Django-controlcenter Documentation

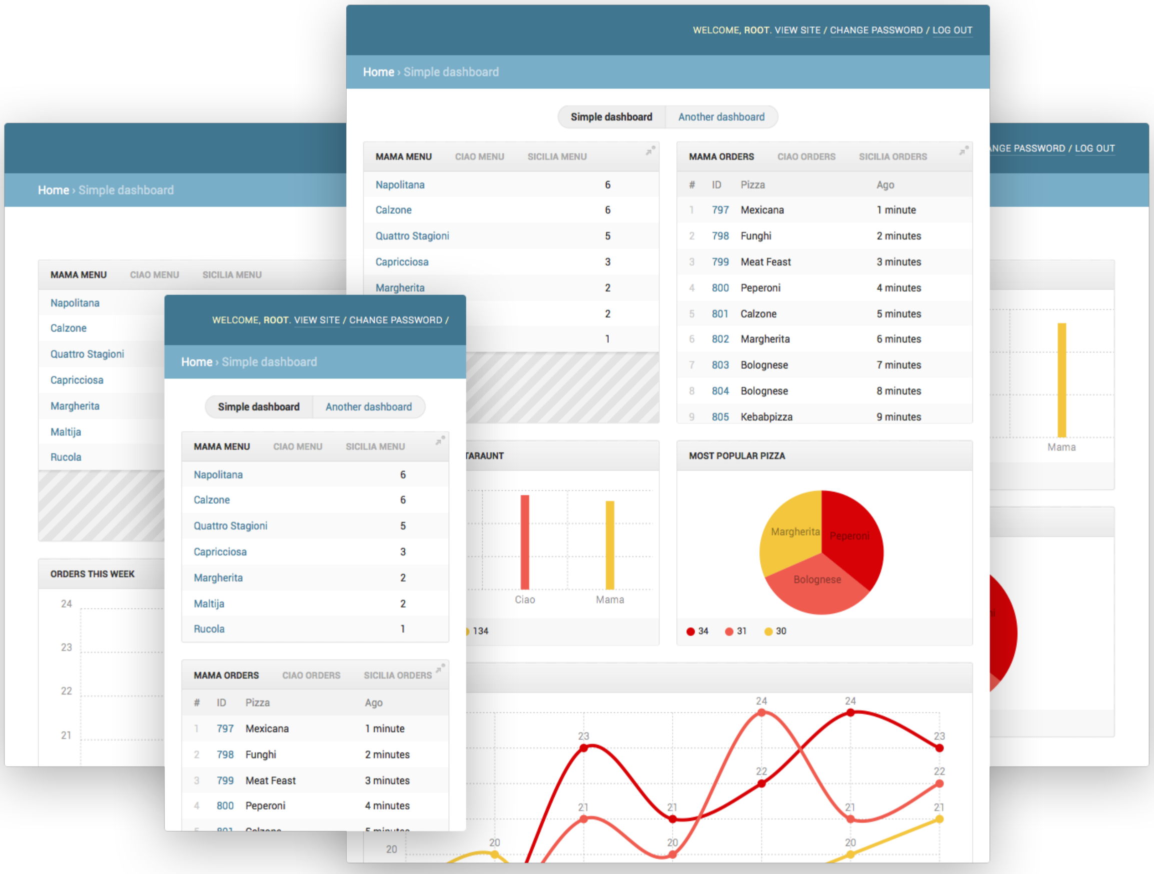
Release 0.2.8

Murad Byashimov

Dec 11, 2018

Contents

Get all your project models on one single page with charts and whistles.



Napolitana	6
Calzone	6
Quattro Stagioni	5
Capricciosa	3
Margherita	2
Maltija	2
Rucola	1

#	ID	Pizza	Ago
1	797	Mexicana	1 minute
2	798	Funghi	2 minutes
3	799	Meat Feast	3 minutes
4	800	Peperoni	4 minutes
5	801	Calzone	5 minutes
6	802	Margherita	6 minutes
7	803	Bolognese	7 minutes
8	804	Bolognese	8 minutes
9	805	Kebabpizza	9 minutes

CHAPTER 1

Rationale

Django-admin is a great tool to control your project activity: new orders, comments, replies, users, feedback – everything is here. The only struggle is to switch between all those pages constantly just to check them out for new entries.

