Horizon Contrib Documentation

Release 1

Michael Kuty & Ales Komarek

Oct 02, 2017

Contents

Library built on top of Django and Horizon(part of OpenStack Dashboard) for building modern web applications.

With this toolkit is building applications blazingly fast and easy !

This library provide generic implementation most of Horizon components, add more tools for easily scaffolding applications and preserves support for complex customizations.

CHAPTER 1

Short story

Horizon is pretty package for fast creating UI for everything. But is designed for model-less applications like an OpenStack Dashboard. If we connect Horizon with typical Django application we must create same pieces of same components and this is really suck ! We want more declarative and less imperative code. For this purpose we create this library which compose common stuff in one place.

CHAPTER 2

Features

- With Django and Conent Types
  - Views - PaginatedIndex, Create, Update, Delete in Angular modal's
  - Tables with inline-ajax update
  - Modal Forms autohandled
  - Generic - IndexView with pagination, CRUD actions and AJAX inline-edit.

no implementation required, all Django stuff is generated automatically like an admin, but in more customizeable and extendable form.

- Rest API Dashboards
  - APIModel
  - ClientBase - simple implementation which uses `requests`
  - Generic - Tables, Views, Actions

and plus all features defined under Django because if we have model most of things works well without any modification.

Manager has all responsibilty for get data from remote API. It's simple object which has similar methods with django model managers. And it's bound to Abstract model.

- Others
  - ReactJS integration - for large tables with thousands rows we have integrated https://github.com/glittershark/reactable as `ReactTable`
  - tabs, templates (modal login, ...)
  - set of common filters, templatetags

Examle App

Contents:

Short tutorial

Installation

```
pip install horizon-contrib

pip install git+https://github.com/michaelkuty/horizon-contrib.git@develop
```

Configuration

```
INSTALLED_APPS += ('horizon_contrib',)
```

Optionaly include `horizon_contrib.urls` with `namespace='horizon'`. This is only for generic functionality like a index,create,update,delete views.

```
from django.conf.urls import patterns, include, url

urlpatterns = patterns('',
    ...
    url(r'^contrib/', include('horizon_contrib.urls', namespace='horizon'), ),
    ...
)
```

Note: `namespace` is important for url `reverse`

Django example

Your models.py

```
from django import models

class Project(models.Model):

    name = models.CharField..
    description = models.CharField..
    ...

    class Meta:
        verbose_name = 'Project'
```

Include our urls.

```
from django.conf.urls import patterns, include, url

urlpatterns = patterns('',
    ...
    url(r'^contrib/', include('horizon_contrib.urls', namespace='horizon'), ),
    ...
)
```

Visit these urls

```
/contrib/models/<model_name>/index/
/contrib/models/<model_name>/create/
/contrib/models/<model_name>/<model_id>/update/
```

Note: For these purpose must be `django.contrib.contenttypes` in `INSTALLED_APPS`.

REST API Dashboards

Dashboard structure:

```
my_dashboard
    |-- __init__.py
    |-- projects
        |-- __init__.py
        |-- managers.py
        |-- models.py
        |-- panel.py
    |-- dashboard.py
```

Your `models.py`

```
from horizon_contrib.api import APIModel
from horizon_contrib.common import register_model

class Project(APIModel):

    name = models.CharField('id', primary_key=True)  # default primary is id
    description = models.CharField..
    ...

    objects = Manager()  # see below

    class Meta:
        verbose_name = 'Project'
        abstract = True

register_model(Project)  # supply Django Content Type framework
```

New `managers.py`

```
from horizon_contrib.api import Manager

class Manager(Manager):

    def all(self, *args, **kwargs):
        return self.request('/projects')
```

Finally `panel.py`

```
from horizon_contrib.panel import ModelPanel
from horizon_redmine.dashboard import RedmineDashboard

class ProjectPanel(ModelPanel):
    name = "Projects"
```

```
    slug = 'projects'
    model_class = 'project'

RedmineDashboard.register(ProjectPanel)
```

navigate your browser to `/contrib/models/project/index` ! or `/contrib/models/project/create`

Manager usual usage.

