Django ezTables Documentation

Release 0.3.3.dev

Axel Haustant

July 29, 2015

Contents

Django ezTables provides easy integration between jQuery DataTables and Django.

CHAPTER 1

Compatibility

Django ezTables requires Python 2.6+, Django 1.4+ and Django.js 0.7.6+.

CHAPTER 2

Installation

You can install Django ezTables with pip:

```
$ pip install django-eztables
```

or with easy_install:

```
$ easy_install django-eztables
```

Add `djangojs` and `eztables` to your `settings.INSTALLED_APPS`.

If you want to run the test suite, you will need some additionnal dependencies. You can install them in the same time with:

```
$ pip install django-eztables[tests]
```

CHAPTER 3

Features

- Datatables.net, plugins and localization integration with Django.
- **Server-side processing with a simple view supporting:**
  - sorting (single and multi columns)
  - filtering with regex support (global and by column)
  - formatting using format pattern
- Deferred loading support.
- Twitter Bootstrap integration.

CHAPTER 4

Demo

You can try the demo by cloning this repository and running the test server with the provided data:

```
$ python manage.py syncdb
$ python manage.py loaddata eztables/demo/fixtures/browsers.json
$ python manage.py runserver
```

Then open your browser to http://localhost:8000

CHAPTER 5

Documentation

5.1 Template tags

5.1.1 Initialization

You can either:

- load the template tag lib into each template manually:

```
{% load eztables %}
```

- load the template tag lib by adding it to your `views.py`:

```
from django.template import add_to_builtins

add_to_builtins('eztables.templatetags.eztables')
```

5.1.2 Usage

datatables_js

A `{% datatables_js %}` tag is available to include the datatables javascript library. After loading, you can use the Datatables library as usual:

```
<table id="my-table">
</table>

{% datatables_js %}
<script>
    $('#my-table').dataTable();
</script>
```

bootstrap support

If you want to use the twitter bootstrap style (based on this blog post), 2 template tags are provided for:

- `{% datatables_bootstrap_js %}` for the javascript part.
- `{% datatables_bootstrap_css %}` for the css part.

```
<head>
    {% datatables_bootstrap_css %}

    {% datatables_js %}
    {% datatables_bootstrap_js %}
</head>

<table id="my-table">
</table>

<script>
    $('#my-table').dataTable({
        "bPaginate": true,
        "sPaginationType": "bootstrap",
        "bScrollCollapse": true
    });
</script>
```

5.2 Server-side processing

Django ezTable provide a single view to implement server-side pagination: *eztables.views.DatatablesView*.

It follows the Django Class-based Views pattern and can render Array-based or Object-based JSON.

As it extends `django.views.generic.list.MultipleObjectMixin` it expects the `model` attribute to be set in both case.

Both modes expect a `fields` attribute that can optionnaly contains format patterns.

The exemple will use the same models as the demo:

```
from django.db import models


class Engine(models.Model):
    name = models.CharField(max_length=128)
    version = models.CharField(max_length=8, blank=True)
    css_grade = models.CharField(max_length=3)

    def __unicode__(self):
        return '%s %s (%s)' % (self.name, self.version or '-', self.css_grade)


class Browser(models.Model):
    name = models.CharField(max_length=128)
    platform = models.CharField(max_length=128)
    version = models.CharField(max_length=8, blank=True)
    engine = models.ForeignKey(Engine)

    def __unicode__(self):
        return '%s %s' % (self.name, self.version or '-')
```

5.2.1 Array-based JSON

To render an array-based JSON, you must provide `fields` as a `list` or a `tuple` containing the field names.

```
from eztables.views import DatatablesView
from myapp.models import Browser

class BrowserDatatablesView(DatatablesView):
    model = Browser
    fields = (
        'engine__name',
        'name',
        'platform',
        'engine__version',
        'engine__css_grade',
    )
```

