
django-tables2

Release 1.2.3

July 25, 2016

| | | |
|----------|------------------------------------|----------|
| 1 | Table of contents | 3 |
| 1.1 | Getting started | 3 |
| 1.2 | Customizing the table | 6 |
| 1.3 | Reference | 17 |
| 1.4 | FAQ | 39 |
| 1.5 | Upgrading and change log | 40 |
| 1.6 | Glossary | 49 |

Its features include:

- Any iterable can be a data-source, but special support for Django querysets is included.
- The builtin UI does not rely on JavaScript.
- Support for automatic table generation based on a Django model.
- Supports custom column functionality via subclassing.
- Pagination.
- Column based table sorting.
- Template tag to enable trivial rendering to HTML.
- Generic view mixin.

About the app:

- [Available on pypi](#)
- Tested with python 2.7, 3.3, 3.4, 3.5 and Django 1.8, 1.9, [Travis CI](#)
- [Documentation on readthedocs.org](#)
- [Bug tracker](#)

Table of contents

1.1 Getting started

Django-tables2 is [Available on pypi](#) and can be installed using pip:

```
pip install django-tables2
```

After installing, add 'django_tables2' to INSTALLED_APPS and add 'django.core.context_processors.request' to the context_processors in your template setting OPTIONS.

1.1.1 Tutorial

After installing django-tables2, you can follow this tutorial to start using django-tables2.

Note: For this tutorial, we'll assume you use Django version 1.8.0 or greater. For Django 1.7, refer to the [Django 1.7 documentation](#).

1. `pip install django-tables2`
2. Add 'django_tables2' to INSTALLED_APPS
3. Add 'django.core.context_processors.request' to the context_processors in your template setting OPTIONS.

We're going to run through creating a tutorial app. Let's start with a simple model:

```
# tutorial/models.py
class Person(models.Model):
    name = models.CharField(verbose_name="full name")
```

Add some data so you have something to display in the table. Now write a view to pass a `Person` queryset into a template:

```
# tutorial/views.py
from django.shortcuts import render

def people(request):
    return render(request, 'people.html', {'people': Person.objects.all()})
```

Finally, implement the template:

```
{% tutorial/templates/people.html %}
{% load render_table from django_tables2 %}
<!doctype html>
<html>
  <head>
    <link rel="stylesheet" href="{% STATIC_URL %}django_tables2/themes/paleblue/css/screen.css" />
  </head>
  <body>
    {% render_table people %}
  </body>
</html>
```

Hook the view up in your URLs, and load the page, you should see:

| ID ▲ | Full Name ▲ |
|-------------|---------------|
| 1 | Bradley Ayers |
| 2 | John Doe |
| Page 1 of 1 | |

While simple, passing a queryset directly to `{% render_table %}` doesn't allow for any customisation. For that, you must define a custom *Table* class:

```
# tutorial/tables.py
import django_tables2 as tables
from .models import Person

class PersonTable(tables.Table):
    class Meta:
        model = Person
        # add class="paleblue" to <table> tag
        attrs = {'class': 'paleblue'}
```

You'll then need to instantiate and configure the table in the view, before adding it to the context:

```
# tutorial/views.py
from django.shortcuts import render
from django_tables2 import RequestConfig
from .models import Person
from .tables import PersonTable

def people(request):
    table = PersonTable(Person.objects.all())
    RequestConfig(request).configure(table)
    return render(request, 'people.html', {'table': table})
```

Using *RequestConfig* automatically pulls values from `request.GET` and updates the table accordingly. This enables data ordering and pagination.

Rather than passing a queryset to `{% render_table %}`, instead pass the table instance:

```
{% render_table table %}
```


At this point you haven't actually customised anything, you've merely added the boilerplate code that `{% render_table %}` does for you when given a `QuerySet`. The remaining sections in this document describe how to change various aspects of the table.

TODO: insert links to various customisation options here.

1.1.2 Populating a table with data

Tables can be created from a range of input data structures. If you've seen the tutorial you'll have seen a `queryset` being used, however any iterable that supports `len()` and contains items that expose key-based access to column values is fine.

List of dicts

An example we will demonstrate using list of dicts. When defining a table it is necessary to declare each column:

```
import django_tables2 as tables

data = [
    {'name': 'Bradley'},
    {'name': 'Stevie'},
]

class NameTable(tables.Table):
    name = tables.Column()

table = NameTable(data)
```

Querysets

If you build use tables to display `QuerySet` data, rather than defining each column manually in the table, the `Table.Meta.model` option allows tables to be dynamically created based on a model:

```
# models.py
class Person(models.Model):
    first_name = models.CharField(max_length=200)
    last_name = models.CharField(max_length=200)
    user = models.ForeignKey('auth.User')
    dob = models.DateField()

# tables.py
import django_tables2 as tables

class PersonTable(tables.Table):
    class Meta:
        model = Person

# views.py
def person_list(request):
    table = PersonTable(Person.objects.all())

    return render(request, 'person_list.html', {
        'table': table
    })
```

This has a number of benefits:

- Less repetition
- Column headers are defined using the field's `verbose_name`
- Specialized columns are used where possible (e.g. `DateColumn` for a `DateTimeField`)

When using this approach, the following options might be useful to customize what fields to show or hide:

- `sequence` – reorder columns
- `fields` – specify model fields to *include*
- `exclude` – specify model fields to *exclude*

1.1.3 Performance

Django-tables tries to be efficient in displaying big datasets. It tries to avoid converting the `QuerySet` instances to lists by using SQL to slice the data and should be able to handle datasets with 100k records without a problem.

However, when using one of the customisation methods described in this documentation, there is lot's of opportunity to introduce slowness. If you experience that, try to strip the table of customisations and re-add them one by one, checking for performance after each step.

1.2 Customizing the table

1.2.1 Alternative column data

Various options are available for changing the way the table is *rendered*. Each approach has a different balance of ease-of-use and flexibility.

Using Accessors

Each column has a 'key' that describes which value to pull from each record to populate the column's cells. By default, this key is just the name given to the column, but it can be changed to allow foreign key traversal or other complex cases.

To reduce ambiguity, rather than calling it a 'key', we use the name 'accessor'.

Accessors are just dotted paths that describe how an object should be traversed to reach a specific value, for example:

```
>>> from django_tables2 import A
>>> data = {'abc': {'one': {'two': 'three'}}}
>>> A('abc.one.two').resolve(data)
'three'
```

Dots represent a relationships, and are attempted in this order:

1. Dictionary lookup `a[b]`
2. Attribute lookup `a.b`
3. List index lookup `a[int(b)]`

If the resulting value is callable, it is called and the return value is used.

Table.render_foo methods

To change how a column is rendered, define a `render_foo` method on the table for example: `render_row_number()` for a column named `row_number`. This approach is suitable if you have a one-off change that you do not want to use in multiple tables.

Supported keyword arguments include:

- `record` – the entire record for the row from the *table data*
- `value` – the value for the cell retrieved from the *table data*
- `column` – the *Column* object
- `bound_column` – the *BoundColumn* object
- `bound_row` – the *BoundRow* object
- `table` – alias for `self`

This example shows how to render the row number in the first row:

```
>>> import django_tables2 as tables
>>> import itertools
>>> class SimpleTable(tables.Table):
...     row_number = tables.Column(empty_values=())
...     id = tables.Column()
...     age = tables.Column()
...
...     def __init__(self, *args, **kwargs):
...         super(SimpleTable, self).__init__(*args, **kwargs)
...         self.counter = itertools.count()
...
...     def render_row_number(self):
...         return 'Row %d' % next(self.counter)
...
...     def render_id(self, value):
...         return '<%s>' % value
...
>>> table = SimpleTable([{'age': 31, 'id': 10}, {'age': 34, 'id': 11}])
>>> print ', '.join(map(str, table.rows[0]))
Row 0, <10>, 31
```

Python's `inspect.getargspec` is used to only pass the arguments declared by the function. This means it's not necessary to add a catch all (`**`) keyword argument.

Important: `render` methods are *only* called if the value for a cell is determined to be not an *empty value*. When a value is in `Column.empty_values`, a default value is rendered instead (both `Column.render` and `Table.render_FOO` are skipped).

Subclassing Column

Defining a column subclass allows functionality to be reused across tables. Columns have a `render` method that behaves the same as *Table.render_foo methods* methods on tables:

```
>>> import django_tables2 as tables
>>>
>>> class UpperColumn(tables.Column):
```

```

...     def render(self, value):
...         return value.upper()
...
>>> class Example(tables.Table):
...     normal = tables.Column()
...     upper = UpperColumn()
...
>>> data = [{'normal': 'Hi there!',
...           'upper': 'Hi there!'}]
...
>>> table = Example(data)
>>> # renders to something like this:
'''<table>
    <thead><tr><th>Normal</th><th>Upper</th></tr></thead>
    <tbody><tr><td>Hi there!</td><td>HI THERE!</td></tr></tbody>
</table>'''

```

See *Table.render_foo methods* for a list of arguments that can be accepted.

For complicated columns, you may want to return HTML from the `render()` method. Make sure to use Django's `html` formatting functions:

```
>>> from django.utils.html import format_html
>>>
>>> class ImageColumn(tables.Column):
...     def render(self, value):
...         return format_html('', value)
... 
```

1.2.2 Alternative column ordering

When using `queryset` data, one might want to show a computed value which is not in the database. In this case, attempting to order the column will cause an exception:

```
# models.py
class Person(models.Model):
    first_name = models.CharField(max_length=200)
    family_name = models.CharField(max_length=200)

    @property
    def name(self):
        return '{} {}'.format(self.first_name, self.family_name)

# tables.py
class PersonTable(tables.Table):
    name = tables.Column()
```

```
>>> table = PersonTable(Person.objects.all())
>>> table.order_by = 'name'
>>>
>>> # will result in:
FieldError: Cannot resolve keyword 'name' into field. Choices are: first_name, family_name
```

To prevent this, `django-tables2` allows two ways to specify custom ordering: `accessors` and `order_FOO()` methods.

Ordering by accessors

You can supply an `order_by` argument containing a name or a tuple of the names of the columns the database should use to sort it:

```
class PersonTable(tables.Table):
    name = tables.Column(order_by=('first_name', 'family_name'))
```

Accessor syntax can be used as well, as long as they point to a model field.

If ordering does not make sense for a particular column, it can be disabled via the `orderable` argument:

```
class SimpleTable(tables.Table):
    name = tables.Column()
    actions = tables.Column(orderable=False)
```

table.order_FOO() methods

Another solution for alternative ordering is being able to chain functions on to the original queryset. This method allows more complex functionality giving the ability to use all of Django's QuerySet API.

Adding a `Table.order_FOO` method (where `FOO` is the name of the column), gives you the ability to chain to, or modify, the original queryset when that column is selected to be ordered.