`utils/redmine_client.py`

```
from django.conf import settings
from horizon_contrib.api import Manager

class RedmineManager(Manager):

    # here will change client behaviour

    # def request(...)

    def set_api(self):
        self.api = '%s://%s:%s' % (
            settings.REDMINE_PROTOCOL,
            settings.REDMINE_HOST,
            settings.REDMINE_PORT)
```

`managers.py`

```
from django.conf import settings
from ..utils.redmine_client import RedmineManager

class ProjectManager(RedmineManager):

    def all(self, *args, **kwargs):
        return self.request('/projects')
```

Generic module

This module provide same functionality as Django Admin, but in AngularJS cloak with custom actions, modal forms client side categorizing, filtering and many more.

With Horizon

Dashboard structure:

```
|-- my_dashboard
    |-- __init__.py
    |-- projects
        |-- __init__.py
        |-- models.py
        |-- forms.py
        |-- managers.py
        |-- urls.py
        |-- views.py
```

```
    |-- issues
        |-- __init__.py
        |-- panel.py
    |-- dashboard.py
```

Simply register your ModelClass and connect it to ModelPanel which provides namespace and menu item.

```
from horizon import forms
from horizon_contrib.api import models
from horizon_contrib.common import register_model

from .managers import ProjectManager

class Project(models.APIModel):

    ...

    objects = ProjectManager()  # connect our manager

    class Meta:
        abstract = True
        verbose_name = "Project"
        verbose_name_plural = "Projects"

register_model(Project)  # supply django Content Types
```

```
from horizon_contrib.panel import ModelPanel
from horizon_redmine.dashboard import RedmineDashboard

class ProjectPanel(ModelPanel):
    name = "Projects"
    slug = 'projects'
    model_class = 'project'

RedmineDashboard.register(ProjectPanel)
```

With Django

With Django is all stuff generic we can only used or override some parts of how we want.

Dashboard structure:

```
my_app
    |-- __init__.py
    |-- models.py
    |-- my_dashboard
        |-- __init__.py
        |-- projects
            |-- __init__.py
            |-- panel.py
            |-- forms.py
            |-- tables.py
            |-- urls.py
            |-- views.py
        |-- issues
            |-- __init__.py
```

```
            |-- panel.py
        |-- dashboard.py
```

With reverse

```
from django.core.urlresolvers import reverse

print reverse('horizon:contrib:generic:index', args=['project'])

# render as ReactJS table
print reverse('horizon:contrib:generic:index', args=['project', 'react'])
```

Warning: For these purpose must be `django.contrib.contenttypes` in `INSTALLED_APPS`.

REST API Dashboards

For example all Horizon Dashboard is model-less without DB directly connected.

For usual app we need index view which provide base filtering some actions like a creating, editing and enything else.

In typical application we must define these things

Usual REST app

Minimal

- Model - Table in horizon world
- View - index view
- Data - if we haven't model we must load data from remote host

Dashboard structure:

```
my_dashboard
    |-- __init__.py
    |-- projects
        |-- __init__.py
        |-- panel.py
        |-- forms.py
        |-- tables.py
        |-- urls.py
        |-- views.py
    |-- dashboard.py
```

Optional: Table, Actions, Forms, Workflows, Templates and many more

If we build with horizon contrib we need these components

Minimal with contrib

- Model in horizon_contrib world
- Panel - horizon panel which provide url path
- Data - `Manager` in horizon_contrib world

Dashboard structure:

```
my_dashboard
    |-- __init__.py
    |-- projects
        |-- __init__.py
        |-- managers.py
        |-- models.py
        |-- panel.py
    |-- dashboard.py
```

Optional: Full overrides

New approach

We prefer new way which is different in one aspect. We moved responsibility for load data into Model class. Every object is responsible for his data. This means Model has manager and this managers load all related data. In many cases we would like to use another manager methods like a create,delete,update,get etc..

