(continues on next page)

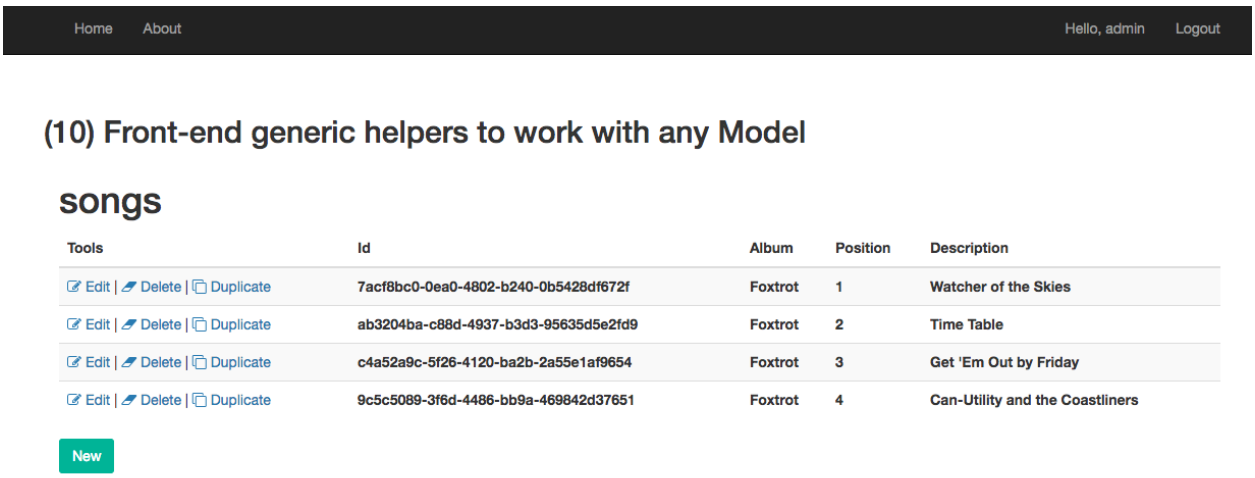
(continued from previous page)

```
    Given an object, returns the "canonical" url for object editing:

        <a href="{{object|change_object_url}}">change this object</a>
    """
    model = object.__class__
    return reverse('frontend:object-change', args=(model._meta.app_label, model._meta.
↪model_name, object.id))


@register.filter
def change_model_url(model, object_id):
    """
    Given a model and an object id, returns the "canonical" url for object editing:

        <a href="{{model|change_model_url:object.id}}">change this object</a>
    """
    return reverse('frontend:object-change', args=(model._meta.app_label, model._meta.
↪model_name, object_id))
```

e riscrivere il template piu' semplicemente come segue:

```
{# add object #}
data-action="{{model|add_model_url}}"

{# change object #}
data-action="{{model|change_model_url:object.id}}"

oppure:

{# change object #}
data-action="{{object|change_object_url}}"
```

1.8.2 Deleting an object



Associamo all'url:

```
path('object/<str:app_label>/<str:model_name>/<uuid:pk>/delete/', views.delete_object,
↪ name="object-delete"),
```

una view responsabile di eseguire la cancellazione di un generico oggetto:

```
################################################################################
# Deleting an object

def delete_object(request, app_label, model_name, pk):

    required_permission = '%s.delete_%s' % (app_label, model_name)
    if not request.user.is_authenticated or not request.user.has_perm(required_
↪permission):
        raise PermissionDenied

    model = apps.get_model(app_label, model_name)
    object = get_object_by_uuid_or_404(model, pk)
    object_id = object.id
    object.delete()

    return HttpResponse(object_id)
```

Verra' invocata via Ajax da una funzione javascript accessoria che chiede preventivamente la conferma dell'utente:

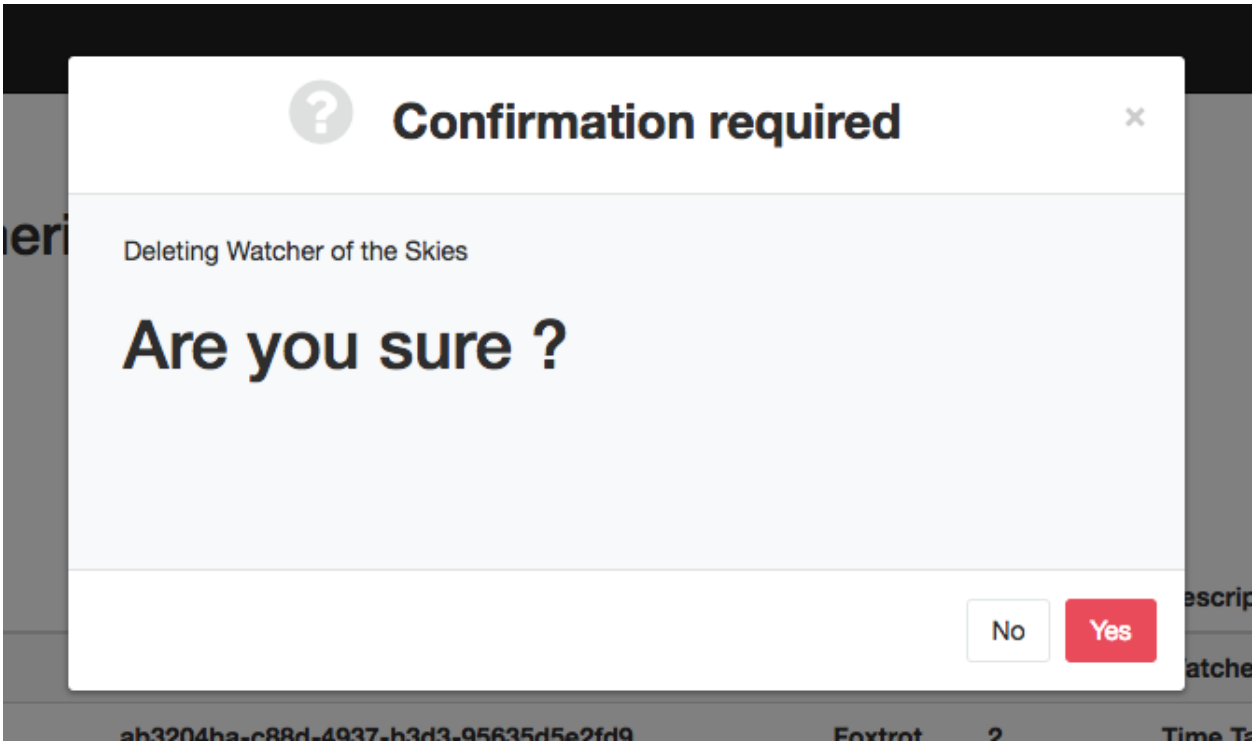
```
function confirmRemoteAction(url, title, afterObjectDeleteCallback) {
    var modal = $('#modal_confirm');
    modal.find('.modal-title').text(title);
    modal.find('.btn-yes').off().on('click', function() {
        // User selected "Yes", so proceed with remote call
        $.ajax({
            type: 'GET',
            url: url
        }).done(function(data) {
            if (afterObjectDeleteCallback) {
                afterObjectDeleteCallback(data);
            }
        }).fail(function(jqXHR, textStatus, errorThrown) {
            alert('SERVER ERROR: ' + errorThrown);
        });
    });
    modal.modal('show');
}
```

e quindi, nel template:

```
<a href=""
   onclick="confirmRemoteAction('{{object|delete_object_url}}', 'Deleting {{object}}',
↪ afterObjectDelete); return false;">
    <i class="fa fa-eraser"></i> Delete
</a>
```

La callback opzionale **afterObjectDelete()** viene invocata dopo l'effettiva cancellazione, ricevendo l'id dell'oggetto eliminato.

Nel progetto d'esempio, per semplicita', si limita a ricaricare la pagina, mentre in casi applicativi reali verra' convenientemente utilizzata per aggiornare "chirurgicamente" la pagina esistente.



