djangocms-cascade Documentation

Release 0.12.5

Jacob Rief

May 09, 2017

Contents

1 Project's home **1**

2 Project's goals **3**

3 Contents: **5**

3.1 For the Impatient . . . 5
3.2 Introduction . . . 6
3.3 Installation . . . 7
3.4 Link Plugin . . . 10
3.5 Bootstrap 3 Grid system . . . 13
3.6 Gallery . . . 30
3.7 Using Fonts with Icons . . . 30
3.8 Handling the client side . . . 32
3.9 Section Bookmarks . . . 34
3.10 Segmentation of the DOM . . . 35
3.11 Working with sharable fields . . . 37
3.12 Customize CSS classes and inline styles . . . 39
3.13 Choose an alternative rendering template . . . 43
3.14 Conditionally hide some plugin . . . 44
3.15 The CMS Clipboard . . . 44
3.16 Extending Cascade . . . 46
3.17 Generic Plugins . . . 50
3.18 Release History . . . 51

4 Indices and tables **61**

CHAPTER 1

Project's home

Check for the latest release of this project on Github.

Please report bugs or ask questions using the Issue Tracker.

CHAPTER 2

Project's goals

1. Create a modular system, which allows programmers to add simple widget code, without having to implement an extra djangoCMS plugins for each of them.
2. Make available a meaningful subset of widgets as available for the most common CSS frameworks, such as Twitter Bootstrap. With these special plugins, in many configurations, **djangoCMS** can be operated using one single template, containing one generic placeholder.
3. Extend this **djangoCMS** plugin, to be used with other CSS frameworks such as Foundation 5, Unsemantic and others.
4. Use the base functionality of **djangoCMS-Cascade** to easily add special plugins. For instance, djangoSHOP implements all its cart and checkout specific forms this way.

CHAPTER 3

Contents:

For the Impatient

This HowTo gives you a quick instruction on how to get a demo of **djangocms-cascade** up and running. It also is a good starting point to ask questions or report bugs, since its backend is used as a fully functional reference implementation, used by the unit tests of project.

Create a Python Virtual Environment

To keep environments separate, create a virtual environment and install external dependencies. Missing packages with JavaScript files and Style Sheets, which are not available via pip must be installed via npm:

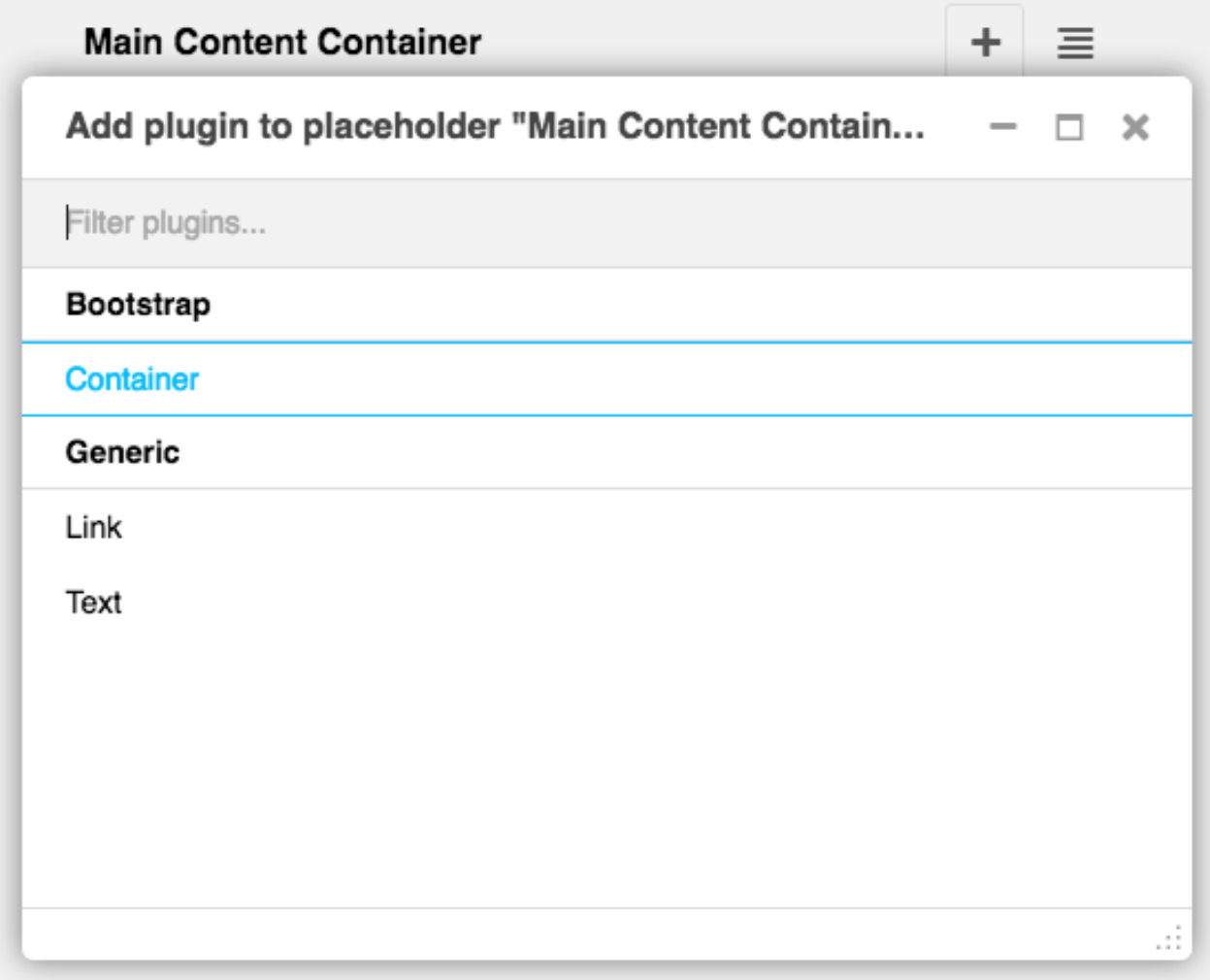
```
$ git clone --depth=1 https://github.com/jrief/djangocms-cascade.git
$ cd djangocms-cascade
$ virtualenv cascadenv
$ source cascadenv/bin/activate
(cascadenv)$ pip install -r requirements/django110.txt
```

Initialize the database, create a superuser and start the development server:

```
$ cd examples
$ npm install
$ ./manage.py migrate
$ ./manage.py createsuperuser
$ ./manage.py runserver
```

Point a browser to http://localhost:8000/admin/login/?next=/ and log in as the super user you just created. Hit "next" and fill out the form to create your first page. Afterwards, click **Structure** on the top of the page. Now a heading named **Main Content** appears. This heading symbolizes our main **djangoCMS** Placeholder.

Locate the plus sign right to the heading and click on it. From its context menu select **Container** located in the section **Bootstrap**:



This brings you into the editor mode for a Bootstrap container. To this container you may add one or more Bootstrap **Rows**. Inside these rows you may organize the layout using some Bootstrap **Columns**.

Please proceed with the detailled explanation on how to use the *Bootstrap's grid* system within **djangocms-cascade**.

Introduction

DjangoCMS-Cascade is a collection of plugins for Django-CMS >=3.3 to add various HTML elements from CSS frameworks, such as Twitter Bootstrap to the Django templatetag placeholder. This Django App makes it very easy to add other CSS frameworks, or to extend an existing collection with additional elements.

DjangoCMS-Cascade allows web editors to layout their pages, without having to create different Django templates for each layout modification. In most cases, one template with one single placeholder is enough. The editor then can subdivide that placeholder into rows and columns, and add additional DOM elements such as buttons, rulers, or even the Bootstrap Carousel. Some basic understanding on how the DOM works is required though.

Twitter Bootstrap is a well documented CSS framework which gives web designers lots of possibilities to add a consistent structure to their pages. This collection of Django-CMS plugins offers a subset of these predefined elements to web designers.

Extensibility

This module requires one database table with one column to store all data in a JSON object. All **DjangoCMS-Cascade** plugins share this same model, therefore they can be easily extended, because new data structures are added to that JSON object without requiring a database migration.

Another three database tables are required for additional optional features.

Naming Conflicts

Some **djangoCMS** plugins may use the same name as plugins from **djangocms-cascade**. To prevent confusion, since version 0.7.2, all Cascade plugins as prefixed with a (koppa) symbol. This can be deactivated or changed by setting `CMSPLUGIN_CASCADE['plugin_prefix']` to `False` or any other symbol.

Installation

Install the latest stable release

```
$ pip install djangocms-cascade
```

or the current development release from github

```
$ pip install -e git+https://github.com/jrief/djangocms-cascade.git#egg=djangocms-
↪cascade
```

Dependencies

Due to some incompatibilities in the API of Django, django-CMS and djangocms-text-ckeditor, please only use this combination of external dependencies:

djangocms-cascade-0.11.x

- Django >=1.8, <=1.9
- DjangoCMS >=3.2, <=3.3
- djangocms-text-ckeditor == 3.0

djangocms-cascade-0.12.x

- Django >=1.9
- DjangoCMS >=3.4.3
- djangocms-text-ckeditor >= 3.3

Create a database schema

```
./manage.py migrate cmsplugin_cascade
```

Install Bootstrap

Since the Bootstrap CSS and JavaScript files are part of their own repository, they are not shipped within this package. Furthermore, as they are not part of the PyPI network, they have to be installed through another package manager, namely bower.

```
cd djangocms-cascade
bower install --require
```

Alternatively copy the installed `bower_components` into a directory of your project or to any other meaningful location, but ensure that the directory `bower_components` can be found by your StaticFileFinder. In doubt, add that directory to your `STATICFILES_DIRS`:

```
STATICFILES_DIRS = (
    os.path.abspath(os.path.join(MY_PROJECT_DIR, 'bower_components')),
)
```

Configuration

Add `'cmsplugin_cascade'` to the list of `INSTALLED_APPS` in the project's `settings.py` file. Optionally add 'cmsplugin_cascade.extra_fields' and/or 'cmsplugin_cascade.sharable' to the list of `INSTALLED_APPS`. Make sure that these entries are located before the entry `cms`.

Configure the CMS plugin

```
INSTALLED_APPS = (
    ...
    'cmsplugin_cascade',
    'cmsplugin_cascade.clipboard',  # optional
    'cmsplugin_cascade.extra_fields',  # optional
    'cmsplugin_cascade.sharable',  # optional
    'cmsplugin_cascade.segmentation',  # optional
    'cms',
    ...
)
```

Activate the plugins

By default, no **djangocms-cascade** plugins is activated. Activate them in the project's `settings.py` with the directive `CMSPLUGIN_CASCADE_PLUGINS`.

To activate all available Bootstrap plugins, use:

```
CMSPLUGIN_CASCADE_PLUGINS = ('cmsplugin_cascade.bootstrap3',)
```

If for some reason, only a subset of the available Bootstrap plugins shall be activated, name each of them. If for example only the grid system shall be used, but no other Bootstrap plugins, then configure:

```
CMSPLUGIN_CASCADE_PLUGINS = ('cmsplugin_cascade.bootstrap3.container',)
```

A very useful plugin is the **LinkPlugin**. It supersedes the djangocms-link-plugin, normally used together with the CMS.

```
CMSPLUGIN_CASCADE_PLUGINS += ('cmsplugin_cascade.link',)
```

Generic Plugins which are not opinionated towards a specific CSS framework, are kept in a separate folder. It is strongly suggested to always activate them:

```
CMSPLUGIN_CASCADE_PLUGINS = ('cmsplugin_cascade.generic',)
```

Sometimes it is useful to do a segmentation. Activate this by adding its plugin:

```
CMSPLUGIN_CASCADE_PLUGINS = ('cmsplugin_cascade.segmentation',)
```

Restrict plugins to a particular placeholder

Warning: You **must** set `parent_classes` for your placeholder, else you won't be able to add a container to your placeholder. This means that as an absolute minimum, you must add this to your settings:

```
CMS_PLACEHOLDER_CONF = {
    'content': {
        'parent_classes': {'BootstrapContainerPlugin': None,},
    },
}
```

Unfortunately **djangoCMS** does not allow to declare dynamically which plugins are eligible to be added as children of other plugins. This is determined while bootstrapping the Django project and thus remain static. We therefore must somehow trick the CMS to behave as we want.

Say, our Placeholder named "Main Content" shall accept the **BootstrapContainerPlugin** as its only child, we then must use this CMS settings directive:

```
CMS_PLACEHOLDER_CONF = {
    'Main Content Placeholder': {
        'plugins': ['BootstrapContainerPlugin'],
        'text_only_plugins': ['TextLinkPlugin'],
        'parent_classes': {'BootstrapContainerPlugin': None},
        'glossary': {
            'breakpoints': ['xs', 'sm', 'md', 'lg'],
            'container_max_widths': {'xs': 750, 'sm': 750, 'md': 970, 'lg': 1170},
            'fluid': False,
            'media_queries': {
                'xs': ['(max-width: 768px)'],
                'sm': ['(min-width: 768px)', '(max-width: 992px)'],
                'md': ['(min-width: 992px)', '(max-width: 1200px)'],
                'lg': ['(min-width: 1200px)'],
            },
        },
    },
}
```

Here we add the **BootstrapContainerPlugin** to `plugins` and `parent_classes`. This is because the Container plugin normally is the root plugin in a placeholder. If this plugin would not restrict its parent plugin classes, we would be allowed to use it as a child of any plugin. This could destroy the page's grid.

Note: Until version 0.7.1 the Container plugin did not restrict it's `parent_classes` and therefore we did not have

to add it to the `CMS_PLACEHOLDER_CONF` settings.

Furthermore, in the above example we must add the **TextLinkPlugin** to `text_only_plugins`. This is because the **TextPlugin** is not part of the Cascade ecosystem and hence does not know which plugins are allowed as its children.

The dictionary named `glossary` sets the initial parameters of the *Bootstrap 3 Grid system*.

Define the leaf plugins

Leaf plugins are those, which contain real data, say text or images. Hence the default setting is to allow the **TextPlugin** and the **FilerImagePlugin** as leafs. This can be overridden using the configuration directive

```
CMSPLUGIN_CASCADE = {
    ...
    'alien_plugins': ('TextPlugin', 'FilerImagePlugin', 'OtherLeafPlugin',),
    ...
}
```

Bootstrap 3 with AngularJS

Some Bootstrap3 plugins can be rendered using templates which are suitable for the very popular Angular UI Bootstrap framework. This can be done during runtime; when editing the plugin a select box appears which allows to chose an alternative template for rendering.

Template Customization

Make sure that the style sheets are referenced correctly by the used templates. DjangoCMS requires Django-Sekizai to organize these includes, so a strong recommendation is to use that Django app.

The templates used for a DjangoCMS project shall include a header, footer, the menu bar and optionally a breadcrumb, but should leave out an empty working area. When using HTML5, wrap this area into an `<article>` or `<section>` element or just use it unwrapped (suggested). This placeholder shall be named using a generic identifier, for instance "Main Content" or similar:

```
{% load cms_tags %}

<!-- wrapping element (optional) -->
    {% placeholder "Main Content" %}
<!-- /wrapping element -->
```

From now on, the page layout can be adopted inside this placeholder, without having to fiddle with template coding anymore.

