django-dash Documentation

Release 0.6.1

Artur Barseghyan <artur.barseghyan@gmail.com>

Feb 21, 2022

Contents

`django-dash` (later on named Dash) is a customisable, modular dashboard application framework for Django.

Dash allows users to create their own custom dashboards. Supports theeming (in Dash themes are called layouts) and multiple workspaces. Dash comes with extensive pythonic API which allows developers to create new Dash plugins, as well as to modify bundled ones.

To make a clearer association, think of Android for tablets (shortcuts, widgets and apps) or Windows 8 for tablets or desktops.

Dash inherits all those concepts and makes it possible to implement a dashboard system for Django applications with minimal efforts.

CHAPTER 1

Prerequisites

- Django 2.2, 3.0 and 3.1
- Python 3.6, 3.7, 3.8 and 3.9.

CHAPTER 2

Key concepts

- Each layout (theme) consist of placeholders. Each plugin widget has its' own specific HTML/JavaScript/CSS.
- There might be multiple themes implemented and installed. Default layout is chosen system wide, but each user (if has an appropriate permission) can choose his preferred layout over all workspaces or even different layouts per workspace.
- Placeholder is a space, in which the plugin widgets are placed.
- Placeholders are rectangles consisting of cells. Each placeholder has its' own custom number of rows and columns.
- Workspace is just another named dashboard. Users switch between workspaces in navigation. Amount of workspaces is unlimited.
- Plugin is a (Django) micro app. Most heavy work should happen in plugin. Plugin may have its' own views, urls, etc. Rendering happens with use of plugin widgets.
- Plugin widgets are mainly responsible for rendering of the plugin data. Each plugin widget has its' own specific HTML/JavaScript/CSS. A single plugin widget is registered for a triple (layout, placeholder, plugin).
- Public dashboard (implemented as a contrib app, which makes it optional) allows users to make their workspaces public. If user chooses to make his dashboard public, default workspace becomes public. As for non-default workspaces, user can still make each of them private or public.

CHAPTER 3

Main features

- Customisable layouts (aka theming).
- Multiple workspaces.
- Tunable access permissions to plugins.
- Public dashboards (as a contrib app).
- Cloneable workspaces. It's possible to clone entire workspace, including all the plugins into another workspace.
- Copy/paste functionality for plugin widgets.

CHAPTER 4

FAQ

- Question: Is it possible to have Dash working with a (pick what's applicable: D3, Polychart2, or some other library for making charts).

  Answer: Yes. Check the source code of the following sample plugins:

  - Sample D3 plugins.
  - Sample Polychart2 plugin.

CHAPTER 5

Some screenshots

See the documentation for some screen shots:

- ReadTheDocs

CHAPTER 6

Demo

6.1 Live demo

See the live demo app on Heroku.

Credentials:

- username: test_user
- password: test_user

See the public dashboard of a test_demo_user to get an idea of what it could become.

6.2 Run demo locally

In order to be able to quickly evaluate the *django-dash*, a demo app (with a quick installer) has been created (works on Ubuntu/Debian, may work on other Linux systems as well, although not guaranteed). Follow the instructions below for having the demo running within a minute.

Grab the latest `django_dash_example_app_installer.sh`:

```
wget https://raw.github.com/barseghyanartur/django-dash/stable/examples/django_dash_
↪example_app_installer.sh
```

Assign execute rights to the installer and run the `django_dash_example_app_installer.sh`:

```
chmod +x django_dash_example_app_installer.sh

./django_dash_example_app_installer.sh
```

Open your browser and test the app.

Dashboard:

- URL: http://127.0.0.1:8001/en/dashboard/

- Admin username: test_admin
- Admin password: test

Django admin interface:

- URL: http://127.0.0.1:8001/en/administration/
- Admin username: test_admin
- Admin password: test

If quick installer doesn't work for you, see the manual steps on running the example project.

Take a look at the templates in "example/example/templates" directory for getting a better idea of how to transform your own or third-party templates into Dash templates.

Also, the example project has example layouts, plugins and widgets implemented. Take it as a good example of how to add widgets for existing plugins to your own custom layout. Make sure to see how same is done for the bundled layouts.

CHAPTER 7

Installation

1. Install latest stable version from PyPI:

```
pip install django-dash
```

   Or latest stable version from GitHub:

```
pip install https://github.com/barseghyanartur/django-dash/archive/stable.tar.gz
```

   Or latest stable version from BitBucket:

```
pip install https://bitbucket.org/barseghyanartur/django-dash/get/stable.tar.gz
```

2. Add *dash* to `INSTALLED_APPS` of the your projects' Django settings. Furthermore, all layouts and plugins to be used, shall be added to the `INSTALLED_APPS` as well.

```
INSTALLED_APPS = (
    # ...
    'dash',
    'dash.contrib.layouts.android',
    'dash.contrib.layouts.bootstrap2',
    'dash.contrib.layouts.windows8',
    'dash.contrib.plugins.dummy',
    'dash.contrib.plugins.image',
    'dash.contrib.plugins.memo',
    'dash.contrib.plugins.rss_feed',
    'dash.contrib.plugins.url',
    'dash.contrib.plugins.video',
    'dash.contrib.plugins.weather',
    # ...
)
```

3. Make sure that `django.core.context_processors.request` is in `TEMPLATE_CONTEXT_PROCESSORS`.
4. Add necessary URL patterns to your `urls` module.

```
re_path(r'^dashboard/', include('dash.urls')),
```

Additionally, add all URLs of any Dash apps or plugins

```
# django-dash RSS contrib plugin URLs:
re_path(r'^dash/contrib/plugins/rss-feed/',
        include('dash.contrib.plugins.rss_feed.urls')),

# django-dash public dashboards contrib app:
re_path(r'^', include('dash.contrib.apps.public_dashboard.urls')),
```

CHAPTER 8

Creating a new layout

Dash comes with several bundled layouts. Do check their source code as example.

Let's say, our imaginary layout has two placeholders. One large placeholder for all kinds of widgets (called `main`) and a tiny one for shortcuts (called `shortcuts`).

Placeholder `main`:

- Single cell size : 150 x 110 pixels
- Dimensions : 6 cols, 5 rows

Placeholder `shortcuts`:

- Single cell size : 60 x 55 pixels
- Dimensions : 1 cols, 10 rows

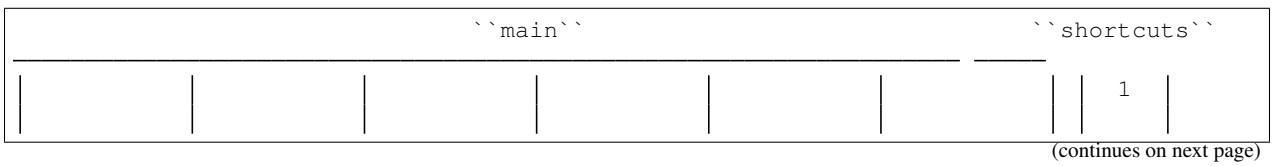
See the figure below to get an idea of what placeholders are:

- Placeholder `main` consts of cells from 11 to 56.
- Placeholder `shortcuts` consists of cells from 1 to 10.

A single plugin widget may occupy one or more cells. Plugin widgets are rectangles.

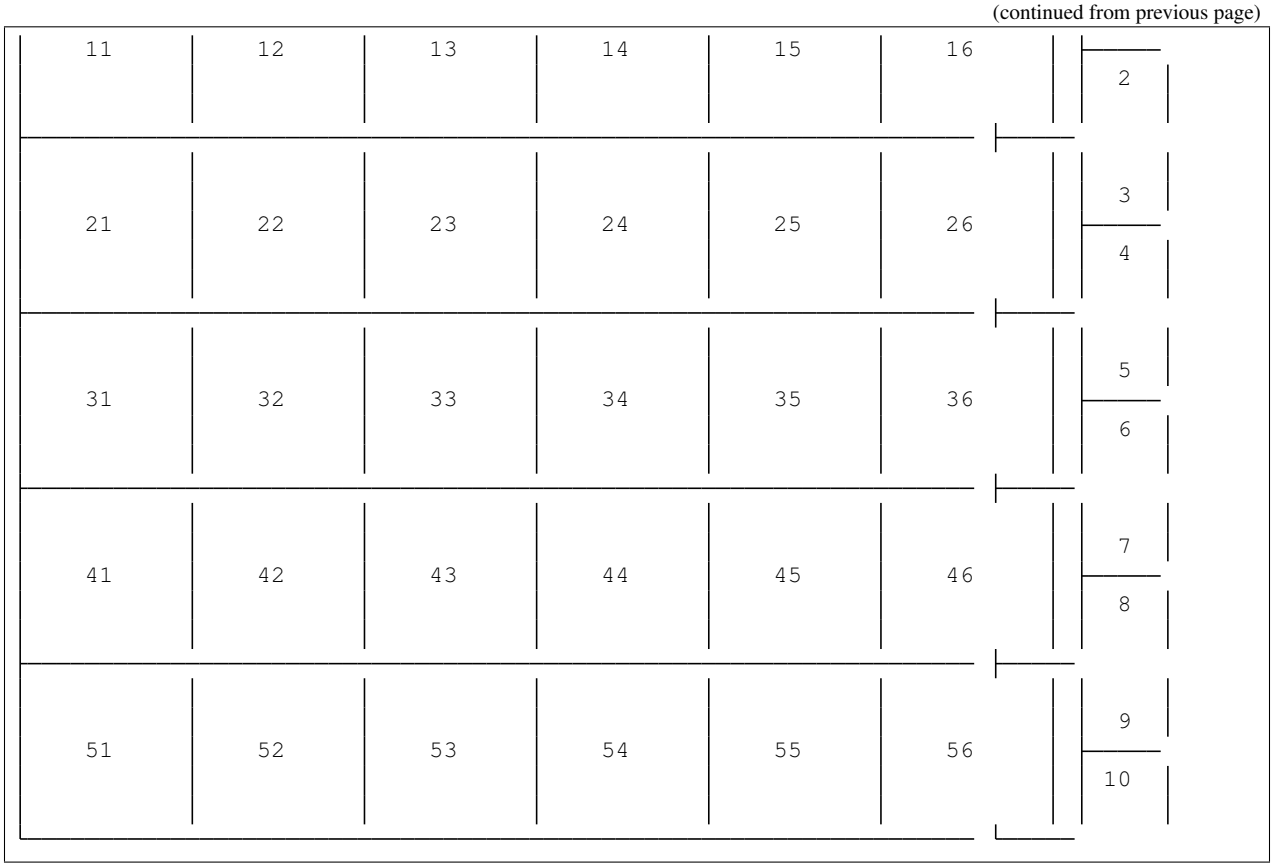
To make it clear, see following cases:

- Plugin widget has 2 cols and 1 row. Then, for example, it may occupy cells (11 and 12).
- Plugin widget has 2 cols and 2 rows. Then, for example, it may occupy cells (11, 12, 21 and 22).
- Plugin widget has 1 col and 3 rows. Then, for example, it may occupy cells (11, 21 and 31).
- Plugin widget has 4 cols and 3 rows. Then, for example, it may occupy cells (22, 23, 24, 25, 32, 33, 34, 35, 42, 43, 44 and 45).



(continues on next page)

(continued from previous page)



There are some rules/guidelines you should follow.

Let's assume that layout is named `example`. The layout directory should then have the following structure.

```
path/to/layout/example/
├── static
│   ├── css
│   │   └── dash_layout_example.css # Contains layout-specific CSS
│   ├── images
│   └── js
│       └── dash_layout_example.js # Contains layout specific JavaScripts
├── templates
│   └── example
│       ├── edit_layout.html # Master edit layout
│       └── view_layout.html # Master view layout
├── __init__.py
├── dash_layouts.py # Where layouts and placeholders are defined and registered
├── dash_plugins.py # Where layout specific plugins and plugin widgets are defined␣
↪and registered
└── dash_widgets.py # Where layout specific plugin widgets are defined
```

Layout and placeholder classes should be placed in the `dash_layouts.py` file.

Each layout should be put into the `INSTALLED_APPS` of your Django projects' `settings.py` module.

```
INSTALLED_APPS = (
    # ...
    'path.to.layout.example',
```

(continues on next page)

(continued from previous page)

```
    # ...
)
```

8.1 path/to/layout/example/dash_layouts.py

Step by step review of a how to create and register a layout and placeholders. Note, that Dash auto-discovers your layouts by name of the file `dash_layouts.py`. The module, in which the layouts are defined, has to be named `dash_layouts.py`.

Required imports.

```
from dash.base import BaseDashboardLayout, BaseDashboardPlaceholder
from dash.base import layout_registry
```

Defining the Main placeholder.

```
class ExampleMainPlaceholder(BaseDashboardPlaceholder):

    uid = 'main'  # Unique ID of the placeholder.
    cols = 6  # Number of columns in the placeholder.
    rows = 5  # Number of rows in the placeholder.
    cell_width = 150  # Width of a single cell in the placeholder.
    cell_height = 110  # Height of a single cell in the placeholder.
```

Defining the Shortcuts placeholder.

```
class ExampleShortcutsPlaceholder(BaseDashboardPlaceholder):

    uid = 'shortcuts'  # UID of the placeholder.
    cols = 1  # Number of columns in the placeholder.
    rows = 10  # Number of rows in the placeholder.
    cell_width = 60  # Width of a single cell in the placeholder.
    cell_height = 55  # Height of a single cell in the placeholder.
```

Defining and registering the Layout.

```
class ExampleLayout(BaseDashboardLayout):

    uid = 'example'  # Layout UID.
    name = 'Example'  # Layout name.

    # View template. Master template used in view mode.
    view_template_name = 'example/view_layout.html'

    # Edit template. Master template used in edit mode.
    edit_template_name = 'example/edit_layout.html'

    # All placeholders listed. Note, that placeholders are rendered in the
    # order specified here.
    placeholders = [ExampleMainPlaceholder, ExampleShortcutsPlaceholder]

    # Cell units used in the entire layout. Allowed values are: 'px',
    # 'pt', 'em' or '%'. In the ``ExampleMainPlaceholder`` cell_width is
    # set to 150. It means that in this particular case its' actual width
```

(continues on next page)

(continued from previous page)

```
    # would be `150px`.
    cell_units = 'px'

    # Layout specific CSS.
    media_css = ('css/dash_layout_example.css',)

    # Layout specific JS.
    media_js = ('js/dash_layout_example.js',)

# Registering the layout.
layout_registry.register(ExampleLayout)
```

8.2 HTML templates

You custom layout should be inherited from base layout templates (view or edit). Both view and edit layouts share a lot of things, still edit layout is a bit more "heavy".

- view_layout.html should inherit from "dash/layouts/base_view_layout.html".
- edit_layout.html should inherit from "dash/layouts/base_edit_layout.html".

Both "dash/layouts/base_view_layout.html" and "dash/layouts/base_edit_layout.html" inherit from "dash/layouts/base_layout.html", which in its' turn inherits from "dash/base.html".

Note, that when rendered to HTML, each Dash template, gets a body class "layout" + layouts' unique identifier (UID). So, the `ExampleLayout` layout would automatically get the class "layout-example".

```
<body class="layout-example">
```

In case of Android layout (UID "android") it would be as follows.

```
<body class="layout-android">
```

Base your layout specific custom CSS on presence of those classes.

Same goes for Placeholders. Each placeholder gets `id_` + placeholders' UID and the classes "placeholder" and "placeholder-" + placeholders' UID. So, the `ExampleMainPlaceholder` would look as follows.

```
<div id="id_main" class="placeholder placeholder-main">
```

And the `ExampleShortcutsPlaceholder` placeholder would look as follows.

```
<div id="id_shortcuts" class="placeholder placeholder-shortcuts">
```

Same goes for plugin widgets. Apart from some other classes that each plugin widget would get for positioning, it gets the "plugin" and "plugin-" + plugin UID. See the following example (for the plugin Dummy with UID "dummy"). Each plugin also gets an automatic UID on the moment when rendered. In the example below it's the "p6d06f17d-e142-4f45-b9c1-893c38fc2b01".

```
<div id="p6d06f17d-e142-4f45-b9c1-893c38fc2b01" class="plugin plugin-dummy">
```

Layout, placeholder, plugin and plugin widget have properties for getting their HTML specific classes and IDs.

Layout (instance)

```
layout.html_class
```

Placeholder (instance)

```
placeholder.html_id
placeholder.html_class
```

Plugin (instance)

```
plugin.html_id
plugin.html_class
```

Plugin widget (static call)

```
plugin_widget.html_class  # Static one
```

CHAPTER 9

Creating a new plugin

Dash comes with several bundled plugins. Do check their source code as example.

Making of a plugin or a plugin widget is quite simple, although there are some rules/guidelines you should follow.

Let's assume that plugin is named `sample_memo`. The plugin directory should then have the following structure.

Note, that you are advised to prefix all your plugin specific media files with `dash_plugin_` for the sake of common sense.

```
path/to/plugin/sample_memo/
├── static
│   ├── css
│   │   └── dash_plugin_sample_memo.css # Plugin specific CSS
│   ├── images
│   └── js
│       └── dash_plugin_sample_memo.js # Plugin specific JavaScripts
├── templates
│   └── sample_memo
│       ├── render_main.html # Plugin widget template for ``main`` Placeholder
│       └── render_short.html # Plugin widget template for ``shortcuts`` Placeholder
├── __init__.py
├── dash_plugins.py # Where plugins and widgets are defined and registered
├── dash_widgets.py # Where the plugin widgets are defined
└── forms.py # Plugin configuration form
```

In some cases, you would need plugin specific overridable settings (see `dash.contrib.plugins.weather` plugin as an example. You are advised to write your settings in such a way, that variables of your Django projects' `settings.py` module would have `DASH_PLUGIN_` prefix.

9.1 path/to/plugin/sample_memo/dash_plugins.py

Step by step review of a how to create and register a plugin and plugin widgets. Note, that Dash auto-discovers your plugins if you place them into a file named `dash_plugins.py` of any Django app listed in `INSTALLED_APPS` of your Django projects' settings module.

9.1.1 Define and register the plugin

As already stated, a single plugin widget is registered for a triple (layout, placeholder, plugin). That means, that if you need two widgets, one sized 1x1 and another sized 2x2, you need two plugins for it. You can either manually define all plugins and widgets for the sizes desired, or define a single base plugin or a widget class and have it factory registered for a number of given sizes. Below, both approaches would be explained.

Required imports.

```
from dash.base import BaseDashboardPlugin, plugin_registry
from path.to.plugin.sample_memo.forms import SampleMemoForm
```

Defining the Sample Memo plugin (2x2) (to be used in the `main` placeholder).

```
class SampleMemo2x2Plugin(BaseDashboardPlugin):

    uid = 'sample_memo_2x2'  # Plugin UID
    name = _("Memo")  # Plugin name
    group = _("Memo")  # Group to which the plugin belongs to
    form = SampleMemoForm  # Plugin forms are explained later
    html_classes = ['sample-memo']  # Optional. Adds extra HTML classes.
```

Registering the Sample Memo plugin.

```
plugin_registry.register(SampleMemo2x2Plugin)
```

Defining the Sample Memo plugin (1x1) (to be used in the `shortcuts` placeholder).

```
class SampleMemo1x1Plugin(SampleMemo2x2Plugin):

    uid = 'sample_memo_1x1'  # Plugin UID
```

Registering the Sample Memo plugin.

```
plugin_registry.register(SampleMemo1x1Plugin)
```

Repeat the steps below for each plugin size (or read about factory registering the plugins and widgets below).

9.1.2 Factory register plugins

Alternatively, you can define just a single plugin base class and have it factory registered for the given sizes. The code below would produce and register classes for in sizes 1x1 and 2x2. When you need to register a plugin for 10 sizes, this approach clearly wins. Besides, it's very easy to get a clear overview of all plugins sizes registered.

Required imports.

```
from dash.base import BaseDashboardPlugin
from dash.factory import plugin_factory
from path.to.plugin.sample_memo.forms import SampleMemoForm
```

Defining the base plugin class.

```
class BaseSampleMemoPlugin(BaseDashboardPlugin):

    name = _("Memo")  # Plugin name
    group = _("Memo")  # Group to which the plugin belongs to
```

(continues on next page)

(continued from previous page)

```
    form = SampleMemoForm  # Plugin forms are explained later
    html_classes = ['sample-memo']  # Optional. Adds extra HTML classes.
```

Note, that we don't provide `uid` property in the base class.

Now, that we have the base plugin defined, factory register it for the sizes given.

```
sizes = (
    (1, 1),
    (2, 2),
)
plugin_factory(BaseSampleMemoPlugin, 'sample_memo', sizes)
```

In the example above, "sample_memo" is the base name of the plugin. Size information would be appended to it ("sample_memo_1x1", "sample_memo_2x2").

9.1.3 Register plugin widgets

Plugin widgets are defined in `dash_widgets.py` module (described later), but registered in the `dash_plugins.py`, which is auto-discovered by Dash.

Required imports.

```
from dash.base import plugin_widget_registry
from path.to.plugin.sample_memo.dash_widgets import (
    SampleMemo1x1ExampleMainWidget,
    SampleMemo2x2ExampleMainWidget,
)
```

Registering the Sample Memo plugin widget for placeholder `main` of layout *example*'.

```
plugin_widget_registry.register(SampleMemo2x2ExampleMainWidget)
```

Registering the Sample Memo plugin widget for placeholder `shortcuts` of layout `example`.

```
plugin_widget_registry.register(SampleMemo1x1ExampleMainWidget)
```

9.2 path/to/plugin/sample_memo/dash_widgets.py

Why to have another file for defining widgets? Just to keep the code clean and less messy, although you could perfectly define all your plugin widgets in the module `dash_plugins.py`, it's recommended to keep it separate.

Take into consideration, that `dash_widgets.py` is not an auto-discovered file pattern. All your plugin widgets should be registered in modules named `dash_plugins.py`.

9.2.1 Define and register the plugin widget

Required imports.

```
from django.template.loader import render_to_string
from dash.base import BaseDashboardPluginWidget
```

Memo plugin widget for Example layout (placeholder `main`).

```
class SampleMemo2x2ExampleMainWidget(BaseDashboardPluginWidget):

    layout_uid = 'example'  # Layout for which the widget is written
    placeholder_uid = 'main'  # Placeholder within the layout for which
                              # the widget is written
    plugin_uid = 'sample_memo_2x2'  # Plugin for which the widget is
                                    # written
    cols = 2  # Number of widget columns
    rows = 2  # Number of widget rows

    def render(self, request=None):
        context = {'plugin': self.plugin}
        return render_to_string('sample_memo/render_main.html', context)
```

Memo plugin widget for Example layout (placeholder *shortcuts*).

```
class SampleMemo1x1ExampleShortcutWidget(SampleMemo2x2ExampleMainWidget):

    placeholder_uid = 'shortcuts'  # Placeholder within the layout for
                                   # which the widget is written
    cols = 1  # Number of widget columns
    rows = 1  # Number of widget rows

    def render(self, request=None):
        context = {'plugin': self.plugin}
        return render_to_string(
            'sample_memo/render_shortcuts.html', context
        )
```

9.2.2 Factory register plugin widgets

Alternatively, you can define just a single plugin widget base class and have it factory registered for the given sizes. The code below would produce and register classes for in sizes 1x1 and 2x2.

Required imports.

```
from django.template.loader import render_to_string
from dash.factory import plugin_widget_factory
from dash.base import BaseDashboardPluginWidget
```

Defining the base plugin widget class.

```
class BaseSampleMemoWidget(BaseDashboardPluginWidget):

    def render(self, request=None):
        context = {'plugin': self.plugin}
        return render_to_string('sample_memo/render.html', context)
```

Now, that we have the base plugin defined, factory register it for the sizes given.

```
sizes = (
    (1, 1),
    (2, 2),
)
plugin_widget_factory(
```

(continues on next page)

(continued from previous page)

```
    BaseSampleMemoWidget, 'example', 'main', 'sample_memo', sizes
)
```

In the example above:

- "sample_memo" is the base name of the plugin and it should match the name given to plugin factory exactly.
- "example" is the uid of the layout, for which the widget is being registered.
- "main" is the uid of the placeholder, for which the widget it being registered.

9.3 path/to/plugin/sample_memo/forms.py

What are the plugin forms? Very simple - if plugin is configurable, it has a form. If you need to have a custom CSS or a JavaScript included when rendering a specific form, use Django's class Media directive in the form.