1.8.3 Cloning an object

La possibilita' di duplicare un oggetto esistente, normalmente non prevista dalla interfaccia di amministrazione di Django, e' molto interessante in applicazioni fortemente orientate alla gestione dati, perche' consente all'utilizzatore un notevole risparmio di tempo quando e' richiesto l'interimento di dati ripetitivi.

In sostanza consente di fornire caso per caso valori di default "opportuni" in modo arbitrario.

Possiamo predisporre una view che duplica un oggetto esistente analogamente a quanto gia' fatto per la cancellazione:

```
##########################################################################
# Cloning an object

def clone_object(request, app_label, model_name, pk):

    required_permission = '%s.add_%s' % (app_label, model_name)
    if not request.user.is_authenticated or not request.user.has_perm(required_
↪permission):
        raise PermissionDenied

    model = apps.get_model(app_label, model_name)
    object = get_object_by_uuid_or_404(model, pk)
    new_object = object.clone(request)
    return HttpResponse(new_object.id)
```

Qui stiamo supponendo che il Model metta a disposizione un opportuno metodo **clone()**; conviene delegare questa attivita' allo specifico Model, che si preoccupera' di gestire opportunamente le proprie eventuali relazioni M2M, ed eseguire eventuali elaborazioni accessorie (rinumerazione del campo *position*, etc):

```
class Song(BaseModel):
```

(continues on next page)

(continued from previous page)

```
    ...

    def clone(self, request=None):

        if request and not request.user.has_perm('backend.add_song'):
            raise PermissionDenied

        obj = Song.objects.get(id=self.id)
        obj.pk = uuid.uuid4()
        obj.description = increment_revision(self.description)
        obj.save()
        return obj
```

Warning: Supply a default generic clone procedure when the Model doesn't provide it's own

Per duplicare anche le eventuali relazioni, vedere:

https://docs.djangoproject.com/en/1.10/topics/db/queries/#copying-model-instances

La stessa funzione javascript confirmRemoteAction() utilizzata in precedenza puo' essere invocata anche qui per richiedere la conferma dell'utente prima dell'esecuzione:

```
<a href=""
   onclick="confirmRemoteAction('{{object|clone_object_url}}', 'Duplicating {{object}}
↪', afterObjectClone); return false;">
    <i class="fa fa-clone"></i> Duplicate
</a>
```

La callback **afterObjectClone()** riceve l'id dell'oggetto creato.

1.8.4 Checking user permissions

Tutte le viste utilizzate sin qui per manipolare i Models sono gia' protette in termine di permissions accordate all'utente; in caso di violazione, viene lanciata l'eccezione PermissionDenied, e il front-end visualizza un server error.

In alternativa, possiamo inibire o nascondere i controlli di editing dalla pagina quanto l'utente loggato non e' autorizzato alle operazioni.

Il seguente template tag consente di verificare se l'utente e' autorizzato o meno ad eseguire le azioni:

- add
- change
- delete
- view (Django >= 2.1 only)

```
@register.simple_tag(takes_context=True)
def testhasperm(context, model, action):
    """
    Returns True iif the user have the specified permission over the model.
    For 'model', we accept either a Model class, or a string formatted as "app_label.
↪model_name".
    """
```

(continues on next page)

(continued from previous page)

```
    user = context['request'].user
    if isinstance(model, str):
        app_label, model_name = model.split('.')
    else:
        app_label = model._meta.app_label
        model_name = model._meta.model_name
    required_permission = '%s.%s_%s' % (app_label, action, model_name)
    return user.is_authenticated and user.has_perm(required_permission)
```

e puo' essere utilizzata assegnato il valore calcolato ad una variabile per i successivi test:

```
{% testhasperm model 'view' as can_view_objects %}
{% if not can_view_objects %}
    <h2>Sorry, you have no permission to view these objects</h2>
{% endif %}
```