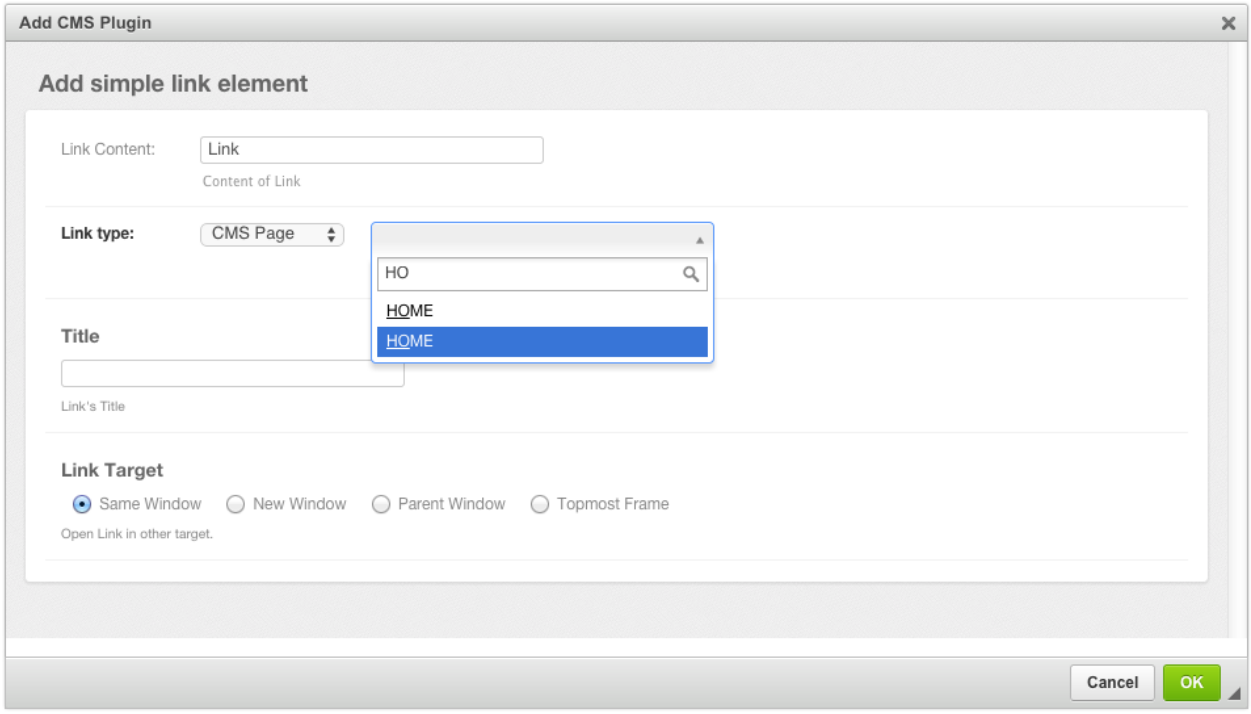
Link Plugin

djangocms-cascade ships with its own link plugin. This is because other plugins from the Cascade eco-system, such as the **BootstrapButtonPlugin**, the **BootstrapImagePlugin** or the **BootstrapPicturePlugin** also require the functionality to set links to internal- and external URLs. Since we do not want to duplicate the linking functionality for each of these plugins, it has been moved into its own base class. Therefore we will use the terminology **TextLinkPlugin** when referring to text-based links.

The de-facto plugin for links, djangocms-link can't be used as a base class for these plugins, hence an alternative implementation has been created within the Cascade framework. The link related data is stored in a sub-dictionary named `link` in our main JSON field.

Prerequisites

Before using this plugin, assure that `'cmsplugin_cascade.link'` is member of the list or tuple `CMSPLUGIN_CASCADE_PLUGINS` in the project's `settings.py`.



The behavior of this Plugin is what you expect from a Link editor. The field **Link Content** is the text displayed between the opening and closing `<a>` tag. If used in combination with djangocms-text-ckeditor the field automatically is filled out.

By changing the **Link type**, the user can choose between three types of Links:

- Internal Links pointing to another page inside the CMS.
- External Links pointing to a valid Internet URL.
- Links pointing to a valid e-mail address.

The optional field **Title** can be used to add a `title="some value"` attribute to the `<a href ...>` element.

With **Link Target**, the user can specify, whether the linked content shall open in the current window or if the browser shall open a new window.

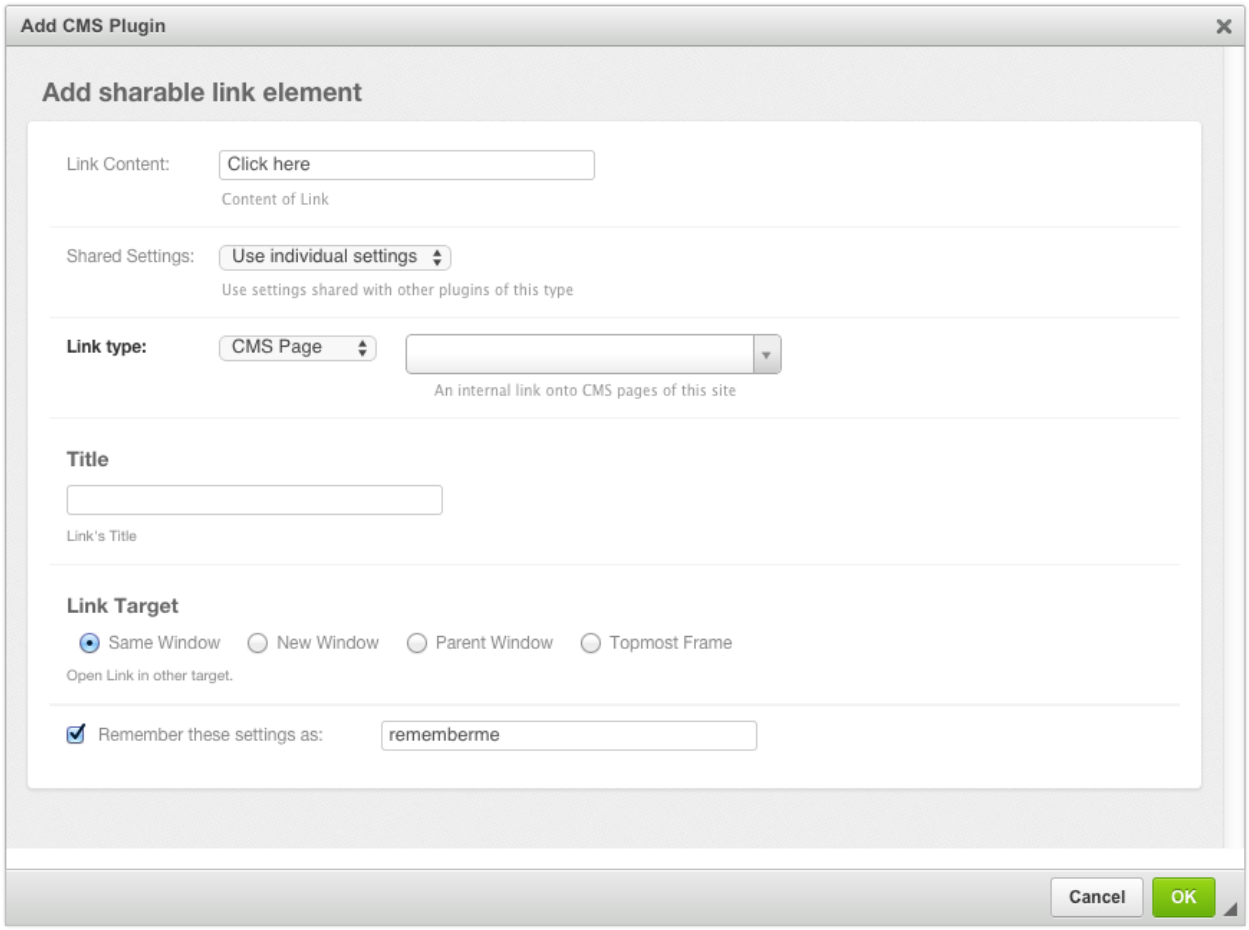
Link Plugin with sharable fields

If your web-site contains many links pointing onto external URLs, you might want to refer to them by a symbolic name, rather than having to reenter the URL repeatedly. With **djangocms-cascade** this can be achieved easily by declaring some of the plugin's fields as "sharable".

Assure that `INSTALLED_APPS` contain `'cmsplugin_cascade.sharable'`, then redefine the **TextLinkPlugin** to have sharable fields in `settings.py`:

```
CMSPLUGIN_CASCADE = {
    ...
    'plugins_with_sharables':
        ...
        'TextLinkPlugin':  ('link',),  # and optionally other fields
        ...
    },
    ...
}
```

This will change the Link Plugin's editor slightly. Note the extra field added to the bottom of the form.



Now the URL for this ink entity is stored in a central entity. This feature is useful, if for instance the URL of an external web page may change in the future. Then the administrator can change that link in the administration area once, rather than having to go through all the pages and check if that link was used.

To retain the Link settings, click onto the checkbox *Remember these settings as: ...* and give it a name of your choice. The next time your create a Shared Link element, you may select a previously named settings from the select field *Shared Settings*. Since these settings can be shared among other plugins, these input fields are disabled and can't be changed anymore.

Changing shared settings

The settings of a shared plugin can be changed globally, for all plugins using them. To edit such a shared setting, in the Django Admin, go into the list view for **Home › Cmsplugin_cascade › Shared between Plugins** and choose the named shared settings.

Please note, that each plugin type can specify which fields shall be sharable between its plugins. In this example, only the Link itself is shared, but one could configure **djangocms-cascade** to also share the `title` and/or the link's `target` tags.

Then only these fields are editable in the detail view **Shared between Plugins**. The interface for other shared plugin may vary substantially, depending of their type definition.

Extending the Link Plugin

While programming third party modules for Django, one might have to access a model instance through a URL and thus add the method get_absolute_url to that Django model. Since such a URL is neither a CMS page, nor a URL to an external web page, it would be convenient to access that model using a special Link type.

For example, in django-shop we can allow to link directly from a CMS page to a shop's product. This is achieved by reconfiguring the Link Plugin inside Cascade with:

```
CMSPLUGIN_CASCADE = {
    ...
    'dependencies': {
        'shop/js/admin/shoplinkplugin.js': 'cascade/js/admin/linkplugin.js',
    },
    'link_plugin_classes': (
        'shop.cascade.plugin_base.CatalogLinkPluginBase',
        'cmsplugin_cascade.link.plugin_base.LinkElementMixin',
        'shop.cascade.plugin_base.CatalogLinkForm',
    ),
}
```

The tuple specified through `link_plugin_classes` replaces the base class for the LinkPlugi class and the form class for its editor. Please refer to the django-shop for implementation details of the classes.

Now the select box for **Link type** will offer one additional option: "Product". When this is selected, the site administrator can choose between all of the shops products.

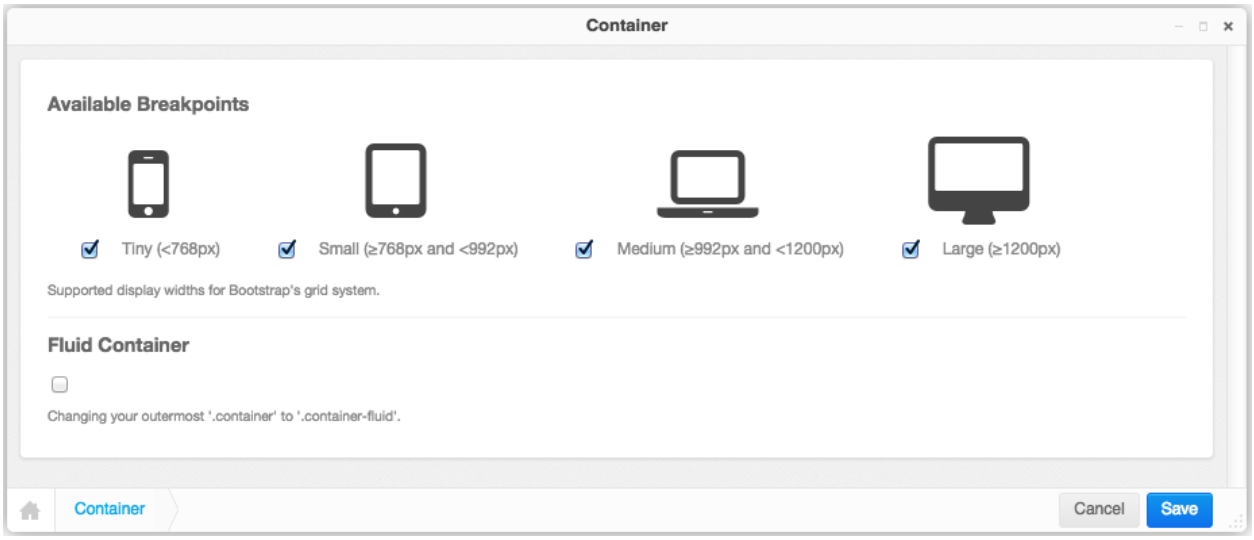
Bootstrap 3 Grid system

In order to take full advantage of **djangocms-cascade**, you should be familiar with the concepts of the Bootstrap Grid System, since all other Bootstrap components depend upon.

Bootstrap Container

A **Container** is the outermost component the Bootstrap framework knows of. Here the designer can specify the breakpoints of a web page. By default, Bootstrap offers 4 breakpoints: "large", "medium", "small" and "tiny". These determine for which kind of screen widths, the grid system may switch the layout.

The editor window for a Container element offers the possibility to deactivate certain breakpoints. While this might make sense under certain conditions, it is safe to always keep all four breakpoints active, since this gives the designer of the web page the maximum flexibility.



Small devices exclusively

If the web page shall be optimized just for small but not for large devices, then disable the breakpoints for **Large** and/or **Medium**. In the project's style-sheets, the maximum width of the container element then must be reduced to that chosen breakpoint:

```
@media(min-width: 1200px) {
  .container {
    max-width: 970px;
  }
}
```

or, if you prefers the SASS syntax:

```
@media(min-width: $screen-lg) {
  .container {
    max-width: $container-desktop;
  }
}
```

Large devices exclusively

If the web page shall be optimized just for large but not for small devices, then disable the breakpoints for **Tiny** and/or **Small**.

Changing the style-sheets then is not required for this configuration setting.

Fluid Container

A variant of the normal Bootstrap Container is the Fluid Container. It can be enabled by a checkbox in the editors window. Fluid Containers have no hards breakpoints, they adopt their width to whatever the browser pretends and are slightly larger than their non-fluid counterpart.

A fluid container makes it impossible to determine the maximum width of responsive images for the *large media breakpoint*, because it is applied whenever the browser width extends 1200 pixels, but there is no upper limit. For responsive images in the smaller breakpoints ("tiny", "small" and "medium") we use the width of the next larger

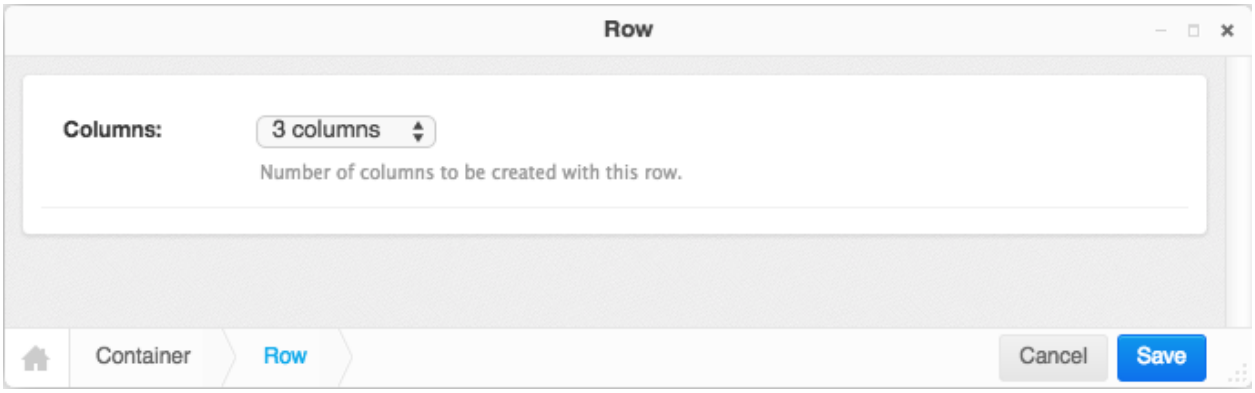
breakpoint, but for images in the "large" media breakpoints we somehow must specify an arbitrary maximum width. The default width is set to 1980 pixels, but can be changed, to say 2500 pixels, using the following configuration in your `settings.py`:

```
CMSPLUGIN_CASCADE = {
    ...
    'bootstrap3': (
        ('xs', (768, 'mobile', _("mobile phones"), 750, 768)),
        ('sm', (768, 'tablet', _("tablets"), 750, 992)),
        ('md', (992, 'laptop', _("laptops"), 970, 1200)),
        ('lg', (1200, 'desktop', _("large desktops"), 1170, 2500)),
    ),
}
```

Note: Fluid container are specially useful for Hero images, full-width Carousels and the Jumbotron plugin. When required, add a free standing fluid container to the placeholder and as it's only child, use the picture or carousel plugin. Its content then is stretched to the browser's full width.