With django-controlcenter you can have all of your models on one single page and build beautiful charts with Chartist.js. Actually they don't even have to be a django models, get your data from wherever you want: RDBMS, NOSQL, text file or even from an external web-page, it doesn't matter.

CHAPTER 2

Quickstart

Install django-controlcenter:

```
pip install -U django-controlcenter
```

Create a dashboard file with unlimited number of widgets and dashboards:

```
from controlcenter import Dashboard, widgets
from project.app.models import Model

class ModelItemList(widgets.ItemList):
    model = Model
    list_display = ('pk', 'field')

class MyDashboard(Dashboard):
    widgets = (
        ModelItemList,
    )
```

Update settings file:

```
INSTALLED_APPS = [
    ...
    'controlcenter',
    ...
]

CONTROLCENTER_DASHBOARDS = (
    ('mydash', 'project.dashboards.MyDashboard'),
)
```

Plug in urls:

```
from django.urls import path
from django.contrib import admin
```

(continues on next page)

(continued from previous page)

```
from controlcenter.views import controlcenter

urlpatterns = [
    path('admin/', admin.site.urls),
    path('admin/dashboard/', controlcenter.urls),
    ...
]
```

Open `/admin/dashboard/mydash/` in browser.

CHAPTER 3

Documentation

Check out the docs for more complete examples.

CHAPTER 4

Compatibility

Tested on py 2.7, 3.4, 3.5, 3.6 with django 1.8—2.1.

CHAPTER 5

Credits

This project uses Chartist.js, Masonry.js and Sortable.js.

CHAPTER 6

Changelog

6.1 0.2.8

- Fixed `key_value_list.html` widget template syntax error.
- Fixed attribute typo `widget.chartist.point_labels -> point_lables`.

Thanks to @minusf.

6.2 0.2.7

- New `TimeSeriesChart` widget. Thanks to @pjdelport.
- New "simple" widgets: `ValueList` and `KeyValueList`. Thanks to @tonysyu.
- Bunch of fixes and improvements, thanks again to @pjdelport.

6.3 0.2.6

- Fixed navigation menu links, thanks to @editorgit

6.4 0.2.5

- It's now possible to use slugs for dashboards instead of those indexes in `CONTROLCENTER_DASHBOARDS`. The old behaviour is supported too.

6.5 0.2.4

- It's compatible with django 1.8—2.1 now
- Custom app name can be passed to `ControlCenter` class

6.6 0.2.3

- Updated column grid, thanks to @pauloxnet.
- Grammar fixes, thanks to @danielquinn.
- It's should be possible now to use a custom dashboard view with a custom template.

6.7 0.2.2

- `dashboard.html` now extends `admin/base_site.html` instead of `admin/base.html` in order to display *branding* block. Thanks to @chadgh.
- Updated `jsonify` tag filter, thanks to @k8n.

6.8 0.2.1

- Django 1.10 support. Tested in tox *only*.
- Updated the SingleBarChart example, thanks to @greeve.

6.9 0.2.0

- Unlimited dashboards support.
- Configuration constructor is moved to a separate project – django-pkgconf. It's a dependency now.

6.10 0.1.2

- Chart `i` series color fix. Thanks to @uncleNight.
- Docs. Finally.

6.11 0.1.1

- Better responsive experience.

6.12 0.1.0

- First public release.

CHAPTER 7

Contents

7.1 Dashboards

Django-controlcenter supports unlimited number of dashboards. You can access them by passing those slugs in `settings.CONTROLCENTER_DASHBOARDS` to url: `/admin/dashboards/<slugs>/`.

7.1.1 Dashboard options

`Dashboard` class has only two properties:

`title` By default the class name is used as title.

`widgets` A list of widgets. To group multiple widgets in one single block pass them in a list or wrap with a special `Group` class for additional options.