The method takes two arguments: `queryset`, and `is_descending`. The return must be a tuple of two elements. The first being the queryset and the second being a boolean; note that modified queryset will only be used if the boolean is `True`.

For example, let's say instead of ordering alphabetically, ordering by amount of characters in the `first_name` is desired. The implementation would look like this:

```
# tables.py
from django.db.models.functions import Length

class PersonTable(tables.Table):
    name = tables.Column()

    def order_name(self, queryset, is_descending):
        queryset = queryset.annotate(
            length=Length('first_name')
        ).order_by((- ' if is_descending else ' + 'length')
        return (queryset, True)
```

As another example, presume the situation calls for being able to order by a mathematical expression. In this scenario, the table needs to be able to be ordered by the sum of both the shirts and the pants. The custom column will have its value rendered using *Table.render_foo methods*.

This can be achieved like this:

```
# models.py
class Person(models.Model):
    first_name = models.CharField(max_length=200)
    family_name = models.CharField(max_length=200)
    shirts = models.IntegerField()
    pants = models.IntegerField()

# tables.py
from django.db.models import F
```

```
class PersonTable(tables.Table):
    clothing = tables.Column()

    class Meta:
        model = Person

    def render_clothing(self, record):
        return str(record.shirts + record.pants)

    def order_clothing(self, queryset, is_descending):
        queryset = queryset.annotate(
            amount=F('shirts') + F('pants')
        ).order_by((- ' if is_descending else ' ) + 'amount')
        return (queryset, True)
```

1.2.3 Column attributes

Column attributes can be specified using the `dict` with specific keys. The dict defines HTML attributes for one of more elements within the column. Depending on the column, different elements are supported, however `th`, `td`, and `cell` are supported universally:

```
>>> import django_tables2 as tables
>>>
>>> class SimpleTable(tables.Table):
...     name = tables.Column(attrs={'th': {'id': 'foo'}})
...
>>> # will render something like this:
'{snip}<thead><tr><th id="foo" class="name">{snip}<tbody><tr><td class="name">{snip}'
```

For `th` and `td`, the column name will be added as a class name. This makes selecting the row for styling easier. Have a look at each column's API reference to find which elements are supported.

1.2.4 Row attributes

Row attributes can be specified using a dict defining the HTML attributes for the `<tr>` element on each row. The values of the dict may be

By default, class names *odd* and *even* are supplied to the rows, which can be customized using the `row_attrs` `Table.Meta` attribute or as argument to the constructor of `Table`, for example:

```
class Table(tables.Table):
    class Meta:
        model = User
        row_attrs = {
            'data-id': lambda record: record.pk
        }
```

will render tables with the following `<tr>` tag

```
<tr class="odd" data-id="1"> [...] </tr>
<tr class="even" data-id="2"> [...] </tr>
```

1.2.5 Customizing headers and footers

By default an header and no footer will be rendered.

Adding column headers

The header cell for each column comes from `header`. By default this method returns `verbose_name`, falling back to the titlised attribute name of the column in the table class.

When using queryset data and a verbose name hasn't been explicitly defined for a column, the corresponding model field's `verbose_name` will be used.

Consider the following:

```
>>> class Region(models.Model):
...     name = models.CharField(max_length=200)
...
>>> class Person(models.Model):
...     first_name = models.CharField(verbose_name='model verbose name', max_length=200)
...     last_name = models.CharField(max_length=200)
...     region = models.ForeignKey('Region')
...
>>> class PersonTable(tables.Table):
...     first_name = tables.Column()
...     ln = tables.Column(accessor='last_name')
...     region_name = tables.Column(accessor='region.name')
...
>>> table = PersonTable(Person.objects.all())
>>> table.columns['first_name'].header
'Model Verbose Name'
>>> table.columns['ln'].header
'Last Name'
>>> table.columns['region_name'].header
'Name'
```

As you can see in the last example (region name), the results are not always desirable when an accessor is used to cross relationships. To get around this be careful to define `Column.verbose_name`.

Changing class names for ordered column headers

When a column is ordered in an ascending state there needs to be a way to show it in the interface. `django-tables2` does this by adding an `asc` class for ascending or a `desc` class for descending. It should also be known that any orderable column is added with an `orderable` class to the column header.

Sometimes there may be a need to change these default classes.

On the `attrs` attribute of the table, you can add a `th` key with the value of a dictionary. Within that `th` dictionary, you may add an `_ordering` key also with the value of a dictionary.

The `_ordering` element is optional and all elements within it are optional. Inside you can have an `orderable` element, which will change the default `orderable` class name. You can also have `ascending` which will change the default `asc` class name. And lastly, you can have `descending` which will change the default `desc` class name.

Example:

```
class Table(tables.Table):
    Meta:
        attrs = {
            'th': {
                '_ordering': {
                    'orderable': 'sortable', # Instead of `orderable`
                    'ascending': 'ascend',    # Instead of `asc`
                }
            }
        }
```

```

        'descending': 'descend' # Instead of `desc`
    }
}

```

It can also be specified at initialization using the `attrs` for both: table and column:

```

ATTRIBUTES = {
    'th': {
        '_ordering': {
            'orderable': 'sortable', # Instead of `orderable`
            'ascending': 'ascend',   # Instead of `asc`
            'descending': 'descend'  # Instead of `desc`
        }
    }
}

table = tables.Table(queryset, attrs=ATTRIBUTES)

# OR

class Table(tables.Table):
    my_column = tables.Column(attrs=ATTRIBUTES)

```

Adding column footers

By default, no footer will be rendered. If you want to add a footer, define a footer on at least one column.

That will make the table render a footer on every view of the table. It's up to you to decide if that makes sense if your table is paginated.

Pass footer-argument to the Column constructor.

The simplest case is just passing a `str` as the footer argument to a column:

```
country = tables.Column(footer='Total:')
```

This will just render the string in the footer. If you need to do more complex things, like showing a sum or an average, you can pass a callable:

```

population = tables.Column(
    footer=lambda table: sum(x['population'] for x in table.data)
)

```

You can expect `table`, `column` and `bound_column` as argument.

Define `render_footer` on a custom column.

If you need the same footer in multiple columns, you can create your own custom column. For example this column that renders the sum of the values in the column:

```

class SummingColumn(tables.Column):
    def render_footer(self, bound_column, table):
        return sum(bound_column.accessor.resolve(row) for row in table.data)

```


Then use this column like so:

```
class Table(tables.Table):
    name = tables.Column()
    country = tables.Column(footer='Total:')
    population = SummingColumn()
```

Note: If you are `summing` over tables with big datasets, chances are it's going to be slow. You should use some database aggregation function instead.

1.2.6 Swapping the position of columns

By default columns are positioned in the same order as they are declared, however when mixing auto-generated columns (via `Table.Meta.model`) with manually declared columns, the column sequence becomes ambiguous.

To resolve the ambiguity, columns sequence can be declared via the `Table.Meta.sequence` option:

```
class PersonTable(tables.Table):
    selection = tables.CheckBoxColumn(accessor='pk', orderable=False)

    class Meta:
        model = Person
        sequence = ('selection', 'first_name', 'last_name')
```

The special value `'...'` can be used to indicate that any omitted columns should be inserted at that location. As such it can be used at most once.

1.2.7 Pagination

Pagination is easy, just call `Table.paginate()` and pass in the current page number:

```
def people_listing(request):
    table = PeopleTable(Person.objects.all())
    table.paginate(page=request.GET.get('page', 1), per_page=25)
    return render(request, 'people_listing.html', {'table': table})
```

If you're using `RequestConfig`, pass pagination options to the constructor:

```
def people_listing(request):
    table = PeopleTable(Person.objects.all())
    RequestConfig(request, paginate={'per_page': 25}).configure(table)
    return render(request, 'people_listing.html', {'table': table})
```

1.2.8 Table Mixins

It's possible to create a mixin for a table that overrides something, however unless it itself is a subclass of `Table` class variable instances of `Column` will **not** be added to the class which is using the mixin.

Example:

```
>>> class UselessMixin(object):
...     extra = tables.Column()
...
>>> class TestTable(UselessMixin, tables.Table):
```

```
...     name = tables.Column()
...
>>> TestTable.base_columns.keys()
['name']
```

To have a mixin contribute a column, it needs to be a subclass of *Table*. With this in mind the previous example *should* have been written as follows:

```
>>> class UsefulMixin(tables.Table):
...     extra = tables.Column()
...
>>> class TestTable(UsefulMixin, tables.Table):
...     name = tables.Column()
...
>>> TestTable.base_columns.keys()
['extra', 'name']
```

1.2.9 Customizing table style

CSS

In order to use CSS to style a table, you'll probably want to add a `class` or `id` attribute to the `<table>` element. `django-tables2` has a hook that allows arbitrary attributes to be added to the `<table>` tag.

```
>>> import django_tables2 as tables
>>>
>>> class SimpleTable(tables.Table):
...     id = tables.Column()
...     age = tables.Column()
...
...     class Meta:
...         attrs = {'class': 'mytable'}
...
>>> table = SimpleTable()
>>> # renders to something like this:
'<table class="mytable">...'
```

Custom Template

And of course if you want full control over the way the table is rendered, ignore the built-in generation tools, and instead pass an instance of your *Table* subclass into your own template, and render it yourself.

Have a look at the `django_tables2/table.html` template for an example.

1.2.10 Querystring fields

Tables pass data via the querystring to indicate ordering and pagination preferences.

The names of the querystring variables are configurable via the options:

- `order_by_field` – default: `'sort'`
- `page_field` – default: `'page'`
- `per_page_field` – default: `'per_page'`, **note:** this field currently isn't used by `{% render_table %}`

Each of these can be specified in three places:

- `Table.Meta.foo`
- `Table(..., foo=...)`
- `Table(...).foo = ...`

If you're using multiple tables on a single page, you'll want to prefix these fields with a table-specific name, in order to prevent links on one table interfere with those on another table:

```
def people_listing(request):
    config = RequestConfig(request)
    table1 = PeopleTable(Person.objects.all(), prefix='1-') # prefix specified
    table2 = PeopleTable(Person.objects.all(), prefix='2-') # prefix specified
    config.configure(table1)
    config.configure(table2)

    return render(request, 'people_listing.html', {
        'table1': table1,
        'table2': table2
    })
```

1.2.11 Controlling localization

Django-tables2 allows you to define which column of a table should or should not be localized. For example you may want to use this feature in following use cases:

- You want to format some columns representing for example numeric values in the given locales even if you don't enable `USE_L10N` in your settings file.
- You don't want to format primary key values in your table even if you enabled `USE_L10N` in your settings file.

This control is done by using two filter functions in Django's `l10n` library named `localize` and `unlocalize`. Check out Django docs about `localization` for more information about them.

There are two ways of controlling localization in your columns.

