Modal Django Forms Documentation

Mario Orlandi

Sep 14, 2018

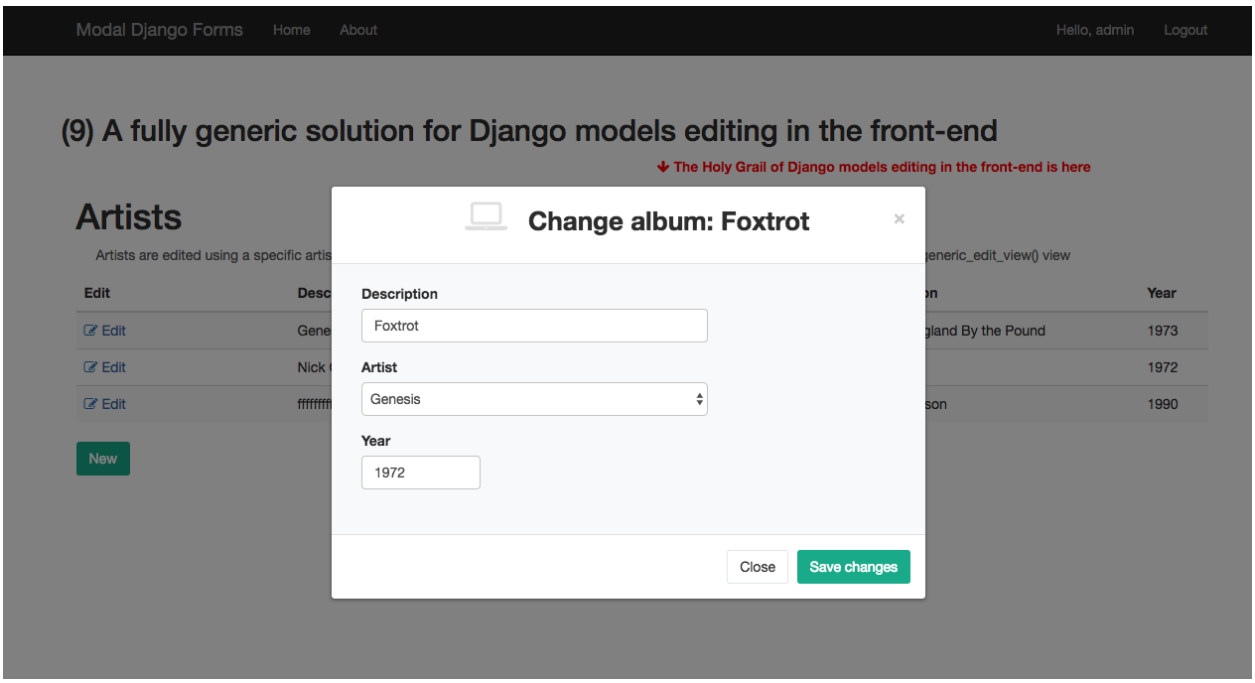
Contents

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();
}
```

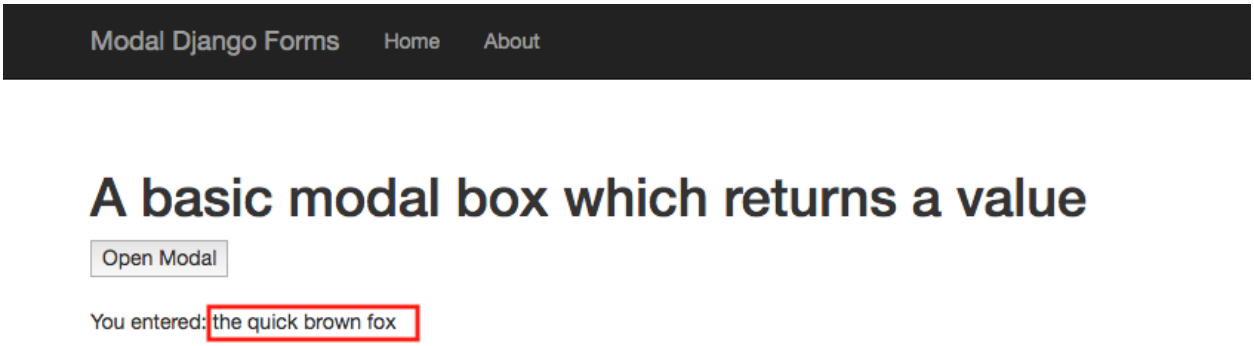
1.1.4 Bootstrap 3 modal plugin

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

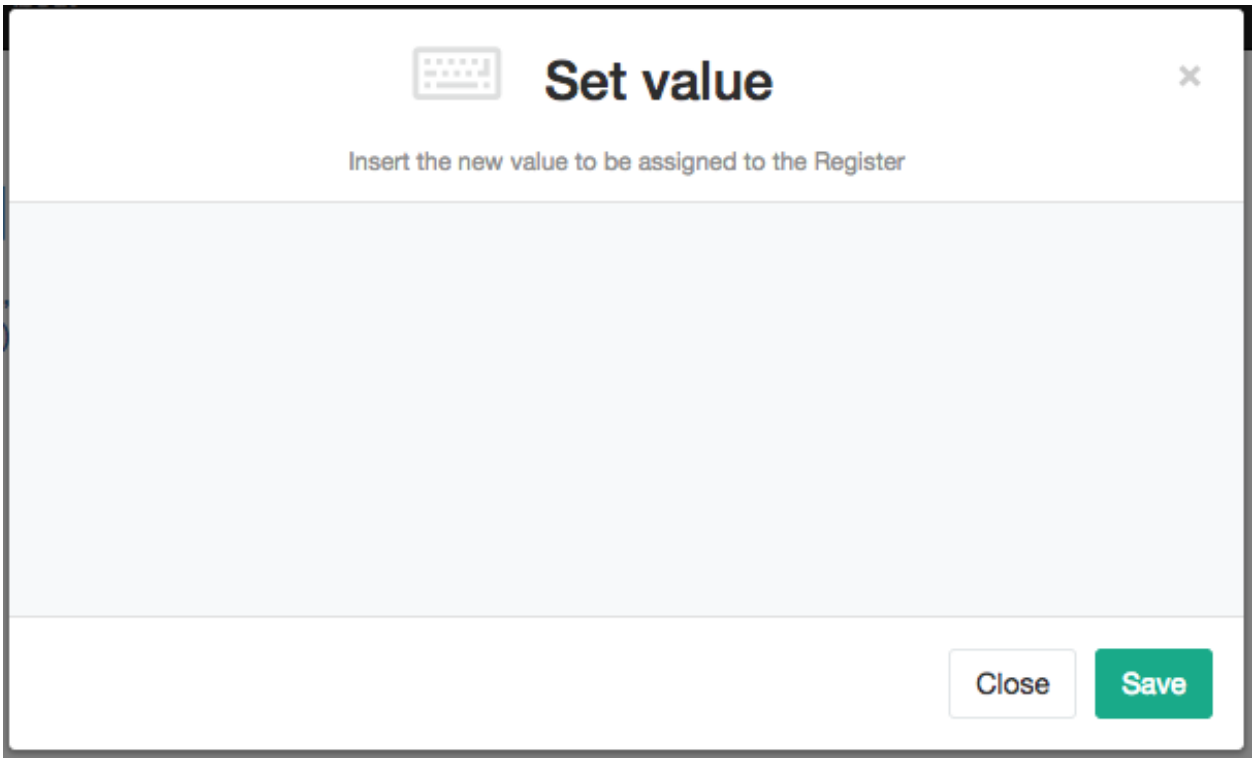
Questo e' sufficiente nei casi in cui il template contenga gia' tutti gli elementi richiesti; ci sono pero' buone possibilita' che un'unica "generica" dialog sia riutilizzabile in diverse circostanze (o forse ovunque) pur di fornire un minimo di informazioni accessorie:

```
<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"></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, element) {
        /*
            You can customize the modal layout specifing optional "data" attributes
```

(continues on next page)



(continued from previous page)

```
        in the element (either <a> or <button>) which triggered the event;
        "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"></i> Open generic modal (no contents)
        </a>
    */
    var 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);
    modal.find('.modal-subtitle').text(subtitle);
    modal.find('.modal-header .title-wrapper i').attr('class', icon_class);
```

(continues on next page)

(continued from previous page)

```
        modal.find('.modal-footer .btn-save').text(button_save_label);
        modal.find('.modal-body').html('');

        return modal;
    }

    function openModalDialog(event, element) {
        var modal = initModalDialog(event, 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' %}
```

(continues on next page)

(continued from previous page)

