
djangoCMS-frontend Documentation

Release 1.3.2

Fabian Braun

Apr 25, 2024

CONTENTS

1	djangocms-frontend	1
2	Key features	3
3	Description	5
3.1	Contents	5
3.1.1	Getting started	5
3.1.2	Grid plugins	17
3.1.3	Component plugins	19
3.1.4	How-to guides	32
3.1.5	Reference	43
3.1.6	Index	49
3.2	Indices and tables	49
	Index	51

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.



djangocms-frontend

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.

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

To nicely integrate formss into your page we recommend **djangocms-form-builder** which works stand-alone but also nicely integrates with **djangocms-frontend**.

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
```

djangocms-frontend has weak dependencies you can install separately or by adding an option:

```
djangoCMS-frontend[djangoCMS-icon] # Installs djangoCMS-icon for icons support in links
djangoCMS-frontend[static-ace] # Installs djangoCMS-static-ace to include the ace code_
↪ editor in static files
djangoCMS-frontend[static-ace, djangoCMS-icon] # comma-separate multiple dependencies
```

`djangoCMS-frontend[static-ace]` is useful if your project cannot or should not access a CDN to load the `ace code editor` for the code plugin. Please be sure to in this case also add `"djangoCMS_static_ace"` to your project's `INSTALLED_APPS`.

Make apps available to your django project

Add the following entries to your `INSTALLED_APPS`:

```
"djangoCMS_icon", # optional
"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.icon",
"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 of 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>&nbsp;
<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
- Icon
- 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 if 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`.

Warning: The order of the apps in `INSTALLED_APPS` is **crucial**.

1. First is `djangoCMS_link` and `djangoCMS_icon` the first of which is needed by `djangoCMS_bootstrap4`,
2. **then come all `djangoCMS_bootstrap4` plugins.**
`djangoCMS_bootstrap4.contrib.bootstrap4_link` uninstalls the `Link` plugin of `djangoCMS_link`
3. At last come all `djangoCMS_frontend` apps.

Warning: The migration process does also migrate `djangoCMS-icon` instances to `djangoCMS-frontend`. If you prefer to use `djangoCMS-icon` instead, remove `"djangoCMS_frontend.contrib.icon"` from installed apps.

```
./manage.py cms delete-orphaned-plugins
./manage.py migrate
./manage.py frontend migrate
```

The migration process displays a counter indicating how many plugins were converted (an integer like 2133 depending how many bootstrap4 plugins you have.):

Migrated 2133 plugins. Successfully migrated plugins.

No further link destinations found. Setup complete.

In the case that no plugins were migrated the output looks very similar but does not contain the counter.

Nothing to migrate

No further link destinations found. Setup complete.

Only plugins managed by apps listed in `INSTALLED_APPS` will be migrated.

Warning: Spin up your development server and test at this point if the migration has succeeded. Open a former `djangoCMS-bootstrap4` plugin and check that it has the appearance of a `djangoCMS-frontend` plugin.