Required imports.

```
from django import forms
from dash.base import DashboardPluginFormBase
```

Memo form (for *Sample Memo* plugin).

```
class SampleMemoForm(forms.Form, DashboardPluginFormBase):

    plugin_data_fields = [
        ("title", ""),
        ("text", "")
    ]

    title = forms.CharField(label=_("Title"), required=False)
    text = forms.CharField(label=_("Text"), required=True,
                           widget=forms.widgets.Textarea)

    def __init__(self, *args, **kwargs):
        super(MemoForm, self).__init__(*args, **kwargs)
```

Now, that everything is ready, make sure your that both layout and the plugin modules are added to `INSTALLED_APPS` for your projects' Django `settings.py` module.

```
INSTALLED_APPS = (
    # ...
    'path.to.layout.example',
    'path.to.plugin.sample_memo',
    # ...
)
```

After it's done, go to terminal and type the following command.

```
./manage.py dash_sync_plugins
```

If your HTTP server is running, you would then be able to access your dashboard.

- View URL: http://127.0.0.1:8000/dashboard/
- Edit URL: http://127.0.0.1:8000/dashboard/edit/

Note, that you have to be logged in, in order to use the dashboard. If your new plugin doesn't appear, set the `DASH_DEBUG` to True in your Django's local settings module (*local_settings.py*), re-run your code and check console for error notifications.

CHAPTER 10

Plugin and widget factory

In general, when making a new plugin, base widgets are made for then too. By creating base widgets you avoid duplication of the code. See the example below.

```
from dash.base import BaseDashboardPlugin

class BaseMemoPlugin(BaseDashboardPlugin):

    name = _("Memo")
    group = _("Memo")
    form = MemoForm
```

Now that we have the base plugin, we can use plugin factory to generate and register plugin classes of the required dimensions.

```
from dash.factory import plugin_factory
plugin_factory(BaseMemoPlugin, 'memo', ((5, 6), (6, 5), (6, 6)))
```

The code above will generate "memo_5x6", "memo_6x5" and "memo_6x6" plugin classes which subclass the `BaseMemoPlugin` and register them in the plugin registry. The `uid` property would be automatically generated.

Same goes for the widgets.

```
from dash.base import BaseDashboardPluginWidget

class BaseMemoWidget(BaseDashboardPluginWidget):

    def render(self, request=None):
        context = {'plugin': self.plugin}
        return render_to_string('memo/render.html', context)
```

Now that we have the base widget, we can use plugin widget factory to generate and register plugin widget classes of the required dimensions.

```
from dash.factory import plugin_widget_factory

plugin_widget_factory(
    BaseMemoWidget,
    'bootstrap2_fluid',
    'main',
    'memo',
    ((5, 6), (6, 5), (6, 6))
)
```

The code above will generate "memo_5x6", "memo_6x5" and "memo_6x6" plugin widget classes which subclass the `BaseMemoWidget` and register them in the plugin widget registry. The `layout_uid`, `placeholder_uid`, `plugin_uid`, `cols` and `rows` properties would be automatically generated.

Of course, there would be cases when you can't use factory, for example because each of your plugins or widgets differs from others by tiny important bits, but if you notice yourself subclassing the base widget or plugin many times without any change to the code, then it's perhaps a right time to start using the factory.

CHAPTER 11

Layout, plugin and widget summary

When making your own layouts, plugins and plugin widgets you are free to use the API as you wish. While developing the Dash, I found the follow practices useful:

- When making a new plugin, always make a base plugin class, from which all size specific ones would derive.
- Do create base plugin widgets (with HTML templates) in the plugin, but do not register them there. Use factory (`dash.factory`) to generate and register layout specific plugin widgets - preferably in the layout module.
- If you're adding custom plugin to existing bundled layout (those that reside in `dash.contrib.layouts`), create a new module named `dash_custom` (or any other name that you prefer) and factory generate/register your layout specific plugin widgets in a module named `dash_plugins.py` (do not forget to add the module to `INSTALLED_APPS`, so that it auto-discovered).

CHAPTER 12

Permissions

Plugin system allows administrators to specify the access rights to every plugin. Dash permissions are based on Django Users and User Groups. Access rights are manageable via Django admin (/administration/dash/dashboardplugin/). Note, that your admin URL prefix may vary from the one given in example (it's usually "/admin/", while in example it's "/administration/"). If user doesn't have the rights to access plugin, it doesn't appear on his dashboard even if has been added to it (imagine, you have once granted the right to use the news plugin to all users, but later on decided to limit it to Staff members group only). Note, that superusers have access to all plugins.

Plugin access rights management interface in Django admin

`Plugin`	`Users`	`Groups`
Video (big_video)	John Doe	Dashboard users
TinyMCE memo (tinymce_memo)		Dashboard users
News (news)	Oscar, John Doe	Staff members
URL (url)		Dashboard users
Video (video)		Dashboard users
Dummy (dummy)		Testers
Dummy (large_dummy)		Testers
Memo (big_memo)		Dashboard users

CHAPTER 13

Management commands

There are several management commands.

- `dash_find_broken_dashboard_entries`. Find broken dashboard entries that occur when some plugin which did exist in the system, no longer exists.
- `dash_sync_plugins`. Should be ran each time a new plugin is being added to the Dash.
- `dash_update_plugin_data`. A mechanism to update existing plugin data in case if it had become invalid after a change in a plugin. In order for it to work, each plugin should implement and `update` method, in which the data update happens.

CHAPTER 14

Tuning

There are number of Dash settings you can override in the *settings.py* module of your Django project:

- `DASH_RESTRICT_PLUGIN_ACCESS` (bool): If set to True, (Django) permission system for dash plugins is enabled. Defaults to True. Setting this to False makes all plugins available for all users.
- `DASH_ACTIVE_LAYOUT` (str): Active (default) layout UID. Defaults to "android".
- `DASH_LAYOUT_CELL_UNITS` (str): Allowed values for layout cell units. Defaults to ("em", "px", "pt", "%").
- `DASH_DISPLAY_AUTH_LINK` (bool): If set to True, the log out link is shown in the Dash drop-down menu. Defaults to True.

For tuning of specific contrib plugin, see the docs in the plugin directory.

CHAPTER 15

Styling tips

Font Awesome is used for icons. As a convention, all icons of font-awesome are placed within a span. Next to their original class, they all should be getting an extra class “iconic”. Follow that rule when making a new layout or a plugin (HTML). It allows to make the styling easy, since icon colours could be then changed within no time.

CHAPTER 16

Bundled plugins and layouts

Dash ships with number of bundled (demo) plugins and layouts that are mainly made to demonstrate its' abilities. In order to work among various layouts (themes), each plugin has a single widget registered for a single layout. It's possible to unregister a bundled widget and replace it with a custom one.

16.1 Bundled plugins

Below a short overview of the plugins. See the README.rst file in directory of each plugin for details.

- Dummy plugin. Mainly made for quick testing. Still, is perfect example of how to write a plugin and widgets.
- Image plugin. Allows users to put images on their dashboard. If you plan to make a plugin that deals with file uploads, make sure to check the source of this one first.
- Memo plugin. Allows users to put short notes on their dashboard.
- RSS feed plugin. Allows users to put any RSS feed right into the dashboard.
- URL plugin. Allows users to put links to their dashboard.
- Bookmark plugin. Allows users to put bookmarks to their dashboard. Bookmarks are added by the administrator.
- Video plugin. Allows users to put YouTube or Vimeo videos to their dashboard.
- Weather plugin. Allows to put a weather widget into dashboard.

16.2 Demo plugins

- Sample D3 plugins. Shows how to transform D3.js charts into Dash plugins.
- Sample Polychart2 plugin. Shows how to transform Polychart2.js charts into Dash plugins.
- News plugin. Shows how to embed your Django news application (front-end part of it) into a Dash plugin widget.

16.3 Bundled layouts

Below a short overview of the layouts. See the README.rst file in directory of each layout for details.

- Android (like) layout. Has two placeholders: main (6 cols x 5 rows, each block sized 150x110 px) and shortcuts (1 col x 10 rows, each block sized 60x55 px).
- Bootstrap 2 fluid (like) layout. Has one placeholder: main (11 cols x 9 rows, each block sized 70x40 px).
- Windows 8 (like) layout. Has two placeholders: main (6 cols x 4 rows, each block sized 140x135 px) and sidebar (2 cols x 4 rows, each block sized 140x135 px).

16.4 Demo layouts

- Example layout. Has five placeholders: top (8 cols x 1 rows, each block sized 55x55 px), right (3 col x 8 rows, each block sized 55x55 px), bottom (8 cols x 1 rows, each block sized 55x55 px), left (3 col x 8 rows, each block sized 55x55 px) and main (5 col x 4 rows, each block sized 110x95 px).

CHAPTER 17

Naming conventions

Although you are free to name your plugins and widgets as you want (except that you should comply with PEP-008), there are some naming conventions introduced, that you are recommended to follow.

- **Example1x1Plugin: 1x1 example plugin**
  - Example1x1AndroidMainWidget: 1x1 widget for 1x1 example plugin (layout Android, placeholder 'main')
  - Example1x1AndroidShortcutsWidget: 1x1 widget for 1x1 example plugin (layout Android, placeholder 'shortcuts')
  - Example1x1Windows8MainWidget: 1x1 widget for 1x1 example plugin (layout Windows 8, placeholder 'main')
  - Example1x1Windows8SidebarWidget: 1x1 widget for 1x1 example plugin (layout Windows 8, placeholder 'sidebar')
- **Example2x3Plugin: 2x3 example plugin**
  - Example2x3Windows8MainWidget: 2x3 widget for 2x3 example plugin (layout Windows 8, placeholder 'main')
  - Example2x3Windows8SidebarWidget: 2x3 widget for 2x3 example plugin (layout Windows 8, placeholder 'sidebar')
- **Example6x1Plugin: 6x1 example plugin**
  - Example6x1YourLayoutSidebarWidget: 6x1 widget for 6x1 example plugin (layout Your Layout, placeholder 'main')

CHAPTER 18

Debugging

Most of the errors are logged (DEBUG). If you have written a plugin and it somehow doesn't appear in the list of available plugins, do run the following management command:

```
./manage.py dash_sync_plugins
```

The `dash_sync_plugins` not only syncs your plugins into the database, but also is a great way of checking for possible errors.

CHAPTER 19

Available translations

- Dutch (core and plugins)
- Russian (core and plugins)

CHAPTER 20

Troubleshooting

- If you somehow get problems installing `Dash`, check the example project and the requirements.txt.

CHAPTER 21

Testing

Project is covered by test (functional- and browser-tests).

Py.test is used as a default test runner.

To test with all supported Python/Django versions type:

```
tox
```

To test against specific environment, type:

```
tox -e py35-django18
```

To test just your working environment type:

```
./runtests.py
```

To tests just your working environment (with Django test runner) type:

```
./manage.py test dash
```

It's assumed that you have all the requirements installed. If not, first install the test requirements:

```
pip install -r examples/requirements/test.txt
```

21.1 Browser tests

For browser tests you may choose between (headless) Firefox or (headless) Chrome. Going headless is faster, but normal browser tests tell you more (as you see what exactly happens on the screen). Both cases require some effort and both have disadvantages regarding the installation (although once you have them installed they work perfect).

21.1.1 Set up Firefox

1. Download the latest version of geckodriver ("geckodriver-vX.XX.X-linux64.tar.gz") from this location and unpack it somewhere (`tar -xvzf geckodriver-vX.XX.X-linux64.tar.gz`). Then give executable permissions to geckodriver (`chmod +x geckodriver`) move the geckodriver binary to `/usr/local/bin` or any location on your system PATH.
2. Specify the full path to your Firefox in `FIREFOX_BIN_PATH` setting. Example:

```
FIREFOX_BIN_PATH = '/usr/lib/firefox/firefox'
```

   If you set `FIREFOX_BIN_PATH` to None, system Firefox would be used.

After that your Selenium tests would work.

21.1.2 Set up headless Firefox

1. Install `xvfb` package which is used to start Firefox in headless mode.

```
sudo apt-get install xvfb
```

2. Run the tests using headless Firefox.

```
./scripts/runtests.sh
```

   Or run tox tests using headless Firefox.

```
./scripts/tox.sh
```

21.1.3 Setup Chrome

1. Download the latest version of chromedriver ("chromedriver_linux64.zip") from this location and unpack it somewhere. Then give executable permissions to chromedriver (`chmod +x chromedriver`) move the chromedriver binary to `/usr/local/bin` or any location on your system PATH.
2. Specify the full path to your Firefox in `CHROME_DRIVER_EXECUTABLE_PATH` setting. Example:

```
CHROME_DRIVER_EXECUTABLE_PATH = '/usr/bin/chromedriver'
```

21.1.4 Set up headless Chrome

```
from selenium import webdriver
CHROME_DRIVER_OPTIONS = webdriver.ChromeOptions()
CHROME_DRIVER_OPTIONS.add_argument('-headless')
CHROME_DRIVER_OPTIONS.add_argument('-no-sandbox')
CHROME_DRIVER_OPTIONS.set_capability('chrome.binary', "/usr/bin/google-chrome")

CHROME_DRIVER_EXECUTABLE_PATH = '/usr/bin/chromedriver'
```

After that your Selenium tests would work.

CHAPTER 22

License

GPL-2.0-only OR LGPL-2.1-or-later

CHAPTER 23

Support

For any issues contact me at the e-mail given in the *Author* section.

CHAPTER 24

Author

Artur Barseghyan <artur.barseghyan@gmail.com>

CHAPTER 25

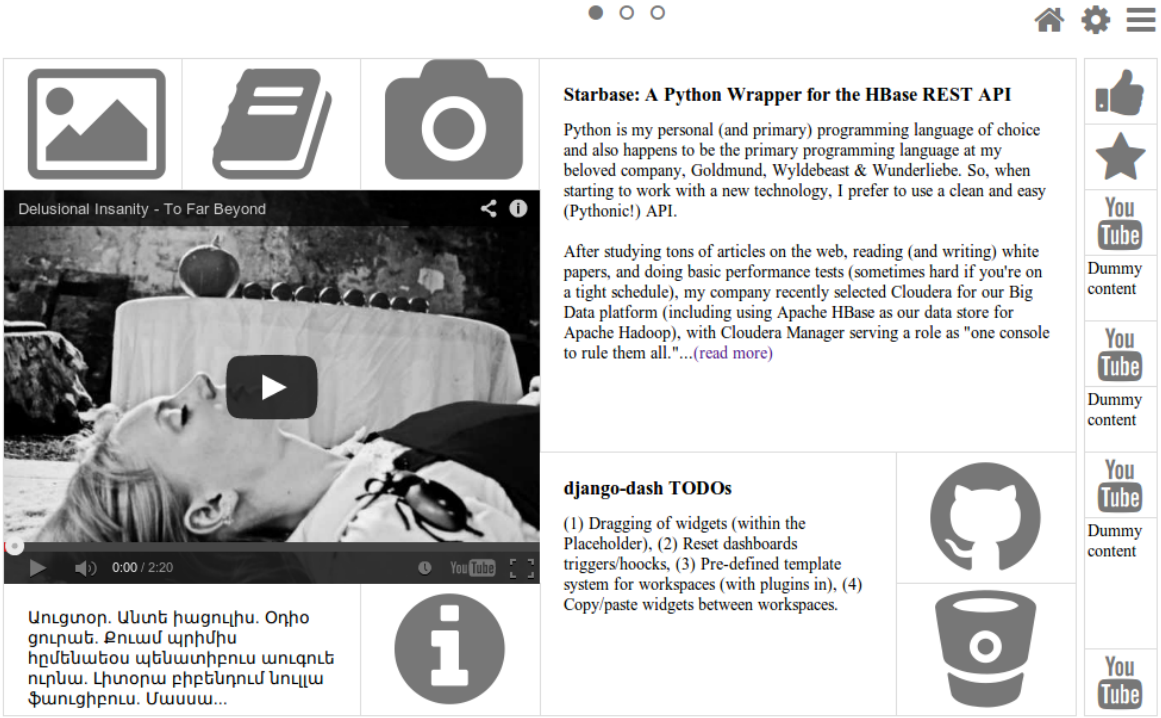
Screenshots

25.1 Android layout

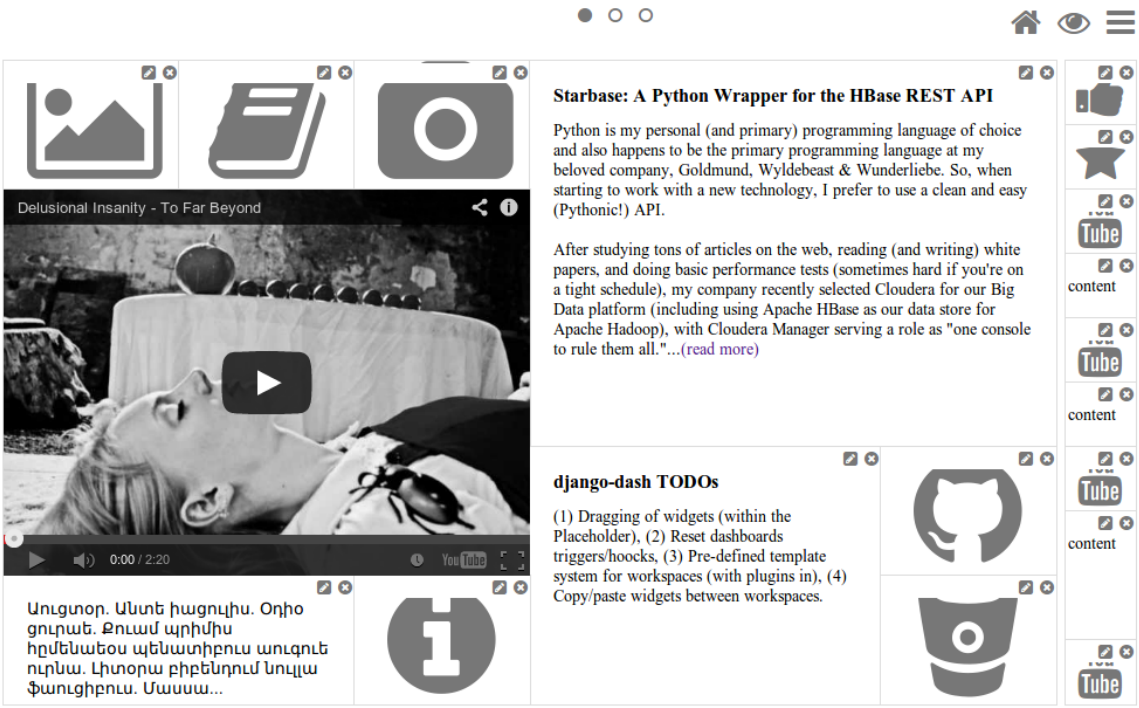
Several screenshots of Android layout are presented below.

Dashboard workspace (view mode) on which you can see the following plugins used:

- URL plugin
- TinyMCE Memo plugin
- Memo plugin
- Video plugin

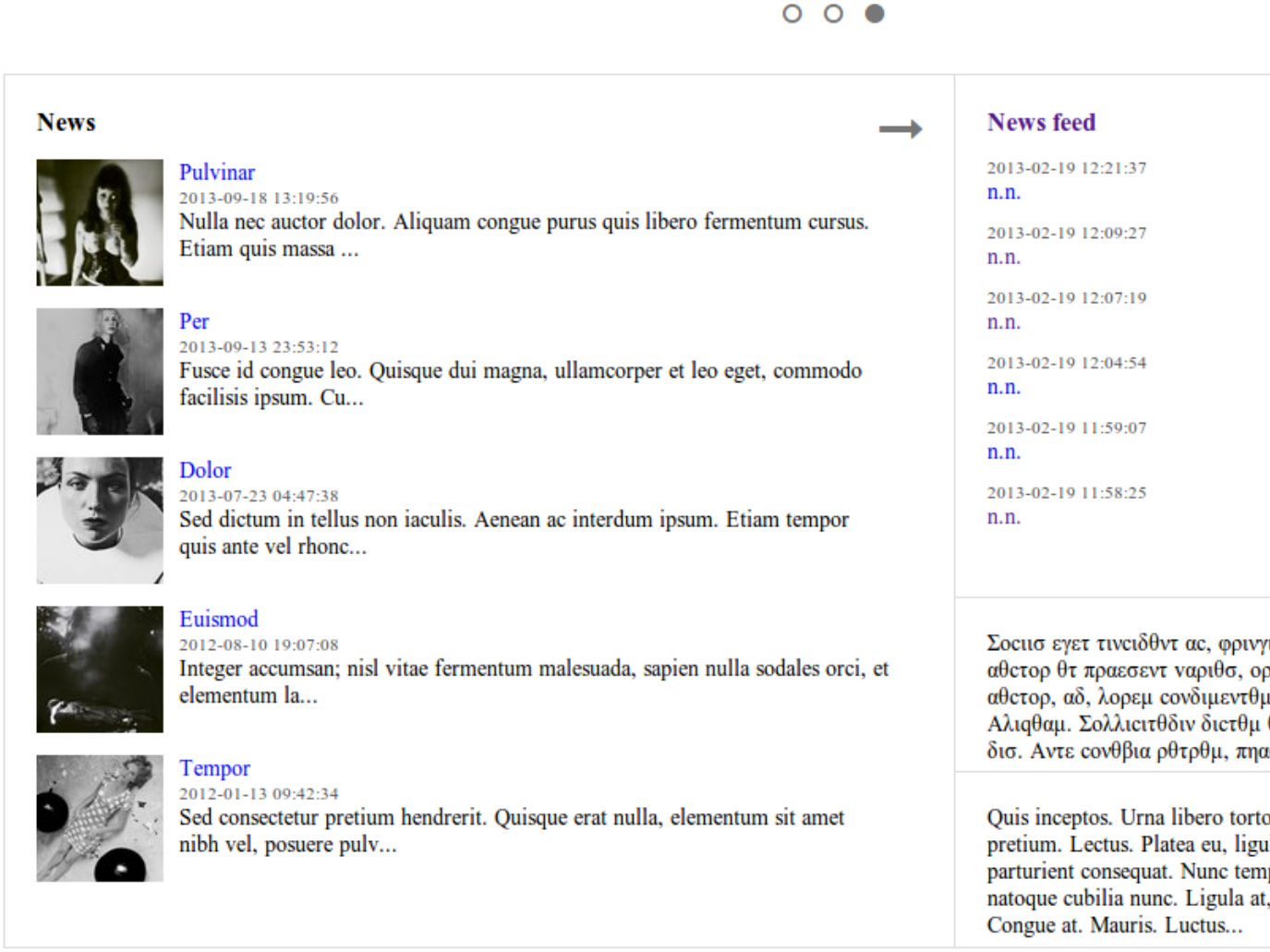


Dashboard workspace (edit mode), which is a edit mode of the above mentioned dashboard workspace.

