django-dashing Documentation

Release 0.2

Mauricio Reyes

May 26, 2015

Contents

django-dashing is a customisable, modular dashboard application framework for Django to visualize interesting data about your project. Inspired in the exceptionally handsome dashboard framework Dashing

CHAPTER 1

Prerequisites

- Django 1.5.+

CHAPTER 2

Key concepts

- Use premade widgets, or fully create your own with css, html, and javascript.
- Use the API to push data to your dashboards.
- Drag & Drop interface for re-arranging your widgets.

CHAPTER 3

Topics

General guides to using Django Dashing.

3.1 Getting Started

3.1.1 Installation

1. Install the latest stable version from PyPi:

```
$ pip install django-dashing
```

2. Add *dashing* to `INSTALLED_APPS` of the your projects.

```
INSTALLED_APPS = (
    ...
    'dashing',
)
```

3. Include the polls URLconf in your project urls.py like this:

```
from dashing.utils import router
...

url(r'^dashboard/', include(router.urls)),
```

4. Start the development server and visit http://127.0.0.1:8000/dashboard/ to view the dummy dashboard.

To create a custom dashboard you need create a `dashing-config.js` file in the static directory and optionally a custom `dashing/dashboard.html` template file.

3.1.2 Django Settings

Configuration for Django Dashing is all namespaced inside a single Django setting, named DASHING.

For example your project's `settings.py` file might include something like this:

```
DASHING = {
    'INSTALLED_WIDGETS': ('number', 'list', 'graph',),
    'PERMISSION_CLASSES':  (
        'dashing.permissions.IsAuthenticated',
    )
}
```

Accessing settings

If you need to access the values of Django Dashing settings in your project, you should use the dashing_settings object. For example:

```
from dashing.settings import dashing_settings

print dashing_settings.INSTALLED_WIDGETS
```

Settings

INSTALLED_WIDGETS

A list or tuple of name of widgets to load when the dashboard is displayed, searches for resources of widgets (js, css and html) in the static directory, if not found then search in the remote repository

Default:

```
('number', 'list', 'graph', 'clock',),
```

PERMISSION_CLASSES

A list or tuple of permission classes, that determine the default set of permissions checked when displaying the dashboard.

The default permissions classes provided are: *AllowAny*, *IsAuthenticated*, and *IsAdminUser*

Default:

```
('dashing.permissions.AllowAny',)
```

REPOSITORY

A remote location with a repositories.json file, here are specified the third-party widgets with the remote location to download the static files

3.1.3 Config File

You need put the `dashing-config.js` in the static directory to begin creating widgets for your project. You can change the patch and name if you write a template file.

The dashing config file should start with the creation of a new dashboard `var dashboard = new Dashboard();` and start to place widgets with the following syntax `dashboard.addWidget(<name_of_widget>, <type_of_widget>, <options>);` where *name_of_widget* is the name that describe the objective of the widget (should be unique) *type_of_widget* is a valid widget type (Clock, Graph, List, Number) and options depends of each widget.

This is the default `dashing-config.js` file, use as a guide for writing your own:

```
/* global $, Dashboard */

var dashboard = new Dashboard();

dashboard.addWidget('clock_widget', 'Clock');

dashboard.addWidget('current_valuation_widget', 'Number', {
    getData: function () {
        $.extend(this.scope, {
            title: 'Current Valuation',
```

```
            moreInfo: 'In billions',
            updatedAt: 'Last updated at 14:10',
            detail: '64%',
            value: '$35'
        });
    }
});

dashboard.addWidget('buzzwords_widget', 'List', {
    getData: function () {
        $.extend(this.scope, {
            title: 'Buzzwords',
            moreInfo: '# of times said around the office',
            updatedAt: 'Last updated at 18:58',
            data: [{label: 'Exit strategy', value: 24},
                   {label: 'Web 2.0', value: 12},
                   {label: 'Turn-key', value: 2},
                   {label: 'Enterprise', value: 12},
                   {label: 'Pivoting', value: 3},
                   {label: 'Leverage', value: 10},
                   {label: 'Streamlininess', value: 4},
                   {label: 'Paradigm shift', value: 6},
                   {label: 'Synergy', value: 7}]
        });
    }
});

dashboard.addWidget('convergence_widget', 'Graph', {
    getData: function () {
        $.extend(this.scope, {
            title: 'Convergence',
            value: '41',
            moreInfo: '',
            data: [
                    { x: 0, y: 40 },
                    { x: 1, y: 49 },
                    { x: 2, y: 38 },
                    { x: 3, y: 30 },
                    { x: 4, y: 32 }
                ]
            });
    }
});
```