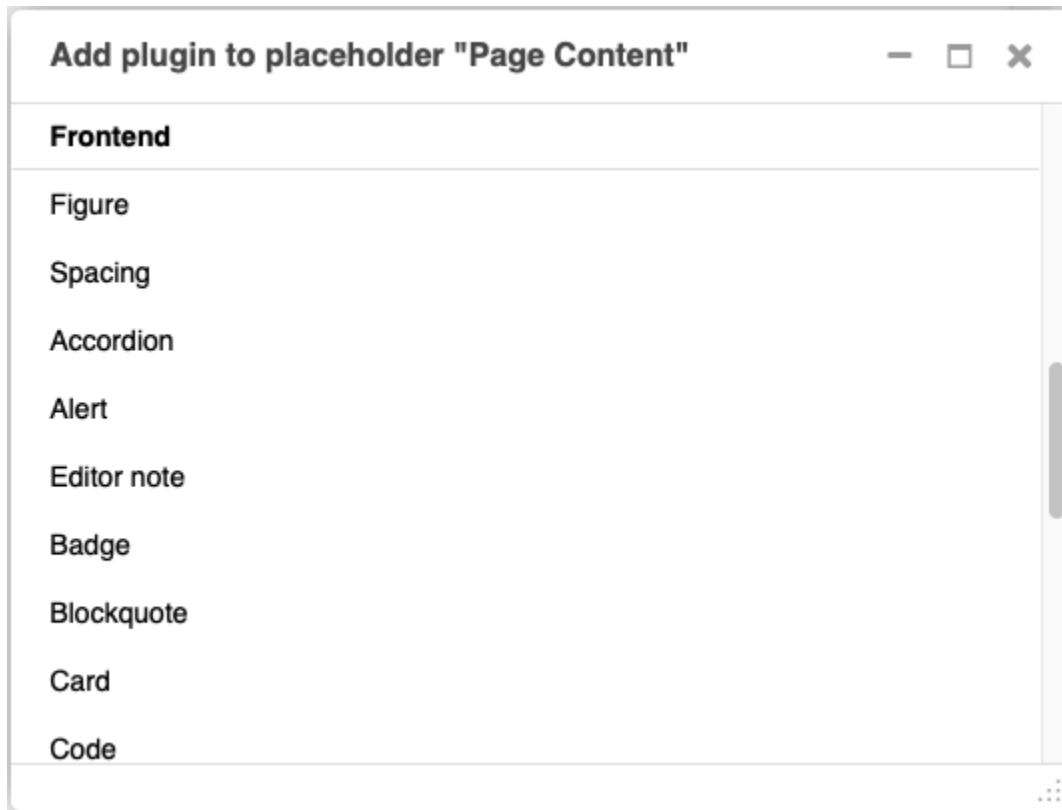
Therefore only after you finish the migration you can remove all `djangoCMS-bootstrap4` apps **and** `djangoCMS_link` from `INSTALLED_APPS` and you may delete the now empty database tables of `djangoCMS-bootstrap4`. You identify them by their name pattern:

```
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:

Please correct the error below.



238

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.

Column
Add Column

Column

Spacing

Visibility

Advanced settings

Column alignment:

Content alignment:

Responsive settings

Extra small

Small

Medium

Large

Extra large

XX large

Reset

Column size

col:

col-sm:

col-md:

col-lg:

col-xl:

col-xxl:

Order

order:

order-sm:

order-md:

order-lg:

order-xl:

order-xxl:

Offset

offset:

offset-sm:

offset-md:

offset-lg:

offset-xl:

offset-xxl:

Margin left

☐ ms-auto

☐ ms-sm-auto

☐ ms-md-auto

☐ ms-lg-auto

☐ ms-xl-auto

☐ ms-xxl-auto

Margin right

☐ me-auto

☐ me-sm-auto

☐ me-md-auto

☐ me-lg-auto

☐ me-xl-auto

☐ me-xxl-auto

Cancel

Save

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:

Accordion Item #1
^

This is the first item's accordion body. It is shown by default, until the user selects another accordion item.

It's also worth noting that just about any CMS plugin can go within the accordion item though the transition does limit overflow.

Accordion Item #2
v

Accordion Item #3
v

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

Page Content
EXPAND ALL
 +
 ≡

▼ **Container** (Container)

▼ **Accordion** (3 entries)

▼ **Accordion item** Accordion ...

Text This is the...

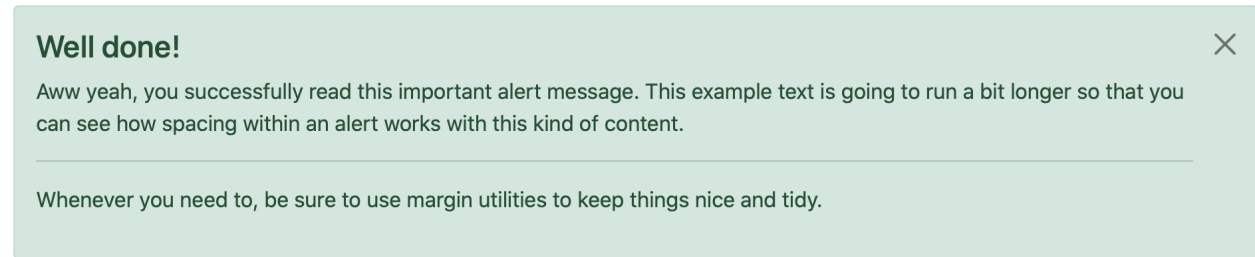
► **Accordion item** Accordion ...

► **Accordion item** Accordion ...

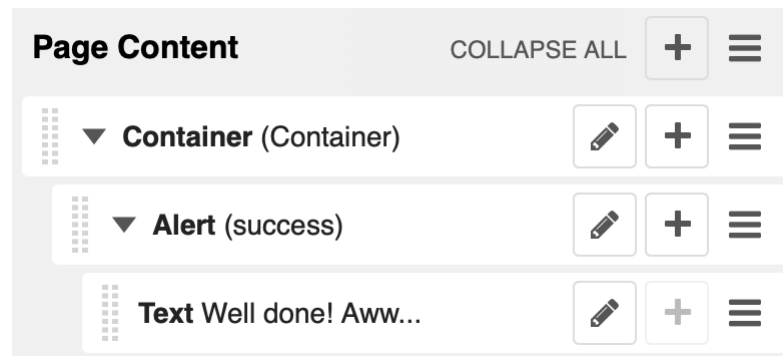
Also see Bootstrap 5 [Accordion](#) documentation.

Alert component

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



Alerts can be marked dismissible which implies that a close button is added on the right hand side.