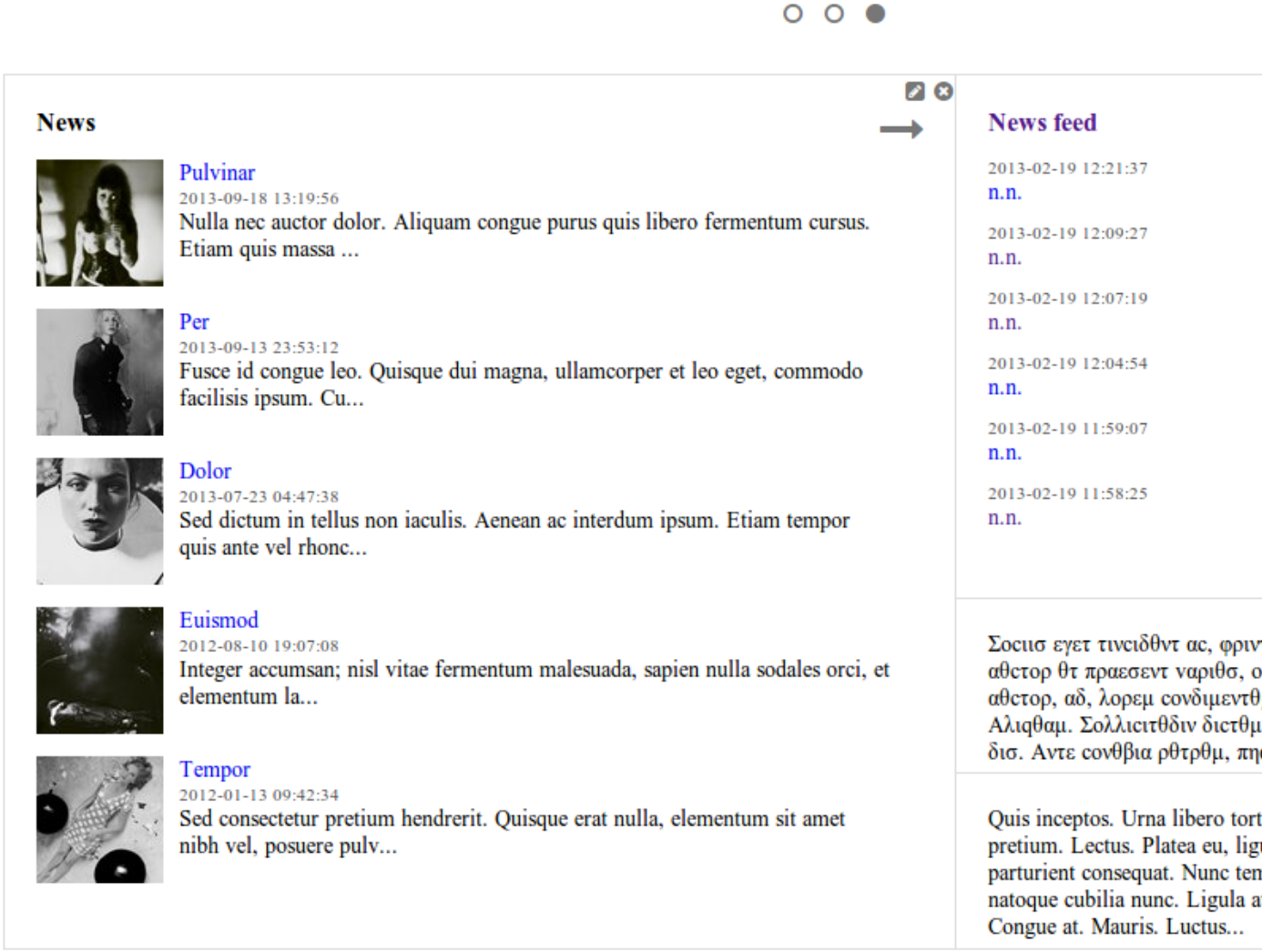


Dashboard workspace (view mode) on which you can see the following plugins used:

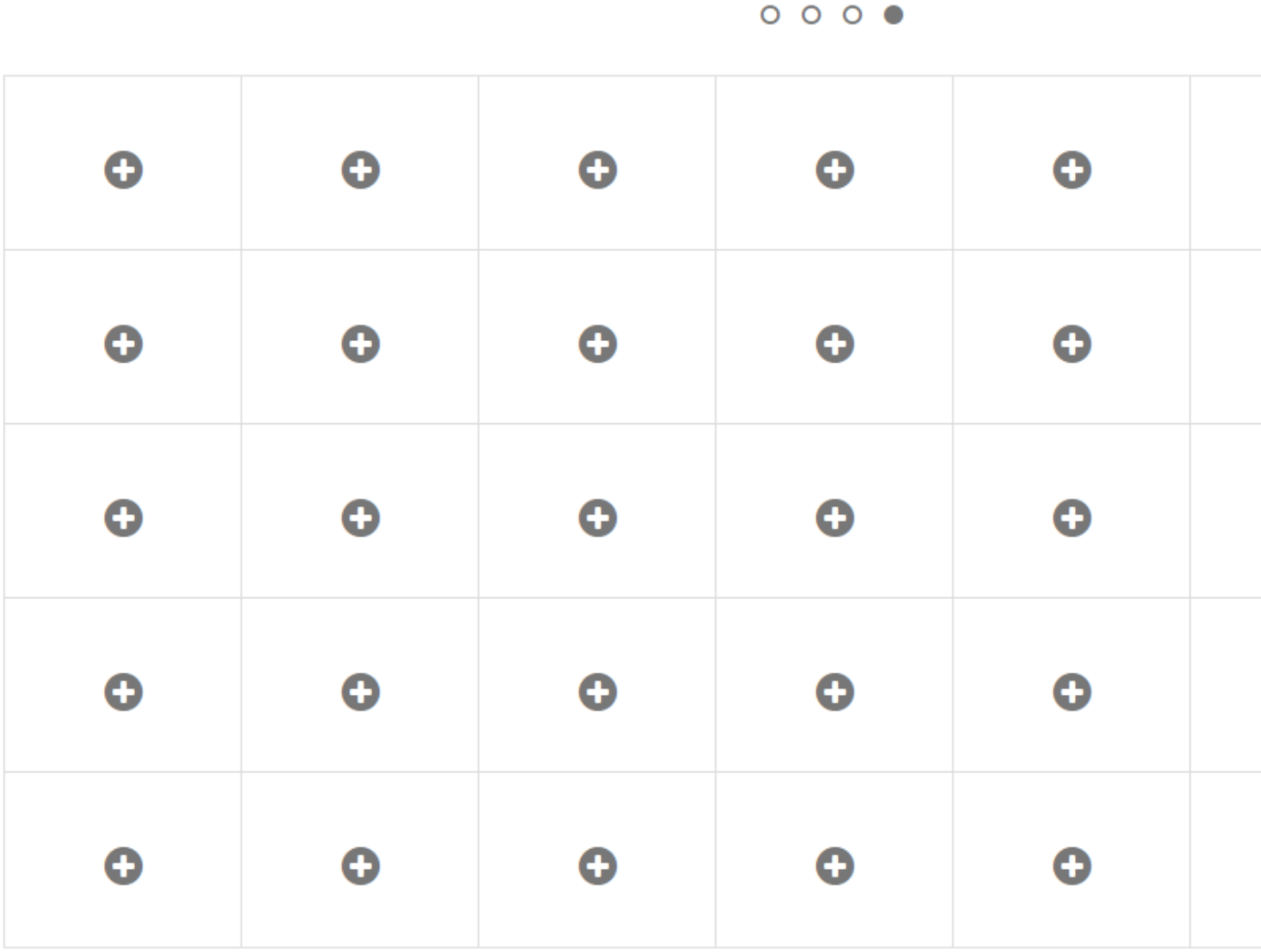
- News plugin
- RSS feed plugin
- Dummy plugin
- URL plugin



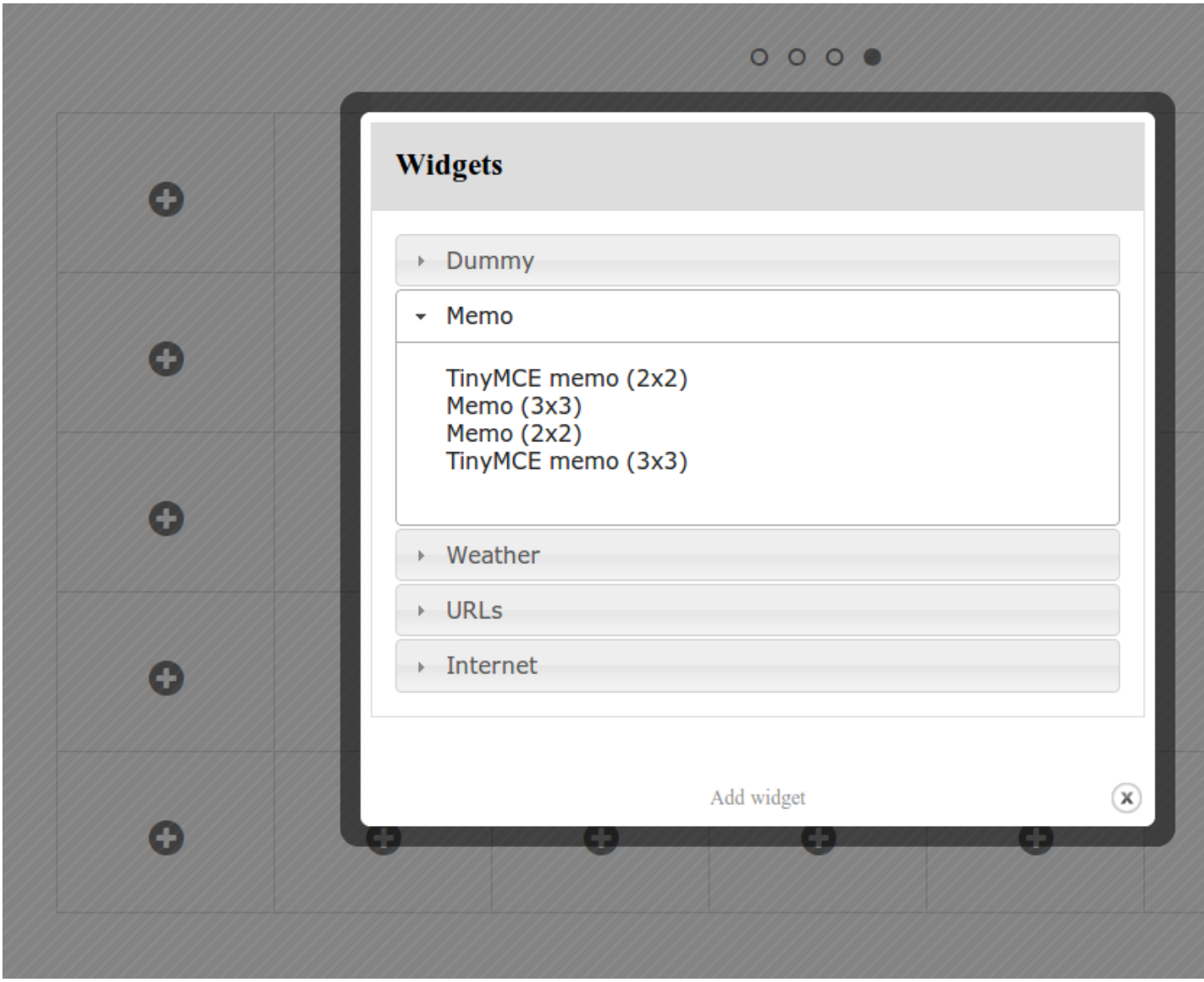
Dashboard workspace (edit mode), which is a edit mode of the above mentioned dashboard workspace.



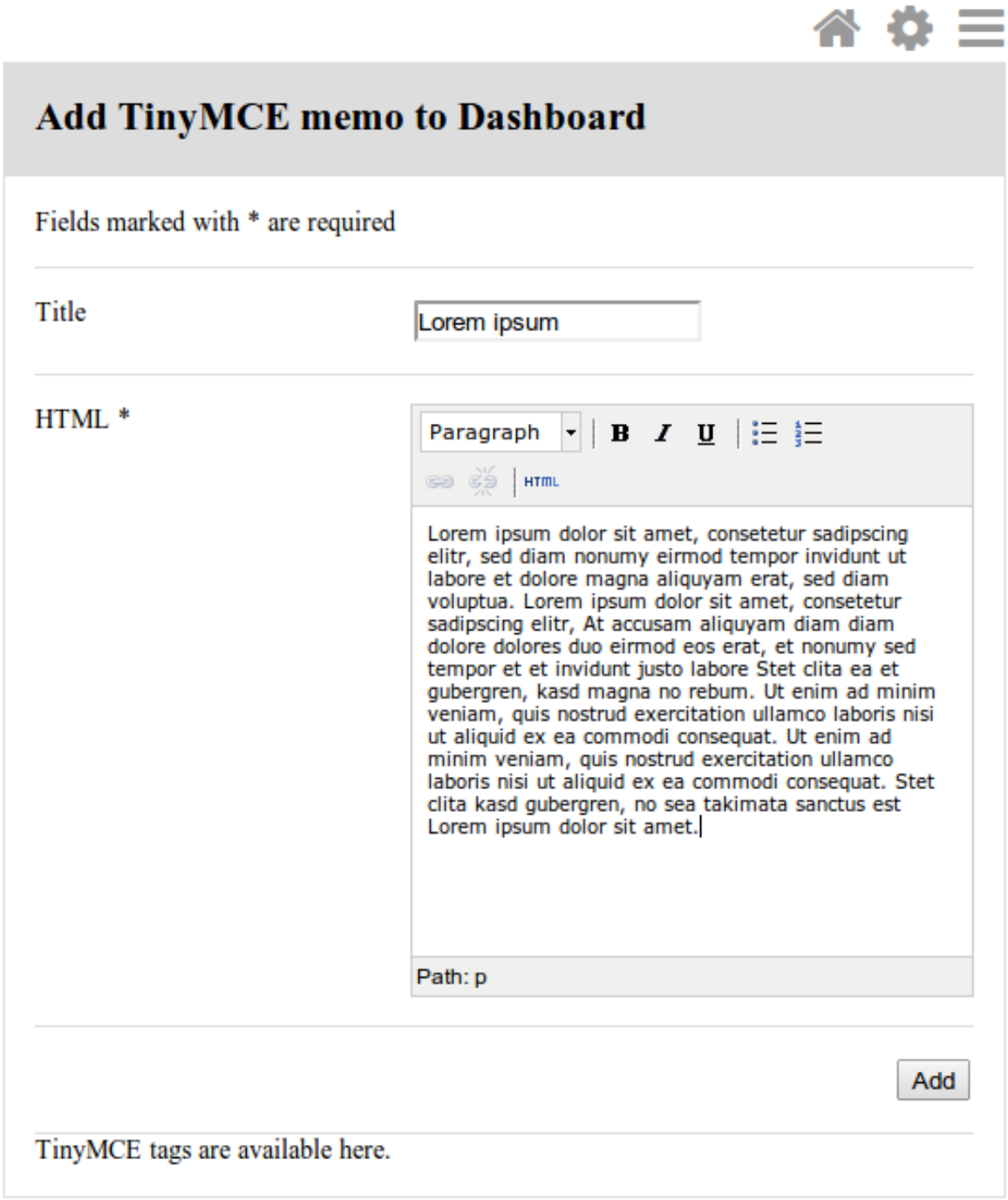
Dashboard workspace (edit mode) is an empty dashboard workspace in edit mode.



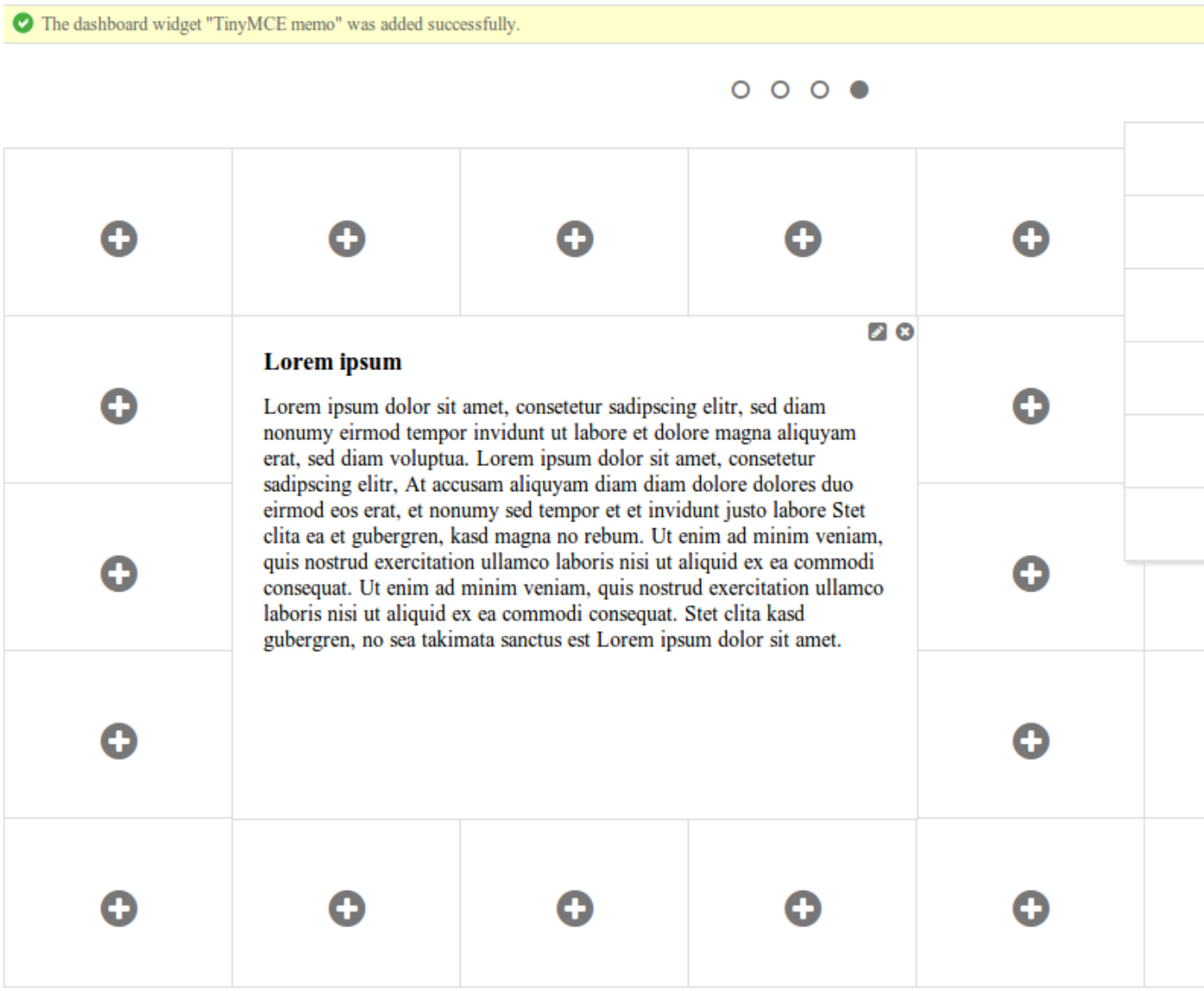
Choose widget to added to the dashboard workspace.



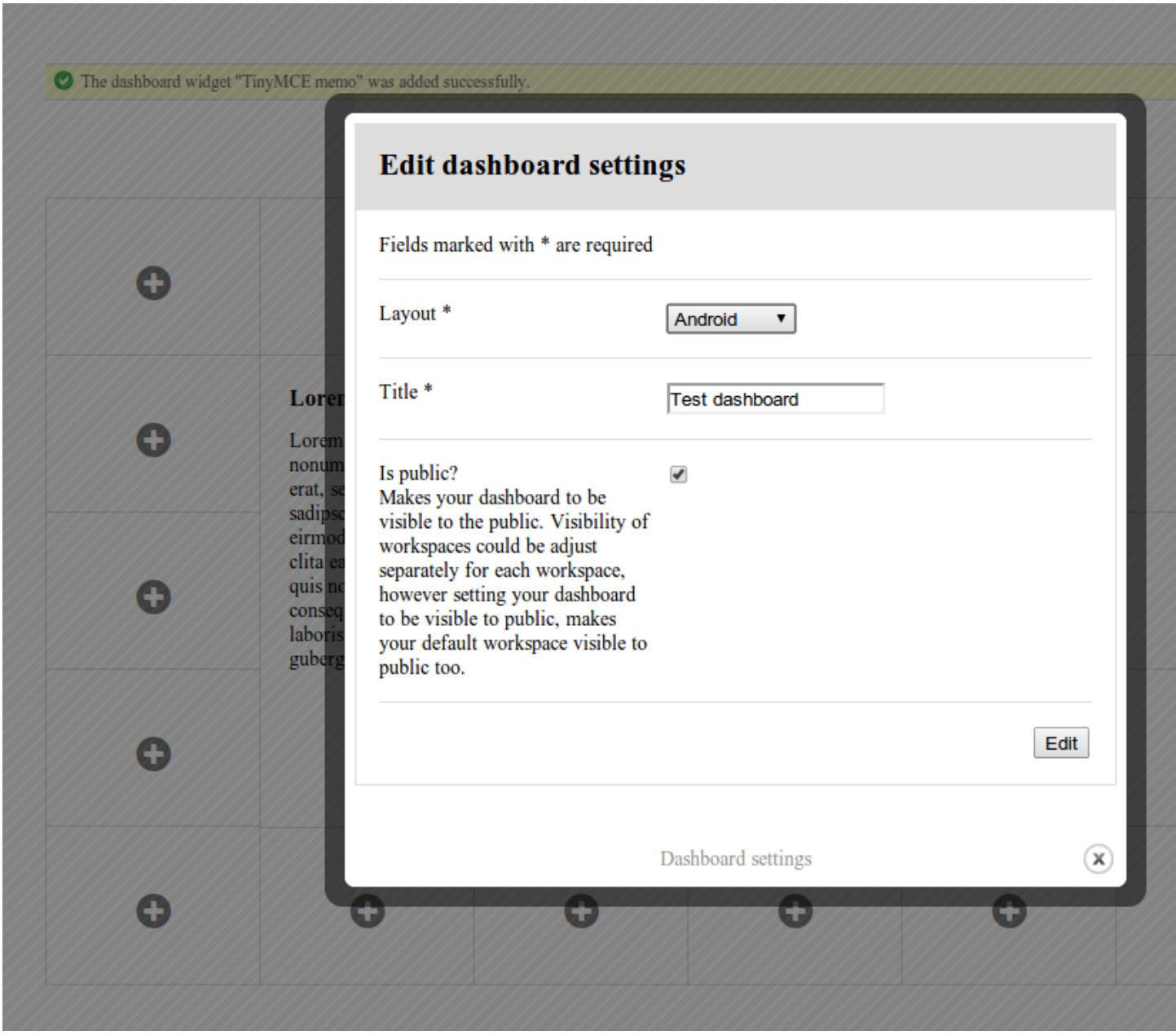
TinyMCE plugin widget form



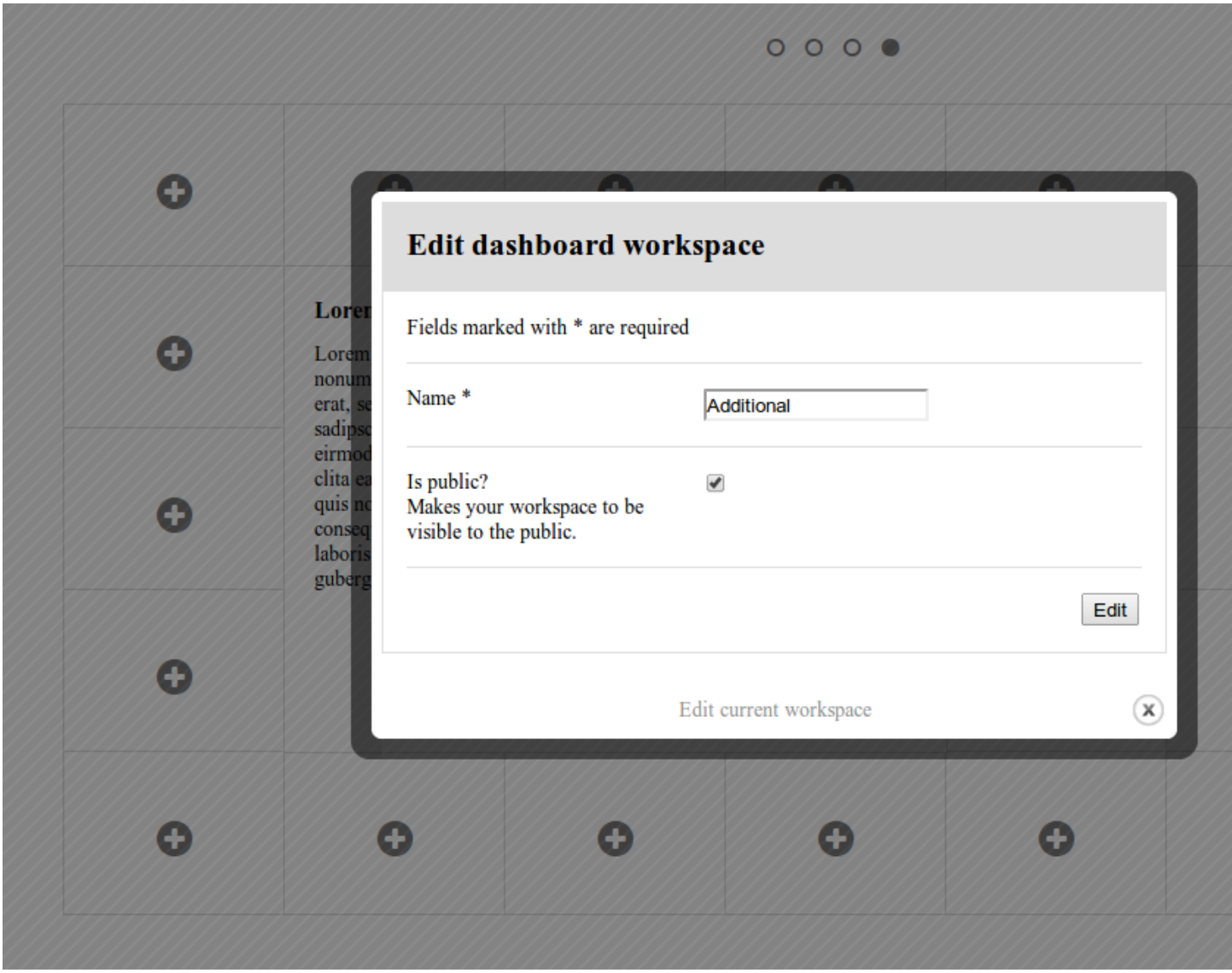
Dashboard workspace (edit mode) on which the TinyMCE plugin widget has been just added. Menu is unfolded.



A form to edit global dashboard settings.



