django-datatable-view Documentation

Release 0.9

Autumn Valenta

October 05, 2018

Contents

1 0.9 Migration Guide **3**
- 1.1 dataTables.js 1.10 . . . 3
- 1.2 Update configuration style . . . 3
- 1.3 New vocabulary . . . 4
- 1.4 No more automatic column callbacks . . . 4
- 1.5 No more automatic dataTables.js initialization . . . 4
- 1.6 Double check your default structure template . . . 5
- 1.7 Update complex column definitions . . . 5
- 1.8 Custom model fields . . . 5
- 1.9 Experiment with the new `ValuesDatatable` . . . 5

2 Topics **7**
- 2.1 The client and server interaction model . . . 7
- 2.2 Searching . . . 8
- 2.3 Sorting . . . 9
- 2.4 Third-party model fields . . . 11
- 2.5 Caching . . . 11

3 `datatableview` module documentation **15**
- 3.1 `views` . . . 15
- 3.2 `datatables` . . . 15
- 3.3 `columns` . . . 18
- 3.4 `forms` . . . 18
- 3.5 `helpers` . . . 18

4 Indices and tables **21**

Python Module Index **23**

For working demos and example code with explanations on common configurations, visit the demo site at http://example.com.

Contents:

CHAPTER 1

0.9 Migration Guide

The jump from the 0.8.x series to 0.9 is covered in sections below.

dataTables.js 1.10

Note See the official 1.10 announcement if you've been living under a rock!

DataTables 1.10 provides a brand new api for getting things done, and it's a good thing too, because doing anything fancy in the old api pretty much required Allan to write yet another block of example code that everyone just copies and pastes.

For our 0.9 release of django-datatable-view, we still use the "legacy" constructor to get things going, but that's okay, because the legacy api is still completely supported (even if all of its Hungarian notation keeps us up at night). The drawback at this stage is that we can't yet accept configuration settings that are "new-style only".

Despite the fact that we're using the legacy constructor for a while longer, you can access the table's fancy new API object with one simple line:

```
// Standard initialization
var opts = {};
var datatable = datatableview.initialize($('.datatable'), opts);

// Get a reference to the new API object
var table = datatable.api();
```

Update configuration style

Note See Datatable object and Meta for examples.

The preferred way to configure columns for a view is now to use the `Datatable` class. It has similarities to the Django `ModelForm`: the class uses an inner *`Meta`* class to specify all of the options that we used to provide in your view's `datatable_options` dict.

You want to just unpack the keys and values from your existing `datatable_options` dict and set those as attributes on a *`Meta`*. Then just assign this `Datatable` subclass on your view:

```
class MyDatatable(Datatable):
    class Meta:
        columns = [ ... ]
        search_fields = [ ... ]
        # etc
```

```
class MyDatatableView(DatatableView):
    datatable_class = MyDatatable
```

An alternate abbreviated style is available: as with class-based views that use Django forms, you can set these `Meta` attributes directly on the view class, shown in more detail here. Please note that if you're declaring anything fancier than simple model fields or methods as columns (typically anything that would have required the 2-tuple or 3-tuple column syntax), please use the new `Datatable` object strategy.

The new `Datatable` object doubles as the old 0.8 `DatatableOptions` template renderable object. `DatatableOptions` and `utils.get_datatable_structure()` have both been removed, since `Datatable` itself is all you need.

New vocabulary

> **Celebrate** We're becoming more sophisticated!

Now that we spent a bunch of time learning how to use the tools we created, it felt like a good time to change some of the terms used internally.

In connection with the new `Datatable` object that helps you design the datatable, **we've started referring to column data callbacks as "processors"**. This means that we will stop relying on callbacks in the documentation being named in the pattern of `'get_column_FOO_data()'`. Instead, you'll notice names like `'get_FOO_data()'`, and we'll be specifying the callback in a column definition via a `processor` keyword argument. See Postprocessors for a examples of this.

No more automatic column callbacks

> **The Zen of Python** Explicit is better than implicit.

We knew that implicit callbacks was a bad idea, but in our defense, the deprecated column format was really cumbersome to use, and implicit callbacks were saving us some keystrokes. **This behavior is going away in version 1.0.** We continue to support implicit callbacks so that 0.9 is a backwards-compatible release with 0.8. If you have any column callbacks (we're calling them "processors" now) that aren't explicitly named in the column definition, please update your code soon!