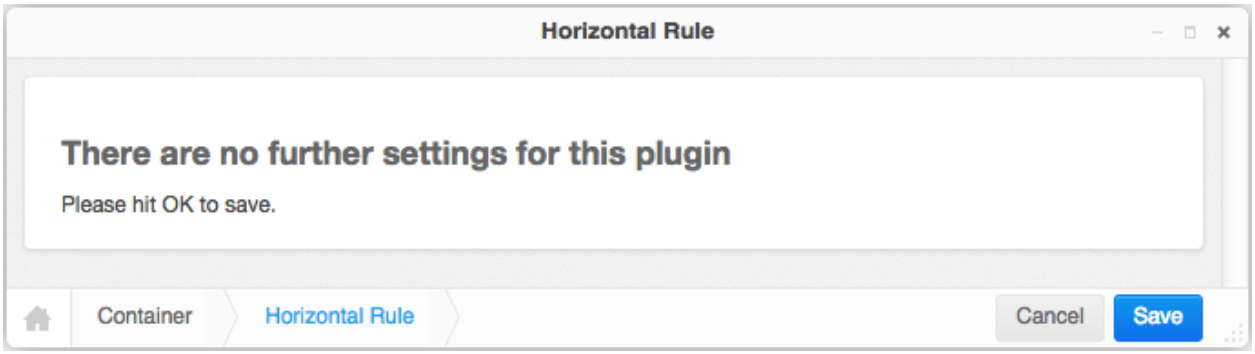
Bootstrap Row

Each Bootstrap Container may contain one or more Bootstrap Rows. A row does not accept any configuration setting. However, while editing, one can specify the number of columns. When adding or changing a row, then this number of columns are added if its value exceeds the current number of columns. Reducing the number of columns does not delete any of them; they must explicitly be chosen from the context menu in structure view.



Horizontal Rule

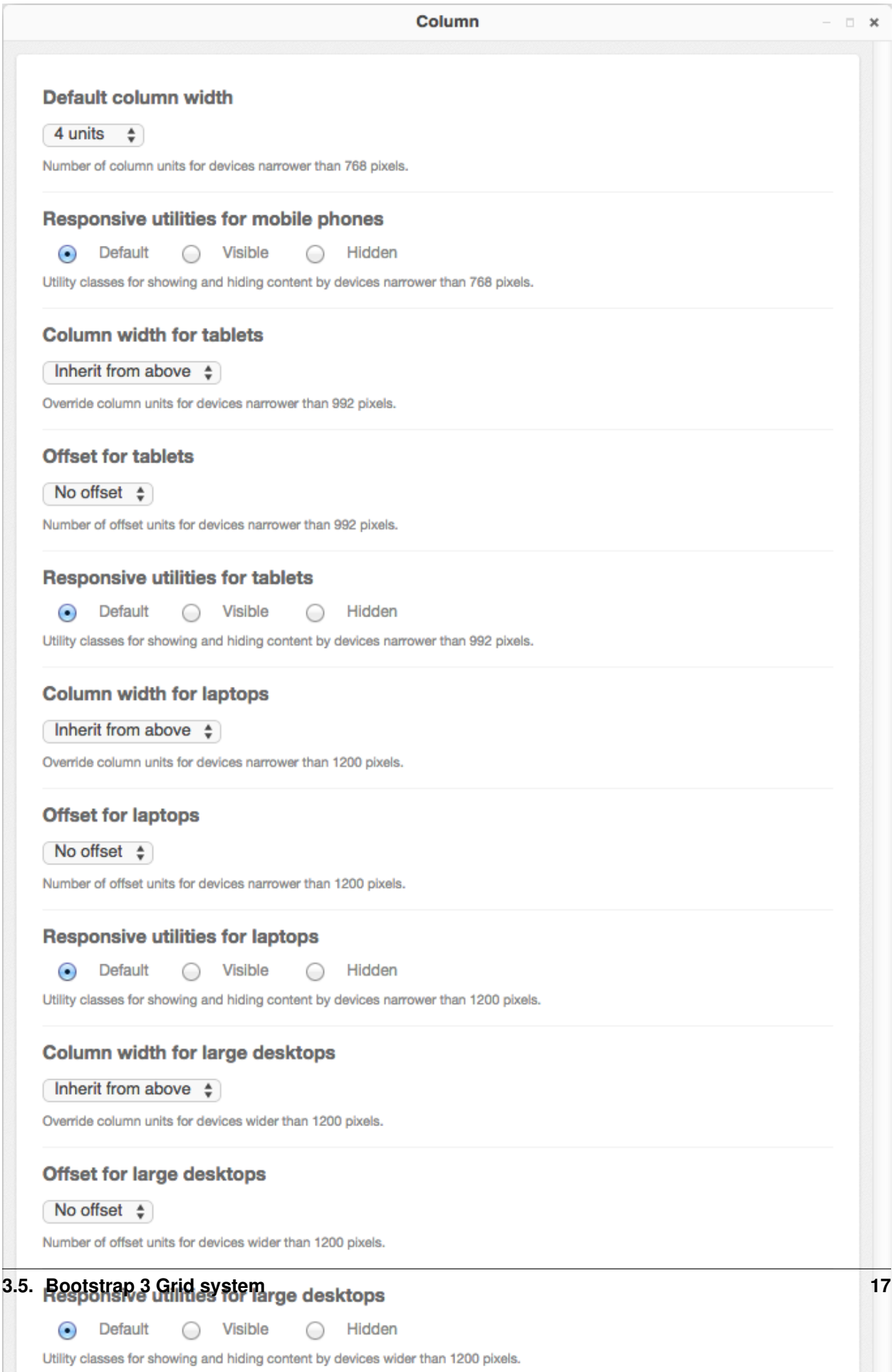
A horizontal rule is used to separate rows optically from each other.



Column

In the column editor, one can specify the width, the offset and the visibility of each column. These values can be set for each of the four breakpoints (*tiny*, *small*, *medium* and *large*), as specified by the Container plugin.

At the beginning this may feel rather complicate, but consider that **Bootstrap 3 is mobile first**, therefore all column settings, *first* are applied to the narrow breakpoints, which *later* can be overridden for larger breakpoints at a later stage. This is the reason why this editor starts with the *column widths* and *column offsets* for tiny rather than for large displays.


Column
Default column width
4 units
Number of column units for devices narrower than 768 pixels.
Responsive utilities for mobile phones
Default
Visible
Hidden
Utility classes for showing and hiding content by devices narrower than 768 pixels.
Column width for tablets
Inherit from above
Override column units for devices narrower than 992 pixels.
Offset for tablets
No offset
Number of offset units for devices narrower than 992 pixels.
Responsive utilities for tablets
Default
Visible
Hidden
Utility classes for showing and hiding content by devices narrower than 992 pixels.
Column width for laptops
Inherit from above
Override column units for devices narrower than 1200 pixels.
Offset for laptops
No offset
Number of offset units for devices narrower than 1200 pixels.
Responsive utilities for laptops
Default
Visible
Hidden
Utility classes for showing and hiding content by devices narrower than 1200 pixels.
Column width for large desktops
Inherit from above
Override column units for devices wider than 1200 pixels.
Offset for large desktops
No offset
Number of offset units for devices wider than 1200 pixels.
Default
Visible
Hidden
Utility classes for showing and hiding content by devices wider than 1200 pixels.

Note: If the current column is member of a container which disables some of its breakpoints (*large*, *medium*, *small* or *tiny*), then that column editor shows up only with the input fields for the enabled breakpoints.

Complete DOM Structure

After having added a container with different rows and columns, you may add the leaf plugins. These hold the actual content, such as text and images.



By pressing the button **Publish changes**, the single blocks are regrouped and displayed using the Bootstrap's grid system.

Adding Plugins into a hard coded grid

Sometimes the given Django template already defines a Bootstrap Container, or Row inside a Container element. Example:

```
<div class="container">
    {% placeholder "Row Content" %}
</div>
```

or

```
<div class="container">
    <div class="row">
        {% placeholder "Column Content" %}
    </div>
</div>
```

Here the Django templatetag `{% placeholder "Row Content" %}` requires a Row- rather than a Container-plugin; and the templatetag `{% placeholder "Column Content" %}` requires a Column-plugin. Hence we must tell **djangocms-cascade** which breakpoints shall be allowed and what the containers extensions shall be. This must be hard-coded inside your `setting.py`:

```
CMS_PLACEHOLDER_CONF = {
    # for a row-like placeholder configuration ...
```

```
    'Row Content': {
        'plugins': ['BootstrapRowPlugin'],
        'parent_classes': {'BootstrapRowPlugin': []},
        'require_parent': False,
        'glossary': {
            'breakpoints': ['xs', 'sm', 'md', 'lg'],
            'container_max_widths': {'xs': 750, 'sm': 750, 'md': 970, 'lg': 1170},
            'fluid': False,
            'media_queries': {
                'xs': ['(max-width: 768px)'],
                'sm': ['(min-width: 768px)', '(max-width: 992px)'],
                'md': ['(min-width: 992px)', '(max-width: 1200px)'],
                'lg': ['(min-width: 1200px)'],
            },
        }
    },
    # or, for a column-like placeholder configuration ...
    'Columnn Content': {
        'plugins': ['BootstrapColumnPlugin'],
        'parent_classes': {'BootstrapColumnPlugin': []},
        'require_parent': False,
        'glossary': {
            'breakpoints': ['xs', 'sm', 'md', 'lg'],
            'container_max_widths': {'xs': 750, 'sm': 750, 'md': 970, 'lg': 1170},
            'fluid': False,
            'media_queries': {
                'xs': ['(max-width: 768px)'],
                'sm': ['(min-width: 768px)', '(max-width: 992px)'],
                'md': ['(min-width: 992px)', '(max-width: 1200px)'],
                'lg': ['(min-width: 1200px)'],
            },
        }
    },
}
```

Please refer to the DjangoCMS documentation for details about these settings with the exception of the dictionary `glossary`. This latter setting is special to **djangocms-cascade**: It gives the placeholder the ability to behave like a plugin for the Cascade app. Remember, each **djangocms-cascade** plugin stores all of its settings inside a Python dictionary which is serialized into a single database field. By having a placeholder behaving like a plugin, here this so named *glossary* is emulated using an additional entry inside the setting `CMS_PLACEHOLDER_CONF`, and it should:

- include all the settings a child plugin would expect from a real container plugin
- reflect how hard coded container was defined (e.g. whether it is fluid or not)

Nested Columns and Rows

One of the great features of Bootstrap is the ability to nest Rows inside Columns. These nested Rows then can contain Columns of 2nd level order. A quick example:

```
<div class="container">
  <div class="row">
    <div class="col-md-3">
      Left column
    </div>
    <div class="col-md-9">
      <div class="row">
```

```
        <div class="col-md-6">
          Left nested column
        </div>
        <div class="col-md-6">
          Right nested column
        </div>
      </div>
    </div>
  </div>
</div>
```

rendered, it would look like:



If a responsive image shall be placed inside a column, we must estimate the width of this image, so that when rendered, it fits exactly into that column. We want easy-thumbnails to resize our images to the columns width and not having the browser to up- or down-scale them.

Therefore **djangocms-cascade** keeps track of all the breakpoints and the chosen column widths. For simplicity, this example only uses the breakpoint "medium". The default Boostrap settings for this width is 992 pixels. Doing simple math, the outer left column widths gives 3 / 12 * 992 = 248 pixels. Hence, adding a responsive image to that column means, that **easy-thumnails** automatically resizes it to a width of 248 pixels.

To calculate the width of the nested columns, first evaluate the width of the outer right column, which is 9 / 12 * 992 = 744 pixels. Then this width is subdivided again, using the width of the nested columns, which is 6 / 12 * 744 = 372 pixels.

These calculations are always performed recursively for all nested column and for all available breakpoints.

Warning: As the name implies, a container marked as *fluid*, does not specify a fixed width. Hence instead of the inner width, the container's outer width is used as its maximum. For the large media query (with a browser width of 1200 pixels or more), the maximum width is limited to 1980 pixels.

Other Bootstrap3 specific Plugins

HTML5 <picture> and the new <img> elements

Bootstrap's responsive grid system, helps developers to adapt their site layout to a wide range of devices, from smartphones to large displays. This works fine as long as the content can adopt to the different widths. Adding the CSS class `img-responsive` to an `<img ... />` tag, resizes that image to fit into the surrounding column. However, since images are delivered by the server in one specific size, they either are too small and must be upscaled, resulting in an grainy image, or are too big, resulting in a waste of bandwidth and slowing down the user experience, when surfing over slow networks.

Adaptive resizing the images

An obvious idea would be to let the server decide, which image resolution fits best to the browsing device. This however is bad practice. Images typically are served upon a GET-request pointing onto a specific URL. GET-requests shall be idempotent and thus are predestined to be cached by proxies on the way to the client. Therefore it is a very bad idea to let the client transmit its screen width via a cookie, and deliver different images depending on this value.

Since the sever side approach doesn't work, it is the browsers responsibility to select the appropriate image size. An ideal adaptive image strategy should do the following:

- Images should fit the screen, regardless of their size. An adaptive strategy needs to resize the image, so that it can resize into the current column width.
- Downloading images shall minimize the required bandwidth. Large images are enjoying greater popularity with the advent of Retina displays, but those devices normally are connected to the Internet using DSL rather than mobiles, which run on 3G.
- Not all images look good when squeezed onto a small display, particularly images with a lot of detail. When displaying an image on a mobile device, you might want to crop only the interesting part of it.

As these criteria can't be fulfilled using the well known `<img src="..." />` element, **djangocms-cascade** offers two responsive variants recently added to the HTML5 standard:

One is the `<img>` tag, but with the additional attributes `sizes` and `srcset`. This element can be used as a direct replacement for `<img src="...">`.

The other is a new element named `<picture>`. Use this element, if the image's shape or details shall adopt their shape and/or details to the displaying media device. The correct terminology for this kind of behavior is art direction.



But in the majority of use cases, the **Bootstrap Image Plugin** will work for you. Use the **Bootstrap Picture Plugin** only in those few cases, where in addition to the image width, you also want to change the aspect ratio and/or zoom factor, depending on the display's sizes.

Using these new elements, the browser always fetches the image which best fits the current layout. Additionally, if the browser runs on a high resolution (Retina) display, an image with double resolution is downloaded. This results in much sharper images.

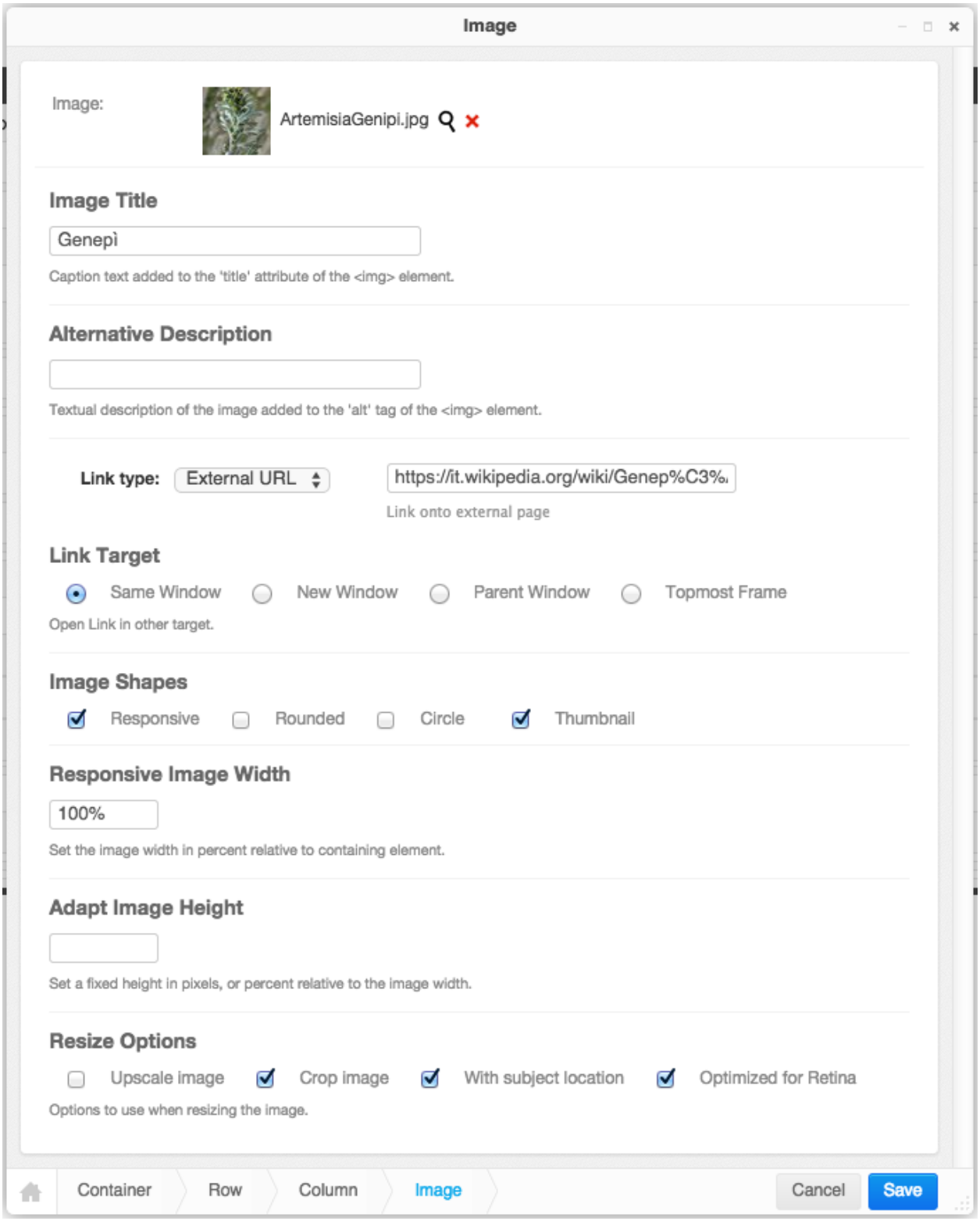
Browser support

Since Chrome 38, the `<img ... />` element fully supports srcset and sizes. It also supports the `<picture>` element right out of the box. Here is a list of native browser support for the picture and the image element with attribute srcset.

