# Trionyx Documentation

***Release 2.0.2***

**Maikel Martens**

**Dec 24, 2019**

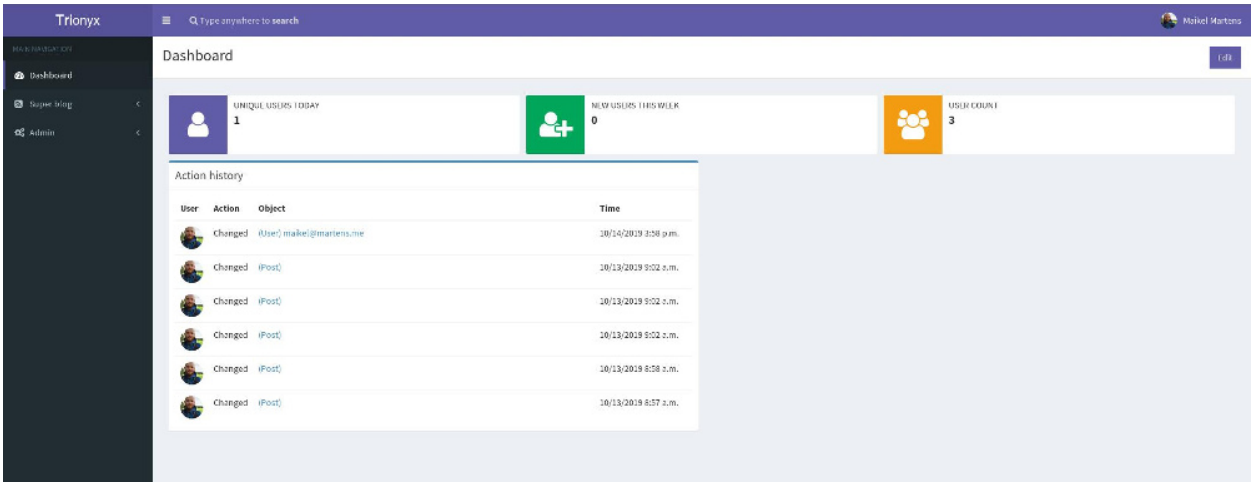
# Contents:

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 <project name>
```

Follow the steps given. And by the end you have a running base project ready to extend.

CHAPTER 3

# Getting started

Django already gives a solid foundation for building apps. Trionyx add some improvements like auto loading signals/forms/cron's. It also add new things as default views/api/background tasks (celery).

In this *getting started* guide we will create a new app, to show you a little of the basics on how Trionyx work and what it can do.

**This guide assumes you have followed the** *installation instructions*, **and you are now in the root of your new project with the virtual environment active**

*This guide requires no previous knowledge of Django, but it wont go in depth on how Django works*

## 3.1 Your first app

First we need to create a new app. Default Trionyx structure all apps go in the **apps** folder. To create a base app use the following manage command:

```
./manage.py create_app knowledgebase
```

In your apps folder should be your new app knowledgebase. For using the app we need to add it the INSTALLED_APPS:

```
# config/settings/base.py
# ...

INSTALLED_APPS += [
    'apps.knowledgebase',
]

# ...
```

## 3.2 File structure

A Trionyx file structure looks as follows. The ones marked as **bold**, are the ones you typically use in a Trionyx app. The others are used with a Django app and can be used if you need more customization.

- **<app name>**
  - **migrations**: DB migrations, these are auto generated by Django
  - static: folder with your static files (js/css/images/icons)
  - templates: HTML template files
  - **apps.py**: Contains the app and *model* configuration
  - **cron.py**: *Cron configuration*
  - **forms.py**: Create and *register* your forms.
  - **layouts.py**: *Create* your layouts
  - **models.py**: Define your models
  - **tasks.py**: *Create* your background tasks
  - urls.y: Define your custom urls
  - views.py: Create your custom views

## 3.3 Create model

We need a model to store our articles, this is just a simple Django model. Only difference is that we extend from BaseModel that add some extra Trionyx fields and functions.