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 or 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.



Card title

Some quick example text to build on the card title and make up the bulk of the card's content.

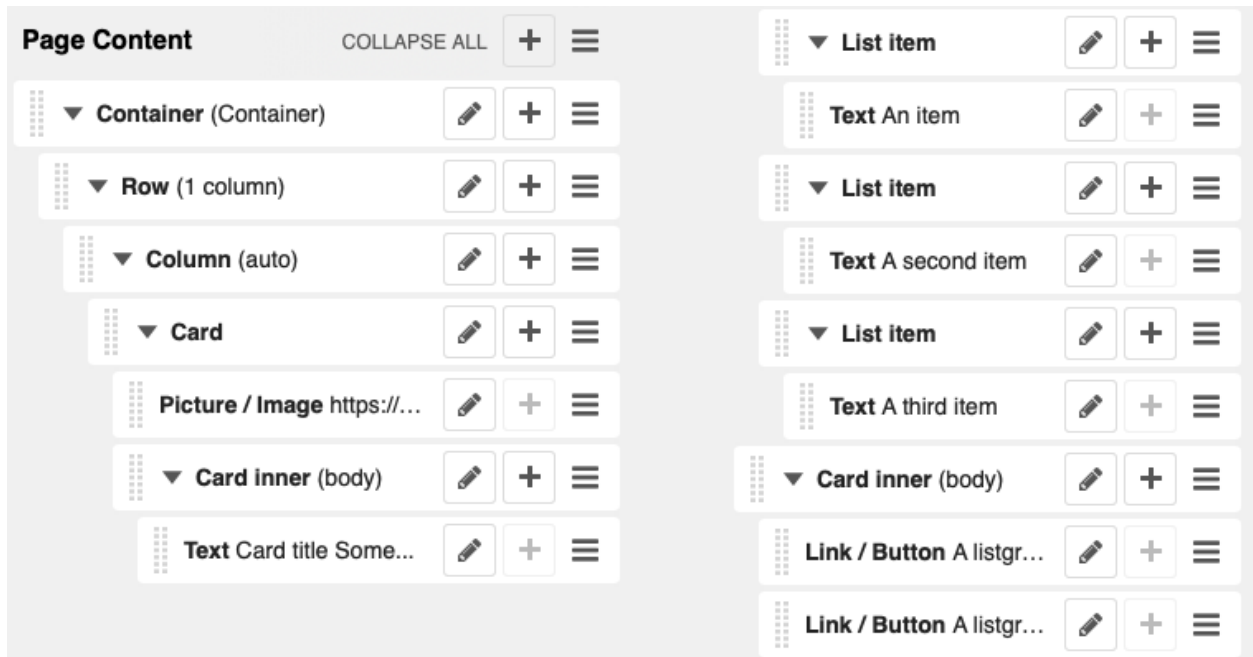
An item

A second item

A third item

[A listgroup link](#) [A listgroup link](#)

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.

Card

Add Card

Card

Background

Margin

Visibility

Advanced settings

Alignment:

Text context:

Primary

Secondary

Success

Danger

Warning

Info

Light

Dark

White

Card outline context:

Primary

Secondary

Success

Danger

Warning

Info

Light

Dark

Transparent

Uses the border to indicate context.

☐ Full height

If checked cards in one row will automatically extend to the full row height.

Cancel

Save

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.

Card inner

Add Card inner

Card inner

Background

Padding

Visibility

Advanced settings

Inner type:

Body

Header

Footer

Image overlay

Define the structure of the plugin.

Content alignment:

Cancel

Save

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.

Each slide requires a Carousel Slide child plugin. The simplest case specifies an image, potentially a caption and a link which is followed once the slide is clicked.

Since the design of carousels is somewhat opinionated template sets can be specified using the `DJANGOCMS_FRONTEND_CAROUSEL_TEMPLATES` setting.

Note: A Carousel Slide plugin can have child plugins itself. If an image is specified the child plugins add to the caption. If no image is specified the child plugins make up the slide.

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.

Added in version 1.2: djangoCMS-text-ckeditor supports dragging and dropping images into a rich text field. If you add the following line to your *settings.py* file, djangoCMS-text-ckeditor will automatically convert an image dropped into it to a djangoCMS-frontend image component.