```
    ...
{% 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:

```
<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.

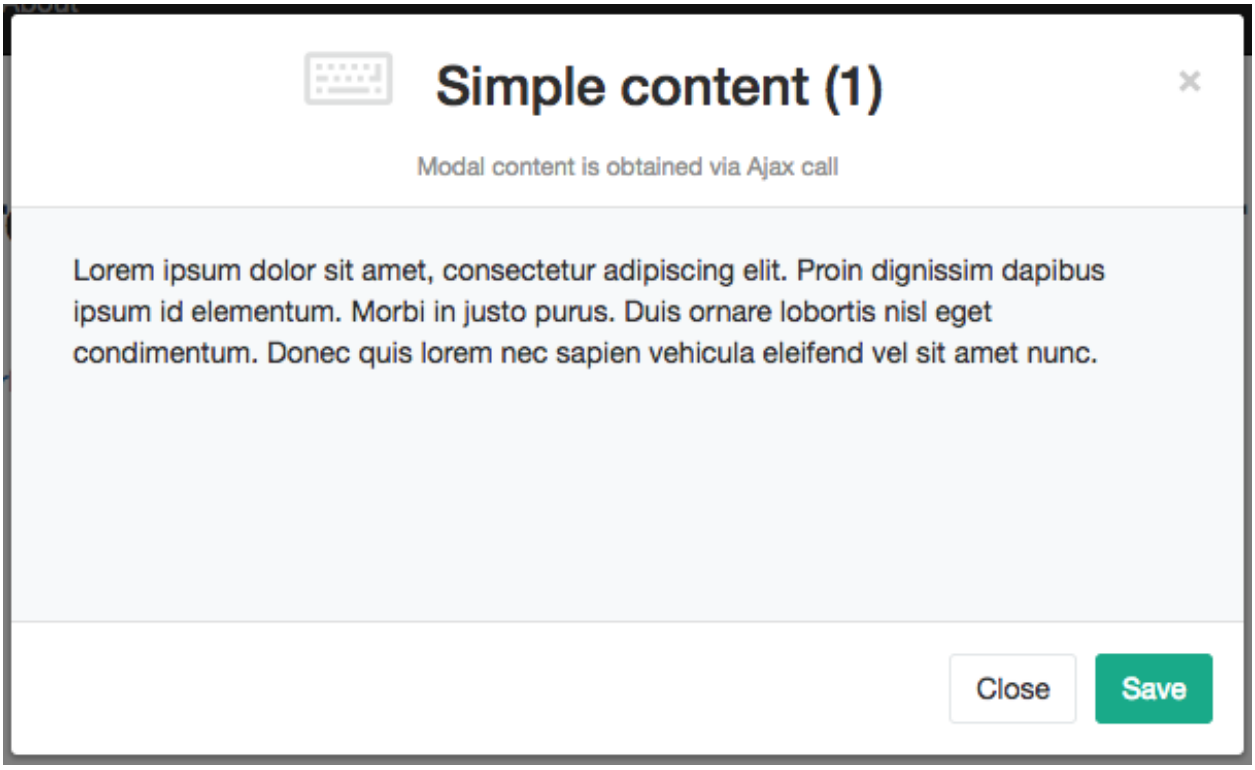
Un limite di questa semplice soluzione e' che non siamo in grado di rilevare eventuali errori del server, e quindi in caso di errore la dialog verrebbe comunque aperta (con contenuto vuoto).

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');
```

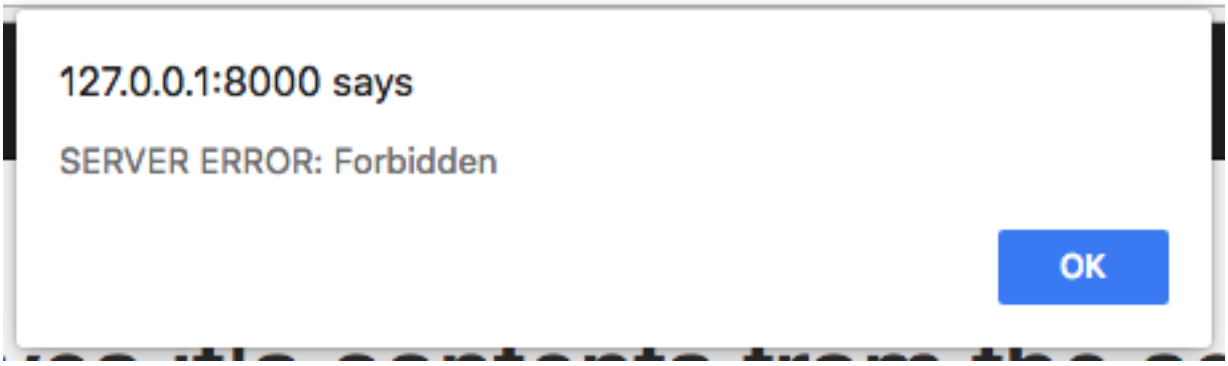
(continues on next page)



