Trionyx Documentation

Release 1.0.6

Maikel Martens

Nov 14, 2019

Contents:

CHAPTER 1

Introduction

Trionyx is a Django web stack/framework for creating business applications. It's focus is for small company's that want to use business application instead of Excel and Google doc sheets.

With Trionyx the developer/business can focus on there domain models, business rules and processes and Trionyx will take care of the interface.

CHAPTER 2

Installation

To install Trionyx, run:

```
pip install Trionyx
```

2.1 Create new project

For creating a new project, run:

```
trionyx create_project new_project
```

CHAPTER 3

Settings

All Trionyx base settings

trionyx.settings.**gettext_noop**(*s*)
: Return same string, Dummy function to find translatable strings with makemessages

trionyx.settings.**get_env_var**(*setting, default=None, configs={'ALLOWED_HOSTS': ['localhost', '127.0.0.1'], 'DEBUG': True, 'SECRET_KEY': 'Not secure key'}*)
: Get environment variable from the environment json file

  Default environment file is *environment.json* in the root of project, Other file path can be set with the *TRIONYX_CONFIG* environment variable

trionyx.settings.**LOGIN_EXEMPT_URLS = ['static', 'api']**
: A list of urls that dont require a login

trionyx.settings.**TX_APP_NAME = 'Trionyx'**
: Full application name

trionyx.settings.**TX_LOGO_NAME_START = 'Tri'**
: The first characters of the name that are bold

trionyx.settings.**TX_LOGO_NAME_END = 'onyx'**
: The rest of the characters

trionyx.settings.**TX_LOGO_NAME_SMALL_START = 'T'**
: The first character or characters of the small logo that is bold

trionyx.settings.**TX_LOGO_NAME_SMALL_END = 'X'**
: The last character or characters of the small logo that is normal

trionyx.settings.**TX_THEME_COLOR = 'purple'**
: The theme skin color (header). Aviable colors: blue, yellow, green, purple, red, black. All colors have a light version blue-light

trionyx.settings.**TX_DEFAULT_DASHBOARD**()
: Return default dashboard

trionyx.settings.**TX_MODEL_OVERWRITES = {}**
Config to overwrite models, its a dict where the key is the original *app_label.model_name* and value is the new one.

```
TX_MODEL_OVERWRITES = {
    'trionyx.User': 'local.User',
}
```

trionyx.settings.**TX_MODEL_CONFIGS = {}**
Dict with configs for non Trionyx model, example:

```
TX_MODEL_CONFIGS = {
    'auth.group': {
        'list_default_fields': ['name'],
        'disable_search_index': False,
    }
}
```

trionyx.settings.**TX_DB_LOG_LEVEL = 30**
The DB log level for logging

CHAPTER 4

Config

4.1 Model configuration

class trionyx.config.**ModelConfig**(*model: django.db.models.base.Model*, *MetaConfig=None*)

ModelConfig holds all config related to a model that is used for trionyx functionality.

Model configs are auto loaded from the apps config file. In the apps config class create a class with same name as model and set appropriate config as class attribute.

```
# apps.blog.apps.py
class BlogConfig(BaseConfig):
    ...

    # Example config for Category model
    class Category:
        verbose_name = '{name}'
        list_default_fields = ['id', 'created_at', 'name']
        list_search_fields = ['name', 'description']
```

menu_name = None

Menu name, default is model verbose_name_plural

menu_order = None

Menu order

menu_exclude = False

Exclude model from menu

menu_root = False

Add menu item to root instead of under the app menu

menu_icon = None

Menu css icon, is ony used when root menu item

global_search = True

Enable global search for model

disable_search_index = False
Disable search index, use full for model with no list view but with allot of records

search_fields = ()
Fields to use for searching, default is all CharField and TextField

search_exclude_fields = ()
Fields you don't want to use for search

search_title = None
Search title of model works the same as *verbose_name*, defaults to __str__. Is given high priority in search and is used in global search

search_description = None
Search description of model works the same as *verbose_name*, default is empty, Is given medium priority and is used in global search page

list_fields = None
Customise the available fields for model list view, default all model fields are available.

list_fields is an array of dict with the field description, the following options are available:

- **field**: Model field name (is used for sort and getting value if no renderer is supplied)
- **label**: Column name in list view, if not set verbose_name of model field is used
- **renderer**: function(model, field) that returns a JSON serializable date, when not set model field is used.

```
list_fields = [
    {
        'field': 'first_name',
        'label': 'Real first name',
        'renderer': lambda model field: model.first_name.upper()
    }
]
```

list_default_fields = None
Array of fields that default is used in form list

list_select_related = None
Array of fields to add foreign-key relationships to query, use this for relations that are used in search or renderer

list_default_sort = '-pk'
Default sort field for list view

api_fields = None
Fields used in API model serializer, fallback on fields used in create and edit forms

api_disable = False
Disable API for model

verbose_name = '{model_name}({id})'
Verbose name used for displaying model, default value is "{model_name}({id})"

format can be used to get model attributes value, there are two extra values supplied:

- app_label: App name
- model_name: Class name of model

view_header_buttons = None
List with button configurations to be displayed in view header bar

```
view_header_buttons = [
    {
        'label': 'Send email', # string or function
        'url': lambda obj : reverse('blog.post', kwargs={'pk': obj.id}), #␣
↪string or function
        'type': 'default',
        'show': lambda obj, alias : True, # Function that gives True or False␣
↪if button must be displayed
        'modal': True,
    }
]
```

disable_add = False
Disable add for this model

disable_change = False
Disable change for this model

disable_delete = False
Disable delete for this model

auditlog_disable = False
Disable auditlog for this model

auditlog_ignore_fields = None
Auditlog fields to be ignored

hide_permissions = False
Dont show model in permissions tree, prevent clutter from internal models

get_app_verbose_name(*title: bool = True*) → str
Get app verbose name

get_verbose_name(*title: bool = True*) → str
Get class verbose name

get_verbose_name_plural(*title: bool = True*) → str
Get class plural verbose name

is_trionyx_model
Check if config is for Trionyx model

has_config(*name: str*) → bool
Check if config is set

get_field(*field_name*)
Get model field by name

get_fields(*inlcude_base: bool = False*, *include_id: bool = False*)
Get model fields

get_url(*view_name: str*, *model: django.db.models.base.Model = None*, *code: str = None*) → str
Get url for model

get_absolute_url(*model: django.db.models.base.Model*) → str
Get model url

get_list_fields() → [<class 'dict'>]
Get all list fields

get_field_type(*field: django.db.models.fields.Field*) → str
Get field type base on model field class

CHAPTER 5

Layout and Components

Layouts are used to render a view for an object. Layouts are defined and registered in layouts.py in an app.

Example of a tab layout for the user profile:

```
@tabs.register('trionyx.profile')
def account_overview(obj):
    return Container(
        Row(
            Column2(
                Panel(
                    'Avatar',
                    Img(src="{}{}".format(settings.MEDIA_URL, obj.avatar)),
                    collapse=True,
                ),
            ),
            Column10(
                Panel(
                    'Account information',
                    DescriptionList(
                        'email',
                        'first_name',
                        'last_name',
                    ),
                )
            ),
        )
    )
```

class trionyx.layout.**Layout**(**components*, ***options*)

Layout object that holds components

get_paths()

Get all paths in layout for easy lookup

find_component_by_path(*path*)

Find component by path, gives back component and parent

find_component_by_id(*id=None*, *current_comp=None*)
Find component by id, gives back component and parent

render(*request=None*)
Render layout for given request

collect_css_files(*component=None*)
Collect all css files

collect_js_files(*component=None*)
Collect all js files

set_object(*object*)
Set object for rendering layout and set object to all components

Parameters **object** –

Returns

add_component(*component*, *id=None*, *path=None*, *before=False*)
Add component to existing layout can insert component before or after component

Parameters

- **component** –
- **id** – component id
- **path** – component path, example: container.row.column6[1].panel

Returns

delete_component(*id=None*, *path=None*)
Delete component for given path or id

Parameters

- **id** – component id
- **path** – component path, example: container.row.column6[1].panel

Returns

class trionyx.layout.**Component**(**components*, ***options*)
Base component can be use as an holder for other components

template_name = None
Component template to be rendered, default template only renders child components

js_files = None
List of required javascript files

css_files = None
List of required css files

css_id
Generate random css id for component

set_object(*object*, *force=False*)
Set object for rendering component and set object to all components