```
# apps/knowledgebase/models.py
from trionyx import models
from django.contrib.contenttypes import fields

class Article(models.BaseModel):

    title = models.CharField(max_length=255)
    content = models.TextField()

    # Generic relation so that different model types can be linked
    # More info: https://docs.djangoproject.com/en/2.2/ref/contrib/contenttypes/
↪#generic-relations
    linked_object_type = models.ForeignKey(
        'contenttypes.ContentType',
        models.SET_NULL,
        blank=True,
        null=True,
    )
    linked_object_id = models.BigIntegerField(blank=True, null=True)
    linked_object = fields.GenericForeignKey('linked_object_type', 'linked_object_id')
```

After you created the model you need to make a migration (tells the database what to do). And then run the migration to create the database table.

./manage.py makemigrations ./manage.py migrate

If you run your project with *make run* and you login on it. You will see the menu has a Knowledgebase -> Article entry. You can create/view/edit articles but default list view is only id, and form is not user friendly.

## 3.4 Custom Form

Lets update the create and edit form to only show the title and content. And improve the content form field by using a wysiwyg editor.

```
# apps/knowledgebase/forms.py
from trionyx import forms
from .models import Article

@forms.register(default_create=True, default_edit=True)
class ArticleForm(forms.ModelForm):
    content = forms.Wysiwyg()

    # We are going to use this later
    linked_object_type = forms.ModelChoiceField(ContentType.objects.all(),␣
↪required=False, widget=forms.HiddenInput())
    linked_object_id = forms.IntegerField(required=False, widget=forms.HiddenInput())

    class Meta:
        model = Article
        fields = ['title', 'content']
```

If you refresh your page you should see an improved create form. When you created an article it is rendered with a simple default layout, we are going to change that later. First do some configuration so that there is a better verbose name, one menu item and a better default list view.

## 3.5 Model configuration

You can configure your model in the *apps.py*, lets change some for Article:

```
# apps/knowledgebase/apps.py
from trionyx.trionyx.apps import BaseConfig, ModelConfig

class Config(BaseConfig):
    """Knowledgebase configuration"""

    name = 'apps.knowledgebase'
    verbose_name = 'Knowledgebase'

    class Article(ModelConfig):

        # Improve default list view for users
        list_default_fields = ['created_at', 'created_by', 'title']

        # Set a clear verbose name instead of 'Article(1)'
        verbose_name = '{title}'

        # Move menu item to root and set a nice icon
        menu_root = True
        menu_icon = 'fa fa-book'
```

If you take a look now at the list view it looks much more informative. And the extra submenu is also replaced by only one menu item with a nice icon.

## 3.6 Custom Layout

Lets update the layout to remove some unnecessary fields and add some new ones. Layouts are build with components, so you dont need to write HTML. If you need something custom and dont want to build everything in HTML. There is the *trionyx.layout.HtmlTemplate* component that renders a given django template and context.

```
# apps/knowledgebase/layouts.py
from trionyx.views import tabs
from trionyx.layout import Column12, Panel, TableDescription

#register a new tab default this will be `general`
@tabs.register('knowledgebase.article')
def article_layout(obj):
    return Column12(
        Panel(
            obj.title, # For panel the first argument is the title,
            # all other arguments are components
            TableDescription(
                'created_by',
                'created_at',
                'updated_at',
                'content',
            )
        )
    )
```

This looks nice for your model, lets use that generic field that we created on the model and form. As you can see with the tab register you can create a tab for every model you want. We are going to at a knowledgebase tab to the *admin -> users*:

```
# apps/knowledgebase/layouts.py
from trionyx.views import tabs
from trionyx.layout import Column12, Panel, TableDescription, Button, Component, Html
from django.contrib.contenttypes.models import ContentType
from trionyx.urls import model_url

from .models import Article

# ...

@tabs.register('trionyx.user', code='knowledgebase')
def user_layout(obj):
    content_type = ContentType.objects.get_for_model(obj)

    return Column12(
        # Render a create button
        Button(
            'create article',
            url=model_url(Article, 'dialog-create', params={
                'linked_object_type': content_type.id,
                'linked_object_id': obj.id,
            }),
```

(continues on next page)

(continued from previous page)

```
            dialog=True,
            dialog_reload_tab='knowledgebase',
            css_class='btn btn-flat bg-theme btn-block'
        ),
        # Render every article in a new Panel
        *[
            Panel(
                art.title,
                TableDescription(
                    'created_by',
                    'created_at',
                    'updated_at',
                    # Components that accept fields can do so in different formats
                    # Default is string of field name and it will get the label and
↪value

                    {
                        'label': 'Content',
                        'value': Component(
                            Html(art.content),
                            Button(
                                'Edit',
                                url=model_url(art, 'dialog-edit'),
                                dialog=True,
                                dialog_reload_tab='knowledgebase',
                                css_class='btn btn-flat bg-theme btn-block'
                            ),
                        )
                    },
                    object=art,
                )
            ) for art in Article.objects.filter(
                linked_object_type=content_type,
                linked_object_id=obj.id,
            )
        ]
    )
```

If you reload the page on a user you will see a new tab *knowledgebase*. Articles that you create here are shown in the tab.

## 3.7 Signals

Django uses signals to allow you to get notifications on certain events from other apps. In this example we are going to use signals on our own Model to send an email to all users when a new article is created.

```
# apps/knowledgebase/signals.py
from django.db.models.signals import post_save
from django.dispatch import receiver
from trionyx.trionyx.models import User
from .models import Article

@receiver(post_save, sender=Article)
def notify_users(sender, instance, created=False, **kwargs):
```

(continues on next page)

(continued from previous page)

```
    if created:
        for user in User.objects.all():
            user.send_email(
                subject=f"New Article: {instance.title}",
                body=instance.content,
            )
```

You can find more information about signals here, Only thing that Trionyx does is auto import *signals.py* from all apps.

## 3.8 Background Task

Trionyx uses celery for background tasks, it comes preconfigured with 3 queue's. For more information go *here*. For our app we are going to use a schedulad background task to send a summary every week.

```
# apps/knowledgebase/tasks.py
from trionyx.tasks import shared_task
from django.utils import timezone
from trionyx.trionyx.models import User
from .models import Article

@shared_task()
def email_summary():
    count = Article.objects.filter(created_at__gt=timezone.now() - timezone.
↪timedelta(days=7)).count()
    for user in User.objects.all():
        user.send_email(
            subject=f"There are {count} new articles",
            body=f"There are {count} new articles",
        )
```

To make this task run every week we need to add it to the cron. You can do this from inside your app by creating a *cron.py*.

```
# apps/knowledgebase/cron.py
from celery.schedules import crontab

schedule = {
    'article-summary-every-sunday': {
        'task': 'apps.knowledgebase.tasks.email_summary',
        'schedule': crontab(minute=0, hour=0, day_of_week=0)
    },
}
```

Now *email_summary* will be run every sunday. For more information on scheduling go here

## 3.9 API

If you go to http://localhost:8000/api/ you can see that Trionyx automatically created an API entry point. Trionyx make use of the Django REST framework you can easily create your own endpoint or change the serializer user by the generated end point.

```python
# apps/knowledgebase/serializers.py or apps/knowledgebase/api/serializers.py
from trionyx.api import serializers
from trionyx.trionyx.models import User
from .models import Article

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ['id', 'email', 'first_name', 'last_name']

@serializers.register
class UserSerializer(serializers.ModelSerializer):
    created_by = UserSerializer()

    class Meta:
        model = Article
        fields = ['created_by', 'title', 'content']
```

I hope you have a better understanding on how to use Trionyx. And that it can help you build you business application with the focus on your data and processes.

# CHAPTER 4

# 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: str*, *default: Optional[Any] = None*, *configs: Dict[str, Any] = {'DEBUG': True, 'SECRET_KEY': 'Not a secret'}*) → Any
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.**get_watson_search_config**(*language*)
Get watson language config, default to pg_catalog.english for not supported language

List of supported languages can be found on https://github.com/etianen/django-watson/wiki/Language-support#language-support

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_COMPANY_NAME = 'Trionyx'**
Company name

trionyx.settings.**TX_COMPANY_ADDRESS_LINES = []**
Company address lines

trionyx.settings.**TX_COMPANY_TELEPHONE = ''**
Company telephone number

trionyx.settings.**TX_COMPANY_WEBSITE = ''**
Company website address

trionyx.settings.**TX_COMPANY_EMAIL = ''**
Company email address

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 5

# Config

## 5.1 Model configuration

**class** trionyx.config.**ModelConfig**(*model: Type[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(ModelConfig):
        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

**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, context: context.get('page') == 'view', # Function␣
↪that gives True or False if button must be displayed
        'dialog': True,
        'dialog_options': """function(data, dialog){
            // Example that will close dialog on success
            if (data.success) {
                dialog.close();
            }
        }"""
    }
]
```

**display_add_button = True**
Display add button for this model

**display_change_button = True**
Display change button for this model

**display_delete_button = True**
Display delete button for this model

**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

**has_permission**(*action*, *obj=None*, *user=None*)
Check if action can be performed on object

**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**() → List[dict]
Get all list fields

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

**get_header_buttons**(*obj=None*, *context=None*)
Get header buttons for given page and object

# CHAPTER 6

## 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.**Colors**

**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*, *append=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
- **append** – append component to selected component from id or path

**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 = ''**
Component template to be rendered, default template only renders child components

**js_files = []**
List of required javascript files

**css_files = []**
List of required css files

**set_object**(*object*, *force=False*, *layout_id=None*)
Set object for rendering component and set object to all components

when object is set the layout should be complete with all components. So we also use it to set the layout_id so it's available in the updated method and also prevent whole other lookup of all components.

**Parameters** **object** –

**Returns**

**updated**()
Object updated hook method that is called when component is updated with object

**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 (Can also be a component)
- **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)

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

**valid_attr = []**
Valid attributes that can be used

**color_class = ''**
When color is set the will be used as class example: btn btn-{color}

**attr = {}**
Dict with html attributes

**get_attr_text**()
Get html attr text to render in template

**class** trionyx.layout.**OnclickTag**(**components*, *url=None*, *model_url=None*, *model_params=None*, *model_code=None*, *sidebar=False*, *dialog=False*, *dialog_options=None*, *dialog_reload_tab=None*, *dialog_reload_sidebar=False*, *dialog_reload_layout=False*, ***options*)
HTML tag with onlick for url or dialog

**updated**()
Set onClick url based on object

**format_dialog_options**()
Fromat options to JS dict

**class** trionyx.layout.**Html**(*html=None*, ***kwargs*)
Renders html in a tag when set

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

**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.**Badge**(*field=None*, *html=None*, ***kwargs*)
Bootstrap badge

**updated**()
Set HTML with rendered field

**class** trionyx.layout.**Alert**(*html*, *alert='success'*, *no_margin=False*, ***options*)
Bootstrap alert

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

**class** trionyx.layout.**Button**(*label*, ***options*)
Bootstrap button

**class** trionyx.layout.**Thumbnail**(*src*, ***options*)
Bootstrap Thumbnail

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

**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 7

# Forms

Default Trionyx will generate a form for all fields without any layout. Forms can be created and registered in the *forms.py*.

trionyx.forms.**register**(*code: Optional[str] = None*, *model_alias: Optional[str] = None*, *default_create: Optional[bool] = False*, *default_edit: Optional[bool] = False*, *minimal: Optional[bool] = False*)

Register form for given model_alias, if no model_alias is given the Meta.model is used to generate the model alias.