(continued from previous page)

```
        }).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

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**:

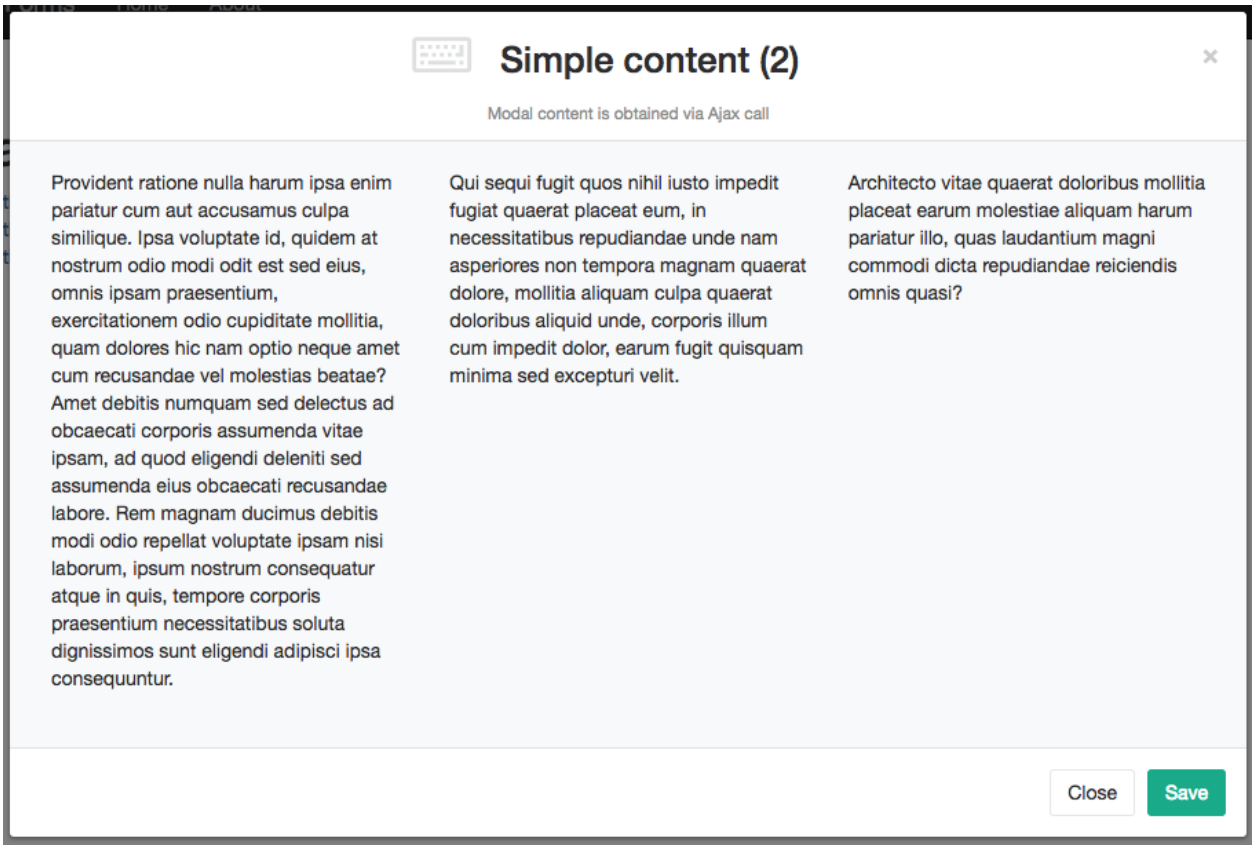


Fig. 3: Modal dialog

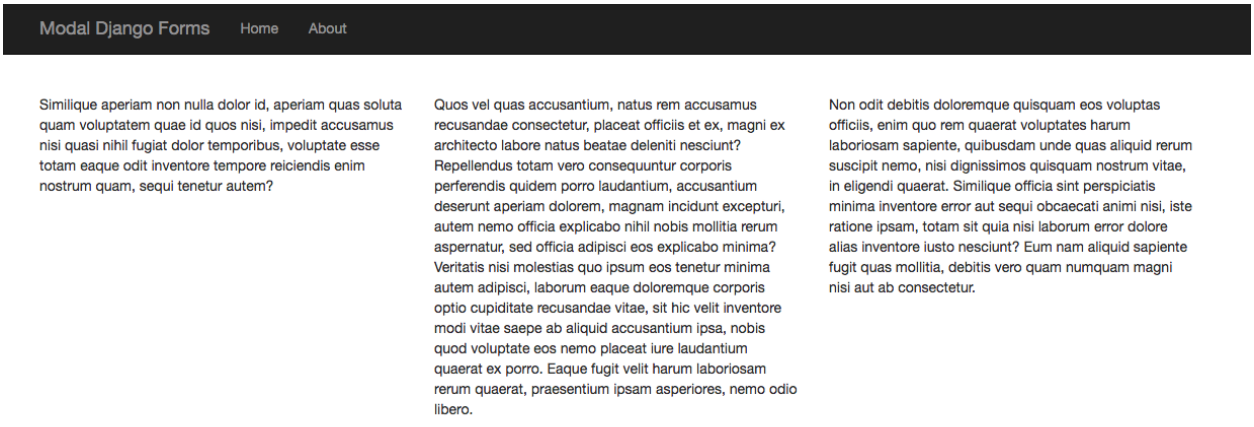


Fig. 4: Same content in a standalone page

- 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) just calling a single helper function **formAjaxSubmit()** which I'll explain in detail later.