For legacy browsers, there is a JavaScript library named picturefill.js, which emulates the built in behavior of these new features. But even without that library, **djangocms-cascade** renders these HTML elements in a way to fall back on a sensible default image.

Image Plugin Reference

In edit mode, double clicking on an image, opens the **Image Plugin** editor. This editor offers the following fields in order to adapt an image to the current layout.


Image
Image:
ArtemisiaGenipi.jpg
Image Title
Genepì
Caption text added to the 'title' attribute of the <img> element.
Alternative Description
Textual description of the image added to the 'alt' tag of the <img> element.
Link type:
External URL
https://it.wikipedia.org/wiki/Genep%C3%
Link onto external page
Link Target
Same Window
New Window
Parent Window
Topmost Frame
Open Link in other target.
Image Shapes
Responsive
Rounded
Circle
Thumbnail
Responsive Image Width
100%
Set the image width in percent relative to containing element.
Adapt Image Height
Set a fixed height in pixels, or percent relative to the image width.
Resize Options
Upscale image
Crop image
With subject location
Optimized for Retina
Options to use when resizing the image.
Container
Row
Column
Image
Cancel
Save

Image

Clicking on the magnifying glass opens a pop-up window from django-filer where you can choose the appropriate image.

Image Title

This optional field shall be used to set the `<img title="some text" .../>` tag inside this HTML element.

Alternative Description

This field shall be used to set the `alt` tag inside the `<picture>` or `<img>` element. While the editor does require this field to be filled, it is strongly recommended to add some basic information about that picture.

Link type

Using this select box, one can choose to add an internal, or external link to the image. Please check the appropriate section for details.

Image Shapes

These checkboxes control the four CSS classes from the Bootstrap3 framework: `img-responsive`, `img-rounded`, `img-circle` and `img-thumbnail`. While rendering HTML, they will be added to the `<img ... />` element.

Here the option *Responsive* has a special meaning. The problem with responsive images is, that their size depends on the media width of the device displaying the image. Therefore we can not use the well known `<img ... />` element with a fixed `width=".."` and `height=".."`. Instead, when rendering responsive images, the additional attributes `srcset` and `sizes` are added to the element. The attribute `srcset` contains the URLs, of up to four differently scaled images. The width of these images is determined by the maximum width of the wrapping container `<div>`, normally a Bootstrap column.

Responsive Image Width

This field is only available for *responsive* images. If set to 100% (the default), the image will spawn the whole column width. By setting this to a smaller value, one may group more than one image side by side into one column.

Fixed Image Width

This field is only available for *non-responsive* images. Here an image size must be specified in pixels. The image then will be rendered with a fixed width, independently of the current screen width. Images rendered with a fixed width do not neither contain the attributes `srcset` nor `sizes`.

Adapt Image Height

Leaving this empty (the default), keeps the natural aspect ratio of an image. By setting this to a percentage value, the image's height is resized to its current used width, hence setting this to `100%` reshapes the image into a square. Note

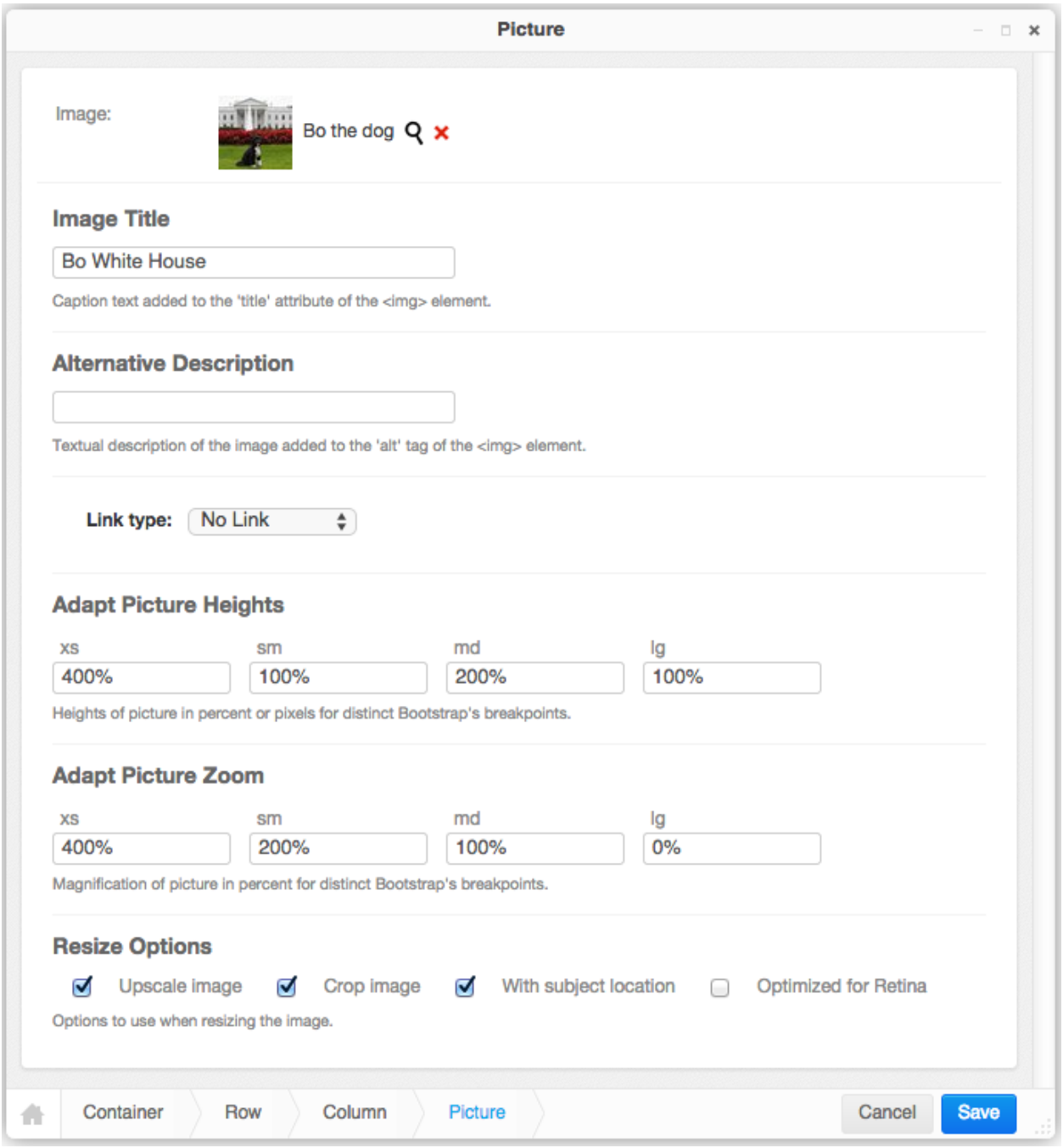
that this normally requires to *crop* the image, see *Resize Options* below. Setting this value in pixels, set the image to a fixed height.

Resize Options

- **Upscale image**: If the original image is smaller than the desired drawing area, then the image is upscaled. This in general leads to blurry images and should be avoided.
- **Crop image**: If the aspect ratio of the image and the desired drawing area do not correlate, than the image is cropped to fit, rather than leaving white space arround it.
- **With subject location**: When cropping, use the red circle to locate the most important part of the image. This is a feature of Django's Filer.
- **Optimized for Retina**: Currently only available for images marked as *responsive*, this option adds an images variant suitable for Retina displays.

Picture Plugin Reference

A picture is another wording for image. It offers some rarely required options when working with images using art direction. By double-clicking onto a picture, its editor pops up.



The field **Image**, **Image Title**, **Alternative Description**, **Link type** and **Resize Options** behave exactly the same as for the **Image Plugin**.

Beware that *Pictures* always are considered as responsive, and they always spawn to the whole width of the wrapping element, using the CSS style `width: 100%`. They make the most sense for large images extending over a large area. Therefore it is not possible to specify a width for a picture.

Adapt Picture Heights

Depending on the current screen's width, one may set different heights for an image. This is useful in order to adopt the aspect ratio of an image, when switching from desktops to mobile devices. Normally, one should use a fixed height

in pixels here, but when specifying the heights in percent, these heights are considered relative to the current image height.

Adapt Picture Zoom

Depending on the current screen's width, one may set different zoom levels for an image. This is useful for keeping the level of detail constant, at the cost of cropping more of the image's margins.

Template tag for the Bootstrap3 Navbar

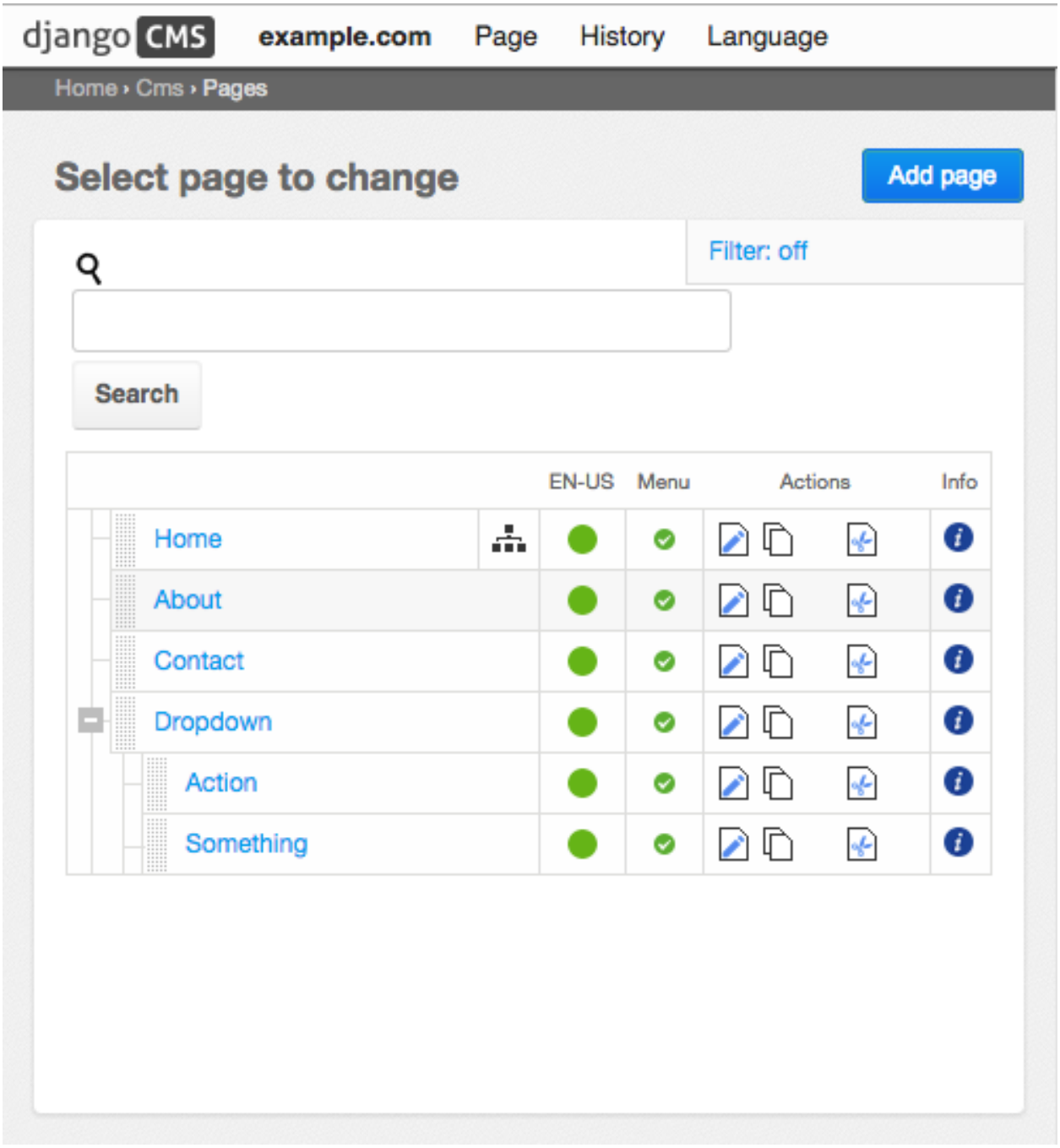
Warning: This template tag is now deprecated. It's functionality has been split off into a new project that can be found here: Django CMS Bootstrap 3.

Although it's not derived from the `CascadeElement` class, this Django app is shipped with a template tag to render the main menu inside a Bootstrap Navbar. This tag is named `main_menu` and shall be used instead of `show_menu`, as shipped with the DjangoCMS menu app.

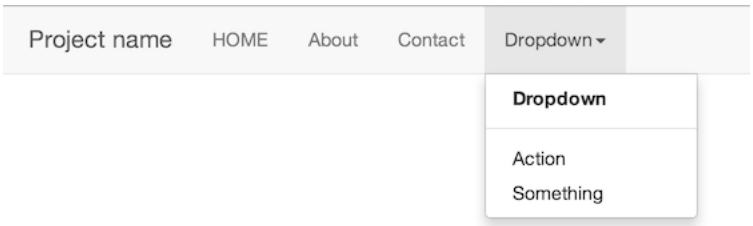
Render a Navbar according to the Bootstrap3 guide:

```
{% load bootstrap3_tags %}
...
<div class="navbar navbar-default navbar-fixed-top" role="navigation">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=
↪".navbar-collapse">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="/">Project name</a>
    </div>
    <div class="collapse navbar-collapse">
      <ul class="nav navbar-nav">{% main_menu %}</ul>
    </div>
  </div>
</div>
```

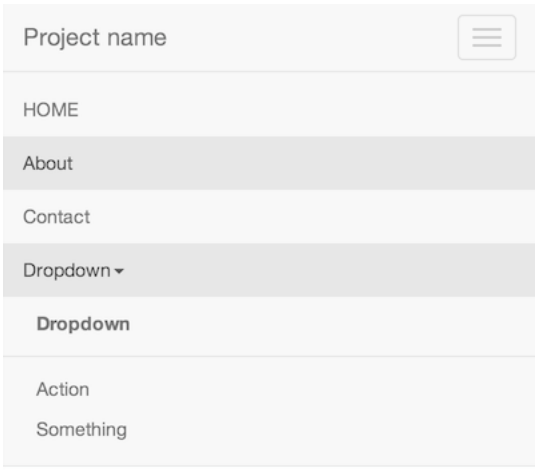
Assume, the page hierarchy in DjangoCMS is set up like this:



then in the front-end, the navigation bar will be rendered as



on computer displays, and as



on mobile devices.

Note: Bootstrap3 does not support "hover", since this event can't be handled by touch screens. Therefore the client has to click on the menu item, rather than moving the mouse cursor over it. In order to make CMS pages with children selectable, those menu items are duplicated. For instance, clicking on **Dropdown** in the Navbar, just opens the pull-down menu. Here the menu item for the page named "Dropdown" is rendered again. Clicking on this item, finally loads that page from the CMS.

Note: Bootstrap3 does not support nested menus, because they wouldn't be usable on mobile devices. Therefore the template tag `main_menu` renders only one level of children, no matter how deep the page hierarchy is in DjangoCMS.

Panel element

Bootstrap is shipped with CSS helpers to facilitate the creation of Panels. In **djangocms-cascade** these panels can be added to any placholder. In the context menu of a placeholder, select **Panel** below the section **Bootstrap** and chose the style. The panel heading and footer are optional. As body, the panel element accepts other plugins, normally this is a Text plugin.