Parameters **object** –

Returns

render(*context*, *request=None*)
Render component

class trionyx.layout.**ComponentFieldsMixin**

Mixin for adding fields support and rendering of object(s) with fields.

fields = []

List of fields to be rendered. Item can be a string or dict, default options:

- **field**: Name of object attribute or dict key to be rendered
- **label**: Label of field
- **value**: Value to be rendered
- **format**: String format for rendering field, default is '{0}'
- **renderer**: Render function for rendering value, result will be given to format. (lambda value, **options: value)
- **component**: Render field with given component, row object will be set as the component object

Based on the order the fields are in the list a __index__ is set with the list index, this is used for rendering a object that is a list.

```
fields = [
    'first_name',
    'last_name'
]

fields = [
    'first_name',
    {
        'label': 'Real last name',
        'value': object.last_name
    }
]
```

fields_options = {}

Options available for the field, this is not required to set options on field.

- **default**: Default option value when not set.

```
fields_options = {
    'width': {
        'default': '150px',
    }
}
```

objects = []

List of object to be rendered, this can be a QuerySet, list or string. When its a string it will get the attribute of the object.

The items in the objects list can be a mix of Models, dicts or lists.

add_field(*field*, *index=None*)

Add field

get_fields()

Get all fields

parse_field(*field_data*, *index=0*)

Parse field and add missing options

parse_string_field(*field_data*)

Parse a string field to dict with options

String value is used as field name. Options can be given after = symbol. Where key value is separated by : and different options by ;, when no : is used then the value becomes True.

Example 1: *field_name*

```
# Output
{
    'field': 'field_name'
}
```

Example 3 *field_name=option1:some value;option2: other value*

```
# Output
{
    'field': 'field_name',
    'option1': 'some value',
    'option2': 'other value',
}
```

Example 3 *field_name=option1;option2: other value*

```
# Output
{
    'field': 'field_name',
    'option1': True,
    'option2': 'other value',
}
```

Parameters **field_data** (*str*) –

Return dict

render_field(*field*, *data*)
Render field for given data

get_rendered_object(*obj=None*)
Render object

get_rendered_objects()
Render objects

class trionyx.layout.**HtmlTemplate**(*template_name*, *context=None*, *css_files=None*, *js_files=None*)
HtmlTemplate render django html template

render(*context*, *request=None*)
Render component

class trionyx.layout.**HtmlTagWrapper**(**args*, ***kwargs*)
HtmlTagWrapper wraps given component in given html tag

tag = 'div'
Html tag nam

attr = None
Dict with html attributes

get_attr_text()
Get html attr text to render in template

class trionyx.layout.**Html**(*html=None*, ***kwargs*)
Html single html tag

valid_attr = []
Valid attributes that can be used

class trionyx.layout.**Img**(*html=None*, ***kwargs*)
Img tag

class trionyx.layout.**Input**(*form_field=None*, *has_error=False*, ***kwargs*)
Input tag

class trionyx.layout.**ButtonGroup**(**args*, ***kwargs*)
Bootstrap button group

class trionyx.layout.**Button**(*label*, *url=None*, *model_url=None*, *model_params=None*, *model_code=None*, *dialog=False*, *dialog_options=None*, *dialog_reload_tab=None*, ***options*)
Bootstrap button

- link_url
- dialog_url
- onClick

set_object(*object*)
Set object and onClick

format_dialog_options()
Fromat options to JS dict

class trionyx.layout.**Container**(**args*, ***kwargs*)
Bootstrap container

class trionyx.layout.**Row**(**args*, ***kwargs*)
Bootstrap row

class trionyx.layout.**Column**(**args*, ***kwargs*)
Bootstrap Column

class trionyx.layout.**Column2**(**args*, ***kwargs*)
Bootstrap Column 2

class trionyx.layout.**Column3**(**args*, ***kwargs*)
Bootstrap Column 3

class trionyx.layout.**Column4**(**args*, ***kwargs*)
Bootstrap Column 4

class trionyx.layout.**Column5**(**args*, ***kwargs*)
Bootstrap Column 5

class trionyx.layout.**Column6**(**args*, ***kwargs*)
Bootstrap Column 6

class trionyx.layout.**Column7**(**args*, ***kwargs*)
Bootstrap Column 7

class trionyx.layout.**Column8**(**args*, ***kwargs*)
Bootstrap Column 8

class trionyx.layout.**Column9**(**args*, ***kwargs*)
Bootstrap Column 9

class trionyx.layout.**Column10**(**args*, ***kwargs*)
Bootstrap Column 10

class trionyx.layout.**Column11**(**args*, ***kwargs*)
Bootstrap Column 11

class trionyx.layout.**Column12**(**args*, ***kwargs*)
Bootstrap Column 12

class trionyx.layout.**Panel**(*title*, **components*, ***options*)
Bootstrap panel available options