3.1.4 Template File

You can create a *dashboard.html* file to add your custom stylesheets and scripts or specify a custom route to your dashing-config.js file. You will place inside the template directory in `dashing/dashboard.html`

Your `dashing/dashing.html` might looks like this:

```
{% extends 'dashing/base.html' %}
{% load staticfiles %}

{% block 'stylesheets' %}
<link rel="stylesheet" href="{% static 'my/custom/style.css' %}">
{% endblock %}
```

```
{% block 'scripts' %}
<script type="text/javascript" src="{% static 'my/custom/script.js' %}"></script>
{% endblock %}

{% block 'config_file' %}
<script type="text/javascript" src="{% static 'my/custom/dashing-config.js' %}"></script>
{% endblock %}
```

3.1.5 Python Widget Classes

Django Dashing provides a useful set of classes to return the expected data for the default widgets, you can create a *widgets.py* file and inherit of these classes or create your own widgets inherit from `dashing.widgets.Widget`.

A custom widget can look like this:

```
class CustomWidget(NumberWidget):
    title = 'My Custom Widget'
    value = 25

    def get_more_info(self):
        more_info = 'Random additional info'
        return more_info
```

To register the url to serve this widget you must use the register method from `dashing.utils.router`, then in *urls.py* file put

```
from dashing.utils import router

router.register(CustomWidget, 'custom_widget', eg_kwargs_param="[A-Za-z0-9_-]+")
```

Now we can access to CustomWidget from '/dashboard/widgets/custom_widget/(?P<eg_kwargs_param>[A-Za-z0-9_-]+)' if '/dashboard/' is the root of our dashboard.

The kwargs are optional and you can add as many as you want.

3.2 Dashboards

3.2.1 Single Dashboard

To initialize a single dashboard you need to create a Dashboard object and pass valid options as shown below:

```
var dashboard = new Dashboard(options);
```

Where the *options* are a json object with the following specifications

Options

name ***(optional)*** The name of widget. *(default:* `undefined` *)*

viewportWidth ***(optional)*** Width of viewport where expected that the dashboard was displayed. *(default:* `$(window).width()` *)*

viewportHeight ***(optional)*** Height of viewport where expected that the dashboard was displayed. *(default:* `$(window).height()` *)*

widgetMargins ***(optional)*** Margin between each widget. *(default:* `[5, 5]` *)*

widgetBaseDimensions ***(optional)*** Default width and height of each widget in the dashboard. *(default:* `[370, 340]` *)*

Dashboard methods

name Return the name of the dashboard or **'unnamed'**.

show Show the dashboard (if it's hidden) and publish the event **shown** for this dashboard.

hide Show the dashboard (if it's showing) and publish the event **hidden** for this dashboard.

grid Return the *gridster* element for this dashboard.

activeWidgets Return the loaded widgets for this dashboard.

addWidget To add a new Widget, for example:

```
myDashboard.addWidget('myWheaterWidget', 'Weather', {
    WOEID: 395269
});
```

For details you can check the widgets topic

listWidgets Returns the list of widgets created on this dashboard

subscribe To subscribe an event in this dashboard you can do the following:

```
myDashboard.subscribe('myEvent', function() {
    console.log('event fired!');
});
```

publish To publish an event in this dashboard you can do the following:

```
myDashboard.publish('myEvent');
```

3.2.2 Multiple Dashboards

To initialize a multiple dashboard you need to create a DashboardSet object and pass valid options as shown below:

```
var myDashboardSet = new DashboardSet();
```

DashboardSet methods

addDashboard To add a new Dashboard:

```
myDashboardSet.addDashboard(name, options)
```

Where *name* is a string with the name of the dashboard and *options* is a json object with the same format of the options of the *Dashboard* object.

getDashboard To get a Dashboard from the DashboardSet object:

```
myDashboardSet.getDashboard(name)
```

addAction To add a button on the overlay menu that runs arbitrary javascript code, for example:

```
myDashboardSet.addAction('Go to Google', function() {
    window.location.href = 'https://google.com/';
})
```

Swap between dashboards

Manual

To swap between dashboards you need to press the *ctrl* key to display the menu.