**Parameters**

- **code** (*str*) – Code to identify form
- **model_alias** (*str*) – Alias for a model (if not provided the Meta.model is used)
- **default_create** (*bool*) – Use this form for create
- **default_edit** (*bool*) – Use this form for editing
- **minimal** (*bool*) – Use this form for minimal create

```
# <app>/forms.py
from trionyx import forms

@forms.register(default_create=True, default_edit=True)
class UserForm(forms.ModelForm):

    class Meta:
        model = User
```

## 7.1 Layout

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

```
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',
            ),
        )
```

## 7.2 Crispy Forms

### 7.2.1 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"/>'),
```

(continues on next page)

(continued from previous page)

```
        '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=(
```

(continues on next page)

(continued from previous page)

```
        {'style': 'width: 30px;'},
        {'class': 'second_widget_class'}
    ),
)
```

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

### 7.2.2 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*)

## 7.3 Trionyx

### 7.3.1 TimePicker

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

CHAPTER 8

# Celery background tasks

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

## 8.1 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//*

### 8.1.1 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.

### 8.1.2 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*

## 8.2 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')
```

## 8.3 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='*'),
    }
}
```

## 8.4 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
```

## 8.5 Live setup (systemd)

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

### 8.5.1 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
```

### 8.5.2 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 9

# 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 = ''**
Name for widget is also used as default title

**description = ''**
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: django.http.request.HttpRequest*, *config: dict*)
Get data for widget, function needs te be overwritten on widget implementation

**config_fields**
Get the config field names

CHAPTER 10

# How to write reusable apps

**Todo:** Write docs

CHAPTER 11

# Changelog

## 11.1 [2.0.2] - 24-12-2019

### 11.1.1 Fixed

- Fix inlineforms not working in popup
- Widget config dialog wasn't shown

## 11.2 [2.0.1] - 19-12-2019

### 11.2.1 Added

- Add helper function for setting the Watson search language

### 11.2.2 Changed

- Small improvements to prevent double SQL calls
- #39 Make python version configurable for Makefile

### 11.2.3 Fixed

- Ansible role name is not found
- JsonField does not work in combination with jsonfield module

## 11.3 [2.0.0] - 11-12-2019

**Compatibility breaking changes: drop support for python 3.5**

### 11.3.1 Added

- Add generic model sidebar
- Add Summernote wysiwyg editor
- Add more tests and MyPy
- Add getting started guide to docs and improve README
- Add more bootstrap components
- Add frontend layout update function
- Add system variables
- Add helper class for app settings
- Add support for inline forms queryset
- Add company information to settings
- Add price template filter
- Add ability for forms to set page title and submit label
- Add options to display create/change/delete buttons
- Add signals for permissions

### 11.3.2 Changed

- Drop support for python 3.5
- Improve api serializer registration
- Improve list view column sizes
- Move from vitualenv to venv
- Make inline formset dynamic
- Make delete button available on edit page
- Make header buttons generic and show them on list and edit page
- Header buttons can be shown based on tab view

### 11.3.3 Fixed

- Cant go to tab if code is same as code in jstree
- Several small fixes and changes

## 11.4 [1.0.5] - 31-10-2019

### 11.4.1 Fixed

- Fixed model overwrite configs/forms/menu

## 11.5 [1.0.4] - 31-10-2019

### 11.5.1 Changed

- Improved new project creation

### 11.5.2 Fixed

- Filter related choices are not shown

## 11.6 [1.0.3] - 30-10-2019

### 11.6.1 Fixed

- Fixed to early reverse lookup
