canary Documentation

Release 0.1

Branton K. Davis

August 18, 2012

CONTENTS

Contents:

CHAPTER

ONE

INSTALLATION

1.1 Installing Canary Reports

It's very easy to install Canary Reports:

1. Download the source (available on bitbucket) and put the canary directory somewhere in your Python path:

   ```
   $ git clone git@bitbucket.org:bluecamel/django-canary.git django-canary
   ```

2. Add `canary` to `INSTALLED_APPS`.
3. Run `collectstatic` or copy the `static/canary` directory to your static media directory.

1.2 Running the Demo Project

All of the examples in the documentation are included in the `demo` project. If you'd like to play with the examples yourself, simply:

1. Make sure `canary` and `canary.demo` are in your `PYTHONPATH` or `sys.path`.
2. Install `inplaceeditform`. This is a dependency for editable columns:
   - Make sure `inplaceeditform` is in your `PYTHONPATH` or `sys.path`.
   - Make sure you've run `collectstatic` or otherwise managed your static files.
3. Change the database configuration in `demo/settings.py` to suit your preferences. If you have *sqlite* installed, you don't need to do anything.
4. Create the database tables needed for the app:

   ```
   $ python manage.py syncdb
   ```

5. Load initial data to use to play with the reports:

   ```
   $ python manage.py loaddata canary_test_data
   ```

6. Start your development server:

   ```
   $ python manage.py runserver
   ```

7. Go to http://localhost/canary_examples/ to play with the reports.

1.3 Installing the Examples App

You can also just install the `examples` app in your own project:

1. Add `canary.demo.examples` to `INSTALLED_APPS`.
2. Create the database tables needed for the app:

   ```
   $ python manage.py syncdb
   ```

3. Add this line to your urls.py:

   ```
   url(r'canary_examples/', include('canary.demo.examples.urls')),
   ```

4. Start your development server:

   ```
   $ python mnage.py runserver
   ```

5. Go to http://localhost/canary_examples/ to play with the reports.

1.4 What now?

Read through the *Introduction to Django Canary Reporting*.

CHAPTER

TWO

INTRODUCTION TO DJANGO CANARY REPORTING

This project is intended to allow you to build simple reports using models or QuerySets. You get quite a bit of functionality for free (choice filters, foreign key filters, multi-column sorting and pagination). However, you can do much more by extending the built-in columns, filters, sorts, or even views, which control report generation. The default styling is very simple, and intended to blend in with the default Django admin, but the templates are also very simple and therefore easy to extend or replace.

2.1 Getting started (Views)

Assume that we have the following Django models:

```
from django.db import models


class Company(models.Model):
    # constants
    INDUSTRY_CHOICE_TECHNOLOGY = 1
    INDUSTRY_CHOICE_MONKEYS = 2
    INDUSTRY_CHOICE_TECHNOLOGY_MONKEYS = 3
    INDUSTRY_CHOICE_MONKEY_TECHNOLOGY = 4
    INDUSTRY_CHOICES = (
        (INDUSTRY_CHOICE_TECHNOLOGY, 'Technology'),
        (INDUSTRY_CHOICE_MONKEYS, 'Monkeys'),
        (INDUSTRY_CHOICE_TECHNOLOGY_MONKEYS, 'Technology Monkeys'),
        (INDUSTRY_CHOICE_MONKEY_TECHNOLOGY, 'Monkey Technology')
    )

    # columns
    name = models.CharField(max_length=100)
    industry = models.IntegerField(max_length=1, choices=INDUSTRY_CHOICES)

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

    class Meta:
        verbose_name_plural = u'Companies'


class Contact(models.Model):
    first_name = models.CharField(max_length=100)
    middle_name = models.CharField(max_length=100, blank=True, null=True)
    last_name = models.CharField(max_length=100)

    active = models.BooleanField(default=True)

    company = models.ForeignKey(Company)

    def __unicode__(self):
        return unicode(self.get_full_name())

    def get_full_name(self):
        """
            Return a string containing the contact's full name.
        """
        name_fields = []
        for name_field in (self.first_name, self.middle_name, self.last_name):
            if name_field:
                name_fields.append(name_field)
        return ' '.join(name_fields)

    @property
    def has_first_name(self):
        """
            Return a boolean, indicating whether the contact has a middle name.
        """
        return bool(self.middle_name)
```