First one is setting the `localize` attribute in your column definition to `True` or `False`. Like so:

```
class PersonTable(tables.Table):
    id = tables.Column(name='id', accessor='pk', localize=False)
    class Meta:
        model = Person
```

Note: The default value of the `localize` attribute is `None` which means the formatting of columns is dependant from the `USE_L10N` setting.

The second way is to define a `localize` and/or `unlocalize` tuples in your tables `Meta` class (just like with `fields` or `exclude`). You can do this like so:

```
class PersonTable(tables.Table):
    id = tables.Column(accessor='pk')
    value = tables.Column(accessor='some_numerical_field')
    class Meta:
        model = Person
        unlocalize = ('id', )
        localize = ('value', )
```

If you define the same column in both `localize` and `unlocalize` then the value of this column will be ‘unlocalized’ which means that `unlocalize` has higher precedence.

1.2.12 Class Based Generic Mixins

Django-tables2 comes with two class based view mixins: *SingleTableMixin* and *MultiTableMixin*.

A single table using *SingleTableMixin*

SingleTableMixin makes it trivial to incorporate a table into a view or template.

The following view parameters are supported:

- `table_class` — the table class to use, e.g. `SimpleTable`
- `table_data` (or `get_table_data()`) — the data used to populate the table
- `context_table_name` — the name of template variable containing the table object
- `table_pagination` (or `get_table_pagination`) — pagination options to pass to *RequestConfig*. Set `table_pagination=False` to disable pagination.

For example:

```
from django_tables2 import SingleTableView

class Person(models.Model):
    first_name = models.CharField(max_length=200)
    last_name = models.CharField(max_length=200)

class PersonTable(tables.Table):
    class Meta:
        model = Person

class PersonList(SingleTableView):
    model = Person
    table_class = PersonTable
```

The template could then be as simple as:

```
{% load render_table from django_tables2 %}
{% render_table table %}
```

Such little code is possible due to the example above taking advantage of default values and *SingleTableMixin*’s eagerness at finding data sources when one isn’t explicitly defined.

Note: You don’t have to base your view on *ListView*, you’re able to mix *SingleTableMixin* directly.

Multiple tables using *MultipleObjectMixin*

If you need more than one table in a single view you can use *MultiTableMixin*. It manages multiple tables for you and takes care of adding the appropriate prefixes for them. Just define a list of tables in the `tables` attribute:

```

from django_tables2 import MultiTableMixin
from django.views.generic.base import TemplateView

class PersonTableView(MultiTableMixin, TemplateView):
    template_name = 'multiTable.html'
    tables = [
        PersonTable(qs),
        PersonTable(qs, exclude=('country', ))
    ]

    table_pagination = {
        'per_page': 10
    }

```

In the template, you get a variable `tables`, which you can loop over like this:

```

{% for table in tables %}
    {% render_table table %}
{% endfor %}

```

1.3 Reference

1.3.1 Built-in columns

For common use-cases the following columns are included:

- *BooleanColumn* – renders boolean values
- *Column* – generic column
- *CheckBoxColumn* – renders checkbox form inputs
- *DateColumn* – date formatting
- *DateTimeColumn* – datetime formatting in the local timezone
- *FileColumn* – renders files as links
- *EmailColumn* – renders `` tags
- *LinkColumn* – renders `` tags (compose a django url)
- *RelatedLinkColumn* – renders `` tags linking related objects
- *TemplateColumn* – renders template code
- *URLColumn* – renders `` tags (absolute url)

1.3.2 Template tags

`render_table`

Renders a *Table* object to HTML and enables as many features in the output as possible.

```

{% load django_tables2 %}
{% render_table table %}

```

```
{# Alternatively a specific template can be used #}
{% render_table table "path/to/custom_table_template.html" %}
```

If the second argument (template path) is given, the template will be rendered with a `RequestContext` and the table will be in the variable `table`.

Note: This tag temporarily modifies the `Table` object during rendering. A context attribute is added to the table, providing columns with access to the current context for their own rendering (e.g. `TemplateColumn`).

This tag requires that the template in which it's rendered contains the `HttpRequest` inside a `request` variable. This can be achieved by ensuring the `TEMPLATES[]['OPTIONS']['context_processors']` setting contains `django.template.context_processors.request`. Please refer to the Django documentation for the `TEMPLATES`-setting.

querystring

A utility that allows you to update a portion of the query-string without overwriting the entire thing.

Let's assume we have the querystring `?search=pirates&sort=name&page=5` and we want to update the `sort` parameter:

```
{% querystring "sort"="dob" %}           # ?search=pirates&sort=dob&page=5
{% querystring "sort"="" %}             # ?search=pirates&page=5
{% querystring "sort="" "search"="" %}  # ?page=5

{% with "search" as key %}              # supports variables as keys
{% querystring key="robots" %}          # ?search=robots&page=5
{% endwith %}
```

This tag requires the `django.core.context_processors.request` context processor, see [render_table](#).

1.3.3 API Reference

Accessor (A)

class `django_tables2.utils.Accessor`

A string describing a path from one object to another via attribute/index accesses. For convenience, the class has an alias `A` to allow for more concise code.

Relations are separated by a `.` character.

RequestConfig

class `django_tables2.config.RequestConfig` (*request*, *paginate=True*)

A configurator that uses request data to setup a table.

A single `RequestConfig` can be used for multiple tables in one view. See [pagination]

Parameters `paginate` (*dict or bool*) – Indicates whether to paginate, and if so, what default values to use. If the value evaluates to `False`, pagination will be disabled. A `dict` can be used to specify default values for the call to `paginate` (e.g. to define a default `per_page` value).

A special `silent` item can be used to enable automatic handling of pagination exceptions using the following logic:

- If `PageNotAnInteger` is raised, show the first page.
- If `EmptyPage` is raised, show the last page.

Table

```
class django_tables2.tables.Table(data, order_by=None, orderable=None, empty_text=None,
                                  exclude=None, attrs=None, row_attrs=None, sequence=None,
                                  prefix=None, order_by_field=None, page_field=None,
                                  per_page_field=None, template=None, default=None, re-
                                  quest=None, show_header=None, show_footer=True)
```

A representation of a table.

Parameters

- **data** (*queryset, list of dicts*) – The data to display.
- **order_by** – (tuple or str): The default ordering tuple or comma separated str. A hyphen – can be used to prefix a column name to indicate *descending* order, for example: ('name', '-age') or name, -age.
- **orderable** (*bool*) – Enable/disable column ordering on this table
- **empty_text** (*str*) – Empty text to render when the table has no data. (default `Table.Meta.empty_text`)
- **exclude** (*iterable or str*) – The names of columns that shouldn't be included in the table.
- **attrs** (*dict*) – HTML attributes to add to the `<table>` tag. When accessing the attribute, the value is always returned as an `AttributeDict` to allow easily conversion to HTML.
- **row_attrs** – Add custom html attributes to the table rows. Allows custom HTML attributes to be specified which will be added to the `<tr>` tag of the rendered table.
- **sequence** (*iterable*) – The sequence/order of columns the columns (from left to right). Items in the sequence must be *column names*, or '...' (string containing three periods). can be used as a catch-all for columns that aren't specified.
- **prefix** (*str*) – A prefix for querystring fields. To avoid name-clashes when using multiple tables on single page.
- **order_by_field** (*str*) – If not `None`, defines the name of the *order by* querystring field in the url.
- **page_field** (*str*) – If not `None`, defines the name of the *current page* querystring field.
- **per_page_field** (*str*) – If not `None`, defines the name of the *per page* querystring field.
- **template** (*str*) – The template to render when using `{% render_table %}` (default `'django_tables2/table.html'`)
- **default** (*str*) – Text to render in empty cells (determined by `Column.empty_values`, default `Table.Meta.default`)
- **request** – Django's request to avoid using `RequestConfig`
- **show_header** (*bool*) – If `False`, the table will not have a header (`<thead>`), defaults to `True`

- **show_footer** (*bool*) – If `False`, the table footer will not be rendered, even if some columns have a footer, defaults to `True`.

as_html (*request*)

Render the table to an HTML table, adding `request` to the context.

paginate (*klass=<class 'django.core.paginator.Paginator'>, per_page=None, page=1, *args, **kwargs*)

Paginates the table using a paginator and creates a `page` property containing information for the current page.

Parameters

- **klass** (*Paginator*) – A paginator class to paginate the results.
- **per_page** (*int*) – Number of records to display on each page.
- **page** (*int*) – Page to display.

Extra arguments are passed to the paginator.

Pagination exceptions (`EmptyPage` and `PageNotAnInteger`) may be raised from this method and should be handled by the caller.

Table.Meta

class Table.Meta

Provides a way to define *global* settings for table, as opposed to defining them for each instance.

For example, if you want to create a table of users with their primary key added as a `data-id` attribute on each `<tr>`, You can use the following:

```
class UsersTable(tables.Table):
    class Meta:
        row_attrs = {'data-id': lambda record: record.pk}
```

Which adds the desired `row_attrs` to every instance of `UsersTable`, in contrast of defining it at construction time:

```
table = tables.Table(User.objects.all(),
    row_attrs={'data-id': lambda record: record.pk})
```

Some settings are only available in `Table.Meta` and not as an argument to the `Table` constructor.

Note: If you define a class `Meta` on a child of a table already having a class `Meta` defined, you need to specify the parent's `Meta` class as the parent for the class `Meta` in the child:

```
class PersonTable(table.Table):
    class Meta:
        model = Person
        exclude = ('email', )

class PersonWithEmailTable(PersonTable):
    class Meta(PersonTable.Meta):
        exclude = ()
```

All attributes are overwritten if defined in the child's class `Meta`, no merging is attempted.

Arguments:

attrs (dict): Add custom HTML attributes to the table. Allows custom HTML attributes to be specified which will be added to the `<table>` tag of any table rendered via `Table.as_html()` or the `render_table` template tag.

This is typically used to enable a theme for a table (which is done by adding a CSS class to the `<table>` element):

```
class SimpleTable(tables.Table):
    name = tables.Column()

    class Meta:
        attrs = {'class': 'paleblue'}
```

If you supply a callable as a value in the dict, it will be called at table instantiation and the returned value will be used:

Consider this example where each table gets an unique "id" attribute:

```
import itertools
counter = itertools.count()

class UniqueIdTable(tables.Table):
    name = tables.Column()

    class Meta:
        attrs = {'id': lambda: 'table_%d' % next(counter)}
```

Note: This functionality is also available via the `attrs` keyword argument to a table's constructor.

row_attrs (dict): Add custom html attributes to the table rows. Allows custom HTML attributes to be specified which will be added to the `<tr>` tag of the rendered table.

