Django-Aboutconfig Documentation

Release 1.1.0

Kirill Stepanov

Apr 11, 2020

Contents

1 Installation **3**

2 Configuration Options **5**
2.1 ABOUTCONFIG_CACHE_NAME 5
2.2 ABOUTCONFIG_CACHE_TTL 5
2.3 ABOUTCONFIG_AUTOLOAD 5

3 Usage **7**
3.1 Adding configuration 7
3.2 Using in templates 8
3.3 Using in python code 8

4 API **9**
4.1 Basic API 9
4.2 Advanced API 9

5 Custom Data Types **11**

6 Indices and tables **13**

Python Module Index **15**

Index **17**

Contents:

CHAPTER 1

Installation

Installation is very straight forward. Either install the package from pip (`pip install django-aboutconfig`) or from the sources.

After installing the package, apply migrations with `manage.py migrate` and collect static with `manage.py collectstatic`.

That's it, you're done.

Next, *set the configuration options*.

CHAPTER 2

Configuration Options

Some configuration options are provided for you to tinker with.

2.1 ABOUTCONFIG_CACHE_NAME

Configured cache's alias to use for storing the data. The alias refers to the ones used in the Django config.

Default value: `'default'`

2.2 ABOUTCONFIG_CACHE_TTL

How long the configured data should stay in cache (seconds). This value is passed directly to Django's caching mechanism, so that means a value of `None` is equal to indefinite TTL.

Default value: `None`

2.3 ABOUTCONFIG_AUTOLOAD

Whether to automatically load the data up into cache on start-up or not. You may want to disable this if you only want data to be loaded into cache on demand.

Default value: `True`

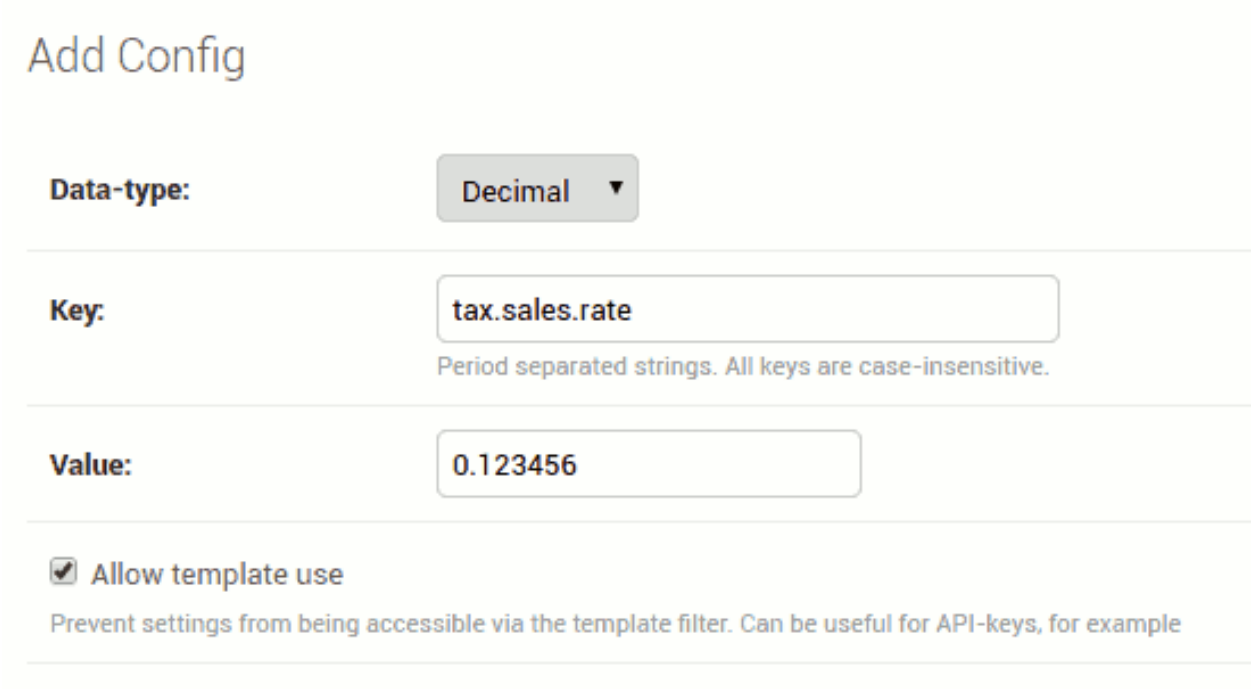
CHAPTER 3

Usage

3.1 Adding configuration

By default this library includes four commonly used data types: strings, booleans, integers and decimals. You can add custom ones if you feel constrained by these.

Head over to the django admin and add some configuration like so:



Note: Switching the type of data will cause the currently entered value to be coersed into something the new type

requires, if possible.

3.2 Using in templates

Most of the time you will be using these configured values inside templates like so:

```
{% load config %}
The website admin's email is {{ 'admin.details.email'|get_config }}.
```

An assignment tag also exists for convenience:

```
{% load config %}
{% get_config 'admin.details.email' as email %}
The website admin's email is <a href="mailto:{{ email }}">{{ email }}</a>.
```

If the configuration cannot be found, `None` is returned. You have to be careful when falsy values are possible because the *default* filter will not work in those cases. To produce a default value for a missing configuration key, use *default_if_none*.

```
{% load config %}
Does the key "foo.bar" exist? {{ 'foo.bar'|get_config|default_if_none:'No' }}.
```

Note: If the configuration is marked as not available for template use, it will act the same way as if it doesn't exist. Data is always available in python code.

3.3 Using in python code

There's only one straightfoward utility function available, and it can be used like so

```
from aboutconfig import get_config

def my_view(request):
    # some code...
    admin_email = get_config('admin.details.email')
    # some more code...
```

CHAPTER 4

API

4.1 Basic API

`aboutconfig.`**`get_config`**(*key: str*, *value_only: bool = True*) → Any

Get configured value by key.

By default returns value only. If `value_only` is `False`, returns an instance of aboutconfig.utils.DataTuple which also contains the `allow_template_use` value.

This is a lazy wrapper around the internal `utils.get_config()` function.

4.2 Advanced API

Use this as a reference for creating your own data types.

CHAPTER 5

Custom Data Types

Note: TODO

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

aboutconfig, 9
aboutconfig.serializers, 9

Index

A

`aboutconfig` (*module*), 9
`aboutconfig.serializers` (*module*), 9

G

`get_config()` (*in module aboutconfig*), 9