```
<script language="javascript">

    function openMyModal(event) {
        var modal = initModalDialog(event, '#modal_generic');
        var url = $(event.target).attr('href');
        $.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>
```

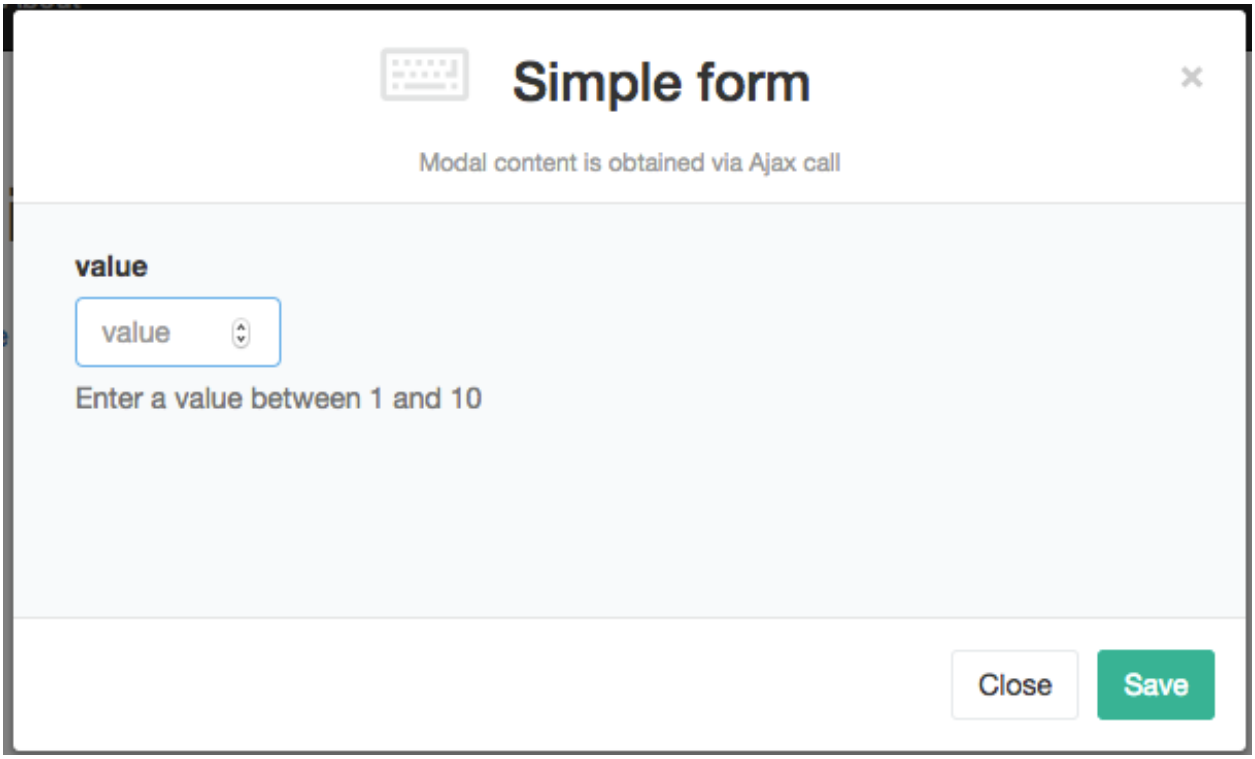


Fig. 5: A form in the modal dialog

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

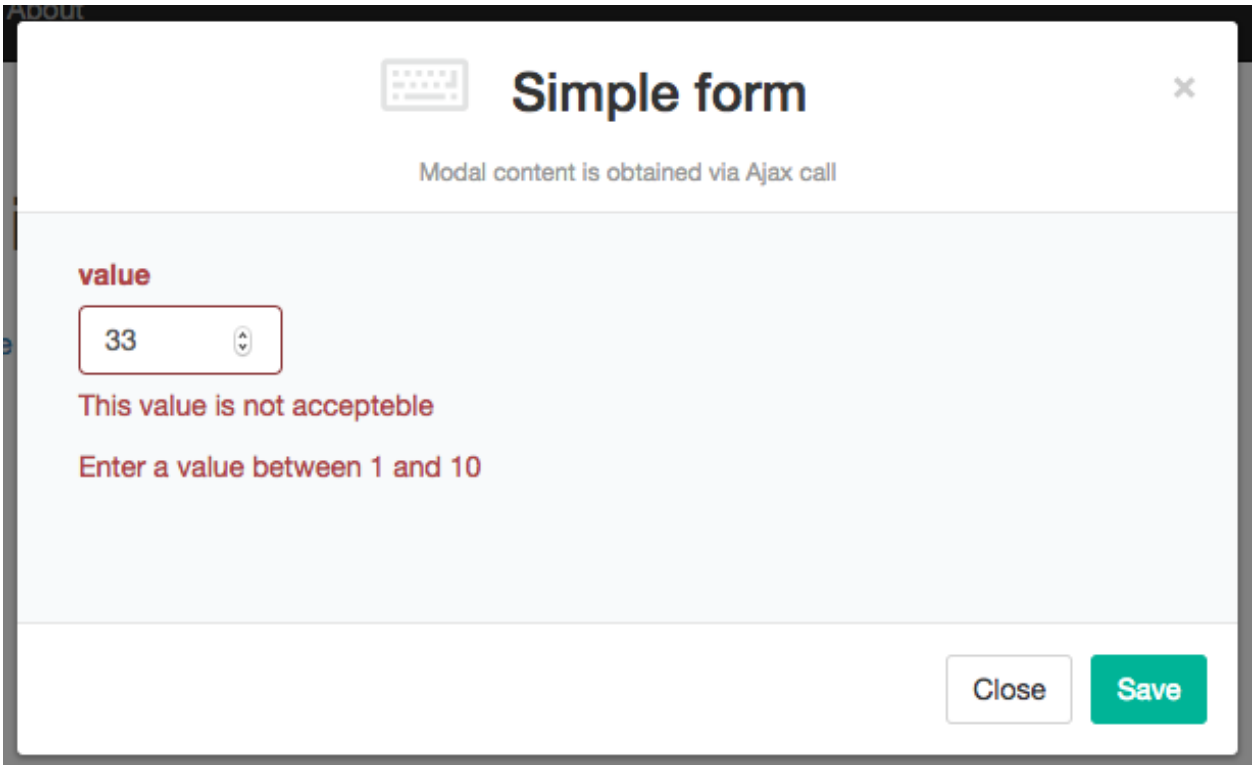
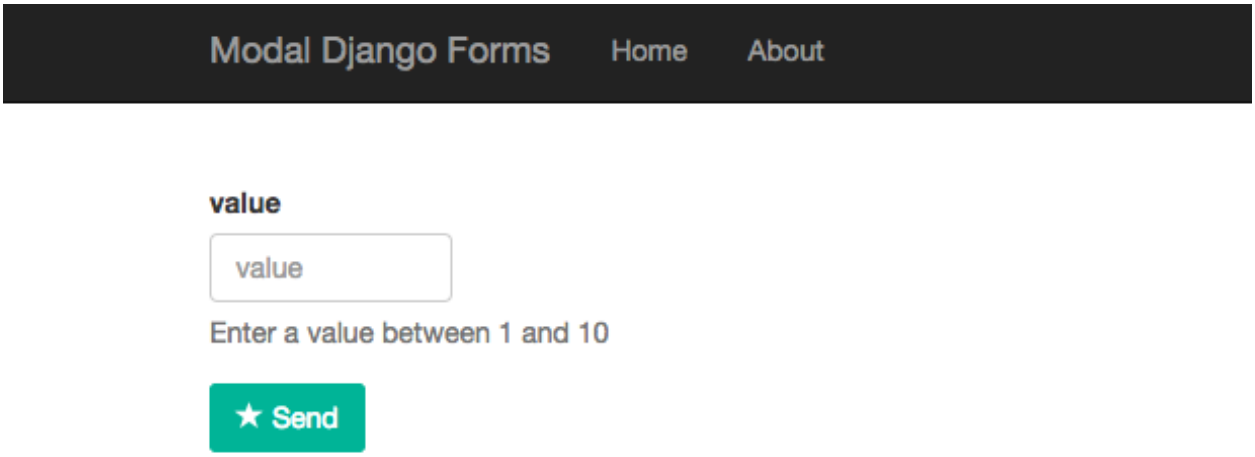


Fig. 6: When form did not validate, we keep the dialog open



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');

        // 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); }

        setTimeout(function() {
            modal.find('form input:visible').first().focus();
        }, 1000);

        // 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(),
                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
                    $(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');
```