- title
- footer_components
- collapse
- contextual: primary, success, info, warning, danger

class trionyx.layout.**DescriptionList**(**fields*, ***options*)
Bootstrap description, fields are the params. available options

- horizontal

class trionyx.layout.**TableDescription**(**fields*, ***options*)
Bootstrap table description, fields are the params

class trionyx.layout.**Table**(*objects*, **fields*, ***options*)
Bootstrap table

footer: array with first items array/queryset and other items are the fields, Same way how the constructor works

footer_objects = None
Can be string with field name relation, Queryset or list

get_footer_fields()
Get all footer fields

get_rendered_footer_object(*obj*)
Render footer object

get_rendered_footer_objects()
Render footer objects

CHAPTER 6

Forms

Forms are rendered in Trionyx with crispy forms using the bootstrap3 template.

Example:

```
from django import forms
from crispy_forms.helper import FormHelper
from crispy_forms.layout import Layout, Fieldset, Div

class UserUpdateForm(forms.ModelForm):
    # your form fields

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.helper = FormHelper()
        self.helper.layout = Layout(
            'email',
            Div(
                Fieldset(
                    'Personal info',
                    'first_name',
                    'last_name',
                    css_class="col-md-6",
                ),
                Div(
                    'avatar',
                    css_class="col-md-6",
                ),
                css_class="row"
            ),
            Fieldset(
                'Change password',
                'new_password1',
                'new_password2',
            ),
        )
```

6.1 Crispy Forms

6.1.1 Standard

class crispy_forms.layout.**Layout**(**fields*)

Form Layout. It is conformed by Layout objects: *Fieldset*, *Row*, *Column*, *MultiField*, *HTML*, *ButtonHolder*, *Button*, *Hidden*, *Reset*, *Submit* and fields. Form fields have to be strings. Layout objects *Fieldset*, *Row*, *Column*, *MultiField* and *ButtonHolder* can hold other Layout objects within. Though *ButtonHolder* should only hold *HTML* and BaseInput inherited classes: *Button*, *Hidden*, *Reset* and *Submit*.

Example:

```
helper.layout = Layout(
    Fieldset('Company data',
        'is_company'
    ),
    Fieldset(_('Contact details'),
        'email',
        Row('password1', 'password2'),
        'first_name',
        'last_name',
        HTML('<img src="/media/somepicture.jpg"/>'),
        'company'
    ),
    ButtonHolder(
        Submit('Save', 'Save', css_class='button white'),
    ),
)
```

class crispy_forms.layout.**ButtonHolder**(**fields*, ***kwargs*)

Layout object. It wraps fields in a

This is where you should put Layout objects that render to form buttons like Submit. It should only hold *HTML* and *BaseInput* inherited objects.

Example:

```
ButtonHolder(
    HTML(<span style="display: hidden;">Information Saved</span>),
    Submit('Save', 'Save')
)
```

class crispy_forms.layout.**BaseInput**(*name*, *value*, ***kwargs*)

A base class to reduce the amount of code in the Input classes.

render(*form*, *form_style*, *context*, *template_pack=<SimpleLazyObject: 'bootstrap3'>*, ***kwargs*)

Renders an *<input />* if container is used as a Layout object. Input button value can be a variable in context.

class crispy_forms.layout.**Submit**(**args*, ***kwargs*)

Used to create a Submit button descriptor for the {% crispy %} template tag:

```
submit = Submit('Search the Site', 'search this site')
```

Note: The first argument is also slugified and turned into the id for the submit button.

class crispy_forms.layout.**Button**(**args*, ***kwargs*)
Used to create a Submit input descriptor for the {% crispy %} template tag:

```
button = Button('Button 1', 'Press Me!')
```

Note: The first argument is also slugified and turned into the id for the button.