You can simply instanciate your datatable with:

```
$(function(){
    $('#browser-table').dataTable({
        "bPaginate": true,
        "sPaginationType": "bootstrap",
        "bProcessing": true,
        "bServerSide": true,
        "sAjaxSource": Django.url('dt-browsers-default')
    });
});
```

5.2.2 Object-based JSON

To render an array-based JSON, you must provide `fields` as a `dict` containing the mapping between the JSON fields names and the model fields.

```
from eztables.views import DatatablesView
from myapp.models import Browser

class ObjectBrowserDatatablesView(DatatablesView):
    model = Browser
    fields = {
        'name': 'name',
        'engine': 'engine__name',
        'platform': 'platform',
        'engine_version': 'engine__version',
        'css_grade': 'engine__css_grade',
    }
```

You need to use the `aoColumns` properties in the DataTables initialization:

```
$(function(){
    $('#browser-table').dataTable({
        "bPaginate": true,
        "sPaginationType": "bootstrap",
        "bProcessing": true,
        "bServerSide": true,
        "sAjaxSource": Django.url('dt-browsers-objects'),
        "aoColumns": [
            { "mData": "engine" },
            { "mData": "name" },
            { "mData": "platform" },
            { "mData": "engine_version" },
```

```
            { "mData": "css_grade" }
        ]
    });
});
```

5.2.3 Format patterns

You can optionally provide some format patterns in the field definition:

```
from eztables.views import DatatablesView
from myapp.models import Browser

class FormattedBrowserDatatablesView(DatatablesView):
    model = Browser
    fields = (
        'engine__name',
        '{name} {version}',
        'platform',
        'engine__version',
        'engine__css_grade',
    )

class FormattedObjectBrowserDatatablesView(DatatablesView):
    model = Browser
    fields = {
        'name': '{name} {version}',
        'engine': 'engine__name',
        'platform': 'platform',
        'engine_version': 'engine__version',
        'css_grade': 'engine__css_grade',
    }
```

5.2.4 Custom sort

You can implement a custom sort method. It have to be named `sort_col_X` where `X` should be the index given by the datatables request (correspond to the filtered column).

It takes the requested direction (`''` or `'-'`) as a parameter and should return one or more Django order statement.

```
class CustomSortBrowserDatatablesView(BrowserDatatablesView):

    def sort_col_1(self, direction):
        '''Sort on version instead of name'''
        return '%sversion' % direction

    def sort_col_2(self, direction):
        '''Sort on name and platform instead of platform'''
        return ('%sname' % direction, '%splatform' % direction)
```

5.2.5 Custom search

You can implement a custom search method. It has to be named `search_col_X` where `X` should be the index given by the datatables request (correspond to the filtered column).

It takes the search term and the queryset to filter as a parameter and should return the filtered queryset.

```
class CustomSearchBrowserDatatablesView(BrowserDatatablesView):

    def search_col_1(self, search, queryset):
        '''Search on version instead of name'''
        return queryset.filter(version__icontains=search)
```

5.2.6 SQLite Warnings

Be carefull some field types are not compatible with regex search on SQLite and will be ignored (filtering will no performed on this fields).

Ignored fields type are:

- BigIntegerField
- BooleanField
- DecimalField
- FloatField
- IntegerField
- NullBooleanField
- PositiveIntegerField
- PositiveSmallIntegerField
- SmallIntegerField

5.3 Localization

Django ezTable embed DataTables localizations.

They have the following naming convention:

```
{{STATIC_URL}}/js/libs/datatables/language.{{LANG}}.json
```

You can simply retrieve them with django.js:

```
$('#my-table').dataTable({
    ...
    "oLanguage": {
        "sUrl": Django.file("js/libs/datatables/language.fr.json")
    }
    ...
});
```

You can obtain the current language code with django.js too:

```
var code = Django.context.LANGUAGE_CODE;
```

Be careful, no localization is provided for the English language.