Dashboard structure:

```
my_dashboard
    |-- __init__.py
    |-- projects
        |-- __init__.py
        |-- models.py    # define data structure
        |-- managers.py  # load remote data
        |-- panel.py     # register namespace
    |-- dashboard.py
```

Manager

First we define our manager. It can be anything, but must provide one method called `all` for index views.

For this example returns only array with one dictionary which presents data from remote API.

Manager can be located anywhere we recommend in the `managers.py`, but is not golden rule.

```
from horizon_contrib.api import Manager

class ProjectManager(Manager):

    def all(self, *args, **kwargs):
        return [{'id': 1, 'project': 'Horizon', 'description': 'Foo'}]
```

Note: See Manager class in the code, its simple object based on `ClientBase` which has `request` method.

Usually we do this

```
class ProjectManager(Manager):

    def all(self, *args, **kwargs):
        # call GET -> protocol:host:port/api/projects and returns lis of projects
        return self.request('api/projects')
```

And onther methods for us these methods can be implemented later or not. Depends only what we need.

```
class ProjectManager(object):

    def create(self, *args, **kwargs):
        raise NotImplementedError

    def update(self, *args, **kwargs):
        raise NotImplementedError

    def delete(self, *args, **kwargs):
        raise NotImplementedError

    # and common methods
    def order_by(self, *args, **kwargs):
        raise NotImplementedError

    def filter(self, *args, **kwargs):
        raise NotImplementedError
```

Now define our model, in this case is simple Project.

Model

```
from horizon import forms
from horizon_contrib.api import models
from horizon_contrib.common import register_model

from .managers import ProjectManager

class Project(models.APIModel):

    id = models.IntegerField("ID", required=False)
    name = models.CharField("ID", required=False)
    description = models.CharField("ID", required=False, widget=forms.widgets.
↪Textarea)

    objects = ProjectManager()  # connect our manager

    def __unicode__(self):
        return str(self.name)

    def __repr__(self):
        return str(self.name)

    class Meta:
        abstract = True
        verbose_name = "Project"
        verbose_name_plural = "Projects"

register_model(Project)  # supply django Content Types
```

Benefits

One of benefits is unification and consistency of yours APIs across all your apps.

```
from .models import Project

Project.objects.all()

[{'id': 1, 'project': 'Horizon', 'description': 'Foo'}]

new_project = Project(**{'name': 'Foo', 'description': 'Bar'})

new_project.save()

# raise NotImplementedError from your manager class, becase ``save`` is proxied to␣
↪him in default state.

project = Project.objects.get(id=1)
project.delete()
```

Managers

For advance working with managers we simple extends our `ProjectManager`

```
class ProjectManager(object):
    ...
    SCOPE = "projects"

    def get(self, request, id):
        return self.request(
            request,
            '/{0}/{1}/'.format(self.SCOPE, id))
```

Note: We known API base url from `settings` and now provide model endpoint. Benefits from this see below.

Complex model usual has many to many or querysets of objects

```
from horizon import forms
from horizon_contrib.api import models

from horizon_contrib.api import Manager
from .managers import ProjectManager

class CategoryManager(Manager):

    SCOPE = 'project/categories'  # for now we haven`t parent manager

class Project(models.APIModel):

    id = models.IntegerField("ID", required=False)
    name = models.CharField("ID", required=False)
    description = models.CharField("ID", required=False, widget=forms.widgets.
↪Textarea)
```

```
    objects = ProjectManager()  # connect our manager
    categories = CategoryManager()

    class Meta:
        abstract = True
        verbose_name = "Project"
        verbose_name_plural = "Projects"
```

```
Project.categories.all()
```

Horizon world

Minimal required definition is `panel.py` which connect model class with url namespace and menu item.