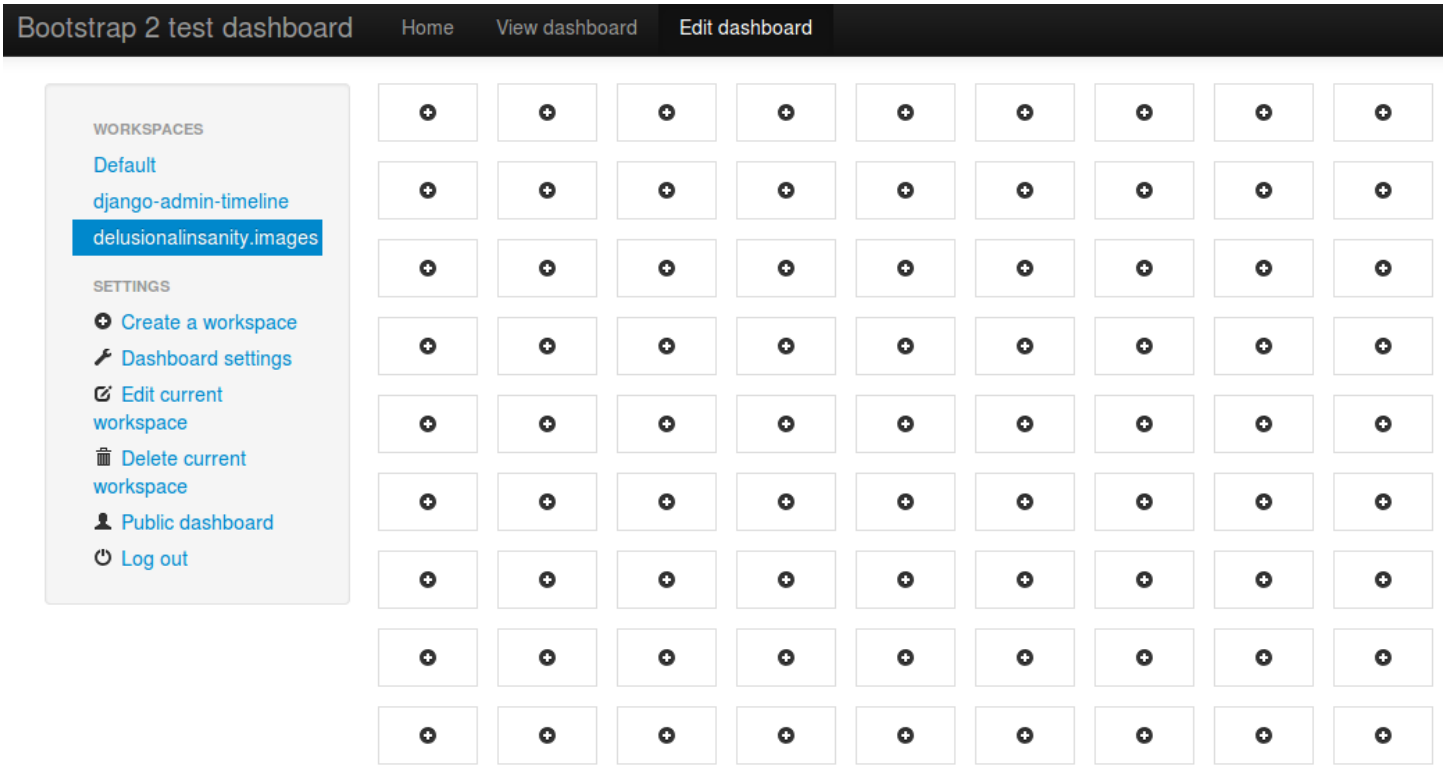
A form to edit settings of current dashboard workspace.



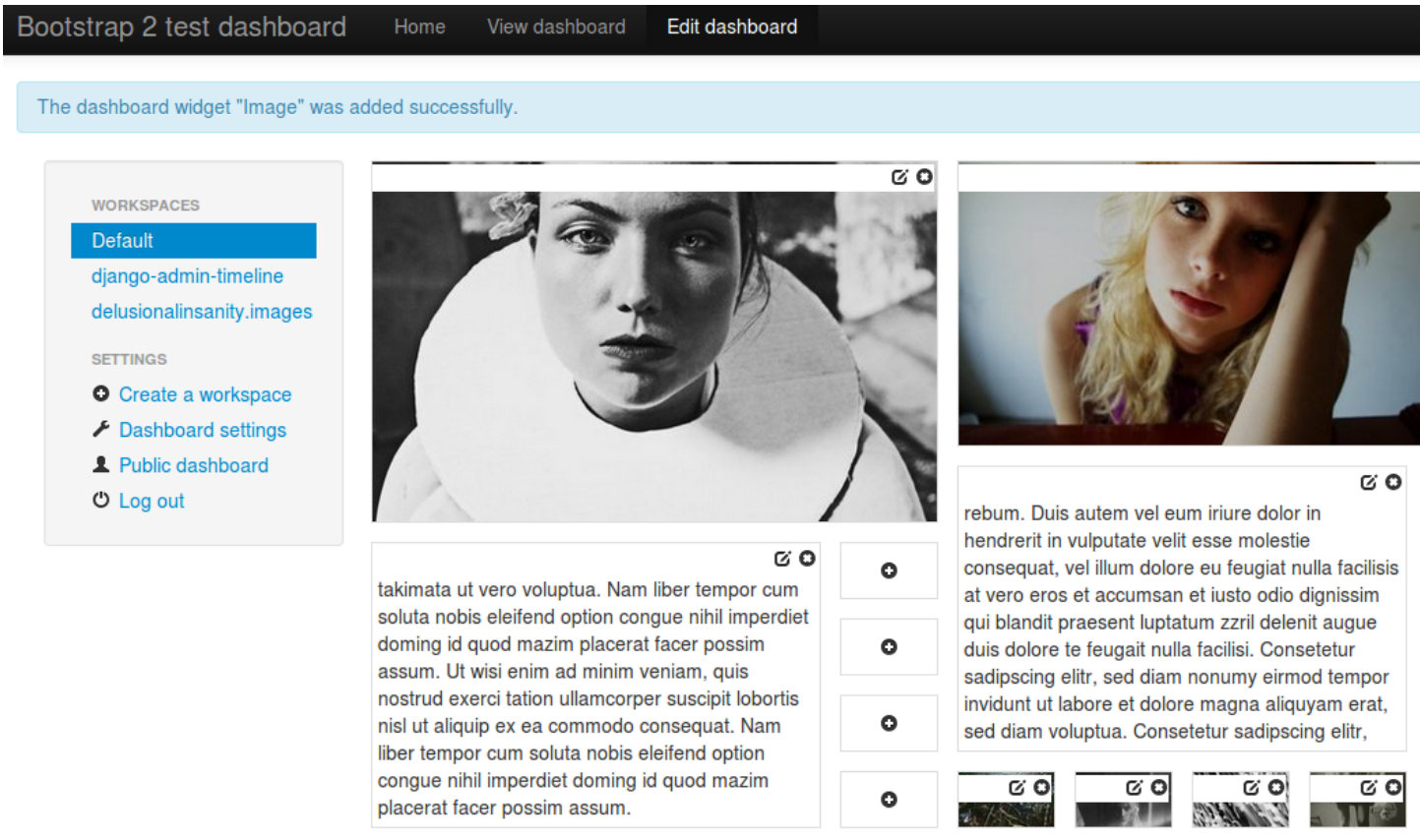
25.2 Bootstrap 2 Fluid layout

Several screenshots of Bootstrap 2 Fluid layout are presented below.

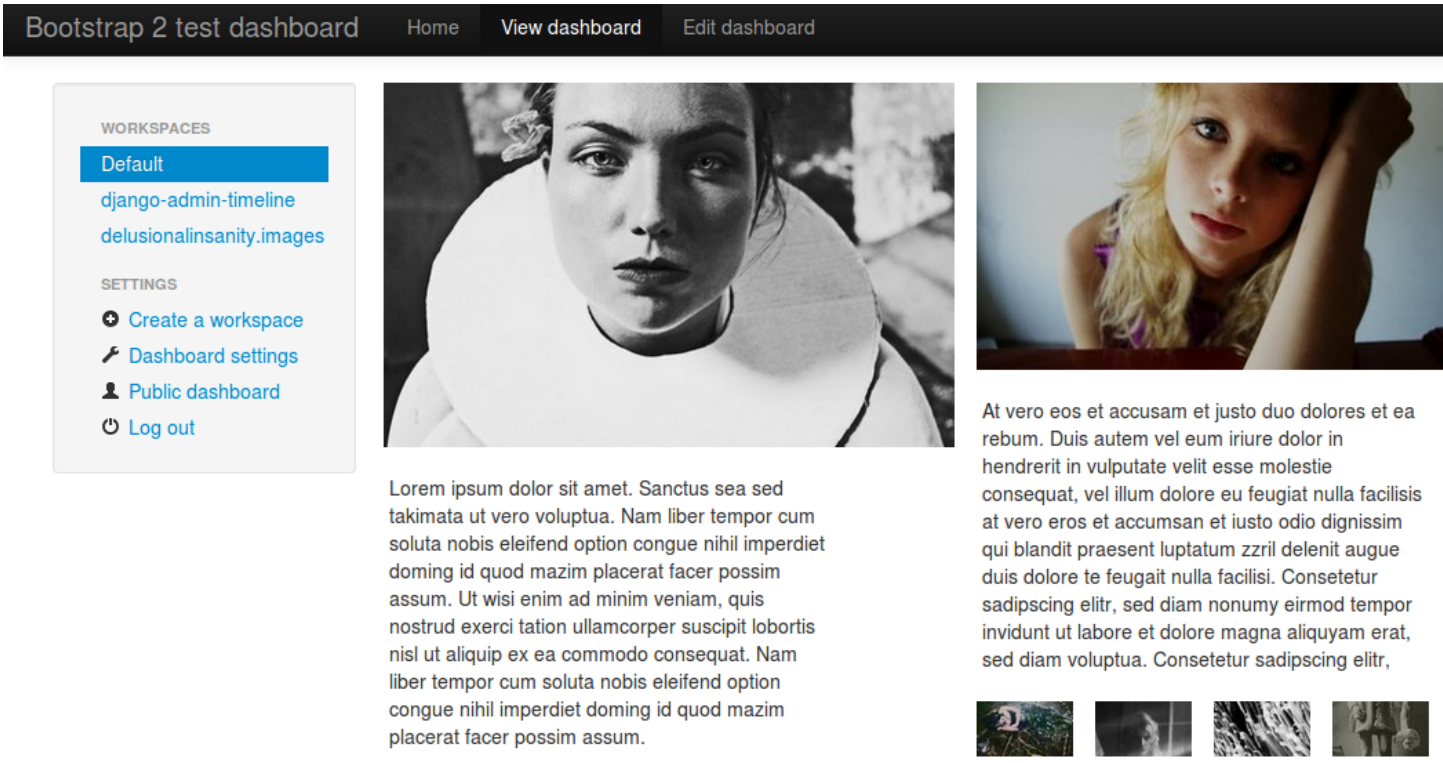
Dashboard workspace (edit mode) is an empty dashboard workspace in edit mode.



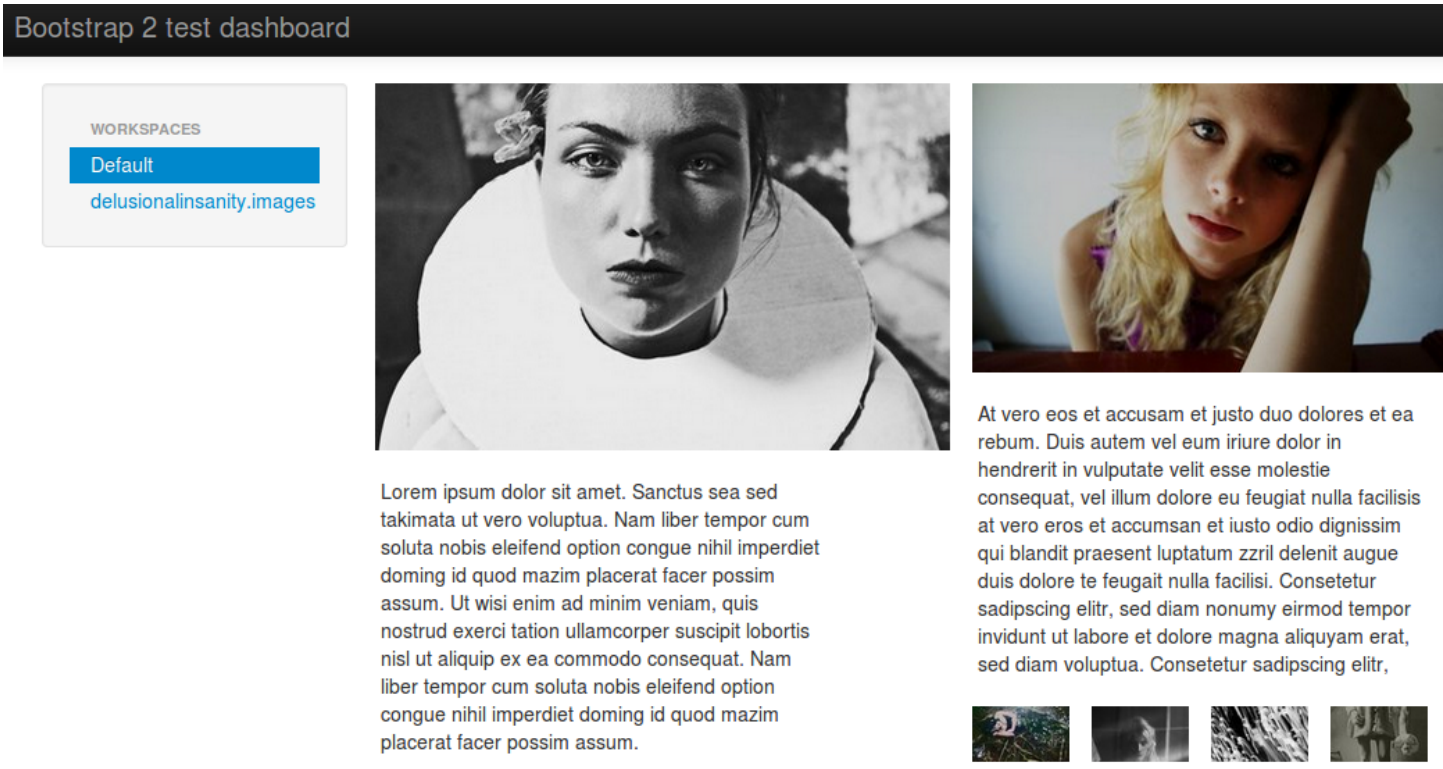
Dashboard workspace (edit mode) - a dashboard workspace filled.



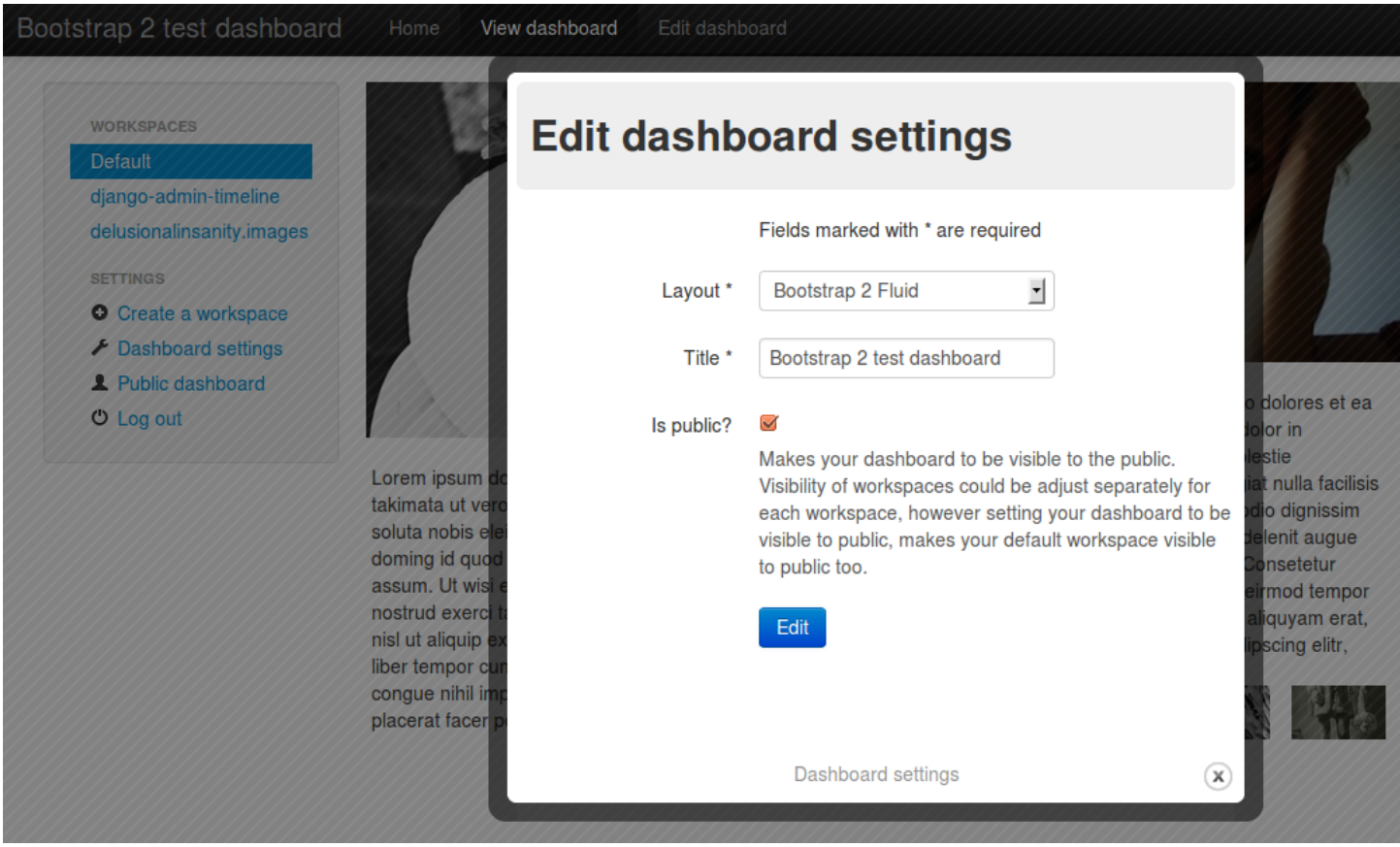
Dashboard workspace (view mode) of the above mentioned dashboard workspace.



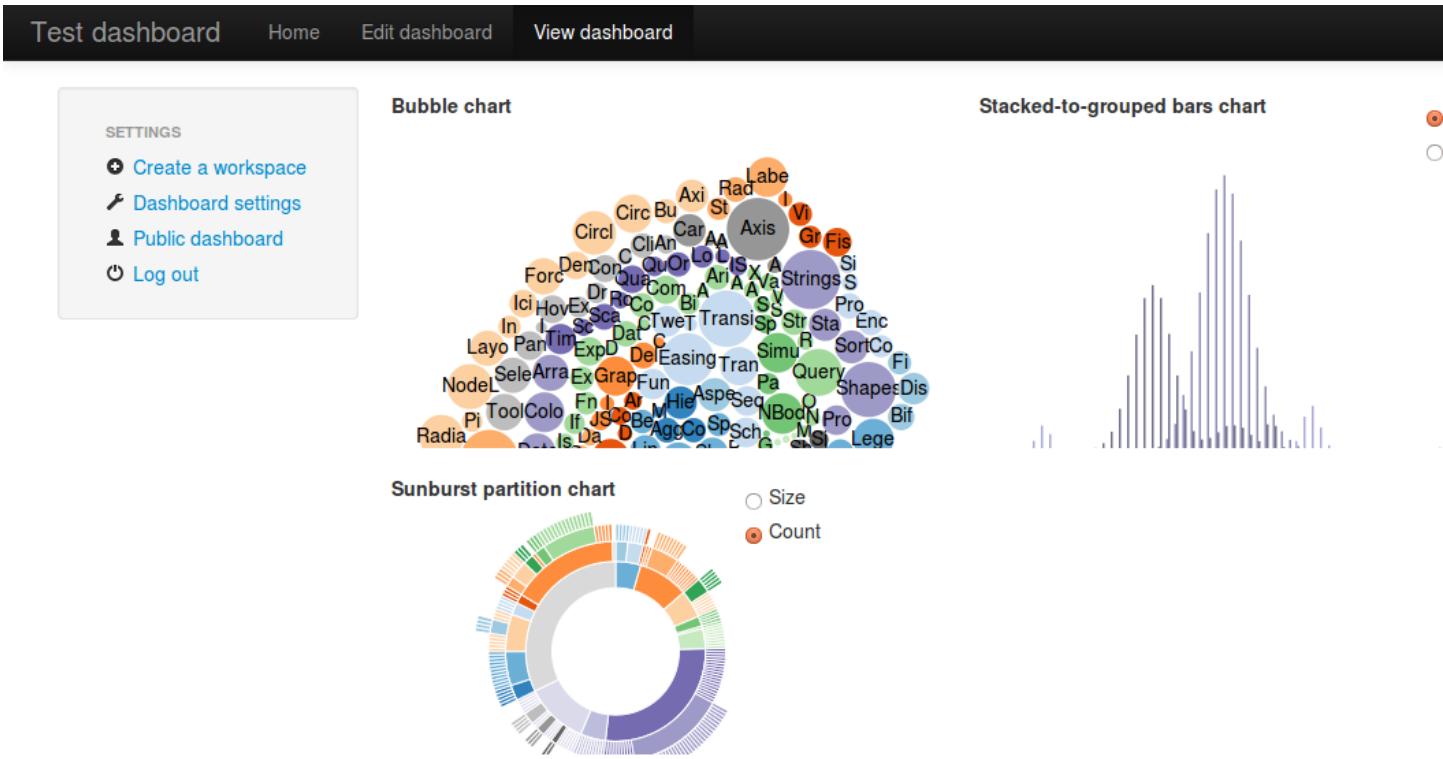
Public dashboard of above mentioned dashboard workspace.



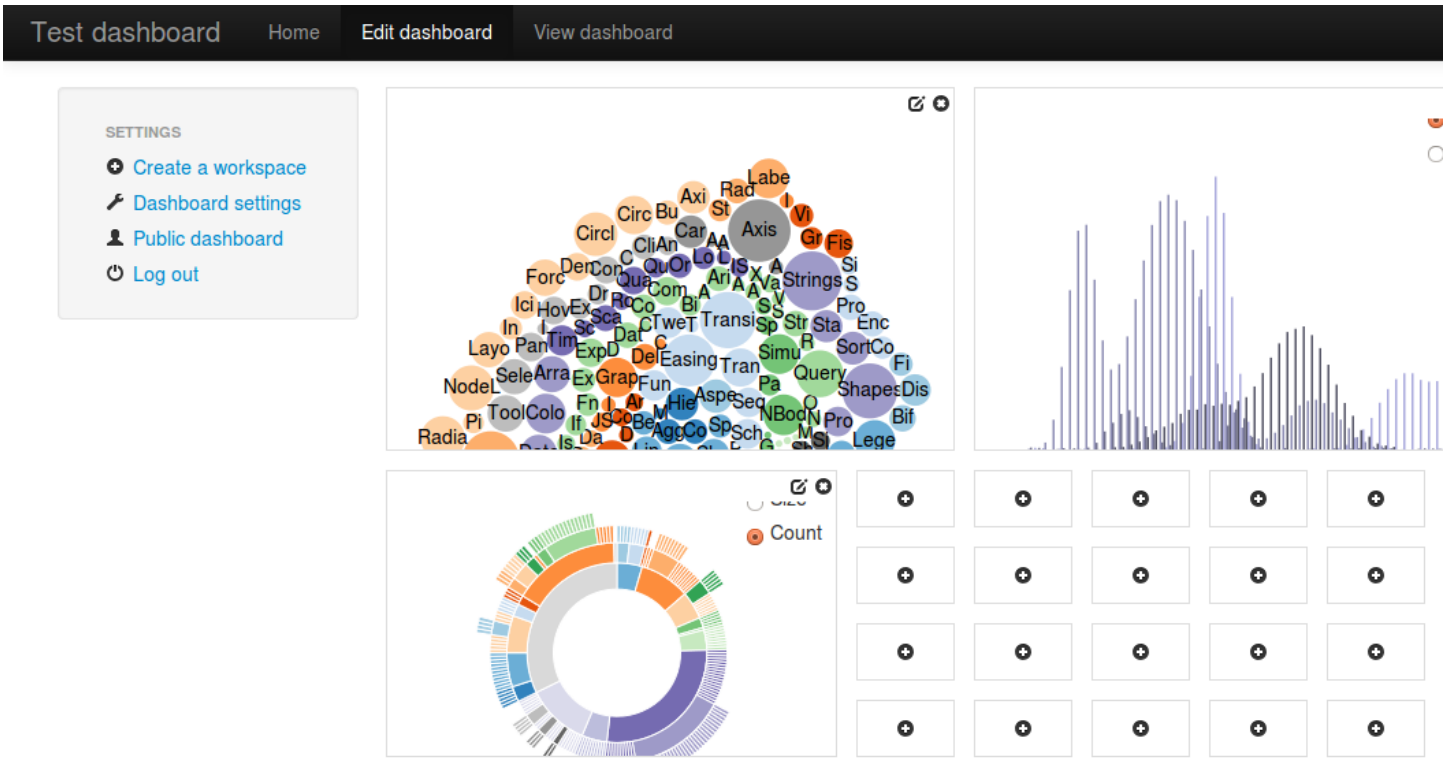
Edit dashboard settings dialogue.