Here is an example:

```
from controlcenter import Dashboard, widgets

class OrdersDashboard(Dashboard):
    title = 'Orders'
    widgets = (
        NewOrders,
        (OrdersInProgress, FinishedOrders),
        Group([FinishedOrdersChart, ThisWeekOrdersChart],
              # Makes whole block larger
              width=widgets.LARGE,
              # Add html attribute class
              attrs={'class': 'my_fancy_group'})
    )
```

7.1.2 The grid

Dashboard is a responsive grid that appropriately scales up to 6 columns as the viewport size increases. It uses Masonry.js to make a better grid layout.

Viewport/column width	1	2	3	4	5	6
initial	100%					
> 768px	50%			100%		
> 1000px	25%	33%	50%	66%	75%	100%

Most useful sizes are available in `widgets` module:

```
SMALL = 1
MEDIUM = 2
LARGE = 3
LARGER = 4
LARGEST = 5
FULL = 6
```

7.1.3 Media class

`Dashboard` uses Media class from django to include static files on page:

```
class OrdersDashboard(Dashboard):
    title = 'Orders'
    widgets = (
        NewOrders,
        ...
    )

    class Media:
        css = {
            'all': 'my.css'
        }
```

7.1.4 Group options

Every element in `Dashboard.widgets` is automatically wrapped with a Group instance even if it's a single widget. This is the necessary process to make possible stack widgets together in one single block. You can define Group manually to control it's html attributes or override widget's width and height properties. For example:

```
class MyWidget(widgets.ItemList):
    model = Pizza
    values_list = ('name', 'price')
    width = widgets.LARGE

class MyDashboard(Dashboard):
    widgets = (
        widgets.Group([MyWidget], width=widgets.LARGER, height=300),
    )
```

`attrs` A dictionary of html attributes to set to the group (`class`, `id`, `data-foo`, etc.).

`width` An integer specifying the width in *columns*. By default the biggest value within the group is chosen.

height An integer specifying the `max-height` of the block in pixels. If necessary a scroll appears.

> **Note:** By default Group has the height of the biggest widget within group. Switching tabs (widgets) won't change it, because that will make the whole grid float.

`Group` supports the following methods:

get_id Returns `id` from `attrs` or a joined string of widget slugs (names) with `_and_` separator.

get_class Returns `class` from `attrs`.

get_attrs Returns `attrs` without `id` and `class` keys.

get_width Returns `width` if provided or biggest value in the group.

get_height Returns `height` if provided or biggest value in the group.

7.2 Widget options

`Widget` is a base class of all widgets. It was designed to handle as many cases as possible, it's very flexible and doesn't expect all properties to be set.

Available properties:

title If not provided class name is used instead.

model The model to display data for.

queryset A `QuerySet`. If not provided `model._default_manager` is called.

changelist_url Adds a clickable arrow at the corner of the widget with the link to model's admin changelist page. There are several ways to build the url:

```
class ModelItemList(widgets.ItemList):
    # Pass the model to get plain 'changelist' url
    changelist_url = Model

    # Add GET params in dictionary to filter and order the queryset
    changelist_url = model, {'status__exact': 0, 'o': '-7.-1'}

    # Or pass them as a string
    changelist_url = model, 'status__exact=0&o=-7.-1'

    # No model required at all
    changelist_url = '/admin/model/'

    # External url example
    changelist_url = 'https://duckduckgo.com/'
```

cache_timeout Widget's body cache timeout in seconds. Default is `None`.

template_name Template file name.

template_name_prefix A path to the directory with widget's template.

limit_to An integer specifying how many items should be returned by `Widget.values` method. By default it's `10`.

width Widget's width. See *Group options* width.

height Widget's height. See *Group options* height.

request Every widget gets request object on initialization and stores it inside itself. This is literally turns `Widget` into a tiny `View`:

```
class OrderWidget(widgets.Widget):
    model = Order

    def get_queryset(self):
        queryset = super(MyWidget, self).get_queryset()
        if not self.request.user.is_superuser:
            # Limits queryset for managers.
            return queryset.filter(manager=self.request.user)

        # Superusers can filter by status.
        status = self.request.GET.get('status')
        if status in Order.STATUSES:
            return queryset.filter(status=status)
        return queryset
```

Available methods:

get_template_name Returns the template file path.

values This method is automatically wrapped with cached_property descriptor to prevent multiple connections with whatever you use as a database. This also guarantees that the data won't be updated/changed during widget render process.

