djangocms-frontend Documentation

Release 0.9.1

Fabian Braun

May 05, 2022

CONTENTS

DJANGOCMS-FRONTEND

djangocms-frontend is a plugin bundle based on djangocms_bootstrap5. Its objective is to provide a set of popular frontend components independent of the currently used frontend framework such as Bootstrap, or its specific version.

KEY FEATURES

- Support of Bootstrap 5.

- **Separation of plugins from css framework**, i.e., no need to rebuild you site's plugin tree if css framework is changed in the future, e.g., from Bootstrap 5 to a future version.

- **New link plugin** allowing to link to internal pages provided by other applications, such as djangocms-blog.

- **Nice and well-arranged admin frontend** of djangocms-bootstrap4

- Management command to **migrate from djangocms-bootstrap4**. This command automatically migrates all djangocms-bootstrap4 plugins to djangocms-frontend.

- **Extensible** within the project and with separate project (e.g., a theme app)

- **Accordion** plugin and simple **forms** plugin w/ Bootstrap-styled forms on your cms page.

DESCRIPTION

The plugins are framework agnostic and the framework can be changed by adapting your project's settings. Also, it is designed to avoid having to rebuild your CMS plugin tree when upgrading e.g. from one version of your frontend framework to the next.

django CMS Frontend uses django-entangled by Jacob Rief to avoid bloating your project's database with css framework-dependent tables. Instead all design parameters are stored in a common JSON field and future releases of improved frontend features will not require to rebuild your full plugin tree.

The link plugin has been rewritten to not allow internal links to other CMS pages, but also to other django models such as, e.g., posts of djangocms-blog.

djangocms-frontend provides a set of plugins to structure your layout. This includes three basic elements

The grid The grid is the basis for responsive page design. It splits the page into containers, rows and columns. Depending on the device, columns are shown next to each other (larger screens) or one below the other (smaller screens).

Components Components structure information on your site by giving them an easy to grasp and easy to use look. Alerts or cards are examples of components.

Forms (work in progress) Finally, djangocms-frontend lets you display forms in a nice way. Also, it handles form submit actions, validation etc. Forms can be easily structured using fieldsets known from django's admin app. But djangocms-frontend also works with third-party apps like django-crispy-forms for even more complex layouts.

3.1 Contents

3.1.1 Getting started

Installation

Install package

For a manual install run `pip install djangocms-frontend`

Alternatively, add the following line to your project's `requirements.txt`:

```
djangocms-frontend
```

Make apps available to your django project

Add the following entries to your `INSTALLED_APPS`:

```
'djangocms_icon',
'easy_thumbnails',
'djangocms_frontend',
'djangocms_frontend.contrib.accordion',
'djangocms_frontend.contrib.alert',
'djangocms_frontend.contrib.badge',
'djangocms_frontend.contrib.card',
'djangocms_frontend.contrib.carousel',
'djangocms_frontend.contrib.collapse',
'djangocms_frontend.contrib.content',
'djangocms_frontend.contrib.grid',
'djangocms_frontend.contrib.image',
'djangocms_frontend.contrib.jumbotron',
'djangocms_frontend.contrib.link',
'djangocms_frontend.contrib.listgroup',
'djangocms_frontend.contrib.media',
'djangocms_frontend.contrib.tabs',
'djangocms_frontend.contrib.utilities',
```

Note: Using Django 2.2 to 3.1

You will need to add `django-jsonfield-backport` to your `requirements.txt` and add `"django_jsonfield_backport"` to your `INSTALLED_APPS`.

Create necessary database table

Finally, run `python manage.py migrate`

djangocms-frontend now is ready for use!

Adding styles and javascript manually

django CMS frontend **does not** automatically add the styles or javascript files to your frontend, these need to be added at your discretion.

Out of the box, **djangocms-frontend** is configured to work with Bootstrap 5. Styles should be added to your <head> section of your project template (often called `base.html`). Javascript should be added at the end of the <body> section or your template. For illustration and an easier start, **djangocms-frontend** comes with example templates.

Using example templates of djangocms-frontend

djangocms-frontend comes with example templates. The simplest way to activate Bootstrap 5 is by using the following base template (`base.html`)

```
{% extends "bootstrap5/base.html" %}
{% block brand %}<a href="/">My Site</a>{% endblock %}
```

Note: We recommend developing your own `base.html` for your projects. The example templates load CSS and JS files from a CDN. Good reasons to do so are

- **djangocms-frontend** does not contain CSS or JS files from Bootstrap or any other framework for that matter. The example templates load CSS and JS from a CDN.

- It is considered safer to host CSS and JS files yourself. Otherwise you do not have control over the CSS and/or JS that is delivered.

- It is a common practice to customize at least the CSS part, e.g. with brand colors.

The example template is customisable by a set of template blocks:

{% block title %} Renders the page title. Defaults to `{% page_attribute "page_title" %}`

{% block content %} Here goes the main content of the page. The default setup is a `<section>` with a placeholder called "Page Content" and a `<footer>` with a static placeholder (identical on all pages) called "Footer":

```
{% block content %}
    <section>
        {% placeholder "Page Content" %}
    </section> 
    <footer>
        {% static_placeholder "Footer" %}
    </footer>
{% endblock content %}
```

{% block navbar %} This block renders a navigation bar using the Bootstrap 5 `navbar` classes and django CMS' menu system. If you need to add additional navigation on the right hand side of the nav bar populate the block `searchbar` (which can include a search function but does not have to). Also, the block `brand` is rendered in the navigation bar.

{% block base_css %} Loads the framework's CSS. Replace this block if you prefer to include your the CSS from your server.

{% block base_js %} Loads the framework's JS. Replace this block if you prefer to include your the JS from your server. JS is loaded **before** `{% render_block 'js' %}`.

{% block end_js %} Loads additional JS at the end of the page. Currently empty. This block is loaded **after** `{% render_block 'js' %}`.

{% block bottom_css %} Additional CSS placed just before the end of the <body>. Currently empty.

{% block meta %} Contains the meta description of the page. Defaults to:

```
<meta name="description" content="{% page_attribute meta_description %}"/>
<meta property="og:type" content="website"/>
<meta property="og:title" content="{% page_attribute "page_title" %}"/>
<meta property="og:description" content="{% page_attribute meta_description %}"/>
```

{% block canonical_url %} Contains the canonical url of the page. Defaults to:

```
<link rel="canonical" href="{{ request.build_absolute_uri }}"/>
<meta property="og:url" content="{{ request.build_absolute_uri }}"/>
```

Granting rights

If you have restricted rights for users our groups in your projects make sure that editors have the right to to add, change, delete, and - of course - view instances of all djangocms_frontend UI items:

- Accordion
- Alert
- Badge
- Card
- Carousel
- Collapse
- Content
- Forms
- Grid
- Image
- Jumbotron
- Link
- Listgroup
- Media
- Tabs
- Utilities

Otherwise the plugins will not be editable and will not appear in the editors' plugin selection when adding a plugin in the frontend.