Jumbotron

Bootstrap is shipped with CSS helpers to facilitate the creation of a Jumbotron, sometimes also named "Hero" element. In **djangocms-cascade**, such a Jumbotron plugin can be added anywhere, even as the root element of a placeholder, in other words, even outside of a Bootstrap Container plugin. The latter configuration is specially useful for images, which shall extend over the full width of the web page.

If used outside a Bootstrap Container, we first must configure the allowed breakpoints. This is the same behaviour as for the Container plugin. Then we optionally can chose an image or a background color, it's size, attachment, position and repetitions. For more details read this article on how to configure background images using pure CSS.

A Jumbotron without any content has a default height of 96 pixels, which is 48 pixels for the top- and bottom paddings, each. These values are given by the Bootstrap 3 framework.

To increase the height of a Jumbotron you have two choices. The simpler one is to add some content to the Jumbotron plugin which then increases it's height. Another, is to explicitly to set other padding of the Jumbotron plugin.

Tab Sets

Bootstrap is shipped with CSS helpers to facilitate the creation of Tabs. In **djangocms-cascade**, such a Tab plugin can be added anywhere inside columns or rows.

In the context menu of a placeholder, select **Tab Set**. Depending on the chosen number of children, it will add as many **Tab Pane**s. Each **Tab Pane** has a Title field, its content is displayed in the tab. Below a **Tab Pane** you are free to add whatever you want.

Secondary menu

> **Warning:** This plugin is experimental. It may disappear or be replaced. Use it at your own risk!

Often there is a need to add secondary menus at arbitrary locations. The **Secondary menu** plugin can be used in any placeholder to display links onto child pages of a CMS page. Currently only pages marked as **Soft Root** with a defined **Page Id** are allowed as parent of such a secondary menu.

Note: This plugins reqires the template tag `main_menu_below_id` which is shipped with djangocms-bootstrap3

Gallery

A gallery is a collection of images displayed as a group. Since it normally consists of many similar images, **djangocms-cascade** does not require to use child plugins for each image. Instead they can be added directly to the **Bootstrap Gallery Plugin**. Here, **djangocms-cascade** uses a special model, named `cmsplugin_cascade.models.InlineCascadeElement` which also uses a JSON field to store it's payload. It thus can be configured to accept any kind of data, just as it's counterpart `cmsplugin_cascade.models.CascadeElement` does.

Since plugin editors are based on Django's admin backend, the Gallery Plugin uses the Stacked Inline formset to manage it's children. If **django-admin-sortable2** is installed, the entries in the plugin can even be sorted using drag and drop.

Using Fonts with Icons

Introduction

Sometime we want to enrich our web pages with vectorized symbols. A lot of them can be found in various font libraries, such as Font Awesome, Material Icons and many more. A typical approach would be to upload the chosen SVG symbol, and use it as image. This process however is time consuming and error-prone to organize. Therefore, **djangocms-cascade** offers an optional submodule, so that we can work with externally packed icon fonts.

In order to use such a font, currently we must use Fontello, an external service for icon font generation. In the future, this service might be integrated into **djangocms-cascade** itself.

Configuration

To enable this service in **djangocms-cascade**, in `settings.py` add:

```
INSTALLED_APPS = [
    ...
    'cmsplugin_cascade',
    'cmsplugin_cascade.icon',
    ...
]

CMSPLUGIN_CASCADE_PLUGINS = [
    ...
    'cmsplugin_cascade.icon',
    ...
]
```

This submodule, can of course be combined with all other submodules available for the Cascade ecosystem.

If `CMS_PLACEHOLDER_CONF` is used to configure available plugins for each placeholder, assure that the `TextIconPlugin` is added to the list of `text_only_plugins`.

Since the CKEditor widget must load the font stylesheets for it's own WYSIWIG mode, we have to add this special setting to our configuration:

```
from django.core.urlresolvers import reverse_lazy
from cmsplugin_cascade.utils import format_lazy

CKEDITOR_SETTINGS = {
    ...
    'stylesSet': format_lazy(reverse_lazy('admin:cascade_texticon_wysiwig_config')),
}
```

Uploading the Font

In order to start with an external font icon, choose one or more icons and/or whole font families from the Fontello website and download the generated webfont file to a local folder.

In Django's admin backend, change into `Start › django CMS Cascade › Uploaded Icon Fonts` and add an Icon Font object. Choose an appropriate name and upload the just downloaded webfont file, without unzipping it. After the upload completed, all the imported icons appear grouped by their font family name. They now are ready for being used by the Icon plugin.

Using the Icon Plugin

A font symbol can be used everywhere plain text can be added. Inside a **django-CMS** placeholder field add a plugin of type **Icon**. Select a family from one of the uploaded fonts. Now a list of possible symbols appears. Choose the desired symbol, its size and color. Optionally choose a background color, the relative position in respect of its wrapping element and a border width with style and color. After saving the form, that element should appear inside the chosen container.

It is good practice to only use one uploaded icon font per site. If you forgot a symbol, go back to the Fontello site and recreate your icon font. Then replace that icon font by uploading it again.

> **Warning:** If you use more than one font on the same page, please assure that Fontello assigns unique glyph codes to all of the symbols – this usually is not the case. Otherwise, the glyph codes will collapse, and the visual result is not what you expect.

Shared Settings

By default, the **IconPlugin** is configured to allow to share the following styling attributes:

- Icon size
- Icon color
- Background color, or without background
- Text alignment
- Border width, color and style
- Border radius

By storing these attributes under a common name, one can reuse them across various icons, without having to set them for each one, separately. Additionally, each of the shared styling attributes can be changed globally in Django's admin backend at `Start › django CMS Cascade › Shared between Plugins`. For details please refer to the section about *Working with sharable fields*.

Using the Icon Plugin in plain text

If **django-CMS** is configured to use the **djangocms-ckeditor-widget**, then you may use the **Icon Plugin** inside plain text. Place the cursor at the desired location in text and select **Icon** from the pull down menu **CMS Plugins**. This opens a popup where you may select the font family and the symbol. All other attributes described above, are not available with this type of plugin.

Handling the client side

DjangoCMS-Cascade is shipped with a lot of plugins, all having their own inheritance hierarchy. Due to the flexibility of Cascade, this inheritance hierarchy can be extended though some configuration settings, while bootstrapping the runtime environment. Some plugins for instance, can be configured to store some settings in a common data store. This in the admin backend requires a special Javascript plugin, from which the client side must inherit as well.

Hence on the client side, we would like to describe the same inheritance hierarchy using Javascript. Therefore Cascade is equipped with a small, but very powerful library named ring.js. It makes Javascript behave almost like Python. If a Cascade plugin provides a Javascript counterpart, then other Cascade plugins inheriting from the former one, map their inheritance hierarchy in Javascript exactly as provided by the plugins written in Python.

Implementing the client

Say, we want to add some client side code to a Cascade plugin. We first must import that Javascript file through Django's static asset definitions using the `Media` class, or if you prefer in a dynamic property method `media()`.

At some point during the initialization, Cascade must call the constructor of the Javascript plugin we just added. Therefore Cascade plugins provide an extra attribute named `ring_plugin`, which is required to name the Javascript's counterpart of our Python class. You can use any name you want, but it is good practice to use the same name as the plugin.

The Python class of our custom Cascade plugin then might look like:

```
from cmsplugin_cascade.plugin_base import CascadePluginBase

class MyCustomPlugin(CascadePluginBase):
    name = "Custom Plugin"
```

```
    ... other class attributes
    ring_plugin = 'MyCustomPlugin'

    class Media:
        js = ['mycustomproject/js/admin/mycustomplugin.js']
```

whereas it's Javascript counterpart might look like:

Listing 3.1: mycustomproject/js/admin/mycustomplugin.js

```
django.jQuery(function($) {
    'use strict';

    django.cascade.MyCustomPlugin = ring.create({
        constructor: function() {
            // initialization code
        },
        custom_func: function() {
            // custom functionality
        }
    });
});
```

After yours, and all other Cascade plugins have been initialized in the browser, the Cascade framework invokes `new django.cascade.MyCustomPlugin();` to call the constructor function.

Plugin Inheritance

If for instance, our `MyCustomPlugin` requires functionality to set a link, then instead of replication the code required to handle the link input fields, we can rewrite our plugin as:

```
from cmsplugin_cascade.link.config import LinkPluginBase

class MyCustomPlugin(LinkPluginBase):
    ... class attributes as in the previous example
```

Since `LinkPluginBase` provides it's own `ring_plugin` attribute, the corresponding Javascript code *also must inherit* from that base class. Cascade handles this for you automatically, if the Javascript code of the plugin is structured as:

Listing 3.2: mycustomproject/js/admin/mycustomplugin.js

```
django.jQuery(function($) {
    'use strict';

    var plugin_bases = eval(django.cascade.ring_plugin_bases.MyCustomPlugin);

    django.cascade.MyCustomPlugin = ring.create(plugin_bases, {
        constructor: function() {
            this.$super();
            // initialization code
        },
        ...
    });
});
```

The important parts here is the call to `eval(django.cascade.ring_plugin_bases.MyCustomPlugin)`, which resolves the Javascript functions our custom plugin inherits from.

Note: In case you forgot to add a missing Javascript requirement, then ring.js complains with the error message `Uncaught TypeError: Cannot read property '__classId__' of undefined`. If you run into this problem, recheck that all Javascript files have been loaded and initialized in the correct order.

Section Bookmarks

If you have a long page, and you want to allow the visitors of your site to quickly navigate to different sections, then you can use bookmarks and create links to the different sections of any HTML page.

When a user clicks on a bookmark link, then that page will load as usual but will scroll down immediately, so that the bookmark is at the very top of the page. Bookmarks are also known as anchors. They can be added to any HTML element using the attribute `id`. For example:

```
<section id="unique-identifier-for-that-page">
```

For obvious reasons, this identifier must be unambiguous, otherwise the browser does not know where to jump to. Therefore **djangocms-cascade** enforces the uniqueness of all bookmarks used on each CMS page.

Configuration

The HTML standard allows the usage of the `id` attribute on any element, but in practice it only makes sense on `<section>`, `<article>` and the heading elements `<h1>`...‘‘<h6>‘‘. Cascade by default is configured to allow bookmarks on the **SimpleWrapperPlugin** and the **HeadingPlugin**. This can be overridden in the project's configuration settings using:

```
CMSPLUGIN_CASCADE = {
    ...
    'plugins_with_bookmark': [list-of-plugins],
    ...
}
```

Hashbang Mode

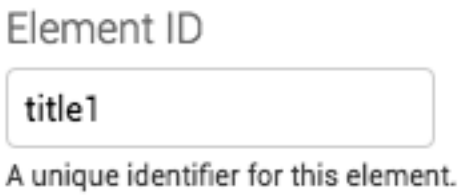
Links onto bookmarks do not work properly in hashbang mode. Depending on the HTML settings, you may have to prefix them with `/` or `!`. Therefore **djangocms-cascade** offers a configuration directive:

```
CMSPLUGIN_CASCADE = {
    ...
    'bookmark_prefix': '/',
    ...
}
```

which automatically prefixes the used bookmark.

Usage

When editing a plugin that is eligible for adding a bookmark, an extra input field is shown:



You may add any identifier to this field, as long as it is unique on that page. Otherwise the plugin's editor will be reject the given inputs, while saving.

Hyperlinking to a Bookmark

When editing a **TextLink**, **BootstrapButton** or the link fields inside the **Image** or **Picture** plugins, the user gets an additional drop-down menu to choose one of the bookmarks for the given page. This additional drop-down is only available if the **Link** is of type *CMS page*.



If no bookmarks have been associated with the chosen CMS page, the drop-down menu displays only *Page root*, which is the default.

Segmentation of the DOM

The **SegmentationPlugin** allows to personalize the DOM structure, depending on the context used to render the corresponding page. Since **djangoCMS** always uses a RequestContext while rendering its pages, we always have access onto the request object. Some use cases are:

- Depending on the user, show a different portion of the DOM, if he is a certain user or not logged in at all.
- Show different parts of the DOM, depending on the browsers estimated geolocation. Useful to render different content depending on the visitors country.
- Show different parts of the DOM, depending on the supplied marketing channel.
- Show different parts of the DOM, depending on the content in the session objects from previous visits of the users.
- Segment visitors into different groups used for A/B-testing.

Configuration

The **SegmentationPlugin** must be activated separately on top of other **djangocms-cascade** plugins. In `settings.py`, add to

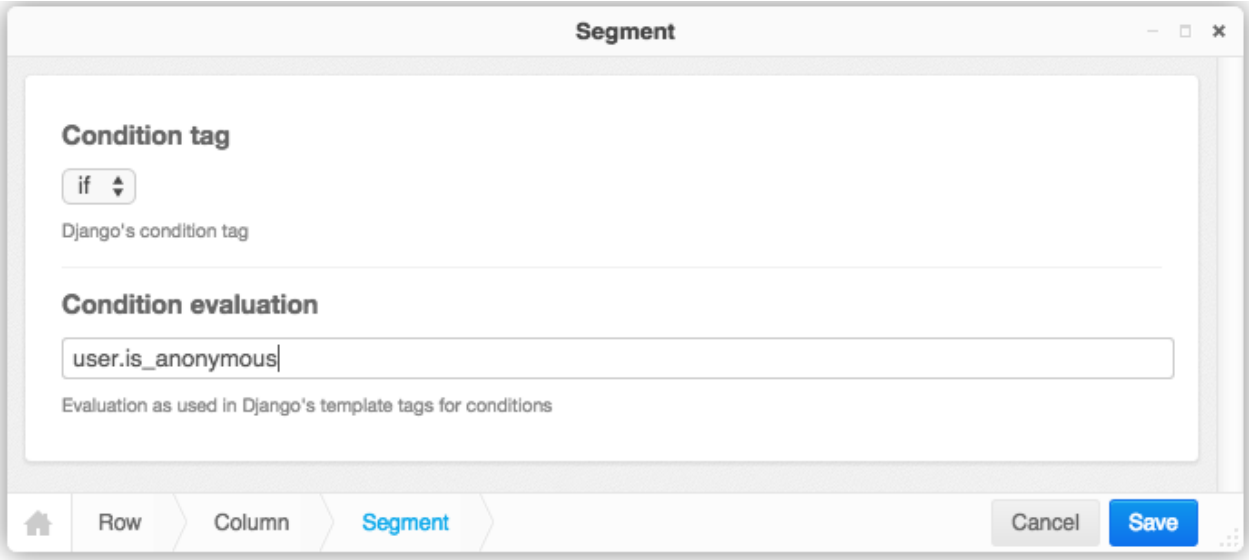
```
INSTALLED_APPS = (
    ...
    'cmsplugin_cascade',
    'cmsplugin_cascade.segmentation',
    ...
)
```

Then, depending on what kind of data shall be emulated, add a list of two-tuples to the configuration settings `CMSPLUGIN_CASCADE['segmentation_mixins']`. The first entry of each two-tuple specifies the mixin class added the the proxy model for the `SegmentationPlugin`. The second entry specifies the mixin class added the model admin class for the `SegmentationPlugin`.

```
# this entry is optional:
CMSPLUGIN_CASCADE = {
    ...
    'segmentation_mixins': (
        ('cmsplugin_cascade.segmentation.mixins.EmulateUserModelMixin', 'cmsplugin_
↪cascade.segmentation.mixins.EmulateUserAdminMixin',),  # the default
        # other segmentation plugin classes
    ),
    ...
}
```

Usage

When editing **djangoCMS** plugins in **Structure** mode, below the section **Generic** a new plugin type appears, named **Segment**.



This plugin now behaves as an `if` block, which is rendered only, if the specified condition evaluates to true. The syntax used to specify the condition, is the same as used in the Django template language. Therefore it is possible to evaluate against more than one condition and combine them with `and`, `or` and `not` as described in boolean operators in the Django docs

Immediately below a segmentation block using the condition tag `if`, it is possible to use the tags `elif` or `else`. This kind of conditional blocks is well known to Python programmers.

Note, that when rendering pages in djangoCMS, a RequestContext- rather than a Context-object is used. This RequestContext is populated by the `user` object if `'django.contrib.auth.context_processors.auth'` is added to your settings.py `TEMPLATE_CONTEXT_PROCESSORS`. This therefore is a prerequisite when the Segmentation plugin evaluates conditions such as `user.username == "john"`.

Emulating Users

As of version 0.5.0, in **djangocms-cascade** a staff user or administrator can emulate the currently logged in user. If this plugin is activated, in the CMS toolbar a new menu tag appears named "Segmentation". Here a staff user can select another user. All evaluation conditions then evaluate against this selected user, instead of the currently logged in user.