Note: Everything you wrap with cached_property becomes a property and can be only accessed as an attribute (without brackets). Don't use yield or return generator, they can't be cached properly (or cache them on you own).

```
class OrderWidget(widgets.Widget)
    def values(self):
        vals = super(MyWidget, self).values  # No brackets!
        return [(date.strftime('%m.%d'), order)
                for date, order in vals]  # No yield or generators!

    def dates(self):
        return [date for date, order in self.values]

    # `values` are cached and can be accessed
    # as many times as you want
    def orders(self):
        return [order for date, order in self.values]
```

Note: By default `limit_to` is used to limit queryset in here and not in `get_queryset` because if `QuerySet` is sliced ones it's can't be adjusted anymore, i.e. calling `super(...).get_queryset()` makes no sense in a subclass.

7.3 ItemList options

`ItemList` is very similar to django's ModelAdmin. It renders a list of objects returned by *Widget* `values` method. The most awesome thing about this widget is it can handle almost everything: a list of model objects, namedtuples, dictionaries and sequences (generators are not sequences).

```
class ModelItemList(widgets.ItemList):
    model = Model
    queryset = model.active_objects.all()
    list_display = ('pk', 'field', 'get_foo')
    list_display_links = ('field', 'get_foo')
    template_name = 'my_custom_template.html'

    def get_foo(self, obj):
        return 'foo'
    get_foo.allow_tags = True
    get_foo.short_description = 'Foo!'
```

`list_display` For model objects, namedtuples and dictionaries, `list_display` is a list of fields or keys of object. For sequences index of each item in `list_display` becomes a key in object, i.e. `dict(zip(list_display, sequence))`.

Widget's and model's class methods can be used in `list_display` just like in `ModelAdmin. list_display``_. They must take an extra parameter for the object returned by ``values`. They may have two properties `allow_tags` (`True` or `False` to allow or escape html tags) and `short_description` (for column name).

`list_display_links` Keys or fields should be linked to object's admin page. If nothing is provided `ItemList` will try to link the first column.

Note: If `ItemList.values` method doesn't return a list of model objects and `ItemList.model` is not defined, therefore there is no way left to build object's url.

`empty_message` If no items returned by `values` this message is displayed.

`sortable` Set `True` to make the list sortable.

Note: `ModelAdmin` gets sorted data from the database and `ItemList` uses **Sortable.js_** to sort rows in browser and it's not aware about fields data-type. That means you should be careful with sorting stuff like this: `%d.%m`.

7.4 Chart options

Django-controlcenter uses Chartist.js to create beautiful, responsive and dpi independent svg charts. `Chart` class has three extra cached methods:

`labels` Represents values on x-axis.

`series` Represents values on y-axis.

Note: Except for the `SingleBarChart` and `SinglePieChart` classes, this method must return a list of lists.

legend Chartist.js doesn't display series on chart which is really odd. As a workaround you can duplicate values on x-axis and then put labels in legend (and vice versa). Here is an example:

```
class MyBarChart(widgets.SingleBarChart):
    def series(self):
        # Y-axis
        return [y for x, y in self.values]

    def labels(self):
        # Duplicates series on x-axis
        return self.series

    def legend(self):
        # Displays labels in legend
        return [x for x, y in self.values]
```

7.4.1 Chartist

`Chart` may have a special `Chartist` class inside itself to configure Chartist.js:

```
class MyChart(widgets.Chart):
    class Chartist:
        point_labels = True
        options = {
            'reverseData': True,
            ...
        }
```

When you define `Chartist` it inherits chart's parent's `Chartist` properties automatically. The reason why hacky inheritance is used is the `options` property.

options It's a nested dictionary of options to be passed to Chartist.js constructor. Python dictionaries can't be inherited properly in a classic way. That's why when you define `options` in child `Chartist` class it deep copies and merges parent's one with it's own.

```
class MyChart(widgets.Chart):
    class Chartist:
        point_labels = True
        options = {
            'reverseData': True,
            'foo': {
                'bar': True
            }
        }

class MyNewChart(MyChart):
    class Chartist:
        options = {
            'fullWidth': True,
        }

# MyNewChart.Chartist copies MyChart.Chartist attributes
MyNewChart.chartist.options['reverseData']  # True
MyNewChart.chartist.options['foo']['bar']  # True
```

(continues on next page)

(continued from previous page)

```
MyNewChart.chartist.options['fullWidth']  # True
MyNewChart.chartist.point_labels  # True
```

`klass` Type of the chart. Available values are defined in `widgets` module: `LINE`, `BAR` and `PIE`. Default is `LINE`.