Panel

panel.py

```
from horizon_contrib.panel import ModelPanel
from horizon_redmine.dashboard import RedmineDashboard

class ProjectPanel(ModelPanel):
    name = "Projects"
    slug = 'projects'
    model_class = 'project'

    # react = True enable reactjs table

RedmineDashboard.register(ProjectPanel)
```

But usualy we must override many internals.

Table

Define your table for index view

```
from horizon_contrib.tables import ModelTable

class ProjectTable(ModelTable):

    class Meta:

        model_class = Project
```

View

```
from horizon_contrib.tables import PaginatedView

from .tables import ProjectTable

class IndexView(PaginatedView):
```

```
    table_class = ProjectTable
```

yes and urls forms actions etc. and still again

View call `table.get_table_data` which returns `model_class.objects.all()` in default state

```
class IndexView().get_data()

[{'id': 1, 'project': 'Horizon', 'description': 'Foo'}]
```

Tables

ModelTable

tables.py

```
from horizon_contrib.tables.base import ModelTable

from .models import MyModelClass

class MyModelTable(ModelTable):

    class Meta:

        model_class = MyModelClass

        # or as string, but this makes some additional db queries
        model_class = "mymodelclass"
```

and then *views.py*

```
from horizon_contrib.tables.views import IndexView

from .tables import MyModelTable

class IndexView(BaseIndexView):
    table_class = MyModelTable
    template_name = 'myapp/mymodel/index.html' # or leave blank
```

Note: for easy table inheritence we supports `model_class` directly on the table class

```
...

class MyModelTable(ModelTable):

    model_class = MyModelClass

...
```

Specifing columns and ordering

```
from horizon_contrib.tables import ModelTable

class MyModelTable(ModelTable):

    class Meta:
        columns = ("project", "issue", ..)
        order_by = ("id") # queryset.order_by(self._meta.order_by)
```

Note: order by is used for generic queryset for more customization override `get_table_data`

Custom columns

```
from horizon import tables
from horizon_contrib.tables import ModelTable

class MyModelTable(ModelTable):

    project = tables.Column('project', ..)

    class Meta:
        extra_columns = True # generates other columns within ``project``
        # default is False
```

Note: In the default state if we specified one column no other columns will be generated for this purpose set `extra_columns = True`

Load Data into Table

Note: This is main change against Horizon, but old way is still supported and it's only about overriding `get_data` on the DataTable View.

With Django model simply do this

```
class MyModelTable(ModelTable):

    def get_table_data(self):
        return self._model_class.objects.all().order_by("status__id")
```

PaginatedTable

tables.py

```
from horizon_contrib.tables.base import PaginatedTable

class MyModelTable(PaginatedTable):
```

```
    class Meta:

        model_class = MyModelClass
```

and then *views.py*

```
from horizon_contrib.tables.views import PaginatedView

from .tables import MyModelTable

class IndexView(IndexView):
    table_class = MyModelTable
```

PaginatedModelTable

this table combine ModelTable and Pagination

```
from horizon_contrib.tables import PaginatedModelTable

class MyModelTable(PaginatedModelTable):

    model_class = "mymodelclass"
```

and then *views.py*

```
from horizon_contrib.tables.views import PaginatedView

from .tables import PaginatedModelTable

class IndexView(IndexView):
    table_class = PaginatedModelTable
```

PaginatedApiTable

Table which implements standard Django Rest Framework pagination style.

You can declare manager if you use `PaginatedManager` class from `horizon_contrib.api` or just implement `get_page_data` method.

```
from horizon_contrib.tables import PaginatedApiTable

class MyApiTable(PaginatedApiTable):

    manager = api.helpdesk.tickets

def get_page_data(self, page=1):
    """returns data for specific page
    """
    self._paginator = self.manager.list(
        self.request,
        search_opts={'page': page})
    return self._paginator
```

and then *views.py*

```
from horizon_contrib.tables.views import PaginatedView

from .tables import PaginatedApiTable

class IndexView(PaginatedView):
    table_class = PaginatedApiTable
```

If you want loading data in view override `get_data` method.

```
from horizon_contrib.tables.views import PaginatedView

from .tables import PaginatedApiTable

class IndexView(PaginatedView):
    table_class = PaginatedApiTable

    def get_data(self):
        objects = []
        table = self.get_table()
        page = self.request.GET.get('page', 1)

        if table:
            try:
                objects = helpdesk.tickets.closed(
                    self.request, search_opts={'page': page})
            except Exception as e:
                raise e
            else:
                table._paginator = objects
        return objects
```

LinkedListColumn

Generates links from list of items.

```
extensions = LinkedListColumn(
    'extensions', verbose_name=_("Extensions"),
    url="horizon:location:hosts:update")

extensions = LinkedListColumn(
    ...,
    url="horizon:location:hosts:update", datum_pk='key', label='item.name')
```

Inheritance of the 'Meta' class

```
from horizon_contrib.tables import ModelTable

class IssueTable(ModelTable):

    subject = tables.Column('subject')

    class Meta:
```

```
        model_class = "mymodelclass"
        extra_columns = True

class UserIssueTable(ModelTable):

    class Meta(IssueTable.Meta):

        extra_columns = False
```

Forms

SelfHandlingModelForm

forms.py

```
from horizon_contrib.forms import SelfHandlingModelForm

class IssueCreateForm(SelfHandlingModelForm):

    class Meta:
        model = Issue
        fields = ['project', 'priority', 'description', 'due_date']
        widgets = {
            'description': Textarea,
            'due_date': DateTimeWidget(attrs={'id': "due_date"}, options=settings.
↪DATE_PAST_OPTIONS)
        }

    # handle it or leave blank or call super where is implemented basic logic for␣
↪saving models
    # but in many cases is not sufficient and we must override this
    def handle(self, request, data, model_class):

        model_instance = model_class.objects.get(id=data.pop("object_id"))
```

views.py

```
from horizon_contrib.forms import ModalFormView

class CreateView(ModalFormView):

    form_class = IssueCreateForm
    success_url = "horizon:redmine:..."

    template_name = 'redmine/issue/create.html'
```

or simple use our `CreateView` which is based on SelfHandlingForm but use Django ModelForm

here we can specified fields array which be used for modelform factory or our `form_class`.

CreateView

```
from horizon_contrib.forms import CreateView

class CreateView(CreateView):

    name = _('Create Whatever') # your "action" name

    form_class = None # your form
    template_name = 'horizon_contrib/forms/create.html' # your template

    success_url = ...
```

UpdateView

Nothing special here. This view is same as `CreateView`, but tries getting initial data

Modal Tabs

You can use standard Horizon Workflows, but for many scenarios we need custom tabs with simply describe.

```
from horizon_contrib.tabs import ModelFormTab, TableTab
from horizon_contrib.forms import SHMForm # shortcat for SelfHandlingModelForm

from .tables import NoteFormSetTable, DocumentTable

class IssueUpdateForm(SHMForm):

    class Meta:
        model = Issue

    def __init__(self, *args, **kwargs):

        request = kwargs.pop("request", None)
        issue = kwargs.pop("issue", None)

        super(IssueUpdateForm, self).__init__(*args, **kwargs)

        # CRISPY layout
        self.helper.layout = TabHolder(
            Tab(
                u"Issue",
                Div(
                    'project', 'priority', 'status',
                                'tracker', 'assigned_to', 'subject',
                    css_class="col-lg-6 field-wrapper"
                ),
                Div(
                    'start_date', 'due_date', 'description',
                    css_class="col-lg-6 field-wrapper"
                )
            ),
        )
        TableTab(
            u"Notes",
```

```
            table=NoteFormSetTable(request, data=journal_set.filter(notes__regex = r'.
↪{1}.*')), # only with notes
        ),

        documents = [..]

        self.helper.layout.extend([TableTab(
                u"Files",
                table=DocumentTable(request, data=documents),
            )])
```

Actions

Horizon has two types of actions

- TableActions
- RowActions

some actions can by used for both categories.