```
TEXT_SAVE_IMAGE_FUNCTION = 'djangoCMS_frontend.contrib.image.image_save.create_image_
↪plugin'
```

Please note, that images dropped into djangoCMS-text-ckeditor are base64- encoded and take a quite a bit of band width. You may have to increase your *DATA_UPLOAD_MAX_MEMORY_SIZE* setting in *settings.py*.

We recommend not using this feature but instead adding a image component through the “CMS Plugin” menu of Ckeditor.

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 blockquote.

Code component

Have code snippets on your site using this plugin, either inline or as a code block. djangoCMS-frontend offers the [ace code editor](#) to enter code bits.

Warning: By default the ace code editor javascript code is retrieved over the internet from a cdn. If you do not want this to happen, e.g., for data privacy reasons or because your system is not connected to the internet, please use the weak dependency on [djangoCMS-static-ace](#) by changing your requirement from `djangoCMS-frontend` to `djangoCMS-frontend[static-ace]` and include `"djangoCMS_static_ace"` in your `INSTALLED_APPS`.

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).

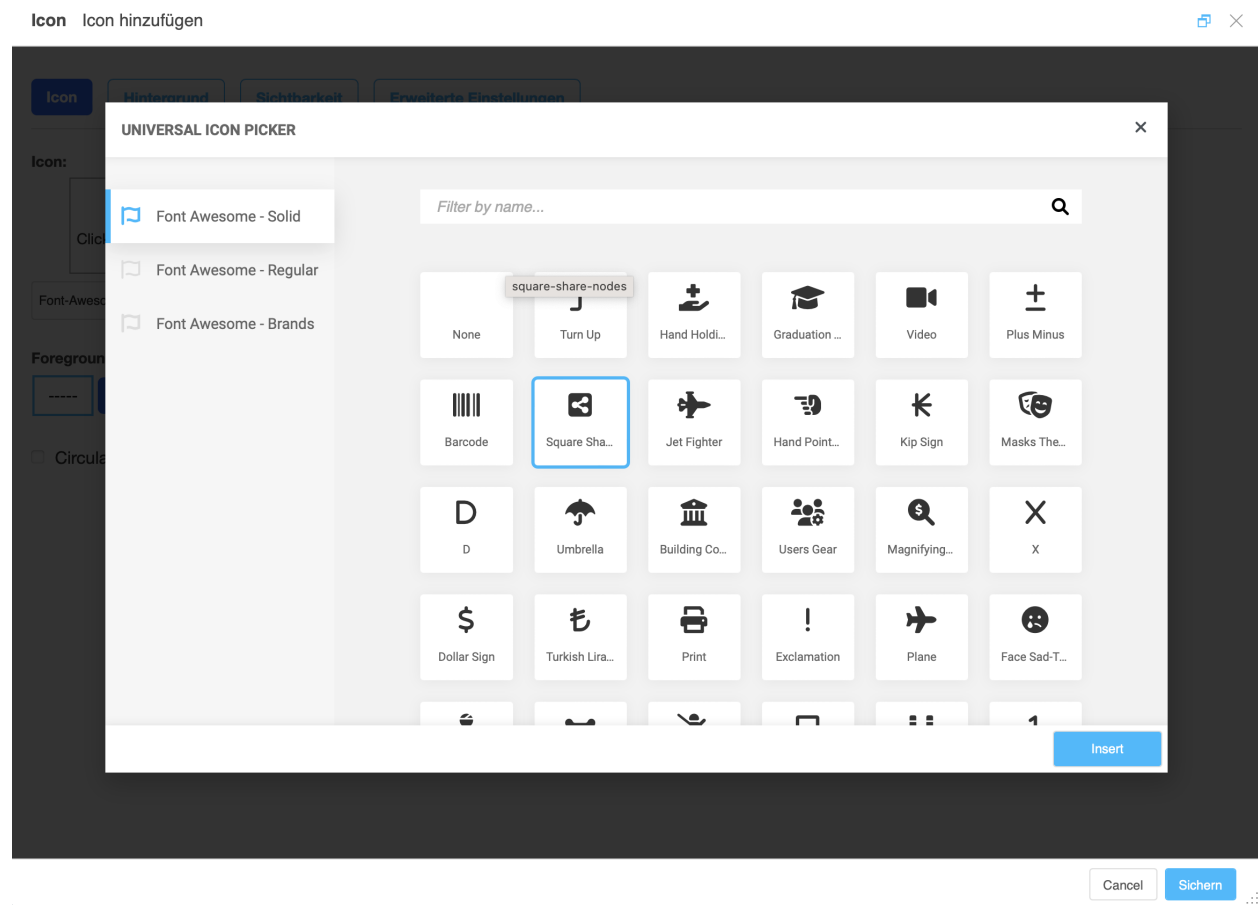
Tabs component

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.

Icon component

Added in version 1.1.

djangoCMS-frontend's icon plugin supports a variety of popular icon fonts. The icon component is centered around Gilles Migliori's [universal icon picker](#).



A version of it is bundled with djangoCMS-frontend. It currently contains support for the following icon sets:

- Bootstrap icons
- Elegant icons (bundled)
- Feather icons (bundled)
- Fomatic UI icons (bundled)
- Font awesome (regular, solid and brands)
- Foundation icons (bundled)
- Material icons (filled, outlined, sharp, two-tone)
- Open iconic
- Tabler icons
- Eric Flower's Weather icons (bundled)

Note: The icon picker needs a config file for each icon set. This requires regular update. Please be patient if new icons do not appear immediately in djangoCMS-frontend's icon picker or - even better - [create a pull request!](#)

Warning: You may either use djangoCMS-icon or djangoCMS-frontend's icon contrib package but not both, since they both register an `IconPlugin`.

Icon fonts

As marked in the overview above, some MIT licenced icon fonts are bundled for convenience. They are available to the web page through static files.

For other icon sets source files are loaded from CDN through the internet by default. This is not necessarily a configuration you want to have in a production situation. To specify where to get the required css files from please use the `DJANGOCMS_FRONTEND_ICON_LIBRARIES` setting.

To just restrict the available choices of icon sets for the user use the `DJANGOCMS_FRONTEND_ICON_LIBRARIES_SHOWN` setting.

Icons can be sized. Options for icon sizes are defined by the `DJANGOCMS_FRONTEND_ICON_SIZE_CHOICES` setting.

Adding custom icon fonts

To add a custom icon font you need to generate a config file. This is a json file that tells the icon picker which icons are available. As an example check out the [config file for Bootstrap Icons](#):