Since changing them for each of the plugins manually can become tiresome a management command can support you.

First manually define the permissions for the model `FrontendUIItem` of the app `djangocms_frontend`. **Then** you can synchronize all permissions of the installed UI items by typing

```
./manage.py frontend sync_permissions users
./manage.py frontend sync_permissions groups
```

These commands transfer the permissions for `FrontendUIItem` to all installed plugins for each user or group, respectively.

The first command is only necessary of you define by-user permissions. Depending on the number of users it may take some time.

Attention: If in doubt, please make a backup of your database tables. This operation cannot be undone!

Migrating from djangocms-bootstrap4

In the case you have a running django CMS project using djangocms-bootstrap4 you can try to run the automatic migration process. This process converts all plugin instances of djangocms-bootstrap4 into corresponding djangocms-frontend plugins.

Note: Bootstrap 4 and Bootstrap 5 differ, hence even a successful migration will require manual work to fix differences. The migration command is a support to reduce the amount of manual work. It will not do everything automatically!

The more your existing installation uses the attributes field (found in "advanced settings") the more likely it is, that you will have to do some manual adjustment. While the migration command does adjust settings in the attributes field it cannot know the specifics of your project.

> **Attention:** Please do **back up** your database before you do run the management command!

For this to work, the both the djangocms-frontend **and** the djangocms-bootstrap4 apps need to be included in `INSTALLED_APPS`.

```
./manage.py frontend migrate
```

After you finish the migration you can remove all djangocms-bootstrap4 apps from `INSTALLED_APPS` and you may delete the now empty database tables of djangocms-bootstrap4. You identify them by their name pattern:

```
bootstrap4_alerts_bootstrap4alerts
bootstrap4_badge_bootstrap4badge
...
bootstrap4_utilities_bootstrap4spacing
```

Using djangocms-frontend

djangocms-frontend offers a set of plugins to allow for an easy and clean structure of your CMS contents.

All plugins are listed in the section "Frontend" when adding a plugin to a placeholder:
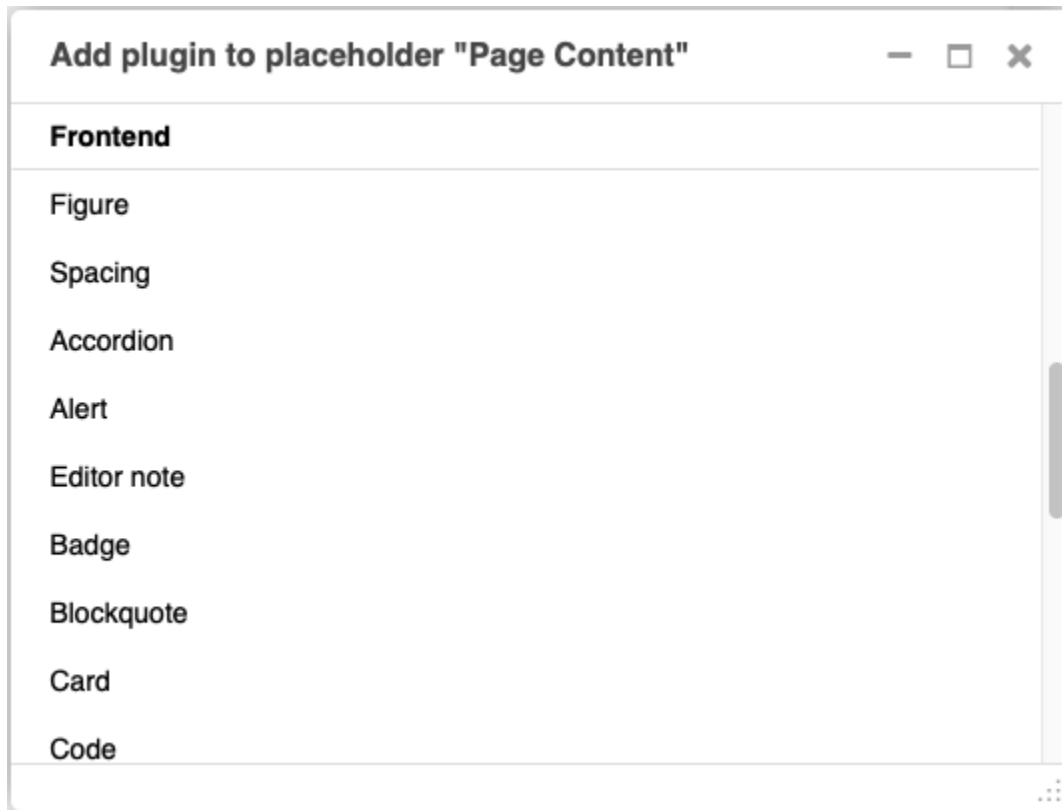
Frontend editing of plugins has been updated compared to **djangocms-bootstrap4** with three aims:

- Keep the essential editing required minimal and well-arranged on the editing forms.

- Eliminate the need for regularly adding html classes or other attributes like styles.

- Keep the possibility to change the html classes or tags in the rare case it is needed.

The editing has therefore been categorized in tabs starting with a plugin's key tab for its most important information. Other tabs add general modifications to the plugin, their availability depending on the plugin type. The well-known "advanced settings" is available to all plugins, however, its use should in most case be covered by the new other tabs:

Container Add Container	— ☐ ✖

Container   **Background**   **Spacing**   **Visibility**   **Advanced settings**

Container type:

Container	▼

Defines if the grid should use fixed width (`.container`) or fluid width (`.container-fluid`).

Cancel   Save

Background tab

The background tab allows to set a background context leading to the background being colored appropriately.

The background properties can be modified by changing **opacity** and its ability cast a **shadow**. Shadows allow the whole element to appear elevated from the background.

Container Add Container — □ ✕

| Container | **Background** | Spacing | Visibility | **Advanced settings** |

Background context:

----- | Primary | Secondary | Success | Danger | Warning | Info | Light | Dark | Transparent

Background opacity:

100% | 75% | 50% | 25% | 10%

Opacity of card background color (only if no outline selected)

Shadow:

----- | Keiner | S | M | L

Use shadows to optically lift cards from the background.

Cancel   Save

Spacing tab

The spacing tab is used to set margins and padding and to select which devices they should be applied.

For both margin and padding the settings can be made independently for horizontal and vertical padding.

| Container | Add Container | | | | — □ ✕ |

| Container | Background | **Spacing** | Visibility | Advanced settings |

Horizontal margin: **Vertical margin:**

▯▮ ▯▯ ▯▯ ------ ▾   ▯ ▯ ▯ ------ ▾

Apply margin on device:

📱 📱 📱 💻 🖥 🖥

Select only devices on which the margin should be applied. On other devices larger than the first selected device the margin will be set to zero.