`scale` Aspect ratio of the chart. By default it's `octave`. See the full list of available values on official web-site (press 'Show default settings').

`LineChart` Displays point labels on `LINE` chart.

Note: If you don't want to use Chartist.js, don't forget to override `Dashboard.Media` to make not load useless static files.

7.4.2 LineChart

Line chart with point labels and useful Chartist.js settings. This chart type is usually used to display latest data dynamic sorted by date which comes in backward order from database (because you order entries by date and then slice them). `LineChart` passes `'reverseData': True` option to Chartist constructor which reverses `series` and `labels`.

7.4.3 TimeSeriesChart

A variant of `LineChart` for time-series data.

This chart does not define `labels`. Instead, each `series` must consist of pairs of `x` and `y` values, where `x` is a POSIX timestamp (as returned by datetime.timestamp).

```
class MyTimeSeriesChart(widgets.TimeSeriesChart):

    def series(self):
        return [
            [{'x': when.timestamp(), 'y': value} for (when, value) in samples],
        ]
```

The X-axis timestamp labels will be formatted using Date.toLocaleString.

To customise the timestamp label formatting, specify `Date.toLocaleString`'s `options` parameter using the `timestamp_options` configuration property. For example, to only show the year and short month as labels:

```
class MyTimeSeriesChart(widgets.TimeSeriesChart):
    class Chartist:
        timestamp_options = {
            'year': 'numeric',
            'month': 'short',
        }
```

To specify when ticks shown, see the Chartist.FixedScaleAxis documentation. For example:

```
class MyTimeSeriesChart(widgets.TimeSeriesChart):
    class Chartist:
        options = {
            'axisX': {
                # Use 'divisions' for a fixed number of sub-division ticks.
```

(continues on next page)

(continued from previous page)

```
                'divisions': 4,
                # Alternatively, use 'ticks' to explicitly specify a list of␣
↪timestamps.
            },
        }
```

7.4.4 BarChart

Bar type chart.

7.4.5 PieChart

Pie type chart.

Note: `PieChart.series` must return a flat list.

7.4.6 SingleBarChart, SinglePieChart, SingleLineChart

A special classes for charts with a single series. Simply define *label* and *series* fields in `values_list` then provide `model` or `queryset`. That's it.

This widget will render a bar chart of top three players:

```
class MySingleBarChart(widgets.SingleBarChart):
    # label and series
    values_list = ('username', 'score')
    # Data source
    queryset = Player.objects.order_by('-score')
    limit_to = 3
```

Note: `SingleLineChart.series` must return a list with a single list.

7.4.7 Chartist colors

There are two themes for charts. See *Customization*.

7.5 Customization

This options can be set in your settings file.

CONTROLCENTER_CHARTIST_COLORS Chart color theme: `default` (for Chartist.js colors) or `material` (for Google material colors).

CONTROLCENTER_SHARP A string specifying the header of row number column. By default it's `#`.

7.6 Examples

Lets say we have an app with this models:

```
from django.db import models


class Pizza(models.Model):
    name = models.CharField(max_length=100, unique=True)

    def __str__(self):
        return self.name


class Restaurant(models.Model):
    name = models.CharField(max_length=100, unique=True)
    menu = models.ManyToManyField(Pizza, related_name='restaurants')

    def __str__(self):
        return self.name


class Order(models.Model):
    created = models.DateTimeField(auto_now_add=True)
    restaurant = models.ForeignKey(Restaurant, related_name='orders')
    pizza = models.ForeignKey(Pizza, related_name='orders')
```