```
{
  "prefix": "bi bi-",
  "icon-style": "bi",
  "list-icon": "bi bi-badge1",
  "icons": [
    "123",
    "alarm-fill",
    "alarm",
    "align-bottom",
    "align-center",
    "align-end",
    ...
  ]
}
```

Icons are rendered as `<i><>` tags with classes.

`.prefix` defines a string that is prepended to all icons. For Bootstrap icons that's the class `bi` and the prefix for the icon selecting class `bi-`.

The list `.icons` contains all available icons in the set.

`.list-icon` contains the classes for the example icon. You can probably ignore it.

`.icon-style` currently is unused. It may in future determine how icons are rendered. Currently all icons are rendered by `<i class="bla"></i>` except material design icon which are rendered by `<i class="material-icon">bla</i>`.

Using svg sprites

Currently only font-based icons are supported.

Icon plugins inside text plugins

The icon plugin is text-enabled, i.e., you can add it to a text plugin through djangoCMS-text-ckeditor's CKEDITOR. By default, however, CKEDITOR removes empty `` or `<i>` tags which most icons use.

To disable this behavior of CKEDITOR, you need to add a `customConfig` entry in `CKEDITOR_SETTINGS`, e.g.,

```
CKEDITOR_SETTINGS = {
    ...,
    "customConfig": "icon/ckeditor.icons.js",
    ...,
}
```

This will load the `ckeditor.icons.js` file which in turn will allow empty `span`` and ``i` tags.

If you already have a `customConfig` file specified it will suffice to add the following two lines to it.

```
CKEDITOR.dtd.$removeEmpty.span = 0;
CKEDITOR.dtd.$removeEmpty.i = 0;
```

Finally, you potentially need to add the necessary icon css files to the `contentCss` property of `CKEDITOR_SETTINGS`, e.g., for bootstrap icons from their cdn

```
CKEDITOR_SETTINGS = {
    ...,
    "contentCss": [
        ...,
        "https://cdn.jsdelivr.net/npm/bootstrap-icons@1.10.3/font/bootstrap-icons.css",
        ...,
    ],
    ...,
}
```

Navigation

Warning: The navigation plugin is still experimental. We believe for most use cases it is better to create custom navigation using the standard, template-based django CMS template tags like `show_menu`.

If your navigation deviates from you tree structure and needs to be managed by hand, you may want to consider using the navigation plugin.

The navigation plugin is a plugin that allows you to create a navigation. All navigation elements are child plugins of the navigation plugin: The navigation plugin itself is just a container for the navigation elements.

3.1.4 How-to guides

The how-to guides provide clear and concise instructions for users to accomplish specific tasks or solve particular problems. They aim to help with step-by-step guidance.

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):

```
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",
    }
]
```

(continues on next page)

(continued from previous page)

```
} ,
]
```

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 add your own frontend plugin

Creating the plugin

In order to integrate your custom plugins with Django CMS Frontend within your Django app, you should follow these steps.

1. **Define a Plugin Model:** As the first step, you need to define a model for your plugin. This is done in the `models.py` of your Django app. The Plugin model needs to be a **proxy model** of the Django CMS Frontend's `FrontendUIItem` class.

The plugin model defines what kind information will be associated with instances of your plugin.

Here is an example of a hypothetical plugin model:

```
# models.py
from djangocms_frontend.models import FrontendUIItem
```

(continues on next page)

(continued from previous page)

```
class YourPluginModel(FrontendUIItem):
    class Meta:
        proxy = True # MUST be a proxy model
        verbose_name = _("Your Plugin")

    def short_description(self):
        return f'"{self.field_name}"'
```

In this example, the *YourPluginModel* is a proxy of the *FrontendUIItem*, which is the base class for all Django CMS Frontend plugins. It includes a short description method that provides a description for each instance of the plugin.

Note: Keep in mind proxy models don't allow adding fields to the model. If your plugin needs to include additional fields, consider using *AbstractFrontendUIItem* as the base class and remove `proxy = True` from the Meta class.

2. **Define a Plugin Form:** This form will declare which data to store in the *FrontendUIItem*'s JSON field. The *EntangledModelForm* will automatically store the data in the JSON field. The `untangled_fields` attribute can be used to define fields that should be stored as regular model fields. This is useful for fields that are used in the frontend, but not in the backend. The `untangled_fields` attribute is optional.

It will be used in the frontend to create and edit plugin instances.

2. **Define a Plugin Form:** You should also define a form that will instruct Django on how to handle the input for creating and editing instances of your plugin. The form should specify which data will be stored in the *FrontendUIItem*'s JSON field.

Here is an example of a form for the *YourPluginModel*:

```
# forms.py
from django import forms
from djangoCMS_frontend.models import FrontendUIItem
from entangled.forms import EntangledModelForm

class YourPluginForm(EntangledModelForm):
    class Meta:
        model = FrontendUIItem
        entangled_fields = {
            "config": [
                "field_name", # This field will be stored in the config JSON
            ]
        }
        # untangled_fields = ("tag_type",) # Only if you use the tag_type field
        field_name = forms.CharField(max_length=50)
```

3. **Create a Plugin Class:** In the same app, create a file named `cms_plugins.py`. Inside this file, define a class for your plugin by extending *CMSPluginBase*.