class crispy_forms.layout.**Hidden**(*name*, *value*, ***kwargs*)
Used to create a Hidden input descriptor for the {% crispy %} template tag.

class crispy_forms.layout.**Reset**(**args*, ***kwargs*)
Used to create a Reset button input descriptor for the {% crispy %} template tag:

```
reset = Reset('Reset This Form', 'Revert Me!')
```

Note: The first argument is also slugified and turned into the id for the reset.

class crispy_forms.layout.**Fieldset**(*legend*, **fields*, ***kwargs*)
Layout object. It wraps fields in a <fieldset>

Example:

```
Fieldset("Text for the legend",
    'form_field_1',
    'form_field_2'
)
```

The first parameter is the text for the fieldset legend. This text is context aware, so you can do things like:

```
Fieldset("Data for {{ user.username }}",
    'form_field_1',
    'form_field_2'
)
```

class crispy_forms.layout.**MultiField**(*label*, **fields*, ***kwargs*)
MultiField container. Renders to a MultiField

class crispy_forms.layout.**Div**(**fields*, ***kwargs*)
Layout object. It wraps fields in a

You can set *css_id* for a DOM id and *css_class* for a DOM class. Example:

```
Div('form_field_1', 'form_field_2', css_id='div-example', css_class='divs')
```

class crispy_forms.layout.**Row**(**args*, ***kwargs*)
Layout object. It wraps fields in a div whose default class is "formRow". Example:

```
Row('form_field_1', 'form_field_2', 'form_field_3')
```

class crispy_forms.layout.**Column**(**fields*, ***kwargs*)
Layout object. It wraps fields in a div whose default class is "formColumn". Example:

```
Column('form_field_1', 'form_field_2')
```

class crispy_forms.layout.**HTML**(*html*)
Layout object. It can contain pure HTML and it has access to the whole context of the page where the form is being rendered.

Examples:

```
HTML("{% if saved %}Data saved{% endif %}")
HTML('<input type="hidden" name="{{ step_field }}" value="{{ step0 }}" />')
```

class crispy_forms.layout.**Field**(**args*, ***kwargs*)
Layout object, It contains one field name, and you can add attributes to it easily. For setting class attributes, you need to use *css_class*, as *class* is a Python keyword.

Example:

```
Field('field_name', style="color: #333;", css_class="whatever", id="field_name")
```

class crispy_forms.layout.**MultiWidgetField**(**args*, ***kwargs*)
Layout object. For fields with `MultiWidget` as *widget*, you can pass additional attributes to each widget.

Example:

```
MultiWidgetField(
    'multiwidget_field_name',
    attrs=(
        {'style': 'width: 30px;'},
        {'class': 'second_widget_class'}
    ),
)
```

Note: To override widget's css class use `class` not `css_class`.

6.1.2 Bootstrap

class crispy_forms.bootstrap.**PrependedAppendedText**(*field*, *prepended_text=None*, *appended_text=None*, **args*, ***kwargs*)

class crispy_forms.bootstrap.**AppendedText**(*field*, *text*, **args*, ***kwargs*)

class crispy_forms.bootstrap.**PrependedText**(*field*, *text*, **args*, ***kwargs*)

class crispy_forms.bootstrap.**FormActions**(**fields*, ***kwargs*)
Bootstrap layout object. It wraps fields in a

Example:

```
FormActions(
    HTML(<span style="display: hidden;">Information Saved</span>),
    Submit('Save', 'Save', css_class='btn-primary')
)
```

class crispy_forms.bootstrap.**InlineCheckboxes**(**args*, ***kwargs*)
Layout object for rendering checkboxes inline:

```
InlineCheckboxes('field_name')
```

class crispy_forms.bootstrap.**InlineRadios**(**args*, ***kwargs*)
Layout object for rendering radiobuttons inline:

```
InlineRadios('field_name')
```

class crispy_forms.bootstrap.**FieldWithButtons**(**fields*, ***kwargs*)

class crispy_forms.bootstrap.**StrictButton**(*content*, ***kwargs*)
Layout object for rendering an HTML button:

```
Button("button content", css_class="extra")
```

class crispy_forms.bootstrap.**Container**(*name*, **fields*, ***kwargs*)
Base class used for *Tab* and *AccordionGroup*, represents a basic container concept

class crispy_forms.bootstrap.**ContainerHolder**(**fields*, ***kwargs*)
Base class used for *TabHolder* and *Accordion*, groups containers

first_container_with_errors(*errors*)
Returns the first container with errors, otherwise returns None.