I'm going to put all imports in here just to not mess up the code blocks:

```
# project/dashboards.py

import datetime

from django.db.models import Count
from django.utils import timezone
from controlcenter import Dashboard, widgets
from .pizza.models import Order, Pizza, Restaurant
```

7.6.1 Scrollable ItemList with fixed height

Set `height` to make `ItemList` scrollable.

```
class MenuWidget(widgets.ItemList):
    # This widget displays a list of pizzas ordered today
    # in the restaurant
    title = 'Ciao today orders'
    model = Pizza
    list_display = ['name', 'ocount']
    list_display_links = ['name']

    # By default ItemList limits queryset to 10 items, but we need all of them
    limit_to = None

    # Sets widget's max-height to 300 px and makes it scrollable
    height = 300
```

(continues on next page)

(continued from previous page)

```
    def get_queryset(self):
        restaurant = super(MenuWidget, self).get_queryset().get()
        today = timezone.now().date()
        return (restaurant.menu
                          .filter(orders__created__gte=today, name='ciao')
                          .order_by('-ocount')
                          .annotate(ocount=Count('orders')))
```

7.6.2 Sortable and numerated ItemList

To make `ItemList` numerate rows simply add `SHARP` sign to `list_display`. To make it sortable set `sortable = True`. Remember: it's client-side sorting.

```
from controlcenter import app_settings
from django.utils.timesince import timesince


class LatestOrdersWidget(widgets.ItemList):
    # Displays latest 20 orders in the the restaurant
    title = 'Ciao latest orders'
    model = Order
    queryset = (model.objects
                     .select_related('pizza')
                     .filter(created__gte=timezone.now().date(),
                             name='ciao')
                     .order_by('pk'))
    # This is the magic
    list_display = [app_settings.SHARP, 'pk', 'pizza', 'ago']

    # If list_display_links is not defined, first column to be linked
    list_display_links = ['pk']

    # Makes list sortable
    sortable = True

    # Shows last 20
    limit_to = 20

    # Display time since instead of date.__str__
    def ago(self, obj):
        return timesince(obj.created)
```

7.6.3 Building multiple widgets with meta-class

Lets assume we have not filtered previous widgets querysets to Ciao restaurant. Then we can create widgets in a loop.

```
from controlcenter.widgets.core import WidgetMeta

RESTAURANTS = [
    'Mama',
    'Ciao',
    'Sicilia',
```

(continues on next page)

(continued from previous page)

```
]

# Metaclass arguments are: class name, base, properties.
menu_widgets = [WidgetMeta('{}MenuWidget'.format(name),
                           (MenuWidget,),
                           {'queryset': Restaurant.objects.filter(name=name),
                            # Adds human readable dashboard title
                            'title': name + ' menu',
                            # A link to model admin page
                            'changelist_url': (
                                  Pizza, {'restaurants__name__exact': name})})
                for name in RESTAURANTS]

latest_orders_widget = [WidgetMeta(
                           '{}LatestOrders'.format(name),
                           (LatestOrdersWidget,),
                           {'queryset': (LatestOrdersWidget
                                             .queryset
                                             .filter(restaurant__name=name)),
                            'title': name + ' orders',
                            'changelist_url': (
                                  Order, {'restaurant__name__exact': name})})
                        for name in RESTAURANTS]
```

7.6.4 Displaying series in legend

```
class RestaurantSingleBarChart(widgets.SingleBarChart):
    # Displays score of each restaurant.
    title = 'Most popular restaurant'
    model = Restaurant

    class Chartist:
        options = {
            # Displays only integer values on y-axis
            'onlyInteger': True,
            # Visual tuning
            'chartPadding': {
                'top': 24,
                'right': 0,
                'bottom': 0,
                'left': 0,
            }
        }

    def legend(self):
        # Duplicates series in legend, because Chartist.js
        # doesn't display values on bars
        return self.series

    def values(self):
        # Returns pairs of restaurant names and order count.
        queryset = self.get_queryset()
        return (queryset.values_list('name')
                        .annotate(baked=Count('orders'))
                        .order_by('-baked')[:self.limit_to])
```