To create a simple report using these models, just create a report view. Put this in your views.py (or maybe admin_views.py or reports.py). We'll start with the most basic option, which is a ModelReport:

```
from canary.views import ModelReport

from models import Contact


class ContactsAll(ModelReport):
    class Meta:
        model = Contact

contacts_all = ContactsAll.as_view()
```

There's nothing terribly special about the view handling. All of the report views in the package are extensions of Django's `TemplateView`, so you can configure your URLs in whatever way you like. Here, we'll setup our urls.py like so:

```
from django.conf.urls.defaults import patterns, url
from django.contrib.admin.views.decorators import staff_member_required

import views


urlpatterns = patterns('',
    url(r'^contacts/all/$', staff_member_required(
        views.contacts_all), name='contacts_all'),
)
```

This will create a report containing all columns in the Contact model, each sortable and searchable. Not bad for just a few lines of code.

Now, let's say that we want to have more control over the queryset used to generate the report. Maybe this report should only include active contacts. Well, this is easy to do by giving the queryset to the view. Our new view might look like this:

```
class ContactsActive(ModelReport):
    class Meta:
        model = Contact
        queryset = Contact.objects.filter(active=True)
```

What if we didn't want the report to display all of the columns from the model? Well, that's easy too, and very similar to Django's ModelAdmin or ModelForm. You can either specify the columns that you want to include in the report:

```
class ContactsActiveInclude(ModelReport):
    class Meta:
        model = Contact
        queryset = Contact.objects.filter(active=True)
        include = ('first_name', 'middle_name', 'last_name', 'company', 'created', 'updated')
```

Or you can specify the columns that you want to exclude:

```
class ContactsActiveExclude(ModelReport):
    class Meta:
        model = Contact
        queryset = Contact.objects.filter(active=True)
        exclude = ('active',)
```

Both of the previous examples have the same result of including all columns but 'active'.

2.2 A little more advanced (Columns)

Now, let's say that we wanted to add the industry column from the Company field. The quickest way to do this is by upgrading from a ModelReport to a QueryReportView. This is a bit similar to moving from a Django ModelForm to a Form, only with a little more magic. You actually define each column that you want to include in the report, but the columns can get their data from any attribute, method, or even a chain of related models, all coming from a QuerySet that you provide. We'll dig into more advanced configurations later, but for now, let's see how we would add info from a related model:

```
class ContactsActiveColumns(QueryReport):
    first_name = columns.Text()
    middle_name = columns.Text()
    last_name = columns.Text()

    industry = columns.Text('company__get_industry_display')

    class Meta:
    queryset = Contact.objects.filter(active=True)
```

The first three columns are pretty basic. The Text column is the most basic column. It grabs its data from the correponding column from an object in the `QuerySet`. If the column name is different or you want to reference a related column, you can include an argument, as with the industry column in the example above. The format for referencing related columns should be familiar to Django folks. The string is very similar to a python dot path, where '__' replaces the dot. So, `industry = columns.Text('company__get_industry_display')` tells the column to get it's data from `obj.company.get_industry_display`, where `obj` is an object in the `QuerySet`.

What if you want to include the results of a view method in your report. That's easy too. Just define a new method on the view and include the "path" to the attribute/method in your column definition:

```
class ContactsActiveViewMethod(QueryReport):
    name = columns.Text('get_full_name')
    industry = columns.Text('get_industry')

    def get_industry(self, contact):
        """
        The first argument is passed the current object from the QuerySet.  you can return anything y
        """
        return contact.company.get_industry_display()

        class Meta:
        queryset = Contact.objects.filter(active=True)
```

Read more about *Attribute Resolution*.