Padding

Horizontal padding: **Vertical padding:**

▥ ▥ ▥ ------ ▾   ▭ ▭ ▯ ------ ▾

Cancel   Save

Visibility tab

The visibility tab controls on which devices the elements and its children should be visible. Use this to design different content for different devices.

Container Add Container — □ ✕

| Container | Background | Spacing | **Visibility** | Advanced settings |

Show element on device:

Select only devices on which this element should be shown.

Cancel   Save

Advanced settings tab

The advanced tab lets you chose which tag (typically a `div`) should be used to render the element. You may also add attributes like additional classes as a `class` attribute, an `id` or styles in the `style` attribute.

Container   Add Container   — □ ✕

| Container | Background | Spacing | Visibility | **Advanced settings** |

Advanced settings lets you add html attributes to render this element. Use them wisely and rarely.

Tag type:

div   section   article   header   footer   aside

Attributes:

➕

Cancel   Save

> **Warning:** Using the advanced tab requires some technical knowledge on the sites installation, e.g., what css classes are available. This is why the advanced settings tab should only be used rarely. If you find yourself using it regularly, extending **djangocms-frontend** using a theme might be the more editor-friendly and more maintainable solution.

The advanced tab label carries a blue-ish dot to indicate that attributes are set in the advanced settings tab. These attributes can change the appearance of the element significantly which is why the dot reminds the editor that there are advanced settings present.

| Picture / Image | Format | Link settings | Cropping settings | Visibility | **Advanced settings** |

238

Advanced settings lets you add html attributes to render this element. Use them wisely and rarely.

Attributes:

| style | transform: scale(0.5); | ✖ |

Error indicators

In case the form is not valid when the user tries to save all fields that are marked invalid will have an error message attached. Since not all fields are visible in tabbed editing tabs containing an error have a red badge at the upper right corner:

3.1.2 Grid plugins

For details on how grids work, see, e.g. the Bootstrap 5 documentation.

Container

A container is an invisible element that wraps other content. There are in two types of containers:

Container All other containers restrict the width of their content depending on the used device.

Fluid container A fluid container occupies the full width available - no matter how wide the viewport (or containing) element is.

Full container A full container is like a fluid container and occupies the full width available. Additionally, it does not have a padding. Its content can therefore fill the entire area. Full containers are useful if you want to add a background color or shadow to another DOM element, like, e.g., the contents of a column.

Note:

New feature: Containers can have a background color ("context"), opacity and shadow.

Row

A row contains one or more columns. By default columns are displayed next to each other.

To automatically create not only a row but also some columns within that row, enter the number of columns you will be using. You can always later add more columns to the row or delete columns from the row.

Vertical aligned defines how columns of different height are positioned against each other.

Horizontal alignment defines how columns **that do not fill an entire row** are distributed horizontally.

Note: New feature:

The section "Row-cols settings" defines how many columns should be next to each other for a given display size. The "row-cols" entry defines the number of columns on mobile devices (and above if no other setting is given), the "row-cols-xl" entry the number of columns on a xl screen.

Column

The column settings is largely about how much of the grid space the column will use horizontally. To this end, the grid is divided in (usually) 12 strips of equal width.

Auto sizing If no information on the column size is given, the column will be autosizing. This means that all autosizing columns of a row will occupy the same fraction of the space left, e.g. by sized columns.

Specifically sized columns If you enter a number the column for the specific screen size will exactly have the specified width. The unit of width is one twelfth of the surrounding's row width.

Natural width: If you need a column to take its natural width, enter `0` for its column size. This will display `auto` for columns size.

Also, you can adjust the vertical alignment of the specific column from the row's default setting.

Finally, you can set the alignment of the content to left (right in a rtl environment), center or right (left in a rtl environment). This comes handy if, e.g., the column is supposed to contain centered content.

Note:

Removed: The column type entry has been removed since it was a legacy from Bootstrap version 3.

3.1.3 Component plugins

`djangocms-frontend` adds a set of plugins to Django-CMS to allow for quick usage of components defined by the underlying css framework, e.g. bootstrap 5.

While `djangocms-frontend` is set up to become framework agnostic its heritage from `djangocms-bootstrap4` is intentionally and quite visible. Hence for the time being, this documentation references the Bootstrap 5 documentation.

Accordion component

Build vertically collapsing sections using accordions:

Accordions consist of an Accordion plugin which has an Accordion Item plugin for each collapsable section.

Also see Bootstrap 5 Accordion documentation.

Alert component

Alerts provide contextual feedback messages for typical user actions with a handful of available alert messages.

> **Well done!**   ✕
>
> Aww yeah, you successfully read this important alert message. This example text is going to run a bit longer so that you can see how spacing within an alert works with this kind of content.
>
> ───
>
> Whenever you need to, be sure to use margin utilities to keep things nice and tidy.
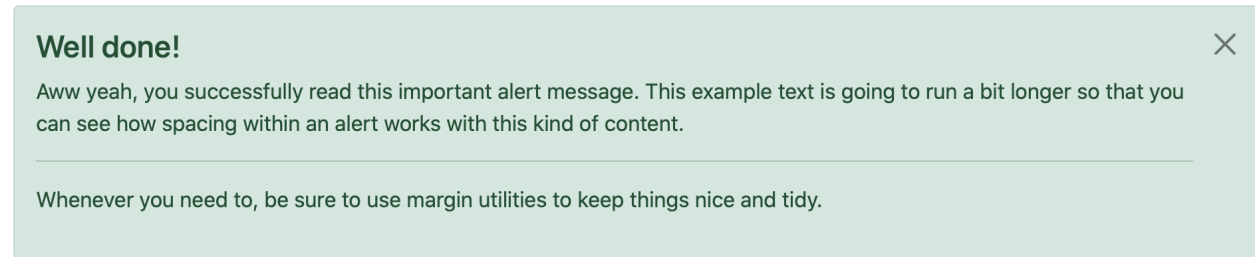
Alerts can be marked dismissible which implies that a close button is added on the right hand side.

| Page Content | COLLAPSE ALL | ✏ + ≡ |
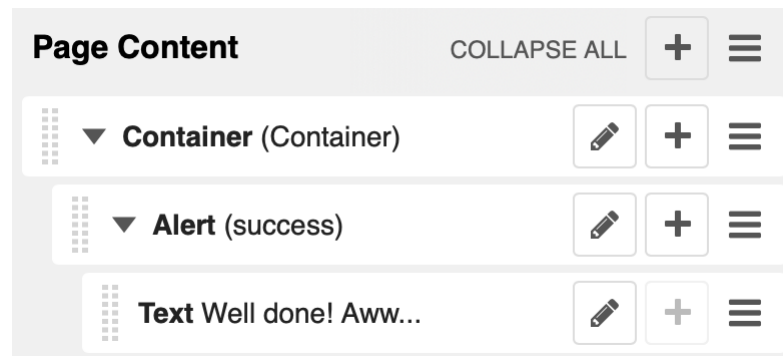
Note:

New features: Alerts can have **shadows** to optically lift them.

