djqgrid Documentation

Release 0.2

Itay Zandbank

January 11, 2017

Contents

djqgrid is a Django wrapper for jqGrid.

djqgrid lets you define grids in a Django-familiar way, while taking care of most of the mundane Python-JavaScript bridge for you.

CHAPTER 1

Installation

To install djqgrid, please follow these steps:

1. Install with `pip install djqgrid`
2. Add `djqgrid` to your `INSTALLED_APPS`.
3. Reference the `jqGrid` and `jQueryUI` JavaScript and CSS files
4. Reference the script at `{% static "js/djqgrid_utils.js" %}`
5. Add the `djqgrid` URLs to `urls.py`:

```
urlpatterns += patterns('', url(r^'grid_json/', include (djqgrid.urls))
```

CHAPTER 2

Example

Using djqgrid is relatively straightforward, but it does take a little work.

2.1 Define your model

```
class MyModel(models.Model):
    name = models.CharField(max_length=40)
    desc = models.CharField(max_length=100)
    url = models.URLField()
    height = models.IntField()
```

2.2 Define your grid

```
class MyGrid(Grid):
    model = MyModel

    name = TextColumn(title='Name', model_path='name')
    height = TextColumn(title='Height', model_path='height', align='right')
    desc = LinkColumn(title='Description', model_path='desc', url_builder=lambda m: m.url)
```

What we have here is a grid associated with `MyModel` objects - each grid row represents one object. The grid has three columns:

1. Name - a basic column containing `model.name`
2. Height - containing `model.height`, but right aligned
3. Description - containing a link - its text will be `model.desc` and the URL will be `model.url`

One thing to note is `align='right'` - this property is passed directly to jqGrid in the column's *colModel*. Any property can be passed to jqGrid this way. For example `TextColumn(title=..., model_path=..., editable=true)` creates an editable column.

2.3 Add the grid to your view and template

The view:

```
define myview(request):
    grid = MyGrid()
    return render(request, 'my_template.html', { grid: grid })
```

The template:

```
{% load djqgrid %}

<div id="grid-div">
    {% jqgrid grid %}
</div>
```

Now run the view. You should see a very nice grid that supports paging and sorting.

CHAPTER 3

Reference

djqgrid consists of several components:

3.1 The Grid Class

class djqgrid.grid.**Grid**(***kwargs*)

The Grid class everybody will use.

We use BaseGrid, and add the fields metaclass magic, as is done in the Django forms

Creates a Grid

Args: **kwargs: All the arguments will go to the grid's `option` array.

The grid's `option` array will be used to instantiate the jqGrid on the browser side: `$("#grid").jqGrid(options)`

If you want to pass handlers for events, such as `loadComplete`, wrap then with `function` so that the options are rendered correctly in JavaScript. For example:

```
grid = MyGrid(loadComplete=function('loadCompleteHandler'))
```

For more information, see the `json_helpers` module.

_apply_query(*queryset*, *querydict*)

This function lets a Grid instance change the default query, if it's ever necessary

Args: queryset: The default query (`self.model.objects`) querydict: The request's query string

Returns: The actual queryset to use

Override this function if the default query is not enough

_apply_sort(*queryset*, *querydict*)

Applys sorting on the queryset.

jqGrid supports sorting, by passing `sidx` and `sord` in the request's query string. `_apply_sort` applies this sorting on a queryset.

Args: queryset: A queryset for the objects of tje grid querydict: The request's querydict with sidx and sord

Returns: An ordered queryset.

_get_additional_data(*model*)

Retrieves additional data to be sent back to the client.

Args: model: The model of the currently rendered row

Returns: A dictionary with more data that will be sent to the client, or `None` if there's no such data

_get_query_results(*querydict*)
Returns a queryset to populate the grid

Args: querydict: The request's query dictionary

Returns: The queryset that will be used to populate the grid. Paging will be applied by the caller.

_model_to_dict(*model*)
Takes a model and converts it to a Python dictionary that will be sent to the jqGrid.

This is done by going over all the columns, and rendering each of them to both text and HTML. The text is put in the result dictionary. The result dictionary also has an `html` dictionary, which contains the HTML rendering of each column.

This is done to solve a bug with jqGrid's inline editing, that couldn't handle HTML values of cells properly. So instead we use a formatter, called `customHtmlFormatter` (in `djqgrid_utils.js`) that can take the value from the html dictionary.

