Django-Vertical_Multi-Columns

Release 0.0.7

Susan Wright

Apr 20, 2021

USER GUIDE

Be nice to your end user . . . minimize the scrolling and paging required with vertically sorted side-by-side columns in your Django templates.

CHAPTER

ONE

ABOUT

Django-Vertical-Multi-Columns (VMC) is a reusable Django application allowing users to display a list of items in vertically sorted side-by-side columns rather than in one long list.

In a typical "list" view where you need to display many choices, you would likely sort them into some logical order (e.g. alphabetical) and display them in one long vertical list on a web page. Django makes it quite easy to do this.

Scanning up and down a vertical list to find something is quite natural for people. But if the list is long, it can be quite annoying to be forced to do a lot of scrolling or paging up and down to find something.

Django-virtual-multi-columns (VMC) solves the excessive scrolling/paging problem. It enables a view class to generate content for your template that you can easily display in side-by-side columns that are still sorted vertically.

Even if there are so many choices that some scrolling or paging is still required, there will be less of it. The more columns you can fit on the page, the less scrolling or paging there will be.

CHAPTER

TWO

INSTALLATION

Django-vertical-multi-columns can be installed from PyPI with tools like `pip`:

```
pip install django-vertical-multi-columns
```

Add `'vertical-multi-columns'` to your `INSTALLED_APPS`.

```
INSTALLED_APPS = [
    ...
    'vertical-multi-columns',
]
```

You can specify a default number of columns in your Django settings:

```
VERTICAL_MULTI_COLUMNS = [
        {'NUMBER_OF_COLUMNS': 3}
]
```

Django-vertical-multi-columns is tested against these versions of Python and Django.

- **Python**: 3.7, 3.8, 3.9
- **Django**: 2.2, 3.0, 3.1, 3.2

CHAPTER

THREE

USAGE

There are 3 VMC views available. All are subclasses of Django's ListView so in addition to the specific VMC capabilities described below, all ListView's capabilities are still available.

3.1 View Options

EvenVMCView - Spreads your data across the number of columns you specify, keeping the columns largely the same length.

CriteriaVMCView - You provide a list of functions, one per column, plus a list of data keys referenced in the functions. VMC uses these to determine in which column each item should be placed.

DefinedVMCView - You already have the data arranged in the columns you want displayed. You provide the column list and VMC does the rest.

3.2 Setting the Number of Columns

There are several ways to specify how many columns you want generated. In priority order:

1. Pass kwarg `num_columns` to `super().__init__()` in your VMC view's `__init__()`.

```
def __init__(self):
    super().__init__(num_columns=5)
```

2. In your Django settings, specify a default number of columns to be generated. This is overridden if you pass a num_columns kwarg.

```
VERTICAL_MULTI_COLUMNS = [
        {NUMBER_OF_COLUMNS=3}
]
```

3. If there is no setting and you don't pass a num_columns kwarg, the number of columns defaults to 3.

3.3 Method Overrides

You must override some methods in the VMC classes.

EvenVMCView: Define a method:

- `get_data()` to return a list of sorted data in decoded JSON format.

CriteriaVMCView: Define 2 methods:

- `get_data()` to return a list of sorted data in decoded JSON format.
- `get_column_criteria()` to return two things:
  - a list of the functions VMC should use to place your data items in columns.
  - a list of the dictionary keys referenced in the functions.

NOTE: See How Passed CriteriaVMCView Functions Work below for a more in depth explanation.

DefinedVMCView: Define a method:

- `get_data()` to return a list of pre-defined columns in JSON format.

3.4 Sample Code

This example implements EvenVMCView but all the VMC views are fairly similar. Note that the example is pulling API data via requests but data from any source can be used as long as it can be converted into decoded JSON.

```
from vertical_multi_columns.configure import EvenVMCView
import requests
```

```
class MyEvenView(EvenVMCView):
    def __init__(self, **kwargs):
    #You can pass an optional num_columns kwarg to override the value in settings.
    # If there is nothing in settings and you don't pass num_columns, the number of␣
↪columns will be 3.
    super().__init__(num_columns=5)

def get_data(self):
    # Write logic to retrieve the data to be displayed (often from an API).
    # Sort it appropriately.
    # Note that data must be in decoded JSON format.
    resp = requests.get(<api_url>)
    resp.raise_for_status()
    deserialized_api_data = resp.json()
    sorted_api_data = sorted(deserialized_api_data, key=lambda i: i['<field>'])
    return sorted_api_data
except requests.exceptions.RequestException as err:
    messages.error(self.request, 'Something went wrong ... ' + str(err))
    return []


template_name = '<your_template>.html'
context_object_name = "<your_choice>"
```

3.5 Sample Template

A sample template is provided in the django-virtual-multi-columns library to demonstrate how to reference the output of your VMC view.