Also see Bootstrap 5 Alerts documentation.

Badge component

Badges are small count and labeling components usually in headers and buttons.

While often useful if populated automatically as opposed to statically in a plugin, badges are useful, e.g., to mark featured or new headers.

Well done! New

Also see Bootstrap 5 Badge documentation.

Card component

A card is a flexible and extensible content container. It includes options for headers and footers, a wide variety of content, contextual background colors, and powerful display options.

A card consists of the card wrapper itself, the Card Plugin. It can contain one ore more instances of a Card Inner Plugin for header, body or footer, but also potentially an Image Plugin for the card image or list group components.

The corresponding plugin tree is here:
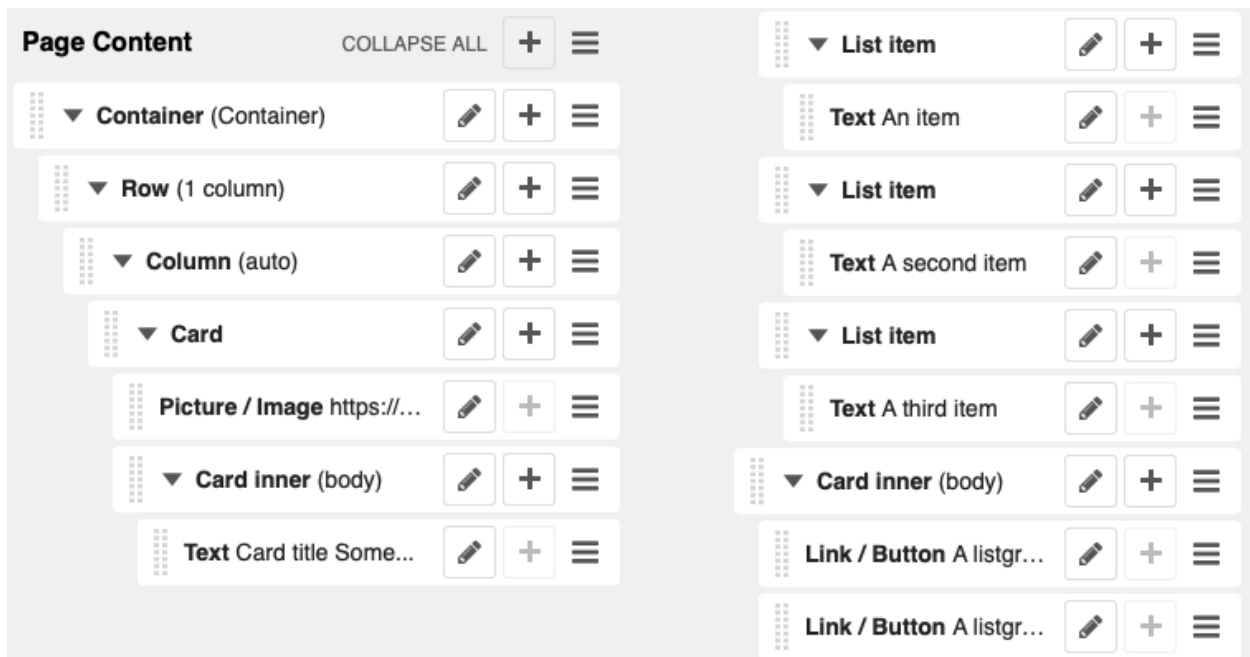
Cards can be grouped by a **Card Layout component** offering the ability group cards or display a grid of cards. The latter can be controlled by responsive tools. If you need more granular responsive settings, please revert to *Grid plugins* and build your own custom grid.

> **Warning:** djangocms-bootstrap4 Card Decks are not supported by Bootstrap 5. Card decks will be converted to grids of cards upon *Migrating from djangocms-bootstrap4*.

Card

The card resides in a Card plugin which allows for coloring, opacity and shadow options.

Content is added to the card plugin by creating child plugins. These can be of the type *Card inner*, *Picture / Image*, *List group*, or Row.

Note:

New feature: By adding images or list groups directly to a card, unnecessary margins are avoided.

Card inner

The Card Inner plugin allows to add the card header, body, footer or an overlay space for a card image.

Here is an example of the new card **Image overlay** feature:

Also see Bootstrap 5 Card documentation.

Carousel component

A Carousel is a set of images (potentially with some description) that slide in (or fade in) one after the other after a certain amount of time.

Collapse component

The Collapse hides text behind its headline and offers the user a trigger (e.g., a button) to reveal itself.

Compared to the accordion component the collapse component often is more flexible but also requires more detailed styling.

Jumbotron component

The jumbotron component is a large header, used e.g. as a page header. It has been part of Bootstrap 4 and is still supported as a convenient way to generate page headers.

Note: The jumbotron header is not reflected by the table of contents component.

Link / Button component

The link / button plugin creates a styled link or button (using the <a> HTML tag).

It is designed to allow for external and internal links. Internal links point to a CMS page or pages of other Django applications. They are dynamic, i.e. if the page's url changes (e.g. because it is moved in the page tree) all links pointing to the page change accordingly.

Note: djangocms-frontend uses django-cms' function `get_page_choices(lang)` to get the list of available pages in the current language.

The developer can extend the list of available internal link targets to pages outside the CMS page tree using the `DJANGOCMS_FRONTEND_LINK_MODELS` setting in the project's `.settings` file. The link/button component can point to any page controlled by a Django model if the model class implements the `get_absolute_url` method. A typical use case would, e.g., blog entries of djangocms-blog. (This approach was inspired by mkoisten's djangocms-styledlink.)

For more information, see *How to add internal link targets outside the CMS*

Note: Only those destinations (outside the CMS) are shown for which a model admin is registered and the logged in user has view permissions: A user will only see a destination if they can view it in the admin site.

List group component

List groups are a flexible and powerful component for displaying a series of content. Modify and extend them to support just about any content within.

The component consists of a wrapper - ListGroup Plugin - and the items of the list - ListGroupItem Plugin. If the list item is a link it suffices to insert a Link Plugin directly as a child of the ListGroup Plugin.

List group

The only setting is the list group flush setting. If checked, the list group will be rendered without borders to blend into the surrounding element, e.g. a card.

List group item

Simple content can be specified by providing "One line content". More complex content of a list group item is rendered by child plugins. If child plugins are available the "one line content" is ignored.

List group items can have a context (color), and three state: Regular, active and disabled.

Note:

New feature: Links can be added to list groups and automatically are interpreted as list group items.

Media component

The media component is another legacy component from djangocms-bootstrap4. **djangocms-frontend** recreates it using responsive utilities.

