CWF Documentation

Release 0.1

March 22, 2014

Contents

1 Tests **3**

2 A CWF Section **5**
- 2.1 Configuring a section . . . 6
- 2.2 Adding child sections . . . 6
- 2.3 Section datastructure . . . 7

3 Url for a section **9**
- 3.1 How the url is interpreted . . . 9
- 3.2 Section url options . . . 9

4 Views for a url **11**
- 4.1 Section view options . . . 11

5 Generated Menus **15**
- 5.1 Section menu options . . . 15
- 5.2 Creating the menu . . . 18

6 Sections **19**

7 Views **21**
- 7.1 Base View . . . 21
- 7.2 Calling a view . . . 22
- 7.3 View State . . . 22
- 7.4 Cleaning View kwargs . . . 22
- 7.5 Getting result for a view . . . 23
- 7.6 Rendering a view . . . 23
- 7.7 Special Views . . . 24
- 7.8 Rendering Helper . . . 24
- 7.9 Admin Views Urls . . . 25
- 7.10 Redirect Helper . . . 25
- 7.11 Menu Rendering . . . 26

8 Splitter **29**
- 8.1 Import helpers . . . 29
- 8.2 Website declaration . . . 31
- 8.3 Creating a website . . . 32

9 Binaries **39**

9.1 cwf-manager . . . 39
9.2 cwf-debugger . . . 40

10 Admin **41**
10.1 Buttons . . . 41
10.2 How it works . . . 42
10.3 Button View . . . 43

11 Template Tags **45**
11.1 varset . . . 45
11.2 wrapped . . . 45

12 Templates **47**
12.1 Admin Templates . . . 47
12.2 Menu Templates . . . 50
12.3 Available Templates . . . 51
12.4 Including Templates . . . 51

13 CWF **53**
13.1 Changelog . . . 53
13.2 Getting Started . . . 54
13.3 Features . . . 54

Python Module Index **55**

As with any python project, it is recommended you use virtualenv.

You may setup virtualenv as you like, I use virtualenvwrapper:

```
$ sudo pip install virtualenvwrapper

# Change where your virtualenvs are
$ echo "export WORKON_HOME=$HOME/venv" >> ~/.zshrc

# Make sure your ~/.bashrc or ~/.zshrc has the virtualenvwrapper script
$ echo "[[ -s /usr/local/bin/virtualenvwrapper.sh ]] && source /usr/local/bin/virtualenvwrapper.sh" >

$ source ~/.zshrc

# Create the virtualenv
$ mkvirtualenv cwf
```

Once your virtualenv is created, then you activate it before doing anything in that virtualenv:

```
# If you are using vanilla virtualenv
$ source <venv_path>/bin/activate

# Or if you are using virtualenvwrapper
$ workon cwf
```

Once you're inside your virtualenv (or without that if you are installing cwf system wide):

If you want to install it from pypi, then do the following:

```
$ pip install cwf
```

Otherwise, if you are installing it from source

```
$ pip install .
```

If you're developing for cwf, then it's recommended you do that with the "-e" flag (behaves like `python setup.py develop`):

```
# If you want to constantly make changes to cwf
# Remove the need to keep reinstalling it
$ pip install -e .
```

CHAPTER 1

Tests

Running the tests after *Installation* is really easy:

```
$ pip install -r test_requirements.txt
$ ./test.sh
```

CHAPTER 2

A CWF Section

In Django, you define the urls in your site by creating urlpatterns.

This is usually done by hand, but this means that if you want to generate a menu system on your site from your urls, you have to essentially duplicate the structure of your website.

CWF instead provides classes that can be used to generate both the urlpatterns and a data structure that can be used to generate a menu from.

Usage looks like:

```
from cwf.sections import Section

section = Section().configure(module="webthing.views")

section.first(name="root").configure(target='base')
section.add('login', name='login').configure(target='login')
section.add('logout', name='logout').configure(target='logout')

numbers = section.add('numbers')
numbers.add("one").configure(target="one")
numbers.add("two").configure(target="two")

section.add('\d+').configure(''
    , target='digits'
    , match='digit'
    )

urlpatterns = section.patterns()
```

For this very simple case, it would be the same as:

```
from django.conf.urls import patterns

urlpatterns = patterns(''
    , (r'^$', 'webthing.views.base', name="root")
    , (r'^login/$', 'webthing.views.login', name="login")
    , (r'^logout/$', 'webthing.views.logout', name="logout")
    , (r'^numbers/one/$', 'webthing.views.one')
    , (r'^numbers/two/$', 'webthing.views.two')
    , (r'^(?P<digit>\d+)/$', 'webthing.views.digits')
    )
```

2.1 Configuring a section

Every section has an `options` object that knows what options are available , what values those options can be, and some helpful methods for determining information about the section from those options.

You can configure these options by using "section.configure".

For example:

```
tests = section.add("tests").configure(''
    , kls = 'testViews'
    , redirect = 'register'
    , needs_auth = True
    )
```

Note: Positional arguments to configure are ignored.

There are options available for how the section appears in the *urlpatterns* ; what *view* is used; and how it appears in the *menu* that can be generated.

2.2 Adding child sections

Sections represent only one part of the url. To make urls with more than one part to it, we need to build up a heirarchy of sections. We can do this by adding child sections to each section.

When we then did "section.first" and "section.add" we created new sections with the `url` and `name` as passed into those functions and these new sections got the first section as it's `parent`. The first section also records these new sections on itself.

Note: section.first() behaves exactly as section.add() except the section will consider this child section to be first before any other child sections and there can only be one "first" section.

You can also add sections via the `merge`, `adopt` and `copy` functions on the section.

Note: Creating a section this way will copy most options from the parent onto the child.

2.2.1 Adding a Child

The "section.first" and "section.add" methods are shortcuts to "section.add_child" where the only difference is "section.first" will call "add_child" with "first=True"

"section.first" will also default url to an empty string, whereas "section.add" will complain if no url is provided.

These functions will return the child that was added.

2.2.2 Merging children

If you do a "section.merge(another_section)", then you will add the children from `another_section` onto `section`.

If you specify `take_base=True`, then it will also take the first child of `another_section` and put it onto `section` as the first child.

Note: merging always does a *copy*.

2.2.3 Adopting children

You may do a "section.adopt(other_section1, other_section2)" and it will change the parent of these children to `section` and add them as children of `section`.

If you also specify "clone=True", then it will use *section.copy* to make a clone of the children before adding them as children.

You may also specify as keyword arguments `consider_for_menu` and `include_as` and these will be used when putting the child onto the `section`. See *Section datastructure* for what that means.

2.2.4 Copying children

Doing a "section.copy(other_section)" will make a *clone* of `other_section` and recursively *merge* the children of `other_section` onto the clone before adding the clone as a child of `section`.

It will also take in `consider_for_menu` and `include_as` (see *Section datastructure*)

2.2.5 Cloning children

You can use the "section.clone()" method to create a clone of the `section`.

It will create a new Section object with the `url`, `name` and `parent` of the `section` being cloned and then copy a clone of "section.options" onto the clone.

It will not pass on any reference or clone of the children from the original section onto the clone.

2.3 Section datastructure

The section has two attributes it uses to hold it's children:

`_base` This holds a single *item*. And is what the section considers as the "first" child.

`_children` An array of *items*.

2.3.1 Section Item

There are three pieces of information that is required to make it easy for us to generate a menu from this information: The section itself, whether to include the section in the menu; and what to include the section as if it needs to be included as anything special.

To achieve this, each child of a section is held in an instance of `cwf.sections.section.Item`. This is an object that holds `section`, `consider_for_menu` and `include_as`.

This is so that sections can use the same sections as children but have them appear in the menu and url scheme differently depending on which parent owns them.

CHAPTER 3

Url for a section

Sections will appear in the urlpatterns based on the url, name and parent it has on itself; and depending on some of the *options* it has.

The url and name attributes are assigned when the section is created , and parent is assigned implicitly when you add *children* to a section.

3.1 How the url is interpreted

The url for each section is built when the urlpatterns are created and are generated by concatenating the urls from the lineage of parent sections to the section we're adding a url for.

If there is no parents and our section's url is None then it is replaced with '.*'.

If there are no parents and our section's url is an empty string then it is replaced with '^$'.