This can be used to add each record's primary key to each row:

```
class PersonTable(tables.Table):
    class Meta:
        model = Person
        row_attrs = {'data-id': lambda record: record.pk}

# will result in
# <tr data-id="1">...</tr>
```

New in version 1.2.0.

Note: This functionality is also available via the `row_attrs` keyword argument to a table's constructor.

empty_text (str): Defines the text to display when the table has no rows. If the table is empty and `bool(empty_text)` is `True`, a row is displayed containing `empty_text`. This allows a message such as *There are currently no FOO.* to be displayed.

Note: This functionality is also available via the `empty_text` keyword argument to a table's constructor.

show_header (bool): Whether or not to show the table header. Defines whether the table header should be displayed or not, by default, the header shows the column names.

Note: This functionality is also available via the `show_header` keyword argument to a table's constructor.

exclude (tuple or str): Exclude columns from the table. This is useful in subclasses to exclude columns in a parent:

```
>>> class Person(tables.Table):
...     first_name = tables.Column()
...     last_name = tables.Column()
...
>>> Person.base_columns
{'first_name': <django_tables2.columns.Column object at 0x10046df10>,
'last_name': <django_tables2.columns.Column object at 0x10046d8d0>}
>>> class ForgetfulPerson(Person):
...     class Meta:
...         exclude = ('last_name', )
...
>>> ForgetfulPerson.base_columns
{'first_name': <django_tables2.columns.Column object at 0x10046df10>}
```

Note: This functionality is also available via the `exclude` keyword argument to a table's constructor.

However, unlike some of the other `Table.Meta` options, providing the `exclude` keyword to a table's constructor **won't override** the `Meta.exclude`. Instead, it will be effectively be *added* to it. i.e. you can't use the constructor's `exclude` argument to *undo* an exclusion.

fields (tuple or str): Fields to show in the table. Used in conjunction with `model`, specifies which fields should have columns in the table. If `None`, all fields are used, otherwise only those named:

```
# models.py
class Person(models.Model):
    first_name = models.CharField(max_length=200)
    last_name = models.CharField(max_length=200)

# tables.py
class PersonTable(tables.Table):
    class Meta:
        model = Person
        fields = ('first_name', )
```

model (django.core.db.models.Model): Create columns from model. A model to inspect and automatically create corresponding columns.

This option allows a Django model to be specified to cause the table to automatically generate columns that correspond to the fields in a model.

order_by (tuple or str): The default ordering tuple or comma separated str. A hyphen - can be used to prefix a column name to indicate *descending* order, for example: ('name', '-age') or name,-age.

Note: This functionality is also available via the `order_by` keyword argument to a table's constructor.

sequence (iterable): The sequence of the table columns. This allows the default order of columns (the order they were defined in the Table) to be overridden.

The special item '...' can be used as a placeholder that will be replaced with all the columns that weren't explicitly listed. This allows you to add columns to the front or back when using inheritance.

Example:

```
>>> class Person (tables.Table):
...     first_name = tables.Column()
...     last_name = tables.Column()
...
...     class Meta:
...         sequence = ('last_name', '...')
...
>>> Person.base_columns.keys()
['last_name', 'first_name']
```

The '...' item can be used at most once in the sequence value. If it's not used, every column *must* be explicitly included. e.g. in the above example, `sequence = ('last_name',)` would be **invalid** because neither '...' or 'first_name' were included.

Note: This functionality is also available via the `sequence` keyword argument to a table's constructor.

orderable (bool): Default value for column's *orderable* attribute. If the table and column don't specify a value, a column's `orderable` value will fallback to this. This provides an easy mechanism to disable ordering on an entire table, without adding `orderable=False` to each column in a table.

Note: This functionality is also available via the `orderable` keyword argument to a table's constructor.

template (str): The default template to use when rendering the table.

Note: This functionality is also available via the `template` keyword argument to a table's constructor.

localize (str or tuple): Specifies which fields should be localized in the table. Read [Controlling localization](#) for more information.

unlocalize (str or tuple): Specifies which fields should be unlocalized in the table. Read [Controlling localization](#) for more information.

Column

class `django_tables2.columns.Column` (*verbose_name=None, accessor=None, default=None, visible=True, orderable=None, attrs=None, order_by=None, empty_values=None, localize=None, footer=None*)

Represents a single column of a table.

`Column` objects control the way a column (including the cells that fall within it) are rendered.

Parameters

- **attrs** (*dict*) – HTML attributes for elements that make up the column. This API is extended by subclasses to allow arbitrary HTML attributes to be added to the output.

By default *Column* supports:

- *th* – `table/thead/tr/th` elements
- *td* – `table/tbody/tr/td` elements
- *cell* – fallback if *th* or *td* isn't defined

- **accessor** (str or *Accessor*) – An accessor that describes how to extract values for this column from the *table data*.
- **default** (str or callable) – The default value for the column. This can be a value or a callable object ¹. If an object in the data provides `None` for a column, the default will be used instead.

The default value may affect ordering, depending on the type of data the table is using. The only case where ordering is not affected is when a `QuerySet` is used as the table data (since sorting is performed by the database).

- **order_by** (str, tuple or *Accessor*) – Allows one or more accessors to be used for ordering rather than *accessor*.
- **orderable** (bool) – If `False`, this column will not be allowed to influence row ordering/sorting.
- **verbose_name** (str) – A human readable version of the column name.
- **visible** (bool) – If `True`, this column will be rendered.
- **localize** – If the cells in this column will be localized by the `localize` filter:
 - If `True`, force localization
 - If `False`, values are not localized
 - If `None` (default), localization depends on the `USE_L10N` setting.

BooleanColumn

class `django_tables2.columns.BooleanColumn` (*null=False, yesno=u'u2714, u2718', **kwargs*)
A column suitable for rendering boolean data.

Parameters

- **null** (bool) – is `None` different from `False`?
- **yesno** (str) – text to display for True/False values, comma separated

Rendered values are wrapped in a `` to allow customisation by themes. By default the span is given the class `true, false`.

In addition to *attrs* keys supported by *Column*, the following are available:

- *span* – adds attributes to the `` tag

CheckBoxColumn

class `django_tables2.columns.CheckBoxColumn` (*attrs=None, checked=None, **extra*)
A subclass of *Column* that renders as a checkbox form input.

This column allows a user to *select* a set of rows. The selection information can then be used to apply some operation (e.g. “delete”) onto the set of objects that correspond to the selected rows.

¹ The provided callable object must not expect to receive any arguments.

The value that is extracted from the *table data* for this column is used as the value for the checkbox, i.e. `<input type="checkbox" value="..." />`

This class implements some sensible defaults:

- HTML input's name attribute is the *column name* (can override via *attrs* argument).
- *orderable* defaults to `False`.

Parameters

- **attrs** (*dict*) – In addition to *attrs* keys supported by *Column*, the following are available:
 - *input* – `<input>` elements in both `<td>` and `<th>`.
 - *th_input* – Replaces *input* attrs in header cells.
 - *td_input* – Replaces *input* attrs in body cells.
- **checked** (*Accessor*, bool, callable) – Allow rendering the checkbox as checked. If it resolves to a truthy value, the checkbox will be rendered as checked.

Note: You'd expect that you could select multiple checkboxes in the rendered table and then *do something* with that. This functionality isn't implemented. If you want something to actually happen, you'll need to implement that yourself.

is_checked (*value*, *record*)

Determine if the checkbox should be checked

DateColumn

class django_tables2.columns.**DateColumn** (*format=None*, *short=True*, **args*, ***kwargs*)

A column that renders dates in the local timezone.

Parameters

- **format** (*str*) – format string in same format as Django's date template filter (optional)
- **short** (*bool*) – if *format* is not specified, use Django's `SHORT_DATE_FORMAT` setting, otherwise use `DATE_FORMAT`

DateTimeColumn

class django_tables2.columns.**DateTimeColumn** (*format=None*, *short=True*, **args*, ***kwargs*)

A column that renders datetimes in the local timezone.

Parameters

- **format** (*str*) – format string for datetime (optional). Note that *format* uses Django's date template tag syntax.
- **short** (*bool*) – if *format* is not specified, use Django's `SHORT_DATETIME_FORMAT`, else `DATETIME_FORMAT`

EmailColumn

class django_tables2.columns.**EmailColumn** (*attrs=None, text=None, *args, **kwargs*)
Render email addresses to mailto-links.

Parameters

- **attrs** (*dict*) – HTML attributes that are added to the rendered `...` tag
- **text** – Either static text, or a callable. If set, this will be used to render the text inside link instead of the value

Example:

```
# models.py
class Person(models.Model):
    name = models.CharField(max_length=200)
    email = models.EmailField()

# tables.py
class PeopleTable(tables.Table):
    name = tables.Column()
    email = tables.EmailColumn()

# result
# [...]<a href="mailto:email@example.com">email@example.com</a>
```

FileColumn

class django_tables2.columns.**FileColumn** (*verify_exists=True, **kwargs*)
Attempts to render FieldFile (or other storage backend File) as a hyperlink.

When the file is accessible via a URL, the file is rendered as a hyperlink. The basename is used as the text:

```
<a href="/media/path/to/receipt.pdf" title="path/to/receipt.pdf">receipt.pdf</a>
```

When unable to determine the URL, a span is used instead:

```
<span title="path/to/receipt.pdf">receipt.pdf</span>
```

Column.attrs keys a and span can be used to add additional attributes.

Parameters

- **verify_exists** (*bool*) – attempt to determine if the file exists If *verify_exists*, the HTML class exists or missing is added to the element to indicate the integrity of the storage.
- **text** (*str or callable*) – Either static text, or a callable. If set, this will be used to render the text inside the link instead of the file's basename (default)

LinkColumn

class django_tables2.columns.**LinkColumn** (*viewname=None, urlconf=None, args=None, kwargs=None, current_app=None, attrs=None, **extra*)

Renders a normal value as an internal hyperlink to another page.

It's common to have the primary value in a row hyperlinked to the page dedicated to that record.

The first arguments are identical to that of `reverse` and allows an internal URL to be described. If this argument is `None`, then `get_absolute_url`. (see Django references) will be used. The last argument `attrs` allows custom HTML attributes to be added to the rendered `` tag.

Parameters

- **viewname** (*str*) – See `reverse`, or use `None` to use the model's `get_absolute_url`
- **urlconf** (*str*) – See `reverse`.
- **args** (*list*) – See `reverse`.²
- **kwargs** (*dict*) – See `reverse`.²
- **current_app** (*str*) – See `reverse`.
- **attrs** (*dict*) – HTML attributes that are added to the rendered `<a ...>...` tag.
- **text** (*str or callable*) – Either static text, or a callable. If set, this will be used to render the text inside link instead of value (default). The callable gets the record being rendered as argument.