Un'altra possibilita' e' quella di utilizzare un template tag "ishasperm" per condizionare l'inclusione del controllo:

```
{% ifhasperm model 'change' %}
    <a href=""
       data-action="{{model|change_model_url:object.id}}"
       onclick="openModalDialogWithForm(event, '#modal_generic', null,␣
↪afterObjectChangeSuccess); return false;"
       data-title="Update {{ model|model_verbose_name }}: {{ object }}">
        <i class="fa fa-edit"></i> Edit
    </a>
    |
{% endifhasperm %}
```

dove:

```
@register.tag
def ifhasperm(parser, token):
    """
    Check user permission over specified model.
    (You can specify either a model or an object).
    """

    # Separating the tag name from the parameters
    try:
        tag, model, action = token.contents.split()
    except (ValueError, TypeError):
        raise template.TemplateSyntaxError(
            "'%s' tag takes three parameters" % tag)

    default_states = ['ifhasperm', 'else']
    end_tag = 'endifhasperm'

    # Place to store the states and their values
    states = {}

    # Let's iterate over our context and find our tokens
    while token.contents != end_tag:
        current = token.contents
        states[current.split()[0]] = parser.parse(default_states + [end_tag])
        token = parser.next_token()
```

(continues on next page)

(continued from previous page)

```
    model_var = parser.compile_filter(model)
    action_var = parser.compile_filter(action)
    return CheckPermNode(states, model_var, action_var)


class CheckPermNode(template.Node):
    def __init__(self, states, model_var, action_var):
        self.states = states
        self.model_var = model_var
        self.action_var = action_var

    def render(self, context):

        # Resolving variables passed by the user
        model = self.model_var.resolve(context)
        action = self.action_var.resolve(context)

        # Check user permission
        if testhasperm(context, model, action):
            html = self.states['ifhasperm'].render(context)
        else:
            html = self.states['else'].render(context) if 'else' in self.states else '
↪'

        return html
```

1.9 Possible enhancements

The Django admin offers a rich environment for data editing; we might further evolve our code to provide similar functionalities:

Dynamic default values accept optional "initial" values from the url parameters in the "add new" view; useful for example to set a parent relation

Fieldsets Check this for inspiration: https://schinckel.net/2013/06/14/django-fieldsets/

Filtered lookup of related models As ModelAdmin does with **formfield_for_foreignkey** and **formfield_for_manytomany**