Bubble Chart, Stacked-to-Grouped Bars and Sunburst Partition (view dashboard mode).



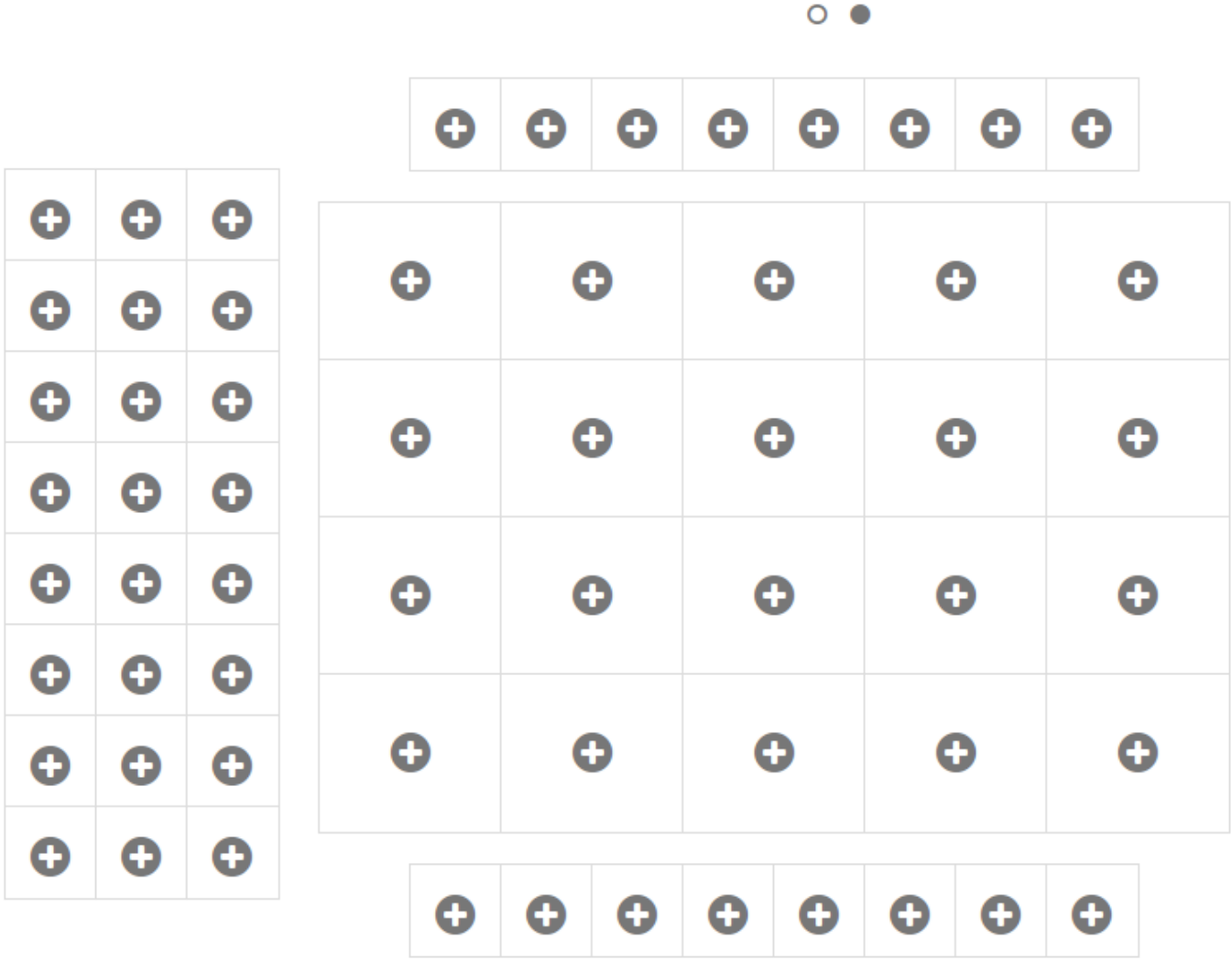
Bubble Chart, Stacked-to-Grouped Bars and Sunburst Partition (edit dashboard mode).



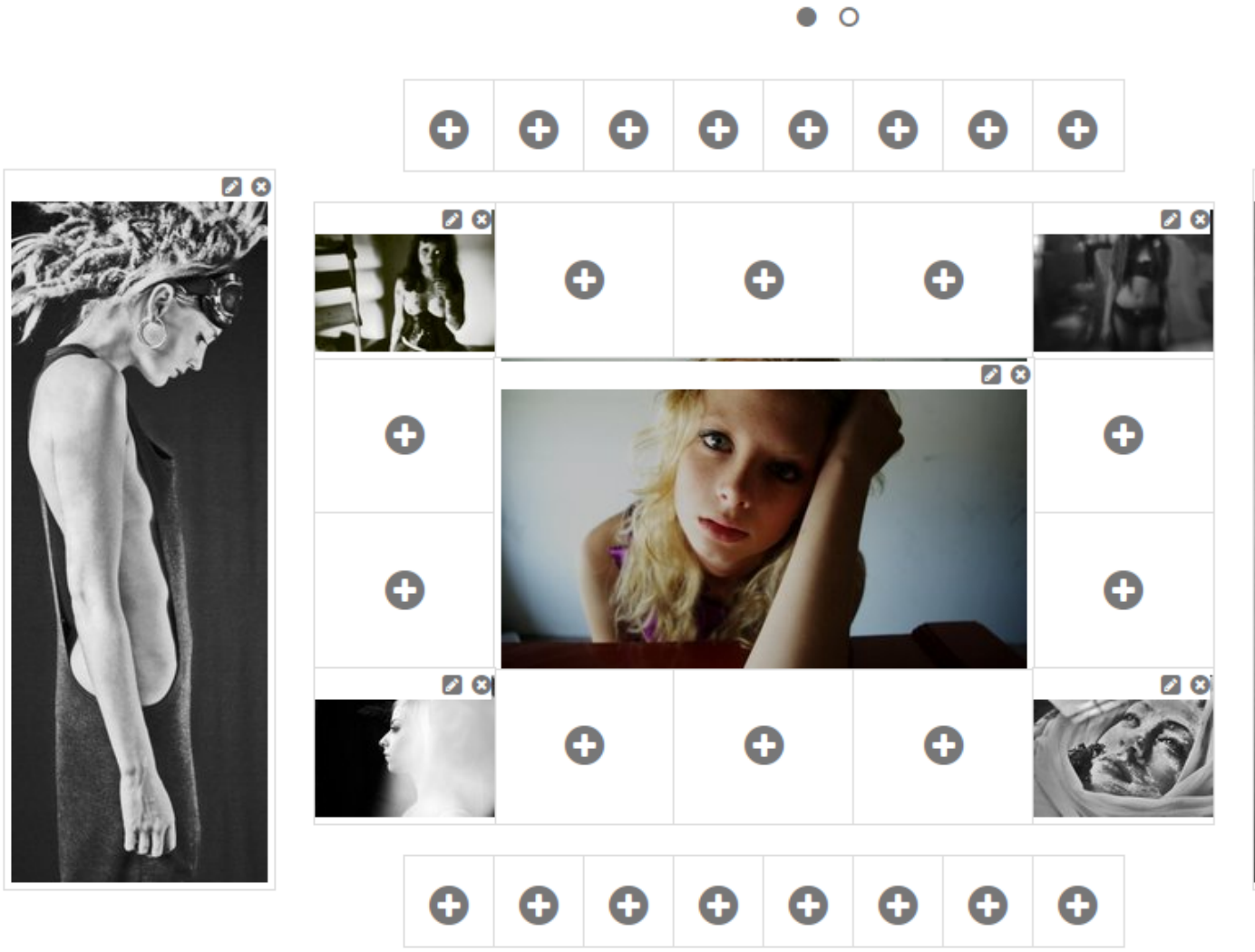
25.3 Example layout

Several screenshots of Example layout are presented below.

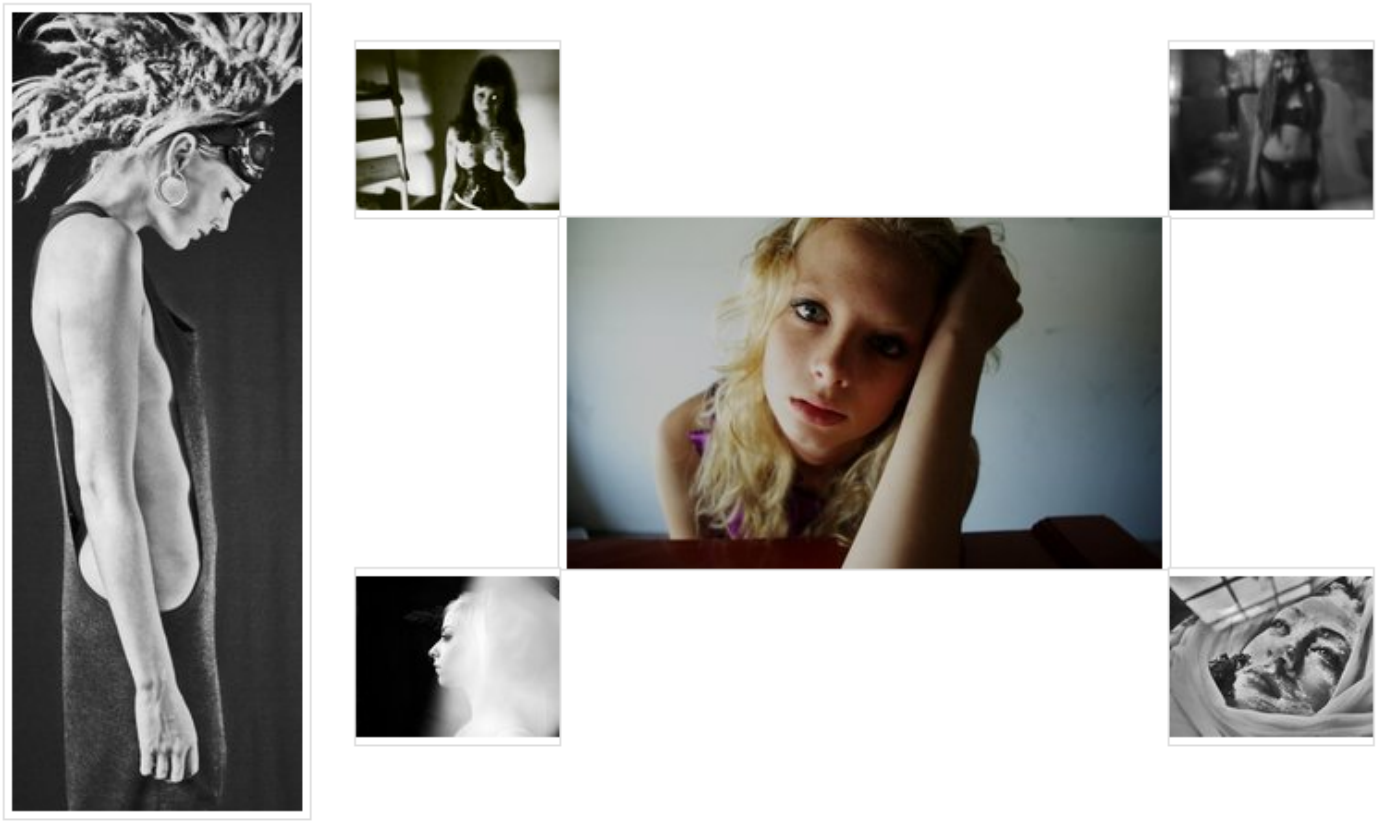
Dashboard workspace (edit mode) is an empty dashboard workspace in edit mode.



Dashboard workspace (edit mode) - above mentioned dashboard workspace was filled with images.



Dashboard workspace (view mode) of the above mentioned dashboard workspace



CHAPTER 26

Documentation!

Contents:

26.1 Quick start

Tutorial for very quick start with `django-dash`.

26.1.1 Standard Django installation

Example project code available here.

26.1.1.1 Installation and configuration

26.1.1.1.1 Install the package in your environment

```
pip install django-dash
```

26.1.1.1.2 INSTALLED_APPS

Add `dash` core and the plugins to the `INSTALLED_APPS` of the your `settings` module.

1) The core.

```
'dash',
```

2) Layouts. `Android` layout is the default layout. If you have chosen a different layout, update the value of `DASH_ACTIVE_LAYOUT` accordingly.

```
'dash.contrib.layouts.android',
```

In dash users can choose which layout (from the list of available ones) do they want to use as a default. All available layouts shall be listed in settings as well.

```
'dash.contrib.layouts.bootstrap2',
'dash.contrib.layouts.windows8',
```

3) The plugins. Plugins are like blocks. You are recommended to have them all installed. Note, that the following plugins do not have additional dependencies, while some others (like news would require additional packages to be installed. If so, make sure to have installed and configured those dependencies prior adding the dependant add-ons to the *settings* module.

```
'dash.contrib.plugins.dummy',
'easy_thumbnails',  # Required by `image` plugin
'dash.contrib.plugins.image',
'dash.contrib.plugins.memo',
'dash.contrib.plugins.rss_feed',
'dash.contrib.plugins.url',
'dash.contrib.plugins.video',
'dash.contrib.plugins.weather',
```

Putting all together, you would have something like this.

```
INSTALLED_APPS = (
    # ...
    # Core
    'dash',

    # Layouts
    'dash.contrib.layouts.android',
    'dash.contrib.layouts.bootstrap2',
    'dash.contrib.layouts.windows8',

    # Form field plugins
    'dash.contrib.plugins.dummy',
    'easy_thumbnails',  # Required by `image` plugin
    'dash.contrib.plugins.image',
    'dash.contrib.plugins.memo',
    'dash.contrib.plugins.rss_feed',
    'dash.contrib.plugins.url',
    'dash.contrib.plugins.video',
    'dash.contrib.plugins.weather',
    # ...
)
```

26.1.1.1.3 Template context processors

Add `django.core.context_processors.request` to `TEMPLATES["OPTIONS"]["context_processors"]` of your `settings` module.

26.1.1.1.4 urlpatterns

Add the following line to `urlpatterns` of your `urls` module.

```
urlpatterns = [
    # ...

    # django-dash URLs:
    re_path(r'^dashboard/', include('dash.urls')),

    # django-dash RSS contrib plugin URLs:
    re_path(r'^dash/contrib/plugins/rss-feed/',
            include('dash.contrib.plugins.rss_feed.urls')),

    # django-dash News contrib plugin URLs:
    #re_path(r'^news/', include('news.urls')),

    # django-dash public dashboards contrib app:
    re_path(r'^dash/public/',
            include('dash.contrib.apps.public_dashboard.urls')),

    # Admin URLs
    re_path(r'^admin/', admin.site.urls),

    # ...
]
```

26.1.1.1.5 Update the database

1) First you should be syncing/migrating the database. Depending on your Django version and migration app, this step may vary. Typically as follows:

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

2) Sync installed `dash` plugins. Go to terminal and type the following command.

```
./manage.py dash_sync_plugins
```

26.1.1.1.6 Specify the active layout

Specify the active/default layout in your `settings` module.

```
DASH_ACTIVE_LAYOUT = 'android'
```

26.1.1.1.7 Permissions

`dash` has been built with permissions in mind. Every single plugin is permission based. If user hasn't been given permission to work with a plugin, he won't be. If you want to switch the permission checks off, set the value of `DASH_RESTRICT_PLUGIN_ACCESS` to False in your `settings` module.

```
DASH_RESTRICT_PLUGIN_ACCESS = False
```

Otherwise, after having completed all the steps above, do log into the Django administration and assign the permissions (to certain user or a group) for every single form element or form handler plugin. Bulk assignments work as well.

- http://yourdomain.com/admin/dash/plugin/

Also, make sure to have the Django model permissions set for following models:

- dash.models.DashboardEntry
- dash.models.DashboardPlugin
- dash.models.DashboardSettings
- dash.models.DashboardWorkspace

26.2 Release history and notes

Sequence based identifiers are used for versioning (schema follows below):

```
major.minor[.revision]
```

- It's always safe to upgrade within the same minor version (for example, from 0.3 to 0.3.2).
- Minor version changes might be backwards incompatible. Read the release notes carefully before upgrading (for example, when upgrading from 0.3.2 to 0.4).
- All backwards incompatible changes are mentioned in this document.

26.2.1 0.6.1

2021-03-05

- Minor fixes.
- More tests.

26.2.2 0.6

2021-03-02

Note: Release dedicated to defenders of Armenia and Artsakh (Nagorno Karabakh) and all the victims of Turkish and Azerbaijani aggression.

- Drop support for Django < 2.2.
- Drop support for Python < 3.6.
- Add support for Django 2.2, 3.0 and 3.1.
- Fixes in documentation.

26.2.3 0.5.5

2019-08-06

- Some improvements in Django 1.11 support.

26.2.4 0.5.5

2018-02-09

- Minor fixes.

26.2.5 0.5.4

2017-12-27

- Django 2.0 support (experimental).
- Some work on removing Django < 1.8 code.

26.2.6 0.5.3

2017-09-05

- From now on, URL names for login and logout views are defined in the settings, in `AUTH_LOGIN_URL_NAME` and `AUTH_LOGOUT_URL_NAME` respectively. Defaults are `auth_login` and `auth_logout` (as in `django-registration` package, that most of the developers would be using).

26.2.7 0.5.2

2017-03-13

- Fixes in demo.
- Minor fixes.

26.2.8 0.5.1

2017-03-12

- Django 1.11 support.

26.2.9 0.5

2017-03-11

This is a transitional release. In upcoming versions support for older versions of Django (1.5, 1.6 and 1.7) will be dropped.

- From now on it's possible to set a layout for the workspace. It means you may have different layouts for various workspaces of the same dashboard.
- Moving static files of Android, Bootstrap2 and Windows8 layouts into a separate directory (android, bootstrap2 and windows8 respectively).
- The `vishap` package dependency updated to the version 0.1.3 (which contained an small yet important fix).
- First discover the plugin modules, then the layouts (was the opposite).
- Remove redundant assets (resulted to smaller package size).
- Added the following templates to the layout definitions for simpler further customisation: 'dash/add_dashboard_entry_ajax.html' and 'dash/edit_dashboard_entry_ajax.html'.

- Better referencing the custom user model in foreign key relations by using `settings.AUTH_USER_MODEL` instead `django.contrib.auth.get_user_model`.
- Fix wrong app label of the dummy plugin (`dash.contrib.plugins.dummy`), which caused import errors on Django >= 1.7.
- From now on it's possible to localise (translated) URLs.
- Compatibility with Django 1.8/1.9/1.10.
- Remove redundant dependencies. Mention, that some of the plugins do have additional dependencies.
- Performing additional checks for collision detection when inserting a new plugin.
- Improved autodiscover for Django>=1.7. Fix exception when using a dotted path to an `AppConfig` in `INSTALLED_APPS` (instead of using the path to the app).
- Fixed wrongly formed app config labels.
- Minor Python3 improvements.
- Clean up the documentation.
- Make a quick start.
- Upgraded `jquery.colorbox` to the latest version (1.6.4).
- Stopped using `django-localeurl`.
- Stopped using `django-slim` for translations.
- PEP8 conform code.
- Use OneToOneField instead of ForeignKey on DashboardSettings module. Remove null=True from DashPlugin module users and groups relations in order to get rid of Django system warnings. Added necessary migrations.
- Using pytest as test runner. Using coverage.

26.2.10 0.4.13

2015-03-20

- Minor fixes.

26.2.11 0.4.12

2015-01-08

This release contains a small, yet important fix. You are recommended to upgrade to this version as soon as possible.

- Improved Django 1.7 support.
- Support for wheel packages.
- Soften requirements.
- Mention the heroku demo app in the docs.
- Fix a mistake in *dash.utils.get_user_plugin_uids* function due to which the list of allowed user plugin uids for non-admins was always empty.

26.2.12 0.4.11

2014-12-21

- Clipboard module for copy, cut and paste operations.
- Make it possible to provide a template for rendering the plugin widgets popup dialogue.
- Improvements in Bootstrap 2 layout (using Bootstrap 2 own accordion instead of the one coming with jQuery UI in the plugin widgets popup).
- If *ujson* or *simplejson* are installed, they're used in preference to stldib *json* module.
- Minor improvements and fixes.

26.2.13 0.4.10

2014-12-10

- Minor fixes in Image plugin.
- Minor fixes in RSS feed plugin.

26.2.14 0.4.9

2014-10-22

- Fixed exceptions raised when unicode characters were used as dashboard names.
- Softened setup requirements.
- Moved *dash.contrib.plugins.news* into the *examples.example* example project. If you have used it, change the path in your projects' *settings.py* module accordingly.
- Documentation improvements.

26.2.15 0.4.8

2014-10-12

- Django 1.7 support.

26.2.16 0.4.7

2014-10-01

- Sort widgets alphabetically.
- UI improvements.

26.2.17 0.4.6

2014-07-09

- Allow custom user model.

26.2.18 0.4.5

2014-05-21

- Added 'rem', 'in', 'cm', 'mm', 'ex' and 'pc' units to the list of available units.
- Softened dependencies.

26.2.19 0.4.4

2014-03-26

- Minor fixes.

26.2.20 0.4.3

2013-12-21

- Add Bookmark plugin.
- Improvements (simplification) of the API related to force-updating of plugin data, triggered by developers upon changes in source models, used by certain plugins.

26.2.21 0.4.2

2013-12-08

- Fix extra (duplicate) menu appearing on the public dashboard of the "Bootstrap2 Fluid" layout.

26.2.22 0.4.1

2013-12-08

- Added Dutch and Russian translations for the missing parts.

26.2.23 0.4

2013-12-07

While core stayed almost intact, there have been major changes made to plugins and widgets. If you have written your own plugins and widgets, having inherited existing ones, review your code before updating to this version. It would be very simple to migrate, though. All layout specific widgets have been moved to layout modules, having the plugins only implemented base widgets, which are used (subclassed) by plugins and widgets specified in layouts. Moreover, a factory feature for plugins and widgets has been introduced. Take *android* layout as example.

- Plugin and widget factory added, which decreases the amount of plugin and widget code by 90%.
- Dashboard workspace cloning feature added. There are two options. Either clone your own workspace or if someone has marked his workspace as public and cloneable, an extra option appears on the public dashboard, which allows you to clone given workspace.
- Clone dashboard entry feature added (at the moment, API level only).
- In bootstrap 2 fluid layout, the menu items "Edit dashboard" and "View dashboard" swapped positions.

- Default widgets added for all plugins. All existing widgets relocated. If you have inherited from any layout specific widget, you will need to update your code.
- Bulk change users and groups in dashboard plugins Django admin interface.
- Weather 1x1 widget which formerly had uid "weather" got changed to "weather_1x1". If you used that widget, you may want to update your database.
- Fixed bug in public dashboard app, when requesting placeholders by their name in the template scope didn't work (while iteration through the placeholders did work).

26.2.24 0.3.2

2013-11-24

- Fix image plugin bug occurring with "Fit width" and "Fit height" resize methods.

26.2.25 0.3.1

2013-11-24

- Fixed issue when the left gray menu (workspaces) is empty in cases when only default workspace is available.

26.2.26 0.3

2013-11-24

- Bootstrap 2 Fluid layout added.
- Fixed permission issue (non-admins not able to edit current workspace).
- Fixed image plugin unique file names issue.
- Fixed bug with placeholder rendering (wrong template chosen).
- Placeholder cell margins definable for each placeholder.
- Customisable form snippets for layouts.
- The very essential core CSS moved to a separate file (dash_core.css).
- Plugin and widget documentation brought in accordance with new naming conventions.
- Overall cleanup and improvements.

26.2.27 0.2.4

2013-11-09

- Now when workspace is deleted, the plugin `delete_plugin_data` method is fired for all dashboard entries so that all the related plugin data is wiped as well.
- Move layout borders into separate stylesheet, making it easy to switch between those.