Picture / image component

The image or picture component make responsive picture uploads available as well as responsive embedding of external pictures.

Spacing component

The spacing component provides horizontal and/or vertical spacing. If used without child plugins it just provides the amount of space specified on the specified sides.

Note: If no spacing is selected the spacing component can be used to individually style the content using the attributes fields in "Advanced Settings".

Blockquote component

Creates a `<blockquote>` tag.

Note:

New feature: Alternatively to the un-formatted quote text, child plugins can be used to fill the content of the block-quote.

Code component

Have code snippets on your site using this plugin, either inline or as a code block.

Figure component

The figure component supplies a wrapper and a caption for a figure. The figure itself is placed inside the figure component (as child plugins).

Tabs component

Note: Bootstrap 5 comes with a fade animation. Additional animations will have to be provided by you or a third party. If you use a CSS animation library, you can make these animations available by adjusting the `DJANGOCMS_FRONTEND_TAB_EFFECTS` setting.

3.1.4 Forms

Note: Forms a available as of version 0.9.1

> **Warning:** Currently, for forms to work they require **jQuery** to be installed by the user.

djangocms-frontend supports rendering of styled forms which is part of all major frontend frameworks, like Bootstrap 5. The objective is to tightly integrate forms in the website design. Djangocms-frontend allows as many forms as you wish on one page. All forms are **ajax/xhr-based**. To this end, djangocms-frontend extends the django CMS plugin model allowing a form plugin to receive ajax post requests.

There are two different ways to manage forms with **djangocms-frontend**:

1. **Building a form with django CMS' powerful structure board.** This is fast an easy. It integrates smoothly with other design elements, especially the grid elements allowing to design simple responsive forms.

   Form actions can be configured by form. Built in actions include saving the results in the database for later evaluation and mailing submitted forms to the site admins. Other form actions can be registered.

   If you prefer to have a central form repository, we suggest **djangocms-alias** to manage your forms centrally. Djangocms-alias becomes your form editors and forms can be placed on pages by refering to them with their alias.

---

2. **Registering an application-specific form with djangocms-frontend.** If you already have forms you may register them with djangecms-frontend and allow editors to use them in the form plugin. If you use **django-crispy-forms** all form layouts will be retained. If you only have simpler design requirements, **djangocms-frontend** allows you to use fieldsets as with admin forms.

Building forms

Form plugin

All forms live in the Form plugin. A form plugin can be positioned everywhere except inside another form plugin.

If you want to use the structure board to build your form you will have to add the form components as child plugins to a form plugin. If you have registiered an application-specific form with djangocms-frontend you will be able to select one of the registered forms for be shown by the form plugin. (If you do both, the selected form takes precedence over the child plugins.)

Form Change Form — ☐ ✕

| Form | Submit button | Advanced settings | 547 |

Form:

Contact form ▾

Form name:

my-contact-form

☐ Login required to submit form

To avoid issues with user experience use this type of form only on pages, which require login.

☐ User can reopen form

Requires "Login required" to be checked to work.

☐ Floating labels

Margin between fields:

* 1 ▾

Cancel   Delete   Save

In the tab "Submit button" the name and appearance of the submit button is configured.

Form fields

The form plugin accepts all regular plugins **plus** special plugins that represent form fields. These are:

- Text
- Textarea
- Integer
- Decimal
- Boolean
- Date
- Time
- Date and Time
- Select/Choice
- URL
- Email

Each field requires an input of then specific form. Some fields (e.g., Boolean or Select/Choice) offer options on the specific input widget.

Djangocms frontend will use framework specific widgets or fall back to standard widgets browsers offer (e.g., date picker).

Using forms from other apps

Other apps can register forms to be used by **djangocms-frontend**. Once at least one form is registered, the form will appear as a option in the Form plugin.

Registration is can simply be done by a decorator or function call:

```python
from django import forms
from djangocms_frontend.contrib.forms import register_with_frontend


@register_with_frontend
class MyCoolForm(forms.Form):
    ...


class MyOtherCoolForm(forms.Form):
    ...


register_with_frontend(MyOtherCoolForm)
```

There are three ways **djangocms-frontend** can render registered forms:

1. **Regular form rendering**: all fields a shown below one another. This is only advisable for very simple forms (e.g. a contact form with name, email, and text body).

2. **Adding a ``fieldsets`` argument to the form**: The `fieldsets` work as you know them from `ModelAdmin`. See Django documentation. This may be the most convenient way of building not-too-complex forms. **djangocms-frontend** uses the grid system to generate the form layout.

3. **Using the third party package `django-crispy-forms <https://github.com/django-crispy-forms/django-crispy-forms>`_:** If installed and the form has a property `helper` the form is automatically rendered using **django-crispy-forms**. Note, however, that the submit button is rendered by the plugin. Hence do not include it into the form (which is possible with **django-crispy-forms**).

3.1.5 How-to guides

How to add internal link targets outside the CMS

By default the link/button component offers available CMS pages of the selected language as internal links.

The developer may extend this setting to include other page-generating Django models as well by adding the `DJANGOCMS_FRONTEND_LINK_MODELS` setting to the project's `settings.py` file.

settings.**DJANGOCMS_FRONTEND_LINK_MODELS**

> `DJANGOCMS_FRONTEND_LINK_MODELS` contains a list of additional models that can be linked.
>
> Each model is specified within its own dict. The resulting drop-down list will contain objects grouped by their type. The order of the types in the list is defined by the order of their definition in this setting.
>
> The only required attribute for each model is `class_path`, which must be the full python path to the model.
>
> Additional attributes are:
>
> **type:** This is the name that will appear in the grouped dropdown menu. If not specified, the name of the class will be used E.g., "`Page`".
>
> **filter:** You can specify additional filtering rules here. This must be specified as a dict but is converted directly into kwargs internally, so, `{'published':  True}` becomes `filter(published=True)` for example.
>
> **order_by:** Specify the ordering of any found objects exactly as you would in a queryset. If this is not provided, the objects will be ordered in the natural order of your model, if any.
>
> **search:** Specifies which (text) field of the model should be searched when the user types a search string.

Note: Each of the defined models must define a `get_absolute_url()` method on its objects or the configuration will be rejected.