Sometimes more information is required by the JavaScript code (for example, to choose row CSS styles). It is possible to add additional information. The method `_get_additional_data` returns this additional information, which is put in result JSON as well.

So, for example, a Grid with two columns will have a JSON looking likes this:

{col1: 'col1-text', col2: 'col2-text', html: {

'col1': 'col1-html', 'col2': 'col2-html'

}, additional: { ... }

}

classmethod get_grid_id()
Returns the grid's class ID.

This is done by computing a CRC32 of the class's name. Using CRC32 is not a security risk - IDs are passed in the HTTP anyway, so they are no secret and being able to generate them is not going to help an attacker.

get_json_data(*querydict*)
Returns a JSON string with the grid's contents

Args: querydict: The request's query dictionary

Returns: JSON string with the grid's contents

DO NOT override this method unless absolutely necessary. `_apply_query` and `_get_additional_data` should be overridden instead.

get_options(*override=None*)
Returns the grid's options - this options will be passed to the JavaScript `jqGrid` function

Args: override: A dictionary that overrides the options provided when the grid was initialized.

Returns: A dictionary with the grid's options.

Some fields cannot be overridden: - `colNames` are created from the grid's `Column` fields - `colModels` are also created from the grid's `Column` fields. - `url` always points to the `djqgrid.views.query` view with the grid's ID.

3.2 Columns

class djqgrid.columns.**Column**(*title*, *model_path*, ***kwargs*)
A Column object represents one Grid column

The Column object defines how a column's value is retrieved and how it is formatted.

Column is effectively an abstract class - Grids should be using its derived classes.

Initializes a column.

Args: title: The column's title in the grid. title.lower() is the default column name. model_path: The column's data field's path in the model.

For example, if this is the `Name` column in a grid of `Person` models, and the person's name is in the `fullname` attribute of the `Person` model, `model_path` should be `fullname`. `model_path` can also access attributes in nested objects: `innerobj.attr` will be resolved to `model_instance.innerobj.attr`

****kwargs**: All other arguments are copied as is to the column's `colModel`.

get_sort_name()
Returns the column's "sort-name", which is used to apply ordering on a queryset.

The default implementation is to take the model_path and replace each . with __ .

model
Returns the `colModel`

render_html(*model*)
Returns an HTML representation of the column's value.

The default implementation is to return the text value.

render_text(*model*)
Returns a text representation of the column's value.

The default implementation is to convert `get_model_value` to a unicode string.

title
Returns the name that goes in the colName JSON

class djqgrid.columns.**TextColumn**(*title*, *model_path*, ***kwargs*)
A column that contains simple text strings.

This is basically just a Column. We create a new class because it seems more tidy.

Initializes a column.

Args: title: The column's title in the grid. title.lower() is the default column name. model_path: The column's data field's path in the model.

For example, if this is the `Name` column in a grid of `Person` models, and the person's name is in the `fullname` attribute of the `Person` model, `model_path` should be `fullname`. `model_path` can also access attributes in nested objects: `innerobj.attr` will be resolved to `model_instance.innerobj.attr`

****kwargs**: All other arguments are copied as is to the column's `colModel`.

class djqgrid.columns.**TemplateColumn**(*title*, *model_path*, *template=None*, *template_name=None*, ***kwargs*)
A column that is rendered by a Django template

Use a `TemplateColumn` when you want the column to contain complex HTML content.

Due to a bug in jqGrid, cell values cannot contain HTML - it conflicts with inline editing. So instead, the cell values returned by the default `Grid.get_json_data` method will be the text rendering of the column. The HTML data will be placed in the `html` property of the row's data.

To actually display the HTML data we use a formatter, called `customHtmlFormatter`, which resides in `djqgrid_utils.js`

Initializes a TemplateColumn

Args: title: The column's title model_path: The column's model path template: A Django template that will be rendered template_name: A name of a Django template that will be rendered. **kwargs: Additional arguments that will be passed directly to the column's `colModel`

You can specify a template *or* a template_name, but not both.

class `djqgrid.columns.`**`LinkColumn`**(*title*, *model_path*, *url_builder*, ***kwargs*)
A column that is rendered as a single `<a href=...>` link

Initializes a LinkColumn.