Automatic

To swap the dashboards automatically you can set the option *rollingChoices* as *true* when the dashboardSet is created as follows:

```
myDashboardSet = new DashboardSet({
    rollingChoices: true
}),
```

Then you can select the rolling time in the *ctrl* menu. Or you can add the parameter *roll=<value>* to the URL, where the value has to be specified in microseconds, for example:

```
http://127.0.0.1:8000/dashboard/?roll=3000
```

Dashboard Events

Each single dashboard publishes a **shown** or **hidden** event when the dashboard are loaded or unloaded, you can use it as follows:

```
myDashboard = myDashboardSet.addDashboard('New Dashboard');
myDashboard.subscribe('shown', function() {alert('new dashboard shown')});
myDashboard.subscribe('hidden', function() {alert('new dashboard hidden')});
```

3.3 Widgets

To place widgets in your dashboard you need to create a javascript file, where you call each widget that you need to place with the correct options, each widget provides an event that you can call in any javascript file to update the data.

For example if you create a number widget

```
var dashboard = new Dashboard();

...

dashboard.addWidget('current_valuation_widget', 'Number', {
    getData: function () {
        $.extend(this.scope, {
            title: 'Current Valuation',
            moreInfo: 'In billions',
            updatedAt: 'Last updated at 14:10',
            value: '$35',
            detail: '64%'
        });
    }
});
```

Then you can publish at any moment the event `dashboard.publish('example_widget/getData')` to get new data and update the widget.

Note that in this example the *getData* method will be executed each 1000 milliseconds because it is the default value of *interval* option in a *Number* widget.

3.3.1 Clock Widget

This widget can display a specific day or an hour.

Options

row Number of rows occupied by the widget. *(default: 1)*

col Number of columns occupied by the widget. *(default: 1)*

scope JSON object that represent the date and time in format

```
{
    time: 'hh:mm:ss',
    date: 'Month Day DD sYYYY'
}
```

getData Function responsible to update the *scope* value, this function is executed each time interval specified in *interval* variable. You can rewrite this function to get data from an external source. *(default: return the browser time in a valid JSON format)*

getWidget Return the DOM element that represent the widget.

interval Actualization interval of widget scope on milliseconds. *(default: 500)*

3.3.2 Graph Widget

This widget can display a value with an associated graph as a background.

Options

row Number of rows occupied by the widget. *(default: 1)*

col Number of columns occupied by the widget. *(default: 2)*

scope JSON object in this format

```
{
    data: [
            {x: /x0/, y: /y0/},
            {x: /x1/, y: /y1/}
            ...
          ],
    value: /string/,
    title: /string/,
    moreInfo: /string/,
    beforeRender: /function/,
    afterRender: /function/,
    xFormat: /function/,
    yFormat: /function/,
    properties: /object/,
}
```

getData Function responsible to update the widget *scope*, this function is executed each time interval specified in *interval* variable. You can rewrite this function to get data from an external source. *(default: empty function)*

getWidget Return the DOM element that represents the widget.

interval Actualization interval of widget scope on milliseconds. *(default: 1000)*

Graph options

To render the graph this widget use Rickshaw library, for now the config options are quite limited, if you need to be more specific you can overwrite the rivetsjs binder (rv-dashing-graph) or write a custom widget using this as a guide.

To configure the X and Y axis you must define custom methods *xFormat* and *yFormat* in the scope, also you can use the methods *beforeRender* and *afterRender* to execute arbitrary javascript before or after rendering, for example:

```
function xFormat(n) {
    return '(' + n + ')';
};
Dashing.utils.get('my-registered-widget-url-name', function(scope) {
    scope.xFormat = xFormat;
    scope.afterRender = function() {
        alert('graph shown');
    };
    $.extend(self.scope, scope);
});
```