(continues on next page)

(continued from previous page)

```
                        if (cbAfterSuccess) { cbAfterSuccess(modal); }
                    }
                },
                error: function(xhr, ajaxOptions, 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(),
        ...
```

Note tha if the form specifies an action, we use it as end-point of the ajax call; if not, we're using the same view for both rendering and form processing, so 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.

We also need to cope with 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:

```
// 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();
    });
}
```

During content loading, we add a "loading" class to the dialog header, making a spinner icon visible.

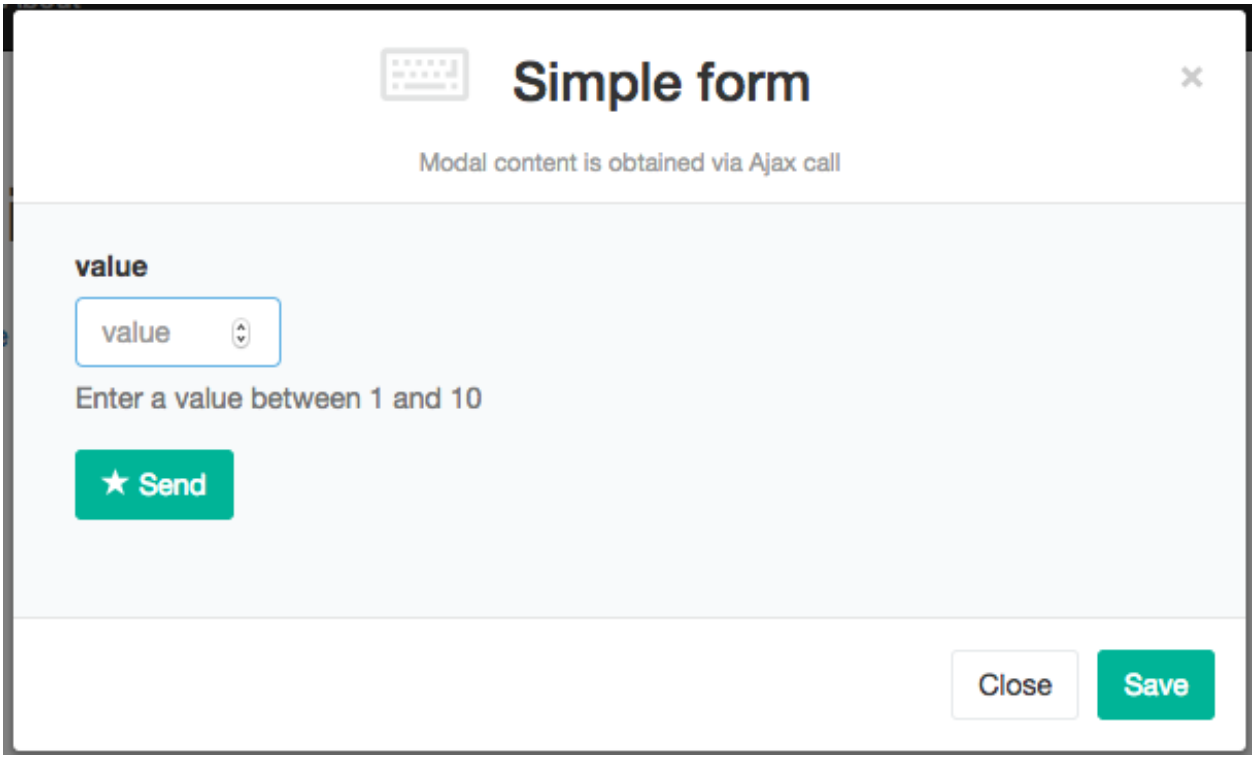


Fig. 7: Can we hide the "Send" button and use the "Save" button from the footer instead ?

1.4.3 Optional callbacks

- cbAfterLoad: called every time new content has been loaded; you can use it to bind more form controls
- 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) {
    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' in the sample project

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()

    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.

On successful creation, we might want to update the user interface; in the example, for simplicity, we just reload the entire page, but also display 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.

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
```

(continues on next page)

(continued from previous page)

```
        fields = [
            'description',
            'notes',
        ]
```

Finally, here's the object id retrival aftee 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 other models

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 required.

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):

    # Retrieve object
    if pk is None:
        # "Add" mode
        object = None
        required_permission = 'backend.add_artist'
    else:
```

(continues on next page)

(continued from previous page)

```
        # 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 = ArtistUpdateForm(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 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 = ArtistUpdateForm(instance=object)

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

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

The urls are organized as follows:

```
urlpatterns = [
    ...
    path('artist/add/', views.artist_edit, {'pk': None, }, 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
```

(continues on next page)

(continued from previous page)

```
        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:

```
<script language="javascript">

    function onObjectEdit(event, cbAfterLoad, cbAfterSuccess) {
        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, 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, pk, model_form_class):

    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:
```

(continues on next page)

(continued from previous page)

```
        # "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)

    # 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,
    })
```

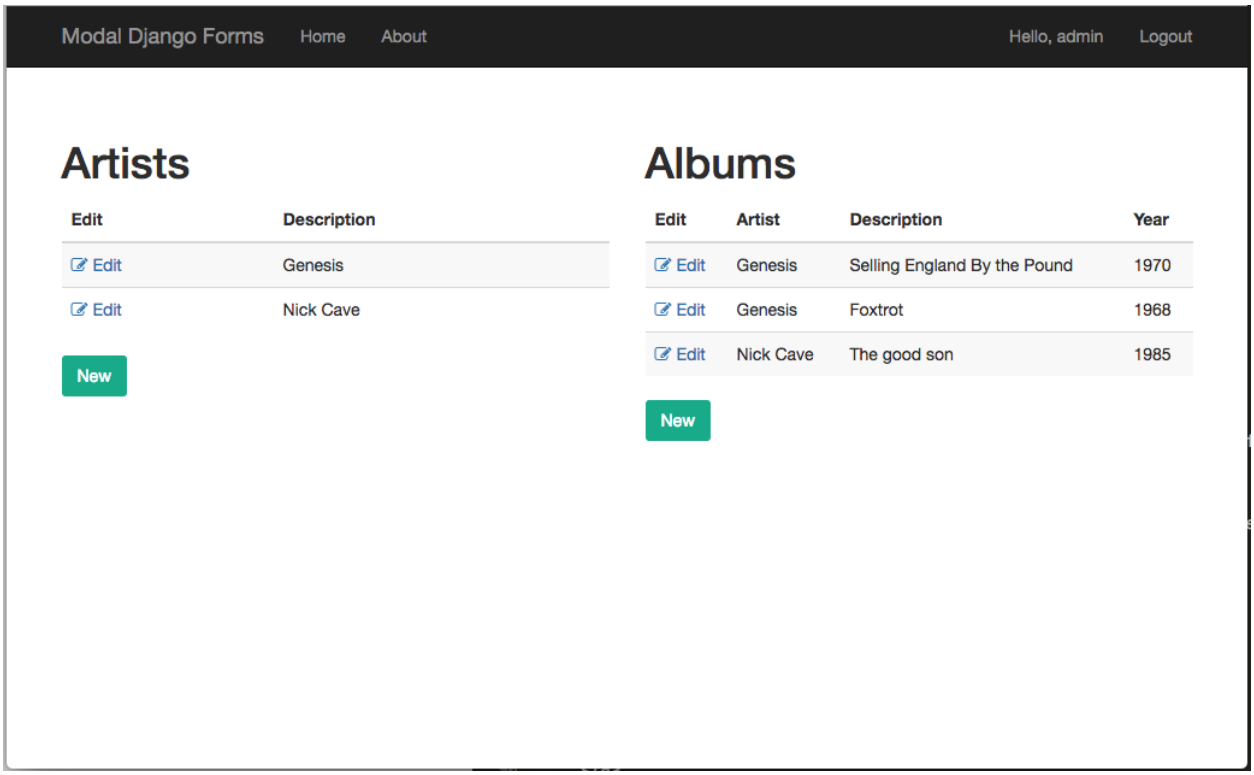
Adding a ModelForm specification at run-time is all what we need; from it, we deduct the Model that's it.