Example:

```
# models.py
class Person(models.Model):
    name = models.CharField(max_length=200)

# urls.py
urlpatterns = patterns('',
    url('people/(\d+)/', views.people_detail, name='people_detail')
)

# tables.py
from django_tables2.utils import A # alias for Accessor

class PeopleTable(tables.Table):
    name = tables.LinkColumn('people_detail', args=[A('pk')])
```

In order to override the text value (i.e. `<a ... >text`) consider the following example:

```
# tables.py
from django_tables2.utils import A # alias for Accessor

class PeopleTable(tables.Table):
    name = tables.LinkColumn('people_detail', text='static text', args=[A('pk')])
    age = tables.LinkColumn('people_detail', text=lambda record: record.name, args=[A('pk')])
```

In the first example, a static text would be rendered ('static text') In the second example, you can specify a callable which accepts a record object (and thus can return anything from it)

In addition to `attrs` keys supported by `Column`, the following are available:

- **a** – `<a>` elements in `<td>`.

compose_url (*record, *args, **kwargs*)

Compose the url if the column is constructed with a viewname.

² In order to create a link to a URL that relies on information in the current row, `Accessor` objects can be used in the `args` or `kwargs` arguments. The accessor will be resolved using the row's record before `reverse` is called.

RelatedLinkColumn

```
class django_tables2.columns.RelatedLinkColumn (viewname=None, urlconf=None,
                                                args=None, kwargs=None, current_app=None, attrs=None, **extra)
```

Render a link to a related object using related object's `get_absolute_url`, same parameters as `~.LinkColumn`

TemplateColumn

```
class django_tables2.columns.TemplateColumn (template_code=None, template_name=None, **extra)
```

A subclass of `Column` that renders some template code to use as the cell value.

Parameters

- **template_code** (*str*) – template code to render
- **template_name** (*str*) – name of the template to render

A `Template` object is created from the `template_code` or `template_name` and rendered with a context containing:

- *record* – data record for the current row
- *value* – value from *record* that corresponds to the current column
- *default* – appropriate default value to use as fallback

Example:

```
class ExampleTable (tables.Table):
    foo = tables.TemplateColumn('{{ record.bar }}')
    # contents of `myapp/bar_column.html` is `{{ value }}`
    bar = tables.TemplateColumn(template_name='myapp/name2_column.html')
```

Both columns will have the same output.

Important: In order to use template tags or filters that require a `RequestContext`, the table **must** be rendered via `{% render_table %}`.

URLColumn

```
class django_tables2.columns.URLColumn (attrs=None, text=None, *args, **kwargs)
```

Renders URL values as hyperlinks.

Parameters

- **text** (*str* or *callable*) – Either static text, or a callable. If set, this will be used to render the text inside link instead of value (default)
- **attrs** (*dict*) – Additional attributes for the `<a>` tag

Example:

```
>>> class CompaniesTable (tables.Table):
...     www = tables.URLColumn()
...
>>> table = CompaniesTable([{'www': 'http://google.com'}])
```



```
>>> table.rows[0].get_cell('www')
'<a href="http://google.com">http://google.com</a>'
```

SingleTableMixin

class django_tables2.views.**SingleTableMixin**

Adds a Table object to the context. Typically used with TemplateResponseMixin.

table_class

subclass of *Table*

table_data

data used to populate the table, any compatible data source.

context_table_name

str

name of the table's template variable (default: 'table')

table_pagination

dict

controls table pagination. If a *dict*, passed as the *paginate* keyword argument to *RequestConfig*. As such, any Truthy value enables pagination.

This mixin plays nice with the Django's `MultipleObjectMixin` by using `get_queryset` as a fallback for the table data source.

get_context_data (***kwargs*)

Overriden version of *TemplateResponseMixin* to inject the table into the template's context.

get_table (***kwargs*)

Return a table object to use. The table has automatic support for sorting and pagination.

get_table_data ()

Return the table data that should be used to populate the rows.

MultiTableMixin

class django_tables2.views.**MultiTableMixin**

Adds a Table object to the context. Typically used with TemplateResponseMixin.

the *tables* attribute must be either a list of *Table* instances or classes extended from *Table* which are not already instantiated. In that case, *tables_data* must be defined, having an entry containing the data for each table in *tables*.

tables

list of *Table* instances or list of *Table* child objects.

tables_data

if defined, *tables* is assumed to be a list of table classes which will be instantiated with the corresponding item from this list of *TableData* instances.

table_prefix

str

Prefix to be used for each table. The string must contain one instance of `{ }`, which will be replaced by an integer different for each table in the view. Default is `'table_{ }'`.

context_table_name

str

name of the table's template variable (default: 'tables')

New in version 1.2.3.

See [Internal APIs](#) for internal classes.

1.3.4 Internal APIs

The items documented here are internal and subject to change.

BoundColumns

class `django_tables2.columns.BoundColumns` (*table*)

Container for spawning *BoundColumn* objects.

This is bound to a table and provides its `Table.columns` property. It provides access to those columns in different ways (iterator, item-based, filtered and unfiltered etc), stuff that would not be possible with a simple iterator in the table class.

A `BoundColumns` object is a container for holding `BoundColumn` objects. It provides methods that make accessing columns easier than if they were stored in a `list` or `dict`. `Columns` has a similar API to a `dict` (it actually uses a `OrderedDict` internally).

At the moment you'll only come across this class when you access a `Table.columns` property.

Parameters `table` (*Table*) – the table containing the columns

__contains__ (*item*)

Check if a column is contained within a `Columns` object.

item can either be a `BoundColumn` object, or the name of a column.

__getitem__ (*index*)

Retrieve a specific `BoundColumn` object.

index can either be 0-indexed or the name of a column

```
columns['speed']    # returns a bound column with name 'speed'
columns[0]          # returns the first column
```

__iter__ ()

Convenience API, alias of *itervisible*.

__len__ ()

Return how many `BoundColumn` objects are contained (and visible).

__weakref__

list of weak references to the object (if defined)

iterall ()

Return an iterator that exposes all *BoundColumn* objects, regardless of visibility or sortability.

iteritems ()

Return an iterator of (*name*, *column*) pairs (where *column* is a `BoundColumn`).

This method is the mechanism for retrieving columns that takes into consideration all of the ordering and filtering modifiers that a table supports (e.g. *exclude* and *sequence*).

iterorderable()

Same as `BoundColumns.all` but only returns orderable columns.

This is useful in templates, where iterating over the full set and checking `{% if column.orderable %}` can be problematic in conjunction with e.g. `{{ forloop.last }}` (the last column might not be the actual last that is rendered).

itervisible()

Same as *iterorderable* but only returns visible *BoundColumn* objects.

This is geared towards table rendering.

BoundColumn

class `django_tables2.columns.BoundColumn(table, column, name)`

A *run-time* version of *Column*. The difference between *BoundColumn* and *Column*, is that *BoundColumn* objects include the relationship between a *Column* and a *Table*. In practice, this means that a *BoundColumn* knows the “variable name” given to the *Column* when it was declared on the *Table*.

For convenience, all *Column* properties are available from this class.

Parameters

- **table** (*Table* object) – the table in which this column exists
- **column** (*Column* object) – the type of column
- **name** (*string object*) – the variable name of the column used to when defining the *Table*. In this example the name is `age`:

```
class SimpleTable(tables.Table):
    age = tables.Column()
```

__weakref__

list of weak references to the object (if defined)

accessor

Returns the string used to access data for this column out of the data source.

attrs

Proxy to `Column.attrs` but injects some values of our own.

`th` and `td` are guaranteed to be defined (irrespective of what’s actually defined in the column `attrs`. This makes writing templates easier.

default

Returns the default value for this column.

header

The value that should be used in the header cell for this column.

localize

Returns `True`, `False` or `None` as described in `Column.localize`

order_by

Returns an *OrderByTuple* of appropriately prefixed data source keys used to sort this column.

See *order_by_alias* for details.

order_by_alias

Returns an *OrderBy* describing the current state of ordering for this column.

The following attempts to explain the difference between `order_by` and `order_by_alias`.

`order_by_alias` returns and `OrderBy` instance that's based on the *name* of the column, rather than the keys used to order the table data. Understanding the difference is essential.

Having an alias *and* a keys version is necessary because an N-tuple (of data source keys) can be used by the column to order the data, and it's ambiguous when mapping from N-tuple to column (since multiple columns could use the same N-tuple).

The solution is to use order by *aliases* (which are really just prefixed column names) that describe the ordering *state* of the column, rather than the specific keys in the data source should be ordered.

e.g.:

```
>>> class SimpleTable(tables.Table):
...     name = tables.Column(order_by=("firstname", "last_name"))
...
>>> table = SimpleTable([], order_by=("-name", ))
>>> table.columns["name"].order_by_alias
"-name"
>>> table.columns["name"].order_by
("-first_name", "-last_name")
```

The `OrderBy` returned has been patched to include an extra attribute `next`, which returns a version of the alias that would be transitioned to if the user toggles sorting on this column, e.g.:

```
not sorted -> ascending
ascending  -> descending
descending -> ascending
```

This is useful otherwise in templates you'd need something like:

```
{% if column.is_ordered %} {% querystring table.prefixed_order_by_field=column.order_by_alias.opposite %} {% else %} {% querystring table.prefixed_order_by_field=column.order_by_alias %} {% endif %}
```

orderable

Return a `bool` depending on whether this column supports ordering.

verbose_name

Return the verbose name for this column, or fallback to the titlised column name.

If the table is using queryset data, then use the corresponding model field's `verbose_name`. If it's traversing a relationship, then get the last field in the accessor (i.e. stop when the relationship turns from ORM relationships to object attributes [e.g. `person.upper` should stop at `person`]).

visible

Returns a `bool` depending on whether this column is visible.

BoundRows

class `django_tables2.rows.BoundRows` (*data*, *table*)

Container for spawning `BoundRow` objects.

Parameters

- **data** – iterable of records
- **table** – the `Table` in which the rows exist

This is used for `Table.rows`.

`__getitem__` (*key*)

Slicing returns a new *BoundRows* instance, indexing returns a single *BoundRow* instance.

`__weakref__`

list of weak references to the object (if defined)

BoundRow

class `django_tables2.rows.BoundRow` (*record*, *table*)

Represents a *specific* row in a table.