26.2.28 0.2.3

2013-11-08

- Making it possible to refer to a placeholder by it's uid in templates.
- Nice example project with example layouts, plugins and widgets.
- Added notes about Django 1.6 support (seems to work, although not yet proclaimed to be flawlessly supported).
- Some core improvements.
- Updated demo installer.

26.2.29 0.2.2

2013-11-07

- Fixed bug with string translation (cyrillic) when adding a dashboard widget.
- Russian translations added.

26.2.30 0.2.1

2013-11-07

- Fixed resizing of images in Image widget for Windows 8 layout.

26.2.31 0.2

2013-11-07

- Added Image plugin.
- All existing plugin and widget names are brought in accordance with new naming convention (http://pythonhosted.org/django-dash/#naming-conventions). If you're using the old plugins, you're likely want to clean up your dashboard and start over.
- Some improvements of core.
- Adding `get_size`, `get_width` and `get_height` methods to the plugin widget class.

26.2.32 0.1.4

2013-11-05

- Added Dutch translations.
- Better documentation.

26.2.33 0.1.3

2013-11-01

- Fix adding up assets when switching between dashboard workspaces.
- Better documentation.

26.2.34 0.1.2

2013-10-31

- Replace `DISPLAY_LOGOUT_LINK` with `DISPLAY_AUTH_LINK`.
- Better documentation.

26.2.35 0.1.1

2013-10-31

- Adding home page to example project.
- Better documentation.

26.2.36 0.1

2013-10-30

- Initial.

26.3 dash package

26.3.1 Subpackages

26.3.1.1 dash.contrib package

26.3.1.1.1 Subpackages

26.3.1.1.1.1 dash.contrib.apps package

26.3.1.1.1.2 Subpackages

26.3.1.1.1.3 dash.contrib.apps.public_dashboard package

26.3.1.1.1.4 Submodules

26.3.1.1.1.5 dash.contrib.apps.public_dashboard.apps module

class dash.contrib.apps.public_dashboard.apps.**Config**(*app_name*, *app_module*)

Bases: django.apps.config.AppConfig

Config.

label = 'dash_contrib_apps_public_dashboard'

name = 'dash.contrib.apps.public_dashboard'

26.3.1.1.1.6 dash.contrib.apps.public_dashboard.urls module

26.3.1.1.1.7 dash.contrib.apps.public_dashboard.views module

26.3.1.1.1.8 Module contents

26.3.1.1.1.9 Module contents

26.3.1.1.1.10 dash.contrib.layouts package

26.3.1.1.1.11 Subpackages

26.3.1.1.1.12 dash.contrib.layouts.android package

26.3.1.1.1.13 Submodules

26.3.1.1.1.14 dash.contrib.layouts.android.apps module

class dash.contrib.layouts.android.apps.**Config**(*app_name*, *app_module*)
Bases: django.apps.config.AppConfig

Config.

label = 'dash_contrib_layouts_android'

name = 'dash.contrib.layouts.android'

26.3.1.1.1.15 dash.contrib.layouts.android.dash_layouts module

class dash.contrib.layouts.android.dash_layouts.**AndroidLayout**(*user=None*)
Bases: *dash.base.BaseDashboardLayout*

Android layout.

cell_units = 'px'

edit_template_name = 'android/edit_layout.html'

media_css = ('css/dash_dotted_borders.css', 'android/css/dash_layout_android.css')

name = 'Android'

placeholders = [<class 'dash.contrib.layouts.android.dash_layouts.AndroidMainPlacehold

uid = 'android'

view_template_name = 'android/view_layout.html'

26.3.1.1.1.16 dash.contrib.layouts.android.dash_plugins module

26.3.1.1.1.17 dash.contrib.layouts.android.dash_widgets module

class dash.contrib.layouts.android.dash_widgets.**BaseBookmarkAndroidWidget**(*plugin*)
Bases: *dash.contrib.plugins.url.dash_widgets.BaseBookmarkWidget*

Base Bookmark plugin widget for Android layout.

media_css = ('css/dash_plugin_bookmark_android.css',)

class dash.contrib.layouts.android.dash_widgets.**URL1x1AndroidMainWidget**(*plugin*)
Bases: *dash.contrib.plugins.url.dash_widgets.URL1x1Widget*

URL plugin widget for Android layout (placeholder *main*).

layout_uid = 'android'

media_css = ('css/dash_plugin_url_android.css',)

placeholder_uid = 'main'

class dash.contrib.layouts.android.dash_widgets.**URL1x1AndroidShortcutWidget**(*plugin*)
Bases: *dash.contrib.layouts.android.dash_widgets.URL1x1AndroidMainWidget*

URL plugin widget for Android layout (placeholder *shortcuts*).

placeholder_uid = 'shortcuts'

26.3.1.1.1.18 Module contents

26.3.1.1.1.19 dash.contrib.layouts.bootstrap2 package

26.3.1.1.1.20 Submodules

26.3.1.1.1.21 dash.contrib.layouts.bootstrap2.apps module

class dash.contrib.layouts.bootstrap2.apps.**Config**(*app_name*, *app_module*)
Bases: django.apps.config.AppConfig

Config.

label = 'dash_contrib_layouts_bootstrap2'

name = 'dash.contrib.layouts.bootstrap2'

26.3.1.1.1.22 dash.contrib.layouts.bootstrap2.conf module

dash.contrib.layouts.bootstrap2.conf.**get_setting**(*setting*, *override=None*)
Get setting.

Get a setting from dash.contrib.layouts.bootstrap2 conf module, falling back to the default.

If override is not None, it will be used instead of the setting.

Parameters

- **setting** – String with setting name
- **override** – Value to use when no setting is available. Defaults to None.

Returns Setting value.

26.3.1.1.1.23 dash.contrib.layouts.bootstrap2.dash_layouts module

class dash.contrib.layouts.bootstrap2.dash_layouts.**Bootstrap2FluidLayout**(*user=None*)
Bases: *dash.base.BaseDashboardLayout*

Bootstrap 2 Fluid layout.

cell_units = 'px'

edit_template_name = 'bootstrap2/fluid_edit_layout.html'

form_snippet_template_name = 'bootstrap2/snippets/generic_form_snippet.html'

get_view_template_name(*request=None*, *origin=None*)
Override the master view template for public dashboard app.

media_css = ('bootstrap2/css/bootstrap.css', 'bootstrap2/css/dash_layout_bootstap2_flu

media_js = ('bootstrap2/js/bootstrap.js', 'bootstrap2/js/dash_layout_bootstap2_fluid.j

name = 'Bootstrap 2 Fluid'

placeholders = [<class 'dash.contrib.layouts.bootstrap2.dash_layouts.Bootstrap2FluidMa

plugin_widgets_template_name_ajax = 'bootstrap2/plugin_widgets_ajax.html'

uid = 'bootstrap2_fluid'

view_template_name = 'bootstrap2/fluid_view_layout.html'

26.3.1.1.1.24 dash.contrib.layouts.bootstrap2.dash_plugins module

26.3.1.1.1.25 dash.contrib.layouts.bootstrap2.dash_widgets module

class dash.contrib.layouts.bootstrap2.dash_widgets.**BaseBookmarkBootstrapTwoWidget**(*plugin*)
Bases: *dash.contrib.plugins.url.dash_widgets.BaseBookmarkWidget*

Base Bookmark plugin widget for Bootstrap 2 Fluid layout.

media_css = ('css/dash_plugin_bookmark_bootstrap2.css',)

class dash.contrib.layouts.bootstrap2.dash_widgets.**URLBootstrapTwo1x1Bootstrap2FluidMainWi**
Bases: *dash.contrib.plugins.url.dash_widgets.BaseURLWidget*

URL plugin 1x1 widget for Bootstrap 2 Fluid layout.

Placeholder *main* widget.

layout_uid = 'bootstrap2_fluid'

media_css = ('css/dash_plugin_url_bootstrap2.css',)

placeholder_uid = 'main'

plugin_uid = 'url_bootstrap_two_1x1'

class dash.contrib.layouts.bootstrap2.dash_widgets.**URLBootstrapTwo2x2Bootstrap2FluidMainWi**
Bases: *dash.contrib.layouts.bootstrap2.dash_widgets.URLBootstrapTwo1x1Bootstrap2FluidMai*

URL2x2 plugin widget for Bootstrap 2 Fluid layout.

Placeholder *main* widget.

cols = 2

```
plugin_uid = 'url_bootstrap_two_2x2'
rows = 2
```

26.3.1.1.1.26 dash.contrib.layouts.bootstrap2.defaults module

26.3.1.1.1.27 dash.contrib.layouts.bootstrap2.forms module

26.3.1.1.1.28 dash.contrib.layouts.bootstrap2.settings module

26.3.1.1.1.29 Module contents

26.3.1.1.1.30 dash.contrib.layouts.windows8 package

26.3.1.1.1.31 Submodules

26.3.1.1.1.32 dash.contrib.layouts.windows8.apps module

class dash.contrib.layouts.windows8.apps.**Config**(*app_name*, *app_module*)
Bases: django.apps.config.AppConfig

Config.

```
label = 'dash_contrib_layouts_windows8'
name = 'dash.contrib.layouts.windows8'
```

26.3.1.1.1.33 dash.contrib.layouts.windows8.dash_layouts module

class dash.contrib.layouts.windows8.dash_layouts.**Windows8Layout**(*user=None*)
Bases: *dash.base.BaseDashboardLayout*

Windows8 layout.

```
cell_units = 'px'
edit_template_name = 'windows8/edit_layout.html'
media_css = ('windows8/css/dash_solid_borders.css', 'windows8/css/dash_layout_windows8
name = 'Windows 8'
placeholders = [<class 'dash.contrib.layouts.windows8.dash_layouts.Windows8MainPlaceho
uid = 'windows8'
view_template_name = 'windows8/view_layout.html'
```

26.3.1.1.1.34 dash.contrib.layouts.windows8.dash_plugins module

26.3.1.1.1.35 dash.contrib.layouts.windows8.dash_widgets module

class dash.contrib.layouts.windows8.dash_widgets.**BaseBookmarkWindows8Widget**(*plugin*)
Bases: *dash.contrib.plugins.url.dash_widgets.BaseBookmarkWidget*

Base Bookmark plugin widget for Windows8 layout.

media_css = ('css/dash_plugin_bookmark_windows8.css',)

class dash.contrib.layouts.windows8.dash_widgets.**URL1x1Windows8MainWidget**(*plugin*)
Bases: *dash.contrib.plugins.url.dash_widgets.URL1x1Widget*

URL plugin widget for Windows 8 layout (placeholder *main*).

layout_uid = 'windows8'

media_css = ('css/dash_plugin_url_windows8.css',)

placeholder_uid = 'main'

class dash.contrib.layouts.windows8.dash_widgets.**URL1x1Windows8SidebarWidget**(*plugin*)
Bases: *dash.contrib.layouts.windows8.dash_widgets.URL1x1Windows8MainWidget*

URL plugin widget for Windows 8 layout (placeholder *sidebar*).

placeholder_uid = 'sidebar'

26.3.1.1.1.36 Module contents

26.3.1.1.1.37 Module contents

26.3.1.1.1.38 dash.contrib.plugins package

26.3.1.1.1.39 Subpackages

26.3.1.1.1.40 dash.contrib.plugins.dummy package

26.3.1.1.1.41 Submodules

26.3.1.1.1.42 dash.contrib.plugins.dummy.apps module

class dash.contrib.plugins.dummy.apps.**Config**(*app_name*, *app_module*)
Bases: django.apps.config.AppConfig

label = 'dash_contrib_plugins_dummy'

name = 'dash.contrib.plugins.dummy'

26.3.1.1.1.43 dash.contrib.plugins.dummy.dash_plugins module

class dash.contrib.plugins.dummy.dash_plugins.**BaseDummyPlugin**(*layout_uid*, *placeholder_uid*, *workspace=None*, *user=None*, *position=None*)
Bases: *dash.base.BaseDashboardPlugin*

Base dummy plugin.

form
alias of *dash.contrib.plugins.dummy.forms.DummyForm*

get_form()
Get form.

group

name

post_processor()
If no text available, use dummy.

26.3.1.1.1.44 dash.contrib.plugins.dummy.dash_widgets module

class dash.contrib.plugins.dummy.dash_widgets.**BaseDummyWidget**(*plugin*)
Bases: *dash.base.BaseDashboardPluginWidget*

Base dummy plugin widget.

media_css = []

media_js = []

render(*request=None*)
Render.

class dash.contrib.plugins.dummy.dash_widgets.**Dummy1x1Widget**(*plugin*)
Bases: *dash.contrib.plugins.dummy.dash_widgets.BaseDummyWidget*

1x1 dummy plugin widget.

plugin_uid = 'dummy_1x1'

class dash.contrib.plugins.dummy.dash_widgets.**Dummy1x2Widget**(*plugin*)
Bases: *dash.contrib.plugins.dummy.dash_widgets.BaseDummyWidget*

1x2 dummy plugin widget.

cols = 1

plugin_uid = 'dummy_1x2'

rows = 2

class dash.contrib.plugins.dummy.dash_widgets.**Dummy2x1Widget**(*plugin*)
Bases: *dash.contrib.plugins.dummy.dash_widgets.BaseDummyWidget*

2x1 dummy plugin widget.

cols = 2

plugin_uid = 'dummy_2x1'

rows = 1

class dash.contrib.plugins.dummy.dash_widgets.**Dummy2x2Widget**(*plugin*)
Bases: *dash.contrib.plugins.dummy.dash_widgets.BaseDummyWidget*

2x2 dummy plugin widget.

cols = 2

plugin_uid = 'dummy_2x2'

rows = 2

class dash.contrib.plugins.dummy.dash_widgets.**Dummy3x3Widget**(*plugin*)
Bases: *dash.contrib.plugins.dummy.dash_widgets.BaseDummyWidget*

3x3 dummy plugin widget.

cols = 3

plugin_uid = 'dummy_3x3'

rows = 3

26.3.1.1.1.45 dash.contrib.plugins.dummy.defaults module

26.3.1.1.1.46 dash.contrib.plugins.dummy.forms module

class dash.contrib.plugins.dummy.forms.**DummyForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*, *renderer=None*)
Bases: django.forms.forms.Form, *dash.base.DashboardPluginFormBase*

Dummy form (for main *placeholder*).

base_fields = {'generate_lipsum': <django.forms.fields.BooleanField object>, 'lipsum_

declared_fields = {'generate_lipsum': <django.forms.fields.BooleanField object>, 'lip

media

plugin_data_fields = [('show_title', False), ('generate_lipsum', False), ('lipsum_lang

save_plugin_data(*request=None*)
Save plugin data.

We want to save the generated lorem ipsum text for later use. Thus, although we don't show it to the user, in case when generate_lipsum field is set to True, we silently generate the text and save it into the plugin data.

class dash.contrib.plugins.dummy.forms.**DummyShortcutsForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*, *renderer=None*)
Bases: *dash.contrib.plugins.dummy.forms.DummyForm*

Dummy form for *shortucts* placeholder.

base_fields = {'generate_lipsum': <django.forms.fields.BooleanField object>, 'lipsum_

declared_fields = {'generate_lipsum': <django.forms.fields.BooleanField object>, 'lip

media

26.3.1.1.1.47 Module contents

26.3.1.1.1.48 dash.contrib.plugins.image package

26.3.1.1.1.49 Submodules

26.3.1.1.1.50 dash.contrib.plugins.image.apps module

class dash.contrib.plugins.image.apps.**Config**(*app_name*, *app_module*)
Bases: django.apps.config.AppConfig

Config.

label = 'dash_contrib_plugins_image'

name = 'dash.contrib.plugins.image'

26.3.1.1.1.51 dash.contrib.plugins.image.conf module

dash.contrib.plugins.image.conf.**get_setting**(*setting*, *override=None*)
Get setting.

Get a setting from dash.contrib.plugins.image conf module, falling back to the default.

If override is not None, it will be used instead of the setting.

Parameters

- **setting** – String with setting name
- **override** – Value to use when no setting is available. Defaults to None.

Returns Setting value.

26.3.1.1.1.52 dash.contrib.plugins.image.dash_plugins module

class dash.contrib.plugins.image.dash_plugins.**BaseImagePlugin**(*layout_uid*, *placeholder_uid*, *workspace=None*, *user=None*, *position=None*)
Bases: *dash.base.BaseDashboardPlugin*

Base image plugin.

clone_plugin_data(*dashboard_entry*)
Clone plugin data, which means we make a copy of the original image.

TODO: Perhaps rely more on data of dashboard_entry?

delete_plugin_data()
Deletes uploaded file.

form
alias of *dash.contrib.plugins.image.forms.ImageForm*

group

html_classes = ['pictonic']

name

26.3.1.1.1.53 dash.contrib.plugins.image.dash_widgets module

class dash.contrib.plugins.image.dash_widgets.**BaseImageWidget**(*plugin*)
Bases: *dash.base.BaseDashboardPluginWidget*

Base image plugin widget.

media_css = ('css/dash_plugin_image.css',)

media_js = ('js/dash_plugin_image.js',)

render(*request=None*)
Render.

class dash.contrib.plugins.image.dash_widgets.**Image1x1Widget**(*plugin*)
Bases: *dash.contrib.plugins.image.dash_widgets.BaseImageWidget*

Image1x1 plugin widget.

plugin_uid = 'image_1x1'

class dash.contrib.plugins.image.dash_widgets.**Image1x2Widget**(*plugin*)
Bases: *dash.contrib.plugins.image.dash_widgets.BaseImageWidget*

Image1x2 plugin widget.

cols = 1

plugin_uid = 'image_1x2'

rows = 2

class dash.contrib.plugins.image.dash_widgets.**Image2x1Widget**(*plugin*)
Bases: *dash.contrib.plugins.image.dash_widgets.BaseImageWidget*

Image2x1 plugin widget.

cols = 2

plugin_uid = 'image_2x1'

rows = 1

class dash.contrib.plugins.image.dash_widgets.**Image2x2Widget**(*plugin*)
Bases: *dash.contrib.plugins.image.dash_widgets.BaseImageWidget*

Image2x2 plugin widget.

cols = 2

plugin_uid = 'image_2x2'

rows = 2

class dash.contrib.plugins.image.dash_widgets.**Image2x3Widget**(*plugin*)
Bases: *dash.contrib.plugins.image.dash_widgets.BaseImageWidget*

Image2x3 plugin widget.

cols = 2

plugin_uid = 'image_2x3'

rows = 3

class dash.contrib.plugins.image.dash_widgets.**Image3x2Widget**(*plugin*)
Bases: *dash.contrib.plugins.image.dash_widgets.BaseImageWidget*

Image3x2 plugin widget.

cols = 3

plugin_uid = 'image_3x2'

rows = 2

class dash.contrib.plugins.image.dash_widgets.**Image3x3Widget**(*plugin*)
Bases: *dash.contrib.plugins.image.dash_widgets.BaseImageWidget*

Image3x3 plugin widget.

cols = 3

plugin_uid = 'image_3x3'

rows = 3

class dash.contrib.plugins.image.dash_widgets.**Image3x4Widget**(*plugin*)
Bases: *dash.contrib.plugins.image.dash_widgets.BaseImageWidget*

Image3x4 plugin widget.

cols = 3

plugin_uid = 'image_3x4'

rows = 4

class dash.contrib.plugins.image.dash_widgets.**Image4x3Widget**(*plugin*)
Bases: *dash.contrib.plugins.image.dash_widgets.BaseImageWidget*

Image4x3 plugin widget.

cols = 4

plugin_uid = 'image_4x3'

rows = 3

class dash.contrib.plugins.image.dash_widgets.**Image4x4Widget**(*plugin*)
Bases: *dash.contrib.plugins.image.dash_widgets.BaseImageWidget*

Image4x4 plugin widget.

cols = 4

plugin_uid = 'image_4x4'

rows = 4

class dash.contrib.plugins.image.dash_widgets.**Image4x5Widget**(*plugin*)
Bases: *dash.contrib.plugins.image.dash_widgets.BaseImageWidget*

Image4x5 plugin widget.

cols = 4

plugin_uid = 'image_4x5'

rows = 5

class dash.contrib.plugins.image.dash_widgets.**Image5x4Widget**(*plugin*)
Bases: *dash.contrib.plugins.image.dash_widgets.BaseImageWidget*

Image5x4 plugin widget.

cols = 5

plugin_uid = 'image_5x4'

rows = 4

class dash.contrib.plugins.image.dash_widgets.**Image5x5Widget**(*plugin*)
Bases: *dash.contrib.plugins.image.dash_widgets.BaseImageWidget*

Image5x5 plugin widget.

cols = 5

plugin_uid = 'image_5x5'

rows = 5

26.3.1.1.1.54 dash.contrib.plugins.image.defaults module

26.3.1.1.1.55 dash.contrib.plugins.image.forms module

class dash.contrib.plugins.image.forms.**ImageForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*, *renderer=None*)
Bases: django.forms.forms.Form, *dash.base.DashboardPluginFormBase*

Image form for *ImagePlugin* plugin.

base_fields = {'fit_method': <django.forms.fields.ChoiceField object>, 'image': <dja

declared_fields = {'fit_method': <django.forms.fields.ChoiceField object>, 'image':

media

plugin_data_fields = [('title', ''), ('image', ''), ('fit_method', 'center'), ('show_l

save_plugin_data(*request=None*)
Saving the plugin data and moving the file.

26.3.1.1.1.56 dash.contrib.plugins.image.helpers module

dash.contrib.plugins.image.helpers.**clone_file**(*source_filename*, *relative_path=True*)
Clone the file.

Parameters

- **source_filename** (*string*) – Source filename.
- **relative_path** (*str*) –

Return string Filename of the cloned file.

dash.contrib.plugins.image.helpers.**delete_file**(*image_file*)
Delete file from disc.

dash.contrib.plugins.image.helpers.**ensure_unique_filename**(*destination*)
Ensure unique filename.

Makes sure filenames are never overwritten. If file name already exists, makes a new one based on the first 50 chars of the original file name, having a uuid4 appended afterwards.

Parameters **destination** (*string*) –

Return string

dash.contrib.plugins.image.helpers.**get_crop_filter**(*fit_method*)
Get crop filter.

dash.contrib.plugins.image.helpers.**handle_uploaded_file**(*image_file*)
Handle uploaded file.

Parameters **image_file** (*django.core.files.uploadedfile.InMemoryUploadedFile*) –

Return string Path to the image (relative).