Just remember to supply a ModelForm in the urls and you've done:

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

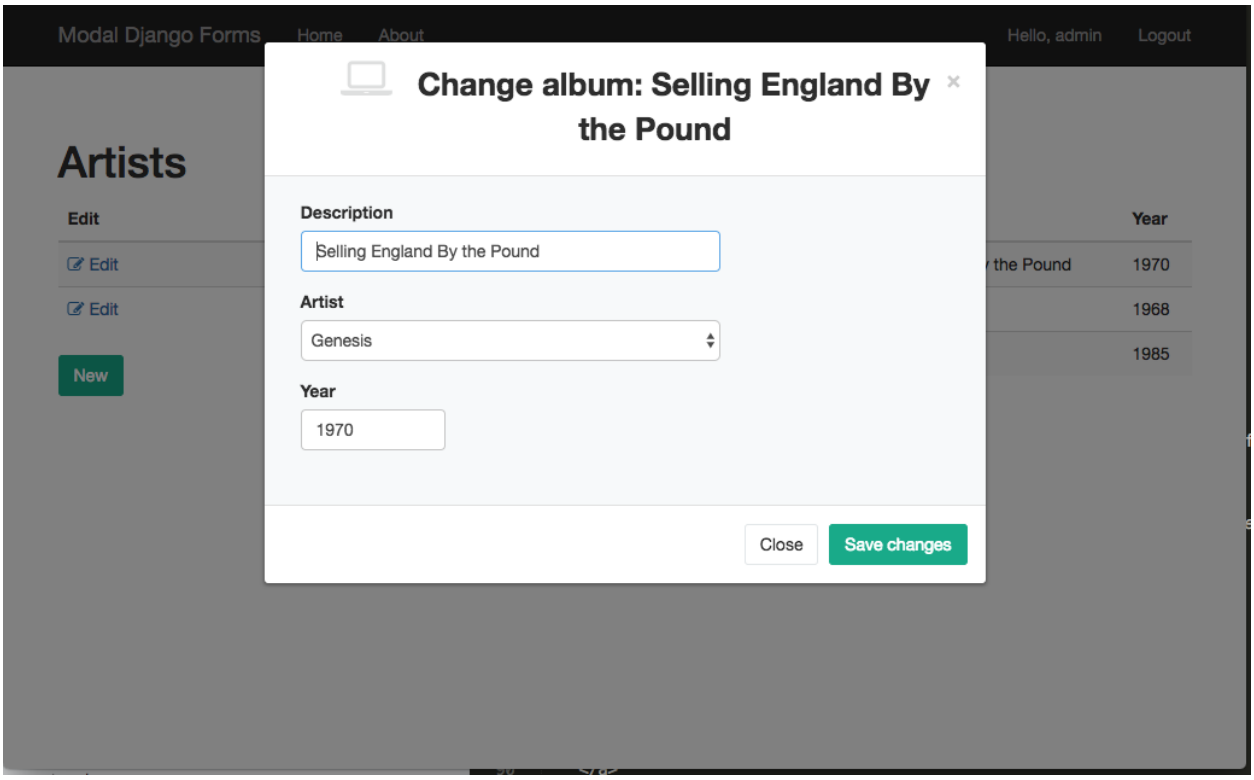
In the page template, we bind the links to the views as follows:

```
<!-- Change -->
<a href=""
   data-action="{% url 'frontend:album-change' row.id %}"
   onclick="onObjectEdit(event, 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="onObjectEdit(event, null, afterObjectEditSuccess); return false;"
    type="button"class="btn btn-primary">
    New
</button>
```



1.8 References

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



1.9 Possible enhancements

Django admin offers a rich environment for data editing; we might evolve the project to provide similar functionalities:

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/

Minor issues:

- Add a localized datepicker in the examples
- Give an example for object deletion (upon confirmation)
- Add a ModelMultipleChoiceField and multiSelect example in the sample project
- Accept and optional "next" parameter for redirection after successfull form submission (for standalone pages)

CHAPTER 2

Indices and tables

- genindex
- modindex
- search