It is quite simple to add other overriding emulations. Have a look at the class `cmsplugin_cascade.segmentation.mixins.EmulateUserMixin`. This class then has to be added to your configuration settings `CMSPLUGIN_CASCADE_SEGMENTATION_MIXINS`. It then overrides the evaluation conditions and the toolbar menu.

Working with sharable fields

Sometime you'd want to remember sizes, links or any other options for rendering a plugin instance across the project. In order to not have to do this job for each managed entity, you can remember these settings using a name of your choice, controllable in a special section of the administration backend.

Now, whenever someone adds a new instance using this plugin, a select box with these remembered settings appears. He then can choose from one of the remembered settings, which frees him to reenter all the values.

Configure a Cascade Plugins to optionally share some fields

Configuring a plugin to share specific fields with other plugins of the same type is very easy. In the projects `settings.py`, assure that `'cmsplugin_cascade.sharable'` is part of your `INSTALLED_APPS`.

Then add a dictionary of Cascade plugins, with a list of fields which shall be sharable. For example, with this settings, the image plugin can be configured to share its sizes and rendering options among each other.

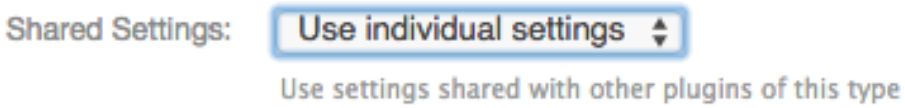
```
CMSPLUGIN_CASCADE = {
    ...
    'plugins_with_sharables': {
        'BootstrapImagePlugin': ('image-shapes', 'image-width-responsive', 'image-
↪width-fixed', 'image-height', 'resize-options',),
    },
    ...
}
```

Control some named settings

Whenever a plugin is configured to allow to share fields, at the bottom of the plugin editor a special field appears:

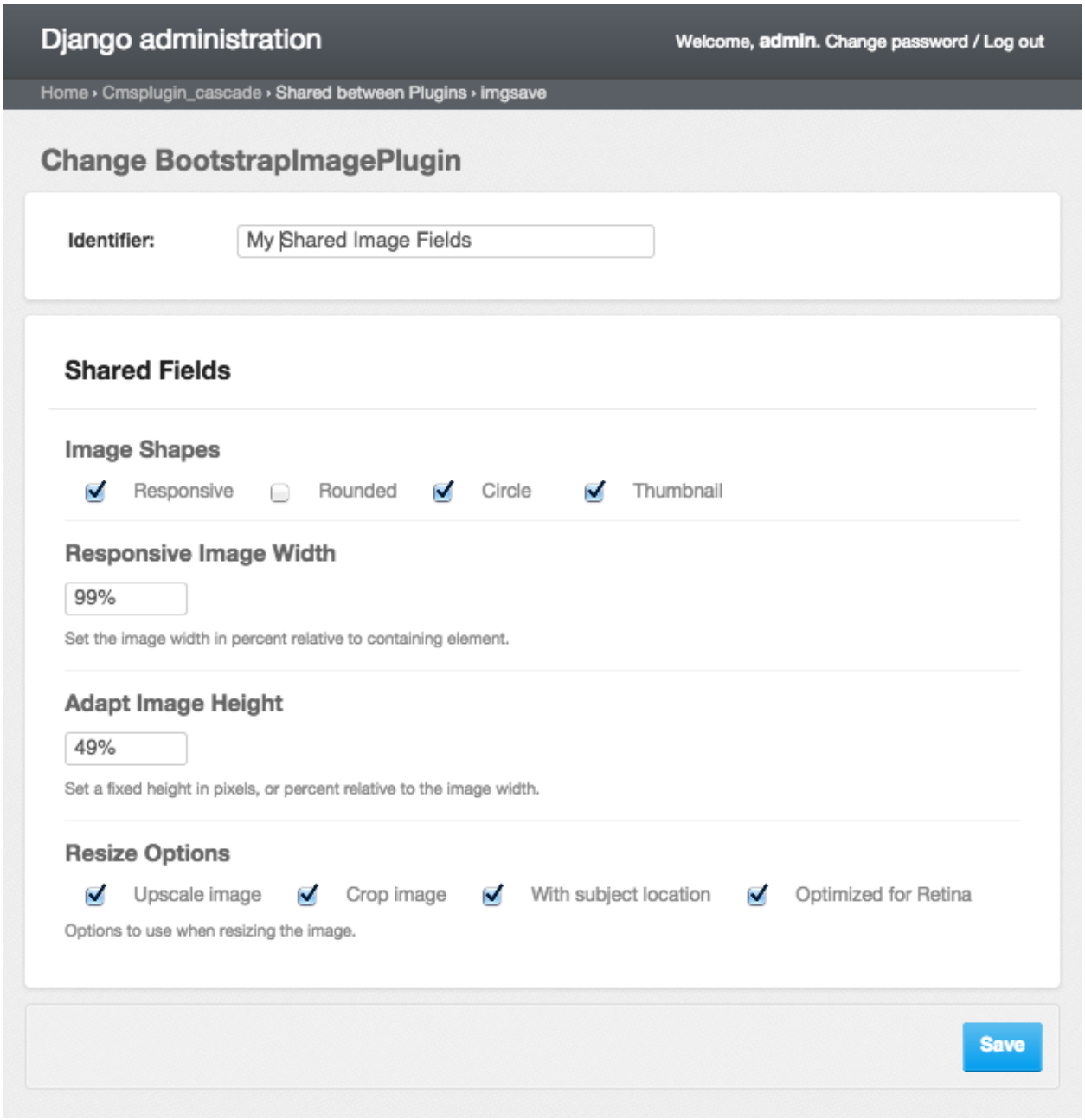


By activating the checkbox, adding an arbitrary name next to it and saving the plugin, an entity of sharable fields is saved in the database. Now, whenever someone starts to edit a plugin of this type, a select box appears on the top of the editor:



By choosing a previously named shared settings, the configured fields are disabled for input and replaced by their shared field's counterparts.

In order to edit these shared fields in the administration backend, one must access **Home › Cmsplugin_cascade › Shared between Plugins**. By choosing a named shared setting, one can enter into the shared field's editor. This editor auto adopts to the fields declared as shared, hence will change from entity to entity. For the above example, it may look like this:



In this editor one can change these shared settings globally, for all plugin instances where this named shared settings have been applied to.

Customize CSS classes and inline styles

Plugins shipped with **djangocms-cascade** offer a basic set of CSS classes as declared by the chosen CSS framework. These offered classes normally do not fulfill the requirements for real world sites.

While **djangocms-cascade** is easily expendable, it would be overkill to re-implement the available plugins, just to add an extra field for a customized CSS class or an extra inline style. For that purpose, one can add a set of potential CSS classes and potential CSS inline styles for Cascade plugins, enabled for this feature. Moreover, this feature can be adopted individually on a per-site base.

Configure a Cascade plugins to accept extra fields

It is possible to configure each plugin to accept an additional ID tag, one ore more CSS classes or some inline styles. By default the plugins: BootstrapButtonPlugin, BootstrapRowPlugin, BootstrapJumbotronPlugin and the SimpleWrapperPlugin are eligible for accepting extra styles. Additionally, by default the user can override the margins of the HeadingPlugin and the HorizontalRulePlugin.

To override these defaults, first assure that `'cmsplugin_cascade.extra_fields'` is part of your `INSTALLED_APPS`. Then add a dictionary of Cascade plugins, which shall be extendible to the project's `settings.py`, for instance:

```
CMSPLUGIN_CASCADE = {
    ...
    'plugins_with_extra_fields': {
        'BootstrapButtonPlugin': PluginExtraFieldsConfig(),
        'BootstrapRowPlugin': PluginExtraFieldsConfig(),
        'BootstrapJumbotronPlugin': PluginExtraFieldsConfig(inline_styles={
            'extra_fields:Paddings': ['padding-top', 'padding-bottom'],
            'extra_units:Paddings': 'px,em'}),
        'SimpleWrapperPlugin': PluginExtraFieldsConfig(),
        'HeadingPlugin': PluginExtraFieldsConfig(inline_styles={
            'extra_fields:Paddings': ['margin-top', 'margin-right', 'margin-bottom',
↪'margin-left'],
            'extra_units:Paddings': 'px,em'}, allow_override=False),
        'HorizontalRulePlugin': PluginExtraFieldsConfig(inline_styles={
            'extra_fields:Paddings': ['margin-top', 'margin-bottom'],
            'extra_units:Paddings': 'px,em'}, allow_override=False),
    },
    ...
}
```

Here the class `PluginExtraFieldsConfig` can be used to fine-tune which extra fields can be set while editing the plugin. Assigning that class without arguments to a plugin, allows us to specify the extra fields using the Django administration backend at:

Home › django CMS Cascade › Custom CSS classes and styles

Here the site administrator can specify for each concrete plugin, which extra CSS classes, ID tags and extra inline styles shall be used.

If we use `PluginExtraFieldsConfig(allow_override=False)`, then we can not override the configuration using the administration backend, but must specify all settings in it's constructor:

class cmsplugin_cascade.extra_fields.config.**PluginExtraFieldsConfig**(*allow_id_tag=False*, *css_classes=None*, *inline_styles=None*, *allow_override=True*)

Each Cascade Plugin can be configured to accept extra fields, such as an ID tag, one or more CSS classes or inlines styles. It is possible to configure these fields globally using an instance of this class, or to configure them on a per site base using the appropriate admin's backend interface at:

Start › django CMS Cascade › Custom CSS classes and styles › PluginExtraFields

Parameters

- **allow_id_tag** – If `True`, allows to set the `id` attribute in HTML elements.
- **css_classes** – A dictionary containing:

`class_names` a comma separated string of allowed class names. `multiple` a Boolean indicating if more multiple classes are allowed concurrently.

Parameters

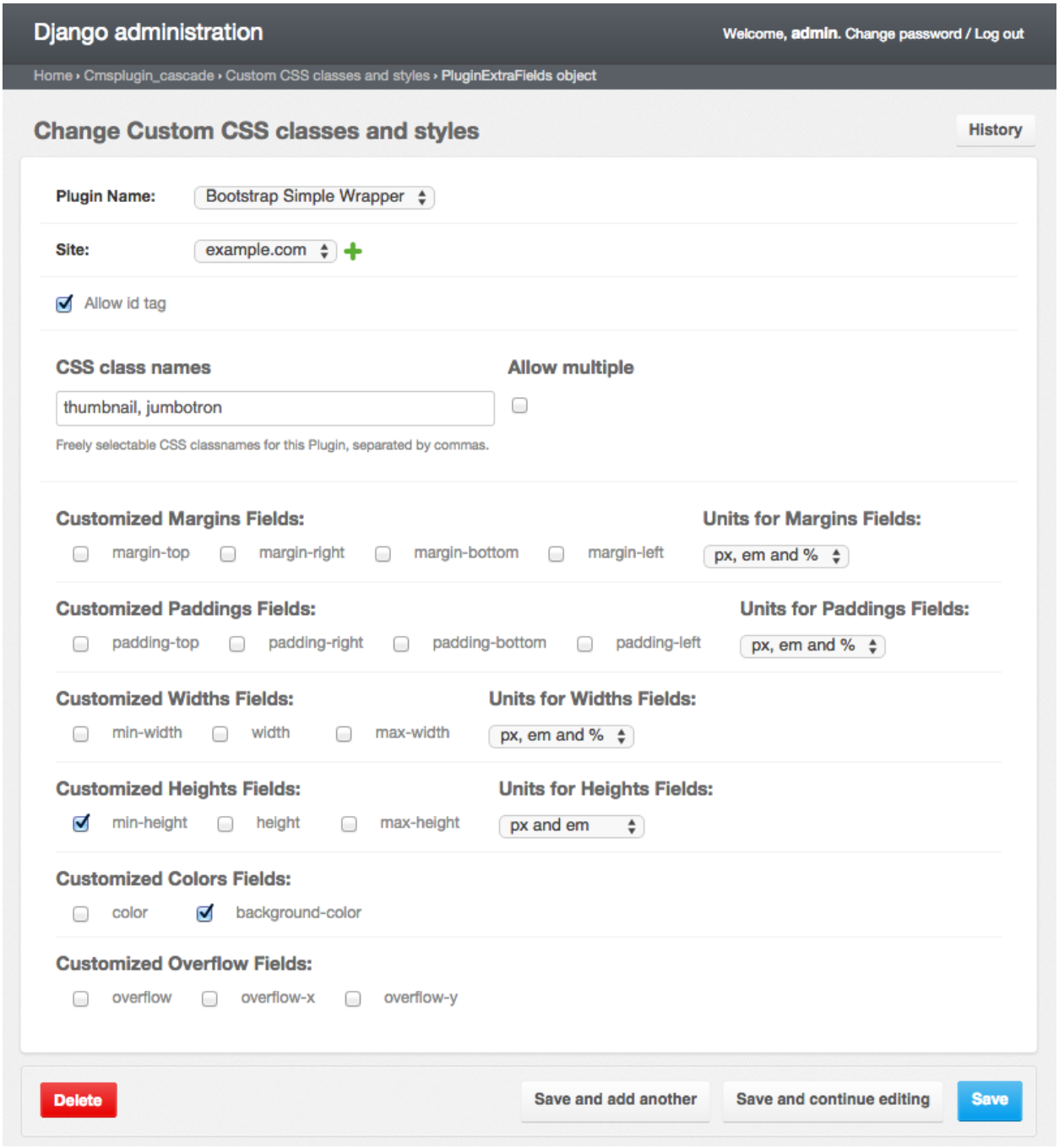
- **`inline_styles`** – A dictionary containing:
- **`allow_override`** – If `True`, allows to override this configuration using the admin's backend interface.

Enable extra fields through the administration backend

To enable this feature, in the administration backend navigate to

Home › django CMS Cascade › Custom CSS classes and styles and click onto the button named **Add Custom CSS classes styles**.

From the field named "Plugin Name", select one of the available plugins, for example **Bootstrap Simple Wrapper**. Then, from the field named "Site", select the current site.



Allow ID

With “Allow id tag” enabled, an extra field will appear on the named plugin editor. There a user can add any arbitrary name which will be rendered as `id="any_name"` for the corresponding plugin instance.

CSS classes

In the field named “CSS class names”, the administrator may specify arbitrary CSS classes separated by commas. One of these CSS classes then can be added to the corresponding Cascade plugin. If more than one CSS class shall be

addable concurrently, activate the checkbox named "Allow multiple".

CSS inline styles

The administrator may activate all kinds of CSS inline styles by clicking on the named checkbox. For settings describing distances, additionally specify the allowed units to be used.

Now, if a user opens the corresponding plugin inside the **Structure View**, he will see an extra select field to choose the CSS class and some input fields to enter say, extra margins, heights or whatever has been activated.

Use it rarely, use it wise

Adding too many styling fields to a plugin can mess up any web project. Therefore be advised to use this feature rarely and wise. If many people have write access to plugins, set extra permissions on this table, in order to not mess things up. For instance, it rarely makes sense to activate `min-width`, `width` and `max-width`.

Choose an alternative rendering template

Sometimes you must render a plugin with a slightly different template, other than the given default. A possible solution is to create a new plugin, inheriting from the given one and overriding the `render_template` attribute with a customized template. This however adds another plugin to the list of registered CMS plugins.

A simpler solution to solve this problem, is to allow a plugin to be rendered with a customized template out of a set of alternatives.

Change the path for template lookups

Some Bootstrap Plugins are shipped with templates, which are optimized to be rendered by Angular-UI rather than the default jQuery. These alternative templates are located in the folder `cascade/bootstrap3/angular-ui`. If your project uses AngularJS instead of jQuery, then configure the lookup path in `settings.py` with

```
CMSPLUGIN_CASCADE = {
    ...
    'bootstrap3': {
        ...
        'template_basedir': 'angular-ui',
    },
}
```

This lookup path is applied only to the Plugin's field `render_template` prepared for it. Such a template contains the placeholder `{}`, which is expanded to the configured `template_basedir`.

For instance, the **CarouselPlugin** defines its `render_template` such as:

```
class CarouselPlugin(BootstrapPluginBase):
    ...
    render_template = 'cascade/bootstrap3/{}/carousel.html'
    ...
```

Configure Cascade Plugins to be rendered using alternative templates

All plugins which offer more than one rendering template, shall be added in the projects `settings.py` to the dictionary `CMSPLUGIN_CASCADE['plugins_with_extra_render_templates']`. Each item in this dictionary consists of a key, naming the plugin, and a value containing a list of two-tuples. The first element of this two-tuple must be the templates filename, while the second element shall contain an arbitrary name to identify that template.