5.4 Integration with other tools

5.4.1 Django Pipeline

If you want to compress Django ezTables with Django Pipeline, you should change the way you load it.

First add jQuery, Django.js and jQuery Datatables (and optionnaly bootstrap support) to your pipelines in your `settings.py`:

```
PIPELINE_JS = {
    'base': {
        'source_filenames': (
            '...',
            'js/libs/jquery-2.0.0.js',
            'js/djangojs/django.js',
            'js/libs/datatables/jquery.dataTables.js',
            'js/libs/datatables/datatables.bootstrap.js',
            '...',
        ),
        'output_filename': 'js/base.min.js',
    },
}
```

Add Datatables Bootstrap CSS to your CSS pipeline:

```
PIPELINE_CSS = {
    'base': {
        'source_filenames': (
            '...',
            'css/datatables.bootstrap.css',
            '...',
        ),
        'output_filename': 'js/base.min.css',
    },
}
```

Instead of using the `django_js` template tag:

```
{% load js %}
{% django_js %}
```

you should use the `django_js_init` and include your compressed bundle:

```
{% load js compressed %}
{% django_js_init %}
{% compressed_js "base" %}
```

5.4.2 RequireJS

Django ezTables works with RequireJS but it requires some extras step to do it.

Preloading prerequisites

You should use the `django_js_init` template tag before loading your application with RequireJS.

```
{% load js %}
{% django_js_init %}
<script data-main="scripts/main" src="scripts/require.js"></script>
```

It works with django-require too:

```
{% load js require %}
{% django_js_init %}
{% require_module 'main' %}
```

path and shim configuration

You should add extras paths and shim configurations for Django.js and Datatables:

```
require.config({
    paths: {
        'jquery': 'libs/jquery-2.0.0',
        'django': 'djangojs/django',
        'datatables': 'libs/datatables/js/jquery.dataTables.min',
        'datatables.bootstrap': 'libs/datatables/js/datatables.bootstrap'
    },

    shim: {
        'bootstrap': {
            deps: ['jquery'],
            exports: '$.fn.popover'
        },
        'django': {
            "deps": ["jquery"],
            "exports": "Django"
        },
        'datatables': {
            deps: ["jquery"],
            "exports": "$.fn.dataTable"
        },
        'datatables.bootstrap': {
            deps: ["datatables"]
        }
    }
});
```

Paths are relative to `{{STATIC_URL}}/js`. Adapt it to your configuration.

5.5 API

5.5.1 `eztables` – Main package

5.5.2 `eztables.forms` – DataTables pagination form processing

class `eztables.forms.`**`DatatablesForm`**(**args*, ***kwargs*)

Bases: `django.forms.forms.Form`

Datatables server side processing Form

See: http://www.datatables.net/usage/server-side

bRegex = None
True if the global filter should be treated as a regular expression for advanced filtering, false if not.

iColumns = None
Number of columns being displayed (useful for getting individual column search info)

iDisplayLength = None
Number of records that the table can display in the current draw. It is expected that the number of records returned will be equal to this number, unless the server has fewer records to return.

iDisplayStart = None
Display start point in the current data set.

iSortingCols = None
Number of columns to sort on

sEcho = None
Information for DataTables to use for rendering.

sSearch = None
Global search field

5.5.3 `eztables.views` – DataTables server-side processing view

class `eztables.views.`**`DatatablesView`**(***kwargs*)
Bases: `django.views.generic.list.MultipleObjectMixin`, `django.views.generic.base.View`

Render a paginated server-side Datatables JSON view.