open_target_group_for_form(*form*)
Makes sure that the first group that should be open is open. This is either the first group with errors or the first group in the container, unless that first group was originally set to active=False.

class crispy_forms.bootstrap.**Tab**(*name*, **fields*, ***kwargs*)
Tab object. It wraps fields in a div whose default class is "tab-pane" and takes a name as first argument. Example:

```
Tab('tab_name', 'form_field_1', 'form_field_2', 'form_field_3')
```

render_link(*template_pack=<SimpleLazyObject: 'bootstrap3'>*, ***kwargs*)
Render the link for the tab-pane. It must be called after render so css_class is updated with active if needed.

class crispy_forms.bootstrap.**TabHolder**(**fields*, ***kwargs*)
TabHolder object. It wraps Tab objects in a container. Requires bootstrap-tab.js:

```
TabHolder(
    Tab('form_field_1', 'form_field_2'),
    Tab('form_field_3')
)
```

class crispy_forms.bootstrap.**AccordionGroup**(*name*, **fields*, ***kwargs*)
Accordion Group (pane) object. It wraps given fields inside an accordion tab. It takes accordion tab name as first argument:

```
AccordionGroup("group name", "form_field_1", "form_field_2")
```

class crispy_forms.bootstrap.**Accordion**(**fields*, ***kwargs*)
Accordion menu object. It wraps *AccordionGroup* objects in a container:

```
Accordion(
    AccordionGroup("group name", "form_field_1", "form_field_2"),
    AccordionGroup("another group name", "form_field")
)
```

class crispy_forms.bootstrap.**Alert**(*content*, *dismiss=True*, *block=False*, ***kwargs*)
Alert generates markup in the form of an alert dialog

Alert(content='Warning! Best check yo self, you're not looking too good.')

class crispy_forms.bootstrap.**UneditableField**(*field*, **args*, ***kwargs*)
Layout object for rendering fields as uneditable in bootstrap

Example:

```
UneditableField('field_name', css_class="input-xlarge")
```

class crispy_forms.bootstrap.**InlineField**(**args*, ***kwargs*)

6.2 Trionyx

6.2.1 TimePicker

class trionyx.forms.layout.**TimePicker**(*field*, ***kwargs*)
Timepicker field renderer

CHAPTER 7

Celery background tasks

Trionyx uses Celery for background tasks, for full documentation go to Celery 4.1 documentation.

7.1 Configuration

Default there is no configuration required if standard RabbitMQ server is installed on same server. Default broker url is: *amqp://guest:guest@localhost:5672//*

7.1.1 Queue's

Default Trionyx configuration has three queue's:

- **cron**: Every tasks started by Celery beat is default put in the cron queue.
- **low_prio**: Is the default Queue every other tasks started by other processes are put in this queue.
- **high_prio**: Queue can be used for putting high priority tasks, default no tasks are put in high_prio queue.

7.1.2 Time limit

Default configuration sets the soft time limit of tasks to 1 hour and hard time limit to 1 hour and 5 minutes. You can catch a soft time limit with the *SoftTimeLimitExceeded*, and with the default configuration you have 5 minutes to clean up a task.

You can change the time limit with the settings *CELERY_TASK_SOFT_TIME_LIMIT* and *CELERY_TASK_TIME_LIMIT*

7.2 Creating background task

Tasks mused by defined in the file *tasks.py* in your Django app. Tasks in the tasks.py will by auto detected by Celery.

Example of a task with arguments:

```
from celery import shared_task

@shared_task
def send_email(email):
    # Send email

# You can call this task normally by:
send_email('test@example.com')

# Or you can run this task in the background by:
send_email.delay('test@example.com')
```

7.3 Running task periodically (cron)

You can run a task periodically by defining a schedule in the *cron.py* in you Django app.

```
from celery.schedules import crontab

schedule = {
    'spammer': {
        'task': 'app.test.tasks.send_email',
        'schedule': crontab(minute='*'),
    }
}
```

7.4 Running celery (development)

If you have a working broker installed and configured you can run celery with:

```
celery worker -A celery_app -B -l info
```