So now that you can control the columns that end up in your report, let's look at how you can control the display of individual columns. There are a few built-in columns that you can use to change the display. For example, you might want a column's data to display as a link on each row. Let's say that we'd like to link to the admin view of the contact object:

```
class ActiveContactReport(QueryReportView):
    name = columns.AdminLink('get_full_name', reverse_path='contacts_contact_change', reverse_args=('
    industry = columns.Text('company__industry')

    class Meta:
        queryset = Contact.objects.filter(active=True)
```

Whoah!! There's a bit of magic going on here. If you're used to reversing Django's admin URLs, you probably already know what's going on. If not, you might want to take a look at Django's admin reversing docs. The second and third attributes to the AdminLink class are the reverse path and args for the Django `reverse` command. So, the `name` definition above results in calling `reverse('admin:contacts_contact_change', args=(obj.id,))` for each object in the QuerySet, and thus providing the link to the change view of that object.

Pretty cool, yeah? What if you just want to provide a link using `reverse`? Well, check out the `ReverseLink` class. Actually, `AdminLink` is just a simple extension of `ReverseLink`. None of this suits you? Fine, make your own! Extend the `Link` class or even the base `Text` or `Column` classes to do whatever you want. In fact, the goal is to provide a lot of basic types of column widgets that most people can just use, but allow you to easily extend any of them for your own use.

Learn more about *Columns*.

2.3 Filters

Filters are a core concept to the project and where a lot of the power and flexibility comes from. Every column can accept a list of filters that can be applied to the QuerySet for that column. Some filters are so common that they are included by the `Column` class. For example, a date range filter is so common that there is a `Date` column that includes the `DateRange` filter by default. Of course, this is easy to override.

Let's add date columns to our report to see how the `DateRange` filter is implemented by default:

```
class ActiveContactReport(QueryReportView):
    first_name = columns.Text()
    middle_name = columns.Text()
    last_name = columns.Text()

    created = columns.Date()
    updated = columns.Date()
    industry = columns.Text('company__get_industry_display')

    class Meta:
        queryset = Contact.objects.filter(active=True)
```

Both the `created` and `updated` columns get a DateRange filter by default. You may have also noticed by now that the `Text` column includes the `Search` filter by default, as long as the column can be queried via the Django ORM.

Now, what if the attribute that displays the data is not the same as the column that filters the data? Well, that's not a big deal. You just manually define the filter, and provide it with the column(s) to use for filtering:

```
from canary import filters

class ActiveContactReport(QueryReportView):
    full_name = columns.Text('get_full_name', filters=[
        filters.Search(columns=('first_name, middle_name, last_name')])

    created = columns.Date()
    updated = columns.Date()

    industry = columns.Text('company__industry')

    class Meta:
        queryset = Contact.objects.filter(active=True)
```

Note: The Search filter may not have worked on the earlier examples for this very reason. For example, the ContactsActiveViewMethod in the demo project is updated from the example to specify the correct columns to search.

Filters are where you can add a lot of custom functionality to your reports. Learn more about *Filters*

CHAPTER

THREE

VIEWS

More information about views coming soon...

3.1 Class Reference

- *canary.views.BasicReport*
- *canary.views.QueryReport*
- *canary.views.ModelReport*

class `canary.views.`**`BasicReport`**

Very basic base class that inlcudes the metaclass and sets the default template_name.

`add_extra_media`(*extra_css*, *extra_js*, *js_context*)

Add additional media to the view.

Args:

extra_css: A dictionary with CSS info, such as:

{ 'css/extra_styles.css': {'media': 'screen, projection'}, 'css/more_styles.css': None

}

extra_js: An iterable of iterables with JS info, such as:

{ 'js/extra_scripty_magics.js': None, 'js/more_scripters.js': None

}

js_context: A dictionary of JSON strings, such as:

{ 'editableURL': '/canary/edit_column/',

}

The extra data you want to include is then available to your included JS scripts, from canary.editableURL.

Sets: extra_css: Adds to existing extra_css. extra_js: Adds to existing extra_js. js_context: Adds to existing js_context.

`get_extra_css`()

Return the dictionary with default parameters set.

Returns:

A dictionary, such as:

> > **{** ‘css/extra_styles.css’: {‘media’: ‘screen, projection’, ‘type’: ‘text/css’, ‘rel’: ‘stylesheet’}, ‘css/more_styles.css’: {‘type’: ‘text/css’, ‘rel’: ‘stylesheet’},
> >
> > }

`get_extra_js``()`
Return the dictionary with default parameters set.