26.3.1.1.1.57 dash.contrib.plugins.image.settings module

- FIT_METHOD_CROP_SMART (string)
- FIT_METHOD_CROP_CENTER (string)
- FIT_METHOD_CROP_SCALE (string)
- FIT_METHOD_FIT_WIDTH (string)
- FIT_METHOD_FIT_HEIGHT (string)
- DEFAULT_FIT_METHOD (string)
- FIT_METHODS_CHOICES (tuple)
- FIT_METHODS_CHOICES_WITH_EMPTY_OPTION (list)
- IMAGES_UPLOAD_DIR (string)

26.3.1.1.1.58 Module contents

26.3.1.1.1.59 dash.contrib.plugins.memo package

26.3.1.1.1.60 Submodules

26.3.1.1.1.61 dash.contrib.plugins.memo.apps module

class dash.contrib.plugins.memo.apps.**Config**(*app_name*, *app_module*)
Bases: django.apps.config.AppConfig

Config.

label = 'dash_contrib_plugins_memo'

name = 'dash.contrib.plugins.memo'

26.3.1.1.1.62 dash.contrib.plugins.memo.dash_plugins module

class dash.contrib.plugins.memo.dash_plugins.**BaseMemoPlugin**(*layout_uid*, *placeholder_uid*, *workspace=None*, *user=None*, *position=None*)

Bases: *dash.base.BaseDashboardPlugin*

Base memo plugin.

form

alias of *dash.contrib.plugins.memo.forms.MemoForm*

group

name

class dash.contrib.plugins.memo.dash_plugins.**BaseTinyMCEMemoPlugin**(*layout_uid*, *placeholder_uid*, *workspace=None*, *user=None*, *position=None*)

Bases: *dash.base.BaseDashboardPlugin*

Memo dashboard plugin.

form

alias of *dash.contrib.plugins.memo.forms.TinyMCEMemoForm*

group

help_text

name

26.3.1.1.1.63 dash.contrib.plugins.memo.dash_widgets module

class dash.contrib.plugins.memo.dash_widgets.**BaseMemoWidget**(*plugin*)

Bases: *dash.base.BaseDashboardPluginWidget*

Base memo plugin widget.

render(*request=None*)

Render.

class dash.contrib.plugins.memo.dash_widgets.**Memo1x1Widget**(*plugin*)

Bases: *dash.contrib.plugins.memo.dash_widgets.BaseMemoWidget*

Memo 1x1 plugin widget.

cols = 1

plugin_uid = 'memo_1x1'

rows = 1

class dash.contrib.plugins.memo.dash_widgets.**Memo2x2Widget**(*plugin*)
Bases: *dash.contrib.plugins.memo.dash_widgets.BaseMemoWidget*

Memo 2x2 plugin widget.

cols = 2

plugin_uid = 'memo_2x2'

rows = 2

class dash.contrib.plugins.memo.dash_widgets.**Memo3x3Widget**(*plugin*)
Bases: *dash.contrib.plugins.memo.dash_widgets.BaseMemoWidget*

Memo 3x3 plugin widget.

cols = 3

plugin_uid = 'memo_3x3'

rows = 3

class dash.contrib.plugins.memo.dash_widgets.**Memo4x5Widget**(*plugin*)
Bases: *dash.contrib.plugins.memo.dash_widgets.BaseMemoWidget*

Memo 4x5 plugin widget.

cols = 4

plugin_uid = 'memo_4x5'

rows = 5

class dash.contrib.plugins.memo.dash_widgets.**Memo5x5Widget**(*plugin*)
Bases: *dash.contrib.plugins.memo.dash_widgets.BaseMemoWidget*

Memo 5x5 plugin widget.

cols = 5

plugin_uid = 'memo_5x5'

rows = 5

class dash.contrib.plugins.memo.dash_widgets.**Memo6x6Widget**(*plugin*)
Bases: *dash.contrib.plugins.memo.dash_widgets.BaseMemoWidget*

Memo 6x6 plugin widget.

cols = 6

plugin_uid = 'memo_6x6'

rows = 6

class dash.contrib.plugins.memo.dash_widgets.**BaseTinyMCEMemoWidget**(*plugin*)
Bases: *dash.base.BaseDashboardPluginWidget*

Base TinyMCE memo plugin widget.

render(*request=None*)
Render.

class dash.contrib.plugins.memo.dash_widgets.**TinyMCEMemo2x2Widget**(*plugin*)
Bases: *dash.contrib.plugins.memo.dash_widgets.BaseTinyMCEMemoWidget*

TinyMCE memo 2x2 plugin widget.

cols = 2

plugin_uid = 'tinymce_memo_2x2'

rows = 2

class dash.contrib.plugins.memo.dash_widgets.**TinyMCEMemo3x3Widget**(*plugin*)
Bases: *dash.contrib.plugins.memo.dash_widgets.BaseTinyMCEMemoWidget*

TinyMCE memo 3x3 plugin widget.

cols = 3

plugin_uid = 'tinymce_memo_3x3'

rows = 3

class dash.contrib.plugins.memo.dash_widgets.**TinyMCEMemo4x4Widget**(*plugin*)
Bases: *dash.contrib.plugins.memo.dash_widgets.BaseTinyMCEMemoWidget*

TinyMCE memo 4x4 plugin widget.

cols = 4

plugin_uid = 'tinymce_memo_4x4'

rows = 4

class dash.contrib.plugins.memo.dash_widgets.**TinyMCEMemo5x5Widget**(*plugin*)
Bases: *dash.contrib.plugins.memo.dash_widgets.BaseTinyMCEMemoWidget*

TinyMCE memo 5x5 plugin widget.

cols = 5

plugin_uid = 'tinymce_memo_5x5'

rows = 5

26.3.1.1.1.64 dash.contrib.plugins.memo.forms module

class dash.contrib.plugins.memo.forms.**MemoForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*, *renderer=None*)
Bases: django.forms.forms.Form, *dash.base.DashboardPluginFormBase*

Memo form (for Memo plugin).

base_fields = {'text': <django.forms.fields.CharField object>, 'title': <django.forms

declared_fields = {'text': <django.forms.fields.CharField object>, 'title': <django.

media

plugin_data_fields = [('title', ''), ('text', '')]

class dash.contrib.plugins.memo.forms.**TinyMCEMemoForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*, *renderer=None*)

Bases: django.forms.forms.Form, *dash.base.DashboardPluginFormBase*

TinyMCE memo form (for TinyMCEMemo plugin).

base_fields = {'text': <django.forms.fields.CharField object>, 'title': <django.form

declared_fields = {'text': <django.forms.fields.CharField object>, 'title': <django.

media

plugin_data_fields = [('title', ''), ('text', '')]

26.3.1.1.1.65 Module contents

26.3.1.1.1.66 dash.contrib.plugins.rss_feed package

26.3.1.1.1.67 Subpackages

26.3.1.1.1.68 dash.contrib.plugins.rss_feed.templatetags package

26.3.1.1.1.69 Submodules

26.3.1.1.1.70 dash.contrib.plugins.rss_feed.templatetags.rss_feed_tags module

dash.contrib.plugins.rss_feed.templatetags.rss_feed_tags.**convert_to_datetime**(*value*)

Convert to datetime.

26.3.1.1.1.71 Module contents

26.3.1.1.1.72 Submodules

26.3.1.1.1.73 dash.contrib.plugins.rss_feed.apps module

class dash.contrib.plugins.rss_feed.apps.**Config**(*app_name*, *app_module*)

Bases: django.apps.config.AppConfig

label = 'dash_contrib_plugins_rss_feed'

name = 'dash.contrib.plugins.rss_feed'

26.3.1.1.1.74 dash.contrib.plugins.rss_feed.dash_plugins module

class dash.contrib.plugins.rss_feed.dash_plugins.**BaseReadRSSFeedPlugin**(*layout_uid*, *placeholder_uid*, *workspace=None*, *user=None*, *position=None*)

Bases: *dash.base.BaseDashboardPlugin*

Base Read RSS feed into HTML plugin.

form
alias of *dash.contrib.plugins.rss_feed.forms.ReadRSSFeedForm*

group

name

26.3.1.1.1.75 dash.contrib.plugins.rss_feed.dash_widgets module

class dash.contrib.plugins.rss_feed.dash_widgets.**BaseReadRSSFeedWidget**(*plugin*)

Bases: *dash.base.BaseDashboardPluginWidget*

Base read RSS feed plugin widget.

media_css = ['css/dash_plugin_read_rss_feed.css']

media_js = ['js/dash_plugin_read_rss_feed.js']

render(*request=None*)
Render.

class dash.contrib.plugins.rss_feed.dash_widgets.**ReadRSSFeed2x3Widget**(*plugin*)

Bases: *dash.contrib.plugins.rss_feed.dash_widgets.BaseReadRSSFeedWidget*

Read RSS feed 2x3 plugin widget.

cols = 2

plugin_uid = 'read_rss_feed_2x3'

rows = 3

class dash.contrib.plugins.rss_feed.dash_widgets.**ReadRSSFeed3x3Widget**(*plugin*)

Bases: *dash.contrib.plugins.rss_feed.dash_widgets.BaseReadRSSFeedWidget*

Big read RSS 3x3 feed plugin widget.

cols = 3

plugin_uid = 'read_rss_feed_3x3'

rows = 3

26.3.1.1.1.76 dash.contrib.plugins.rss_feed.defaults module

26.3.1.1.1.77 dash.contrib.plugins.rss_feed.forms module

class dash.contrib.plugins.rss_feed.forms.**ReadRSSFeedForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*, *renderer=None*)

Bases: `django.forms.forms.Form`, *dash.base.DashboardPluginFormBase*

Form for main `ReadRSSFeedPlugin`.

base_fields = {'cache_for': <django.forms.fields.IntegerField object>, 'custom_feed_t

declared_fields = {'cache_for': <django.forms.fields.IntegerField object>, 'custom_fe

media

plugin_data_fields = [('feed_url', ''), ('custom_feed_title', ''), ('show_feed_title',

26.3.1.1.1.78 dash.contrib.plugins.rss_feed.helpers module

dash.contrib.plugins.rss_feed.helpers.**max_num_template**(*max_items*, *default*)

Nax num of items in a template.

26.3.1.1.1.79 dash.contrib.plugins.rss_feed.urls module

26.3.1.1.1.80 dash.contrib.plugins.rss_feed.views module

dash.contrib.plugins.rss_feed.views.**get_feed**(*request*, *layout_uid*, *placeholder_uid*, *plugin_uid*, *template_name='rss_feed/get_feed.html'*, *template_name_ajax='rss_feed/get_feed_ajax.html'*)

Get feed.

Parameters

- **request** (*django.http.HttpRequest*) –
- **layout_uid** (*str*) –
- **placeholder_uid** (*str*) –
- **plugin_uid** (*str*) –
- **template_name** (*str*) –
- **template_name_ajax** (*str*) –

Return django.http.HttpResponse

26.3.1.1.1.81 Module contents

26.3.1.1.1.82 dash.contrib.plugins.url package

26.3.1.1.1.83 Submodules

26.3.1.1.1.84 dash.contrib.plugins.url.admin module

26.3.1.1.1.85 dash.contrib.plugins.url.apps module

class dash.contrib.plugins.url.apps.**Config**(*app_name*, *app_module*)
Bases: django.apps.config.AppConfig

Config.

label = 'dash_contrib_plugins_url'

name = 'dash.contrib.plugins.url'

26.3.1.1.1.86 dash.contrib.plugins.url.conf module

dash.contrib.plugins.url.conf.**get_setting**(*setting*, *override=None*)
Get setting.

Get a setting from dash.contrib.plugins.url conf module, falling back to the default.

If override is not None, it will be used instead of the setting.

Parameters

- **setting** – String with setting name
- **override** – Value to use when no setting is available. Defaults to None.

Returns Setting value.

26.3.1.1.1.87 dash.contrib.plugins.url.dash_plugins module

26.3.1.1.1.88 dash.contrib.plugins.url.dash_widgets module

class dash.contrib.plugins.url.dash_widgets.**BaseBookmarkWidget**(*plugin*)
Bases: *dash.base.BaseDashboardPluginWidget*

Bookmark plugin widget.

render(*request=None*)
Render.

class dash.contrib.plugins.url.dash_widgets.**BaseURLWidget**(*plugin*)
Bases: *dash.base.BaseDashboardPluginWidget*

URL plugin widget.

render(*request=None*)
Render.

class dash.contrib.plugins.url.dash_widgets.**URL1x1Widget**(*plugin*)
Bases: *dash.contrib.plugins.url.dash_widgets.BaseURLWidget*

URL plugin 1x1 widget.

plugin_uid = 'url_1x1'

class dash.contrib.plugins.url.dash_widgets.**URL2x2Widget**(*plugin*)
Bases: *dash.contrib.plugins.url.dash_widgets.BaseURLWidget*

URL plugin 2x2 widget.

cols = 2

plugin_uid = 'url_2x2'

rows = 2

26.3.1.1.1.89 dash.contrib.plugins.url.defaults module

26.3.1.1.1.90 dash.contrib.plugins.url.forms module

26.3.1.1.1.91 dash.contrib.plugins.url.models module

26.3.1.1.1.92 dash.contrib.plugins.url.settings module

26.3.1.1.1.93 Module contents

26.3.1.1.1.94 dash.contrib.plugins.video package

26.3.1.1.1.95 Submodules

26.3.1.1.1.96 dash.contrib.plugins.video.apps module

class dash.contrib.plugins.video.apps.**Config**(*app_name*, *app_module*)
Bases: django.apps.config.AppConfig

Config.

label = 'dash_contrib_plugins_video'

name = 'dash.contrib.plugins.video'

26.3.1.1.1.97 dash.contrib.plugins.video.dash_plugins module

class dash.contrib.plugins.video.dash_plugins.**BaseVideoPlugin**(*layout_uid*, *placeholder_uid*, *workspace=None*, *user=None*, *position=None*)
Bases: *dash.base.BaseDashboardPlugin*

Base Video plugin.

form
alias of *dash.contrib.plugins.video.forms.VideoForm*

group

html_classes = ['video']

name

post_processor()
Post process.

26.3.1.1.1.98 dash.contrib.plugins.video.dash_widgets module

class dash.contrib.plugins.video.dash_widgets.**BaseVideoWidget**(*plugin*)
Bases: *dash.base.BaseDashboardPluginWidget*

Base video plugin widget.

media_css = ('css/dash_plugin_video.css',)

render(*request=None*)
Render.

class dash.contrib.plugins.video.dash_widgets.**Video1x1Widget**(*plugin*)
Bases: *dash.contrib.plugins.video.dash_widgets.BaseVideoWidget*

Video plugin 1x1 widget.

plugin_uid = 'video_1x1'

class dash.contrib.plugins.video.dash_widgets.**Video2x2Widget**(*plugin*)
Bases: *dash.contrib.plugins.video.dash_widgets.BaseVideoWidget*

Video plugin 2x2 widget.

cols = 2

plugin_uid = 'video_2x2'

rows = 2

class dash.contrib.plugins.video.dash_widgets.**Video3x3Widget**(*plugin*)
Bases: *dash.contrib.plugins.video.dash_widgets.BaseVideoWidget*

Video plugin 3x3 widget.

cols = 3

plugin_uid = 'video_3x3'

rows = 3

class dash.contrib.plugins.video.dash_widgets.**Video4x4Widget**(*plugin*)
Bases: *dash.contrib.plugins.video.dash_widgets.BaseVideoWidget*

Video plugin 4x4 widget.

cols = 4

plugin_uid = 'video_4x4'

rows = 4

class dash.contrib.plugins.video.dash_widgets.**Video5x5Widget**(*plugin*)
Bases: *dash.contrib.plugins.video.dash_widgets.BaseVideoWidget*

Video plugin 5x5 widget.

cols = 5

plugin_uid = 'video_5x5'

rows = 5

26.3.1.1.1.99 dash.contrib.plugins.video.forms module

class dash.contrib.plugins.video.forms.**VideoForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*, *use_required_attribute=None*, *renderer=None*)

Bases: django.forms.forms.Form, *dash.base.DashboardPluginFormBase*

Video form for VideoPlugin plugin.

base_fields = {'title': <django.forms.fields.CharField object>, 'url': <django.forms

declared_fields = {'title': <django.forms.fields.CharField object>, 'url': <django.f

media

plugin_data_fields = [('title', ''), ('url', '')]

26.3.1.1.1.100 Module contents

26.3.1.1.1.101 dash.contrib.plugins.weather package

26.3.1.1.1.102 Submodules

26.3.1.1.1.103 dash.contrib.plugins.weather.apps module

class dash.contrib.plugins.weather.apps.**Config**(*app_name*, *app_module*)

Bases: django.apps.config.AppConfig

Config.

label = 'dash_contrib_plugins_weather'

name = 'dash.contrib.plugins.weather'

26.3.1.1.1.104 dash.contrib.plugins.weather.conf module

dash.contrib.plugins.weather.conf.**get_setting**(*setting*, *override=None*)

Get setting.

Get a setting from dash.contrib.plugins.weather conf module, falling back to the default.

If override is not None, it will be used instead of the setting.

Parameters

- **setting** – String with setting name
- **override** – Value to use when no setting is available. Defaults to None.

Returns Setting value.

26.3.1.1.1.105 dash.contrib.plugins.weather.dash_plugins module

class dash.contrib.plugins.weather.dash_plugins.**BaseWeatherPlugin**(*layout_uid*, *placeholder_uid*, *workspace=None*, *user=None*, *position=None*)

Bases: *dash.base.BaseDashboardPlugin*

Base Weather plugin.

form
alias of *dash.contrib.plugins.weather.forms.WeatherForm*

group

name

post_processor()
Post process.

If no text available, use dummy.

26.3.1.1.1.106 dash.contrib.plugins.weather.dash_widgets module

class dash.contrib.plugins.weather.dash_widgets.**BaseWeatherWidget**(*plugin*)

Bases: *dash.base.BaseDashboardPluginWidget*

Base weather plugin widget.

media_css = ['css/dash_plugin_weather.css']

render(*request=None*)
Render.

class dash.contrib.plugins.weather.dash_widgets.**Weather2x2Widget**(*plugin*)

Bases: *dash.contrib.plugins.weather.dash_widgets.BaseWeatherWidget*

Weather plugin 2x2 widget.

cols = 2

plugin_uid = 'weather_2x2'

rows = 2

class dash.contrib.plugins.weather.dash_widgets.**Weather3x3Widget**(*plugin*)

Bases: *dash.contrib.plugins.weather.dash_widgets.BaseWeatherWidget*

Weather plugin 3x3 widget.

cols = 3

plugin_uid = 'weather_3x3'

rows = 3

26.3.1.1.1.107 dash.contrib.plugins.weather.defaults module

26.3.1.1.1.108 dash.contrib.plugins.weather.forms module

class dash.contrib.plugins.weather.forms.**WeatherForm**(**args*, ***kwargs*)
Bases: django.forms.forms.Form, *dash.base.DashboardPluginFormBase*

Form for main WeatherPlugin.

base_fields = {'cache_for': <django.forms.fields.IntegerField object>, 'custom_title'

declared_fields = {'cache_for': <django.forms.fields.IntegerField object>, 'custom_ti

media

plugin_data_fields = [('custom_title', ''), ('show_title', True), ('cache_for', 3600),

save_plugin_data(*request=None*)
Save plugin data.

For showing the weather, we need an IP address. Although we don't make it possible for the user to specify it manually, we silently obtain it and save into the plugin data.

26.3.1.1.1.109 dash.contrib.plugins.weather.settings module

26.3.1.1.1.110 Module contents

26.3.1.1.1.111 Module contents

26.3.1.1.2 Module contents

26.3.1.2 dash.lib package

26.3.1.2.1 Subpackages

26.3.1.2.1.1 dash.lib.tinymce package

26.3.1.2.1.2 Submodules

26.3.1.2.1.3 dash.lib.tinymce.settings module

26.3.1.2.1.4 dash.lib.tinymce.widgets module

This TinyMCE widget was copied and extended from this code by John D'Agostino: http://code.djangoproject.com/wiki/CustomWidgetsTinyMCE

class dash.lib.tinymce.widgets.**AdminTinyMCE**(*attrs=None*)
Bases: django.contrib.admin.widgets.AdminTextareaWidget, *dash.lib.tinymce.widgets.TinyMCE*

Admin TinyMce.

media

class dash.lib.tinymce.widgets.**TinyMCE**(*content_language=None*, *attrs=None*, *mce_attrs=None*)

Bases: django.forms.widgets.Textarea

TinyMCE widget.

Set settings.TINYMCE_JS_URL to set the location of the javascript file. Default is "MEDIA_URL + 'js/tiny_mce/tiny_mce.js'". You can customize the configuration with the mce_attrs argument to the constructor.

In addition to the standard configuration you can set the 'content_language' parameter. It takes the value of the 'language' parameter by default.

In addition to the default settings from settings.TINYMCE_DEFAULT_CONFIG, this widget sets the 'language', 'directionality' and 'spellchecker_languages' parameters by default. The first is derived from the current Django language, the others from the 'content_language' parameter.

media

Media.

render(*name*, *value*, *attrs=None*, *renderer=None*)

Render.

dash.lib.tinymce.widgets.**get_language_config**(*content_language=None*)

Get language config.

26.3.1.2.1.5 Module contents

26.3.1.2.2 Module contents

26.3.1.3 dash.management package

26.3.1.3.1 Subpackages

26.3.1.3.1.1 dash.management.commands package

26.3.1.3.1.2 Submodules

26.3.1.3.1.3 dash.management.commands.dash_find_broken_dashboard_entries module

26.3.1.3.1.4 dash.management.commands.dash_sync_plugins module

26.3.1.3.1.5 dash.management.commands.dash_update_plugin_data module

26.3.1.3.1.6 Module contents

26.3.1.3.2 Module contents

26.3.1.4 dash.templatetags package

26.3.1.4.1 Submodules

26.3.1.4.2 dash.templatetags.dash_tags module

26.3.1.4.3 Module contents

26.3.2 Submodules

26.3.3 dash.admin module

26.3.4 dash.base module

All *uids* are supposed to be pythonic function names (see PEP http://www.python.org/dev/peps/pep-0008/#function-names).

class dash.base.**BaseDashboardLayout**(*user=None*)

Bases: object

Base layout.

Layouts consist of placeholders.

Properties

- *uid* (string): Layout unique identifier (globally).
- *name* (string): Layout name.

- *description* (string): Layout description.
- *placeholders* (iterable): Iterable (list, tuple or set) of ‘‘dash.base.BaseDashboardPlaceholder‘ subclasses.
- *view_template_name* (string): Template used to render the layout (view).
- *edit_template_name* (string): Template used to render the layout (edit).
- *plugin_widgets_template_name_ajax* (string): Template used to render the plugin widgets popup.
- *form_snippet_template_name* (string): Template used to render the forms.
- *html_classes* (string): Extra HTML class that layout should get.
- *cell_units* (string):
- *media_css* (list): List all specific stylesheets.
- *media_js* (list): List all specific javascripts.

add_dashboard_entry_ajax_template_name = None

add_dashboard_entry_template_name = None

cell_units = None

collect_widget_media(*dashboard_entries*)
Collect the widget media files.

Parameters **dashboard_entries** (*iterable*) – Iterable of `dash.models.DashboardEntry` instances.

Return list

create_dashboard_workspace_ajax_template_name = None

create_dashboard_workspace_template_name = None

description = None

edit_dashboard_entry_ajax_template_name = None

edit_dashboard_entry_template_name = None

edit_dashboard_workspace_ajax_template_name = None

edit_dashboard_workspace_template_name = None

edit_template_name = None

edit_template_name_ajax = None

form_snippet_template_name = 'dash/snippets/generic_form_snippet.html'

get_css(*placeholders*)
Get placeholder specific css.

Parameters **placeholders** (*iterable*) – Iterable of `dash.base.BaseDashboardPlaceholder` subclassed instances.

Return string

get_edit_template_name(*request=None*)

get_grouped_dashboard_entries(*dashboard_entries*)
Get dashboard entries grouped by placeholder.

Parameters **dashboard_entries** (*iterable*) – Iterable of `dash.models.DashboardEntry` objects.

Return list

get_media_css()
Get all CSS media files (for the layout + plugins).

Return list

get_media_js()
Get all JavaScript media files (for the layout + plugins).

Return list

get_placeholder(*uid*, *default=None*)

get_placeholder_instances(*dashboard_entries=None*, *workspace=None*, *request=None*)
Get placeholder instances.

Parameters

- **dashboard_entries** (*iterable*) – Iterable of `dash.models.DashboardEntry` objects.
- **workspace** (*str*) –
- **request** (*django.http.HttpRequest*) –

Return list List of *dash.base.BaseDashboardPlaceholder* subclassed instances.

get_placeholder_uids(*request=None*)
Get the list of placeholder uids.

Parameters **request** (*django.http.HttpRequest*) –

Return list

get_placeholders(*request=None*)
Get the list of placeholders registered for the layout.

Parameters **request** (*django.http.HttpRequest*) –

Return iterable List of placeholder classes. Override in your layout if you need a custom behaviour.

get_view_template_name(*request=None*, *origin=None*)
Get the view template name.

Parameters

- **request** (*django.http.HttpRequest*) –
- **origin** (*string*) – Origin of the request. Hook to provide custom templates for apps. Example value: 'public_dashboard'. Take the *public_dashboard* app as example.

html_class
Class used in the HTML.

Return string

html_classes = []

media_css = []

media_js = []

name = None

placeholders = []

plugin_widgets_template_name_ajax = 'dash/plugin_widgets_ajax.html'

primary_html_class

render_for_edit(*dashboard_entries=None*, *workspace=None*, *request=None*)
Render the layout.

NOTE: This is not used at the moment. You most likely want the `dash.views.edit_dashboard` view.

Parameters

- **dashboard_entries** (*iterable*) –
- **workspace** (*string*) – Current workspace.
- **request** (*django.http.HttpRequest*) –

Return string

render_for_view(*dashboard_entries=None*, *workspace=None*, *request=None*)
Render the layout.

NOTE: This is not used at the moment. You most likely want the `dash.views.dashboard` view.

Parameters

- **dashboard_entries** (*iterable*) –
- **workspace** (*string*) – Current workspace.
- **request** (*django.http.HttpRequest*) –

Return string

uid = None

view_template_name = None

view_template_name_ajax = None

class dash.base.**BaseDashboardPlaceholder**(*layout*)
Bases: `object`

Base placeholder.

Properties

- *uid* (string): Unique identifier (shouldn't repeat within a single layout).
- *cols* (int): Number of cols in the placeholder.
- *rows* (int): Number of rows in the placeholder.
- *cell_width* (int): Single cell (1x1) width.
- *cell_height* (int): Single cell (1x1) height.
- *cell_margin_top* (int): Top margin of a single cell.
- *cell_margin_right* (int): Right margin of a single cell.
- *cell_margin_bottom* (int): Bottom margin of a single cell.
- *cell_margin_left* (int): Left margin of a single cell.
- *view_template_name* (string): Template to be used for rendering the placeholder in view mode.