Filter Action

Simple filter action which provide initial for our `client-side` filtering. Server-side is not implemented for now, but is not too many work and it's in the plan.

default search in all fields

```
from horizon_contrib.tables.actions import FilterAction
```

or specify one field

```
from horizon_contrib.tables.actions import FilterAction

class MyFilter(FilterAction):

    fields = ['name', 'subject']

    lookups = ['project__name']
```

DeleteAction

Simple action based on horizon's `DeleteAction`. It's `BatchAction` for more detail see Horizon doc.

```
from horizon import tables
from horizon_contrib.tables import DeleteAction

class MyTable(tables.DataTable):

    class Meta:
        table_actions = (DeleteAction,)
        row_actions = (DeleteAction,)
```

Warning: For this time is not implemented in transaction !

CreateAction

There is nothing special it's only `LinkAction` with implemented `get_link_url`

```
from horizon_contrib.tables.actions import CreateAction
```

UpdateAction

There is nothing special it's only `LinkAction` with implemented `get_link_url`

```
from horizon_contrib.tables.actions import UpdateAction
```

Note: for more details and customization follow `UpdateView`

Packs of Actions

For less code is there some action packs

- `CD_ACTIONS` - `CREATE` and `DELETE` can be used for table and row actions
- `ROW_ACTIONS` - same as CD_ACTIONS, but with UpdateAction
- `TABLE_ACTIONS` - same as CD_ACTIONS but with FilterAction

```
from horizon import tables
from horizon_contrib.tables import ROW_ACTIONS, TABLE_ACTIONS

class MyTable(tables.DataTable)

    class Meta:
        row_actions = ROW_ACTIONS
        tables_actions = TABLE_ACTIONS
```

Warning: In default state these actions sets works only with our table classes !

UpdateColumnAction

This action is used for column as additional attribute and provide Ajax update power.

Optionaly can be provided form `field` with `widget`.

```
from horizon import tables
from horizon_contrib.tables.actions import UpdateAction

my_column = tables.Columns('my_column', update_action=UpdateAction)
```

ReactJS Integration

Why

For some customers we need render hundreds rows in one table with many columns and all with inline AJAX updates for each column. With AngularJS is too slow any sotrable actions, filters and initial render is unusable.

With ReactJS we can render thousands of rows without any lags.

Note: For now it's an implementation with many dysfunctions versus standard AngularJS. But ready for experimenting.

Beginnings

For this purpose is there initial implementation of ReactJS SortTable which is available for some scenarios.

In previous chapter we introduced how we work with tables and their views.

GenericView has one additionaly argument which specify used table. If we want render React table for our panel we can simply set `react=True` on `Panel` class and voila our index is rendered as ReactJS table as we used to.

As you can see in the url if we append `/react` to our url contrib render ReactJS table. For example our index `contrib/models/<my_class>/index/` and append `/react` finally we have `contrib/models/<my_class>/index/react/`

Usage

```
pip install xstatic-react
```

Add to `settings.py`

```
import xstatic.pkg.react

STATICFILES_DIRS = [
    ('lib', xstatic.main.XStatic(xstatic.pkg.react).base_dir),

]
```

- `/contrib/models/project/react` ..

In the Panel

```
from horizon_contrib.panel import ModelPanel
from horizon_redmine.dashboard import RedmineDashboard

class ProjectPanel(ModelPanel):
    name = "Projects"
    slug = 'projects'
    model_class = 'project'

    react = True

RedmineDashboard.register(ProjectPanel)
```

As table

```
from horizon_contrib.tables import ReactTable


class MyReactTable(ReactTable):

    ...
```

Note: For now we use the in-browser JSX transformer.