```
# cms_plugins.py
from djangoCMS_frontend.cms_plugins import CMSUIPlugin
from cms.plugin_pool import plugin_pool
from . import models, forms
```

(continues on next page)

(continued from previous page)

```
@plugin_pool.register_plugin
class YourPlugin(CMSUIPlugin):
    model = models>YourPluginModel
    form = forms>YourPluginForm
    name = "Your Plugin Name"
    render_template = "your_plugin_template.html"

    fieldsets = [
        # All fields must be listed in the form, either as entangled or untangled
        # fields.
        (None, {
            "fields": [
                "field_name",
            ]
        }),
    ]

    def render(self, context, instance, placeholder):
        context.update({"instance": instance})
        return context
```

4. **Create a Plugin Template:** Create an HTML template for your plugin in your app's templates directory. This template will define how your plugin is rendered on the page.

```
<!-- your_plugin_template.html -->
<div class="your-plugin-class">
    {{ instance.field_name }} or {{ instance.config.field_name }}
</div>
```

The “entangled” fields in the JSON config can either be accessed using `instance.config.field_name` or by using the `instance.field_name` syntax. The

latter is only possible if the field model does not have a property with the same name.

Remember, developing custom plugins requires a good understanding of Django's and Django CMS's architecture. Additionally, consider the security implications of your plugin, especially if it handles user input.

Extending the plugin

django CMS Frontend comes with a set of mixins that can be used to extend the functionality of your plugin. These mixins are:

- **Attributes:** Adds a set of attributes to the plugin. Attributes are key-value pairs that can be used to store additional data in the plugin. Attributes are stored in the `attributes` JSON field of the `FrontendUIItem` model.
- **Background:** Adds background formatting to the plugin.
- **Responsive:** Adds responsive formatting to the plugin.
- **Spacing:** Adds spacing formatting to the plugin.
- **Sizing:** Adds sizing formatting to the plugin.

- **Title:** Adds an optional title to the plugin which can be used to display a title above the plugin or just to simplify the navigation of the plugin tree.

Each mixin comes in two flavours, one for the plugin and one for the plugin form. The plugin mixin is used to add the functionality to the plugin, while the form mixin is used to add their fields to the plugin form. The mixins are designed to be used together.

For example, if you want to use the attributes mixin, you need to add the `AttributesMixin` to your plugin and the `AttributesMixinForm` to your plugin form:

```
from djangoCMS_frontend.cms_plugins import AttributesMixin, AttributesMixinForm

class YourPlugin(AttributesMixin, CMSUIPlugin):
    ...

class YourPluginForm(AttributesMixinForm, EntangledModelForm):
    ...
```

Re-using links and images

django CMS Frontend comes with a set of classes that can be used to re-use links and images in your plugin. These mixins are:

- **LinkPluginMixin:** Adds a link to the plugin. The link can be used to link the plugin to a page, a file or an external URL. Include **GetLinkMixin** with your plugin model and base the admin form on **AbstractLinkForm** (can also be used as a mixin):

```
from djangoCMS_frontend.contrib.link.cms_plugins import LinkPluginMixin
from djangoCMS_frontend.contrib.link.models import GetLinkMixin
from djangoCMS_frontend.contrib.link.forms import AbstractLinkForm

class YourPlugin(LinkPluginMixin, CMSUIPlugin):
    ...

class YourPluginForm(AbstractLinkForm):
    link_is_optional = False # True, if the link is optional
    ...

class YourPluginModel(GetLinkMixin, FrontendUIItem):
    ...
```

- **ImageMixin:** Adds an image to the plugin *model*. Base your plugin form on **ImageForm** (can also be used as a mixin):

```
from djangoCMS_frontend.contrib.image.models import ImageMixin
from djangoCMS_frontend.contrib.image.forms import ImageForm

class YourPluginForm(ImageForm):
    ...

class YourPluginModel(ImageMixin, FrontendUIItem):
    image_field = "image" # The name of the image field in the config JSON
    ...
```


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
```

(continues on next page)

(continued from previous page)

```
└─ 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`):

```
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 = (
    ("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="",
    )
```

(continues on next page)

(continued from previous page)

```

        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`:

```

from django.utils.translation import gettext as _
from.djangocms_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")

```

(continues on next page)

(continued from previous page)

```
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 sekizai_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 #}
```

(continues on next page)

(continued from previous page)

```
{% 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:

```
/* 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 add the tab editing style to your custom 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 %}
```

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
- Add custom templates

How to migrate other plugin packages

The management command `migrate` converts any plugin from **djangoCMS_bootstrap4** and **djangoCMS_styled_link** to **djangoCMS-frontend**. This behaviour can be extended adding custom migration modules to the `DJANGOCMS_FRONTEND_ADDITIONAL_MIGRATIONS` setting.

A migration module must contain this three objects:

plugin_migrations

Dictionary with the configuration of migration process for each plugin class.

data_migration

Dictionary with methods to transform attributes of the plugins.

plugin_prefix

String with the prefix of the plugin_types that are being migrated. The migration process alerts if there are remaining plugins with this prefix.

Check the source code of `management/bootstrap4_migration.py` to get more details about this three objects.

3.1.5 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 from 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.

Note: The following settings of `djangoCMS-picture` are respected.

`settings.DJANGOCMS_PICTURE_ALIGN`

You can override alignment styles with `DJANGOCMS_PICTURE_ALIGN`, for example:

```
DJANGOCMS_PICTURE_ALIGN = [
    ('top', _('Top Aligned')),
]
```

This will generate a class prefixed with `align-`. The example above would produce a `class="align-top"`. Adding a class key to the image attributes automatically merges the alignment with the attribute class.

`settings.DJANGOCMS_PICTURE_RATIO`

You can use `DJANGOCMS_PICTURE_RATIO` to set the width/height ratio of images if these values are not set explicitly on the image:

```
DJANGOCMS_PICTURE_RATIO = 1.618
```

We use the [golden ratio](#), approximately 1.618, as a default value for this.

`settings.DJANGOCMS_PICTURE_RESPONSIVE_IMAGES`

You can enable responsive images technique by setting `DJANGOCMS_PICTURE_RESPONSIVE_IMAGES` to `True`.

`settings.DJANGOCMS_PICTURE_RESPONSIVE_IMAGES_VIEWPORT_BREAKPOINTS`

If `DJANGOCMS_PICTURE_RESPONSIVE_IMAGES` is set to `True`, uploaded images will create thumbnails of different sizes according to `DJANGOCMS_PICTURE_RESPONSIVE_IMAGES_VIEWPORT_BREAKPOINTS` (which defaults to `[576, 768, 992]`) and browser will be responsible for choosing the best image to display (based upon the screen viewport).

`settings.DJANGOCMS_PICTURE_TEMPLATES`

This add-on provides a default template for all instances. You can provide additional template choices by adding a `DJANGOCMS_PICTURE_TEMPLATES` setting:

```
DJANGOCMS_PICTURE_TEMPLATES = [
    ('background', _('Background image')),
]
```

You'll need to create the `background` folder inside `templates/djangocms_picture/` otherwise you will get a *template does not exist* error. You can do this by copying the default folder inside that directory and renaming it to `background`.

`settings.TEXT_SAVE_IMAGE_FUNCTION`

If you want to use `djangoCMS-text-ckeditor`'s [Drag & Drop Images](#) so be sure to set `TEXT_SAVE_IMAGE_FUNCTION`:

```
TEXT_SAVE_IMAGE_FUNCTION = 'djangoCMS_frontend.contrib.image.image_save.create_
↪image_plugin'
```

Otherwise set `TEXT_SAVE_IMAGE_FUNCTION = None`

settings.DJANGOCMS_FRONTEND_ICON_LIBRARIES

Default:

```
DJANGOCMS_FRONTEND_ICON_LIBRARIES = {
    'font-awesome': (
        'font-awesome.min.json',
        'https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0/css/all.min.css'
    ),
    'bootstrap-icons': (
        'bootstrap-icons.min.json',
        'https://cdn.jsdelivr.net/npm/bootstrap-icons@1.10.3/font/bootstrap-icons.
↪css'
    ),
    'material-icons-filled': (
        'material-icons-filled.min.json',
        'https://fonts.googleapis.com/css2?family=Material+Icons'
    ),
    ...
}
```

For each available icon set there is an entry in this dictionary. The key is the basis for the displayed name. The value is a 2-tuple:

1. The name of the config file which is a static file with the path `djangoCMS_frontend/icon/vendor/assets/icon-libraries/`.
2. The name of the css file defining the icons. It is either a path or a file name. If it is a file name the css file is fetched from `djangoCMS_frontend/icon/vendor/assets/stylesheets/`.

settings.DJANGOCMS_FRONTEND_ICON_LIBRARIES_SHOWN

Default:

```
DJANGOCMS_FRONTEND_ICON_LIBRARIES_SHOWN = (
    "font-awesome",
    "bootstrap-icons",
    "material-icons-filled",
    "material-icons-outlined",
    "material-icons-round",
    "material-icons-sharp",
    "material-icons-two-tone",
    "fomantic-ui",
    "foundation-icons",
    "elegant-icons",
    "feather-icons",
    "open-iconic",
    "tabler-icons",
    "weather-icons",
)
```

This settings allows to restrict the number of icon sets shown to the user. Typically one or two icon sets should be sufficient to keep a consistent icon experience.

Warning: This setting only has an effect if `DJANGOCMS_FRONTEND_ICON_LIBRARIES` is not explicitly set.

`settings.DJANGOCMS_FRONTEND_ICON_SIZE_CHOICES`

Default:

```
DJANGOCMS_FRONTEND_ICON_SIZE_CHOICES = (
    ("", _("Regular")),
    ("200%", _("x 2")),
    ("300%", _("x 3")),
    ("400%", _("x 4")),
    ("500%", _("x 5")),
    ("800%", _("x 8")),
    ("1200%", _("x 12")),
)
```

This list of options defines the icon size choices a user can select. The values (first tuple element) are CSS units for the `font-size` CSS property. Besides relative units (%) any CSS unit can be used, e.g. 112pt.

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 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 ``. 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**. Other packages can be migrated adding custom migration modules to the `DJANGOCMS_FRONTEND_ADDITIONAL_MIGRATIONS` setting.

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
```

3.1.6 Index

3.2 Indices and tables

- [Index](#)
- [search](#)

A

Accordion, 19
 add_classes() (*FrontendUIItem* method), 47
 Alert, 20

B

Badge, 21
 base.html, 7
 Blockquote, 28
 Button, 26
 ButtonGroup (*built-in class*), 48

C

Card, 21
 CardInner, 21
 CardLayout, 21
 Carousel, 25
 Code, 28
 ColoredButtonGroup (*built-in class*), 48
 Column, 18
 config (*FrontendUIItem* attribute), 47
 Container, 17
 Create a theme, 37

D

DeviceChoiceField (*built-in class*), 48
 DJANGOCMS_FRONTEND_ADMIN_CSS (*settings attribute*), 44
 DJANGOCMS_FRONTEND_CAROUSEL_ASPECT_RATIOS (*settings attribute*), 44
 DJANGOCMS_FRONTEND_CAROUSEL_TEMPLATES (*settings attribute*), 43
 DJANGOCMS_FRONTEND_COLOR_STYLE_CHOICES (*settings attribute*), 44
 DJANGOCMS_FRONTEND_GRID_CONTAINERS (*settings attribute*), 43
 DJANGOCMS_FRONTEND_GRID_SIZE (*settings attribute*), 43
 DJANGOCMS_FRONTEND_ICON_LIBRARIES (*settings attribute*), 46
 DJANGOCMS_FRONTEND_ICON_LIBRARIES_SHOWN (*settings attribute*), 46

DJANGOCMS_FRONTEND_ICON_SIZE_CHOICES (*settings attribute*), 47
 DJANGOCMS_FRONTEND_JUMBOTRON_TEMPLATES (*settings attribute*), 43
 DJANGOCMS_FRONTEND_LINK_MODELS (*settings attribute*), 32
 DJANGOCMS_FRONTEND_LINK_TEMPLATES (*settings attribute*), 43
 DJANGOCMS_FRONTEND_MINIMUM_INPUT_LENGTH (*settings attribute*), 45
 DJANGOCMS_FRONTEND_SPACER_SIZES (*settings attribute*), 44
 DJANGOCMS_FRONTEND_TAB_TEMPLATES (*settings attribute*), 43
 DJANGOCMS_FRONTEND_TAG_CHOICES (*settings attribute*), 43
 DJANGOCMS_FRONTEND_USE_ICONS (*settings attribute*), 43
 DJANGOCMS_PICTURE_ALIGN (*settings attribute*), 45
 DJANGOCMS_PICTURE_RATIO (*settings attribute*), 45
 DJANGOCMS_PICTURE_RESPONSIVE_IMAGES (*settings attribute*), 45
 DJANGOCMS_PICTURE_RESPONSIVE_IMAGES_VIEWPORT_BREAKPOINTS (*settings attribute*), 45
 DJANGOCMS_PICTURE_TEMPLATES (*settings attribute*), 45

E

Extend plugins, 33

F

Figure, 28
 FormMixins, 37
 FrontendUIItem (*built-in class*), 47

G

get_attributes() (*FrontendUIItem* method), 48
 get_short_description() (*FrontendUIItem* method), 48

H

How-to, 31

I

Icon, [28](#)

IconGroup (*built-in class*), [48](#)

IconMultiselect (*built-in class*), [48](#)

Image, [27](#)

initialize_from_form() (*FrontendUIItem method*),
[48](#)

Installation, [5](#)

J

Jumbotron, [26](#)

L

Link, [26](#)

M

manage.py, [9](#)

migrate, [9](#)

Migration from Bootstrap 4, [9](#)

O

OptionalDeviceChoiceField (*built-in class*), [48](#)

P

Picture, [27](#)

Plugins, [19](#)

R

RenderMixins, [37](#)

Row, [17](#)

S

Spacer, [27](#)

Spacing, [27](#)

T

Tabs, [28](#)

tag_type (*FrontendUIItem attribute*), [47](#)

TEXT_SAVE_IMAGE_FUNCTION (*settings attribute*), [45](#)

Themes, [37](#)

U

ui_item (*FrontendUIItem attribute*), [47](#)

W

Working example, [37](#)