7.5 Live setup (systemd)

For live deployment you want to run celery as a daemon, more info in the Celery documentation

7.5.1 celery.service

/etc/systemd/system/celery.service

```
[Unit]
Description=Celery Service
After=network.target

[Service]
Type=forking
# Change this to Username and group that Trionyx project is running on.
User=celery
```

(continues on next page)

(continued from previous page)

```
Group=celery

EnvironmentFile=-/etc/conf.d/celery

# Change this to root of your Trionyx project
WorkingDirectory=/root/of/trionyx/projext

ExecStart=/bin/sh -c '${CELERY_BIN} multi start ${CELERYD_NODES} \
  -A ${CELERY_APP} --pidfile=${CELERYD_PID_FILE} \
  --logfile=${CELERYD_LOG_FILE} --loglevel=${CELERYD_LOG_LEVEL} ${CELERYD_OPTS}'
ExecStop=/bin/sh -c '${CELERY_BIN} multi stopwait ${CELERYD_NODES} \
  --pidfile=${CELERYD_PID_FILE}'
ExecReload=/bin/sh -c '${CELERY_BIN} multi restart ${CELERYD_NODES} \
  -A ${CELERY_APP} --pidfile=${CELERYD_PID_FILE} \
  --logfile=${CELERYD_LOG_FILE} --loglevel=${CELERYD_LOG_LEVEL} ${CELERYD_OPTS}'

[Install]
WantedBy=multi-user.target
```

7.5.2 Configuration file

/etc/conf.d/celery

```
CELERYD_NODES="cron_worker low_prio_worker high_prio_worker"

# Absolute or relative path to the 'celery' command:
CELERY_BIN="/usr/local/bin/celery"

CELERY_APP="celery_app"

# Extra command-line arguments to the worker
CELERYD_OPTS="-Ofair \
-Q:cron_worker          cron        -c:cron_worker      4 \
-Q:low_prio_worker      low_prio    -c:low_prio_worker  8 \
-Q:high_prio_worker     high_prio   -c:high_prio_worker 4"

# - %n will be replaced with the first part of the nodename.
# - %I will be replaced with the current child process index
#   and is important when using the prefork pool to avoid race conditions.
CELERYD_PID_FILE="/var/run/celery/%n.pid"
CELERYD_LOG_FILE="/var/log/celery/%n%I.log"
CELERYD_LOG_LEVEL="INFO"
```

Note: Make sure that the PID and LOG file directory is writable for the user that is running Celery.

CHAPTER 8

Widgets

Widgets are used on the dashboard and are rendered with Vue.js component.

class trionyx.widgets.**BaseWidget**

Base widget to extend for creating custom widgets. Custom widgets are created in *widgets.py* in root of app folder.

Example of random widget:

```
# <app dir>/widgets.py
RandomWidget(BaseWidget):
    code = 'random'
    name = 'Random widget'
    description = 'Shows random string'

    def get_data(self, request, config):
        return utils.random_string(16)
```

```
<!-- template path: widgets/random.html -->
<script type="text/x-template" id="widget-random-template">
    <div :class="widgetClass">
        <div class="box-header with-border">
            <!-- Get title from config, your form fields are also available in␣
↪the config -->
            <h3 class="box-title">[[widget.config.title]]</h3>
        </div>
        <!-- /.box-header -->
        <div class="box-body">
            <!-- vue data property will be filled with the get_data results␣
↪method --->
            [[data]]
        </div>
      </div>
</script>
```

(continues on next page)

(continued from previous page)

```
<script>
    <!-- The component must be called `widget-<code>` -->
    Vue.component('widget-random', {
        mixins: [TxWidgetMixin],
        template: '#widget-random-template',
    });
</script>
```

code = None
Code for widget

name = None
Name for widget is also used as default title

description = None
Short description on what the widget does

config_form_class = None
Form class used to change the widget. The form cleaned_data is used as the config

default_width = 4
Default width of widget, is based on grid system with max 12 columns

default_height = 20
Default height of widget, each step is 10px

template
Template path *widgets/{code}.html* overwrite to set custom path

image
Image path *img/widgets/{code}.jpg* overwrite to set custom path

get_data(*request*, *config*)
Get data for widget, function needs te be overwritten on widget implementation

config_fields
Get the config field names

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

c

t

Index

N

O

P

R

S

T

U

V