Example for a configuration that allows linking CMS pages plus two different page types from two djangocms-blog apps called "Blog" and "Content hub" (having the `app_config_id` 1 and 2, respectively):

```python
DJANGOCMS_FRONTEND_LINK_MODELS = [
    {
        "type": _("Blog pages"),
        "class_path": "djangocms_blog.models.Post",
        "filter": {"publish": True, "app_config_id": 1},
         "search": "translations__title",
    },
    {
        "type": _("Content hub pages"),
        "class_path": "djangocms_blog.models.Post",
        "filter": {"publish": True, "app_config_id": 2},
         "search": "translations__title",
    },
]
```

Another example might be (taken from djangocms-styledlink documentation):

```
DJANGOCMS_FRONTEND_LINK_MODELS = [
    {
        'type': 'Clients',
        'class_path': 'myapp.Client',
        'manager_method': 'published',
        'order_by': 'title'
    },
    {
        'type': 'Projects',
        'class_path': 'myapp.Project',
        'filter': { 'approved': True },
        'order_by': 'title',
    },
    {
        'type': 'Solutions',
        'class_path': 'myapp.Solution',
        'filter': { 'published': True },
        'order_by': 'name',
    }
]
```

The link/button plugin uses select2 to show all available link targets. This allows you to search the page titles.

> **Warning:** If you have a huge number (> 1,000) of link target (i.e., pages or blog entries or whatever) the current implementation might slow down the editing process. In your `settings` file you can set `DJANGOCMS_FRONTEND_MINIMUM_INPUT_LENGTH` to a value greater than 1 and **djangocms-frontend** will wait until the user inputs at least this many characters before querying potential link targets.

How to extend existing plugins

Existing plugins can be extended through two type of class mixins. `djangocms-frontend` looks for these mixins in two places:

1. In the theme module. Its name is specified by the setting `DJANGOCMS_FRONTEND_THEME` and defaults to `djangocms_frontend`. For a theme app called `theme` and the bootstrap5 framework this would be `theme.frontends.bootstrap5.py`.

2. In `djangocms_frontend.contrib.*app*.frontends.*framework*.py`. For the alert app and the bootstrap5 framework this would be `djangocms_frontend.contrib.alert.frontends.bootstrap5.py`.

Both mixins are included if they exist and all methods have to call the super methods to ensure all form extensions and render functionalities are processed.

The theme module is primarily thought to allow for third party extensions in terms of functionality and/or design.

The framework module is primarily thought to allow for adaptation of `djangocms-frontend` to other css frameworks besides Bootstrap 5.

RenderMixins

The render mixins are called "*PluginName*RenderMixin", e.g. `AlertRenderMixin` and are applied to the plugin class. This allows for the redefinition of the `CMSPlugin.render` method, especially to prepare the context for rendering.

Also it can add fields to the front end editing form by subclassing `CMSPlugin.get_fieldsets`. This allows for extension or change of the plugin's admin form. The admin form is used to edit or create a plugin.

FormMixins

Form mixins are used to add fields to a plugin's admin form. These fields are available to the render mixins and, of course, to the plugin templates.

Form mixins are called "*PluginName*FormMixin", e.g. `AlertFormMixin` and are applied to the editing form class. Form mixins are a subclass of `entangled.EntangledModelFormMixin`.

Working example

Let's say you wanted to extend the `GridContainerPlugin` to offer the option for a background image, and say a blur effect. The way to do it is to create a theme app. You are free to chose its name. For this example we take it to be "theme". Please replace "theme" by your own theme's name.

First, create a directory structure like this:

```
theme
├── __init__.py
├── forms.py
├── frameworks
│   ├── __init__.py
│   └── bootstrap5.py
├── static
│   └── css
│       └── background_image.css
└── templates
    └── djangocms_frontend
        └── bootstrap5
            └── grid_container.html
```

All `__init__.py` files remain empty.

Next, you add some fields to the `GridContainerForm` (in `theme/forms.py`):

```python
from django import forms
from django.db.models import ManyToOneRel
from django.utils.translation import gettext as _
from djangocms_frontend import settings
from entangled.forms import EntangledModelFormMixin
from filer.fields.image import AdminImageFormField, FilerImageField
from filer.models import Image


IMAGE_POSITIONING = (
```

(continues on next page)

```python
        ("center center", _("Fully Centered")),
        ("left top", _("Top left")),
        ("center top", _("Top center")),
        ("right top", _("Top right")),
        ("left center", _("Center left")),
        ("right center", _("Center right")),
        ("left bottom", _("Bottom left")),
        ("center bottom", _("Bottom center")),
        ("right bottom", _("Bottom right")),
)


class GridContainerFormMixin(EntangledModelFormMixin):
    class Meta:
        entangled_fields = {
            "config": [
                "container_image",
                "image_position",
                "container_blur",
            ]
        }

    container_image = AdminImageFormField(
        rel=ManyToOneRel(FilerImageField, Image, "id"),
        queryset=Image.objects.all(),
        to_field_name="id",
        label=_("Image"),
        required=False,
        help_text=_("If provided used as a cover for container."),
    )
    image_position = forms.ChoiceField(
        required=False,
        choices=settings.EMPTY_CHOICE + IMAGE_POSITIONING,
        initial="",
        label=_("Background image position"),
    )
    container_blur = forms.IntegerField(
        required=False,
        initial=0,
        min_value=0,
        max_value=10,
        help_text=_("Blur of container image (in px)."),
    )
```

> **Warning:** These form fields are mixed to the original form. Please make sure to avoid name collisions for the fields.

> **Note:** If you need to add many form mixins, consider turning `forms.py` into a package, i.e. create a directory `forms` and distribute the mixins over several files, e.g., `forms/marketing_forms.py` etc., and importing the all mixins

---

relevant to **djangocms-frontend** into the directory's `__init__.py`.

Rendering should be done with the Bootstrap 5 framework. Hence all rendering mixins go into `theme/bootstrap5.py`. Since we are extending the `GridContainer` plugin the appropriate mixin to define is `GridContainerMixin`:

```python
from django.utils.translation import gettext as _
from djangocms_frontend.helpers import insert_fields


class GridContainerRenderMixin:
    render_template = "djangocms_frontend/bootstrap5/grid_container.html"

    def get_fieldsets(self, request, obj=None):
        """Extend the fieldset of the plugin to contain the new fields
        defined in forms.py"""
        return insert_fields(
            super().get_fieldsets(request, obj),
            (
                "container_image",
                (
                    "image_position",
                    "container_blur",
                ),
            ),
            block=None,   # Create a new fieldset (called block here)
            position=1,   # at position 1 (i.e. directly after the mail fieldset)
            blockname=_("Image"),   # and call the fieldset "Image"
        )

    def render(self, context, instance, placeholder):
        """Render should process the form fields and turn them into appropriate
        context items or add corresponding classes to the instance"""
        if getattr(instance, "container_image", None):
            instance.add_classes("imagecontainer")
            context["bg_color"] = (
                f"bg-{instance.background_context}"
                if getattr(instance, "background_context", False)
                else ""
            )
        return super().render(context, instance, placeholder)
```

> **Warning:** Do not forget to call `super()` in both the `get_fieldsets` and the `render` method.

The `render` method provides required context data for the extended functionality. In this case it adds `imagecontainer` to the list of classes for the container, processes the background colors since it should appear above the image (and not below), as well as blur.