See: http://www.datatables.net/usage/server-side

`can_regex`(*field*)
Test if a given field supports regex lookups

`column_search`(*queryset*)
Filter a queryset with column search

`get_orders`()
Get ordering fields for `QuerySet.order_by`

`get_page`(*form*)
Get the requested page

`get_queryset`()
Apply Datatables sort and search criterion to QuerySet

`get_row`(*row*)
Format a single row (if necessary)

`get_rows`(*rows*)
Format all rows

`global_search`(*queryset*)
Filter a queryset with global search

`render_to_response`(*form*, ***kwargs*)
Render Datatables expected JSON format

`eztables.views.`**`UNSUPPORTED_REGEX_FIELDS`** **= (<class 'django.db.models.fields.IntegerField'>, <class 'django.db.mod**
SQLite unsupported field types for regex lookups

eztables.views.**get_real_field**(*model*, *field_name*)
Get the real field from a model given its name.

Handle nested models recursively (aka. ___ lookups)

5.5.4 eztables.templatetags.eztables – Template tags

5.6 Contributing

Django ezTables is open-source and very open to contributions.

5.6.1 Submitting issues

Issues are contributions in a way so don't hesitate to submit reports on the official bugtracker.

Provide as much informations as possible to specify the issues:

- the Django.js and Django ezTables versions used
- a stacktrace
- installed applications list
- ...

5.6.2 Submitting patches (bugfix, features, ...)

If you want to contribute some code:

1. fork the official Django ezTables repository
2. create a branch with an explicit name (like `my-new-feature` or `issue-XX`)
3. do your work in it
4. rebase it on the master branch from the official repository (cleanup your history by performing an interactive rebase)
5. submit your pull-request

There are some rules to follow:

- your contribution should be documented (if needed)
- your contribution should be tested and the test suite should pass successfully
- your code should be mostly PEP8 compatible with a 120 characters line length
- your contribution should support both Python 2 and 3 (use `tox` to test)

You need to install some dependencies to hack on Django ezTables:

```
$ pip install -r requirements/develop.pip
```

A Makefile is provided to simplify the common tasks:

```
$ make
Makefile for Django ezTables

Usage:
   make serve            Run the test server
   make test             Run the test suite
   make coverage         Run a caoverage report from the test suite
   make pep8             Run the PEP8 report
   make pylint           Run the pylint report
   make doc              Generate the documentation
   make minify           Minify all JS files with yuglify
   make dist             Generate a distributable package
   make minify           Minify Django.js with yuglify
   make clean            Remove all temporary and generated artifacts
```

To ensure everything is fine before submission, use `tox`. It will run the test suite on all the supported Python version and ensure the documentation is generating.

```
$ pip install tox
$ tox
```

You also need to ensure your code is PEP8 compliant (following the project rules: see `pep8.rc` file):

```
$ make pep8
```

Don't forget client-side code and tests.

You can run the javascript test suite in the browser (http://localhost:8000). Javascript tests are run in the test suite too, but it runs on the minified version of the javascript libary.

You can use the Makefile `minify` task that minify the javascript:

```
$ make minify test
```

Note: minification use `yuglify` so you need to install it before: `npm install -g yuglify`

5.7 Changelog

5.7.1 Current

- nothing yet

5.7.2 0.3.2 (2013-06-07)

- Python 2.6 support

5.7.3 0.3.1 (2013-05-05)

- Prevent errors on regex lookups with SQLite (issue #5)

5.7.4 0.3.0 (2013-04-30)

- Python 3 support
- Documented integration with Django Pipeline or RequireJS
- Package the unminified version too (used when `settings.DEBUG=True`)

5.7.5 0.2.2 (2013-03-02)

- Django 1.5 compatibility
- Added custom search and sort demo

5.7.6 0.2.1 (2013-02-08)

- Fix formatting with unicode

5.7.7 0.2 (2013-02-07)

- Handle custom server-side search implementation
- Handle custom server-side sort implementation

5.7.8 0.1.2 (2013-02-07)

- Fix static files packaging

5.7.9 0.1.1 (2013-02-07)

- Fix requirements packaging

5.7.10 0.1 (2013-02-07)

- Initial implementation

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

e

B

C

D

E

G

I

R

S

U