Returns:

A dictionary, such as:

> **{** ‘js/extra_scripty_magics.js’: {‘type’: ‘text/javascript’}, ‘js/more_scripters.js’: {‘type’: ‘text/javascript’}
>
> }

class `canary.views.`**`QueryReport`**`(`**args*, ***kwargs*`)`
Build a report using a provided queryset, along with column definitions.

`filter_queryset``(`*request*`)`
Your second chance to affect the queryset, with filters.

You can override this method to change the way filters are handled, or even just to add a final filter onto your query.

`get``(`*request*, **args*, ***kwargs*`)`
Handle a GET request.

`get_app_label``()`
Return the app label from the queryset.

`get_model_name``()`
Return the model name from the queryset.

`get_post``(`*request*, **args*, ***kwargs*`)`
Handle any request.

`get_row``(`*obj*`)`
Given the current object from the QuerySet, return a list for the row.

`get_rows``()`
Get a list of row lists from the QuerySet.

`iter_rows``()`
Iterate over the QuerySet, returning a row list on each iteration.

`limit_queryset``(`*request*`)`
Limit/paginate the QuerySet.

Args: request: The Django request object.

Sets: _meta.queryset: The Django QuerySet.

`load_aggregates``(`*request*`)`
Load all aggregates from all columns.

Args: request: The Django request object.

Sets: aggregates: A dictionary of aggregates.

`load_filters``(`*request*`)`
Load all filters from all columns.

Args: request: The Django request object.

Sets: filters: A dictionary of filters. order_by: A list of columns, in order of sort preference.

`load_headers`(*request*)
Load all headers. Headers include filters and sorts, so those must already be loaded.

Args: request: The Django request object.

Sets: headers: A list of ColumnHeader objects.

`load_queryset`(*request*)
Your first chance to affect the queryset.

Args: request: The Django request object.

Sets: queryset: A Django QuerySet.

Raises: AttributeError: QueryReport.Meta.queryset isn't defined.

`load_sort_manager`(*request*)
Create the SortManager.

Args: request: The Django request.

Sets: _meta.sort_manager: SortManager instance.

`post`(*request*, **args*, ***kwargs*)
Handle a POST request.

`resolve_text`(*obj*, *attr*, *default=None*)
Resolve the given attribute from the given object or the current view.

Args: obj: The object from the QuerySet. attr: The attribute to resolve. default: The value to return if the attribute can't be found.

Returns: The resolved value or default.

`sort_queryset`(*request*)
Your last chance to affect the queryset.

Expected to apply an order_by to the QuerySet.

TODO: implement multi-column sort.

Args: request: The Django request object.

Sets: _meta.queryset: A Django QuerySet.

class `canary.views.`**`ModelReport`**(**args*, ***kwargs*)
An extension of QueryReport that builds a report from a given model.

`load_columns`()
Load columns from the model and create canary columns for each.

Sets: _meta.columns: An OrderedDict of column.

`load_queryset`(*request*)
Load the QuerySet. First try to load from a given QuerySet (defined as QuerySet.Meta.queryset). If none was provided, load a QuerySet of all objects for the model.

Args: request: The Django request object.

Sets: _meta.queryset: A Django QuerySet.

CHAPTER

FOUR

COLUMNS

4.1 Class Reference

- *canary.columns.Column*
- *canary.columns.Text*
- *canary.columns.Tag*
- *canary.columns.Link*
- *canary.columns.ReverseLink*
- *canary.columns.AdminLink*
- *canary.columns.BooleanValue*
- *canary.columns.Date*
- *canary.columns.EmailLink*
- *canary.columns.ColumnHeader*
- *canary.columns.get_column_for_field*

class `canary.columns.`**`Column`**(*label=None*, *filters=None*, *sorts=None*, *aggregates=None*)

Base column class that contains all shared functionality.

`__init__`(*label=None*, *filters=None*, *sorts=None*, *aggregates=None*)

Create a Column.