- *edit_template_name* (string): Template to be used for rendering the placeholder in edit mode.
- *html_classes* (string): Extra HTML class that layout should get.

cell_height = None

cell_margin_bottom = 0

cell_margin_left = 0

cell_margin_right = 0

cell_margin_top = 0

cell_units
Cell units.

cell_width = None

cols = None

css
CSS styles for the placeholders and plugins.

The placeholder dimensions as well as columns sizes, should be handled here. Since we are in a placeholder and a placeholder has a defined number of rows and columns and each render has just a fixed amount of rows and columns defined, we can render the top left corners generic css classes.

Cells do NOT have margins or paddings. This is essential (since all the plugins are positioned absolutely). If you want to have padding in your plugin widget, specify the *plugin-content-wrapper* class style in your specific layout/theme.

Example

.placeholder .plugin .plugin-content-wrapper { padding: 5px;

}

Return string

edit_template_name = ''

get_cell_height()
Get a single cell height, with respect to margins.

Return int

get_cell_width()
Get a single cell width, with respect to margins.

Return int

get_edit_template_name()

get_view_template_name()

html_class
Class used in the HTML.

Return string

html_classes = []

html_id
ID (unique) used in the HTML.

Return string

load_dashboard_entries(*dashboard_entries=None*)
Feed the dashboard entries to the layout for rendering later.

Parameters **dashboard_entries** (*iterable*) – Iterable of `dash.models.DashboardEntry` objects.

primary_html_class
Primary HTML class.

render_for_edit()
Render the placeholder for edit mode.

Return string

render_for_view()
Render the placeholder for view mode.

Return string

rows = None

uid = None

view_template_name = ''

widget_inner_height(*rows*)
The inner height of the widget to be rendered.

Return int

widget_inner_width(*cols*)
The inner width of the widget to be rendered.

class dash.base.**BaseDashboardPlugin**(*layout_uid*, *placeholder_uid*, *workspace=None*, *user=None*, *position=None*)
Bases: `object`

Base dashboard plugin from which every plugin should inherit.

Properties

- *uid* (string): Plugin uid (obligatory). Example value: 'dummy', 'wysiwyg', 'news'.
- *name* (string): Plugin name (obligatory). Example value: 'Dummy plugin', 'WYSIWYG', 'Latest news'.
- *description* (string): Plugin description (optional). Example value: 'Dummy plugin used just for testing'.
- *help_text* (string): Plugin help text (optional). This text would be shown in `dash.views.add_dashboard_entry`. and `dash.views.edit_dashboard_entry` views.
- *form*: Plugin form (optional). A subclass of `django.forms.Form`. Should be given in case plugin is configurable.
- *add_form_template* (str) (optional): Add form template (optional). If given, overrides the `dash.views.add_dashboard_entry` default template.
- *edit_form_template* (string): Edit form template (optional). If given, overrides the `dash.views.edit_dashboard_entry` default template.
- *html_classes* (list): List of extra HTML classes for the plugin.
- *group* (string): Plugin are grouped under the specified group. Override in your plugin if necessary.

add_form_template = None

clone_plugin_data(*dashboard_entry*)
Clone plugin data.

Used when copying entries. If any objects or files are created by plugin, they should be cloned.

Parameters **dashboard_entry** (*dash.models.DashboardEntry*) – Instance of dash.models.DashboardEntry.

Return string JSON dumped string of the cloned plugin data. The returned value would be inserted as is into the dash.models.DashboardEntry.plugin_data field.

delete_plugin_data()
Delete plugin data.

Used in dash.views.delete_dashboard_entry. Fired automatically, when dash.models.DashboardEntry object is about to be deleted. Make use of it if your plugin creates database records or files that are not monitored externally but by dash only.

description = None

edit_form_template = None

form = None

get_cloned_plugin_data(*update={}*)
Get the cloned plugin data and returns it in a JSON dumped format.

Parameters **update** (*dict*) –

Return string JSON dumped string of the cloned plugin data.

Example

In the get_cloned_plugin_data method of your plugin, do as follows:

```
>>> def clone_plugin_data(self, dashboard_entry):
>>>     cloned_image = clone_file(self.data.image, relative_path=True)
>>>     return self.get_cloned_plugin_data(
>>>         update={'image': cloned_image}
>>>     )
```

get_form()
Get the plugin form class.

Override this method in your subclassed dash.base.DashboardPlugin class when you need your plugin setup to vary depending on the placeholder, workspace, user or request given. By default returns the value of the form attribute defined in your plugin.

Return django.forms.Form|django.forms.ModelForm Subclass of django.forms.Form or django.forms.ModelForm.

get_initialised_create_form(*data=None*, *files=None*)
Used dash.views.add_dashboard_entry view to gets initialised form for object to be created.

get_initialised_create_form_or_404(*data=None*, *files=None*)
Get initialised create form or 404.

Same as get_initialised_create_form but raises django.http.Http404 on errors.

get_initialised_edit_form(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*, *instance=None*)
Get initialised edit form.

Used in dash.views.edit_dashboard_entry view.

get_initialised_edit_form_or_404(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*)

Get initialised edit form or 404.

Same as `get_initialised_edit_form` but raises `django.http.Http404` on errors.

get_instance()

Get instances.

get_plugin_form_data()

Get plugin form data.

Fed as `initial` argument to the plugin form when initialising the instance for adding or editing the plugin. Override in your plugin class if you need customisations.

get_position()

Get the exact position of the plugin widget in the placeholder (row number, col number).

Return tuple Tuple of row and col numbers.

get_updated_plugin_data(*update={}*)

Get the plugin data and returns it in a JSON dumped format.

Parameters **update** (*dict*) –

Return string JSON dumped string of the cloned plugin data.

get_widget(*request=None*, *as_instance=False*)

Get the plugin widget.

Parameters

- **request** (*django.http.HttpRequest*) –
- **as_instance** (*bool*) –

Return mixed Subclass of `dash.base.BaseDashboardPluginWidget` or instance of subclassed `dash.base.BaseDashboardPluginWidget` object.

group

help_text = None

html_class

HTML class.

A massive work on positioning the plugin and having it to be displayed in a given width is done here. We should be getting the plugin widget for the plugin given and based on its' properties (static!) as well as on plugin position (which we have from model), we can show the plugin with the exact class.

html_classes = []

html_id

HTML id.

load_plugin_data(*plugin_data*)

Load the plugin data saved in `dash.models.DashboardEntry`. Plugin data is saved in JSON string.

Parameters **plugin_data** (*string*) – JSON string with plugin data.

name = None

post_processor()
Post-process data.

Redefine in your subclassed plugin when necessary.

Post process plugin data here (before rendering). This method is being called after the data has been loaded into the plugin.

Note, that request (django.http.HttpRequest) is available (self.request).

pre_processor()
Pre-process data.

Redefine in your subclassed plugin when necessary.

Pre process plugin data (before rendering). This method is being called before the data has been loaded into the plugin.

Note, that request (django.http.HttpRequest) is available (self.request).

process(*plugin_data=None*, *fetch_related_data=False*)
Init plugin with data.

process_plugin_data(*fetch_related_data=False*)
Processes the plugin data.

render(*request=None*)
Render the plugin HTML (for dashboard workspace).

Parameters **request** (*django.http.HttpRequest*) –

Return string

save_plugin_data(*dashboard_entry*, *plugin_data*)
Save plugin data.

Used in bulk update plugin data.

Parameters

- **dashboard_entry** (*dash.models.DashboardEntry*) –
- **plugin_data** (*dict*) –

Return bool True if all went well.

uid = None

update_plugin_data(*dashboard_entry*)
Update plugin data.

Used in `dash.management.commands.dash_update_plugin_data`.

Some plugins would contain data fetched from various sources (models, remote data). Since dashboard entries are by definition loaded extremely much, you are advised to store as much data as possible in `plugin_data` field of `dash.models.DashboardEntry`. Some externally fetched data becomes invalid after some time and needs updating. For that purpose, in case if your plugin needs that, redefine this method in your plugin. If you need your data to be periodically updated, add a cron-job which would run `dash_update_plugin_data` management command (see `dash.management.commands.dash_update_plugin_data` module).

Parameters **dashboard_entry** (*dash.models.DashboardEntry*) – Instance of `dash.models.DashboardEntry`.

Return dict Should return a dictionary containing data of fields to be updated.

class dash.base.**BaseDashboardPluginWidget**(*plugin*)

Bases: object

Base plugin widget.

So, if we would want to register a plugin widget (renderer) for some layout, we would first define the plugin widget and then just write:

```
>>> plugin_widget_registry.register(DummyPluginWidget)
```

Plugin widget is always being registered for a placeholder. Placeholder in its' turn has number of rows and columns. Since we register each widget for a (layout, placeholder, plugin) combination separately, it fits the needs and requirements perfectly. In that way we are able to tell, whether plugin has a widget available and actually valid (qua dimensions) for the placeholder. Plugin is just data. Nothing more. Widget operates with that data. Thus, widget has number of rows and columns it occupies in the placeholder registered. By default, number of rows and columns is set to 1, which means that a plugin occupies just 1 cell. But, certainly, there can be plugins that occupy more space in a placeholder.

cols = 1

get_height()

Get widget height.

Return int

get_size(*delta_width=0*, *delta_height=0*)

Get widget size.

Parameters

- **delta_width** (*int*) –
- **delta_height** (*int*) –

Return tuple

get_width()

Get widget width.

Return int

html_class = ''

html_classes = []

layout_uid = None

media_css = []

media_js = []

placeholder_uid = None

plugin_uid = None

render(*request=None*)

Render.

Parameters **request** (*django.http.HttpRequest*) –

Return str

rows = 1

class dash.base.**DashboardPluginFormBase**
Bases: object

Not a form actually. Defined for magic only.

Property iterable plugin_data_fields Fields to get when calling the get_plugin_data method. These field will be JSON serialized. All other fields, even if they are part of the form, won't be. Make sure all fields are serializable. If some of them aren't, override the save_plugin_data method and make them serializable there. See dash.contrib.plugins.image.forms as a good example.

Example

```
>>> plugin_data_fields = (
>>>     ('name', ''),
>>>     ('active': False)
>>> )
```

get_plugin_data(*request=None*)
Data that would be saved in the plugin_data field of the dash.models.DashboardEntry subclassed model.

Parameters **request** (*django.http.HttpRequest*) –

plugin_data_fields = None

save_plugin_data(*request=None*)
Dummy, but necessary.

class dash.base.**PluginWidgetRegistry**
Bases: object

Registry of dash plugins widgets (renderers).

get(*uid*, *default=None*)
Get the given entry from the registry.

Parameters

- **uid** (*string*) –
- **default** (*mixed*) –

:return mixed.

static namify(*layout*, *placeholder*, *plugin_uid*)

register(*cls*, *force=False*)
Register the plugin renderer in the registry.

Parameters **cls** (*dash.base.BasePluginRenderer*) – Subclass of dash.base.BasePluginRenderer.

type
alias of *BaseDashboardPluginWidget*

unregister(*cls*)

dash.base.**collect_widget_media**(*dashboard_entries*)
Collect the widget media for dashboard entries given.

Parameters **dashboard_entries** (*iterable*) – Iterable of dash.models.DashboardEntry instances.

Return dict Returns a dict containing the 'js' and 'css' keys. Correspondent values of those keys are lists containing paths to the CSS and JS media files.

dash.base.**ensure_autodiscover**()
Ensure that plugins are auto-discovered.

dash.base.**get_layout**(*layout_uid=None*, *as_instance=False*)
Gets the layout by `layout_uid` given.

If left empty, takes the default one chosen in settings.

Raises a `dash.exceptions.NoActiveLayoutChosen` when no default layout could be found.

Return dash.base.BaseDashboardLayout Sublcass of `dash.base.BaseDashboardLayout`.

dash.base.**get_registered_layout_uids**()
Get uids of registered layouts.

dash.base.**get_registered_layouts**()
Get registered layouts.

dash.base.**get_registered_plugin_uids**()
Gets a list of registered plugin uids as a list .

If not yet auto-discovered, auto-discovers them.

Return list

dash.base.**get_registered_plugins**()
Get a list of registered plugins in a form if tuple.

Get a list of registered plugins in a form if tuple (plugin name, plugin description). If not yet auto-discovered, auto-discovers them.

Return list

dash.base.**validate_placeholder_uid**(*layout*, *placeholder_uid*)
Validate the placeholder.

Parameters

- **layout** (*string*) –
- **placeholder_uid** (*string*) –

Return bool

dash.base.**validate_plugin_uid**(*plugin_uid*)
Validate the plugin uid.

Parameters **plugin_uid** (*string*) –

Return bool

26.3.5 dash.clipboard module

26.3.6 dash.compat module

26.3.7 dash.conf module

dash.conf.**get_setting**(*setting*, *override=None*)
Get setting.

Get a setting from `dash` conf module, falling back to the default.

If override is not None, it will be used instead of the setting.

Parameters

- **setting** – String with setting name
- **override** – Value to use when no setting is available. Defaults to None.

Returns Setting value.

26.3.8 dash.constants module

26.3.9 dash.decorators module

dash.decorators.**all_permissions_required**(*perms*, *login_url=None*, *raise_exception=False*)

All permissions required.

Example

```
>>> @login_required
>>> @all_permissions_required([
>>>     'dash.add_dashboardentry',
>>>     'dash.change_dashboardentry',
>>>     'dash.delete_dashboardentry',
>>>     'dash.add_dashboardworkspace',
>>>     'dash.change_dashboardworkspace',
>>>     'dash.delete_dashboardworkspace',
>>>     'dash.add_dashboardsettings',
>>>     'dash.change_dashboardsettings',
>>>     'dash.delete_dashboardsettings',
>>> ])
>>> def edit_dashboard(request):
>>>     # your code
```

dash.decorators.**any_permission_required**(*perms*, *login_url=None*, *raise_exception=False*)

Any permission required.

Example

```
>>> @login_required
>>> @any_permission_required([
>>>     'dash.add_dashboardentry',
>>>     'dash.change_dashboardentry',
>>>     'dash.delete_dashboardentry',
>>>     'dash.add_dashboardworkspace',
>>>     'dash.change_dashboardworkspace',
>>>     'dash.delete_dashboardworkspace',
>>>     'dash.add_dashboardsettings',
>>>     'dash.change_dashboardsettings',
>>>     'dash.delete_dashboardsettings',
>>> ])
>>> def edit_dashboard(request):
>>>     # your code
```

dash.decorators.**edit_dashboard_permission_required**(*login_url=None*, *raise_exception=False*)

Check if user has permissions to edit dashboard.

Simply, check is successful if any of the following permission checks are satisfied:

- Can add dashboard entry
- Can change dashboard entry
- Can delete dashboard entry
- Can add dashboard workspace
- Can change dashboard workspace
- Can delete dashboard workspace
- Can add dashboard settings
- Can change dashboard settings
- Can delete dashboard settings

Example

```
>>> @login_required
>>> @edit_dashboard_permission_required() # Do not forget the brackets!
>>> def edit_dashboard(request):
>>>     # your code
```

dash.decorators.**permissions_required**(*perms*, *satisfy='all'*, *login_url=None*, *raise_exception=False*)

Check for the permissions given based on the strategy chosen.

Parameters

- **perms** (*iterable*) –
- **satisfy** (*string*) – Allowed values are "all" and "any".
- **login_url** (*string*) –
- **raise_exception** (*bool*) – If set to True, the `PermissionDenied` exception is raised on failures.

Return bool

Example

```
>>> @login_required
>>> @permissions_required(satisfy='any', perms=[
>>>     'dash.add_dashboardentry',
>>>     'dash.change_dashboardentry',
>>>     'dash.delete_dashboardentry',
>>>     'dash.add_dashboardworkspace',
>>>     'dash.change_dashboardworkspace',
>>>     'dash.delete_dashboardworkspace',
>>>     'dash.add_dashboardsettings',
>>>     'dash.change_dashboardsettings',
>>>     'dash.delete_dashboardsettings',
>>> ])
>>> def edit_dashboard(request):
>>>     # your code
```

dash.decorators.**use_clipboard_permission_required**(*login_url=None*, *raise_exception=False*)

Check if user has permissions to use clipboard.

Simply, check is successful if all of the following permission checks are satisfied:

- Can add dashboard entry
- Can delete dashboard entry

Example

```
>>> @login_required
>>> @use_clipboard_permission_required() # Do not forget the brackets!
>>> def cut_dashboard_entry(request):
>>>     # your code
```

26.3.10 dash.defaults module

26.3.11 dash.discover module

dash.discover.**autodiscover**()
Auto-discovers files that should be found by dash.

26.3.12 dash.exceptions module

exception dash.exceptions.**BaseException**
Bases: Exception

Base django-dash exception.

exception dash.exceptions.**ImproperlyConfigured**
Bases: *dash.exceptions.BaseException*

Raised when django-dash is somehow improperly configured.

exception dash.exceptions.**InvalidRegistryItemType**
Bases: ValueError

Raised when an attempt is made to register an item of improper type.

Raised when an attempt is made to register an item of improper type in the registry .

exception dash.exceptions.**LayoutDoesNotExist**
Bases: *dash.exceptions.BaseException*

Raised when layout does not exist.

exception dash.exceptions.**NoActiveLayoutChosen**
Bases: *dash.exceptions.BaseException*

Raised when no active layout is chosen.

exception dash.exceptions.**PluginWidgetOutOfPlaceholderBoundaries**
Bases: *dash.exceptions.BaseException*

Raised when plugin widget is out of placeholder boundaries.

26.3.13 dash.factory module

dash.factory.**plugin_factory**(*base_class*, *plugin_uid_prefix*, *sizes=[]*)

Plugin factory.

Parameters

- **base_class** (*class*) – Subclass of.
- **base_class** –
- **plugin_uid_prefix** (*string*) –
- **sizes** (*iterable*) –
- **sizes** – Iterable of tuples.

Example

```
>>> from dash.contrib.plugins.image.dash_plugins import BaseImagePlugin
>>> plugin_factory(
>>>     BaseImagePlugin, 'image', zip(range(6, 10), range(6, 10))
>>> )
```

The example above will update the plugin registry with the following dictionary: >>> { >>> 'image_6x6': dash.factory.Plugin, >>> 'image_7x7': dash.factory.Plugin, >>> 'image_8x8': dash.factory.Plugin, >>> 'image_9x9': dash.factory.Plugin, >>> }

The generated class (one of them), would look as follows:

```
>>> class Plugin(BaseImagePlugin):
>>>     uid = 'image_6x6'
```

The `uid` property is generated automatically.

dash.factory.**plugin_widget_factory**(*base_class*, *layout_uid*, *placeholder_uid*, *plugin_uid_prefix*, *sizes=[]*)

Plugin widget factory.

Parameters

- **base_class** (*dash.base.BaseDashboardWidget*) – Subclass of.
- **layout_uid** (*string*) – Layout UID, for which widgets are generated.
- **placeholder_uid** (*string*) – Placeholder UID, for which widgets are generated.
- **plugin_uid_prefix** (*string*) – Prefix of the plugin UID.
- **sizes** (*iterable*) – Iterable of tuples.

Example

```
>>> from dash.contrib.plugins.image.dash_widgets import BaseImageWidget
>>> plugin_widget_factory(
>>>     BaseImageWidget,
>>>     'android',
>>>     'main',
>>>     'image',
>>>     zip(range(6, 10), range(6, 10))
>>> )
```

The example above will update the plugin widget registry with the following dictionary:

```
>>> {
>>>     'android.main.image_6x6': dash.factory.Widget,
>>>     'android.main.image_7x7': dash.factory.Widget,
>>>     'android.main.image_8x8': dash.factory.Widget,
>>>     'android.main.image_9x9': dash.factory.Widget,
>>> }
```

The generated class (one of them), would look as follows:

```
>>> class Widget(BaseImageWidget):
>>>     layout_uid = 'android'
>>>     placeholder_uid = 'main'
>>>     plugin_uid = 'image_6x6'
>>>     cols = 6
>>>     rows = 6
```

The `layout_uid`, `placeholder_uid`, `plugin_uid`, `cols` and `rows` properties are generated automatically.

26.3.14 dash.fields module

class dash.fields.**OrderField**(*verbose_name=None*, *name=None*, *primary_key=False*, *max_length=None*, *unique=False*, *blank=False*, *null=False*, *db_index=False*, *rel=None*, *default=<class 'django.db.models.fields.NOT_PROVIDED'>*, *editable=True*, *serialize=True*, *unique_for_date=None*, *unique_for_month=None*, *unique_for_year=None*, *choices=None*, *help_text=''*, *db_column=None*, *db_tablespace=None*, *auto_created=False*, *validators=()*, *error_messages=None*)

Bases: `django.db.models.fields.IntegerField`

OrderField.

@author http://djangosnippets.org/users/zenx/ @source http://djangosnippets.org/snippets/1861/

OrderField for models from //ianonpython.blogspot.com/2008/08/orderfield-for-django-models.html and updated to use a django aggregation function. This field sets a default value as an auto-increment of the maximum value of the field +1.

Ignores the incoming value and instead gets the maximum plus one of the field.

This works really well in combination with "sortable_list.js". There are several things you should know:

- order field shall be null=True, blank=True
- order field shall not be unique

If above mentioned is True, you can use jQuery drag-n-drop widget in your Django-admin.

See the following example:

```
>>> class Media:  # This belongs to your Admin class (admin.ModelAdmin)
>>>     js = [
>>>         '/media/js/jquery-1.6.2.min.js',
>>>         '/media/js/jquery-ui-1.8.16.custom.min.js',
>>>         '/media/js/sortable_list.js'
>>>     ]
```

formfield_(***kwargs*)

pre_save(*model_instance*, *add*)
Return field's value just before saving.

26.3.15 dash.forms module

26.3.16 dash.helpers module

dash.helpers.**clean_plugin_data**(*dashboard_entries*, *request=None*)
Clean up the plugin data.

Clean up the plugin data (database, files) for the dashboard_entries given.

Parameters

- **dashboard_entries** (*iterable*) –
- **request** (*django.http.HttpRequest*) –

Return bool Boolean True if no errors occurred and False otherwise.

dash.helpers.**clone_plugin_data**(*dashboard_entry*, *request=None*)
Clone plugin data of a dashboard entry.

dash.helpers.**iterable_to_dict**(*items*, *key_attr_name*)
Convert iterable of certain objects to dict.

Parameters

- **items** (*iterable*) –
- **key_attr_name** (*string*) – Attribute to use as a dictionary key.

Return dict

dash.helpers.**lists_overlap**(*sub*, *main*)

dash.helpers.**safe_text**(*text*)
Safe text (encode).

Return str

dash.helpers.**slugify_workspace**(*val*)
Slugify workspace.

dash.helpers.**uniquify_sequence**(*sequence*)
Makes the sequence unique.

Makes sure items in the given sequence are unique, having the original order preserved.

Parameters **sequence** (*iterable*) –

Return list

dash.helpers.**update_plugin_data**(*dashboard_entry*, *request=None*)
Update plugin data of a dashboard entry.

26.3.17 dash.json_package module

26.3.18 dash.models module

26.3.19 dash.settings module

- *RESTRICT_PLUGIN_ACCESS* (bool): If set to True, (Django) permission system for dash plugins is enabled.
- *PLUGINS_MODULE_NAME* (str): Name of the module to placed in the (external) apps in which the dash plugin code should be implemented and registered.
- *ACTIVE_LAYOUT* (str): Active layout UID.
- *LAYOUTS_MODULE_NAME* (str): Name of the python module to be placed in (external) apps in which the dash layouts should be implemented and registered.
- *DEFAULT_WORKSPACE_NAME* (str): Name of the default workspace.
- *DEFAULT_PLACEHOLDER_VIEW_TEMPLATE_NAME* (str): Default template name for the placeholder view.
- *DEFAULT_PLACEHOLDER_EDIT_TEMPLATE_NAME* (str): Default template name for the placeholder edit.
- *LAYOUT_CELL_UNITS* (str): Layout cell units. Allowed values are *em*, *px*, *pt*, *%*.
- *DISPLAY_AUTH_LINK* (bool): If set to True, the log in or log out link is shown in the Dash drop-down menu.
- *DEBUG* (bool)

26.3.20 dash.tests module

26.3.21 dash.urls module

26.3.22 dash.utils module

26.3.23 dash.views module

26.3.24 dash.widgets module

class dash.widgets.**BooleanRadioSelect**(**args*, ***kwargs*)

Bases: `django.forms.widgets.RadioSelect`

Boolean radio select for Django.

Example

```
>>> class DummyForm(forms.Form):
>>>     agree = forms.BooleanField(label=_("Agree?"), required=False,
>>>                                widget=BooleanRadioSelect)
```

media

26.3.25 Module contents

CHAPTER 27

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

C

D

E

F

G

M

N

O

P

R

S

T

U

V

W