The `get_fieldsets` method is used to make django CMS show the new form fields in the plugin's edit modal (admin form, technically speaking).

Then, a new template is needed (in `theme/templates/djangocms_frontend/bootstrap5/grid_container.html`):

```
{% load cms_tags static %}{% spaceless %}
  <{{ instance.tag_type }}{{ instance.get_attributes }}
  {% if instance.background_opacity and not instance.image %}
    {% if instance.container_blur %}
      backdrop-filter: blur({{ instance.container_blur }}px);
    {% endif %}"
  {% endif %}>
  {% if instance.image %}
    <div class="image"
      style="background-image: url('{{ instance.image.url }}');
             background-position: {{ instance.image_position|default:'center center' }};
             background-repeat: no-repeat;background-size: cover;
             {% if instance.container_blur %}
               filter: blur({{ instance.container_blur }}px);
             {% endif %}">
    </div>
  {% elif instance.container_image %}
    <div class="image placeholder placeholder-wave"></div>
  {% endif %}
  {% if bg_color %}
    <div class="cover {{ bg_color }}"{% if instance.background_opacity %}
         style="opacity: {{ instance.background_opacity }}%"{% endif %}></div>
  {% endif %}
  {% if instance.container_image %}
    <div class="content">
  {% endif %}
    {% for plugin in instance.child_plugin_instances %}
      {% render_plugin plugin %}
    {% endfor %}
  {% if instance.container_image %}</div>{% endif %}
</{{ instance.tag_type }}>{% endspaceless %}
{# Only add if the css is not included in your site's global css #}
{% addtoblock 'css' %}
    <link rel="stylesheet" href="{% static 'css/background_image.css' %}">
{% endaddtoblock
```

Finally, a set of css style complement the new template. The styles can either be added to the css block in the template (if used scarcely and as done in the above example) or directly to your project's css files.

The required styles are:

```css
/* Image Container */

div.imagecontainer {
    position: relative;
    min-height: 112px;
}

div.imagecontainer > div.cover,
div.imagecontainer > div.image {
    position: absolute;
    left:0;
    right:0;
```

```
    top:0;
    bottom:0;
}

div.imagecontainer > div.content {
    position: relative;
}
```

With these three additions, all grid container plugins will now have additional fields to define background images to cover the container area.

If the theme is taken out of the path djangocms-frontend will fall back to its basic functionality, i.e. the background images will not be shown. As long as plugins are not edited the background image information will be preserved.

Note: A few suggestions on extending **djangocms-frontend**:

- You may think of customizing bootstrap by including a folder `sass` in your theme app. For more see Bootstrap 5 documentation on customizing.

- If you need entirely new plugins, create a file `cms_plugins.py` and import `CMSUIPlugin` (import from `djangocms_frontend.cms_plugins`) as base class for the plugins.

- Create `models.py` file for the models (which need to be proxy models of `FrontendUIItem` (import from `djangocms_frontend.models`).

How to create a theme app

`djangocms-frontend` is designed to be "themable". A theme typically will do one or more of the following:

- Style the appearance using css

- Extend standard plugins

- Add custom plugins

How to add the tab editing style to my other plugins

If you prefer the tabbed frontend editing style of **djangocms-frontend** you can easily add it to your own plugins.

If you use the standard editing form, just add a line specifying the `change_form_template` to your plugin class:

```
class MyCoolPlugin(CMSPluginBase):
    ...
    change_form_template = "djangocms_frontend/admin/base.html"
    ...
```

If you already have your own `change_form_template`, make sure it extends `djangocms_frontend/admin/base.html`:

```
{% extends "djangocms_frontend/admin/base.html" %}
{% block ...%}
    ...
{% endblock %}
```

3.1.6 Reference

Settings

djangocms-frontend can be configured by putting the appropriate settings in your project's `settings.py`.

settings.**DJANGOCMS_FRONTEND_TAG_CHOICES**
> Defaults to `['div', 'section', 'article', 'header', 'footer', 'aside']`.
>
> Lists the choices for the tag field of all djangocms-frontend plugins. `div` is the default tag.
>
> These tags appear in Advanced Settings of some elements for editors to chose from.

settings.**DJANGOCMS_FRONTEND_GRID_SIZE**
> Defaults to 12.

settings.**DJANGOCMS_FRONTEND_GRID_CONTAINERS**
> Default:

```
(
    ("container", _("Container")),
    ("container-fluid", _("Fluid container")),
    ("container-full", _("Full container")),
)
```

settings.**DJANGOCMS_FRONTEND_USE_ICONS**
> Defaults to `True`.
>
> Decides if icons should be offered, e.g. in links.

settings.**DJANGOCMS_FRONTEND_CAROUSEL_TEMPLATES**
> Defaults to `(('default', _('Default')),)`
>
> If more than one option is given editors can select which template a carousel uses for rendering. Carousel expects the templates in a template folder under `djangocms_frontend/bootstrap5/carousel/{{ name }}/`. `{{ name }}` denotes the value of the template, i.e. `default` in the default example.
>
> Carousel requires at least two files: `carousel.html` and `slide.html`.

settings.**DJANGOCMS_FRONTEND_TAB_TEMPLATES**
> Defaults to `(('default', _('Default')),)`
>
> If more than one option is given editors can select which template a tab element uses for rendering. Tabs expects the templates in a template folder under `djangocms_frontend/bootstrap5/tabs/{{ name }}/`. `{{ name }}` denotes the value of the template, i.e. `default` in the default example.
>
> Tabs requires at least two files: `tabs.html` and `item.html`.

settings.**DJANGOCMS_FRONTEND_LINK_TEMPLATES**
> Defaults to `(('default', _('Default')),)`
>
> If more than one option is given editors can select which template a link or button element uses for rendering. Link expects the templates in a template folder under `djangocms_frontend/bootstrap5/link/{{ name }}/`. `{{ name }}` denotes the value of the template, i.e. `default` in the default example.
>
> Link requires at least one file: `link.html`.

settings.**DJANGOCMS_FRONTEND_JUMBOTRON_TEMPLATES**
> Defaults to `(('default', _('Default')),)`
>
> Jumbotrons have been discontinued form Bootstrap 5 (and are not present in other frameworks either). The default template mimics the Bootstrap 4's jumbotron.

If more than one option is given editors can select which template a jumbotron element uses for rendering. Jumbotron expects the template in a template folder under `djangocms_frontend/bootstrap5/jumbotron/` `{{ name }}/`. `{{ name }}` denotes the value of the template, i.e. `default` in the default example.