Args: label: The label used for column display (e.g. the header). filters: A list of filters to apply to the column.

class `canary.columns.`**`Text`**(*text=None*, *default=None*, **args*, ***kwargs*)

The most basic column that most people will use.

Extends canary.columns.Column.

`__init__`(*text=None*, *default=None*, **args*, ***kwargs*)

Create a Text column.

Unless overridden, the column is sortable and the Search filter is included by default.

Args: text: The column or attribute name to use for getting the text to display. default: A default value to display if the text attribute can't be resolved.

Example: thinger = Text('contact__name', None, 'Thinger', (SearchFilter,))

get_field_name()
Used by filters in order to filter by a Django column name. It is used to build the kwargs passed to QuerySet.filter.

See the filter_queryset methods on included filters to see how this is used.

resolve_text(*view*, *obj*, *default=None*)
Resolve all attributes of the column that are defined in Column.resolve.

Args: view: The report view. obj: The QuerySet object. default: A default value to use if the text can't be resolved.

Returns: A dictionary where the attribute names from Column.resolve are the keys. The values are the resolved values.

render(*view*, *obj*)
Render the column value for an object in the QuerySet.

Args: view: The report view. obj: The QuerySet object.

Returns: A string containing the resolved text.

class canary.columns.**Tag**(*text=None*, *autoid=None*, *autoid_prefix=None*, *params=None*, *allow_empty=False*, **args*, ***kwargs*)
The next most basic Column, this is the base of all columns that should be displayed as HTML elements.

Extends canary.columns.Text.

__init__(*text=None*, *autoid=None*, *autoid_prefix=None*, *params=None*, *allow_empty=False*, **args*, ***kwargs*)
Create a Tag column.

Args: text: The column name to use for getting the text to display. autoid: The attribute to use for the element id. autoid_prefix: The prefix to use for the element id. params: A dictionary of parameters to include in the element declaration. allow_empty: A boolean indicating whether the element should be rendered even if there is not content.

Example: a = Tag('title__name', 'title__id', 'report-link', ('report-link',), {'href': ''}, False, True, None, 'Thinger', (SearchFilter,))

render(*view*, *obj*)
Renders the filter controls (form, buttons).

Args: view: The canary view. obj: The QuerySet object.

Returns: A string of HTML containing the filter controls. By default, this is rendered as part of canary.columns.ColumnHeader.render.

class canary.columns.**Link**(*url=None*, **args*, ***kwargs*)
A link column. Displays the value as an anchor tag.

Extends canary.columns.Tag.

__init__(*url=None*, **args*, ***kwargs*)
Create a Link column.

Args: url: The attribute to reolve to get the url for a row object.

Example: thinger = Link('title__get_absolute_url', 'title__name', 'title__id', 'report-link-', ('report-link',), {'rel': 'title__id'}, False, True, None, 'Thinger', (SearchFilter,))

render(*view*, *obj*)
Renders the filter controls (form, buttons).

Args: view: The canary view. obj: The QuerySet object.

Returns: A string of HTML containing the filter controls. By default, this is rendered as part of canary.columns.ColumnHeader.render.

class `canary.columns.`**`ReverseLink`**(*text*, *reverse_path*, *reverse_args=None*, *reverse_kwargs=None*, **args*, ***kwargs*)

A link column, where the URL is built using Django's reverse.

Extends canary.columns.Link.

`__init__`(*text*, *reverse_path*, *reverse_args=None*, *reverse_kwargs=None*, **args*, ***kwargs*)

Create a ReverseLink column.

The reverse_path, reverse_args and reverse_kwargs arguments are the same as you would use when reversing urls with Django's reverse utility (found in django.core.urlresolvers).

The big difference is that you can provide attributes that will be resolved from the view or QuerySet object in order to provide (for example) the id or slug of the object being displayed.

Args: text: The attribute to resolve to use for getting the text to display. reverse_path: The path to be passed to Django's reverse method.