7.6.5 LineChart widget with multiple series

```
from collections import defaultdict

class OrderLineChart(widgets.LineChart):
    # Displays orders dynamic for last 7 days
    title = 'Orders this week'
    model = Order
    limit_to = 7
    # Lets make it bigger
    width = widgets.LARGER

    class Chartist:
        # Visual tuning
        options = {
            'axisX': {
                'labelOffset': {
                    'x': -24,
                    'y': 0
                },
            },
            'chartPadding': {
                'top': 24,
                'right': 24,
            }
        }

    def legend(self):
        # Displays restaurant names in legend
        return RESTAURANTS

    def labels(self):
        # Days on x-axis
        today = timezone.now().date()
        labels = [(today - datetime.timedelta(days=x)).strftime('%d.%m')
                  for x in range(self.limit_to)]
        return labels

    def series(self):
        # Some dates might not exist in database (no orders are made that
        # day), makes sure the chart will get valid values.
        series = []
        for restaurant in self.legend:
            # Sets zero if date not found
            item = self.values.get(restaurant, {})
            series.append([item.get(label, 0) for label in self.labels])
        return series

    def values(self):
        # Increases limit_to by multiplying it on restaurant quantity
        limit_to = self.limit_to * len(self.legend)
        queryset = self.get_queryset()
        # This is how `GROUP BY` can be made in django by two fields:
        # restaurant name and date.
        # Ordered.created is datetime type but we need to group by days,
        # here we use `DATE` function (sqlite3) to convert values to
        # date type.
```

(continues on next page)

(continued from previous page)

```
        # We have to sort by the same field or it won't work
        # with django ORM.
        queryset = (queryset.extra({'baked':
                                    'DATE(created)'})
                            .select_related('restaurant')
                            .values_list('restaurant__name', 'baked')
                            .order_by('-baked')
                            .annotate(ocount=Count('pk'))[:limit_to])

        # The key is restaurant name and the value is a dictionary of
        # date:order_count pair.
        values = defaultdict(dict)
        for restaurant, date, count in queryset:
            # `DATE` returns `YYYY-MM-DD` string.
            # But we want `DD-MM`
            day_month = '{2}.{1}'.format(*date.split('-'))
            values[restaurant][day_month] = count
        return values
```

7.6.6 Simple data widgets

There's also support for displaying plain python data as widgets. Currently, two base classes are provided for rendering data: *ValueList*, which handles list data, and *KeyValueList*, which handles dictionary data. Each value (or key) can be a simple string or it can be dictionaries or objects with the following attributes:

- `label`: Label displayed in the widget
- `url`: If present, the label become a hyperlink to this url
- `help_text`: If present, display additional text accompanying label

If you want to specify these fields for a dictionary key, you'll need use `DataItem` from `controlcenter.widgets.contrib`, since you can't use a dictionary as a key to a dictionary because it's not hashable.

```
from controlcenter.widgets.contrib import simple as widgets
from controlcenter.utils import DataItem
from django.conf import settings


class DebuggingEndpointsWidget(widgets.ValueList):
    title = 'Debugging Endpoints'
    subtitle = 'Links for debugging application issues'

    def get_data(self):
        return [
            # Plain text displays as a row in the widget.
            'Not really sure why you would want plain text here',
            # Dictionary defining a display label and a url.
            {'label': 'Datadog Dashboard', 'url': 'https://example.com'},
            # `DataItem` can be used as an alternative to dictionaries.
            DataItem(label='Healthcheck', url='https://example.com',
                     help_text='Healthcheck report for external dependencies'),
        ]


class AppInfoWidget(widgets.KeyValueList):
```

(continues on next page)

(continued from previous page)

```
    title = 'App info'

    def get_data(self):
        return {
            # A simple key-value pair
            'Language code': settings.LANGUAGE_CODE,
            # A dictionary value can be used to display a link
            'Default timezone': {
                'label': settings.TIME_ZONE,
                'url': 'https://docs.djangoproject.com/en/2.1/topics/i18n/timezones/',
            },
            # To display a key with a link, you must use `DataItem` instead
            # of a dictionary, since keys must be hashable.
            DataItem(
                label='Debug on',
                url='https://docs.djangoproject.com/en/2.1/ref/settings/#debug'
            ): settings.DEBUG,
        }
```

7.7 Indices and tables

- genindex
- search