ReactJS DataTable

```
var Table = Reactable.Table;
var Tr = Reactable.Tr;
var Td = Reactable.Td;

var HorizonReactDataTable = React.createClass({
  getInitialState: function(){
    return {data: []};
  },
  render: function(){
    return (
    <Table
      id="{{ table.slugify_name }}"
      className="{% block table_css_classes %}table table-bordered table-striped␣
↪datatable {{ table.css_classes }}{% endblock %}"
      sortable={true}
      data={this.state.data}
      >
    {% for row in rows %}

      <Tr{{ row.attr_string|safe }}>
          {% spaceless %}
              {% for cell in row %}
                 <Td{{ cell.attr_string|safe }}>{%if cell.wrap_list %}<ul>{% endif %}{
↪{ cell.value }}{%if cell.wrap_list %}</ul>{% endif %}</Td>
              {% endfor %}
          {% endspaceless %}
          <Td>{{ row.render_row_actions }}</Td>
      </Tr>

    {% endfor %}

    </Table>
    )
  },
  componentDidMount: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
```

```
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
 }
});
/* here we expect url/json as data url */
React.renderComponent(<HorizonReactDataTable url="json"/>,
document.getElementById('{{ table.slugify_name }}'));
```

Note: Implementation uses `https://github.com/glittershark/reactable`

Column Filters

Some common filters

- timestamp_to_datetime
- nonbreakable_spaces
- unit_times
- join_list_with_comma
- join_list_with_newline
- status_icon

CHAPTER 3

Requires

- Django
- Horizon - part of OpenStack Dashboard

CHAPTER 4

Tested with

- Horizon 2012+ (Icehouse +)
- Django 1.4 +
- Python 2.6 +

CHAPTER 5

Installation

```
pip install horizon-contrib

pip install git+https://github.com/michaelkuty/horizon-contrib.git@develop
```

CHAPTER 6

Configuration

```
INSTALLED_APPS += ('horizon_contrib',)
```

Optionaly include `horizon_contrib.urls` with `namespace='horizon'`. This is only for generic functionality like a index,create,update,delete views.

```
from django.conf.urls import patterns, include, url

urlpatterns = patterns('',
    ...
    url(r'^contrib/', include('horizon_contrib.urls', namespace='horizon'), ),
    ...
)
```

Note: `namespace` is important for url `reverse`

CHAPTER 7

Django example

With Django model everythings works well without any code. Only navigate your browser to

- `/contrib/models/project/index`
- `/contrib/models/project/create`
- `/contrib/models/project/1/update`

For override behaviour see doc.

Note: `project` in url is model name

CHAPTER 8

Horizon example REST-API !

Dashboard structure:

```
my_dashboard
    |-- __init__.py
    |-- projects
        |-- __init__.py
        |-- models.py    # define data structure
        |-- managers.py  # load remote data
        |-- panel.py     # register namespace
    |-- dashboard.py
```

Your `models.py`

```
from horizon_contrib.api import APIModel
from horizon_contrib.common import register_model

class Project(APIModel):

    name = models.CharField('id', primary_key=True)   # default primary is id
    description = models.CharField..
    ...

    objects = Manager()   # see below

    class Meta:
        verbose_name = 'Project'
        abstract = True

register_model(Project)   # supply Django Content Type framework
```

New `managers.py`

```
from horizon_contrib.api import Manager

class Manager(Manager):
```

```
    def all(self, *args, **kwargs):
        return self.request('/projects')
```

Finally `panel.py`

```
from horizon_contrib.panel import ModelPanel
from horizon_redmine.dashboard import RedmineDashboard

class ProjectPanel(ModelPanel):
    name = "Projects"
    slug = 'projects'
    model_class = 'project'

RedmineDashboard.register(ProjectPanel)
```

navigate your browser to `/contrib/models/project/index` ! or `/contrib/models/project/create`

Warning: This project depends on Horizon library, but isn't in the requirements ! You may use `leonardo-horizon` or openstack horizon.

CHAPTER 9

Read more

- http://horizon-contrib.readthedocs.org
- https://www.djangoproject.com/
- https://github.com/openstack/horizon
- http://docs.openstack.org/developer/horizon/

Indices and tables

- genindex
- modindex
- search