Example: thinger = ReverseLink('title__name', 'titles_title_change', ('title__id',), {}, 'title__id', 'report-admin-link-', ('report-link',), {'rel': 'title__id'}, False, True, None, 'Thinger', (SearchFilter,))

In the example, the URL is built in a way similar to: title_id = obj.title.id url = reverse('titles_title_change', (title_id,), {})

`render`(*view*, *obj*)

Renders the filter controls (form, buttons).

Args: view: The canary view. obj: The QuerySet object.

Returns: A string of HTML containing the filter controls. By default, this is rendered as part of canary.columns.ColumnHeader.render.

class `canary.columns.`**`AdminLink`**(*text*, *reverse_path*, *reverse_args=None*, *reverse_kwargs=None*, **args*, ***kwargs*)

A link column, where the URL is built using Django's reverse for an admin URL.

Extends canary.columns.ReverseLink.

`__init__`(*text*, *reverse_path*, *reverse_args=None*, *reverse_kwargs=None*, **args*, ***kwargs*)

Create an AdminLink column.

The reverse_path, reverse_args and reverse_kwargs arguments are the same as you would use when reversing urls with Django's reverse utility (found in django.core.urlresolvers), only the class prefixes the path with 'admin:' for you, as a convenience.

The big difference is that you can provide attributes that will be resolved from the view or QuerySet object in order to provide (for example) the id or slug of the object being displayed.

Args: text: The attribute to resolve to use for getting the text to display. reverse_path: The path to be passed to Django's reverse method (prefixed by 'admin:').

Example: thinger = AdminLink('title__name', 'titles_title_change', ('title__id',), {}, 'title__id', 'report-admin-link-', ('report-link',), {'rel': 'title__id'}, False, True, None, 'Thinger', (SearchFilter,))

In the example, the URL is built in a way similar to: title_id = obj.title.id url = reverse('admin:titles_title_change', (title_id,), {})

`render`(*view*, *obj*)

Renders the filter controls (form, buttons).

Args: view: The canary view. obj: The QuerySet object.

Returns: A string of HTML containing the filter controls. By default, this is rendered as part of canary.columns.ColumnHeader.render.

class `canary.columns.`**`BooleanValue`**(**args*, ***kwargs*)

A column that gets its value from a Django models.BooleanField.

Extends canary.columns.Tag.

`__init__`(**args*, ***kwargs*)

Create a BooleanValue column.

Example: thinger = BooleanValue(label='Thinger', (SearchFilter,))

`resolve_text`(*view*, *obj*, *default=None*)

Overriding to inject the 'text' attribute into the dictionary of resolved values.

Args: view: The report view. obj: The QuerySet object.

Returns: A dictionary where the attribute names from Column.resolve are the keys. The values are the resolved values.

class `canary.columns.`**`Date`**(*text=None*, *default=None*, *strftime=None*, **args*, ***kwargs*)

A column that gets its value from a Django models.DateField.

Extends canary.columns.Text.

`__init__`(*text=None*, *default=None*, *strftime=None*, **args*, ***kwargs*)

Create a Date column.

Unless overridden, the column is sortable and the DateRange filter is included by default.

Args: text: The column name to use for getting the text to display. default: A default value to display if the text attribute can't be resolved. strftime: A string for date formatting.

Sets: strftime: A string for date formatting.

Example: thinger = Text('contact__name', None, '%Y/%m/%d', 'Thinger', None, (SearchFilter,))

class `canary.columns.`**`EmailLink`**(*mailto=None*, **args*, ***kwargs*)

A column that display an e-mail link.

Extends canary.columns.Link.

`__init__`(*mailto=None*, **args*, ***kwargs*)

Create an EmailLink column.

Args: mailto: The attribute to resolve to use for the e-mail address.

`resolve_text`(*view*, *obj*, *default=None*)

Overriding to inject the 'mailto' and 'url' attributes into the dictionary of resolved values.

Args: view: The report view. obj: The QuerySet object.

Returns: A dictionary where the attribute names from Column.resolve are the keys. The values are the resolved values.

class `canary.columns.`**`ColumnHeader`**(*column*, *view*, *id=None*)

A header column.

`__init__`(*column*, *view*, *id=None*)

Create a ColumnHeader column.

Args: column: The canary column. view: The canary report view. id: An attribute to resolve to use to create an HTML id.

render(*request*)

Render the header column.