Also, you can specify any properties that the graph constructor accepts in the *scope* object, for example a valid *scope* may be:

```
{
    data: [
        { x: 0, y: 29 },
        { x: 1, y: 42 },
        { x: 2, y: 12 }
    ],
    value: 12,
    title: 'Yeah!',
    moreInfo: 'Django Rocks',
    properties: {
        renderer: 'line',
        padding: {
            top: 0.1,
            right: 0.1
        }
    },
}
```

Python Class

This class helps return valid scope to be used by the widget, you can see the definition in GitHub

Here is an example of a graph widget where *value* is displayed the total number of Errands and in *data* returns an array with the last two hour of activity

```
from dashing.widgets import GraphWidget

class HourlyErrandsWidget(GraphWidget):
    title = 'Hourly Errands'
    more_info = ''

    def get_value(self):
        return SearchQuerySet().filter(django_ct='errands.errand').count()

    def get_data(self):
        latest_hours = datetime.now() - timedelta(hours=2)
```

```
        latest_errands = SearchQuerySet().filter(
                            django_ct='errands.errand',
                            created__gt=latest_hours).values('created')
        intervals = []
        for errand in latest_errands:
            delta = datetime.now() - errand['created']

            for m in range(10, 120, 10):
                if delta < timedelta(minutes=m):
                    intervals.append(13 - m/10)
                    break

        rlist = Counter([x for x in intervals])
        return [{'x': x, 'y': y} for x, y in rlist.most_common()]
```

3.3.3 List Widget

This widget can display a list of elements with an associated value.

Options

row Number of rows occupied by the widget. *(default: 2)*

col Number of columns occupied by the widget. *(default: 1)*

render Function responsible of modify the DOM elements of the widget.

scope JSON object in this format

```
    {
        data: [
                {
                    label: /string/,
                    name: /string/
                },
                {
                    label: /string/,
                    name: /string/
                },
                ...
              ],
        title: /string/,
        moreInfo: /string/,
        updatedAt: /string/
    }
```

getData Function responsible to update the *scope* value, this function is executed each time interval specified in *interval* variable. You can rewrite this function to get data from an external source. *(default: empty function)*

getWidget Return the DOM element that represent the widget.

interval Actualization interval of widget data on milliseconds. *(default: 10000)*

Python Class

This class helps to return valid data to be use by the widget, you can see the definition in GitHub

Here's an example of a graph widget where in the *scope* returns an array with the messengers who have more requests

```
from dashing.widgets import ListWidget

class ActiveMessengersWidget(ListWidget):
    title = 'Active Messengers'
    moreInfo = 'Those who have more requests'

    def get_updated_at(self):
        modified = SearchQuerySet().filter(
            django_ct='errand').order_by('-modified')[0].modified
        return u'Last updated {}'.format(modified)

    def get_data(self):
        messengers = SearchQuerySet().filter(
                                django_ct='messengers', active=True)
        rlist = Counter([x for x in messengers])
        return [{'label':x, 'value':y} for x, y in rlist.most_common(20)]
```

3.3.4 Number Widget

This widget can display a value with other interesting information.

Options

row Number of rows occupied by the widget. *(default: 1)*

col Number of columns occupied by the widget. *(default: 1)*

scope JSON object in this format

```
    {
        value: /string/,
        title: /string/,
        detail: /string/,
        moreInfo: /string/,
        updatedAt: /string/
    }
```

getData Function responsible to update the *scope* value, this function is executed each time interval specified in *interval* variable. You can rewrite this function to get data from an external source. *(default: empty function)*

getWidget Return the DOM element that represent the widget.

interval Actualization interval of widget scope on milliseconds. *(default: 1000)*

Python Class

This class helps to return valid data to be used by the widget, you can see the definition in GitHub

Here is an example of a graph widget where in *value* is displayed the total of payments and in the detail and moreInfo shows other information of interest

```
from dashing.widgets import NumberWidget

class PaymentsWidget(NumberWidget):
    title = 'Payments Customers'
```

```
    def get_value(self):
        return Payment.objects.all().count()

    def get_detail(self):
        payments = Payment.objects.all()
        total = len([x for x in payments if x.status == Payment.STATUS.waiting])
        return '{} to approve'.format(total)

    def get_more_info(self):
        payments = Payment.objects.all()
        total = len([x for x in payments if x.status == Payment.STATUS.rejected])
        return '{} rejected'.format(total)
```

3.4 Custom Widgets

To make a custom widget you must create three static files to define configuration parameters and appearance. In addition, you can create a python class to communicate with the Django project.