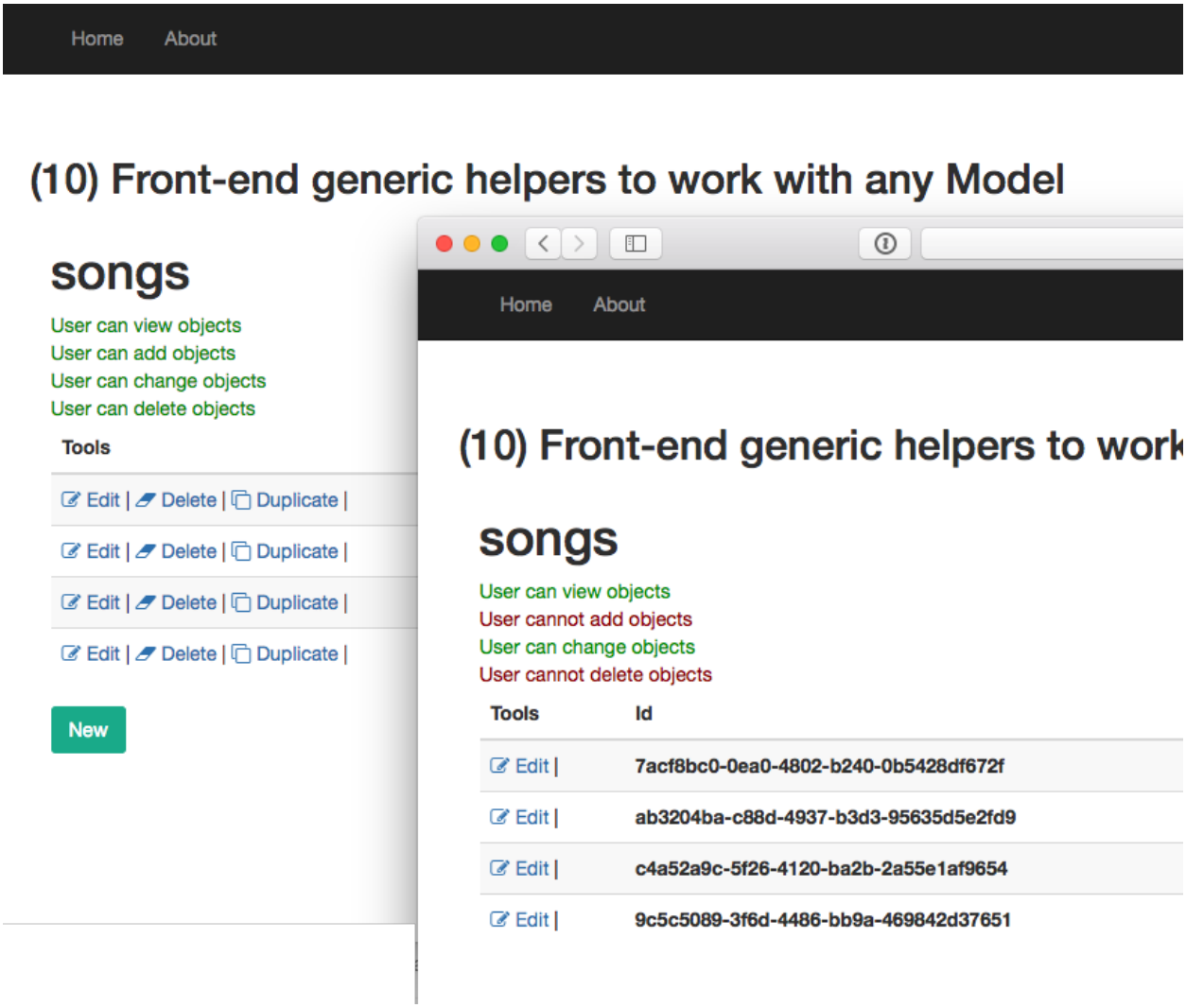
Support for raw_id_fields Check https://github.com/lincolnloop/django-dynamic-raw-id/ and https://www.abidibo.net/blog/2015/02/06/pretty-raw_id_fields-django-salmonella-and-django-grappelli/

Support for inlines ...

Support for autocompletion ...

Minor issues:

- Add a localized datepicker in the examples
- Add a ModelMultipleChoiceField (with multiSelect javascript widget) example in the sample project
- Accept and optional "next" parameter for redirection after successfull form submission (for standalone pages only)


Home
About
(10) Front-end generic helpers to work with any Model
songs
User can view objects
User can add objects
User can change objects
User can delete objects
Tools
Edit | Delete | Duplicate |
Edit | Delete | Duplicate |
Edit | Delete | Duplicate |
Edit | Delete | Duplicate |
New
Home
About
(10) Front-end generic helpers to work
songs
User can view objects
User cannot add objects
User can change objects
User cannot delete objects
Tools
Id
Edit |
7acf8bc0-0ea0-4802-b240-0b5428df672f
Edit |
ab3204ba-c88d-4937-b3d3-95635d5e2fd9
Edit |
c4a52a9c-5f26-4120-ba2b-2a55e1af9654
Edit |
9c5c5089-3f6d-4486-bb9a-469842d37651

1.10 References

- Use Django's Class-Based Views with Bootstrap Modals
- Ajax form submission with Django and Bootstrap Modals
- Modal django forms with bootstrap 4

1.11 Contributing

Thank you for posting comments, suggestions and pull requests to:

https://github.com/morlandi/editing-django-models-in-the-frontend

CHAPTER 2

Sample code

A sample Django project which uses all the techniques discussed here is available on Github:

https://github.com/morlandi/editing-django-models-in-the-frontend

CHAPTER 3

Search docs

- search