Example:

```
CMSPLUGIN_CASCADE = {
    ...
    'plugins_with_extra_render_templates': {
        'TextLinkPlugin': (
            ('cascade/link/text-link.html', _("default")),
            ('cascade/link/text-link-linebreak.html', _("with linebreak")),
        )
    },
    ...
}
```

Usage

When editing a **djangoCMS** plugins with alternative rendering templates, the plugin editor adds a select box containing choices for alternative rendering templates. Choose one other than the default, and the plugin will be rendered using that template.

Conditionally hide some plugin

Sometimes a placholder contains some plugins, which temporarily should not show up while rendering. If this feature is enabled, then instead of deleting them, it is possible to hide them.

Enable the meachanism

In the projects `settings.py`, add:

```
CMSPLUGIN_CASCADE = {
    ...
    'allow_plugin_hiding': True,
    ...
}
```

By default, this feature is disabled. If enabled, **djangocms-cascade** adds a checkbox to every plugin editor. This checkbox is labeled *Hide plugin*. If checked, the plugin and all of it's children are not rendered in the current tree. To easily distinguish hidden plugins in structure mode, they are rendered using a shaded background.

The CMS Clipboard

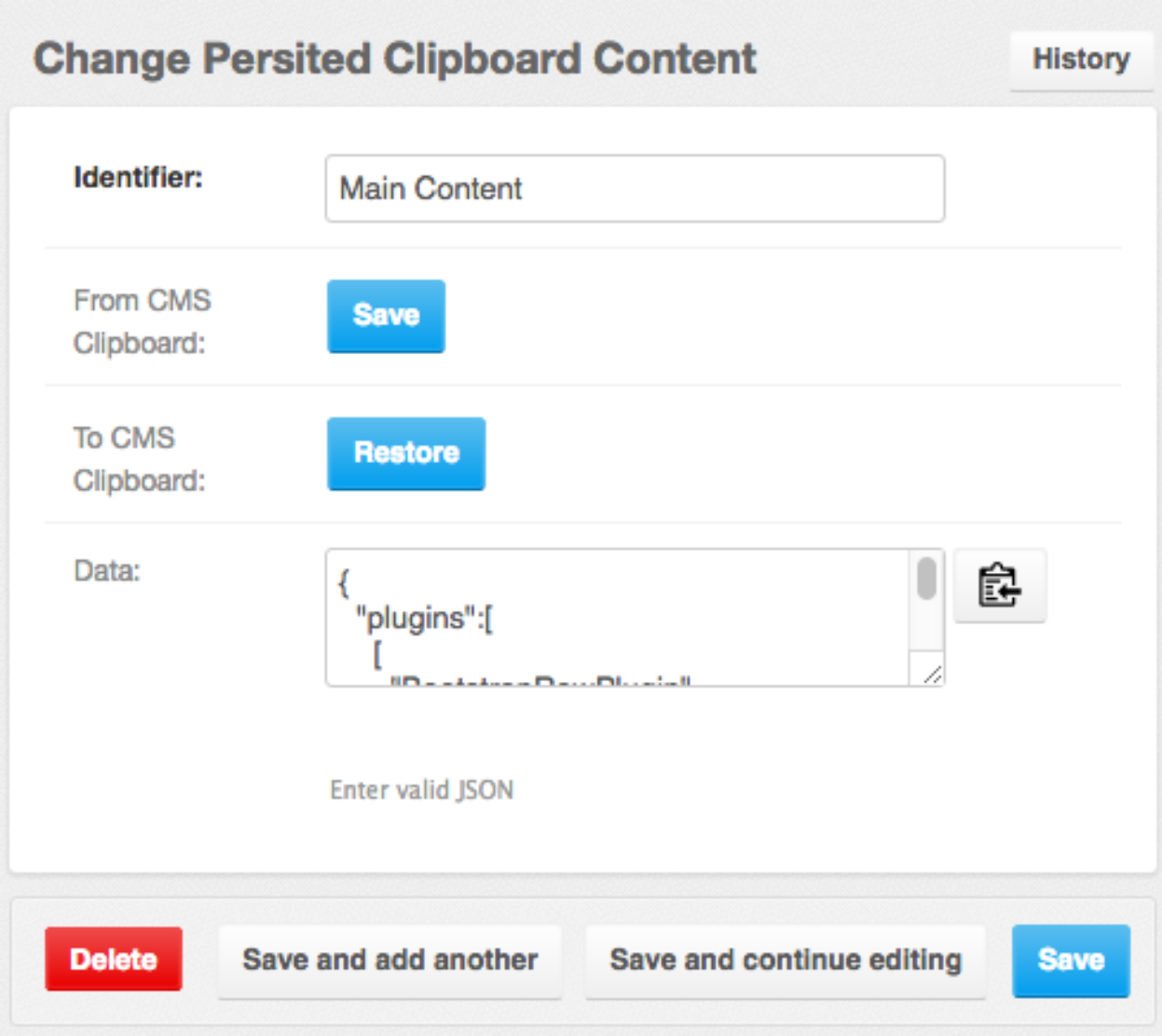
DjangoCMS offers a Clipboard where one can copy or cut and add a subtree of plugins to the DOM. This Clipboard is very handy when copying plugins from one placeholder to another one, or to another CMS page. In version 0.7.2 **djangocms-cascade** extended the functionality of this clipboard, so that the content of the CMS clipboard can be

persited to – and restored from the database. This allows the site-administrator to prepare a toolset of plugin-trees, which can be inserted anywhere at any time.

Persisting the Clipboard

In the context menu of a CMS plugin, use **Cut** or **Copy** to move a plugin together with its children to the CMS clipboard. In **Edit Mode** this clipboard is available from the primary menu item within the CMS toolbar. From this clipboard, the copy plugins can be dragged and dropped to any CMS placeholder which is allowed to accept the root node.

Since the content of the clipboard is overridden by every operation which cuts or copies a tree of plugins, **djangocms-cascade** offers some functionality to persist the clipboard's content. To do this, locate **Persited Clipboard Content** in Django's administration backend.



The **Identifier** field is used to give a unique name to the persited clipboard entity.

The **Save** button fetches the content from the CMS clipboard and persists it.

The **Restore** button replaces the content of the CMS clipboard with the current persisted entity. This is the opposite

operation of **Save**.

Since the clipboard content is serialized using JSON, the site administrator can grab and paste it into another site using **djangocms-cascade**, if persisting clipboards are enabled.

Configuration

Persisting the clipboards content must be configured in the projects `settings.py`:

```
INSTALLED_APPS = (
    ...
    'cmsplugin_cascade',
    'cmsplugin_cascade.clipboard',
    ...
)
```

Caveats

Only CMS plugins from the Cascade eco-system are eligible to be used for persisting. This is because they already use a JSON representation of their content. The only exception is the **TextPlugin**, since **djangocms-cascade** added some serialization code.

Extending Cascade

All Cascade plugins are derived from the same base class `CascadeModelBase`, which stores all its model fields inside a dictionary, serialized as JSON string in the database. This makes it much easier to extend the Cascade eco-system, since no database migration[1] is required when adding a new, or extending plugins from this project.

The database model `CascadeModelBase` stores all the plugin settings in a single JSON field named `glossary`. This in practice behaves like a Django context, but in order to avoid confusion with the latter, it has been named "glossary".

Note: Custom Cascade plugins should set the `app_label` attribute (see below). This is important so migrations for the proxy models generated by Cascade are created in the correct app.

If this attribute is not set, Cascade will default to the left-most part of the plugin's module path. So if your plugin lives in `myapp.cascadeplugins`, Cascade will use `myapp` as the app label. We recommend that you always set `app_label` explicitly.

Simple Example

This plugin is very simple and just renders static content which has been declared in the template.

```
from cms.plugin_pool import plugin_pool
from cmsplugin_cascade.plugin_base import CascadePluginBase

class StylishPlugin(CascadePluginBase):
    name = 'Stylish Element'
```

[1] After having created a customized plugin, it must be registered in Django's permission system, otherwise only administrators, but no staff users, are allowed to add, change or delete them.

```
    render_template = 'myapp/cascade/stylish-element.html'

plugin_pool.register_plugin(StylishPlugin)
```

If the editor form pops up for this plugin, a dumb message appears: "There are no further settings for this plugin". This is because no editable fields have been added to that plugin yet.

Customize Stored Data

In order to make the plugin remember its settings and other optional data, the programmer must add a list of special form fields to its plugin. These fields then are used to auto-generate the editor for this DjangoCMS plugin.

Each of those form fields handle a special field value, or in some cases, a list of field values. They all require a widget, which is used when rendering the editors form.

Lets add a simple selector to choose between a red and a green color. Do this by adding a `GlossaryField` to the plugin class.

```
from django.forms import widgets
from cmsplugin_cascade.plugin_base import CascadePluginBase, PartialFormField

class StylishPlugin(CascadePluginBase):
    ...
    color = GlossaryField(
        widgets.Select(choices=(('red', 'Red'), ('green', 'Green'),)),
        label="Element's Color",
        initial='red',
        help_text="Specify the color of the DOM element."
    )
```

In the plugin's editor, the form now pops up with a single select box, where the user can choose between a *red* and a *green* element.

A `GlossaryField` accepts five arguments:

- The widget. This can be a built-in Django widget or any valid widget derived from it.
- The `label` used to describe the field. If omitted, the `name` of the form field is used.
- The `name` of the field.
- An optional `initial` value to be used with Radio- or Select fields.
- An optional `help_text` to describe the field's purpose.

Widgets for a Partial Form Field

For single text fields or select boxes, Django's built-in widgets, such as `widgets.TextInput` or `widgets.RadioSelect` can be used. Sometimes these simple widgets are not enough, therefore some special input widgets have been prepared to be used with **DjangoCMS-Cascade**. They are all part of the module `cmsplugin_cascade.widgets`.

MultipleTextInputWidget Use this widget to group a list of text input fields together. This for instance is used, to encapsulate all inline styles into one JSON object.

NumberInputWidget The same as Django's `TextInput`-widget, but doing field validation. This checks if the entered input data is a valid number.

MultipleInlineStylesWidget The same as the `MultipleTextInputWidget`, but doing field validation. This checks if the entered input data ends with `px` or `em`.

Overriding the Form

For the editor, **djangocms-cascade** automatically creates a form for each `GlossaryField` in the plugin's class. Sometimes however, you might need more control over the fields displayed in the editor, versus the fields stored inside the `glossary`.

Similar to the Django's `admin.ModelAdmin`, this can be achieved by overriding the plugins form element. Such a customized form can add as many fields as required, while the controlled glossary contains a compact summary.

To override the plugins form, add a member `form` to your plugin. This member variable shall refer to a customized form derived from `forms.models.ModelForm`. For further details about how to use this feature, refer to the supplied implementations.

Overriding the Model

Since all **djangocms-cascade** plugins store their data in a JSON-serializable field, there rarely is a need to add another database field to the common models `CascadeElement` and/or `SharableCascadeElement` and thus no need for database migrations.

However, quite often there is a need to add or override the methods for these models. Therefore each Cascade plugin creates its own proxy model on the fly. These models are derived from `CascadeElement` and/or `SharableCascadeElement` and named like the plugin class, with the suffix `Model`. By default, their behavior is the same as for their parent model classes.

To extend this behavior, the author of a plugin may declare a tuple of mixin classes, which are injected during the creation of the proxy model. Example:

```
class MySpecialPropertyMixin(object):
    def processed_value(self):
        value = self.glossary.get('field_name')
        # process value
        return value

class MySpecialPlugin(LinkPluginBase):
    module = 'My Module'
    name = 'My special Plugin'
    model_mixins = (MySpecialPropertyMixin,)
    render_template = 'my_module/my_special_plugin.html'
    field_name = GlossaryField(widgets.TextInput())
    ...
```

The proxy model created for this plugin class, now contains the extra method `content()`, which for instance may be accessed during template rendering.

`templates/my_module/my_special_plugin.html`:

```
<div>{{ instance.processed_value }}</div>
```

Needless to say, that you can't add any extra database fields to the class named `MySpecialPropertyMixin`, since the corresponding model class is marked as proxy.

Javascript

In case your customized plugin requires some Javascript code to improve the editor's experience, please refer to the section *Handling the client side*.

Transparent Plugins

Some of the plugins in Cascade's ecosystem are considered as *transparent*. This means that they logically don't fit into the given grid-system, but should rather be considered as wrappers of other HTML elements.

For example, the Bootstrap Panel can be added as child of a Column. However, it may contain exactly the same plugins, as the Column does. Now, instead of adding the `PanelPlugin` as a possible parent to all of our existing Bootstrap plugins, we simply declare the Panel as "transparent". It then behaves as it's own parent, allowing all plugins as children, which themselves are permitted to be added to that column.

Transparent plugins can be stacked. For example, the Bootstrap Accordion consists of one or more Accordion Panels. Both of them are considered as *transparent*, which means that we can add all plugins to an Accordion Panels, which we also could add to a Column.

Plugin Attribute Reference

`CascadePluginBase` is derived from `CMSPluginBase`, so all CMSPluginBase attributes can also be overridden by plugins derived from `CascadePluginBase`. Please refer to their documentation for details.

Additionally `BootstrapPluginBase` allows the following attributes:

name This name is shown in the pull down menu in structure view. There is not default value.

app_label The app_label to use on generated proxy models. This should usually be the same as the app_label of the app that defines the plugin.

tag_type A HTML element into which this plugin is wrapped. Generic templates can render their content into any `tag_type`. Specialized rendering templates usually have a hard coded tag type, then this attribute can be omitted.

require_parent Default: `True`. This differs from `CMSPluginBase`.

Is it required that this plugin is a child of another plugin? Otherwise the plugin can be added to any placeholder.

parent_classes Default: None.

A list of Plugin Class Names. If this is set, the plugin may only be added to plugins listed here.

allow_children Default: `True`. This differs from `CMSPluginBase`.

Can this plugin have child plugins? Or can other plugins be placed inside this plugin?

child_classes Default: A list of plugins, which are allowed as children of this plugin. This differs from `CMSPluginBase`, where this attribute is None.

Do not override this attribute. **DjangoCMS-Cascade** automatically generates a list of allowed children plugins, by evaluating the list `parent_classes` from the other plugins in the pool.

Plugins, which are part of the plugin pool, but which do not specify their parents using the list `parent_classes`, may be added as children to the current plugin by adding them to the attribute `generic_child_classes`.

generic_child_classes Default: None.

A list of plugins which shall be added as children to a plugin, but which themselves do not declare this plugin in their `parent_classes`.

default_css_class Default: None.

A CSS class which is always added to the wrapping DOM element.

default_inline_styles Default: None.

A dictionary of inline styles, which is always added to the wrapping DOM element.

get_identifier This is a classmethod, which can be added to a plugin to give it a meaningful name.

Its signature is:

```
@classmethod
def get_identifier(cls, obj):
    return 'A plugin name'
```

This method shall be used to name the plugin in structured view.

form Override the form used by the plugin editor. This must be a class derived from `forms.models.ModelForm`.

model_mixins Tuple of mixin classes, with additional methods to be added the auto-generated proxy model for the given plugin class.

Check section "Overriding the Model" for a detailed explanation.

Deprecated attributes

glossary_fields This list of `PartialFormFields` had been replaced by arbitrary class attributes of type `GlossaryField`.

Plugin Permissions

To register (or unregister) a plugin, simply invoke `./manage.py migrate cmsplugin_cascade`. This will add (or remove) the content type and the model permissions. We therefore can control in a very fine grained manner, which user or group is allowed to edit which types of plugins.

Generic Plugins

Cascade is shipped with a few plugins, which can be used independently of the underlying CSS framework. To avoid duplication, they are bundled into the section **Generic** and are available by default in the placeholders context menu.

All these plugins qualify as plugins with extra fields, which means that they can be configured by the site administrator to accept additional CSS styles and classes.

SimpleWrapperPlugin

Use this plugin to add a wrapping element around a group of other plugins. Currently these HTML elements can be used as wrapper: `<div>`, `<span>`, `<section>`, `<article>`. There is one special wrapper named `naked`. It embeds its children only logically, without actually embedding them into any HTML element.

HorizontalRulePlugin

This plugins adds a horizontal rule `<hr>` to the DOM. It is suggested to enable the `margin-top` and `margin-bottom` CSS styles, so that the ruler can be positioned appropriately.

HeadingPlugin

This plugins adds a text heading `<h1>`...''<h6>'' to the DOM. Although simple headings can be achieved with the **TextPlugin**, there they can't be styled using special CSS classes or styles. Here the **HeadingPlugin** can be used, since any allowed CSS class or style can be added.