Link requires at least one file: `jumbotron.html`.

settings.**DJANGOCMS_FRONTEND_SPACER_SIZES**
　　Default:

```
(
    ('0', '* 0'),
    ('1', '* .25'),
    ('2', '* .5'),
    ('3', '* 1'),
    ('4', '* 1.5'),
    ('5', '* 3'),
)
```

settings.**DJANGOCMS_FRONTEND_CAROUSEL_ASPECT_RATIOS**
　　Default: `((16, 9),)`

　　Additional aspect ratios offered in the carousel component

settings.**DJANGOCMS_FRONTEND_COLOR_STYLE_CHOICES**
　　Default:

```
(
    ("primary", _("Primary")),
    ("secondary", _("Secondary")),
    ("success", _("Success")),
    ("danger", _("Danger")),
    ("warning", _("Warning")),
    ("info", _("Info")),
    ("light", _("Light")),
    ("dark", _("Dark")),
)
```

settings.**DJANGOCMS_FRONTEND_ADMIN_CSS**
　　Default: `None`

　　Adds css format files to the frontend editing forms of **djangocms-frontend**. The syntax is with a `ModelForm`'s `css` attribute of its `Media` class, e.g., `DJANGOCMS_FRONTEND_ADMIN_CSS = {"all": ("css/admin.min. css",)}`.

　　This css might be used to style have theme-specific colors available in the frontend editing forms. The included css file is custom made and should only contain color settings in the form of

```
.frontend-button-group .btn-primary {
    color: #123456;  // add !important here if using djangocms-admin-style
    background-color: #abcdef;
}
```

Note: Changing the `color` attribute might require a `!important` statement if you are using **djangocms-admin-style**.

---

settings.`DJANGOCMS_FRONTEND_MINIMUM_INPUT_LENGTH`
> If unset or smaller than 1 the link plugin will render all link options into its form. If 1 or bigger the link form will wait for the user to type at least this many letters and search link targets matching this search string using an ajax request.

settings.`TEXT_SAVE_IMAGE_FUNCTION`
> Requirement: `TEXT_SAVE_IMAGE_FUNCTION = None`

> **Warning:** Please be aware that this package does not support djangocms-text-ckeditor's Drag & Drop Images so be sure to set `TEXT_SAVE_IMAGE_FUNCTION = None`.

Models

djangocms-frontend subclasses the `CMSPlugin` model.

class `FrontendUIItem`(*CMSPlugin*)
> Import from `djangocms_frontend.models`.

> All concrete models for UI items are proxy models of this class. This implies you can create, delete and update instances of the proxy models and all the data will be saved as if you were using this original (non-proxied) model.

> This way all proxies can have different python methods as needed while still all using the single database table of `FrontendUIItem`.

`FrontendUIItem`.`ui_item`
> This CharField contains the UI item's type without the suffix "Plugin", e.g. "Link" and not "LinkPlugin". This is a convenience field. The plugin type is determined by `CMSPlugin.plugin_type`.

`FrontendUIItem`.`tag_type`
> This is the tag type field determining what tag type the UI item should have. Tag types default to `<div>`.

`FrontendUIItem`.`config`
> The field `config` is the JSON field that contains a dictionary with all specific information needed for the UI item. The entries of the dictionary can be directly **read** as attributes of the `FrontendUIItem` instance. For example, `ui_item.context` will give `ui_item.config["context"]`.

> **Warning:** Note that changes to the `config` must be written directly to the dictionary: `ui_item.config["context"] = None`.

`FrontendUIItem`.`add_classes`(*self*, **args*)
> This helper method allows a Plugin's render method to add framework-specific html classes to be added when a model is rendered. Each positional argument can be a string for a class name or a list of strings to be added to the list of html classes.

> These classes are **not** saved to the database. They merely a are stored to simplify the rendering process and are lost once a UI item has been rendered.

`FrontendUIItem`.`get_attributes`(*self*)
> This method renders all attributes given in the optional `attributes` field (stored in `.config`). The `class` attribute reflects all additional classes that have been passed to the model instance by means of the `.add_classes` method.

FrontendUIItem.**initialize_from_form**(*self*, *form*)
> Since the UIItem models do not have default values for the contents of their `.config` dictionary, a newly created instance of an UI item will not have config data set, not even required data.
>
> This method initializes all fields in `.config` by setting the value to the respective `initial` property of the UI items admin form.

FrontendUIItem.**get_short_description**(*self*)
> returns a plugin-specific short description shown in the structure mode of django CMS.

Form widgets

djangocms-frontend contains button group widgets which can be used as for `forms.ChoiceField`. They might turn out helpful when adding custom plugins.

class ButtonGroup(*forms.RadioSelect*)
> Import from `djangocms_frontend.fields`
>
> The button group widget displays a set of buttons for the user to chose. Usable for up to roughly five options.

class ColoredButtonGroup(*ButtonGroup*)
> Import from `djangocms_frontend.fields`
>
> Used to display the context color selection buttons.

class IconGroup(*ButtonGroup*)
> Import from `djangocms_frontend.fields`.
>
> This widget displays icons in stead of text for the options. Each icon is rendered by `<span class="icon icon-{{value}}"></span>`. Add css in the `Media` subclass to ensure that for each option's value the span renders the appropriate icon.

class IconMultiselect(*forms.CheckboxSelectMultiple*)
> Import from `djangocms_frontend.fields`.
>
> Like `IconGroup` this widget displays a choice of icons. Since it inherits from `CheckboxSelectMultiple` the icons work like checkboxes and not radio buttons.

class OptionalDeviceChoiceField(*forms.MultipleChoiceField*)
> Import from `djangocms_frontend.fields`.
>
> This form field displays a choice of devices corresponding to breakpoints in the responsive grid. The user can select any combination of devices including none and all.
>
> The result is a list of values of the selected choices or None for all devices selected.

class DeviceChoiceField(*OptionalDeviceChoiceField*)
> Import from `djangocms_frontend.fields`.
>
> This form field is identical to the `OptionalDeviceChoiceField` above, but requires the user to select at least one device.

Management commands

Management commands are run by typing `./manage.py frontend command` in the project directory. `command` can be one of the following:

migrate Migrates plugins from other frontend packages to **djangocms-frontend**. Currently supports **djangocms_bootstrap4** and **djangocms_styled_link**.

stale_references If references in a UI item are moved or removed the UI items are designed to fall back gracefully and both not throw errors or be deleted themselves (by a db cascade).

The drawback is, that references might become stale. This command prints all stale references, their plugins and pages/placeholder they belong to.

sync_permissions users or **sync_permissions groups** Django allows to set permissions for each user and group on a per plugin level. This might become somewhat tedious which is why this command will sync permissions. For each user or group it will copy the permissions of `djangocms_frontend.models.FrontendUIItem` to all installed djangocms-frontend plugins. If you need to change permissions for all plugins this requires you only to change them for `FrontendUIItem` and then syncing the new permission with these commands.

Running Tests

You can run tests by executing:

```
virtualenv env
source env/bin/activate
pip install -r tests/requirements.txt
python ./run_tests.py
```

3.1.7 Index

3.2 Indices and tables

- Index
- search