No more automatic dataTables.js initialization

> **Note** Bye bye `function confirm_datatable_options(options){ ... }`

Automatic initialization has gone the way of the buffalo, meaning that it doesn't exist anymore. The global JavaScript function `confirm_datatable_options` only ever existed because auto initialization took away your chance to set custom options during the init process. You should initialize your datatables via a simple call to the global function `datatableview.initialize($('.datatable'), opts)`. This JS function reads DOM attributes from the table structure and builds some of the column options for you, but you can pass literally any other supported option in as the second argument. Just give it an object, and everything will be normal.

There is a configurable Javascript flag `datatableview.auto_initialize` that previously defaulted to `true`, but in 0.9 its default value is now `false`. If you need 0.9 to behave the way it did in 0.8, set this flag globally or per-page as needed. (Be careful not to do it in a `$(document).ready()` handler, since auto initialization runs during that hook. You might end up flagging for auto initialization after datatableview.js has already finished checking it, and nothing will happen.)

Double check your default structure template

Note See Custom render template for examples.

If you haven't gone out of your way to override the default structure template or create your own template, this shouldn't apply to you.

The 0.9 default structure template at `datatableview/default_structure.html` has been modified to include a reference to a `{% templatetag openvariable %} config {% templatetag closevariable %}` variable, which holds all of the configuration values for the table. The render context for this template previously held a few select loose values for putting `data-*` attributes on the main `<table>` tag, but the template should now read from the following values (note the leading `config.`:

- `{{ config.result_counter_id }}`
- `{{ config.page_length }}`

Update complex column definitions

Note See Custom verbose names, Model method-backed columns, Postprocessing values, and Compound columns for examples.

The now-deprecated 0.8 column definition format had a lot of overloaded syntax. It grew out of a desire for a simple zero-configuration example, but became unwieldy, using nested tuples and optional tuple lengths to mean different things.

The new format can be thought of as a clone of the built-in Django forms framework. In that comparison, the new `Datatable` class is like a Form, complete with Meta options that describe its features, and it defines `Column` objects instead of FormFields. A `Datatable` configuration object is then given to the view in the place of the old `datatable_options` dictionary.

In summary, the old `datatable_options` dict is replaced by making a `Datatable` configuration object that has a `Meta`.

The task of showing just a few specific columns is made a bit heavier than before, but (as with the forms framework) the new Meta options can all be provided as class attributes on the view to keep the simplest cases simple.

Custom model fields

Note See Custom model fields for new registration strategy.

Custom model fields were previously registered in a dict in `datatableview.utils.FIELD_TYPES`, where the type (such as `'text'`) would map to a list of model fields that conformed to the text-style ORM query types (such as `__icontains`).

In 0.9, the registration mechanism has changed to a priority system list, which associates instances of the new `Column` class to the model fields it can handle. See Custom model fields for examples showing how to register model fields to a built-in `Column` class, and how to write a new `Column` subclass if there are custom ORM query types that the field should support.

Experiment with the new `ValuesDatatable`

Note See ValuesDatatable object for examples.

An elegant simplification of the datatable strategy is to select the values you want to show directly from the database and just put them through to the frontend with little or no processing. If you can give up declaration of column sources as model methods and properties, and rely just on the data itself to be usable, try swapping in a `ValuesDatatable` as the base class for your table, rather than the default `Datatable`.

This saves Django the trouble of instantiating model instances for each row, and might even encourage the developer to think about their data with fewer layers of abstraction.

CHAPTER 2

Topics

The client and server interaction model

High-level description

Traditionally, developers using `dataTables.js` have approached their table designs from the client side. An ajax backend is just an implementation detail that can be enabled "if you need one."

From the perspective of a Django application, however, we want to flip things around: the `datatableview` module has all of the tools required to build a server-side representation of your table, such as the column names, how it derives the information each column holds, and which sorting and filtering features it will expose.

The execution steps for a server-driven table look like this:

- The developer declares a view.
- The view holds a table configuration object (like a Django `ModelForm`).
- The view puts the table object in the template rendering context.
- The template renders the table object directly into the HTML, which includes its own template fragment to put the basic table structure on the page. (We happen to render a few `data-*` attributes on the `<th>` headers in the default template, but otherwise, the template isn't very interesting.)
- The developer uses a javascript one-liner to initialize the table to get `dataTables.js` involved.

From then on, the process is a loop of the user asking for changes to the table, and the server responding with the new data set:

- The client sends an ajax request with `GET` parameters to the current page url.
- The view uses the same table configuration object as before.
- The view gives the table object the initial queryset.
- The table configuration object overrides its default settings with any applicable `GET` parameters (sorting, searches, current page number, etc).
- The table configuration object applies changes to the queryset.
- The view serializes the final result set and responds to the client.

Expanded details about some of these phases are found below.

The table configuration object

The `Datatable` configuration object encapsulates everything that the server understands about the table. It knows how to render its initial skeleton as HTML, and it knows what to do with a queryset based on incoming `GET` parameter data from the client. It is designed to resemble the Django `ModelForm`.

The resemblance with `ModelForm` includes the use of an inner *`Meta`* class, which can specify which model class the table is working with, which fields from that model to import, which column is sorted by default, which template is used to render the table's HTML skeleton, etc.

`Column` s can be added to the table that aren't just simple model fields, however. Columns can declare any number of `sources`, including the output of instance methods and properties, all of which can then be formatted to a desired HTML result. Columns need not correspond to just a single model field!

The column is responsible for revealing the data about an object (based on the `sources` it was given), and then formatting that data as a suitable final result (including HTML).

Update the configuration from `GET` parameters

Many of the options declared on a `Datatable` are considered protected. The column definitions themselves, for example, cannot be changed by a client playing with `GET` data. Similarly, the table knows which columns it holds, and it will not allow filters or sorting on data that it hasn't been instructed to inspect. `GET` parameters are normalized and ultimately thrown out if they don't agree with what the server-side table knows about the table.

Generating the queryset filter

Because each column in the table has its `sources` plainly declared by the developer, the table gathers all of the sources that represent model fields (even across relationships). For each such source, the table matches it to a core column type and uses that as an interface to ask for a `Q()` filter for a given search term.

The table combines all of the discovered filters together, making a single `Q()` object, and then filters the queryset in a single step.

Read Searching for more information about how a column builds its `Q()` object.

The client table HTML and javascript of course don't know anything about the server's notion of column sources, even when using column-specific filter widgets.

Sorting the table by column

Because a column is allowed to refer to more than one supporting data source, "sorting by a column" actually means that the list of sources is considered as a whole.

Read Sorting to understand the different ways sorting can be handled based on the composition of the column's sources.

As with searching, the client table HTML and javascript have no visibility into the column's underlying sources. It simply asks for a certain column index to be sorted, and the server's table representation decides what that means.

Searching

All searching takes place on the server. Your view's `Datatable` is designed to have all the information it needs to respond to the ajax requests from the client, thanks to each column's `sources` list. The order in which the individual sources are listed does not matter (although it does matter for Sorting).

Sources that refer to non-`ModelField` attributes (such as methods and properties of the object) are not included in searches. Manual searches would mean fetching the full, unfiltered queryset on every single ajax request, just to be sure that no results were excluded before a call to `queryset.filter()`.

Important terms concerning column `sources`:

- **db sources**: Sources that are just fields managed by Django, supporting standard queryset lookups.
- **Virtual sources**: Sources that reference not a model field, but an object instance method or property.
- **Compound column**: A Column that declares more than one source.
- **Pure db column, db-backed column**: A Column that defines only db-backed sources.
- **Pure virtual column, virtual column**: A Column that defines only virtual sources.
- **Sourceless column**: A Column that declares no sources at all (likely relying on its processor callback to compute some display value from the model instance).

Parsing the search string

When given a search string, the `Datatable` splits up the string on spaces (except for quoted strings, which are protected). Each "term" is required to be satisfied somewhere in the object's collection of column `sources`.

For each term, the table's `Column` objects are asked to each provide a filter `Q()` object for that term.

Deriving the `Q()` filter

Terms are just free-form strings from the user, and may not be suitable for the column's data type. For example, the user could search for `"54C-NN"`, and a integer-based column simply cannot coerce that term to something usable. Similar, searching for `"13"` is an integer, but isn't suitable for a `DateTimeField` to query as a `__month`.

Consequently, a column has the right to reject any search term that it is asked to build a query for. This allows columns to protect themselves from building invalid queries, and gives the developer a way to modify their own columns to decide what terms mean in the context of the data type they hold.

A column's `search()` method is called once per term. The default implementation narrows its `sources` down to just those that represent model fields, and then builds a query for each source, combining them with an `OR` operator. All of the different column `Q()` objects are then also combined with the `OR` operator, because global search terms can appear in any column.

The only place an `AND` operator is used is from within the `Datatable`, which is combining all the results from the individual per-column term queries to make sure all terms are found.

Compound columns with different data types

Multiple sources in a single column don't need to be the same data type. This is a quirk of the column system. Each source is automatically matched to one of the provided `Column` classes, looked up based on the source's model field class. This allows the column to ask internal copies of those column classes for query information, respecting the differences between data types and coercion requirements.

Sorting

All sorting takes place on the server. Your view's `Datatable` is designed to have all the information it needs to respond to the ajax requests from the client, thanks to each column's `sources` list. Unlike for searching, the order in

which the individual sources are listed might matter to the user.

Important terms concerning column `sources`:

- **db sources**: Sources that are just fields managed by Django, supporting standard queryset lookups.
- **Virtual sources**: Sources that reference not a model field, but an object instance method or property.
- **Compound column**: A Column that declares more than one source.
- **Pure db column, db-backed column**: A Column that defines only db-backed sources.
- **Pure virtual column, virtual column**: A Column that defines only virtual sources.
- **Sourceless column**: A Column that declares no sources at all (likely relying on its processor callback to compute some display value from the model instance).

Pure database columns

The ideal scenario for speed and simplicity is that all `sources` are simply queryset lookup paths (to a local model field or to one that is related). When this is true, the sources list can be sent directly to `queryset.order_by()`.

Reversing the sort order will reverse all source components, converting a sources list such as `['id', 'name']` to `['-id', '-name']`. This can be sent directly to `queryset.order_by()` as well.

Mixed database and virtual sources

When a column has more than one source, the `Datatable` seeks to determine if there are ANY database sources at all. If there are, then the virtual ones are discarded for the purposes of sorting, and the strategy for pure database sorting can be followed.

The strategic decision to keep or discard virtual sources is a complex one. We can't, in fact, just sort by the database fields first, and then blindly do a Python `sort()` on the resulting list, because the work performed by `queryset.order_by()` would be immediately lost. Any strategy that involves manually sorting on a virtual column must give up queryset ordering entirely, which makes the rationale for abandoning virtual sources easy to see.

Pure virtual columns

When a column provides only virtual sources, the whole queryset will in fact be evaluated as a list and the results sorted in Python accordingly.

Please note that the performance penalty for this is undefined: the larger the queryset (after search filters have been applied), the harder the memory and speed penalty will be.

Columns without sources

When no sources are available, the column automatically become unsortable by default. This is done to avoid allowing the column to claim the option to sort, yet do nothing when the user clicks on it.

Third-party model fields

Registering fields with custom columns

Any model field that subclasses a built-in Django field is automatically supported out of the box, as long as it supports the same query types (`__icontains`, `__year`, etc) as the original field.

A third-party field that is defined from scratch generally needs to become registered with a `Column`. The most straightforward thing to do is to subclass the base `Column`, and set the class attribute `model_field_class` to the third-party field. This will allow any uses of that model field to automatically select this new column as the handler for its values.

Just by defining the column class, it will be registered as a valid candidate when model fields are automatically paired to column classes.

Important gotcha: Make sure the custom class is imported somewhere in the project if you're not already explicitly using it on a table declaration. If the column is never imported, it won't be registered.

If the column needs to indicate support for new query filter types, declare the class attribute `lookup_types` as a list of those operators (without any leading `__`). You should only list query types that make sense when performing a search. For example, an `IntegerField` supports `__lt`, but using that in searches would be unintuitive and confusing, so it is not included in the default implementation of `IntegerColumn`. You may find that `exact` is often the only sensible query type.

New column subclasses are automatically inserted at the top of the priority list when the column system needs to discover a suitable column for a given model field. This is done to make sure that the system doesn't mistake a third-party field that subclasses a built-in one like `CharField` isn't actually mistaken for a simple `CharField`.

Skipping column registration

Some column subclasses are not suitable for registration. For example, a custom column that is intended for use on only *some* `CharField` fields should definitely not attempt to register itself, since this would imply that all instances of `CharField` should use the new column. An example of this is the built-in `DisplayColumn`, which is a convenience class for representing a column that has no sources.

By explicitly setting `model_field_class` to `None`, the column will be unable to register itself as a handler for any specific model field. Consequently, it will be up to you to import and use the column where on tables where it makes sense.

Caching

The caching system is opt-in on a per-`Datatable` basis.

Each Datatable can specify in its *`Meta`* options a value for the *`cache_type`* option.

Caching Strategies

The possible values are available as constants on `datatableview.datatables.cache_types`. Regardless of strategy, your *Settings* will control which Django-defined caching backend to use, and therefore the expiry time and other backend characteristics.

cache_types.DEFAULT

A stand-in for whichever strategy `DATATABLEVIEW_DEFAULT_CACHE_TYPE` in your *Settings* specifies. That setting defaults to `SIMPLE`.

cache_types.SIMPLE

Passes the `object_list` (usually a queryset) directly to the cache backend for pickling. This is a more faithful caching strategy than `PK_LIST` but becomes noticeably slower as the number of cached objects grows.

cache_types.PK_LIST

Assumes that `object_list` is a queryset and stores in the cache only the list of `pk` values for each object. Reading from the cache therefore requires a database query to re-initialize the queryset, but because that query may be substantially faster than producing the original queryset, it is tolerated.

Because this strategy must regenerate the queryset, extra information on the original queryset will be lost, such as calls to `select_related()`, `prefetch_related()`, and `annotate()`.

cache_types.NONE

An explicit option that disables a caching strategy for a table. Useful when subclassing a Datatable to provide customized options.

Settings

There are a few project settings you can use to control features of the caching system when activated on a `Datatable`.

DATATABLEVIEW_CACHE_BACKEND

Default `'default'`

The name of the Django `CACHES` backend to use. This is where cache expiry information will be specified.

DATATABLEVIEW_CACHE_PREFIX

Default `'datatableview_'`

The prefix added to every cache key generated by a table's `get_cache_key()` value.

DATATABLEVIEW_DEFAULT_CACHE_TYPE

Default `'simple'` (`datatableview.datatables.cache_types.SIMPLE`)

The caching strategy to use when a Datatable's Meta option *`cache_type`* is set to `cache_types.DEFAULT`.

DATATABLEVIEW_CACHE_KEY_HASH

Default `True`

Controls whether the values that go into the cache key will be hashed or placed directly into the cache key string.

This may be required for caching backends with requirements about cache key length.

When `False`, a cache key might resemble the following:

```
datatableview_datatable_myproj.myapp.datatables.MyDatatable__view_myproj.myapp.views.MyView__user_77
```

When `True`, the cache key will be a predictable length, and might resemble the following:

```
datatableview_datatable_3da541559918a808c2402bba5012f6c60b27661c__view_1161e6ffd3637b302a5cd74076283a
```

DATATABLEVIEW_CACHE_KEY_HASH_LENGTH

Default `None`

When *DATATABLEVIEW_CACHE_KEY_HASH* is `True`, setting this to an integer will slice each hash substring to the first N characters, allowing you to further control the cache key length.

For example, if set to `10`, the hash-enabled cache key might resemble:

```
datatableview_datatable_3da5415599__view_1161e6ffd3__user_77
```

CHAPTER 3

datatableview module documentation

views

DatatableView

views.xeditable

views.legacy

The `legacy` module holds most of the support utilities required to make the old tuple-based configuration syntax work.

Use `LegacyDatatableView` as your view's base class instead of `DatatableView`, and then declare a class attribute `datatable_options` as usual. This strategy simply translates the old syntax to the new syntax. Certain legacy internal hooks and methods will no longer be available.

datatables

Server-side Datatables are Form-like classes that are responsible for processing ajax queries from the client. A Datatable is referenced by a view, and the view initializes the Datatable with the original queryset. The Datatable is responsible for filtering and sorting the results, and the final object list is handed back to the view for serialization.

A `Datatable`, like a `ModelForm`, should contain an inner `Meta` class that can declare various options for importing model fields as columns, setting the verbose names, etc.

Datatable

ValuesDatatable

Legacy support Datatables

LegacyDatatable

ValuesLegacyDatatable

`Meta` class and options

class `Meta`

`model`

Default `queryset.model`

The model class represented by the table.

`columns`

Default All local non-relationship model fields.

The list of local model fields to be imported from the base model. The appropriate `Column` will be generated for each. Relationship-spanning ORM paths should not be used here, nor any "virtual" data getter like a method or property. For those, you should instead declare an explicit column on the `Datatable` with a name of your choosing, and set the `sources` accordingly.

`exclude`

Default `[]`

A list of model field names to exclude if `columns` is not given.

`cache_type`

Default `None`

The identifier for caching strategy to use on the `object_list` sent to the datatable. See Caching for more information.

`ordering`

Default The `model` 's `Meta.ordering` option.

A list that controls the default table sorting, giving column names in the order of their sort priority. When a Column name is given instead of a model field name, that column's `sources` list will be looked up for any sortable fields it references.

As with model ordering, using a − prefix in front of a name will reverse the order.

`page_length`

Default `25`

The default page length for response results. This can be changed by the user, and is ultimately in the hands of the client-side JS to configure.

`search_fields`

Default `[]`

A list of extra query paths to use when performing searches. This is useful to reveal results that for data points that might not be in the table, but which the user might intuitively expect a match.

Example `['house__city__abbreviation]`

`unsortable_columns`

Default `[]`

A list of model fields from `columns` that should not be sortable when their `Column` instances are created. Explicitly declared columns should send `sortable=False` instead of listing the column here.

hidden_columns

Default `[]`

A list of column names that will be transmitted during ajax requests, but which the client should hide from the table by default. Using this setting does not enhance performance. It is purely for datatable export modes to use as a hint.

structure_template

Default `'datatableview/default_structure.html'`

The template that will be rendered when the `Datatable` instance is coerced to a string (when the datatable is printed out in a template). The template serves as the starting point for the client-side javascript to initialize.

The default template creates `<th>` headers that have a `data-name` attribute that is the slug of the column name for easy CSS targeting, and the default search and sort options that the `datatableview.js` initializer will read to build initialization options.

footer

Default `False`

Controls the existence of a `<tfoot>` element in the table. If `True`, the default `structure_template` will render another set of `<th>` elements with appropriate labels.

This is particularly useful when setting up something like per-column searching, which officially leverages the table footer, replacing each simple footer text label with a search box that applies only to that column's content.

result_counter_id

Default `'id_count'`

A helper setting that names a CSS `id` that the `datatableview.js` initializer will configure to hold a total result counter. This is strictly in addition to the normal readout that appears under a datatable. If you don't want any such external result display, you can ignore this setting.

labels

Default `{}`

A dict of model field names from `columns` that should have their `verbose_name` setting overridden for the table header.

Example `labels = {'name': "Headline"}`

processors = None

Default `{}`

A dict of model field names from `columns` that need to declare a `processor` callback. The mapped values may be direct references to callables, or strings that name a method on the Datatable or view.

Example `processors = {'name': 'get_name_data'}`

columns

Column

Available Columns

Model fields that subclass model fields shown here are automatically covered by these columns, which is why not all built-in model fields require their own column class, or are even listed in the handled classes.

TextColumn

IntegerColumn

FloatColumn

DateColumn

DateTimeColumn

BooleanColumn

DisplayColumn

CompoundColumn

forms

XEditableUpdateForm

The X-Editable mechanism works by sending events to the view that indicate the user's desire to open a field for editing, and their intent to save a new value to the active record.

The ajax `request.POST['name']` data field name that tells us which of the model fields should be targetted by this form. An appropriate formfield is looked up for that model field, and the `request.POST['value']` data will be inserted as the field's value.

helpers

The `helpers` module contains functions that can be supplied directly as a column's `processor`.

Callbacks need to accept the object instance, and arbitrary other `**kwargs`, because the `Datatable` instance will send it contextual information about the column being processed, such as the default value the column contains, the originating view, and any custom keyword arguments supplied by you from `preload_record_data()`.

link_to_model

make_boolean_checkmark

itemgetter

attrgetter

format_date

format

make_xeditable

make_processor

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

d

datatableview.columns, 18
datatableview.datatables, 15
datatableview.forms, 18
datatableview.helpers, 18
datatableview.views.base, 15
datatableview.views.legacy, 15
datatableview.views.xeditable, 15

C

cache_type (Meta attribute), 16
columns (Meta attribute), 16

D

datatableview.columns (module), 18
datatableview.datatables (module), 15
datatableview.forms (module), 18
datatableview.helpers (module), 18
datatableview.views.base (module), 15
datatableview.views.legacy (module), 15
datatableview.views.xeditable (module), 15

E

exclude (Meta attribute), 16

F

footer (Meta attribute), 17

H

hidden_columns (Meta attribute), 17

L

labels (Meta attribute), 17

M

Meta (class in datatableview.datatables), 16
model (Meta attribute), 16

O

ordering (Meta attribute), 16

P

page_length (Meta attribute), 16

R

result_counter_id (Meta attribute), 17

S

search_fields (Meta attribute), 16
structure_template (Meta attribute), 17

U

unsortable_columns (Meta attribute), 16