CustomSnippetPlugin

Not every collection of DOM elements can be composed using the Cascade plugin system. Sometimes one might want to add a simple HTML snippet. Altough it is quite simple to create a customized plugin yourself, an easier approach to just render an arbitrary HTML snippet, is to use the **CustomSnippetPlugin**. This can be achieved by adding the customized template to the project's `settings.py`:

```
CMSPLUGIN_CASCADE = {
    # other settings
    'plugins_with_extra_render_templates': {
        'CustomSnippetPlugin': [
            ('myproject/snippets/custom-template.html', "Custom Template Identifier"),
            # other tuples
        ],
    },
}
```

Now, when editing the page, a plugin named **Custom Snippet** appears in the *Generic* section in the plugin's dropdown menu. This plugin then offers a select element, where the site editor then can chose between the configured templates.

Adding children to a CustomSnippetPlugin

It is even possible to add children to the **CustomSnippetPlugin**. Simple add these templatetag_s to the customized template, and all plugins which are children of the **CustomSnippetPlugin** will be rendered as well.

```
{% load cms_tags %}
<wrapping-element>
{% for plugin in instance.child_plugin_instances %}
    {% render_plugin plugin %}
{% endfor %}
</wrapping-element>
```

Release History

0.12.5

- Fixed: Wrapper for transparent plugins did not find all children which declared these kind of plugins as their parents.

0.12.4

- Fixed: Initial Image is reseted after reopening Image plugin editor.
- Changed order of fields in Accordion plugin editor.
- Moved directory `workdir` for demo project from root folder into examples.

0.12.3

- Fixed: When using an Element ID while adding a Heading Plugin, under certain circumstances the validation ran into an infinite loop.

0.12.2

- Fixed: Allow transparent instances as root objects.

0.12.1

- Fixed: Do not invoke `{% addtoblock "css" %}...` for empty values of `stylesheet_url`.
- Renamed buttons in clipboard admin to "Insert Data" (instead of "Save") and "Restore Data" (instead of "restore").

0.12.0

- Added compatibility for Django version 1.10.
- Added compatibility for django-CMS version 3.4.
- Added monkey patch to resolve issues handled by PR https://github.com/divio/django-cms/pull/5809
- Added compatibility for djangocms-text-ckeditor-3.4.
- **Important for AngularJS users**: Please upgrade to angular-ui-bootstrap version 0.14.3. All versions later than 0.13 use the prefix `uib-` on all AngularJS directives, hence this upgrade is required.
- In the `CarouselSlide` plugin, caption is added as a child `TextPlugin` instead of using the glossary. Currently the migration of `TextLinkPlugins` inside this caption field does not work properly. Please create an issue, if you really need it.
- Added method `value_omitted_from_data` to `JSONMultiWidget` to override the Django method implemented in `django.forms.widgets.MultiWidget`.
- In `cmsplugin_cascade.models.CascadeElement` the foreign key `shared_glossary` now is marked as editable. Instead to plugins without sharable glossary, the attribute `exclude = ['shared_glossary']` is added.
- Instead of handling ring.js plugin inheritance through `get_ring_bases()`, Cascade plugins just have to add `ring_plugin = '...'` to their class declaration.
- Function `cmsplugin_cascade.utils.resolve_dependencies` is deprecated, since Javascript dependencies now are handled via their natural inheritance relation.
- The configuration option `settings.CMSPLUGIN_CASCADE['dependencies']` has been removed.

- Added method `save()` to model `SharedGlossary`, which filters the glossary to be stored to only those fields marked as sharable.
- Accessing the CMS page via `plugin_instance.page` is deprecated and has been replaced by invocations to `plugin_instance.placeholder.page`.
- Removed directory `static/cascade/css/fonts/glyphicons-halflings`, since they are available through the Bootstrap npm packages.
- All Javascript files accessing a property `disabled`, now use the proper jQuery function intended for it.
- Added interface to upload fonts and use them as framed icons, text icons or button decorators.
- The permission system now is fine grained. Administrators can give their staff users add/change/delete permissions to each of the many Cascade plugins. When adding new plugins, this does not even require a database migration.
- Fixed: On saving a **CarouselPlugin**, the glossary of it's children, ie. **CarouselSlidePlugin**, is sanitized.
- Handle the high resolution of the **PicturePlugin** through `srcset` rather than a `@media` query.
- Handle the high resolution background of the **JumbotronPlugin** through `image-set` rather than a `@media` query.
- Use default configurations from provides Cascade settings rathern than from the Django project.

0.11.1

- Added preconfigured `FilePathField` to prevent the creation of useless migration files.
- SegmentPlugin.get_form OrderedDict value lookups now compatible with python3.
- Fixed database migration failing on multiple database setup.

0.11.0

- Instead of adding a list of `PartialFormField``s named ``glossary_fields`, we now can add these fields to the plugin class, as we would in a Django `forms.Form` or `models.Model`, for instance: `fieldname = GlossaryField(widget, label="A Label", initial=some_value)` instead of `glossary_fields = <list-or-tuple-of PartialFormField s>`. This is only important for third party apps inheriting from `CascadePluginBase`.

  Remember: In some field names, the - (dash) has been replaced against an _ (underscore). Therefore please run `./manage.py migrate cmsplugin_cascade` which modifies the plugin's payloads.

0.10.2

- Fix #188: Using shared settings does not remember it's value.

0.10.1

- Fix #185: Undefined variables in case of uncaught exception.

0.10.0

- Added **BootstrapJumbotronPlugin**. This for instance can be used to place background images extending over the full width of a page using a parallax effect.
- *Experimental*: Utility to manage font icons, so that symbol icons can be used anywhere in any size.
- `CMSPLUGIN_CASCADE['plugins_with_extra_fields']` is a dict instead of a tuple. This allows the site administrator to enable extra styles globally and without adding them using the administration backend.
- Tuples in `CMSPLUGIN_CASCADE['bootstrap3']['breakpoints']` now accepts five parameters instead of four. The 5th parameter specifies the image width for fluid containers and the Jumbotron plugin.
- The plugin's change form now can add an introduction and a footnote HTML. This is useful to add some explanation text.

0.9.4

- Added function `.utils.validate_link` to check if submitted link information is valid.

0.9.3

- Fixed: enabled subject_location did not work properly for **ImagePlugin** and **PicturePlugin**.
- Fixed indention in admin interface for extra fields model.
- Moved template 'testing.html' -> 'cascade/testing.html'.
- Added German translations.

0.9.2

- Restore global jQuery object (required by the Select2 widget) in explicit file instead of doing it implicitly in `linkpluginbase.js`

0.9.1

- Prepared for django-1.10
- Upgrade ring.js to version 2.1.0
- In LinkPlugin, forgive if sub-dict `link` was missing in `glossary`
- Fixed HTML escaping problem in Bootstrap Carousel
- Increase height of Select2 fields

0.9.0

- Compatible with django-cms version 3.3.0
- Converted `SharableCascadeElement` into a proxy model, sharing the same data as model `CascadeElement`. This allows adding plugins to `CMSPLUGIN_CASCADE['plugins_with_sharables']` without requiring a data-migration. (**Note:** A migration merges the former two models, so please backup your database before upgrading!)

- Add support for Section Bookmarks.
- Fixed: Do not set width/height on <img>-element inside a <picture>, if wrapping container is fluid.
- Replaced configuration settings `CMSPLUGIN_CASCADE_LINKPLUGIN_CLASSES` against `CMSPLUGIN_CASCADE['link_plugin_classes']` for better consistency.

Note: If you want to continue using django-CMS 3.2 please use djangocms-cascade 0.8.5.

0.8.5

- Dropped support for Python-2.6.

0.8.4

- Fixed a regression in "Restore from clipboard".
- Fixed TextLinkPlugin to work again as child of TextPlugin.
- ContainerPlugin can only be added below a placeholder.
- Prepared demo to work with Django-1.10.
- Plugins marked as "transparent" are only allowed as parents, if they allow children.

0.8.3

- Added `CustomSnippetPlugin`. It allows to add arbitrary custom templates to the project.
- Fixed #160: Error copying Carousel plugin
- Plugins marked as "transparent" can be parents of everybody.
- BootstrapPanelPlugin now accepts inline CSS styles.

0.8.2

- Cascade does not create migrations for proxy models anymore. This created major problems if Cascade components have been switched on and off. All existing migrations of proxy models have been removed from the migration files.
- Fixed: Response of more than one entry on non unique clipboards.
- Added `cmsplugin_cascade.models.SortableInlineCascadeElement` which can be used for keeping sorted inline elements.
- `cmsplugin_cascade.bootstrap3.gallery.BootstrapGalleryPlugin` can sort its images.

0.8.1

- Hotfix: removed invalid dependency in migration 0007.

0.8.0

- Compatible with Django-1.9
- Fixed #133: BootstrapPanelPlugin now supports custom CSS classes.
- Fixed #132: Carousel Slide plugin with different form.
- Fixed migration problems for proxy models outside Cascade.
- Replaced SelectMultiple against CheckboxSelectMultiple in admin for extra fields.
- Removed SegmentationAdmin from admin backend.
- Disallow whitespace in CSS attributes.
- Require django-reversion 1.10.1 or newer.
- Require django-polymorphic 0.9.1 or newer.
- Require django-filer 1.1.1 or newer.
- Require django-treebeard 4.0 or newer.
- Require django-sekizai 0.9.0 or newer.

0.7.3

- Use the outer width for fluid containers. This allows us to add images and carousels which extend the browser's edges.
- Fixed #132: Carousel Slide plugin different form.
- Fixed #133: BootstrapPanelPlugin does not support custom CSS classes.
- Fixed #134: More plugins can be children of the `SimpleWrapperPlugin`. This allows us to be more flexible when building the DOM tree.
- `BootstrapContainerPlugin` now by default accepts extra inline styles and CSS classes.

0.7.2

- Add a possibility to prefix Cascade plugins with a symbol of your choice, to avoid confusion if the same name has been used by another plugin.
- All Bootstrap plugins can override their templates globally though a configuration settings variable. Usefule to switch between jQuery and AngularJS versions of a widget.
- Added TabSet and TabPanel plugins.
- It is possible to persist the content of the clipboard in the database, retrieve and export it as JSON to be reimported on an unrelated site.

0.7.1

- Added a **HeadingPlugin** to add single text headings independently of the HTML TextEditorPlugin.

0.7.0

Cleanup release, removing a lot of legacy code. This adds some incompatibilities to previous versions:

- Instead of half o dozen of configuration directives, now one Python dict is used. Therefore check your `settings.py` for configurations starting with `CMSPLUGIN_CASCADE_...`.
- Tested with **Django-1.8**. Support for version 1.7 and lower has been dropped.
- Tested with **djangoCMS** version 3.2. Support for version 3.0 and lower has been dropped.
- Tested with **django-select2** version 5.2. Support for version 4 has been dropped.
- The demo project now uses SASS instead of plain CSS, but SASS is not a requirement during normal development.

0.6.2

- In Segment: A condition raising a TemplateSyntaxError now renders that error inside a HTML comment. This is useful for debugging non working conditions.
- In Segment: An alternative AdminModel to UserAdmin, using a callable instead of a model field, now works.
- In Segment: It is possible to use `segmentation_list_display = (list-of-fields)` in an alternative AdminModel, to override the list view, when emulating a user.

0.6.1

- Added a panel plugin to support the Bootstrap Panel.
- Added experimental support for secondary menus.
- Renamed `AccordionPlugin` to `BootstrapAccordionPlugin` for consistency and to avoid future naming conflicts.

0.6.0

- Fixed #79: The column width is not reduced in width, if a smaller column precedes a column for a smaller displays.
- Fixed: Added extra space before left prefix in buttons.
- Enhanced: Access the link content through the glossary's `link_content`.
- New: Plugins now can be rendered using an alternative template, choosable through the plugin editor.
- Fixed in SegmentationPlugin: When overriding the context, this updated context was only used for the immediate child of segment. Now the overridden context is applied to all children and grandchildren.
- Changed in SegmentationPlugin: When searching for siblings, use a list index instead of `get_children().get(position=...)`.
- Added unit tests for SegmentationPlugin.
- Added support for **django-reversion**.
- By using the setting `CMSPLUGIN_CASCADE_LINKPLUGIN_CLASSES`, one can replace the class `LinkPluginBase` by an alternative implementation.
- When using *Extra Styles* distances now can have negative values.

- In caption field of `CarouselSlidePlugin` it now is possible to set links onto arbitrary pages.

Possible backwards incompatibility:

- For consistency with naming conventions on other plugins, renamed `cascade/plugins/link.html` -> `cascade/link/link-base.html`. **Check your templates**!
- The setting `CMSPLUGIN_CASCADE_SEGMENTATION_MIXINS` now is a list of two-tuples, where the first declares the plugin's model mixin, while the second declares the model admin mixin.
- Removed from setting: `CMSPLUGIN_CASCADE_BOOTSTRAP3_TEMPLATE_DIR`. The rendering template now can be specified during runtime.
- Refactored and moved `SimpleWrapperPlugin` and `HorizontalRulePlugin` from `cmsplugin_cascade/bootstrap3/` into `cmsplugin_cascade/generic/`. The glossary field `element_tag` has been renamed to `tag_type`.
- Refactored `LinkPluginBase` so that external implementations can create their own version, which then is used as base for TextLinkPlugin, ImagePlugin and PicturePlugin.
- Renamed: `PanelGroupPlugin` -> `Accordion`, `PanelPlugin` -> `AccordionPanelPlugin`, because the Bootstrap project renamed them back to their well known names.

0.5.0

- Added SegmentationPlugin. This allows to conditionally render parts of the DOM, depending on the status of various `request` object members, such as `user`.
- Setting `CASCADE_LEAF_PLUGINS` has been replaced by `CMSPLUGIN_CASCADE_ALIEN_PLUGINS`. This simplifies the programming of third party plugins, since the author of a plugin now only must set the member `alien_child_classes = True`.

0.4.5

- Fixed: If no breakpoints are set, don't delete widths and offsets from the glossary, as otherwise this information is lost.
- Fixed broken import for `PageSelectFormField` when not using **django_select2**.
- Admin form for `PluginExtraFields` now is created on the fly. This fixes a rare circular dependency issue, when accessing `plugin_pool.get_all_plugins()`.

0.4.4

- Removed hard coded input fields for styling margins from **BootstrapButtonPlugin**, since it is possible to add them through the **Extra Fields** dialog box.
- [Column ordering](http://getbootstrap.com/css/#grid-column-ordering) using `col-xx-push-n` and `col-xx-pull-n` has been added.
- Fixed: Media file `linkplugin.js` was missing for **BootstrapButtonPlugin**.
- Hard coded configuration option `EXTRA_INLINE_STYLES` can now be overridden by the projects settings

0.4.3

- The templatetag `bootstrap3_tags` and the templates to build Boostrap3 styled menus, breadcrumbs and paginator, have been moved into their own repository at https://github.com/jrief/djangocms-bootstrap3.
- Column ordering using `col-xx-push-n` and `col-xx-pull-n` has been added.

0.4.2

- Fixed: Allow empty setting for CMSPLUGIN_CASCADE_PLUGINS
- Fixed: Use str(..) instead of b" in combination with from __future__ import unicode_literals

0.4.1

- Fixed: Exception when saving a ContainerPlugin with only one breakpoint.
- The `required` flag on a field for an inherited LinkPlugin is set to False for shared settings.
- Fixed: Client side code for disabling shared settings did not work.

0.4.0

- Renamed `context` from model `CascadeElement` to `glossary`. The identifier ``context`` lead to too much confusion, since it is used all way long in other CMS plugins, where it has a complete different meaning.
- Renamed `partial_fields` in all plugins to `glossary_fields`, since that's the model field where they keep their information.
- Huge refactoring of the code base, allowing a lot of more features.

0.3.2

- Fixed: Missing unicode conversion for method `get_identifier()`
- Fixed: Exception handler for form validation used `getattr` incorrectly.

0.3.1

- Added compatibility layer for Python-3.3.

0.3.0

- Complete rewrite. Now offers elements for Bootstrap 3 and other CSS frameworks.

0.2.0

- Added carousel.

0.1.2

- Fixed: Added missign migration.

0.1.1

- Added unit tests.

0.1.0

- First published revision.

Thanks

This DjangoCMS plugin originally was derived from https://github.com/divio/djangocms-style, so the honor for the idea of this software goes to Divio and specially to Patrick Lauber, aka digi604.

However, since my use case is different, I removed all the existing code and replaced it against something more generic suitable to add a collection of highly configurable plugins.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index

P

PluginExtraFieldsConfig (class in cmsplugin_cascade.extra_fields.config), 40