Args: request: The Django request.

Returns: A string containing HTML for the header column.

static `columns.`**get_column_for_field**(*field*)

Return an approriate canary column for a Django model field.

For example, given a Django models.DateField, this method returns a canary columns.Date object for the field.

This is handy for building a canary report view for a Django model.

Args: field: A Django model field.

Returns: A canary column of an appropriate type.

CHAPTER

FIVE

FILTERS

More info on filters (and how to create your own!!) coming soon...

5.1 Class Reference

- *canary.filters.ColumnFilter*
- *canary.filters.FormFilter*
- *canary.filters.DateRange*
- *canary.filters.Search*
- *canary.filters.Choice*
- *canary.filters.ForeignKey*

class `canary.filters.`**`ColumnFilter`**(*field_name=None*, *label=None*, *prefix=None*)

The base class of all filters.

Yes, all your column filter base belong to us...

`__init__`(*field_name=None*, *label=None*, *prefix=None*)

Create a ColumnFilter.

Args: field_name: The field name to use in filtering the QuerySet. If not provided, the column name is used. label: The label of the submit button. prefix: The prefix to use for naming the HTML object.

`get_field_name`()

Get the field name to be used to filter the QuerySet.

Returns: The field_name, if provided, or the column field name.

`load`(*request*, *queryset*)

Load data for the filter from the current request.

This method is expected to create the self._meta.data object.

Args: request: The Django Request. queryset: The Django QuerySet.

Sets: _meta.data: The data collected from the request object.

class `canary.filters.`**`FormFilter`**(**args*, ***kwargs*)

The base class of all form filters. They use a form to get the data used to filter the QuerySet.

Extends canary.filters.ColumnFilter.

`__init__`(**args*, ***kwargs*)
Create a FormFilter.

`clear_filter_data`()
Clear filter data.

Sets: data: Sets to an empty object.

`get_label`()
Get the filter label.

Returns: The provided label or 'Filter'.

`get_prefix`()
Get the filter prefix.

Returns: The provided prefix or the column name.

`filter_queryset`(*request*, *queryset*)
Filter the QuerySet.

Args: request: The Django request. queryset: The Django QuerySet.

Returns: A Django QuerySet.

Raises: NotImplementedError: If the method is not overridden by the filter class.

`load`(*request*, *queryset*)
Load data from the request.

Args: request: The Django request. queryset: The Django QuerySet.

Sets: data: Populates the object with form data.

`render`(*request*)
Renders the filter controls (form, buttons).

Args: request: The Django request.

Returns: A string of HTML containing the filter controls. By default, this is rendered as part of canary.columns.ColumnHeader.render.

`set_data`(*request*, **args*, ***kwargs*)
Set the form data.

Args: request: The Django request.

Sets: data: The form field data as attributes on an object.

class `canary.filters.`**`DateRange`**(**args*, ***kwargs*)
Filter a date column by a date range.

Extends canary.filters.FormFilter.

`filter_queryset`(*request*, *queryset*)
Filter the QuerySet.

Args: request: The Django request. queryset: The QuerySet.

Returns: A QuerySet.

class `canary.filters.`**`Search`**(*columns=None*, **args*, ***kwargs*)
Filter a column with an icontains filter.

Extends canary.filters.FormFilter.

`__init__`(*columns=None*, **args*, ***kwargs*)
Create a Search filter.

Args: columns: A list of column names to search.

`filter_queryset`(*request*, *queryset*)
Filter the QuerySet.

Args: request: The Django request. queryset: The QuerySet.

Returns: A QuerySet.

class `canary.filters.`**`Choice`**(*choices=None*, **args*, ***kwargs*)
A Choice filter, for use with Django model fields with choices.

Extends canary.filters.ColumnFilter.

`filter_queryset`(*request*, *queryset*)
Filter the QuerySet based on the selected choice.

Args: request: The Django request. queryset: The QuerySet.

Returns: A QuerySet.

`get_data`(*request*, **args*, ***kwargs*)
Get the filter data.

Args: request: The Django request.

Returns: A dictionary of form data.