To name your widgets should follow a naming convention were the name must be unique and searchable through the settings.

3.4.1 Static Files

Template File

Its location should be `<static_directory>/widgets/<widget_name>/<widget_name>.html` this file describes its UI in plain HTML using the Rivets.js conventions to bind data to the script file.

For example `{% static %}widgets/list/list.html` looks like this:

```
<div>
    <h1>{ scope.title }</h1>
    <ul>
      <li rv-each-item="data">
        <span class="label">{ item.label }</span>
        <span class="value">{ item.value }</span>
      </li>
    </ul>
    <p class="more-info">{ moreInfo }</p>
    <p class="updated-at">{ updatedAt }</p>
</div>
```

The classes are only for the stylesheet.

Style File

Your location should be `<static_directory>/widgets/<widget_name>.css` in this file defines the styles of widget.

Script File

Your location should be `<static_directory>/widgets/<widget_name>.js` in this file will be defined the configuration options and default values for the new widget, the idea is to create an object using the `new` keyword, then we define properties and methods using `this` keyword.

We must provide an `__init__` method were we bind the scope with the template and add to the dashboard, this function is quite similar in all widgets, then it is provided by `Dashing.utils.widgetInit` to facilitate implementation and improve the lecture of widgets, also must provide a `scope` element which will be binded to the template, and a `getData` function will surely be the to be overwritten to obtain relevant data as required,

For example `{% static %}widgets/list/list.js` looks like this:

```
/* global Dashboard */

Dashing.widgets.List = function (dashboard) {
    var self = this,
        widget;
    this.__init__ = Dashing.utils.widgetInit(dashboard, 'list');
    this.row = 2;
    this.col = 1;
    this.scope = {};
    this.getWidget = function () {
        return widget;
    };
    this.getData = function () {};
    this.interval = 10000;
};
```

If we want to initialize widget with the scope we can write:

```
...
    this.col = 1;
    this.socpe = {
        title: 'Default Title',
        moreInfo: 'No data to display'
    };
    this.getWidget = function () {
...
```

3.4.2 Python Class

Surely in many cases it may be necessary to give the option to get some Dajngo project data into the widget, for this dashing has a Widget class that can be inherited to deliver properly serialized data, subsequently serving data using the dashing router.

For example ListWidget in `dashing/widgets.py` looks like this:

```
class ListWidget(Widget):
    title = ''
    more_info = ''
    updated_at = ''
    data = []

    def get_title(self):
        return self.title

    def get_more_info(self):
        return self.more_info
```

```
    def get_updated_at(self):
        return self.updated_at

    def get_data(self):
        return self.data

    def get_context(self):
        return {
            'title': self.get_title(),
            'moreInfo': self.get_more_info(),
            'updatedAt': self.get_updated_at(),
            'data': self.get_data(),
        }
```

If you develop your widget with python classes it is necessary that you distribute it via PyPI

3.4.3 Distribution

To distribute a widget you have two options. The fastest way is through Django Dashing Channel but it is a bit limited, and through PyPI, a bit trickier to pack but you have more options when developing the widget.

Via Django Dashing Channel

Using this distribution method the users will only have to add the widget name on `INSTALLED_WIDGETS` then load the dashboard, this locates the static files from a remote location (specified in the preconfigured repository), if the user creates a copy of the files on your local static directory then these will open locally.

You will have to host your files into a CDN, I recommend creating a github project and use RawGit to serve through MaxCDN, you can take dj-dashing-weather-widget project as a guide.

Finally to publish your widget in Django Dashing Channel you need to make a fork of django-dashing-channel, add your repository to repositories.json and send a pull request. In the repository root will be sought the widget static files (.js .css and .html)

You should create a README file for installation instructions.

PyPI Package

If your widget requires python code or you just want to provide an easy way to get the widget locally then a PyPI package is the way to go.

As a requirement it is necessary follow the widgets naming convention (*see static files*). To create a PyPI package see the documentation, and should create a README file for installations instructions.

This is not excluding the previous way, you could create a minimalist version of your widget and upload it to django-dashing-channel and in the project instructions show how to install the PyPI version