Args: title: The column's title model_path: The column's model_path url_builder: a function that takes a model and returns the link's URL **kwargs: Additional arguments passed directly to the column's `colModel`

3.3 The djqgrid Template Tag

To use djqgrid in a template, you need to load the djqgrid template tag first, using

```
{% load jqgrid %}
```

`djqgrid.templatetags.jqgrid.`**`jqgrid`**(*context*, *grid*, *prefix=''*, *pager=True*, *urlquery=None*, ***kwargs*)
Adds a complete jqGrid - HTML and JavaScript - to the template.

Two HTML elements are added, a `<table id='grid'>` and a `<div id='pager'>`. JavaScript code to initialize the jqGrid is also added.

After the grid is set up in the browser, it will access the server again and ask for the grid data. The URL is defined in the Grid object, and is appended by the `urlquery` argument and the current request's query dict. For example, if the template is rendering the URL `/view?p=1`, the grid's URL is `/grid/17?g=2` and `urlquery` is `q=3`, the grid's data will be retrieved from `/grid/17?g=2&q=3&p=1`

Args: context - The template context grid - the Grid prefix - A prefix for the grid and pager element IDs. The default is no prefix, meaning the elements are

named `grid` and `pager`. Adding `prefix='prefix'` creates elements with the IDs `prefix-grid` and `prefix-pager`.

pager - True if a pager is added to the grid. If no pager is added, the row count is set to 99,999. urlquery - An additional query string that will be added to the data request that will be sent to the server. **kwargs - All additional arguments are added as is to the jqGrid initialization option object.

Returns: The generated HTML

3.4 Utility Functions

3.4.1 json_helpers

djqgrid.json_helpers.**function**(*funcname*)

Wraps a JavaScript function name with our token, so it can be unquoted when rendering to JSON.

The jqGrid option object is actually a Python dictionary that is rendered to JSON in the jqgrid template tag.

Unfortunately, Python's `json` module can't output such JSON, it will always put quotes around strings (without the quotes, it's not legal JSON), so we need to work around it to support unquoted strings.

The solution is simple and a bit ugly - we use a token that we wrap around function names. We serialize the dictionary to a JSON string, look for the token and remove the quotes around it.

Our token is `@@`. So in the example about, our Python dict will be: `{'loadComplete': '@@loadCompleteHandler@@')`. When we create the JSON, we remove the quotes and the @@, and end up with the expected string.

The `function` helper function puts the token around the string, so that we can build our dictionary like so: `d = {'loadComplete': function('loadCompleteHandler'))`.

Args: funcname - the function's name

Returns: The wrapped function name

djqgrid.json_helpers.**dumps**(*o*, **args*, ***kwargs*)

Serializes an object to JSON, unquoting function names.

Args: o: Object to serialize *args: Additional arguments passed to `json.dumps` **kwargs: Additional arguments passed to `json.dumps`

3.4.2 grid_registrar

This module handles all the grid registration.

A Grid object is created multiple times - when the HTML containing the grid is rendered, and every time a the grid's data needs to be retrieved. Since each of these times is an independent HTTP request, we need to somehow pass the information of the grid's class. This is done with a *Grid ID*.

Each HTTP request *gets its own Grid instance*, we just pass the information of the Grid's *class* around. Since we can't create a class just because we've received its name in an HTTP request (it's a *huge* security hole), we only create classes that have been registered before. We also don't want to pass class names in HTTP requests, so we pass class IDs.

djqgrid.grid_registrar.**get_grid_class**(*id*)

Returns a class for a given ID.

Args: Grid ID

Returns: The grid's class

Raises: KeyError if the ID hasn't been registered

djqgrid.grid_registrar.**register_grid**(*gridcls*)

Registers a Grid class in the registry.

Calls `gridcls.get_grid_id` to get the class's ID.

Args: gridcls: Class of Grid to registry

3.4.3 views

class djqgrid.views.**JsonResponse**(*content*, *status=None*, *content_type='application/json'*)
Returns a JSON Response

Takes the content object, json.dumps it and returns it an a response.

Args: content: The object to be serialized into JSON mimetype: Response mime type. Default is application/json status: HTTP status code. Default is NONE. content_type: The response content type. Default is application/json

Returns: A Django response object

djqgrid.views.**query**(*request*, *grid_id*)
Returns the grid content in a JSON response

query creates a new Grid instance based on the grid_id (which represents the Grid class), and calls grid.get_json_data to perform the actual query.

Args: request: Django request grid_id: ID of the grid. The ID is generated by Grid.get_grid_id

Returns: The JSON serialized grid contents.

CHAPTER 4

Indices and tables

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

d

Symbols

C

D

F

G

J

L

M

Q

R

T