Otherwise, all duplicate slashes are remove , it is prefixed with a '^' and forced to end with '/$'.

If you set catch_all to False, then it won't append the url with a / or $

3.2 Section url options

You can affect how the section is added via an include via the app_name and namespace options into "section.configure"

See the *Splitter section* for when that would happen.

Configure also provides the match option which will make that section appear as a named regex group in the url:

```
# This section here has configured match to "blah"
Section(r'\d+').configure(match="blah")
```

So for this section, it's url part will look like:

```
r"(?P<blah>\d+)"
```

Note: You can see what url part a section will have by doing:

```
>>> from cwf.sections import Section
>>> from cwf.sections.pattern_list import PatternList
>>> section = Section("\d+").configure(match="blah")
```

```
>>> PatternList(section).url_part()
r'(?P<blah>\d+)'
```

3.2.1 Ommitting a section from urlpatterns

A section will be ommitted entirely from the urlpatterns if it has no configured *view* but all it's children will still appear in the urlpatterns.

For example:

```
from cwf.sections import Section

section = Section().configure(module="webthing.views")

numbers = section.add('numbers')
numbers.add("one").configure(target="one")
numbers.add("two").configure(target="two")

urlpatterns = section.patterns()
```

Is equivalent to:

```
from django.conf.urls import patterns

urlpatterns = patterns(''
    , (r'^numbers/one/$', 'webthing.views.one')
    , (r'^numbers/two/$', 'webthing.views.two')
    )
```

CHAPTER 4

Views for a url

Included in the *options* you have available when you are configuring your sections are some available for specifying what view should be used for that section.

4.1 Section view options

We have options available for specifying whether to redirect to a url or execute a particular view callable; as well as whether to bypass the view altogether with access restriction.

You may also pass extra keyword arguments into the view that gets called via the `extra_context` option.

4.1.1 Setting the view

There are four options availbe to set what view should be invoked when a particular section is accessed:

`redirect` Overwrites the other options and will mean the section redirects to this address. This can be a callable that takes in a request object.

If the url specified doesn't begin with a slash, then it will be made relative to the `request.path`.

All duplicate slashes will be removed from the url before it is used for the redirect.

`target` If this is a callable object, then it is used as the view without any consideration of the other options.

If it is a string, then the *dispatcher* is used.

If it is set to None, (and the section has no redirect) , then the section is considered to have no view

`kls` and `module` These are only considered if target is a string and all three values are used with the *dispatcher*

Note: Sections that don't have a configured view won't appear in the generated `urlpatterns`.

4.1.2 Section Dispatcher

CWF provides an object called the `dispatcher` that is used as the view callable when the `target` is set as a string.

Django provides the ability to set a dictionary of options in the urlpatterns that is passed into the view when it gets called. CWF uses this dictionary to set the `target` and a `kls` values (from combining the `kls` and `module` options) and the dispatcher will use these two values to find the callable at request time.

It does this by creating (and caching) an instance of the `kls` value passed into the dispatcher and then using "getattr(instance, target)" to get the view to use.

This `kls` value is determined by looking at the `kls` and `module` options on the section.

If both are None, then no class can be determined.

If the `kls` is already an object then it is used and `module` is ignored.

If the `kls` is a string and the module is not defined , then the string is sanitised (leading and trailing dots are removed and invalid import names raise exceptions)

If module is a string, then it is sanitised and then concatenated with kls (using a dot as a seperator) and returned.

If module is an object, then the parts of the kls are used with getattr to get the view. So if `kls` option is "some.thing", the `kls` value will be found from "getattr(getattr(module, 'some'), 'thing')".

4.1.3 Forcing a 404 for a url

If the `exists` or `active` options evaluate to `False` then that section will return a 404 when it is accessed.

Note: These can be set to a callable that takes in the request object.

For example:

```
from cwf.sections import Section
from datetime import date

section = Section().configure(module="webthing.views")

def only_at_christmas(request):
    """Conditional that only returns True if it's christmas"""
    today = date.today()
    return today.month == 12 and today.day == 25

christmas = section.add('christmas'
    , target="christmas"
    , active=only_at_christmas
    )

urlpatterns = section.patterns()
```

In this example, the "^christmas$" urlpattern always exists, but it will always return a 404 unless it's christmas day , where it will use the `webthing.views.christmas` view.

The `exists` option behaves exactly as the `active` option , and there is only a logical difference between the two as determined by you as the developer of the website.

4.1.4 Admin only views

You can specify that a view needs authorization to be accessed via the `needs_auth` option.

This can be set to either a boolean, string, list of strings or a callable that takes in the request object.

If it set to a boolean or callable, then `True` means you can only access the view if `request.user.is_authenticated()` evaluates to True.

If it is set to a string, then the user must be authenticated and have the particular permission specified by the string.

If it is set to a list of strings, then the user must be authenticated and have all the permissions specified.

CHAPTER 5

Generated Menus

One of the points of CWF is that it provides an API you can use to generate your urlpatterns and menu system from without repeating yourself.

So part of the *options* available is about saying how the sections are represented in the menu system.

5.1 Section menu options

The options available that directly affect how a section appears in the menu are as follows:

admin Hardcode that a section should be displayed as only available via admin privelege.

display Whether to show this section at all.

Note that this option will not propogate to it's children if the `propogate_display` option is False.

alias What name is displayed to the end user for this section.

values Allows you to have this section appear multiple times with different values. See *Section Values*

promote_children Whether to display the children of this section at the same level as this section. See *Promoted Sections*

propogate_display Whether the `display` option should be propogated to a section's children.

5.1.1 Section Values

There exists a situation where a section can contain multiple values and you want your menu to display all those possible values.

CWF provides the `values` option for specifying this.

This option should be set to an object with a `get_info` method that returns a list of [(url, alias), (url, alias), ...] where each pair represents another value.

To assist with this, CWF provides the `cwf.sections.Values` object.

Note: This functionality will also work with child sections such that each value will get all the children of the original section and those children may have their own collection of values and so on.

The `Values` object takes the following options:

values The values to use.

This can be a list of [value, value, value, ...].

Or it can be a callable lambda ((request, parent_url_parts, path)) : [value, value, value, ...]

Where request is the `request` object, `path` is a list of the parts that makes up the current url. And `parent_url_parts` is all the parts in the url leading up to the parent of the current section.

each A callable lambda ((request, parent_url_parts, path), value) : (url, alias)

If this isn't set, then the `alias` and `url` will both be set to the `value` provided by the `values` option.

sorter If it's Falsey then no sorting will occur.

If it's Truthy then the values are sorted.

If it's a callable, then the values are sorted and `sorter` is used as the second argument to the python `sorted` function.

as_set Say whether to remove duplicate values before we work out the `alias` and `url` for each value.

sort_after_transform Whether to sort after or before we determine the `alias` and `url` for each value.

For example:

```
section = Section().configure(''
    , target = 'nlsBase'
    , alias  = 'NewsLetters'
    )

ye = section.add('\d{4}').configure(''
    , target = 'newsYear'
    , match  = 'year'
    , values = V(
        lambda _ : [date.year for date in News.objects.dates("pubDate", "year", "DESC")]
      , as_set = False
      )
    )

item = ye.add('\d+', name="newsitem").configure(''
    , target = 'newsItem'
    , match  = 'item'
    , values = V(
        lambda (r, parent, p) : News.objects.filter(pubDate__year=parent[-1]).order_by('-pubDate')
      , lambda _, value : (value.pk, unicode(value))
      , as_set  = False
      )
    )

item.add("example1").configure(target="example")
item.add("example2").configure(target="example")
```

This here will be used by the menu generation to produce a menu structure that looks like:

```
2012
    News Item 1
        example1
        example2
    News Item 2
```

```
        example1
        example2

2011
    News Item 3
        example1
        example2
    News Item 4
        example1
        example2
    News Item 5
        example1
        example2
```

Depending on the values in the database table being used here.

Note: CWF doesn't implement any kind of caching yet, so these functions will be called every time the menu is generated.

5.1.2 Promoted Sections

CWF provides the `promote_children` *configuration* option for saying that the child of a section should appear in the menu at the same level as that section.

So for example, say we want to have common settings grouped but keep everything at one level:

```
section = Section().configure(''
    , target = 'base'
    )

group1 = section.add("group1").configure(
    , module="webthing.groupone"
    , promote_children=True
    )
group1.add("one").configure(target="one")
group1.add("two").configure(target="two")

group2 = section.add("group1").configure(''
    , module="webthing.grouptwo"
    , promote_children=True
    )
group2.add("three").configure(target="three")
group2.add("four").configure(target="four")
```

Then we'll get a menu that looks like:

```
one
two
three
four
```

Instead of:

```
group1
    one
    two
group2
```

```
three
four
```

Note: Currently there is a limitation that sections that promote their children are unable to contribute to the url and therefore, children of these sections cannot use the values used by the parent section.

5.2 Creating the menu

You use the `cwf.views.menu:Menu` class to generate the menu system from a section for a particular request.

```
from cwf.views.menu import Menu

def my_view_function(request):
    menu = Menu(request, request.section)

    # Add menu to your template context
    # And do everything else as normal
```

Note: If you add your views using Section, then the view you provide will be wrapped in a function that attaches that `section` to the `request` object before calling your original view

Then in your template:

```
# For the global navigation
{% include "cwf/menu/base.html" with menu=menu.global_nav children_template="cwf/menu/base.html" ign

# For the side navigation
{% include "cwf/menu/base.html" with menu=menu.side_nav children_template="cwf/menu/base.html" %}
```

To understand how to make these templates available and how to customise them , you should read the section on *Templates*.

CHAPTER 6

Sections

CWF was built for and based on the idea that your website can be split into several sections that have their own urls and views; and that the menus for each section map directly to your urls.

Provided under the `cwf.sections` namespace is functionality that can be used to define these sections.

The idea is then you either include the urlpatterns created just as if you defined them by hand, or use the *cwf.splitter* logic to combine the sections into your website.

The following pages guide you through what is possible here:

A CWF Section An introduction to what a Section is.

Url for a section The options available to alter the url for a section.

Views for a url How to say what views should be invoked by a url.

Generated Menus How to alter the look of the menu that is generated.

CHAPTER 7

Views

7.1 Base View

class `cwf.views.base.View`

Base class for cwf views

Django only requires a callable object to represent a view, which means you either provide a function, or an instance of a class that implements the special __call__ method.

Django provides it's own class based views where one class represents one view.

CWF class based views were created before this idea came around and does it a little differently in that it represents a collection of views and which view is used when the instance of the class is called is determined by the `target` parameter which is a string representing the name of the method to use.

This means you can use CWF view as follows:

```
# in webthing.somesection.views

from cwf.views import View
class MyAwesomeView(View):
    def thing(self, request):
        return "path/to/template.html", {}
```

To define the view and either the following with a *Section*

```
# in webthing.somesection.urls

from cwf.sections import Section

section = Section('').configure(''
     , kls = "MyAwesomeView"
     , target = "thing"
     , module = 'webthings.somesection.views'
     )

urlpatterns = section.patterns()
```

Or if you prefer a more manual approach to your urlpatterns:

```
# in webthing.somesection.urls

from webthing.somesection.views import MyAwesomeView
from django.conf.urls.defaults import *
```

```
urlpatterns = patterns(''
    , ( '^$', MyAwesomeView(), {'target':'thing'})
    )
```

Note: You can create as many or as little instances of a View class as you wish as long as you don't put any state on the instance.

7.2 Calling a view

`View.__call__(request, target, *args, **kwargs)`

When an instance of the view class is called, it will do the following:

- Create *view state*
- *Clean kwargs*
- Get a *result* to render
- *Render* the result and return.

7.3 View State

Rather than keeping state on the View instance, CWF will use the `get_state` method to create an object that you can use to store state for the request.

The view will create this `state` object when the view is being called and make it available from `request.state`.

`View.get_state(request, target)`

Return an object that can be used to store state for a request.

For convenience, this object behaves like a Javascript object (supports both dot notation and array notation for accessing and setting variables).

When it's created, it is initialized with some values:

menu If we got here via a CWF Section, then we will be able to create a `cwf.views.menu.Menu` object from that section.

path The path of the request with no leading, trailing; or duplicate slashes.

target The target being reached on the class.

section The CWF section that is executing this view (if one was used)

base_url `request.META.get('SCRIPT_NAME', '')`

7.4 Cleaning View kwargs

All keyword arguments to the view will be cleaned as according to the `clean_view_kwarg` method before being passed into the target.

`View.clean_view_kwargs(kwargs)`

Replace all kwargs with the result of passing them through `clean_view_kwarg`

View.**clean_view_kwarg**(*key*, *item*)
Clean a single view kwarg

If it's a string, make sure it has no trailing slashes , otherwise just return it as is.

7.5 Getting result for a view

View.**get_result**(*request*, *target*, *args*, *kwargs*)
Takes the `request` object, the `target` that is been called and any positional arguments and *cleaned* keyword arguments and returns a result that will be *rendered*.

If the instance has an `override` method , then it will pass all those arguments in that function and return it's result.

Otherwise:

Use `View.has_target()` to check if the `target` exists on the instance and raise an exception if it does not exist.

If the `target` does exist, then pass it into `View.execute()` along with the `request`, `args` and cleaned `kwargs` to get a result.

If the result is a callable then call it with the `request` and return what that gives back.

Otherwise, return the result as is.

View.**override**(*request*, *target*, *args*, *kwargs*)
Not defined by default on the BaseView

If implemented, then `get_result` will use this as a short circuit to skip the normal machinery

View.**execute**(*target*, *request*, *args*, *kwargs*)
Execute target with the request, args and kwargs.

By default this means getting a callable for the `target` using `View.get_target()` and calling it with the `request`, `*args` and `**kwargs`.

View.**has_target**(*target*)
Return whether view has specified target

By default, just use hasattr on the instance

View.**get_target**(*target*)
Return callable for the `target` from this instance.

By default, just use getattr on the instance.

7.6 Rendering a view

View.**rendered_from_result**(*request*, *result*)
If the result being rendered is `None`, then a `Http404` will be raised.

If the result is a list or tuple of two items, then it assumes this list represents `(template, extra)` where `template` is the name of the template to render and `extra` is any extra context to render the template with.

If `template` is None, then `extra` is returned, otherwise it uses the `cwf.views.rendering.Renderer.render()` to render the template and context.

If the result to render is not a two item tuple or list, then it just returns it as is.

7.7 Special Views

Subclasses of `cwf.views.base.View` that override the `cwf.views.base.View.execute()` method.

The `execute` method is used to get a view for the `target` to render and call it.

These classes overwrite this behaviour to restrict or modify the result of calling the target.

7.7.1 Staff View

class `cwf.views.views.`**`StaffView`**
Restrict to staff members

`execute`(*target*, *request*, *args*, *kwargs*)
Use the django provided staff_member_required decorator to ensure only staff members can access any target on this class.

7.7.2 Local Only View

class `cwf.views.views.`**`LocalOnlyView`**
Restrict to local users only

`execute`(*target*, *request*, *args*, *kwargs*)
Raise a 404 if the Remote Address of the user is not 127.0.0.1.

Otherwise proceed as normal.

7.7.3 JSView

class `cwf.views.views.`**`JSView`**
Convert target output into a json response

`execute`(*target*, *request*, *args*, *kwargs*)
Assume the result of calling the target returns `(template, data)`.

Proceed to return the data as a `json response`

7.8 Rendering Helper

class `cwf.views.rendering.`**`Renderer`**
Stateless class that simplifies usage of Django machinary for creating HttpResponse objects

An instantiated instance of this class is provided from `cwf.views.rendering.renderer`

`http`(**args*, ***kwargs*)
Return a HttpResponse object with the args and kwargs passed in

`json`(*data*)
Return HttpResponse object with data dumped as a json string and a 'application/javascript' mimetype

`raise404`()
Raise a Http404

`redirect`(*request*, *address*, **args*, ***kwargs*)
Return a HttpResponseRedirect object

render (*request*, *template*, *extra=None*, *mime='text/html'*, *modify=None*)
Create a template, give it context and display as some mime type

Using a RequestContext object provided by `self.request_context`

If modify is provided and is callable then it will be used to modify the rendered template before creating the HttpResponse object

request_context (*request*, *extra*)
Create a RequestContext object from the request and extra context provided.

If request has a `state` member, that will be used as default context , otherwise an empty dictionary is used, which is updated with the `extra` context provided.

simple_render (*template*, *extra*)
Return the string from rendering specified template with a normal Context object

xml (*data*)
Return HttpResponse object with data and a 'application/xml' mimetype

7.9 Admin Views Urls

Getting the url to a view in the admin section requires using ContentType and urlresolvers and is less than concise.

So, to assist with creating these urls, CWF provides `AdminView`.

class `cwf.views.admin_views.`**AdminView**
Stateless Object that knows how to get url to an admin view

classmethod add_view (*obj*)
Admin add view for a particular model

classmethod change_list (*obj*)
Admin list view for a particular model

classmethod change_view (*obj*)
Admin change view for a particular instance of an object

7.10 Redirect Helper

class `cwf.views.redirect_address.`**RedirectAddress** (*request*, *address*, *relative=True*, *carry_get=False*, *ignore_get=None*)
Helper to determine where to redirect to.

Parameters

- **request** – Object representing current Django request.
- **address** – String representing the address to modify into a redirect url
- **relative** – If we should be redirecting relative to the current page we're on
- **carry_get** – If we should use the GET parameters from the current request in our redirect
- **ignore_get** – List of GET parameters that should be ignored if carry_get is True

7.10.1 Creating the redirect

RedirectAddress.**modified**
Returns the result of `self.modify(unicode(self.address))`

RedirectAddress.**modify**(*address*)
Return a modified version of the address passed in as the redirect url.

Uses the following methods in a pipeline of sorts (in this order):

- `joined_address()`
- `strip_multi_slashes()`
- `add_get_params()`

7.10.2 Information

RedirectAddress.**base_url**
Get base url from request.state if request has an attached `state`.

Otherwise, return `request.META.get('SCRIPT_NAME', '')`

RedirectAddress.**params**
Return dictionary of key value for GET params from `self.request`.

If not `self.ignore_get` then just return `request.GET`

Otherwise, return `request.GET` minus any values whose key is in `self.ignore_get`

7.10.3 Utility

RedirectAddress.**strip_multi_slashes**(*string*)
Replace multiple slashes with single slashes in a string

RedirectAddress.**root_url**(*address*)
Determine if address is a root url (starts with slash)

RedirectAddress.**add_get_params**(*address*)
If `self.carry_get` is False, then just return the address as is.

Otherwise, urlencode the result of `params` and return a string that joins address and the parameters.

RedirectAddress.**joined_address**(*address*)
If address is a `root_url()`, then `join` the address with base_url.

Otherwise, if `self.relative`, then `join` the address with `self.request.path`

If the address is not a root url and `self.relative` is False, then return as is

RedirectAddress.**url_join**(*a*, *b*)
Helper to join two urls such that there is only one `/` between them.

7.11 Menu Rendering

class cwf.views.menu.**Menu**(*request*, *section*)
Knows how to get the information required to render the navigation from a *section*.

Assumes a top nav with one selected item.

And a side nav that is everything under the selected top nav item

If you use a `cwf.views.base.View` then you will have one of these in `request.state` (via the `get_state` method)

This is best used with the CWF *Menu Templates*.

global_nav()
Find the root ancestor of the current section and return a list of navs for just the top level sections.

side_nav()
Find the top nav of the current section and return the list of navs representing the children for that top nav.

These children will recursively have `children` of their own that will go as far down as the section itself and any siblings of this sections' parent.

CWF provides some helpful classes to help with defining view functionality , which can be found in `cwf.views`.

These include:

Base View A Base class for defining view classes.

Special Views Some specialisations of the Base view class

Rendering Helper A rendering helper

Admin Views Urls Helper for getting urls to admin views

Redirect Helper Helper for working with redirects

Menu Rendering Logic for rendering a Menu

CHAPTER 8

Splitter

8.1 Import helpers

There are three helpers under `cwf.splitter.imports` that you can use:

cwf.splitter.imports.steal This lets you inject variables from several files into your current file without relying on magic "from somewhere import *".

cwf.splitter.imports.inject This lets you inject some object into import space.

cwf.splitter.imports.install_failed_import_handler This provides a wrapper around the default __import__ logic that makes autoreload logic for development servers reload modules that never import properly.

8.1.1 steal

A django settings file tends to get very large and a bit unweildy.

You can use this to split your `settings.py` into many files that you combine together.

For example, let's say you have the following folder structure:

```
settings.py
settings_files/
    inclusions.py
    logging.py
    cache.py
    other.py
```

Then you can have the following in your `settings.py`:

```
from cwf.splitter.imports import steal

this_dir = os.path.dirname(__file__)
settings_dir = os.path.abspath(os.path.join(this_dir, 'settings_files'))
steal('inclusions', 'logging', 'cache', 'other'
    , folder=settings_dir, globals=globals(), locals=locals()
    )
```

`steal` will use execfile to insert the variables from those files into the `globals` and `locals` that your provide.

8.1.2 inject

Inject is a special beast that will create a finder object that gets placed into sys.meta_path

What this means is that you can write something like:

```
from cwf.splitter.imports import inject

try:
    import blah
    assert False, "Blah shouldn't exist"
except ImportError:
    assert True

obj = {"one":1, "two":2}
inject(obj, "blah")

try:
    import blah
    assert blah.one == 1
    assert blah.two == 2
    print "successfully injected blah"
except ImportError:
    assert False, "Blah should have been injected"
```

Note: There is a limitation to this in that all packages leading up your new import path must already exist and be folders.

(The limitation is an implementation detail that I haven't found the time to rectify yet)

So if you inject into "blah.things", then "blah" must already exist in your PYTHONPATH and be a folder with an `__init__.py` for this to work.

8.1.3 install_failed_import_handler

When you start a development server that auto reloads for you (for example, the werkzeug powered one *provided* then files will only be reloaded if they are inside `sys.modules`.

It so happens that if a module fails to import, then it won't end up in `sys.modules` and so when you edit such a file to not fail on import, the reloader won't see that it has changed and ignore it.

To get around this, CWF provides `cwf.splitter.imports.install_failed_import_handler` that will wrap the default `__import__` such that any module that fails to import will get a fake module put into `sys.modules` in it's place so that the reloader knows to check that file.

Installation is as simple as:

```
from cwf.splitter.imports import install_failed_import_handler
install_failed_import_handler()
```

It will consider either a `SyntaxError` or `ImportError` as conditions for when a fake version of it should go into `sys.modules`. Regardless of what exception is raised, if any, it will always be reraised so that you are aware when this happens.

8.2 Website declaration

Django doesn't provide any facilities out of the box for splitting up your website into distinct parts with their own urls, views, admin and models.

Instead you must manually import these things in the `package.views` , `package.admin`, `package.views` and `package.models` of your project if it is best for your website to be split up this way.

CWF provides under `cwf.splitter.website` and `cwf.splitter.parts` classes that provide the wiring that allows you to define your website with a folder per part of the site and declaritevely wire them all together into one site.

So, assuming your project is structured as:

```
webthings_main/
    __init__.py
    index/
        __init__.py
        urls.py
        views.py
    news/
        __init__.py
        models.py
        urls.py
        views.py
    events/
        __init__.py
        models.py
        urls.py
        views.py
    photos/
        __init__.py
        models.py
        urls.py
        views.py
```

You can create a `Website` representing these different parts:

```
from cwf.splitter.website import Website
from cwf.splitter.parts import Part

website = Website('webthings_main'
      , Part('index', first=True)
      , Part('news')
      , Part('events')
      , Part('photos')
      )
```

The website object takes in the package where all the parts are defined as importable modules - `webthings_main` in this case; and `Part` objects for the parts of the website we want to include.

The `Part` object holds the name of the part to import and any kwargs that are eventually used when the part is used as a *section*.

Note: The `Part` object also takes in an `active` keyword, which defaults to True. The `Website` object will use this to determine which parts should have its `urls` added to the website.

The `models` and `admin` of a section will be imported regardless of the value of the `active` property.

This object will know how to import the section (part.do_import) and load urls, admin and models from it (part.load('urls'), etc).

The website object will create a `Parts` object that will hold the collection of `Part` objects provided.

The Parts collection has three main responsibilities:

parts.load_admin() Import `part.admin` for all the parts that have an `admin.py` so that any django admin registration logic may be fired.

parts.models() Load all the `part.models` for all the parts that have `models.py` and return a dictionary of all {name:model} for all the models it finds.

> **Note:** For this to work, each `models.py` must define a `__all__` variable with either the names of the models as strings or the model objects themselves.
>
> Also, each model must have `Meta.app_label` set to the same package you provided to the `Website`.

parts.urls Create a *Section* object and add the `section` from each `part.urls` as *children* to root section.

It will then return {'site':site, 'urlpatterns':site.patterns()} where `site` is the root `Section` it just created.

> **Note:** These sections will be added to the urlpatterns using the Django include function.

Website will use this functionality to import the admin logic, *inject* the `models` into `package.models` and *inject* the `site` and `urlpatterns` into `package.urls`.

CWF provides the ability to seperate your website into multiple sections that you can then stitch together.

The following is provided to allow for this:

Import modifiers Functions to help with injecting values into and stealing values from import paths.

Website declaration Classes that let you specify what makes up your website.

8.3 Creating a website

Below is a slightly stripped down version of the how I setup the website that was the inspiration for CWF.

At it's core you have `webthings` and `webthings_main` in your import path where `webthings_main` knows about each part of your website , and `webthings` knows how to combine together the parts of `webthings_main` and what settings to set for the website to work.

So, given the following structure:

```
# Where the logic of the site actually goes.
# each urls.py here uses CWF Section logic
webthings_main/
    __init__.py
    index/
        __init__.py
        urls.py
        views.py
    news/
        __init__.py
        models.py
```

```
        urls.py
        views.py
    events/
        __init__.py
        models.py
        urls.py
        views.py
    photos/
        __init__.py
        models.py
        urls.py
        views.py
    utils/
        template_loaders.py
    templates/
        errors/
            404.html
            500.html

# All the configuration for the website is in a seperate module
webthings/
    __init__.py
    config/
        __init__.py
        common/
            __init__.py
            settings.py
            site.py
            urls.py
        settings/
            __init__.py
            inclusions.py
            logging.py
            testing.py
            other.py
    wsgi/
        prod.py
```

Note: For cwf to be able to put all your models under the `webthings_main.models` namespace, all your models must have `Meta.app_label` set:

```
from django.db import models

class AwesomeModel(models.Model):
    [..]

    class Meta:
        app_label = 'webthings_main'
```

And each `models.py` must have `__all__` explicitly set to all the models contained within:

```
__all__ = [AwesomeModel]
```

webthings.__init__

```
from cwf.splitter.imports import install_failed_import_handler
import sys
import os
```

```
def project_setup():
    # Install failed import handler
    install_failed_import_handler()

    # Setup the site
    from webthings.config.common import site
```

webthings.config.common.site

```
from cwf.splitter.website import Website
from cwf.splitter.parts import Part as P
from cwf.splitter.imports import inject
from cwf.sections import Section

import os

########################
###   SETTINGS
########################

from webthings.config.common import settings

class theSite(object):
    site = Section("", name='webthings').configure(''
      , promote_children=True
      )

    js = Section("", name='js').configure(''
        , alias   = 'js'
        , display = False
        , module  = None
        , kls     = None
        )

# Create webthings.settings
settings.THESITE = theSite
inject(settings, 'webthings.settings')

########################
###   ACTIVE
########################

# Create the website object to specify each part of the website
website = Website('webthings_main'
      , P('index', first=True)
      , P('news')
      , P('events')
      , P('photos')
      )

# And configure the website
website.configure()

# Just make sure we can get the urls
# Note we also have webthings_main.models available now
# Because website.configure gives us webthings_main.urls and webthings_main.models
from webthings_main.urls import site
```

```
########################
###   URLS
########################

# Create urls
# Depends on webthings being setup
# Note that settings.ROOT_URLCONF is set to 'urls'
from webthings.config.common import urls
inject(urls, 'urls')
```

webthings.config.common.settings

```
from cwf.splitter.imports import steal, inject
import os

########################
###   INJECTION
########################

this_dir = os.path.dirname(__file__)
settings_dir = os.path.abspath(os.path.join(this_dir, '..', 'settings'))
steal('inclusions', 'logging', 'testing', 'other'
    , folder=settings_dir, globals=globals(), locals=locals()
    )

if DEBUG:
    LOGGING['loggers']['']['level'] = 'DEBUG'

########################
###   INSTALLED
########################

INSTALLED_APPS += (
      'webthings_main'
    , 'cwf'

    , 'grappelli'
    , 'django.contrib.admin'
    , 'django.contrib.sessions'

    , 'south'
    )

########################
###   TEMPLATES
########################

def __get_template_dirs__():
    """In a function so I don't pollute the settings namespace"""
    import pkg_resources
    webthings_main_folder = pkg_resources.resource_filename("webthings_main", "")
    error_page_templates = pkg_resources.resource_filename("webthings_main", 'templates/errors')
    return (webthings_main_folder, error_page_templates, )

TEMPLATE_DIRS = __get_template_dirs__()

# Use the AppNameLoader as suggested in the templates section of the docs
# To make it so cwf templates are available
```

```
TEMPLATE_LOADERS = (
    'webthings_main.utils.template_loaders.AppNameLoader'
  ,
  )

ROOT_URLCONF = 'urls'
```

webthings.config.common.urls

```
from django.conf.urls.defaults import patterns, include
from django.contrib import admin
from django.conf import settings

########################
###   GRAPPELLI
########################

haveAdmin = 'django.contrib.admin' in settings.INSTALLED_APPS
if haveAdmin:
    admin.autodiscover()

########################
###   CWF GENERATED URLS
########################

theSite = settings.THESITE
site = theSite.site

from webthings_main.urls import site as main_site
site.merge(main_site, take_base=True)

if hasattr(theSite, 'js'):
    js = theSite.js
    site.adopt(js, consider_for_menu=False, include_as = 'js')

urlpatterns = site.patterns()

########################
###   ADMIN URLS
########################

if haveAdmin:
    urlpatterns += patterns('', (r'^grappelli/',  include('grappelli.urls')))
    urlpatterns += patterns('', (r'^admin/',      include(admin.site.urls)))
```

webthings.wsgi.prod

```
import webthings
import sys
import os

def serve_site(production=True, internet=True):
    """
        Ensure we have DJANGO_SETTINGS_MODULE environment variable
        And return wsgi application for django
    """
    os.umask(2)
```

```
    # Setup project and set django_settings_module
    os.environ['DJANGO_SETTINGS_MODULE'] = 'webthings.settings'
    webthings.project_setup()

    # Create and return wsgi app for django
    import django.core.handlers.wsgi
    return django.core.handlers.wsgi.WSGIHandler()

application = serve_site()
```

You can then have a uwsgi configuration that looks like:

```
[uwsgi]
chdir=/path/to/webthings/wsgi
module=prod:application
vacuum=True
home=/path/to/venv_for_webthings
uid=www-data
gid=www-data
disable-logging=true
touch-reload=/path/to/webthings/wsgi/prod.py
```

And an nginx configuration that looks like:

```
server {
    server_name webthings;

    upstream webthings {
        ip_hash;
        server unix:///run/uwsgi/app/webthings/socket;
    }

    location / {
        include     uwsgi_params;
        uwsgi_pass  webthings;
    }
}
```

Binaries

CWF comes with two command line applications that you can use to interact with your Django project:

cwf-manager manage.py without creating the manage.py

cwf-debugger Start your django project using werkzeug and it's awesome debugger.

9.1 cwf-manager

The prescribed setup for a django project suggests creating a manage.py file that is used to access all of django's nice manager functionality from the cli.

`cwf-manager` implements the code that should go into manage.py such that as long as you're in a folder with the same name as your project and you have your project on your python path, then it will be able to access the django admin functionality.

For example, say your project is called `webthing` and the settings.py can be accessed via `webthing.settings` and your in a folder with the name `webthing`, then:

```
$ cwf-manager
    Usage: cwf-manager subcommand [options] [args]

    Options:
      -v VERBOSITY, --verbosity=VERBOSITY
                            Verbosity level; 0=minimal output, 1=normal output,
                            2=verbose output, 3=very verbose output
      --settings=SETTINGS   The Python path to a settings module, e.g.
                            "myproject.settings.main". If this isn't provided, the
                            DJANGO_SETTINGS_MODULE environment variable will be
                            used.
      --pythonpath=PYTHONPATH
                            A directory to add to the Python path, e.g.
                            "/home/djangoprojects/myproject".
      --traceback           Print traceback on exception
      --version             show program's version number and exit
      -h, --help            show this help message and exit

    Type 'cwf-manager help <subcommand>' for help on a specific subcommand.

    Available subcommands:

    [auth]
        changepassword
```

```
        createsuperuser

    [django]
        cleanup
        compilemessages
        createcachetable
        dbshell
        diffsettings
        dumpdata
        flush
        inspectdb
        loaddata
        makemessages
        reset
        runfcgi

[etc]
```

9.1.1 Project Setup

As an added bonus, you can do any project setup you wish before the manager starts up by implementing `webthing.project_setup` as a callable that only takes keyword arguments.

This `project_setup` function is called after the `DJANGO_SETTINGS_MODULE` environment variable is set, and before the django `execute_from_command_line` (the heart of manage.py) is called.

The return of `project_setup` is ignored.

9.2 cwf-debugger

Whilst "*cwf-manager* runserver" provides a fine local server you can develop with; it is nice to have werkzeug's awesome debugger and `cwf-debugger` makes that easy.

The first argument is the import path to your project. So as is for the *bin-cwf-manager*, if your settings.py can be found under `webthings.settings` and you want to run a debug server of `webthings`, you would use:

```
$ cwf-debugger webthings
```

The debuggger has the same *project_setup* semantics as *cwf-manager* and also provides a `-o` flag which you may use to pass in a json formatted string that is used as keyword arguments to `project_setup`.

Note: Unfortunately, the current implementation of cwf-debugger does require a small change to werkzeug : https://github.com/mitsuhiko/werkzeug/issues/220

CHAPTER 10

Admin

10.1 Buttons

You can create buttons using `cwf.admin.Button` and `cwf.admin.ButtonGroup` classes where a `ButtonGroup` is merely a container of multiple `Button`.

A `ButtonGroup` takes in a `name` and an iterable of `buttons`. A `Button` takes in a `url` fragment and the `desc` of the button (the name that should be displayed for that button when rendered as html).

Both `Button` and `ButtonGroup` also take in keyword arguments as listed *below*

10.1.1 Button Options

`Button` and `ButtonGroup` have the following options.

kls Css class to give to the html representation of the button.

Defaults to None.

display A boolean used to make a button not show

for_all True if you want the button to appear in the changelist.

False if you want the button to appear in the changeform.

Defaults to False.

condition A boolean or callable (accepting the instance of the button and instance of the model being edited).

This is used to say whether there is some condition against the button being shown.

new_window Whether clicking the button opens the new request in a new window.

Defaults to False.

needs_auth If a boolean, says whether `request.user.is_authenticated()` needs to be True or not.

If a string, or a list of strings, then the user must have all the permissions as specified by each string.

description A long description for the button. Currently only used for buttons in a `ButtonGroup`

save_on_click Whether to save the form before redirecting to the view for that button.

Note: This is set to False if you have `for_all` set to True.

need_super_user Whether you need to be super user for this button to be visible

return_to_form Whether to redirect straight back to the form after the button executes it's view.

This works for both buttons that appear on change list and those that appear on the change form.

Note: When buttons are given to the template context so that they can be rendered by the *template*, they are first wrapped in a `cwf.admin.buttons.ButtonWrap` which is a container that holds the button , the request context, and the the object being edited.

This container proxies to the button and also provides some functions that interpret the options on the button against the request for the convenience of the template.

10.1.2 Button Html

When rendering a button, you have `button.html` available, which will return the html that can be used to represent the button.

If the button has the `save_on_click` option, then it is rendered as an "<input>" box that will submit the form before going into it's own view , thus effectively saving the modifications you've made to the form before doing anything else.

Otherwise, the button is rendered as an "<a>".

If you use the *templates* provided, then it will also make sure that the buttons that save the form are put at the bottom of the page near the default "save" buttons that the admin provides.

Whereas buttons that don't save the form will appear at the top next to where the admin already provides a button to see the history for an object.

Due to the css that the admin provides, these buttons will also have distinct looks that signify this difference.

CWF provides the ability to add buttons to your admin pages with relative ease.

As long as you have made the CWF templates *available* and your admin class subclasses `cwf.admin.ButtonAdmin`, then you can give your admin a list of buttons that are rendered on your admin page and call particular functions on your admin class when they are pressed.

10.2 How it works

There are two things that are necessary to make extra buttons work on the admin:

Modified urls for that admin Django admin classes have a `urls` function that is used to generate the urlpatterns for that particular admin.

The `cwf.admin.ButtonAdmin` class overrides this function to add urls for each button that will figure out what function on the admin class to use when that url is requested.

Display buttons to press `cwf.admin.ButtonAdmin` will add the buttons you put onto the class into the context when rendering the template for the changeform and the changelist views.

Then, as long as the templates you are using are aware of these buttons, then they may be displayed.

Usage is something like:

```
from django.http import HttpResponse
from django.contrib import admin

from cwf.admin import ButtonAdmin, Button, ButtonGroup
```

```
from webthing.models import AwesomeModel

class AwesomeAdmin(ButtonAdmin):
    [..]

    buttons = (
          Button("one", "First")
        , Button("two", "Second")
        , ButtonGroup("Other",
            ( Button('three', "Third"
                , description="The third button"
                )
            , Button('four', "Fourth"
                , description="The fourth button"
                )
            )
        )

    def tool_one(self, request, ball, button):
        return HttpResponse("one")

    def tool_two(self, request, ball, button):
        return HttpResponse("two")

    def tool_three(self, request, ball, button):
        return HttpResponse("three")

    def tool_four(self, request, ball, button):
        return HttpResponse("four")

    [..]

admin.site.register(AwesomeModel, AwesomeAdmin)
```

See *Buttons* for what options are available.

10.3 Button View

When the admin urls are created, they create views that calls a function on the admin that is found using the `url` of the button.

So if the `url` of the button is "one", then the view that is used for that button will call `tool_one` on the admin class, passing in the request, the object being edited and the instance of the button that was pressed.

This view may return a `HttpResponse` object directly , or it may return a tuple of (File, extra) , where `File` is the path to the template you want to render and `extra` is any extra context that you want to provide to the template.

Note: The template will be rendered with a RequestContext

You may also use the `return_to_form` *option* to make the view automatically redirect to the form where the button was pressed regardless of what is returned from the `tool_` method.

CHAPTER 11

Template Tags

CWF contains two custom templatetags that it uses when it's constructing the template for the *menu generation* functionality provided.

These are:

varset Used to create a variable in the template from a block

wrapped Used to create a html element only if the body of the block provided isn't empty

11.1 varset

Usage for this tag is:

```
{% load varset %}
{% varset name_of_new_variable %}
    {% if some_condition %}
        {{some_value}} blah
    {% else %}
        another value
    {% endif %}
{% endvarset %}
```

After we `load` the varset templatetags, we have available to us a tag called `varset`. It takes the name of the variable to create and will put the contents between the start of that tag till `endvarset` and make that the value of the new variable, which can be used in your template any point after this.

So in the example above, your template will now have a variable available called `name_of_new_variable` with whatever that block of code evaluates into.

11.2 wrapped

Usage for this tag is:

```
{% load wrapped %}
{% wrapped li 'class="awesome" style="background-color:red"' %}
    {% if condition %}
        <p>wicked awesome</p>
    {% endif %}
{% endwrapped %}
```

This allows us to create an html element that will only appear if it has something inside of it.

So in the example above it will create:

```
<li class="awesome" style="background-color:red">
    <p>wicked awesome</p>
</li>
```

If `condition` is truthy. Otherwise, there isn't anything between the `wrapped` and `endwrapped` tags except for whitespace and so it won't even output the `<li>`.

CHAPTER 12

Templates

12.1 Admin Templates

CWF supplies some templates to compliment the *admin functionality*.

These are:

admin/blank.html Renders a blank admin page with the breadcrumbs filled out

admin/change_form.html Renders a model changeform with any buttons you've defined

admin/change_list.html Renders a model changelist with any buttons you've defined

As long as you make the templates *available* then the `changelist` and `changeform` templates will be used automatically by the admin and the `blank` template is available for you to use in any custom admin pages you create.

Note: These templates do assume you're using grappelli

12.1.1 admin/blank.html

This template is useful to extend from when you're returning a template for a *Button View*. It provides an empty page with the breadcrumbs already filled out with Home, the model, the instance of the model and the description of the button as the last part.

It works by extending `admin/base_site.html` and using the following:

`app_label` Name of the current app

`module_name` The name of the model in the app that is being edited

`object` The object being edited

`bread_title` The name of the current page

As part of the `base_site.html` template it inherits from, you have the `content` block available to add information to the body of the page.

Note: If there is no object then it assumes you're url has one less part to it and will change the breadcrumb to only display Home, app, model, button.

12.1.2 admin/change_form.html

This is the form that is used by django admin to display the form when you are adding or editing an instance of a model.

The CWF version of this template will display the buttons that have the `for_all` *option* set to False.

Those buttons that don't save the template when they are clicked are displayed at the top of the page and those that do save the form when they are clicked are displayed at the bottom of the page. See *Button Html* for more information.

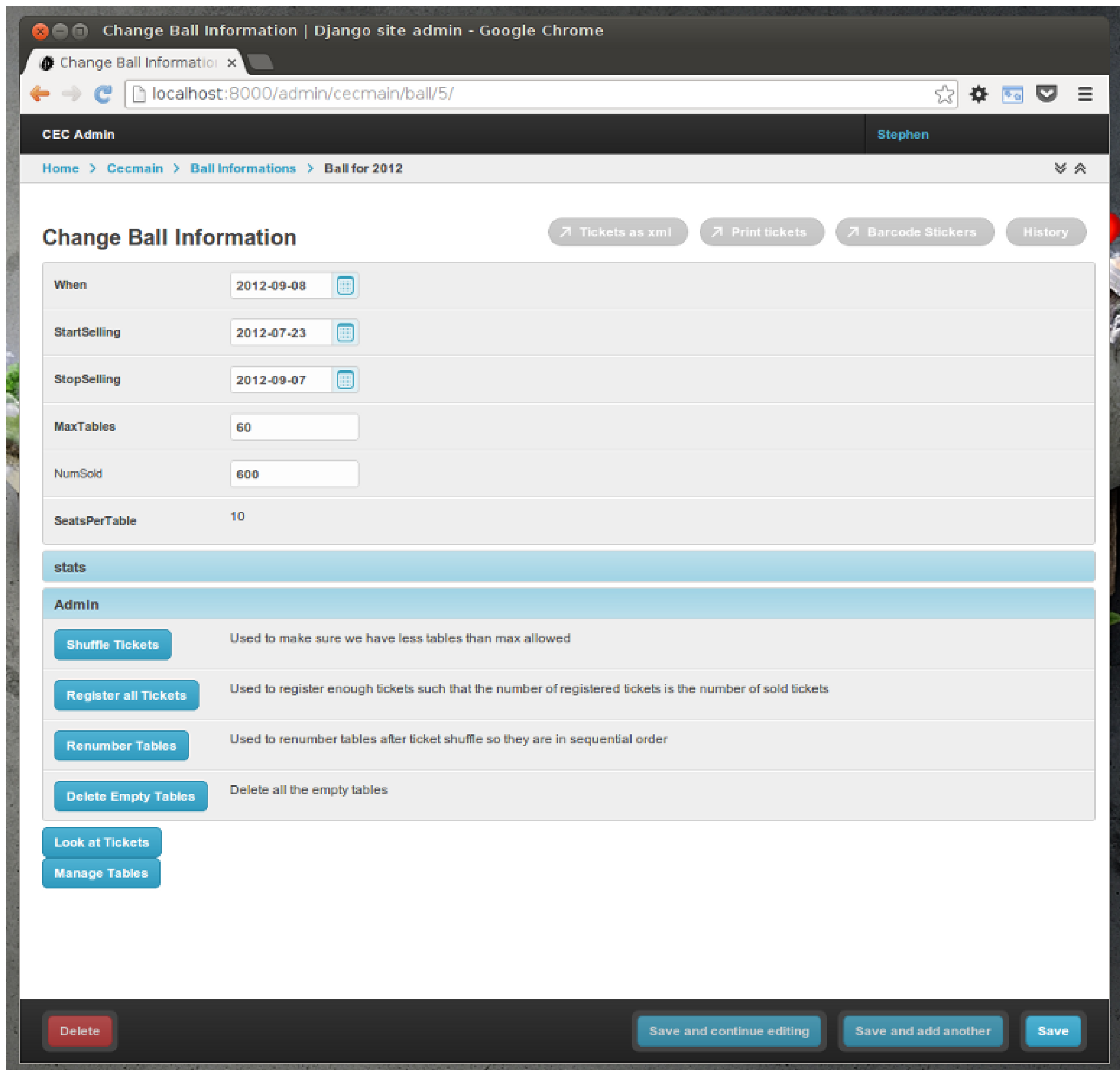
`ButtonGroups` are displayed in a collapsible panel of rows where each row has two columns: The button itself and it's description.

All buttons are displayed using the *button html*.

For example, say we have the following button specification in our admin:

```
class BallAdmin(ButtonAdmin):
    [..]

    buttons = (
          Button("printxml", "Tickets as xml", need_super_user=False, new_window=True, save_on_click=
        , Button("printable", "Print tickets", need_super_user=False, new_window=True, save_on_click=
        , Button("barcodes", "Barcode Stickers", need_super_user=False, new_window=True, save_on_clic

        , ButtonGroup("Admin",
            ( Button('shuffletickets', "Shuffle Tickets", need_super_user=False, needs_auth='cecmain
                , description="Used to make sure we have less tables than max allowed"
                )
            , Button('registerall', "Register all Tickets", need_super_user=False, needs_auth='cecma
                , description="Used to register enough tickets such that the number of registered ti
                )
            , Button('renumber', "Renumber Tables", need_super_user=False, needs_auth='cecmain.admin_
                , description="Used to renumber tables after ticket shuffle so they are in sequentia
                )
            , Button('deleteempty', "Delete Empty Tables", need_super_user=False, needs_auth='cecmai
                , description="Delete all the empty tables"
                )
            )
            , need_super_user=False, needs_auth='cecmain.admin_ball'
        )

        , Button('configure', "Configure Test Ball", new_window=True, need_super_user=False, needs_au
            # Only show configure for test ball
            , condition = lambda button, ball : ball and ball.is_test_ball()
            )
        , Button('tickets', "Look at Tickets", need_super_user=False)
        , Button('manageTables', "Manage Tables", need_super_user=False)
        )

    [..]
```

We'll get a changeform that looks something like:



12.1.3 admin/change_list.html

This is the form that is used by django admin to display the list of instances of a particular model.

The CWF version of this template will display the buttons that have the `for_all` *option* set to True.

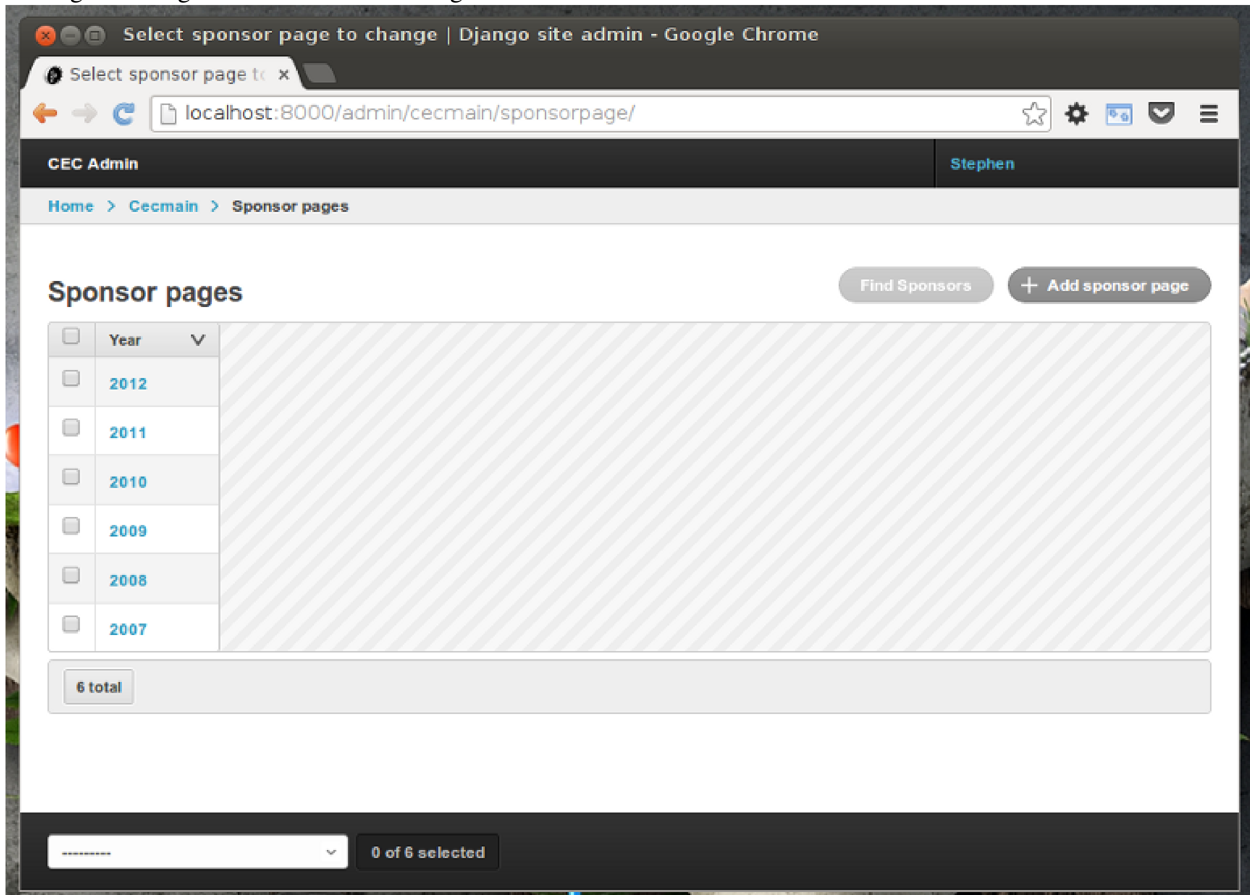
They are displayed at the top of the page and doesn't support button groups.

So say we had the following code in our admin:

```
class SponsorAdmin(ButtonAdmin):
    [..]

    buttons = (
          Button('findsponsors', "Find Sponsors", for_all=True)
        )
```

```
[..]
```

We'll get a changelist that looks something like:



12.2 Menu Templates

There is only one template supplied for rendering the menus, which is found inside the "templates/menu" folder under cwf root. It's name is `base.html` and it can be used to recursively generate the menu system.

Note: As long as it is *available* and it has the *menu object*.

It uses the following variables:

`children_template` What template it should use to render the children. This can be set to the same template you are starting the generation from.

i.e.

```
{% include "menu/base.html" with menu=menu.global_nav children_template="menu/base.html" igr
```

`menu` The menu object we're getting the sections from.

`ignore_children` Boolean that says if we should ignore displaying children at all regardless of whether a section actually has children.

This is useful for displaying a global navigation where you only want the root level navs with none of their children.

And supplies the following blocks that you can override:

selected_item_attrs The attributes given to `<li>` sections that represent a selected section.

unselected_item_attrs The attributes given to `<li>` sections that do not represent a selected section

item_link Determine what to display as the link inside each `<li>`.

item_children Determine what to display as children for a section.

It will make sure that each list of children is wrapped in an `<ul>` and that sections that don't have children will not output an empty `<ul></ul>`.

CWF comes with some templates you can use alongside some of the other features that is provided.

12.3 Available Templates

Templates are provided for the following:

Admin Templates Templates for adding buttons to the admin pages.

Menu Templates Templates for displaying *Generated Menus*.

12.4 Including Templates

To be able to use these templates in your code, you need to make them available.

You either do this by adding the directory to CWF templates in your django settings `TEMPLATE_DIRS` option or by adding a loader to the `TEMPLATE_LOADERS` option that understands where CWF is.

The first method could look something like this:

```
def __cwf_dirs__():
    """In a function so I don't pollute the settings namespace"""
    import pkg_resources
    cwf_templates = pkg_resources.resource_filename("cwf", "templates")
    return (cwf_templates, )

TEMPLATE_DIRS = TEMPLATE_DIRS + __cwf_dirs__()
```

Alternatively, you could make a loader like this:

```
from django.template.loaders.app_directories import Loader
from django.utils.importlib import import_module
from django.utils._os import safe_join
from django.conf import settings

import sys
import os

class AppNameLoader(Loader):
    """Loader that will allow the app name in the template location"""
    @property
    def app_template_dirs(self):
```

```
        """Memoize the app template dirs"""
        if not hasattr(self, '_app_template_dirs'):
            self._app_template_dirs = tuple(self.get_app_template_dirs())
        return self._app_template_dirs

    def get_app_template_dirs(self):
        """Yield tuples of (app, template_dir) for each installed app"""
        fs_encoding = sys.getfilesystemencoding() or sys.getdefaultencoding()
        for app in settings.INSTALLED_APPS:
            try:
                mod = import_module(app)
            except ImportError, e:
                raise ImproperlyConfigured('ImportError %s: %s' % (app, e.args[0]))

            template_dir = os.path.join(os.path.dirname(mod.__file__), 'templates')
            if os.path.isdir(template_dir):
                yield app, template_dir.decode(fs_encoding)

    def get_template_sources(self, template_name, template_dirs=None):
        """Get template path relative to template dir of specified app"""
        base = template_name.split('/')[0]
        relpath = os.path.relpath(template_name, base)

        for app, template_dir in self.app_template_dirs:
            if app == base:
                try:
                    yield safe_join(template_dir, relpath)
                except UnicodeDecodeError:
                    # The template dir name was a bytestring that wasn't valid UTF-8.
                    raise
                except ValueError:
                    # The joined path was located outside of template_dir.
                    pass
```

And add it to your template loaders:

```
TEMPLATE_LOADERS = TEMPLATE_LOADERS + 'path.to.AppNameLoader'
```

Which would mean that if 'cwf/menu/base.html' doesn't match for any of the other template loaders you have, then it will take the first part of that url (in this case, 'cwf') and find the template dir for that app and look in there.

CHAPTER 13

CWF

A layer of random utility that sits between Django and your site.

Mainly used so that you can specify what views go to what urls in such a way that we can render a menu system from the same specification.

It also includes utilities to be able to seperate your site into multiple folders within the same project, each including their own views, urls, models and admin code.

Along with some other random helpers to make life with Django even simpler than it already is.

Documentation can be found on readthedocs: https://cwf.readthedocs.org

Install from pypi:

```
pip install cwf
```

13.1 Changelog

1.1.4 - 23rd March 2014

- Fix docs to work with new version of cloud_sptheme
- Added catch_all option for url patterns

1.1.3 - 22nd February 2014

- HttpResponse takes content_type instead of mimetype in new Django

1.1.2 - 22nd February 2014

- Support Django 1.6
- Removed some of the magic from the tests
- Removed the include_defaults option for splitter.Parts.urls

1.1.1 - 12th January 2014

- Make cwf-debugger tell you about https://github.com/mitsuhiko/werkzeug/issues/220

1.1 - 26th October 2013

- Updated noseOfYeti
- cwf-debugger now doesn't fail on projects without a project_setup
- Update to work with newer Django

- Minor typo

1.0 - 13th January 2013

- First public release after many years and a complete rewrite

13.2 Getting Started

Installation How to install CWF and what that means

Tests How to run the CWF tests

13.3 Features

The rest of the documentation is separated into the broad categories of functionality that cwf provides:

Sections Code related to constructing urlpatterns and menus

Views Helpers for creating view callables

Splitter Code related to breaking up your django app into seperate folders

Binaries Useful applications to use from the commandline to interact with your app

Admin Additions to the django admin

Template Tags Some templatetags used by the menu templates

Templates Templates used for admin and menu functionality

Python Module Index

C

cwf.views.views, 24