3.6 When is a VMC View Appropriate?

VMC views are typically meant for situations where you want to display a lot of short data in a more compact space than a straightforward ListView would require.

A common use case is to query an API for a list of choices (e.g. a list of plants or a list of car models) then display the list as links in some sort of list view. The end user selects one of the links which triggers a further call to the API to retrieve more detailed information. You would then display this in a detail view.

Avoid handling very complex hierarchical JSON in a VMC view.

While VMC views do support hierarchical JSON data, it can add unneeded complexity to your Django templates. To avoid that complexity, you are better off either:

- limiting your API call to returning only the data required for a user to make a selection, or
- if hierarchical JSON must be returned by the API, extract only the data you need for a user to make a choice before sending it on to the VMC view.

Note: The example site does demonstrate how hierarchical data can be handled in a view so if you need to go that route, you can do so successfully.

3.7 How Passed CriteriaVMCView Functions Work

You pass two lists to CriteriaVMCView so it can determine in which column each data item should appear. One is a list of functions and the other a list of the keys referenced in the functions. This scenario should help explain how you write those functions.

Say your API call returns a list of plants consisting of the fields 'name' and 'id' and you have decoded the returned JSON.

```
[{'id': 5, 'name': 'Asparagus'}, {'id': 2, 'name': 'Basil'}, ... , {'id':
34, 'name': 'Winter Squash'}]
```

Say you want to display 3 columns . . . plant names starting with A-F in column one, those starting with G-S in column two, and T-Z in column three. The first function you pass should return True if the plant name starts with A-F, the second should return True if the plant name starts with G-S, and the third should return True if the plant name starts with T-Z.

```
def a_to_f(self, args):
        ...
def g_to_s(self, args):
        ...
def t_to_z(self, args):
        ...
```

You also need to pass a list of the keys you reference in any of the functions. In this case, your functions are only querying the 'name' field but if you were querying other keys too, you would include them.

```
keys = ['name']
```

To communicate all this to your VMC view, you need to write a get_column_criteria() method that should look something like this:

```
def get_column_criteria(self):
    functions = [self.a_to_f, self.g_to_s, self.t_to_z]
    keys = ['name']
    return functions, keys
```

Focusing on the a_to_f() function, it is looking for instances in your returned data where the first letter of 'name' is in the range 'ABCDEF'. It will return True if so and False if not.

```
def a_to_f(self, args):
            parms = args
            return 'ABCDEF'.find(parms[0][0]) > -1
```

CriteriaVMCView's logic will apply each of your functions to each item in your data to determine if that item should appear in the corresponding function's column or not.

Say the data item currently being processed is `{'id': 5, 'name': 'Asparagus'}` and a_to_f() is being executed. The 'args' passed to the function by CriteriaVMCView will be string `'Asparagus'` since we said our keys were `['name']`.

Since our function is only interested in the name, it looks only at `parms[0]` which is 'Asparagus'. If there were additional keys passed, they would be parms[1], parms[2], etc. And further, since the function is only interested in the first letter of name, it only looks at `parms[0][0]` which is 'A'. The function returns True if parms[0][0] is in the range A-F and False if it is not.

If True is returned, that item will appear in the column. If False, it will not. Note that items can appear in multiple columns if function criteria overlap. Conversely an item can appear in no columns if none of the function criteria are met.

3.8 How to Contact/Get Support

If you have questions about usage or development you can open an issue on GitHub. You can also contact Susan Wright directly.

CHAPTER

FOUR

EXAMPLE SITE

There is a example site you can install and run to see the VMC views in action. It has no external requirements other than for you to have pip installed both Django itself and the django-vertical_multi_columns package.

The example site demonstrates each of the VMC view types in actual use, some using both fairly simple JSON and some using more complex hierarchical JSON.

Windows commands are shown here. Use the equivalent if you run on Mac or Linux.

1. Create a directory and change into it. Create a Python virtual directory and activate it using your normal method.

```
mkdir <newdirectory>
cd newdirectory
python -m venv <*virtualdirectory*>
.\<*virtualdirectory*\scripts\activate
```

2. Install Django and the django-vertical-multi-columns package.

```
pip install django
pip install django-vertical-multi-columns
```

3. Copy everything found in the repo under *example_site* to your new directory. Change to subdirectory example_site.

```
xcopy /E <*repo_directory*>\example_site
cd example_site
```

4. Change directory up one level and activate the site.

```
cd ..
python manage.py runserver
```

5. Point your browser to localhost:8000. More information about the site is provided there under "About the VMC Example Site.