- Fixed not all quickstart files where included

## 11.7 [1.0.2] - 30-10-2019

### 11.7.1 Changed

- Dialog form initial also uses GET params
- model_url accept GET params as dict
- Improve Button component
- ComponentFieldsMixin fields can now render a Component
- Add option to Component to force update object
- Base Component can be used as an holder for Components to be rendered
- Add debug comments to Component output

### 11.7.2 Fixed

- Delete dialog does not return *success* boolean
- Fixed html component not rendering html and tag not closed

## 11.8 [1.0.1] - 29-10-2019

### 11.8.1 Fixed

- Fixed verbose name has HTML

## 11.9 [1.0.0] - 29-10-2019

**Compatibility breaking changes: Migrations are cleared**

### 11.9.1 Added

- Add get_current_request to utils
- Add DB logger
- Add options to disable create/update/delete for model
- Add debug logging for form errors
- Add audit log for models
- Add user last_online field
- Add support for inline formsets
- Add rest API support
- Add option to add extra buttons to header
- Add search to list fields select popover
- Add Dashboard
- Add Audtilog dashboard widget
- Add model field summary widget
- Add auto import Trionyx apps with pip entries
- Add data choices lists for countries/currencies/timezones
- Add language support + add Dutch translations
- Add user timezone support
- Add CacheLock contectmanager
- Add locale_overide and send_email to user
- Add mass select selector to list view
- Add mass delete action
- Add Load js/css from forms and components
- Add view and edit permissions with jstree
- Add mass update action
- Add BaseTask for tracking background task progress
- Add support for related fields in list and auto add related to queryset

- Add layout component find/add/delete
- Add model overwrites support that are set with settings
- Add renderers for email/url/bool/list

### 11.9.2 Changed

- Set fallback for user profile name and avatar
- Improve header visibility
- Make filters separate vuejs component + function to filter queryset
- Improve theme colors and make theme square
- Update AdminLTE+plugins and Vue.js and in DEBUG use development vuejs
- Refactor inline forms + support single inline form
- Auditlog values are rendered with renderer
- Changed pagination UX
- Show filter label instead of field name

### 11.9.3 Fixed

- Project create settings BASE_DIR was incorrect
- Menu item with empty filtered childs is shown
- Make verbose_name field not required
- Global search is activated on CTRL commands
- Auditlog delete record has no name
- Created by was not set
- Auditlog gives false positives for Decimal fields
- Render date: localtime() cannot be applied to a naive datetime
- Fix model list dragging + fix drag and sort align
- Fixed None value is rendered as the string None

## 11.10 [0.2.0] - 04-06-2019

**Compatibility breaking changes**

### 11.10.1 Added

- Form register and refactor default forms to use this
- Add custom form urls + shortcut model_url function
- Add layout register + layout views
- Add model verbose_name field + change choices to use verbose_name query

- Add permission checks and hide menu/buttons with no permission

### 11.10.2 Changed

- Render fields for verbose_name and search title/description
- Move all dependencies handling to setup.py
- Upgrade to Django 2.2 and update other dependencies
- refactor views/core from Django app to Trionyx package
- Rename navigation to menu
- Move navigtaion.tabs to views.tabs
- Quickstart project settings layout + add environment.json

### 11.10.3 Fixed

- Cant search in fitler select field
- Datetimepicker not working for time
- Travis build error
- Button component

## 11.11 [0.1.1] - 30-05-2019

### 11.11.1 Fixed

- Search for not indexed models
- Lint errors

## 11.12 [0.1.0] - 30-05-2019

### 11.12.1 Added

- Global search
- Add filters to model list page
- Set default form layouts for fields

### 11.12.2 Changed

- Search for not indexed models

### 11.12.3 Fixed

- Make datepicker work with locale input format
- On menu hover resize header
- Keep menu state after page refresh
- Search for not indexed models

CHAPTER 12

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## c



## t

# Index

## F



## G



## H



## I

## J



## L



## M



## N



## O



## P



## R



## S



## T

## U



## V