BoundRow objects are a container that make it easy to access the final ‘rendered’ values for cells in a row. You can simply iterate over a *BoundRow* object and it will take care to return values rendered using the correct method (e.g. *Table.render_foo methods*)

To access the rendered value of each cell in a row, just iterate over it:

```
>>> import django_tables2 as tables
>>> class SimpleTable(tables.Table):
...     a = tables.Column()
...     b = tables.CheckBoxColumn(attrs={'name': 'my_chkbox'})
...
>>> table = SimpleTable([{'a': 1, 'b': 2}])
>>> row = table.rows[0] # we only have one row, so let's use it
>>> for cell in row:
...     print(cell)
...
1
<input type="checkbox" name="my_chkbox" value="2" />
```

Alternatively you can use `row.get_cell()` to retrieve a specific cell:

```
>>> row.get_cell(0)
1
>>> row.get_cell(1)
u'<input type="checkbox" name="my_chkbox" value="2" />'
>>> row.get_cell(2)
...
IndexError: list index out of range
```

Finally you can also use the column names to retrieve a specific cell:

```
>>> row.get_cell('a')
1
>>> row.get_cell('b')
u'<input type="checkbox" name="my_chkbox" value="2" />'
>>> row.get_cell('c')
...
KeyError: 'c'
```

Parameters

- **table** – The *Table* in which this row exists.
- **record** – a single record from the *table data* that is used to populate the row. A record could be a *Model* object, a *dict*, or something else.

`__contains__` (*item*)

Check by both row object and column name.

__iter__()
Iterate over the rendered values for cells in the row.

Under the hood this method just makes a call to `BoundRow.__getitem__` for each cell.

__weakref__
list of weak references to the object (if defined)

_call_render (*bound_column*, *value=None*)
Call the column's render method with appropriate kwargs

attrs
Return the attributes for a certain row.

get_cell (*name*)
Returns the final rendered value for a cell in the row, given the name of a column.

items ()
Returns iterator yielding (*bound_column*, *cell*) pairs.

cell is `row[name]` – the rendered unicode value that should be rendered within `<td>`.

record
The data record from the data source which is used to populate this row with data.

table
The associated [Table](#) object.

TableData

class `django_tables2.tables.TableData` (*data*, *table*)

Exposes a consistent API for [table data](#).

Parameters

- **data** (`QuerySet` or `list` of `dict`) – iterable containing data for each row
- **table** ([Table](#)) –

__getitem__ (*key*)
Slicing returns a new [TableData](#) instance, indexing returns a single record.

__weakref__
list of weak references to the object (if defined)

order_by (*aliases*)
Order the data based on order by aliases (prefixed column names) in the table.

Parameters **aliases** ([OrderByTuple](#)) – optionally prefixed names of columns ('-' indicates descending order) in order of significance with regard to data ordering.

ordering
Returns the list of order by aliases that are enforcing ordering on the data.

If the data is unordered, an empty sequence is returned. If the ordering can not be determined, `None` is returned.

This works by inspecting the actual underlying data. As such it's only supported for querysets.

verbose_name
The full (singular) name for the data.

Queryset data has its model's `verbose_name` honored. List data is checked for a `verbose_name` attribute, and falls back to using `'item'`.

verbose_name_plural

The full (plural) name of the data.

This uses the same approach as `TableData.verbose_name`.

utils**class** `django_tables2.utils.Sequence`

Represents a column sequence, e.g. ('first_name', '...', 'last_name')

This is used to represent `Table.Meta.sequence` or the `Table` constructors's *sequence* keyword argument.

The sequence must be a list of column names and is used to specify the order of the columns on a table. Optionally a '...' item can be inserted, which is treated as a *catch-all* for column names that aren't explicitly specified.

__weakref__

list of weak references to the object (if defined)

expand (*columns*)

Expands the '...' item in the sequence into the appropriate column names that should be placed there.

Raises `ValueError` if the sequence is invalid for the columns.

class `django_tables2.utils.OrderBy`

A single item in an `OrderByTuple` object. This class is essentially just a `str` with some extra properties.

bare

Returns the bare form.

Return type `OrderBy`

The *bare form* is the non-prefixed form. Typically the bare form is just the ascending form.

Example: age is the bare form of `-age`

for_queryset ()

Returns the current instance usable in Django QuerySet's `order_by` arguments.

is_ascending

Returns `True` if this object induces *ascending* ordering.

is_descending

Returns `True` if this object induces *descending* ordering.

opposite

Provides the opposite of the current sorting direction.

Returns object with an opposite sort influence.

Return type `OrderBy`

Example:

```
>>> order_by = OrderBy('name')
>>> order_by.opposite
'-name'
```

class `django_tables2.utils.OrderByTuple`

Stores ordering as (as `OrderBy` objects). The `order_by` property is always converted to an `OrderByTuple` object.

This class is essentially just a `tuple` with some useful extras.

Example:

```
>>> x = OrderByTuple(('name', '-age'))
>>> x['age']
'-age'
>>> x['age'].is_descending
True
>>> x['age'].opposite
'age'
```

__contains__ (*name*)

Determine if a column has an influence on ordering.

Example:

```
>>> x = OrderByTuple(('name', ))
>>> 'name' in x
True
>>> '-name' in x
True
```

Parameters *name* (*str*) – The name of a column. (optionally prefixed)

Returns *True* if the column with *name* influences the ordering.

Return type *bool*

__getitem__ (*index*)

Allows an *OrderBy* object to be extracted via named or integer based indexing.

When using named based indexing, it's fine to used a prefixed named:

```
>>> x = OrderByTuple(('name', '-age'))
>>> x[0]
'name'
>>> x['age']
'-age'
>>> x['-age']
'-age'
```

Parameters *index* (*int*) – Index to query the ordering for.

Returns for the ordering at the index.

Return type *OrderBy*

get (*key*, *fallback*)

Identical to `__getitem__`, but supports fallback value.

opposite

Return version with each *OrderBy* prefix toggled:

```
>>> order_by = OrderByTuple(('name', '-age'))
>>> order_by.opposite
('-name', 'age')
```

class `django_tables2.utils.Accessor`

A string describing a path from one object to another via attribute/index accesses. For convenience, the class has an alias *A* to allow for more concise code.

Relations are separated by a `.` character.

get_field(*model*)

Return the django model field for model in context, following relations

penultimate(*context*, *quiet=True*)

Split the accessor on the right-most dot '.', return a tuple with:

- the resolved left part.
- the remainder

Example:

```
>>> Accessor('a.b.c').penultimate({'a': {'a': 1, 'b': {'c': 2, 'd': 4}}})
({'c': 2, 'd': 4}, 'c')
```

resolve(*context*, *safe=True*, *quiet=False*)

Return an object described by the accessor by traversing the attributes of *context*.

Lookups are attempted in the following order:

- dictionary (e.g. `obj[related]`)
- attribute (e.g. `obj.related`)
- list-index lookup (e.g. `obj[int(related)]`)

Callable objects are called, and their result is used, before proceeding with the resolving.

Example:

```
>>> x = Accessor('__len__')
>>> x.resolve('brad')
4
>>> x = Accessor('0.upper')
>>> x.resolve('brad')
'B'
```

Parameters

- **context** (*object*) – The root/first object to traverse.
- **safe** (*bool*) – Don't call anything with `alters_data = True`
- **quiet** (*bool*) – Smother all exceptions and instead return `None`

Returns target object

Raises

- `TypeError`, `AttributeError`, `KeyError`, `ValueError`
- (unless `quiet == True`)

class `django_tables2.utils.AttributeDict`

A wrapper around `dict` that knows how to render itself as HTML style tag attributes.

The returned string is marked safe, so it can be used safely in a template. See [as_html](#) for a usage example.

__weakref__

list of weak references to the object (if defined)

as_html()

Render to HTML tag attributes.

Example:

```
>>> from django_tables2.utils import AttributeDict
>>> attrs = AttributeDict({'class': 'mytable', 'id': 'someid'})
>>> attrs.as_html()
'class="mytable" id="someid"'
```

Return type `SafeUnicode` object

`django_tables2.utils.signature(fn)`

Returns Returns a (arguments, kwarg_name)-tuple: - the arguments (positional or keyword) - the name of the `**` kwarg catch all.

Return type `tuple`

The self-argument for methods is always removed.

`django_tables2.utils.call_with_appropriate(fn, kwargs)`

Calls the function `fn` with the keyword arguments from `kwargs` it expects

If the `kwargs` argument is defined, pass all arguments, else provide exactly the arguments wanted.

`django_tables2.utils.computed_values(d, *args, **kwargs)`

Returns a new `dict` that has callable values replaced with the return values.

Example:

```
>>> compute_values({'foo': lambda: 'bar'})
{'foo': 'bar'}
```

Arbitrarily deep structures are supported. The logic is as follows:

- 1.If the value is callable, call it and make that the new value.
- 2.If the value is an instance of `dict`, use `ComputableDict` to compute its keys.

Example:

```
>>> def parents():
...     return {
...         'father': lambda: 'Foo',
...         'mother': 'Bar'
...     }
...
>>> a = {
...     'name': 'Brad',
...     'parents': parents
... }
...
>>> computed_values(a)
{'name': 'Brad', 'parents': {'father': 'Foo', 'mother': 'Bar'}}
```

Parameters

- `d(dict)` – The original dictionary.
- `args` – any extra positional arguments will be passed to the callables
- `kwargs` – any extra keyword arguments will be passed to the callables.

Returns with callable values replaced.

Return type `dict`

1.4 FAQ

Some frequently requested questions/examples. All examples assume you import django-tables2 like this:

```
import django_tables2 as tables
```

1.4.1 How should I fix error messages about the request context processor?

The error message looks something like this:

```
Tag {% querystring %} requires django.template.context_processors.request to be
in the template configuration in settings.TEMPLATES[OPTIONS].context_processors)
in order for the included template tags to function correctly.
```

which should be pretty clear, but here is an example template configuration anyway:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': ['templates'],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.contrib.auth.context_processors.auth',
                'django.template.context_processors.request',
                'django.template.context_processors.static',
            ],
        },
    },
]
```

1.4.2 How to create a row counter?

You can use `itertools.counter` to add row count to a table. Note that in a paginated table, every page's counter will start at zero.

Use a `render_counter()`-method:

```
import itertools

class CountryTable(tables.Table):
    counter = tables.Column(empty_values=(), orderable=False)

    def render_counter(self):
        self.row_counter = getattr(self, 'row_counter', itertools.count())
        return next(self.row_counter)
```

Or create a specialized column:

```
import itertools

class CounterColumn(tables.Column):
    def __init__(self, *args, **kwargs):
        self.counter = itertools.count()
        kwargs.update({
            'empty_values': (),
```

```

        'orderable': False
    })
    super(CounterColumn, self).__init__(*args, **kwargs)

    def render(self):
        return next(self.counter)

```

1.4.3 How to add a footer containing a column total?

Using the footer-argument to *Column*:

```

class CountryTable(tables.Table):
    population = tables.Column(
        footer=lambda table: sum(x['population'] for x in table.data)
    )

```

Or by creating a custom column:

```

class SummingColumn(tables.Column):
    def render_footer(self, bound_column, table):
        return sum(bound_column.accessor.resolve(row) for row in table.data)

class Table(tables.Table):
    name = tables.Column(footer='Total:')
    population = SummingColumn()

```

Documentation: [Adding column footers](#)

Note: You table template must include a block rendering the table footer!

1.5 Upgrading and change log

Recent versions of django-tables2 have a corresponding git tag for each version released to pypi.

1.5.1 Change log

1.2.3 (2016-07-05)

- Accept `text` parameter in `FileColumn`, analogous to `LinkColumn` (#343 by @graup)
- Fix `TemplateColumn RemovedInDjango110Warning` fixes #346.
- Use field name in `RelatedColumnLink` (#350, fixes #347)

v1.2.2 (2016-06-04)

- Allow use of custom class names for ordered columns through `attrs`. (#329 by @theTarkus)
- Column ordering queryset passthrough (#330 by @theTarkus)
- Cleanup/restructuring of [documentation](#), (#325)

- Fixed an issue where explicitly defined column options were not preserved over inheritance (#339, issue #337)
- Fixed an issue where `exclude` in combination with `sequence` raised a `KeyError` (#341, issue #205)

v1.2.1 (2016-05-09)

- table footers (#323)
- Non-field based `LinkColumn` only renders default value if lookup fails. (#322)
- Accept `text` parameter in `BaseLinkColumn`-based columns. (#322)
- Pass the table instance into `SingleTableMixin`'s `get_table_pagination` (#320 by @georgema1982, fixes #319)
- Check if the view has `paginate_by` before trying to access it. (fixes #326)

v1.2.0 (2016-05-02)

- Allow custom attributes for rows (fixes #47)

v1.1.8 (2016-05-02)

- Pass the table instance into `SingleTableMixin.get_table_pagination()` (#320 by @georgema1982)[<https://github.com/georgema1982>]
- Ability to change the body of the `<a>`-tag, by passing `text` kwarg to the columns inheriting from `BaseLinkColumn` (#318 by @desecho, #322)
- Non-field based `LinkColumn` only renders default value if lookup fails and text is not set. (#322, fixes #257)

v1.1.7 (2016-04-26)

- Added Italian translation (#315 by @paolodina)
- Added Dutch translation.
- Fixed `{% blocktrans %}` template whitespace issues
- Fixed errors when using a column named `items` (#316)
- Obey `paginate_by` (from `MultipleObjectMixin`) if no later pagination is defined (#242)

v1.1.6 (2016-04-02)

- Correct error message about request context processors for current Django (#314)
- Skipped 1.1.5 due to an error while creating the tag.

v1.1.4 (2016-03-22)

- Fix broken `setup.py` if Django is not installed before `django-tables2` (fixes #312)

v1.1.3 (2016-03-21)

- Drop support for Django 1.7
- Add argument to `CheckBoxColumn` to render it as checked (original PR: #208)

v1.1.2 (2016-02-16)

- Fix `BooleanColumn` with choices set will always render as if `True` (#301)
- Fix a bug with `TemplateColumn` while using cached template loader (#75)

v1.1.1 (2016-01-26)

- Allow `Meta.fields` to be a list as well as a tuple (#250)
- Call `template.render` with a dict in Django \geq 1.8. (#298)
- Added `RelatedLinkColumn()` to render links to related objects (#297)
- Remove default value from request param to `table.as_html()`

v1.1.0 (2016-01-19)

- Add tests for `TimeColumn`
- Remove `sortable` argument for `Table` and `Column` constructors and its associated methods. Deprecated since 2012.
- Remove deprecated aliases for `attrs` in `CheckboxColumn`.
- Remove deprecated `OrderByTuple cmp` method (deprecated since 2013).
- Add bootstrap template and (#293, fixes #141, #285)
- Fix different html for tables with and without pagination (#293, fixes #149, #285)
- Remove `{% nospaceless %}` template tag and remove wrapping template in `{% spaceless %}` **Possible breaking change**, if you use custom templates.

v1.0.7 (2016-01-03)

- Explicitly check if `column.verbose_name` is not `None` to support empty column headers (fixes #280)
- Cleanup the example project to make it work with modern Django versions.
- Do not sort queryset when `orderable=False` (#204 by @bmihelac)
- `show_header` attribute on `Table` allows disabling the header (#175 by @kviktor)
- `LinkColumn` now tries to call `get_absolute_url` on a record if no `viewname` is provided (#283, fixes #231).
- Add `request` argument to `Table.as_html()` to allow passing correct request objects instead of poorly generated ones #282
- Add coverage reporting to build #282
- Drop support for python 3.2 (because of coverage), support ends feb 2016 #282
- move `build_request` from `django_table2.utils` to `tests.utils` and amend tests #282

v1.0.6 (2015-12-29)

- Support for custom text value in LinkColumn (#277 by @toudi)
- Refactor LinkColumn.render_link() to not escape twice #279
- Removed Attrs (wrapper for dict), deprecated on 2012-09-18
- Convert README.md to rst in setup.py to make PyPI look nice (fixes #97)

v1.0.5 (2015-12-17)

- First version released by new maintainer @jieter
- Dropped support for django 1.5 and 1.6, add python 3.5 with django 1.8 and 1.9 to the build matrix (#273)
- Prevent SingleTableView from calling get_queryset twice. (fixes #155)
- Don't call managers when resolving accessors. (#214 by @mbertheau, fixes #211)

v1.0.4 (2015-05-09)

- Fix bug in retrieving field.verbose_name under Django 1.8.

v1.0.3

- Remove setup.cfg as PyPI doesn't actually support it, instead it's a distutils2 thing that's been discontinued.

v1.0.2

- Add setup.cfg to declare README.md for PyPI.

v1.0.1

- Convert README to markdown so it's formatted nicely on PyPI.

v1.0.0

- Travis CI builds pass.
- Added Python 3.4 support.
- Added Django 1.7 and Django 1.8 support.
- Convert tests to using py.test.

v0.16.0

- Django 1.8 fixes
- BoundColumn.verbose_name now titlises only if no verbose_name was given. verbose_name is used verbatim.
- Add max_length attribute to person CharField

- Add Swedish translation
- Update docs presentation on readthedocs

v0.15.0

- Add UK, Russian, Spanish, Portuguese, and Polish translations
- Add support for computed table `attrs`.

v0.14.0

- `querystring` and `seturlparam` template tags now require the request to be in the context (backwards incompatible) – #127
- Add Travis CI support
- Add support for Django 1.5
- Add L10N control for columns #120 (ignored in < Django 1.3)
- Drop Python 2.6.4 support in favour of Python 3.2 support
- Non-queryset data ordering is different between Python 3 and 2. When comparing different types, their truth values are now compared before falling back to string representations of their type.

v0.13.0

- Add `FileColumn`.

v0.12.1

- When resolving an accessor, *all* exceptions are smothered into `None`.

v0.12.0

- Improve performance by removing unnecessary queries
- Simplified pagination:
 - `Table.page` is an instance attribute (no longer `@property`)
 - Exceptions raised by paginators (e.g. `EmptyPage`) are no longer smothered by `Table.page`
 - Pagination exceptions are raised by `Table.paginate`
 - `RequestConfig` can handles pagination errors silently, can be disabled by including `silent=False` in the `paginate` argument value
- Add `DateTimeColumn` and `DateColumn` to handle formatting `datetime` and `timezones`.
- Add `BooleanColumn` to handle `bool` values
- `render_table` can now build and render a table for a queryset, rather than needing to be passed a table instance
- Table columns created automatically from a model now use specialised columns

- `Column.render` is now skipped if the value is considered *empty*, the default value is used instead. Empty values are specified via `Column.empty_values`, by default is `(None, '')` (backward incompatible)
- Default values can now be specified on table instances or `Table.Meta`
- Accessor's now honor `alters_data` during resolving. Fixes issue that would delete all your data when a column had an accessor of `delete`
- Add `default` and `value` to context of `TemplateColumn`
- Add cardinality indication to the pagination area of a table
- `Attrs` is deprecated, use `dict` instead

v0.11.0

- Add `URLColumn` to render URLs in a data source into hyperlinks
- Add `EmailColumn` to render email addresses into hyperlinks
- `TemplateColumn` can now Django's template loaders to render from a file

v0.10.4

- Fix more bugs on Python 2.6.4, all tests now pass.

v0.10.3

- Fix issues for Python 2.6.4 – thanks Steve Sapovits & brianmay
- Reduce Django 1.3 dependency to `Table.as_html` – thanks brianmay

v0.10.2

- Fix MANIFEST.in to include example templates, thanks TWAC.
- Upgrade django-attest to fix problem with tests on Django 1.3.1

v0.10.1

- Fixed support for Django 1.4's paginator (thanks koledennix)
- Some juggling of internal implementation. `TableData` now supports slicing and returns new `TableData` instances. `BoundRows` now takes a single argument `data` (a `TableData` instance).
- Add support for `get_pagination` on `SingleTableMixin`.
- `SingleTableMixin` and `SingleTableView` are now importable directly from `django_tables2`.

v0.10.0

- Renamed `BoundColumn.order_by` to `order_by_alias` and never returns `None` (**Backwards incompatible**). Templates are affected if they use something like:

```
{% querystring table.prefixed_order_by_field=column.order_by.opposite|default:column.name %}
```

Which should be rewritten as:

```
{% querystring table.prefixed_order_by_field=column.order_by_alias.next %}
```

- Added `next` shortcut to `OrderBy` returned from `BoundColumn.order_by_alias`
- Added `OrderByTuple.get()`
- Deprecated `BoundColumn.sortable`, `Column.sortable`, `Table.sortable`, `sortable` CSS class, `BoundColumns.itersortable`, `BoundColumns.sortable`; use `orderable` instead of `sortable`.
- Added `BoundColumn.is_ordered`
- Introduced concept of an `order by alias`, see glossary in the docs for details.

v0.9.6

- Fix bug that caused an ordered column's `th` to have no HTML attributes.

v0.9.5

- Updated example project to add `colspan` on footer cell so table border renders correctly in Webkit.
- Fix regression that caused 'sortable' class on .
- `Table.init` no longer *always* calls `.order_by()` on querysets, fixes #55. This does introduce a slight backwards incompatibility. `Table.order_by` now has the possibility of returning `None`, previously it would *always* return an `OrderByTuple`.
- `DeclarativeColumnsMetaclass.new` now uses `super()`
- Testing now requires `pylint` and `Attest >=0.5.3`

v0.9.4

- Fix regression that caused column `verbose_name` values that were marked as safe to be escaped. Now any `verbose_name` values that are instances of `SafeData` are used unmodified.

v0.9.3

- Fix regression in `SingleTableMixin`.
- Remove stray `print` statement.

v0.9.2

- `SingleTableView` now uses `RequestConfig`. This fixes issues with `order_by_field`, `page_field`, and `per_page_field` not being honored.
- Add `Table.Meta.per_page` and change `Table.paginate` to use it as default.
- Add `title` template filter. It differs from Django's built-in `title` filter because it operates on an individual word basis and leaves words containing capitals untouched. **Warning:** use `{% load ... from ... %}` to avoid inadvertently replacing Django's builtin `title` template filter.

- `BoundColumn.verbose_name` no longer does `capfirst`, titlising is now the responsibility of `Column.header`.
- `BoundColumn.__unicode__` now uses `BoundColumn.header` rather than `BoundColumn.verbose_name`.

v0.9.1

- Fix version in `setup.py` (doh)

v0.9.0

- Add support for column attributes (see `Attrs`)
- Add `BoundRows.items()` to yield `(bound_column, cell)` pairs
- Tried to make docs more concise. Much stronger promotion of using `RequestConfig` and `{% querystring %}`

v0.8.4

- Removed random ‘print’ statements.
- Tweaked ‘paleblue’ theme css to be more flexible:
 - removed `whitespace: no-wrap`
 - header background image to support more than 2 rows of text

v0.8.3

- Fixed stupid import mistake. Tests didn’t pick it up due to them ignoring `ImportError`.

v0.8.2

- `SingleTableView` now inherits from `ListView` which enables automatic `foo_list.html` template name resolution (thanks dramon for reporting)
- `render_table` template tag no suppresses exceptions when `DEBUG=True`

v0.8.1

- Fixed bug in `render_table` when giving it a template (issue #41)

v0.8.0

- Added translation support in the default template via `{% trans %}`
- Removed `basic_table.html`, `Table.as_html()` now renders `table.html` but will clobber the `querystring` of the current request. Use the `render_table` template tag instead
- `render_table` now supports an optional second argument – the template to use when rendering the table
- `Table` now supports declaring which template to use when rendering to HTML
- Django `>=1.3` is now required

- Added support for using django-haystack's `SearchQuerySet` as a data source
- The default template `table.html` now includes block tags to make it easy to extend to change small pieces
- Fixed table template parsing problems being hidden due to a subsequent exception being raised
- `Http404` exceptions are no longer raised during a call to `Table.paginate()`, instead it now occurs when `Table.page` is accessed
- Fixed bug where a table couldn't be rendered more than once if it was paginated
- Accessing `Table.page` now returns a new page every time, rather than reusing a single object

v0.7.8

- Tables now support using both `sequence` and `exclude` (issue #32).
- `Sequence` class moved to `django_tables2/utils.py`.
- Table instances now support modification to the `exclude` property.
- Removed `BoundColumns._spawn_columns`.
- `Table.data`, `Table.rows`, and `Table.columns` are now attributes rather than properties.

1.5.2 Upgrading from django-tables Version 1

- Change your `INSTALLED_APPS` entry from `'django_tables.app'` to `'django_tables2'`.
- Change all your import references from `django_tables` to `django_tables2`.
- Replace all references to the old `MemoryTable` and `ModelTable` classes with simply `Table`.
- In your templates, load the `django_tables2` template library; `{% load django_tables2 %}` instead of `{% load tables %}`.
- A table object is no longer iterable; rather than `for row in table`, instead you now do explicitly: `for row in table.rows`.
- If you were using `row.data` to access a row's underlying data, replace it with `row.record` instead.
- When declaring columns, replace the use of:

```
name_in_dataset = tables.Column(name='wanted_column_name')
```

with:

```
wanted_column_name = tables.Column(accessor='name_in_dataset')
```

- When declaring columns, replace the use of:

```
column_to_override = tables.Column(name='wanted_column_name', data='name_in_dataset')
```

with:

```
wanted_column_name = tables.Column(accessor='name_in_dataset')
```

and exclude `column_to_override` via the table meta data.

- When generating the link to order the column, instead of:

```
{% set_url_param sort=column.name_toggled %}
```

use:

```
{% querystring table.order_by_field=column.order_by_alias.next %}
```

- Replace:

```
{{ column.is_ordered_reverse }} and {{ column.is_ordered_straight }}
```

with:

```
{{ column.order_by.is_descending }} and {{ column.order_by.is_ascending }}
```

1.6 Glossary

accessor Refers to an *Accessor* object

column name The name given to a column. In the follow example, the *column name* is age.

```
class SimpleTable(tables.Table):
    age = tables.Column()
```

empty value An empty value is synonymous with “no value”. Columns have an `empty_values` attribute that contains values that are considered empty. It’s a way to declare which values from the database correspond to *null/blank/missing* etc.

order by alias A prefixed column name that describes how a column should impact the order of data within the table. This allows the implementation of how a column affects ordering to be abstracted, which is useful (e.g. in querystrings).

```
class ExampleTable(tables.Table):
    name = tables.Column(order_by=('first_name', 'last_name'))
```

In this example `-name` and `name` are valid order by aliases. In a querystring you might then have `?order=-name`.

table The traditional concept of a table. i.e. a grid of rows and columns containing data.

view A Django view.

record A single Python object used as the data for a single row.

render The act of serializing a *Table* into HTML.

template A Django template.

table data An iterable of *records* that *Table* uses to populate its rows.

Symbols

[__contains__\(\)](#) (django_tables2.columns.BoundColumns method), 30
[__contains__\(\)](#) (django_tables2.rows.BoundRow method), 33
[__contains__\(\)](#) (django_tables2.utils.OrderByTuple method), 36
[__getitem__\(\)](#) (django_tables2.columns.BoundColumns method), 30
[__getitem__\(\)](#) (django_tables2.rows.BoundRows method), 32
[__getitem__\(\)](#) (django_tables2.tables.TableData method), 34
[__getitem__\(\)](#) (django_tables2.utils.OrderByTuple method), 36
[__iter__\(\)](#) (django_tables2.columns.BoundColumns method), 30
[__iter__\(\)](#) (django_tables2.rows.BoundRow method), 33
[__len__\(\)](#) (django_tables2.columns.BoundColumns method), 30
[__weakref__](#) (django_tables2.columns.BoundColumn attribute), 31
[__weakref__](#) (django_tables2.columns.BoundColumns attribute), 30
[__weakref__](#) (django_tables2.rows.BoundRow attribute), 34
[__weakref__](#) (django_tables2.rows.BoundRows attribute), 33
[__weakref__](#) (django_tables2.tables.TableData attribute), 34
[__weakref__](#) (django_tables2.utils.Sequence attribute), 35
[_call_render\(\)](#) (django_tables2.rows.BoundRow method), 34

A

accessor, 49
 Accessor (class in django_tables2.utils), 18
 accessor (django_tables2.columns.BoundColumn attribute), 31

[as_html\(\)](#) (django_tables2.tables.Table method), 20
[attrs](#) (django_tables2.columns.BoundColumn attribute), 31
[attrs](#) (django_tables2.rows.BoundRow attribute), 34

B

bare (django_tables2.utils.OrderBy attribute), 35
 BooleanColumn (class in django_tables2.columns), 24
 BoundColumn (class in django_tables2.columns), 31
 BoundColumns (class in django_tables2.columns), 30
 BoundRow (class in django_tables2.rows), 33
 BoundRows (class in django_tables2.rows), 32

C

CheckBoxColumn (class in django_tables2.columns), 24
 Column (class in django_tables2.columns), 23
 column name, 49
[compose_url\(\)](#) (django_tables2.columns.LinkColumn method), 27
[context_table_name](#) (MultiTableMixin attribute), 29
[context_table_name](#) (SingleTableMixin attribute), 29

D

DateColumn (class in django_tables2.columns), 25
 DateTimeColumn (class in django_tables2.columns), 25
 default (django_tables2.columns.BoundColumn attribute), 31

E

EmailColumn (class in django_tables2.columns), 26
 empty value, 49
[expand\(\)](#) (django_tables2.utils.Sequence method), 35

F

FileColumn (class in django_tables2.columns), 26
[for_queryset\(\)](#) (django_tables2.utils.OrderBy method), 35

G

[get\(\)](#) (django_tables2.utils.OrderByTuple method), 36

get_cell() (django_tables2.rows.BoundRow method), 34
 get_context_data() (django_tables2.views.SingleTableMixin method), 29
 get_table() (django_tables2.views.SingleTableMixin method), 29
 get_table_data() (django_tables2.views.SingleTableMixin method), 29

H

header (django_tables2.columns.BoundColumn attribute), 31

I

is_ascending (django_tables2.utils.OrderBy attribute), 35
 is_checked() (django_tables2.columns.CheckBoxColumn method), 25
 is_descending (django_tables2.utils.OrderBy attribute), 35
 items() (django_tables2.rows.BoundRow method), 34
 iterall() (django_tables2.columns.BoundColumns method), 30
 iteritems() (django_tables2.columns.BoundColumns method), 30
 iterorderable() (django_tables2.columns.BoundColumns method), 30
 iterable() (django_tables2.columns.BoundColumns method), 31

L

LinkColumn (class in django_tables2.columns), 26
 localize (django_tables2.columns.BoundColumn attribute), 31

M

MultiTableMixin (class in django_tables2.views), 29

O

opposite (django_tables2.utils.OrderBy attribute), 35
 opposite (django_tables2.utils.OrderByTuple attribute), 36
 order by alias, 49
 order_by (django_tables2.columns.BoundColumn attribute), 31
 order_by() (django_tables2.tables.TableData method), 34
 order_by_alias (django_tables2.columns.BoundColumn attribute), 31
 orderable (django_tables2.columns.BoundColumn attribute), 32
 OrderBy (class in django_tables2.utils), 35
 OrderByTuple (class in django_tables2.utils), 35
 ordering (django_tables2.tables.TableData attribute), 34

P

paginate() (django_tables2.tables.Table method), 20

R

record, 49
 record (django_tables2.rows.BoundRow attribute), 34
 RelatedLinkColumn (class in django_tables2.columns), 28
 render, 49
 RequestConfig (class in django_tables2.config), 18

S

Sequence (class in django_tables2.utils), 35
 SingleTableMixin (class in django_tables2.views), 29

T

table, 49
 Table (class in django_tables2.tables), 19
 table (django_tables2.rows.BoundRow attribute), 34
 table data, 49
 Table.Meta (built-in class), 20
 table_class (SingleTableMixin attribute), 29
 table_data (SingleTableMixin attribute), 29
 table_pagination (SingleTableMixin attribute), 29
 table_prefix (MultiTableMixin attribute), 29
 TableData (class in django_tables2.tables), 34
 tables (MultiTableMixin attribute), 29
 tables_data (MultiTableMixin attribute), 29
 template, 49
 TemplateColumn (class in django_tables2.columns), 28

U

URLColumn (class in django_tables2.columns), 28

V

verbose_name (django_tables2.columns.BoundColumn attribute), 32
 verbose_name (django_tables2.tables.TableData attribute), 34
 verbose_name_plural (django_tables2.tables.TableData attribute), 35
 view, 49
 visible (django_tables2.columns.BoundColumn attribute), 32