`get_filter_name`()
Get the filter name.

Returns: The name of the HTML form input element.

`load`(*request*, *queryset*)
Load filter data.

Args: request: The Django request.

Sets: choices: A tuple of choices. data: And object with the filter data as attributes.

`render`(*request*)
Renders the filter controls (form, buttons).

Args: request: The Django request.

Returns: A string of HTML containing the filter controls. By default, this is rendered as part of canary.columns.ColumnHeader.render.

`set_data`(*request*)
Set the filter data.

Args: request: The Django request.

Sets: data: An object with the filter data as attributes.

class `canary.filters.`**`ForeignKey`**(*choices=None*, **args*, ***kwargs*)
A ForeignKey filter, for use with Django models.ForeignKey fields.

Extends canary.filters.Choice.

`filter_queryset`(*request*, *queryset*)
Filter the QuerySet based on the selected choice.

Args: request: The Django request. queryset: The QuerySet.

Returns: A QuerySet.

get_choices(*queryset*)

Get the choice tuple.

Args: queryset: The QuerySet.

Returns: A choice tuple (exactly the same as used to define Django model fields).

get_related_model()

Get the related Django model.

Returns: The related Django model.

Raises: NotImplementedError: Magic is hard. Until I figure out how to work related magics, you must define this.

CHAPTER

SIX

SORTS

Coming soon...

CHAPTER

SEVEN

ATTRIBUTE RESOLOTION

More info about how attribute resolution works coming soon...

7.1 Class Reference

class `canary.resolver.`**`Resolver`**(**objects*)

A class to handle the magic of finding attributes given as arguments to columns and filters.

To use, simply instantiate the class, handing over all objects on which you want to search for the attribute, in order of preference. Then call resolve, passing the attribute and (optionally) a default value to return if the attribute was not found.

Example: value = Resolver(obj, view).resolve('get_absolute_url', '')

`resolve`(*attr*, *default=None*)

Resolve the given attribute from the available objects.

Args: attr: The attribute that you want to find. default: A default value, to be used if the attribute isn't found.

Returns: The resolved attribute (or results of calling a method) or default.

`resolve_callable`(*attr*)

Before returning a found attribute, check if it is a method, and handle appropriately.

Args: attr: The found attribute.

Returns: If the attribute is a method, return the results of calling the method.

If the attribute is a view method, return the results of calling the method (passing the QuerySet object). Note: This relies on the QuerySet having been provided to the class upon instantiation.

`resolve_dict`(*attr*, *default*)

Resolve a dictionary.

Args: attr: The dictionary to be resolved.

Returns: A copy of the dictionary, with all values resolved.

`resolve_list`(*attr*, *default*)

Resolve a list.

Args: attr: The list to be resolved.

Returns: A copy of the list, with all values resolved.

`resolve_string`(*attr*, *default=None*)

Resolve a string.

Args:

attr: A string believed to be an attribute on one of the available objects.

default: A value to return if the attribute can't be resolved.

Returns: The attribute value or the results of calling an attribute that is a method.

If those fail, return default or None.

`resolve_tuple`(*attr*, *default*)

Resolve a tuple.

Args: attr: The tuple to be resolved.

Returns: A copy of the tuple, with all values resolved.

CHAPTER

EIGHT

QUESTIONS I THINK YOU'LL ASK

8.1 What's with the canary???

I was running out of ideas and the thesaurus was running dry. I was trying to think of names that could have a fun logo but related to reporting somehow. I had gone through **Sarge** (honestly...eek) and almost went with **django informer** before I decided to find a tool to help me find related words. So I found OneLook's Reverse Dictionary. It's a bit plain, but is definitely going to be a fixture in my brainstorming shelf.

So, doing a reverse search on **informer** gave me **canary**. And I figure that bird logos haven't done other folks harm. Besides, the cornier part of me digs the *canary in a coal mine* idea, where your data is the coal mine. Hopefully the ease of creating new reports will help people learn more from their data.

Also, the name that I'd been using was **basic_reports**, which is clearly lame, not to mention increasingly inaccurate as I was seeing more potential in the setup and adding more bells.

CHAPTER

NINE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX