django-jinja-knockout Documentation

Release 2.1.0

Dmitriy Sintsov

Apr 24, 2023

Contents

1 Installation **3**
- 1.1 Virtual environment . . . 3
- 1.2 settings.py . . . 3
- 1.3 Context processor . . . 7
- 1.4 Middleware . . . 10
- 1.5 urls.py . . . 11
- 1.6 Templates . . . 11

2 Usage **13**
- 2.1 Key features overview . . . 13
- 2.2 Datatables . . . 13
- 2.3 Client-side . . . 13
- 2.4 admin.py . . . 14
- 2.5 forms.py / formsets.js . . . 14
- 2.6 management/commands/djk_seed.py . . . 14
- 2.7 middleware.py . . . 14
- 2.8 models.py . . . 15
- 2.9 query.py . . . 15
- 2.10 serializers.py . . . 15
- 2.11 tpl.py . . . 15
- 2.12 utils/sdv.py . . . 16
- 2.13 viewmodels.py . . . 16
- 2.14 views submodule . . . 16

3 Modules documentation **17**
- 3.1 Client-side support . . . 17
- 3.2 context_processors.py . . . 30
- 3.3 Datatables . . . 36
- 3.4 Grid configuration . . . 42
- 3.5 Action routing . . . 64
- 3.6 Standard actions . . . 70
- 3.7 ForeignKeyGridWidget . . . 87
- 3.8 MultipleKeyGridWidget . . . 92
- 3.9 Grids interaction . . . 94
- 3.10 Custom action types . . . 98
- 3.11 djk_ui . . . 101
- 3.12 Forms . . . 102

3.13 HTTP response . . . 113
3.14 Jinja2 macros . . . 113
3.15 Management commands . . . 119
3.16 middleware.py . . . 120
3.17 Models . . . 123
3.18 query.py . . . 124
3.19 tpl.py . . . 125
3.20 urls.py . . . 128
3.21 utils/mail.py . . . 131
3.22 utils/sdv.py . . . 133
3.23 Client-side viewmodels and AJAX response routing . . . 134
3.24 Built-in views . . . 153
3.25 widgets.py . . . 159

4 Datatable grids **163**

5 Contributing **165**
5.1 Types of Contributions . . . 165
5.2 Get Started! . . . 166
5.3 Pull Request Guidelines . . . 167

6 Credits **169**
6.1 Development Lead . . . 169
6.2 Contributors . . . 169

7 History **171**
7.1 0.1.0 . . . 171
7.2 0.2.0 . . . 171
7.3 0.3.0 . . . 171
7.4 0.4.0 . . . 172
7.5 0.4.1 . . . 174
7.6 0.4.2 . . . 174
7.7 0.4.3 . . . 175
7.8 0.5.0 . . . 175
7.9 0.6.0 . . . 175
7.10 0.7.0 . . . 176
7.11 0.8.0 . . . 177
7.12 0.8.1 . . . 177
7.13 0.8.2 . . . 178
7.14 0.9.0 . . . 178
7.15 1.0.0 . . . 178
7.16 2.0.0 . . . 179
7.17 2.1.0 . . . 179

Contents:

CHAPTER 1

Installation

- See README for the list of the currently supported Python / Django versions (master / development version), or the `README` for the specific release.
- Django template language is supported via including Jinja2 templates from DTL templates. Pure Jinja2 projects are supported as well.

1.1 Virtual environment

Inside virtualenv of your Django project, install *django-jinja-knockout*:

```
python3 -m pip install django-jinja-knockout
```

To install latest master from repository:

```
python3 -m pip install --upgrade git+https://github.com/Dmitri-Sintsov/django-jinja-
↪knockout.git
```

To install specific tag:

```
python3 -m pip install --upgrade git+https://github.com/Dmitri-Sintsov/django-jinja-
↪knockout.git@v2.0.0
```

1.2 settings.py

One may use existing example of settings.py as the base to develop your own `settings.py`.

1.2.1 DJK_APPS

`DJK_APPS` list is the subset of INSTALLED_APPS list that defines project applications which views will be processed by built-in ContextMiddleware class `process_view()` method via checking the result of `is_our_module()` method.

To apply *django-jinja-knockout* ContextMiddleware to the views of project apps, define `DJK_APPS` list with the list of Django project's own applications like that:

```
DJK_APPS = (
    'djk_sample',
    'club_app',
    'event_app',
)
```

It increases the compatibility with external apps which views do not require to be processed by *django-jinja-knockout* ContextMiddleware.

Add `DJK_APPS` (if there is any) and `django_jinja_knockout` to INSTALLED_APPS in `settings.py`:

```
OPTIONAL_APPS = []

try:
    import django_deno
    # django_deno is an optional Javascript module bundler, not required to run in␣
↪modern browsers
    OPTIONAL_APPS.append('django_deno')
except ImportError:
    pass

# Order of installed apps is important for Django Template loader to find 'djk_sample/
↪templates/base.html'
# before original allauth 'base.html' is found, when allauth DTL templates are used␣
↪instead of built-in
# 'django_jinja_knockout._allauth' Jinja2 templates, thus DJK_APPS are included␣
↪before 'allauth'.
#
# For the same reason, djk_ui app is included before django_jinja_knockout, to make␣
↪it possible to override
# any of django_jinja_knockout template / macro.
    INSTALLED_APPS = [
        'django.contrib.admin',
        'django.contrib.auth',
        'django.contrib.contenttypes',
        'django.contrib.sessions',
        'django.contrib.messages',
        'django.contrib.staticfiles',
        # 'sites' is required by allauth
        'django.contrib.sites',
    ] + OPTIONAL_APPS + [
        'djk_ui',
        'django_jinja_knockout',
        'django_jinja_knockout._allauth',
    ] + DJK_APPS + [
        'allauth',
        'allauth.account',
        # Required for socialaccount template tag library despite we do not use␣
↪social login
```

(continues on next page)

(continued from previous page)

```
        'allauth.socialaccount',
    ]
```

djk_ui app provides pluggable support for Bootstrap 3 / Bootstrap 4.

django_deno may be included to `OPTIONAL_APPS` to provide es6 modules / terser / SystemJS support via deno rollup. See sample project settings.py for the example of actual django_deno configuration.

django-allauth support is not mandatory but optional; just remove the following apps from INSTALLED_APPS in case you do not need it:

```
# The Django sites framework is required for 'allauth'
'django.contrib.sites',
'allauth',
'allauth.account',
'allauth.socialaccount',
'django_deno`,
'django_jinja_knockout._allauth',
```

Built-in allauth DTL templates are supported without any modification. In such case the next module may be removed from the list of INSTALLED_APPS as well:

```
'django_jinja_knockout._allauth',
```

- It is possible to extend *django-jinja-knockout* ContextMiddleware to add new functionality. See djk_sample.ContextMiddleware code for example.

1.2.2 DJK_MIDDLEWARE

apps.DjkAppConfig class has .get_context_middleware() method which should be invoked to get extended middleware class to be used by django-jinja-knockout code and across the project. In case one's project has a middleware extended from django-jinja-knockout middleware, one should specify it import string as `DJK_MIDDLEWARE` variable value in `settings.py` like that:

```
DJK_MIDDLEWARE = 'djk_sample.middleware.ContextMiddleware'
```

1.2.3 FILE_MAX_SIZE

This optional setting allows to specify maximal allowed file size to upload with AjaxForm class:

```
FILE_UPLOAD_HANDLERS = ("django.core.files.uploadhandler.TemporaryFileUploadHandler",)
FILE_MAX_SIZE = 100 * 1024 * 1024
```

1.2.4 LAYOUT_CLASSES

This optional setting allows to override default Bootstrap grid layout classes for bs_form() and bs_inline_formsets() Jinja2 macros used to display `ModelForm` and inline formsets in the *django-jinja-knockout* code. The default value is specified in `djk_ui` app `conf` module, but can be overridden in settings.py:

```
LAYOUT_CLASSES = {
    '': {
        'label': 'col-md-4',
```

(continues on next page)

(continued from previous page)

```
        'field': 'col-md-6',
    },
    'display': {
        'label': 'w-30 table-light',
        'field': 'w-100 table-default',
    },
}
```

1.2.5 OBJECTS_PER_PAGE

Allows to specify default limit for Django paginated querysets for `ListSortingView` / `KoGridView` (see views submodule):

```
# Pagination settings.
OBJECTS_PER_PAGE = 3 if DEBUG else 10
```

1.2.6 USE_JS_TIMEZONE

Optional boolean value (by default is `False`). When `True`, ContextMiddleware class `process_request()` method will autodetect Django timezone from current browser session timezone.

1.2.7 Javascript errors logger

Since version 0.7.0 it's possible to setup Javascript logger which would either display Javascript errors in Bootstrap dialog, or will report these via email to site admins whose emails are specified by settings.ADMINS:

```
ADMINS = [('John Smith', 'user@host.com'),]
if DEBUG:
    # Javascript error will display Bootstrap dialog.
    JS_ERRORS_ALERT = True
else:
    # Javascript error will be reported via ADMINS emails.
    JS_ERRORS_LOGGING = True
```

1.2.8 Context processors

Add *django_jinja_knockout* TemplateContextProcessor to settings.py:

```
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))

TEMPLATES = [
    {
        "BACKEND": "django.template.backends.jinja2.Jinja2",
        "APP_DIRS": True,
        "OPTIONS": {
            'environment': 'django_jinja_knockout.jinja2.environment',
            'context_processors': [
                'django.template.context_processors.i18n',
                'django_jinja_knockout.context_processors.template_context_processor'
            ]
```

(continues on next page)

(continued from previous page)

```
        },
    },
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
                # Next line is required only if project uses Django templates (DTL).
                'django_jinja_knockout.context_processors.template_context_processor'
            ],
        },
    },
]
```

1.2.9 DJK_CLIENT_ROUTES

If you want to use built-in server-side to client-side global route mapping, use `DJK_CLIENT_ROUTES` settings:

```
# List of global client routes that will be injected into every view (globally).
# This is a good idea if some client-side route is frequently used by most of views.
# Alternatively one can specify client route url names per view (see the␣
↪documentation).
# Second element of each tuple defines whether the client-side route should be␣
↪available to anonymous users.
DJK_CLIENT_ROUTES = {
    ('user_change', True),
    ('equipment_grid', True),
}
```

1.3 Context processor

Context processor makes possible to specify client-side routes per view:

```
from django_jinja_knockout.views import page_context_decorator

@page_context_decorator(client_routes={
    'blog_feed',
    'my_grid_url_name',
})
def my_view(request):
    return TemplateResponse(request, 'template.htm', {'data': 12})
```

and per class-based view:

```
from django_jinja_knockout.views import PageContextMixin

class MyView(PageContextMixin)
```

(continues on next page)

(continued from previous page)

```
    client_routes = {
        'blog_feed',
        'my_grid_url_name',
    }
```

for `urls.py` like this:

```
from django_jinja_knockout.urls import UrlPath
from my_blog.views import feed_view
# ...
re_path(r'^blog-(?P<blog_id>\d+)/$', feed_view, name='blog_feed',
    kwargs={'ajax': True, 'permission_required': 'my_blog.add_feed'}),
UrlPath(MyGrid)(
    name='my_grid_url_name',
    base='my-grid',
    kwargs={'view_title': 'My Sample Grid'}
),
```

to make the resolved url available in client-side scripts.

In such case defining *DJK_CLIENT_ROUTES* is not necessary, however one has to specify required client-side url names in every view which includes Javascript template that accesses these url names (for example foreign key widgets of datatables require resolved url names of their view classes).

The current url generated for `'blog_feed'` url name will be available at client-side Javascript as:

```
import { Url } from '../../djk/js/url.js';

Url('blog_feed', {'blog_id': 1});
```

One will be able to call Django view via AJAX request in your Javascript code like this:

```
import { AppGet, AppPost } from '../../djk/js/url.js';

AppPost('blog_feed', {'postvar1': 1, 'postvar2': 2}, {
    kwargs: {'blog_id': 1}
});
AppGet('blog_feed', {'getvar1': 1}, {
    kwargs: {'blog_id': 1}
});
```

where the AJAX response will be treated as the list of viewmodels and will be automatically routed by url.js to appropriate viewmodel handler. Django exceptions and AJAX errors are handled gracefully, displayed in `BootstrapDialog` window by default.

1.3.1 Extending context processor

Extending context processor is useful when templates should receive additional context data by default:

```
from django_jinja_knockout.context_processors import TemplateContextProcessor as␣
↪BaseContextProcessor
from my_project.tpl import format_currency, static_hash

class TemplateContextProcessor(BaseContextProcessor):
```

(continues on next page)

(continued from previous page)

```
    def get_context_data(self):
        context_data = super().get_context_data()
        # Add two custom function to template context.
        context_data.update({
            'format_currency': format_currency,
            'static_hash': static_hash,
        })
        return context_data
```

- See djk_sample.TemplateContextProcessor source code for the trivial example of extending *django-jinja-knockout* TemplateContextProcessor.

1.3.2 DJK_PAGE_CONTEXT_CLS

DJK_PAGE_CONTEXT_CLS setting allows to override default PageContext class:

```
DJK_PAGE_CONTEXT_CLS = 'djk_sample.context_processors.PageContext'
```

That makes possible to add custom client configuration to page_context instance:

```
from django.conf import settings
from django_jinja_knockout.context_processors import PageContext as BasePageContext

class PageContext(BasePageContext):

    def get_client_conf(self):
        client_conf = super().get_client_conf()
        client_conf.update({
            # v2.1.0 - enable built-in custom tags (by default is off)
            'compatTransformTags': True,
            'email_host': settings.EMAIL_HOST,
            'userName': '' if self.request.user.id == 0 else self.request.user.
↪username,
        })
        return client_conf
```

which will be available in Javascript as:

```
import { AppConf } from '../../djk/js/conf.js';

// used by djk_ui ui.js
AppConf('compatTransformTags')
AppConf('email_host')
AppConf('userName')
```

Note that client conf is added globally, while the client data are added per view:

```
from django_jinja_knockout.views import create_page_context

def my_view(request, **kwargs):
    page_context = create_page_context(request=request)
    page_context.update_client_data({'isVerifiedUser': True})
```

to be queried later in Javascript:

```
import { AppClientData } from '../../djk/js/conf.js';

AppClientData('isVerifiedUser')
```

1.4 Middleware

Key functionality of `django-jinja-knockout` middleware is:

- Setting current Django timezone via browser current timezone.
- Getting current request in non-view functions and methods where Django provides no instance of request available.
- Checking `DJK_APPS` applications views for the permissions defined as values of kwargs argument keys in urls.py `re_path()` calls:
- `'allow_anonymous' key` - `True` when view is allowed to anonymous user (`False` by default).
- `'allow_inactive' key` - `True` when view is allowed to inactive user (`False` by default).
- `'permission_required' key` - value is the name of Django app / model permission string required for this view to be called.

All of the keys are optional but some have restricted default values.

Install `django_jinja_knockout.middleware` into settings.py:

```
MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.auth.middleware.SessionAuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    'django.middleware.security.SecurityMiddleware',
    'django_jinja_knockout.middleware.ContextMiddleware',
)
```

Then to use it in a project:

```
from django_jinja_knockout.middleware import ContextMiddleware
```

For example to get current request in non-view functions and methods, one may use:

```
ContextMiddleware.get_request()
```

and to get current request user:

```
ContextMiddleware.get_request().user
```

- Do not forget that request is mocked when running in console, for example in management jobs. It is possible to override the middleware class for custom mocking.

1.4.1 Extending middleware

It's possible to extend built-in ContextMiddleware. In such case *DJK_MIDDLEWARE* string in settings.py should contain full name of the extended class. See djk_sample.ContextMiddleware for the example of extending middleware to enable logging of Django models performed actions via content types framework.

1.5 urls.py

The example of urls.py for Jinja2 `_allauth` templates:

```
# More pretty-looking but possibly not compatible with arbitrary allauth version:
re_path(r'^accounts/', include('django_jinja_knockout._allauth.urls')),
```

The example of urls.py for DTL `allauth` templates:

```
# Standard allauth DTL templates working together with Jinja2 templates via {% load␣
↪jinja %}
re_path(r'^accounts/', include('allauth.urls')),
```

Note that `accounts` urls are not processed by the default *DJK_MIDDLEWARE* thus do not require `is_anonymous` or `permission_required` kwargs keys to be defined.

The example of *DJK_MIDDLEWARE* view urls.py with the view title value and with permission checking (anonymous / inactive users are not allowed by default):

```
from django_jinja_knockout.urls import UrlPath
UrlPath(EquipmentGrid)(
    name='equipment_grid',
    kwargs={
        'view_title': 'Grid with the available equipment',
        'permission_required': 'club_app.change_manufacturer'
    }
),
```

1.6 Templates

1.6.1 Integration of django-jinja-knockout into existing Django / Bootstrap project

If your project base template uses `Jinja2` templating language, there are the following possibilities:

- Extend your `base.htm` template from jinja2/base_min.htm (bs3) / jinja2/base_min.htm (bs4) template.
- Include styles from jinja2/base_head.htm and scripts from jinja2/base_bottom_scripts.htm. These are required to run client-side scripts like app.js and grid.js.

If your project base template uses Django Template Language (`DTL`), there are the following possibilities:

- Extend your `base.html` template from templates/base_min.html (bs3) / templates/base_min.html (bs4) template.
- To ensure that page_context is always available in DTL template:

```
{% load page_context %}
{% init_page_context %}
```

- Include styles from jinja2/base_head.htm and scripts from jinja2/base_bottom_scripts.htm via `{% load jinja %}` template tag library to your `DTL` template:

```
{% load jinja %}
{% jinja 'base_head.htm' %}
{% if messages %}
    {% jinja 'base_messages.htm' %}
{% endif %}
{% jinja 'base_bottom_scripts.htm' %}
```

Do not forget that Jinja2 does not support extending included templates.

Template engines can be mixed with inclusion of Jinja2 templates from DTL templates like this:

```
{% jinja 'bs_navs.htm' with _render_=1 navs=main_navs %}
{% jinja 'bs_inline_formsets.htm' with _render_=1 related_form=form formsets=formsets␣
↪action=view.get_form_action_url opts=view.get_bs_form_opts %}
{% jinja 'bs_list.htm' with _render_=1 view=view object_list=object_list is_
↪paginated=is_paginated page_obj=page_obj %}
{% jinja 'ko_grid.htm' with _render_=1 grid_options=club_grid_options %}
{% jinja 'ko_grid_body.htm' with _render_=1 %}
```

See club_app/templates for full-size examples of including Jinja2 templates from DTL templates.

CHAPTER 2

Usage

The best way to understand how to use django-jinja-knockout features is to install djk_sample sample project and to examine it's source code. The code of sample project is compact enough, while many of features are covered.

2.1 Key features overview

2.2 Datatables

The package includes server-side (Python) and client-side (Javascript) code to quickly create easy to use datatables with standard and custom actions for Django models, including adding, editing, deleting.

See datatables for more info.

2.3 Client-side

There are lots of client-side Javascript included into the package. It includes ready to use components such as:

- Django ModelForm / Formset AJAX dialogs.
- Django models AJAX datatables.
- Nested templating with custom tags.
- Client-side widget support.
- AJAX viewmodels.

See clientside for more info.

2.4 admin.py

- `DjkAdminMixin` - optionally inject css / scripts into django.admin to support widgets.OptionalInput.
- `ProtectMixin` - allow only some model instances to be deleted in django.admin.
- `get_admin_url` - make readonly foreignkey field to be rendered as link to the target model admin change view.
- `get_model_change_link` - generates the link to django admin model edit page.

2.5 forms.py / formsets.js

- Renderers for forms / formsets / form fields.
- AJAX form processing.
- Display read-only "forms" (model views).
- `BootstrapModelForm` - Form with field classes stylized for Bootstrap. Since version 0.4.0 it also always has `request` attribute for convenience to be used in `clean()` method and so on.
- `DisplayModelMetaclass` - Metaclass used to create read-only "forms", to display models as html tables.
- `WidgetInstancesMixin` - Provides model instances bound to `ModelForm` in field widgets. It helps to make custom `get_text_fn` / `get_text_method` callbacks for `DisplayText` form widgets .
- `set_knockout_template` - Monkey-patching methods for formset to support knockout.js version of `empty_form`. Allows to dynamically add / remove new forms to inline formsets, including third-party custom fields with inline Javascript (such as AJAX populated html selects, rich text edit fields).
- `FormWithInlineFormsets` - Layer on top of related form and it's many to one multiple formsets. GET / CREATE / UPDATE. Works both in function views and in class-based views (CBVs).
- `SeparateInitialFormMixin` - Mixed to `BaseInlineFormset` to use different form classes for already existing model objects and for newly added ones (empty_form). May be used with `DisplayModelMetaclass` to display existing forms as read-only, while making newly added ones editable.
- `CustomFullClean` / `StripWhitespaceMixin` mixins for Django forms.

See forms for the detailed explanation.

2.6 management/commands/djk_seed.py

`djk_seed` management command allows to execute post-migration database seeds for specified Django app / model.

See management_commands for more info.

2.7 middleware.py

- Middleware is extendable (inheritable).
- Client-side viewmodels via AJAX result and / or injected into html page / user session.
- Automatic timezone detection and timezone activation from the browser.
- request.custom_scripts for dynamic injection of client-side scripts.

- DJK_APPS views require permission defined in urls.py by default, which increases the default security.
- Request mock-up.
- Mini-router.

See middleware for more info.

2.8 models.py

- Get users with specific permissions.
- Get related fields / related field values.
- Model class / model instance / model fields metadata retrieval.
- `model_values()` to get queryset `.values()` like dict for single Django model object instance.

See models for more info.

2.9 query.py

- Allows to create raw Django querysets with filter methods such as filter() / order_by() / count().
- Allows to convert Python lists to Django-like querysets, which is useful to filter the data received via prefetch_related Django ORM reverse relation query.

It makes possible to use raw SQL queries and Python lists as the arguments of datatables / filtered lists. See query.py for more info.

2.10 serializers.py

Nested serializer for Django model instances with localization / internationalisation. Note that the serializer is written to create logs / archives of model object changes. It's unused by built-in viewmodels / datatables. Datatables use get_str_fields() model serialization method instead.

2.11 tpl.py

- Renderer class for recursive object context rendering.
- PrintList class for nested formatting of Python structures. Includes various formatting wrapper functions.
- HTML / CSS manipulation in Python.
- Date / url / JSON formatting.
- Model / content type links formatters.
- discover_grid_options() enables to embed datatables into arbitrary HTML DOM subtrees.

See tpl for more info.

2.12 utils/sdv.py

Low-level helper functions:

- Class / model helpers.
- Debug logging.
- Iteration.
- Nested data structures access.
- String conversion.

See utils.sdv for more info.

2.13 viewmodels.py

Server-side Python functions and classes to manipulate lists of client-side viewmodels. Mostly are used with AJAX JSON responses and in `app.js` client-side response routing. Read viewmodels documentation for more info.

2.14 views submodule

- Permission / view title kwargs.
- `FormWithInlineFormsetsMixin` - view / edit zero or one ModelForm with one or many related formsets. Supports dynamic formset forms via `formsets.js` and `set_knockout_template` patching.
- `BsTabsMixin` - insert additional context data to support Bootstrap navbars.
- `PageContextMixin` - provides additional template context required to run client-side of the framework.
- `ListSortingView` - non-AJAX filtered / sorted ListView, with partial support of AJAX `KoGridView` settings.
- AJAX views: `ActionsView` / `ModelFormActionsView` / `KoGridView`

See views for the detailed explanation.

CHAPTER 3

Modules documentation

3.1 Client-side support

3.1.1 es6 module loader

Since v2.0, the monolithic app.js which used global `App` container, was refactored into es6 modules, which makes the client-side development more flexible. The modules themselves still use es5 syntax, with the exception of es6 imports / exports. To run the code in outdated browser which does not support es6 modules (eg IE11), django_deno bundling app should be used. It also has optional terser support. There is sample django_deno config (see djk-sample settings.py for full working example):

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # 'sites' is required by allauth
    'django.contrib.sites',
    # django_deno is an optional Javascript module bundler, not required to run in␣
↪modern browsers
    'django_deno',
] + DJANGO_JINJA_APPS + [
    'djk_ui',
    'django_jinja_knockout',
    'django_jinja_knockout._allauth',
] + DJK_APPS + [
    'allauth',
    'allauth.account',
    # Required for socialaccount template tag library despite we do not use social␣
↪login
    'allauth.socialaccount',
```

(continues on next page)

(continued from previous page)

```
]

DENO_ROLLUP_ENTRY_POINTS = [
    'sample/js/app.js',
    'sample/js/club-grid.js',
    'sample/js/member-grid.js',
]

DENO_ROLLUP_BUNDLES = {
    'djk': {
        'writeEntryPoint': 'sample/js/app.js',
        'matches': [
            'djk/js/*',
            'djk/js/lib/*',
            'djk/js/grid/*',
        ],
        'excludes': [],
        'virtualEntryPoints': 'matches',
        'virtualEntryPointsExcludes': 'excludes',
    },
}

# Do not forget to re-run collectrollup management command after changing rollup.js␣
↪bundles module type:
DENO_OUTPUT_MODULE_TYPE = 'module' if DEBUG else 'systemjs-module'
DJK_JS_MODULE_TYPE = DENO_OUTPUT_MODULE_TYPE

# Run $VIRTUAL_ENV/djk-sample/cherry_django.py to check the validity of collectrollup␣
↪command output bundle.
DENO_ROLLUP_COLLECT_OPTIONS = {
    'terser': True,
}

DENO_ENABLE = True
DENO_DEBUG = False
DENO_RELOAD = False
```

Old browsers such as IE11 will use bundled system.js loader. Note that modern browsers do not require any bundling at all, however could benefit from optional generating terser-optimized es6 bundles.

3.1.2 Client-side entry points

See *Injection of custom script urls into loaded page* how to add custom Javascript entry points.

- DENO_ROLLUP_ENTRY_POINTS specifies optional rollup entry points
- set_custom_scripts specifies browser entry points.

Note that `DENO_ROLLUP_ENTRY_POINTS` setting is optional and is used only when django_deno is installed and enabled in `settings.py` to generate the minified bundle and / or to generate IE11 compatible bundle.

Client-side modules include the following features:

- *Viewmodels (client-side response routing)*
- *Underscore.js templates*
- Components

- *Multiple level Javascript class inheritance*
- Dialog BootstrapDialog wrapper.

3.1.3 Client-side initialization

There are two different hooks / methods of client-side initialization:

- documentReadyHooks - the list of function handlers which are called via `$(document).ready()` event handler, so these do not interfere with the third party scripts code.
- initClientHooks - the ordered list of function handlers applied to content generated by the viewmodels / Underscore.js / Knockout.js templates to provide the dynamic styles / event handlers / client-side components. It's processed via calling initClient function. OrderedHooks class instance is used to add hooks in proper order, where the component initialization hook should always be executed at the last step.

Read more about viewmodels here: *Client-side viewmodels and AJAX response routing*.

It supports mandatory 'init' and optional 'dispose' types of handlers for the DOM subtrees, where 'dispose' handlers are called in the reverse order. It's also possible to define custom types of handlers.

To add new client-side initialization handlers of the 'init' / 'dispose' types:

```
import { initClientHooks } from '../../djk/js/initclient.js';

initClientHooks.add({
    init: function($selector) {
        $selector.myPlugin('init');
    },
    dispose: function($selector) {
        $selector.myPlugin('dispose');
    }
});
```

To add only the 'init' type of handler (when disposal is not needed):

```
import { initClientHooks } from '../../djk/js/initclient.js';

initClientHooks.add(function($selector) {
    $selector.myPlugin('init');
});
```

To call all the chain of 'init' handlers:

```
import { initClient } from '../../djk/js/initclient.js';

initClient($selector);
```

To call all the chain of 'dispose' handlers:

```
import { initClient } from '../../djk/js/initclient.js';

initClient($selector, 'dispose');
```

Note that the handlers usually are called automatically, except for grid rows where one has to use grid .useInitClient option to enable .initClient() call for grid rows DOM. See *Datatables* for more info.

Custom `'formset:added'` jQuery event automatically supports client initialization, eg form field classes / form field event handlers when the new form is added to inline formset dynamically.

3.1.4 Viewmodels (client-side response routing)

See *Client-side viewmodels and AJAX response routing* for the detailed explanation.

- Separates AJAX calls from their callback processing, allowing to specify AJAX routes in button html5 data attributes not having to define implicit DOM event handler and implicit callback.
- Allows to write more modular Javascript code.
- Client-side view models can also be executed in Javascript directly.
- Possibility to optionally inject client-side viewmodels into html pages, executing these on load.
- Possibility to execute client-side viewmodels from current user session (persistent onload).
- vmRouter - predefined built-in AJAX response viewmodels router to perform standard client-side actions, such as displaying BootstrapDialogs, manipulate DOM content with graceful AJAX errors handling. It can be used to define new viewmodel handlers.

Simplifying AJAX calls

- Url - mapping of Django server-side route urls to client-side Javascript.
- AjaxButton - automation of button click event AJAX POST handling for Django.
- AjaxForm - Django form AJAX POST submission with validation errors display via response client-side viewmodels.

  Requires `is_ajax=True` argument of *bs_form()* / *bs_inline_formsets()* Jinja2 macros.

  The whole process of server-side to client-side validation errors mapping is performed by the built-in *FormWithInlineFormsetsMixin* class `.form_valid()` / `form_invalid()` methods.

  Supports multiple Django POST routes for the same AJAX form via multiple `input[type="submit"]` buttons in the generated form html body.
- AppGet / AppPost automate execution of AJAX POST handling for Django using named urls like `url(name='my_url_name')` exported to client-side code directly.

3.1.5 Global IoC

Since v2.0, monolithic `App.readyInstances` was replaced by globalIoc instance of ViewModelRouter class, which holds lazy definitions of global instances initialized when browser document is loaded. It allows to override built-in global instances and to add custom global instances in user scripts (usually in the *Client-side entry points*) like this:

```
import { globalIoc } from '../../djk/js/ioc.js';

// Late initialization allows to patch / replace classes in user scripts.
globalIoc.add('UserClass', function(options) {
    return new UserClass(options);
});

// To check whether the class name was already registered:
globalIoc.hasView('UserClass');

// To add custom class just once:
globalIoc.once('UserClass', function(options) {
```

(continues on next page)

(continued from previous page)

```
    return new UserClass(options);
});
```

3.1.6 Component IoC

- Components use the similar componentIoc instance of ViewModelRouter class for the client-side Javascript class registration. See componentIoc sample for the complete example. There is also dialogIoc used by Dialog component and gridActionIoc used by *Datatables*, which allows to optionally override their functionality.

Base example:

```
import { componentIoc } from '../../djk/js/ioc.js';

function UserComponentClass(options) {
    // ... skipped ...
};

componentIoc.add('UserComponentClass', function(options) {
    return new UserComponentClass(options);
});
```

- See *clientside_components*, *Built-in views*, *widgets.py*, *Client-side viewmodels and AJAX response routing* for the examples how to specify custom component class name at server-side via data-component-class html5 attribute.

3.1.7 Client-side localization

It's possible to format Javascript translated messages with Trans function:

```
import { Trans } from '../../djk/js/translate.js';

Trans('Yes')
Trans('No')
Trans('Close')
Trans('Delete "%s"', formModelName)
// named arguments
Trans(
    'Too big file size=%(size)s, max_size=%(maxsize)s',
    {'size': file.size, 'maxsize': maxSize}
)
// with html escape
Trans('Undefined viewModel.view %s', $.htmlEncode(viewModelStr))
```

Automatic translation of html text nodes with `localize-text` class is performed with localize by *Client-side initialization*

```
<div class="localize-text">Hello, world in your language!</div>
```

- See Internationalization in JavaScript code how to setup Javascript messages catalog in Django.
- Internally, sprintf library and Trans is used to convert messages to local versions.
- See bs_range_filter.htm source for the complete example.

3.1.8 Underscore.js templates

Underscore.js templates can be autoloaded as Dialog modal body content. Also they are used in conjunction with Knockout.js templates to generate components, for example AJAX grids (Django datatables).

Template processor is implemented as Tpl class. It's possible to extend or to replace template processor class by calling globalIoc factory method:

```
import { propGet } from '../../djk/js/prop.js';
import { inherit } from '../../djk/js/dash.js';
import { Tpl } from '../../djk/js/tpl.js';
import { globalIoc } from '../../djk/js/ioc.js';

globalIoc.removeAll('Tpl').add('Tpl', function(options) {
    var _options = $.extend({}, options);
    if (propGet(_options, 'meta_is_ie')) {
        return new IeTpl(_options);
    } else {
        return new Tpl(_options);
    }
});

IeTpl = function(options) {
    inherit(Tpl.prototype, this);
    return this.init(options);
};
```

Such custom template processor class could override one of the (sub)templates loading methods such as `.expandTemplate()` or `.compileTemplate()`.

In the underscore.js template execution context, the instance of Tpl class is available as `self` variable. Thus calling Tpl class `.get('varname')` method is performed as `self.get('varname')`. See ko_grid_body() templates for the example of `self.get` method usage.

Internally template processor is used for optional client-side overriding of default grid templates, supported via Tpl constructor `options.templates` argument.

- compileTemplate provides singleton factory for compiled underscore.js templates from `<script>` tag with specified DOM id `tplId`.
- Tpl.domTemplate converts single template with specified DOM id and template arguments into jQuery DOM subtree.
- Tpl.loadTemplates recursively loads existing underscore.js templates by their DOM id into DOM nodes with html5 `data-template-id` attributes for specified `$selector`.
- bindTemplates - templates class factory used by initClient auto-initialization of DOM nodes.

The following html5 data attributes are used by Tpl template processor:

- `data-template-id` - destination DOM node which will be replaced by expanded underscore.js template with specified template id. Attribute can be applied recursively.
- `data-template-class` - optional override of default Tpl template processor class. Allows to process different underscore.js templates with different template processor classes.
- `data-template-args` - optional values of current template processor instance `.extendData()` method argument. This value will be appended to `.data` property of template processor instance. The values stored in `.data` property are used to control template execution flow via `self.get()` method calls in template source code.

- `data-template-args-nesting` - optionally disables appending of `.data` property of the parent template processor instance to `.data` property of current nested child template processor instance.
- `data-template-options` - optional value of template processor class constructor `options` argument, which may have the following keys:
  - `.data` - used by Tpl class `.get()` method to control template execution flow.
  - `.templates` - key map of template ids to optionally substitute some or all of template names.

Template attributes merging

The DOM attributes of the template holder tag different from `data-template-*` are copied to the root DOM node of the expanded template. This allows to get the rid of template wrapper when using the templates as the foundation of components. For example datatables / grid templates do not use separate wrapper tag anymore and thus become simpler.

Custom tags

- Since v2.1.0 built-in custom tags are replaced by *Custom elements*, disabled by default and may be removed in the future.

The built-in template processor supports custom tags via TransformTags Javascript class `applyTags()` method. By default there are the `CARD-*` tags registered, which are transformed to Bootstrap 4 / 5 cards or to Bootstrap 3 panels, depending on the *djk_ui* version.

Custom tags are also applied via initClient to the loaded DOM page and to dynamically loaded AJAX DOM fragments. However because the custom tags are not browser-native, such usage of custom tags is not recommended as extra flicker may occur. Such flicker never occurs in built-in *Underscore.js templates*, because the template tags are substituted before they are attached to the page DOM.

It's possible to add new custom tags via supplying the capitalized `tagName` argument and function processing argument `fn` to TransformTags class `add()` method.

To enable built-in custom tags, set `AppConf('compatTransformTags')` value to `True` via custom page context.

See *DJK_PAGE_CONTEXT_CLS* for more detailed explanation how to set custom client conf values.

3.1.9 Custom elements

Since v2.1.0, built-in Elements class allows to create custom elements in es5 syntax:

```
import { elements } from './elements.js';

elements.newCustomElements(
    {
        ancestor: HTMLDivElement,
        name: 'form-group',
        extendsTagName: 'div',
        classes: ['form-group'],
    },
    {
        ancestor: HTMLLabelElement,
        name: 'form-label',
        extendsTagName: 'label',
```

(continues on next page)

(continued from previous page)

```
        classes: ['control-label'],
    }
)
```

See Elements.builtInProperties list for the description of available elements options.

Custom elements also can be used to simplify creation of Components. For example in document.js:

```
import { elements } from './elements.js';

elements.newCustomElements(
    {
        name: 'list-range-filter',
        defaultAttrs: {
            'data-component-class': 'ListRangeFilter',
        }
    },
    {
        name: 'ko-grid',
        defaultAttrs: {
            'data-component-class': 'Grid',
            'data-template-id': 'ko_grid_body',
        }
    },
);
```

Note that `<list-range-filter>` component does not use the default template, while `<ko-grid>` component does use it.

3.1.10 Components

Components class allows to automatically instantiate Javascript classes by their componentIoc string path specified in element's data-component-class html5 attribute and bind these to that element. It is used to provide Knockout.js `Grid` component auto-loading / auto-binding, but is not limited to.

Components can be also instantiated via target element event instead of document 'ready' event. To enable that, define `data-event` html5 attribute on target element. For example, to bind component classes to button 'click' / 'hover':

```
<button class="component"
    data-event="click"
    data-component-class="GridDialog"
    data-component-options='{"filterOptions": {"pageRoute": "club_member_grid"}}'>
    Click to see project list
</button>
```

When target button is clicked, GridDialog class registered by componentIoc will be instantiated with `data-component-options` value passed as it's constructor argument.

JSON string value of `data-component-options` attribute can be nested object with many parameter values, so for convenience it can be generated in Jinja2 macro, such as ko_grid() See the example of overriding two default templates in cbv_grid_breadcrumbs.htm:

```
{{
ko_grid(
    grid_options={
        'pageRoute': view.request.resolver_match.view_name,
```

(continues on next page)

(continued from previous page)

```
        'pageRouteKwargs': view.kwargs,
    },
    dom_attrs={
        'data-template-options': {
            'templates': {
                'ko_grid_filter_choices': 'ko_grid_breadcrumb_filter_choices',
                'ko_grid_filter_popup': 'ko_grid_breadcrumb_filter_popup',
            }
        },
    }
)
}}
```

By default, current component instance is re-used when the same event is fired multiple times. To have component re-instantiated, one should save target element in component instance like this:

```
MyComponent.runComponent = function(elem) {
    this.componentElement = elem;
    // Run your initialization code here ...
    this.doStuff();
};
```

Then in your component shutdown code call components instance `.unbind()` method, then `.add()` method:

```
import { components } from '../../djk/components.js';

MyComponent.onHide = function() {
    // Run your shutdown code ...
    this.doShutdown();
    // Detect component, so it will work without component instantiation too.
    if (this.componentElement !== null) {
        // Unbind component.
        var desc = components.unbind(this.componentElement);
        if (typeof desc.event !== 'undefined') {
            // Re-bind component to the same element with the same event.
            components.add(this.componentElement, desc.event);
        }
    }
};
```

There is built-in $.component plugin, which allows to get the Javascript component instance bound to particular DOM element. It returns either an component object, *null* when there is no bound component, or an instance of Promise to resolve the lazy loaded component, see $.component promise.

See *Component IoC* how to register custom Javascript component class.

See GridDialog code for the example of built-in component, which allows to fire AJAX datatables via click events.

Because GridDialog class constructor may have many options, including dynamically-generated ones, it's preferable to generate `data-component-options` JSON string value in Python / Jinja2 code (see *String formatting*).

Search for data-component-class in djk-sample code for the examples of both document ready and button click component binding.

Components use ComponentManager class which provides the support for nested components and for sparse components.

Nested components

It's possible to nest component DOM nodes recursively unlimited times:

```
<div class="component" data-component-class="Grid">
    <input type="button" value="Grid button" data-bind="click: onClick()">
    <div class="component" data-component-class="MyComponent">
        <input type="button" value="My component button" data-bind="click: onClick()">
    </div>
</div>
```

The Knockout.js `click` bindings of the `Grid button` will be directed to `Grid` class instance `onClick()` method and from the `My component button` to `MyComponent` class instance `onClick()` method.

Note that to achieve nested binding, DOM subtrees of nested components are detached until the outer components are run. Thus, in case the outer component is run on some event, for example `data-event="click"`, nested component nodes will be hidden until outer component is run via the click event. Thus it's advised to think carefully when using nested components running on events, while the document ready nested components have no such possible limitation.

The limitation is not so big, however because most of the components have dynamic content populated only when they run.

See the demo project example of nested datatable grid component: member_grid_tabs.htm.

Sparse components

In some cases the advanced layout of the page requires one component to be bound to the multiple separate DOM subtrees of the page. In such case sparse components may be used. To specify sparse component, add data-component-selector HTML attribute to it with the jQuery selector that should select sparse DOM nodes bound to that component.

Let's define the datatable grid:

```
{{
    ko_grid(
        grid_options={
            'classPath': 'ClubEditGrid',
            'pageRoute': 'club_edit_grid',
            'pageRouteKwargs': {'club_id': view.kwargs['club_id']},
        },
        dom_attrs={
            'id': 'club_edit_grid',
            'class': 'club-edit-grid',
            'data-component-selector': '.club-edit-grid',
        }
    )
}}
```

Let's define separate row list and the action button to add new row for this grid located in arbitrary location of the page:

```
<div class="club-edit-grid">
    <div data-bind="visible:gridRows().length > 0" style="display: none;">
        <h3>Grid rows:</h3>
        <ul class="auto-highlight" data-bind="foreach: {data: $('#club_edit_grid').
↪component().gridRows, as: 'row'}">
            <li>
```

(continues on next page)

(continued from previous page)

```
                <a data-bind="text: row.displayValues.name, attr: {href: getUrl(
↪'member_detail', {member_id: row.values.member_id})}"></a>
            </li>
        </ul>
    </div>
</div>
<div>This div is the separate content that is not bound to the component.</div>
<div class="club-edit-grid">
    <button class="btn-choice btn-info club-edit-grid" data-bind="click: function() {␣
↪this.performAction('create_inline'); }">
        <span class="iconui iconui-plus"></span> Add row
    </button>
</div>
```

When the document DOM will be ready, `ClubEditGrid` class will be bound to three DOM subtrees, one is generated via `ko_grid()` Jinja2 macro and two located inside separate `<div class="club-edit-grid">` wrappers.

Sparse components may also include inner non-sparse (single DOM subtree) nested components. Nesting of sparse components is unsupported.

3.1.11 Knockout.js subscriber

Javascript mixin class Subscriber may be used to control Knockout.js viewmodel methods subscriptions. To add this mixin to your class:

```
import { inherit } from '../../djk/js/dash.js';
import { Subscriber } from '../../djk/js/ko.js';

inherit(Subscriber.prototype, this);
```

In case there is observable property:

```
this.meta.rowsPerPage = ko.observable();
```

Which changes should be notified to viewmodel method:

```
Grid.on_meta_rowsPerPage = function(newValue) {
    this.actions.perform('list');
};
```

Then to subscribe that method to this.meta.rowsPerPage() changes:

```
this.subscribeToMethod('meta.rowsPerPage');
```

An example of temporary unsubscription / subscription to the method, used to alter observable value without the execution of an observation handler:

```
Grid.listCallback = function(data) {
    // ... skipped ...
    // Temporarily disable meta.rowsPerPage() subscription:
    this.disposeMethod('meta.rowsPerPage');

    // Update observable data but .on_meta_rowsPerPage() will not be executed:
    this.meta.prevRowsPerPage = this.meta.rowsPerPage();
```

(continues on next page)

(continued from previous page)

```
    this.meta.rowsPerPage(data.rowsPerPage);

    // Re-enable meta.rowsPerPage() subscription:
    this.subscribeToMethod('meta.rowsPerPage');
    // ... skipped ...
}
```

3.1.12 dash.js

This module implements low-level Javascript helpers, such as:

- advanced typechecking `isMapping()` / `isScalar()`
- value conversion `intVal()` / `capitalize()` / `camelCaseToDash()`
- `ODict` ordered dict element, used by `NestedList` / `GridColumn` (See *Datatables* for more info.)
- Multiple level Javascript class inheritance

Multiple level Javascript class inheritance

- `inherit()` - implementation of meta inheritance. Copies parent object `prototype` methods into `instance` of pseudo-child. Supports nested multi-level inheritance with chains of `_super` calls in Javascript via SuperChain class.
- Multi-level inheritance should be specified in descendant to ancestor order.

For example to inherit from base class `ClosablePopover`, then from immediate ancestor class `ButtonPopover`, use the following Javascript code:

```
import { inherit } from '../../djk/js/dash.js';
import { ButtonPopover, ClosablePopover } from '../../djk/js/popover.js';

CustomPopover = function(options) {
    // Immediate ancestor.
    inherit(ButtonPopover.prototype, this);
    // Base ancestor.
    inherit(ClosablePopover.prototype, this);
    this.init(options);
};

(function(CustomPopover) {

    CustomPopover.init = function(options) {
        // Will call ButtonPopover.init(), with current 'this' context when such␣
↪method is defined, or
        // will call ClosablePopover.init(), with current 'this' context, otherwise.
        // ButtonPopover.init() also may call it's this._super._call('init', options)␣
↪via inheritance chain.
        this._super._call('init', options);
    };

})(CustomPopover.prototype);
```

An example of multi-level inheritance from the built-in `grid/dialogs.js`:

```
import { Dialog } from '../dialog.js';

function FilterDialog(options) {

    inherit(Dialog.prototype, this);
    this.create(options);

} void function(FilterDialog) {

    FilterDialog.create = function(options) {
        // ... skipped ...
    };

    // ... skipped ...

}(FilterDialog.prototype);

function GridDialog(options) {

    inherit(FilterDialog.prototype, this);
    inherit(Dialog.prototype, this);
    this.create(options);

} void function(GridDialog) {

    GridDialog.template = 'ko_grid_body';

    GridDialog.create = function(options) {
        this.componentSelector = null;
        this._super._call('create', options);
    };

    // ... skipped ...

}(GridDialog.prototype);
```

See *Datatables* for more info.

3.1.13 popovers.js

Advanced popovers

`ClosablePopover` creates the popover with close button. The popover is shown when mouse enters the target area. It's possible to setup the list of related popovers to auto-close the rest of popovers besides the current one like this:

```
import { ClosablePopover } from '../../djk/js/popover.js';

messagingPopovers = [];

var messagingPopover = new ClosablePopover({
    target: document.getElementById('notification_popover'),
    message: 'Test',
    relatedPopovers: .messagingPopovers,
});
```

`ButtonPopover` creates closable popover with additional dialog button which allows to perform onclick action via

overridable `.clickPopoverButton()` method.

3.1.14 plugins.js

Set of jQuery plugins.

jQuery plugins

- `$.autogrow` plugin to automatically expand text lines of textarea elements;
- `$.linkPreview` plugin to preview outer links in secured html5 iframes;
- `$.scroller` plugin - AJAX driven infinite vertical scroller;
- `$.replaceWithTag` plugin to replace HTML tag with another one, used by initClient and by *Underscore.js templates* to create custom tags.

3.1.15 ko.js

Some of these jQuery plugins have corresponding Knockout.js bindings in `ko.js`, simplifying their usage in client-side scripts:

- `ko.bindingHandlers.autogrow`:

```
<textarea data-bind="autogrow: {rows: 4}"></textarea>
```

- `ko.bindingHandlers.linkPreview`:

```
<div data-bind="html: text, linkPreview"></div>
```

- `ko.bindingHandlers.scroller`:

```
<div class="rows" data-bind="scroller: {top: 'loadPreviousRows', bottom:
↪'loadNextRows'}"></div>
```

To make these bindings available, one has to import and to execute `useKo` function:

```
import { useKo } from '../../djk/js/ko.js';

useKo(ko);
```

which is performed already in document.js.

3.1.16 tooltips.js

- Implements *Client-side viewmodels and AJAX response routing* for Bootstrap tooltips and popovers. These viewmodels are used in client-side part of AJAX forms validation, but not limited to.

3.2 context_processors.py

Context processor injects the tpl / utils.sdv modules to to Jinja2 template context, allowing to write more powerful templates. Any function / class from these modules are immediately available in Jinja2 templates. Additionally some useful functions / classes are loaded (see *Meta and formatting*).

- tpl module implements functions / classes for advanced text / html formatting; see *tpl.py* for detailed information.
- utils.sdv module implements low-level support functions / classes;

3.2.1 Functions to manipulate css classes in Jinja2 templates

- `tpl.add_css_classes()` - similar to jQuery `$.addClass()` function;
- `tpl.has_css_classes()` - similar to jQuery `$.hasClass()` function;
- `tpl.remove_css_classes()` - similar to jQuery `$.removeClass()` function;

Next are the methods that alter 'class' key value of the supplied HTML attrs dict, which is then passed to Django `flatatt()` call / `tpl.json_flatatt()` call:

- `tpl.add_css_classes_to_dict()`
- `tpl.has_css_classes_in_dict()`
- `tpl.prepend_css_classes_to_dict()`
- `tpl.remove_css_classes_from_dict()`

3.2.2 PageContext (page_context)

Since version 1.0.0, PageContext class is used to generate additional template context required to run client-side of the framework. To instantiate this class, create_page_context() function is called. It uses DJK_PAGE_CONTEXT_CLS setting to load class from the string, which value can be overridden in project `settings.py` to add custom data / functionality.

The instance of PageContext is stored into current view TemplateResponse *context_data* dict *'page_context'* key. Such way the instance of PageContext class becomes available in DTL / Jinja2 templates as page_context variable. page_context methods are used to generate html title, client-side JSON configuration variables and dynamic script tags.

To add page_context variable to the current view template context, function views should use page_context_decorator:

```
from django.template.response import TemplateResponse
from django_jinja_knockout.views import page_context_decorator

@page_context_decorator(view_title='Decorated main page title')
def main_page(request, **kwargs):
    return TemplateResponse(request, 'main.htm')
```

or to instantiate page_context manually:

```
from django.template.response import TemplateResponse
from django_jinja_knockout.views import create_page_context

def club_list_view(request, **kwargs):
    page_context = create_page_context(request=request, client_routes={
        'profile_detail',
        'club_view',
    })
    context = {
        'page_context': page_context,
        'clubs': Club.objects.all(),
    }
    return TemplateResponse(request, 'page_clubs.htm', context)
```

To include page_context in the class-based view template, one should inherit from PageContextMixin or it's ancestors as basically all class-based views of `django-jinja-knockout` inherit from it. It has `.view_title`, `.client_data`, `.client_routes`, `.custom_scripts` class attributes to specify page_context argument values:

```
class CreateClub(PageContextMixin):

    view_title = 'Create new club'
    # Will be available as AppClientData['club'] in Javascript code.
    client_data = {
        'club': 12,
    }
    # Will be available as client-side url Url('manufacturer_fk_widget', {'action':
↪'name-of-action'})
    client_routes = {
        'manufacturer_fk_widget',
        'profile_fk_widget',
        'tag_fk_widget',
    }
    # Will be additionally loaded in 'base_bottom_scripts.htm' template.
    custom_scripts = [
        'djk/js/grid.js',
        'js/member-grid.js',
    ]
```

Also, one may add page_context via PageContextMixin `.create_page_context()` singleton method:

```
class ClubPage(PageContextMixin):
    template_name = 'club.htm'

    def get_context_data(self, **kwargs):
        self.create_page_context().add_client_routes({
            'club_detail',
            'profile_detail',
        })
        return super().get_context_data(**kwargs)
```

page_context will be stored into class-based view instance `self.page_context` attribute and injected into TemplateResponse when the view is rendered. One may update already existing view `self.page_context` via .update_page_context() method.

To access client route in Javascript code:

```
import { Url } from '../../djk/js/url.js';

Url('profile_detail', {profile_id: pk})
```

To ensure that page_context is always available in Jinja2 template:

```
{% if page_context is not defined -%}
    {% set page_context = create_page_context(request) -%}
{% endif -%}
```

To ensure that page_context is always available in DTL template:

```
{% load page_context %}
{% init_page_context %}
```

The following page_context methods are used to get page data in templates:

- get_view_title() - see *View title*
- get_client_conf() - see *Injection of Django url routes into loaded page*
- get_client_data() - see *Injection of server-side data into loaded page*
- get_custom_scripts() - see *Injection of custom script urls into loaded page*

Injection of Django url routes into loaded page

- get_client_conf() method returns the dict which is passed to client-side via AppConf Javascript instance with the following keys:
- `'jsErrorsAlert'` - boolean value, whether Javascript errors should produce modal alert;
- `'jsErrorsLogging'` - boolean value, whether Javascript errors should be reported to admin email;
  - See also *Installation* how to setup Javascript error logging.
- `'csrfToken'` - current CSRF token to be used with AJAX POST from Javascript;
- `'languageCode'` - current Django language code;
- `'staticPath'` - root static url path to be used with AJAX requests from Javascript;
- `'userId'` - current user id, 0 for anonymous; used to detect authorized users and with AJAX requests;
- `'url'` - the dict of Django {`url name`: `sprintf pattern`}, generated by get_client_urls() method from the set of Django url names (client_routes) which are later converted to Javascript object to be used with AJAX requests. It allows not to have hard-coded app urls in Javascript code. Url names with kwargs are supported since v0.2.0. Namespaced urls are supported since v0.9.0.

To add client-side accessible url in function-based view:

```
from django.template.response import TemplateResponse
from django_jinja_knockout.views import page_context_decorator

@page_context_decorator(client_routes={
    'club_detail',
    'member_grid',
})
def my_view(request):
    return TemplateResponse(request, 'template.htm', {'data': 12})
```

To statically add client-side accessible urls in CBV:

```
class MyView(PageContextMixin)

    client_routes = {
        'club_detail',
        'member_grid',
    }
```

To dynamically add client-side accessible urls in CBV:

```
class MyView(PageContextMixin)
    # ...
    def get_context_data(self, **kwargs):
        self.create_page_context().add_client_routes({
            'club_detail',
```

(continues on next page)

(continued from previous page)

```
            'member_grid',
        })
```

Single url can be added as:

```
self.create_page_context().add_client_routes('club_detail')
```

page_context_decorator()

page_context_decorator allows to quickly provide `view_title` / `client_data` / `client_routes` / `custom_scripts` for function-based Django views:

```
from django.template.response import TemplateResponse
from django_jinja_knockout.views import page_context_decorator

@page_context_decorator(
    view_title='Decorated main page title',
    client_data={'todo': 'club'},
    client_routes={'club_detail', 'club_edit'},
    custom_scripts=['main.js']
)
def main_page(request, **kwargs):
    return TemplateResponse(request, 'main.htm')
```

Injection of server-side data into loaded page

- get_client_data() method returns the dict, injected as JSON to HTML page, which is accessible at client-side via `AppClientData()` Javascript function call.

Sample template

```
<script type="application/json" class="app-conf">
    {{ page_context.get_client_conf()|to_json(True) }}
</script>
<script type="application/json" class="app-client-data">
    {{ page_context.get_client_data()|to_json(True) }}
</script>
```

To pass data from server-side Python to client-side Javascript, one has to access PageContext singleton instance:

```
self.create_page_context().update_client_data({
    'club_id': self.object_id
})
```

To access the injected data in Javascript code:

```
import { AppClientData } from '../../djk/js/conf.js';

AppClientData('club_id')
```

It may also include optional JSON client-side viewmodels, stored in `onloadViewModels` key, which are executed when html page is loaded (see *Client-side viewmodels and AJAX response routing* for more info):

```
self.create_page_context().update_client_data({
    'onloadViewModels': {
      'view': 'alert',
      'message': 'Hello, world!',
    }
})
```

Injection of custom script urls into loaded page

To inject custom script to the bottom of loaded page, use the following call in Django view:

```
self.create_page_context().set_custom_scripts(
    'my_project/js/my-custom-dialog.js',
    'my_project/js/my-custom-grid.js',
)
```

To dynamically set custom script from within Django template, use PageContext instance stored into page_context template context variable:

```
{% do page_context.set_custom_scripts(
    'my_project/js/my-custom-dialog.js',
    'my_project/js/my-custom-grid.js',
) -%}
```

The order of added scripts is respected, however multiple inclusion of the same script will be omitted to prevent client-side glitches. There is also an additional check against inclusion of duplicate scripts at client-side via `assertUniqueScripts()` function call.

It's also possible to conditionally add extra scripts to the existing set of scripts via PageContext class add_custom_scripts() method which is intended to add legacy es5 scripts / non-JS scripts, because the es6 scripts are imported as es6 modules, see *es6 module loader*.

To set custom tag attributes to PageContext scripts one may pass the dict as the value of `add_custom_scripts()` / `set_custom_scripts()` method. The key `src` of the passed dict will specify the name of script, the rest of it's keys has the values of script attributes, such as `type`. The default `type` key value is `module` for es6 modules which can be overridden by DJK_JS_MODULE_TYPE `settings.py` variable value.

- See set_custom_scripts() sample for the complete example.
- See *es6 module loader* for additional information about DJK_JS_MODULE_TYPE setting.

3.2.3 Meta and formatting

- get_verbose_name() allows to get verbose_name of Django model field, including related (foreign) and reverse related fields.
- Django functions used to format html content: flatatt() / format_html().
- Possibility to raise exceptions in Jinja2 templates:

  ```
  {{ raise('Error message') }}
  ```

3.2.4 Advanced url resolution, both forward and reverse

- `tpl.resolve_cbv()` takes url_name and kwargs and returns a function view or a class-based view for these arguments, when available:

```
tpl.resolve_cbv(url_name, view_kwargs)
```

- `tpl.reverseq()` allows to build reverse urls with optional query string specified as Python dict:

```
tpl.reverseq('my_url_name', kwargs={'club_id': club.pk}, query={'type': 'approved
↪'})
```

See *tpl.py* for more info.

3.2.5 Miscellaneous

- `sdv.dbg()` for optional template variable dump (debug).
- Context processor is inheritable which allows greater flexibility to implement your own custom features by overloading it's methods.

3.3 Datatables

3.3.1 Introduction

Client-side grid.js script and server-side views.ajax.KoGridView Python class provide possibility to create AJAX-powered datatables (grids) for Django models, using underscore.js / knockout.js client-side templates.

It makes datatables from Django models similar to traditional `django.contrib.admin` built-in module, but provides the advantages of HTTP traffic minimization, multiple / nested widgets and custom client-side features such as compound columns.

views.ajax.KoGridView and views.list.ListSortingView have common ancestor class views.base.BaseFilterView, which allows to partially share the functionality between AJAX datatables and traditional paginated lists, although currently datatables (grids) are more featured and support wider variety of model field filters. See *Built-in views* for more info about views.list.ListSortingView.

knockout.js viewmodels are used to display / update AJAX datatables.

Each grid row represents an instance of associated Django model which can be browsed and manipulated by grid class.

There are key advantages of using AJAX calls to render Django Model datatables:

- Minimization of HTTP traffic.
- Possibility of displaying multiple datatables at the same web page and interact between them (for example update another datatable when current datatable is updated). See *Grids interaction*.
- Custom filters / form widgets that may utilize nested AJAX datatables. See *Grid fields*, *Filter fields*.
- In-place CRUD actions for grid rows using associated `ModelForm` and / or inline formsets with AJAX submission directly via BootstrapDialog. See *Standard actions*, widgets.ForeignKeyGridWidget.

Besides pagination of model data rows, default CRUD actions are supported and can be easily enabled for grids datatables. Custom grid actions for the whole grid as well as for the specific columns can be implemented by inheriting / extending Grid Javascript class and / or views.ajax.KoGridView Python class.

3.3.2 Possible ways of usage

- AJAX datatables (grids) injected into Jinja2 templates as client-side components with *ko_grid() macro*.
- Optional *Foreign key filter* for AJAX grid components.
- Django `ModelForm` widget *ForeignKeyGridWidget* which provides `ForeignKeyRawIdWidget` - like functionality for `ModelForm` to select foreign key field value via AJAX query / response.
- Pop-up AJAX datatable browser via GridDialog with optional editor.

3.3.3 Models used in this documentation

This documentation refers to Django models with one to many relationship defined in club_app.models:

```
from django_jinja_knockout.tpl import Str

# ... skipped ...

class Club(models.Model):
    CATEGORY_RECREATIONAL = 0
    CATEGORY_PROFESSIONAL = 1
    CATEGORIES = (
        (CATEGORY_RECREATIONAL, 'Recreational'),
        (CATEGORY_PROFESSIONAL, 'Professional'),
    )
    title = models.CharField(max_length=64, unique=True, verbose_name='Title')
    category = models.IntegerField(
        choices=CATEGORIES, default=CATEGORY_RECREATIONAL, db_index=True, verbose_
↪name='Category'
    )
    foundation_date = models.DateField(db_index=True, verbose_name='Foundation date')

    class Meta:
        verbose_name = 'Sport club'
        verbose_name_plural = 'Sport clubs'
        ordering = ('title', 'category')

    def save(self, *args, **kwargs):
        if self.pk is None:
            if self.foundation_date is None:
                self.foundation_date = timezone.now()
        super().save(*args, **kwargs)

    def get_absolute_url(self):
        url = Str(reverse('club_detail', kwargs={'club_id': self.pk}))
        url.text = str(self.title)
        return url

    def get_str_fields(self):
        return OrderedDict([
            ('title', self.title),
            ('category', self.get_category_display()),
            ('foundation_date', format_local_date(self.foundation_date))
        ])

    def __str__(self):
```

(continues on next page)

(continued from previous page)

```
        return ' › '.join(self.get_str_fields().values())


class Member(models.Model):
    SPORT_BADMINTON = 0
    SPORT_TENNIS = 1
    SPORT_TABLE_TENNIS = 2
    SPORT_SQUASH = 3
    SPORT_ANOTHER = 4
    BASIC_SPORTS = (
        (SPORT_BADMINTON, 'Badminton'),
        (SPORT_TENNIS, 'Tennis'),
        (SPORT_TABLE_TENNIS, 'Table tennis'),
        (SPORT_SQUASH, 'Squash'),
    )
    SPORTS = BASIC_SPORTS + ((SPORT_ANOTHER, 'Another sport'),)
    ROLE_OWNER = 0
    ROLE_FOUNDER = 1
    ROLE_MEMBER = 2
    ROLES = (
        (ROLE_OWNER, 'Owner'),
        (ROLE_FOUNDER, 'Founder'),
        (ROLE_MEMBER, 'Member'),
    )
    profile = models.ForeignKey(Profile, verbose_name='Sportsman')
    club = models.ForeignKey(Club, blank=True, verbose_name='Club')
    last_visit = models.DateTimeField(db_index=True, verbose_name='Last visit time')
    plays = models.IntegerField(choices=SPORTS, default=SPORT_ANOTHER, verbose_name=
↪'Plays sport')
    role = models.IntegerField(choices=ROLES, default=ROLE_MEMBER, verbose_name=
↪'Member role')
    note = models.TextField(max_length=16384, blank=True, default='', verbose_name=
↪'Note')
    is_endorsed = models.BooleanField(default=False, verbose_name='Endorsed')

    class Meta:
        unique_together = ('profile', 'club')
        verbose_name = 'Sport club member'
        verbose_name_plural = 'Sport club members'

    def get_absolute_url(self):
        url = Str(reverse('member_detail', kwargs={'member_id': self.pk}))
        str_fields = flatten_dict(self.get_str_fields(), enclosure_fmt=None)
        url.text = ' / '.join([str_fields['profile'], str_fields['club']])
        return url

    def get_str_fields(self):
        parts = OrderedDict([
            ('profile', self.profile.get_str_fields()),
            ('club', self.club.get_str_fields()),
            ('last_visit', format_local_date(timezone.localtime(self.last_visit))),
            ('plays', self.get_plays_display()),
            ('role', self.get_role_display()),
            ('is_endorsed', 'endorsed' if self.is_endorsed else 'unofficial')
        ])
        return parts
```

(continues on next page)

(continued from previous page)

```
    def __str__(self):
        str_fields = self.get_str_fields()
        return str_dict(str_fields)
```

3.3.4 Simplest datatable

If you have Django model created and migrated, then it is quite easy to add grid for that model to Django app Jinja2 template, providing your templates are inherited from base_min.htm, or based on a custom-based template which includes the same client-side scripts as `base_min.htm` does.

In your app view code (we use club_app.views_ajax in this example) create the following view:

```
class SimpleClubGrid(KoGridView):

    model = Club
    grid_fields = '__all__'
    # Remove next line to disable columns sorting:
    allowed_sort_orders = '__all__'
```

Now let's add an url name (route) in urls.py:

```
from django_jinja_knockout.urls import UrlPath
from club_app.views_ajax import SimpleClubGrid

# ... skipped ...

UrlPath(SimpleClubGrid)(
    name='club_grid_simple',
    kwargs={'view_title': 'Simple club grid', 'permission_required': 'club_app.change_
↪club'}
),
# ... skipped ...
```

UrlPath automatically generates re_path pattern with named capture group `<action>` used by `KoGridView.post()` method for class-based view kwargs value HTTP routing to provide grid pagination and optional CRUD actions. Custom actions might be implemented via ancestor classes of `KoGridView`.

We assume that our datatable grid may later define actions which can change `Club` table rows, thus our view requires `club_app.change_club` permission from built-in `django.contrib.auth` module.

Our datatable grid is works just with few lines of code, but where is the template that generated initial HTML content?

By default, KoGridView uses built-in cbv_grid.htm template, which content looks like this:

```
{% from 'ko_grid.htm' import ko_grid with context %}
{% from 'ko_grid_body.htm' import ko_grid_body with context %}
{% extends 'base.htm' %}

{% block main %}

{{
ko_grid(
    grid_options={
        'pageRoute': view.request.resolver_match.url_name,
    }
)
```

(continues on next page)

(continued from previous page)

```
}}

{% endblock main %}

{% block bottom_scripts %}
    {{ ko_grid_body() }}
{% endblock bottom_scripts %}
```

One may extend this template to customize grid, which we will do later.

Take a note that two Jinja2 macros are imported. Let's explain their purpose.

ko_grid() macro

Jinja2 macro `ko_grid()` generates html code of client-side component which looks like this in the generated page html:

```
<a name="club_grid"></a>
<div class="component"
    data-component-class="ClubGrid"
    id="club_grid"
    data-component-options='{"defaultOrderBy": {"foundation_date": "-"}, "pageRoute":
↪"club_grid_with_action_logging"}'
    data-template-args="{'show_pagination': true, 'show_title': true, 'vscroll': true}
↪"
    data-template-id="ko_grid_body"
    data-template-options="{'meta_is_grid': true}">
</div>
```

The code is inserted into web page body block. This HTML is not the full DOM subtree of grid but an initial stub. It will be automatically expanded with the content of underscore.js template with name `ko_grid_body` by bindTemplates called via initClientHooks. See *Underscore.js templates* for more details.

At the next step, expanded DOM subtree will be automatically bound to newly created instance of Grid Javascript class via components class instance *.add()* method to make the grid "alive".

See *Component IoC* how to register custom Javascript `data-component-class`, like `ClubGrid` mentioned above.

`ko_grid()` macro accepts the following kwargs:

- Mandatory `grid_options` are client-side component options of current grid. It's a dict with the following keys:
  - Mandatory key `'pageRoute'` is used to get Python grid class in `ko_grid()` macro to autoconfigure client-side options of grid (see the macro code in ko_grid.htm for details).
  - The rest of the keys are optional and are passed to the constructor of Grid class. They could be used to modify grid appearance / behavior. See Grid class `.init()` method `.options` property for the current list of possible options. Some of these are:
    * `alwaysShowPagination` - set to `False` to show pagination controls only when there is more than one page of model instances are available.
    * `expandFilterContents` - whether the templates of datatable filters should be expanded as recursive underscore templates; by default is `False`.
    * `defaultOrderBy` - override initial order_by field name (by default Django model `Meta.ordering` is used).

    * `highlightMode` - built-in modes (See *'switch_highlight' action*):
        - `'none'` - do not highlight,
        - `'cycleColumns'` - highlight columns with Bootstrap colors,
        - `'cycleRows'` - highlight rows with Bootstrap colors,
        - `'linearRows'` - highlight rows with CSS gradient,
    * `preloadedMetaList` - see *'meta list' action preload*.
    * `searchPlaceholder` - text to display when search field is empty.
    * `separateMeta` - see *'meta_list' action and custom initial field filters*.
    * `showCompoundKeys` - boolean, whether the names of compound columns should be displayed;
    * `showSelection` - enable selection of single rows (one model instance of grid).
    * `ownerCtrl` - used internally to embed client-side parts of datatables (grids) into another classes, for example into *ForeignKeyGridWidget* dialogs and *Foreign key filter*. The value of this option should be the instance of Javascript class, thus it is unused in server-side `ko_grid()` macro and should be provided in the inherited client-side class instead.
        - See *Customizing visual display of fields at client-side* for a simple example of grid inheritance.
        - See GridDialog code for the example of embedding grid into another Javascript class via `ownerCtrl` property.
    * `selectMultipleRows` - set to `True` to enable multiple rows selection. Can be used to perform action with querysets of models, not just one Model instance. Use `objects = self.get_queryset_for_action()` in Django `KoGridView` derived CBV action handler to get the queryset with selected model instances. See action_delete implementation for example.
    * `vScrollPage` - whether datatable with `"template_args": { "vscroll": true }` should have it's rows scrolled to the top after each page load; by default is `True`.

- Optional `template_args` argument is passed as `data-template-args` attribute to underscore.js template, which is then used to alter visual layout of grid. In our case we assume that rows of `club_app.Club` may be visually long enough so we turn on vertical scrolling for these via `"vscroll": true` (which is off by default).
- Optional `dom_attrs` argument is used to set extra DOM attributes of the component template:

  It may provide the value of component DOM `id` attribute which may then be used to get the instance of component (instance of Grid class). It is especially useful in the pages which define multiple datatables (grids) that interact to each other. See *Grids interaction* for more details.

  It also allows to pass custom values of template `data-template-id`, `data-template-args`, `data-template-options` html attributes used by template processor Tpl. See *Underscore.js templates* for more detail on these attributes usage. See also member_grid_tabs.htm for the example of overriding the template.
- See ko_grid.htm for the source code of *ko_grid() macro*.
- See components.js components instance for the details of client-side components implementation.
- See tpl.js Tpl class for the details of client-side template processor implementation.

ko_grid_body() macro

`ko_grid_body()` macro, defined in ko_grid_body.htm is inserted into web page bottom scripts block. However it does not contain directly executed Javascript code, but a set of recursive `underscore.js` templates (such as

`ko_grid_body`) that are applied automatically to each grid component DOM nodes, generated by before mentioned `ko_grid()` Jinja2 macro.

Since v2.0, *es6 module loader* with *Component IoC* is used to dynamically load Grid class, so the manual inclusion of grid.js script to Jinja2 / DTL templates is not required anymore. Only the Javascript *Client-side entry points* has to be specified. These entry points also may be used with django_deno app to generate IE11 compatible bundle and / or minified es6 bundle.

`ko_grid_body()` macro includes two versions of filter field widgets:

- `ko_grid_filter_choices` / `ko_grid_filter_popup` used by default, when filter values are selected via bootstrap drop-down menus.
- `ko_grid_breadcrumb_filter_choices` / `ko_grid_breadcrumb_filter_popup`, when filter values are displayed as bootstrap breadcrumbs. To activate this version of filter field widgets, one should call `ko_grid_body()` macro like this:

```
{{
    ko_grid_body(
        include_ids=[
            'ko_grid_breadcrumb_filter_choices',
            'ko_grid_breadcrumb_filter_popup'
        ],
        exclude_ids=[
            'ko_grid_filter_choices',
            'ko_grid_filter_popup'
        ]
    )
}}
```

`exclude_ids` argument saves a bit of html removing unused underscore.js templates from the resulting page. It is also possible to have multiple grids datatables with different styles of filters at the same page. In such case `exclude_ids` argument should not be used. There is cbv_grid_breadcrumbs.htm Jinja2 macro that could be used as `template_name` value of `KoGridView` derived grid class attribute to use breadcrumb-style filters. See sample project club_app.views_ajax for the example.

3.4 Grid configuration

Let's see some more advanced grid sample for the `club_app.models.Member`, Django view part:

```
from django_jinja_knockout.views import KoGridView
from .models import Member

class MemberGrid(KoGridView):

    client_routes = {
        'member_grid',
        # url name (route) for 'profile' key of self.allowed_filter_fields
        'profile_fk_widget',
        # url name (route) for 'club' key of self.allowed_filter_fields
        'club_grid_simple'
    }
    # Use custom grid template instead of default 'cbv_grid.htm' template.
    template_name = 'member_grid.htm'
    model = Member
    grid_fields = [
```

(continues on next page)

(continued from previous page)

```
        'profile',
        'club',
        # Compound columns:
        [
            # Will join 'category' field from related 'Club' table automatically via␣
↪Django ORM.
            'club__category',
            'last_visit',
            'plays',
            'role',
        ],
        'note',
        'is_endorsed'
    ]
    # Will include all model field raw values to JSON response.
    exclude_fields = []
    search_fields = [
        ('club__title', 'icontains'),
        ('profile__first_name', 'icontains'),
        ('profile__last_name', 'icontains')
    ]
    allowed_sort_orders = [
        'club',
        'last_visit',
        'plays',
        'is_endorsed'
    ]
    allowed_filter_fields = OrderedDict([
        ('profile', None),
        ('club', None),
        ('last_visit', None),
        ('club__category', None),
        # Include only some Django model choices and disable multiple choices for
↪'plays' filter.
        ('plays', {
            'type': 'choices', 'choices': Member.BASIC_SPORTS, 'multiple_choices':␣
↪False
        }),
        ('role', None),
        ('is_endorsed', None),
    ])
```

See club_app.views_ajax for the full sample.

Client-side response of `KoGridView` *'list' action* returns only raw values of `grid_fields` by default.

- To include all field values, set class-level attribute `exclude_fields` of `KoGridView` ancestor to empty list.
- To exclude some sensitive field values from client-side exposure, add these to `exclude_fields` list.

3.4.1 Grid fields

Django model may have many fields, some of these having long string representation, thus visually grid may become too large to fit the screen and hard to navigate. Not all of the fields always has to be displayed.

Some fields may need to be hidden from user for security purposes. One also might want to display foreign key span relationships, which are implemented in Django ORM via `'__'` separator between related fields name, like

`club__category` in this example.

Set Django grid class `grid_fields` property value to the list of model fields that will be displayed as grid columns. Spanned foreign key relationship are supported as well.

Grid fields dicts

Since v2.0, each value of `grid_fields` can be dict with the following keys:

- `field`: mandatory name of Django Model field or a name of virtual field (*Virtual fields*).
- `name`: optional localized name of field, displayed in datatable header.
- `virtual`: optional boolean, which indicates that the current field is a virtual one (*Virtual fields*).

The example of defining both *Grid fields dicts*, *Compound columns* and *Virtual fields*:

```
from django.utils.translation import gettext as _

from django_jinja_knockout.views import KoGridView

class ControlGrid(KoGridView):

    model = Control

    grid_fields = [
        # Three compound columns:
        [
            'control__start_date',
            'ctrl_id',
            # Virtual field with custom local verbose name
            {'field': 'ctrl_set__count', 'name': _('Number of controls'), 'virtual':␣
↪True},
        ],
        # Two "ordinary" columns:
        'start_date',
        'finish_date`,
        # Two compound columns. Each field has relation spans.
        [
            'control__decline_threshold',
            'control__growth_threshold',
        ],
    ]
```

See club_app.views_ajax for the actual examples of using `grid_fields` dict values.

Compound columns

Compound columns are supported. In the example above, 8 fields will be displayed in 5 columns, conserving horizontal display space of datatable row:

Table 1: MemberGrid

'profile'	'club'	'club__category' 'last_visit' 'plays' 'role'	'note'	'is_endorsed'
profile1	club1	club__category1 last_visit1 plays1 role1	note1	is_endorsed1
profile2	club2	club__category2 last_visit2 plays2 role2	note2	is_endorsed2

`profile` / `club` / `note` fields visual display can take lots of screen space, because first two are foreign fields, while `note` is a `TextField`, thus these are rendered in separate columns of datatable.

`club_category` / `last_visit` / `plays` / `role` fields visual display is short, thus these are grouped into single compound column to preserve display space.

`is_endorsed` field does not take lots of space, however it's a very important one, thus is displayed in separate column.

Traditional non-AJAX views.list.ListSortingView also supports compound columns with the same definition syntax:

```
class ActionList(ContextDataMixin, ListSortingView):
    # Enabled always visible paginator links because there could be many pages of␣
↪actions, potentially.
    always_visible_links = True
    model = Action
    grid_fields = [
        [
            'performer',
            'performer__is_superuser',
            'date',
        ],
        'action_type',
        'content_object'
    ]
    allowed_sort_orders = [
        'performer',
        'date',
        'action_type',
    ]

    def get_allowed_filter_fields(self):
        allowed_filter_fields = {
            'action_type': None,
            'content_type': self.get_contenttype_filter(
                ('club_app', 'club'),
                ('club_app', 'equipment'),
                ('club_app', 'member'),
            )
        }
        return allowed_filter_fields
```

Nested verbose field names

Grid datatables and grid-based classes like *ForeignKeyGridWidget* support displaying verbose / localized field names of Django model instances with their values, including foreign key related model fields. It is supported in the following cases:

- Related model fields display in grid cells;
- Grid row actions;
- *ForeignKeyGridWidget* display of chosen fk value;
- Client-side support of field names display is added into renderNestedList via `options.i18n` mapping.
- Server-side support of rendering verbose field names is implemented in:
  - `tpl` module `print_list()` function now supports optional `show_keys` / `i18n` arguments.
  - `models` module functions used to gather verbose field names of Django model:
    - `model_fields_meta()` - get fields verbose names of the selected model;
    - `yield_related_models()` - get related models of the selected model;
  - `views.ajax.GridActionsMixin` class:
    - `get_model_fields_verbose_names()` - get current grid Django model fields verbose names.
    - `get_related_model_fields_verbose_names()` - get related models fields verbose names.
    - `get_related_models()` returns the list of related models.

The list of current model verbose field names is returned by *'meta' action* as value of `meta.listOptions` property, while the list of related models fields verbose names is returned as value of `meta.fkNestedListOptions` property.

By default the list of related models fields verbose names is collected automatically, but in case grid model has generic relationships, these can be specified manually via class-level `related_models` property like this:

```
from .models import Action, Club, Equipment, Manufactures, Member, Profile
from django_jinja_knockout.views import KoGridView
# ... skipped ...

class ActionGrid(KoGridView):

    client_routes = {
        'user_fk_widget'
    }
    model = Action
    grid_fields = [
        'performer',
        'date',
        'action_type',
        'content_type',
        'content_object'
    ]
    # Autodetection of related_models is impossible because Action model has generic␣
↪relationships.
    related_models = [Club, Equipment, Manufacturer, Member, Profile]

    # ... skipped ...
```

Relation prefixes `club`, `equipment` and so on will be automatically prepended to related models verbose names to avoid the name clash in case different related models fields having the same field name but a different verbose name.

See event_app.views_ajax `ActionGrid` class for the full example.

It is possible to specify relation prefix manually with `related_models` initialized as dict. To use repeated prefix, initialize grid `related_models` class level property as the list of tuple pairs:

```
from .models import EventLog, Club, Equipment, Member
from django_jinja_knockout.views import KoGridView
# ... skipped ...

class EventLogGrid(KoGridView):

    model = EventLog
    grid_fields = [
        'user__username',
        'content_object',
        'content_type',
    ]
    allowed_sort_orders = [
        'user__username',
        'content_type',
    ]
    search_fields = [
        ('user__username', 'icontains'),
    ]
    related_models = [
        ('content_object', Club),
        ('content_object', Equipment),
        ('content_object', Member),
    ]
    # ... skipped ...
```

To override automatic collecting of Django model verbose field names, one has to define Django model `@classmethod get_fields_i18n`, which should return a dict with keys as field names and values as their verbose / localized names.

Customizing visual display of fields at client-side

To alter visual representation of grid row cells, one should override GridRow Javascript class `.display()` method, to implement custom display layout of field values at client-side. The same method also can be used to generate condensed representations of long text values via Bootstrap popovers, or even to display fields as form inputs: using grid as paginated AJAX form - (which is also possible but requires writing custom `underscore.js` grid layout templates, partially covered in *modifying_visual_layout_of_grid*):

```
import { inherit } from '../../djk/js/dash.js';
import { Grid } from '../../djk/js/grid.js';
import { GridRow } from '../../djk/js/grid/row.js';

MemberGridRow = function(options) {
    inherit(GridRow.prototype, this);
    this.init(options);
};

(function(MemberGridRow) {
```

(continues on next page)

(continued from previous page)

```
    MemberGridRow.useInitClient = true;

    MemberGridRow.display = function(field) {
        var displayValue = this._super._call('display', field);
        switch (field) {
        case 'role':
            // Display field value as bootstrap label.
            var types = ['success', 'info', 'primary'];
            displayValue = $('<span>', {
                'class': 'label preformatted'
            })
            .text(displayValue)
            .addClass(
                'label-' + (this.values[field] < types.length ? types[this.
↪values[field]] : 'info')
            );
            break;
        case 'note':
            // Display field value as bootstrap clickable popover.
            var gridColumn = this.ownerGrid.getKoGridColumn(field);
            if (this.values[field] !== '') {
                displayValue = $('<button>', {
                    'class': 'btn btn-info',
                    'data-content': this.values[field],
                    'data-toggle': 'popover',
                    'data-trigger': 'click',
                    'data-placement': 'bottom',
                    'title': gridColumn.name,
                }).text('Full text');
            }
            break;
        case 'is_endorsed':
            // Display field value as form input.
            var attrs = {
                'type': 'checkbox',
                'class': 'form-field club-member',
                'data-pkval': this.getValue(this.ownerGrid.meta.pkField),
                'name': field + '[]',
            };
            if (this.values[field]) {
                attrs['checked'] = 'checked';
            }
            displayValue = $('<input>', attrs);
        }
        return displayValue;
    };

})(MemberGridRow.prototype);


MemberGrid = function(options) {
    inherit(Grid.prototype, this);
    this.init(options);
};

(function(MemberGrid) {
```

(continues on next page)

(continued from previous page)

```
    MemberGrid.iocRow = function(options) {
        return new MemberGridRow(options);
    };

})(MemberGrid.prototype);
```

See member-grid.js for full-size example.

GridRow class `.display()` method used in grid.js `grid_compound_cell` binding supports the following types of values:

- jQuery objects, whose set of elements will be added to cell DOM

get_str_fields model formatting / serialization

- Nested list of values, which is automatically passed to client-side in AJAX response by `KoGridView` when current Django model has `get_str_fields()` method implemented. This method returns str() representation of some or all model fields:

```
class Member(models.Model):

    # ... skipped ...

    # returns the list of str() values for all or some of model fields,
    # optionally spanning relationships via nested lists.
    def get_str_fields(self):
        parts = OrderedDict([
            ('profile', self.profile.get_str_fields()),
            ('club', self.club.get_str_fields()),
            ('last_visit', format_local_date(timezone.localtime(self.last_
↪visit))),
            ('plays', self.get_plays_display()),
            ('role', self.get_role_display()),
            ('is_endorsed', 'endorsed' if self.is_endorsed else 'unofficial')
        ])
        return parts

    # It's preferable to reconstruct model's str() via get_str_fields() to keep␣
↪it DRY.
    def __str__(self):
        str_fields = self.get_str_fields()
        return str_dict(str_fields)
```

Model.get_str_fields() will also be used for automatic formatting of scalar fields via grid row `str_fields` property. See *'list' action* for more info.

- Scalar values usually are server-side Django generated strings. Make sure these strings do not contain unsafe HTML to prevent XSS. Here's the sample implementation in the version 2.0:

```
import { renderValue } from '../../djk/js/nestedlist.js';

// Supports jQuery elements / nested arrays / objects / HTML strings as grid cell␣
↪value.
GridColumnOrder.renderRowValue = function(element, value) {
    renderValue(element, value, this.getNestedListOptions());
};
```

Nested list values are escaped by default in `GridRow.htmlEncode()`, thus are not escaped twice in `GridColumnOrder.renderRowValue()`. This allows to have both escaped and unescaped nested lists in row cells with Grid mark_safe_fields attribute list that allows to disable HTML escaping for the selected grid fields.

Since v2.1.0 it's preferable to implement custom serialization via ObjDict class `get_str_fields()` method. The instantiation of serializer is performed via ObjDict.from_obj static method.

See *ObjDict serialization* just below.

ObjDict serialization

Since v2.1.0, all low-level serialization is performed either via default ObjDict or derived class. It incorporates both serialized Django Model instance fields as dict key / value pairs and *self.obj* attribute as the instance itself. To perform custom serialization and / or to implement field permissions filters, one has to inherit from ObjDict class then specify the child class name as the `models.Model Meta.obj_dict_cls` attribute value:

```
from django_jinja_knockout.obj_dict import ObjDict

class ActionObjDict(ObjDict):

    def can_view_field(self, field_name=None):
        return self.request_user is None or self.request_user == self.obj.performer␣
↪or self.request_user.is_superuser

    def get_str_fields(self):
        return OrderedDict([
            ('performer', self.obj.performer.username),
            ('date', format_local_date(self.obj.date) if self.can_view_field() else␣
↪site.empty_value_display),
            ('action_type', self.obj.get_action_type_display()),
            ('content_type', str(self.obj.content_type)),
            (
                'content_object',
                site.empty_value_display
                if self.obj.content_object is None
                else ObjDict(obj=self.obj.content_object, request_user=self.request_
↪user).get_description()
            )
        ])


class Action(models.Model):

    TYPE_CREATED = 0
    TYPE_MODIFIED = 1
    TYPES = (
        (TYPE_CREATED, 'Created'),
        (TYPE_MODIFIED, 'Modified'),
    )

    performer = models.ForeignKey(User, on_delete=models.CASCADE, related_name='+',␣
↪verbose_name='Performer')
    date = models.DateTimeField(verbose_name='Date', db_index=True)
    action_type = models.IntegerField(choices=TYPES, verbose_name='Type of action')
    content_type = models.ForeignKey(
        ContentType, blank=True, null=True, on_delete=models.CASCADE,
        related_name='related_content', verbose_name='Related object'
```

(continues on next page)

(continued from previous page)

```
    )
    object_id = models.PositiveIntegerField(blank=True, null=True, verbose_name=
↪'Object link')
    content_object = GenericForeignKey('content_type', 'object_id')

    class Meta:
        verbose_name = 'Action'
        verbose_name_plural = 'Actions'
        ordering = ('-date',)
        obj_dict_cls = ActionObjDict

    def get_str_fields(self):
        return ObjDict.from_obj(obj=self).get_str_fields()

    def __str__(self):
        str_fields = self.get_str_fields()
        return str_dict(str_fields)
```

Note that `Action.get_str_fields()` method will automatically instantiate specified `Meta.obj_dict_cls` = `ActionObjDict` class, then will call `ActionObjDict` instance `.get_str_fields()` method.

See `djk-sample` event_app.models for the complete example of custom ObjDict serialization class with user permission check.

ObjDict class `request_user` constructor optional argument may be set to filter the visibility of fields per user in the overridden `ObjDict.get_str_fields()` method, where `request_user` is available (not models but views / forms).

Client-side class overriding

To override client-side class to `MemberGrid` instead of default Grid class, define default grid options like this:

```
from django_jinja_knockout.views import KoGridView
from .models import Member

# ... skipped ...

class MemberGrid(KoGridView):

    model = Member
    # ... skipped ...
    grid_options = {
        'classPath': 'MemberGrid'
    }
```

See *Component IoC* how to register custom Javascript `classPath`, like `MemberGrid` mentioned above.

Virtual fields

views.KoGridView also supports virtual fields, which are not real database table fields, but a calculated values. It supports both SQL calculated fields via Django ORM annotations and virtual fields calculated in Python code. To implement virtual field(s), one has to override the following methods in the grid child class:

```
class ClubGridWithVirtualField(SimpleClubGrid):

    grid_fields = [
        'title',
        'category',
        'foundation_date',
        # Annotated field.
        'total_members',
        # Virtual field.
        'exists_days'
    ]

    def get_base_queryset(self):
        # Django ORM annotated field 'total_members'.
        return super().get_base_queryset().annotate(total_members=Count('member'))

    def get_field_verbose_name(self, field_name):
        if field_name == 'exists_days':
            # Add virtual field.
            return 'Days since foundation'
        elif field_name == 'total_members':
            # Add annotated field.
            return 'Total members'
        else:
            return super().get_field_verbose_name(field_name)

    def get_related_fields(self, query_fields=None):
        query_fields = super().get_related_fields(query_fields)
        # Remove virtual field from queryset values().
        query_fields.remove('exists_days')
        return query_fields

    def get_model_fields(self):
        model_fields = copy(super().get_model_fields())
        # Remove annotated field which is unavailable when creating / updating single␣
↪object which does not uses
        # self.get_base_queryset()
        # Required only because current grid is editable.
        model_fields.remove('total_members')
        return model_fields

    def postprocess_row(self, row, obj):
        # Add virtual field value.
        row['exists_days'] = (timezone.now().date() - obj.foundation_date).days
        if 'total_members' not in row:
            # Add annotated field value which is unavailable when creating / updating␣
↪single object which does not uses
            # self.get_base_queryset()
            # Required only because current grid is editable.
            row['total_members'] = obj.member_set.count()
        row = super().postprocess_row(row, obj)
        return row

    # Optional formatting of virtual field (not required).
    def get_row_str_fields(self, obj, row):
        str_fields = super().get_row_str_fields(obj, row)
        if str_fields is None:
```

(continues on next page)

(continued from previous page)

```
            str_fields = {}
        # Add formatted display of virtual field.
        is_plural = pluralize(row['exists_days'], arg='days')
        str_fields['exists_days'] = '{} {}'.format(row['exists_days'], 'day' if is_
↪plural == '' else is_plural)
        return str_fields
```

See club_app.views_ajax code for full implementation.

3.4.2 Filter fields

Grid supports different types of filters for model fields, to reduce paginated queryset, which helps to locate specific data in the whole model's database table rows set.

Full-length as well as shortcut definitions of field filters are supported:

```
from collections import OrderedDict
from django_jinja_knockout.views import KoGridView
from .models import Model1


class Model1Grid(KoGridView):
    # ... skipped ...

    allowed_filter_fields = OrderedDict([
        (
            # Example of complete filter definition for field type 'choices':
            'field1',
            {
                'type': 'choices',
                'choices': Model1.FIELD1_CHOICES,
                # Do not display 'All' choice which resets the filter:
                'add_reset_choice': False,
                # List of choices that are active by default:
                'active_choices': ['field1_value_1'],
                # Do not allow to select multiple choices:
                'multiple_choices': False
            },
        ),
        # Only some of filter properties are defined, the rest are auto-guessed:
        (
            'field2',
            {
                # Commented out to autodetect field type:
                # 'type': 'choices',
                # Commented out to autodetect field.choices:
                # 'choices': Model1.FIELD1_CHOICES,
                # Is true by default, thus switching to False:
                'multiple_choices': False
            }
        ),
        # Try to autodetect field filter completely:
        ('field3', None),
        # Custom choices filter (not necessarily matching Model1.field4 choices):
        ('field4', CUSTOM_CHOICES_FOR_FIELD4),
        # Select foreign key choices via AJAX grid built into BootstrapDialog.
```

(continues on next page)

(continued from previous page)

```
        # Can be replaced to ('model2_fk', None) to autodetect filter type,
        # but explicit type might be required when using IntegerField as foreign key.
        ('model2_fk', {
            'type': 'fk'
        }),
    ])
```

Next types of built-in field filters are available:

Range filters

- `'number' filter` / `'datetime' filter` / `'date' filter`: Uses RangeFilter / GridRangeFilter to display dialog with range of scalar values. It's applied to the corresponding Django model scalar fields.

Choices filter

- `'choices' filter` is used by default when Django model field has `choices` property defined, like `plays` and `role` fields in the next example:

```
from django.utils.translation import ugettext as _
# ... skipped ...

class Member(models.Model):
    SPORT_BADMINTON = 0
    SPORT_TENNIS = 1
    SPORT_TABLE_TENNIS = 2
    SPORT_SQUASH = 3
    SPORT_ANOTHER = 4
    BASIC_SPORTS = (
        (SPORT_BADMINTON, 'Badminton'),
        (SPORT_TENNIS, 'Tennis'),
        (SPORT_TABLE_TENNIS, 'Table tennis'),
        (SPORT_SQUASH, 'Squash'),
    )
    SPORTS = BASIC_SPORTS + ((SPORT_ANOTHER, 'Another sport'),)
    ROLE_OWNER = 0
    ROLE_FOUNDER = 1
    ROLE_MEMBER = 2
    ROLES = (
        (ROLE_OWNER, 'Owner'),
        (ROLE_FOUNDER, 'Founder'),
        (ROLE_MEMBER, 'Member'),
    )
    profile = models.ForeignKey(Profile, verbose_name='Sportsman')
    club = models.ForeignKey(Club, blank=True, verbose_name='Club')
    last_visit = models.DateTimeField(db_index=True, verbose_name='Last visit time
↪')
    plays = models.IntegerField(choices=SPORTS, default=SPORT_ANOTHER, verbose_
↪name='Plays sport')
    role = models.IntegerField(choices=ROLES, default=ROLE_MEMBER, verbose_name=
↪'Member role')
    note = models.TextField(max_length=16384, blank=True, default='', verbose_
↪name='Note')
    is_endorsed = models.BooleanField(default=False, verbose_name='Endorsed')
```

`'choices' filter` is also automatically populated when the field is an instance of `BooleanField` / `NullBooleanField`.

When using `'choices' filter` for a grid column (Django model field), instance of GridFilter will be created at client-side, representing a dropdown with the list of possible choices from the `Club.CATEGORIES` tuple above:

```
from django_jinja_knockout.views import KoGridView
from .models import Member

class MemberGrid(KoGridView):

    model = Member
    # ... skipped ...

    allowed_filter_fields = OrderedDict([
        ('profile', None),
        ('club', None),
        ('last_visit', None),
        ('club__category', None),
        # Include all Django model field choices, multiple selection will be auto-
↪enabled
        # when there are more than two choices.
        ('plays', None),
        ('role', None),
        ('is_endorsed', None),
    ])
```

Choices can be customized by supplying a dict with additional keys / values. See `play` field filter in the next example:

```
class MemberGrid(KoGridView):

    model = Member
    # ... skipped ...

    allowed_filter_fields = OrderedDict([
        ('profile', None),
        ('club', None),
        ('last_visit', None),
        ('club__category', None),
        # Include only limited BASIC_SPORTS Django model field choices
        # and disable multiple choices for 'plays' filter.
        ('plays', {
            'type': 'choices', 'choices': Member.BASIC_SPORTS, 'multiple_choices':␣
↪False
        }),
        ('role', None),
        ('is_endorsed', None),
    ])
```

Query filters support arrays of choices for filter value:

```
class MemberGrid(KoGridView):

    model = Member
    # ... skipped ...

    allowed_filter_fields = OrderedDict([
        (
```

(continues on next page)

(continued from previous page)

```
            'is_endorsed',
            {
                'choices': ((True, 'Active'), ([None, False], 'Candidate')),
            }
        )
    ])
```

When user will select `Candidate` choice from the drop-down list, two filters will be applied: `None` or `False`.

Foreign key filter

- `'fk' filter`: Uses GridDialog to select filter choices of foreign key field. This widget is similar to `ForeignKeyRawIdWidget` defined in django.contrib.admin.widgets that is used via `raw_id_fields` django.admin class option. Because it completely relies on AJAX calls, one should create grid class for the foreign key field, for example:

```
class ProfileFkWidgetGrid(KoGridView):

    model = Profile
    form = ProfileForm
    enable_deletion = True
    grid_fields = ['first_name', 'last_name']
    allowed_sort_orders = '__all__'
```

Define it's url name (route) in urls.py via *UrlPath*:

```
from django_jinja_knockout.urls import UrlPath

UrlPath(ProfileFkWidgetGrid)(
    name='profile_fk_widget',
    # kwargs={'permission_required': 'club_app.change_profile'}
),
```

Now, to bind 'fk' widget for field `Member.profile` to `profile-fk-widget` url name (route):

```
class MemberGrid(KoGridView):

    client_routes = {
        'member_grid',
        'profile_fk_widget',
        'club_grid_simple'
    }
    template_name = 'member_grid.htm'
    model = Member
    grid_fields = [
        'profile',
        'club',
        'last_visit',
        'plays',
        'role',
        'note',
        'is_endorsed'
    ]
    allowed_filter_fields = OrderedDict([
        ('profile', None),
```

(continues on next page)

(continued from previous page)

```
        ('club', None),
        ('last_visit', None),
        ('plays', None),
        ('role', None),
        ('is_endorsed', None),
    ])

    # ... skipped ...

    # Similar to class property grid_options but allows to generate options␣
↪dynamically and to override them.
    @classmethod
    def get_grid_options(cls):
        return {
            # Note: 'classPath' is not required for standard Grid.
            'classPath': 'MemberGrid',
            'searchPlaceholder': 'Search for club or member profile',
            'fkGridOptions': {
                'profile': {
                    'pageRoute': 'profile_fk_widget'
                },
                'club': {
                    'pageRoute': 'club_grid_simple',
                    # Optional setting for BootstrapDialog:
                    'dialogOptions': {'size': 'size-wide'},
                    # Nested filtering is supported:
                    # 'fkGridOptions': {
                    #     'specialization': {
                    #         'pageRoute': 'specialization_grid'
                    #     }
                    # }
                }
            }
        }
```

Explicit definition of `fkGridOptions` in `get_grid_options()` result is not required, but it's useful to illustrate how foreign key filter widgets are nested:

- Define model `Specialization`.
- Add foreignKey field `specialization = models.ForeignKey(Specialization, verbose_name='Specialization')` to `Profile` model.
- Create `SpecializationGrid` with `model = Specialization`.
- Add url for `SpecializationGrid` with url name (route) `'specialization_grid'` to `urls.py`.
- Append `'specialization_grid'` entry to class `MemberGrid` attribute `client_routes` set.

`KoGridView` is able to autodetect `fkGridOptions` of foreign key fields when these are specified in `allowed_filter_fields` (see discover_grid_options for the implementation), making definitions of foreign key filters shorter and more DRY:

```
class MemberGrid(KoGridView):

    client_routes = {
        'member_grid',
        'profile_fk_widget',
```

(continues on next page)

(continued from previous page)

```
        'club_grid_simple'
    }
    template_name = 'member_grid.htm'
    model = Member
    grid_fields = [
        'profile',
        'club',
        'last_visit',
        'plays',
        'role',
        'note',
        'is_endorsed'
    ]
    allowed_filter_fields = OrderedDict([
        ('profile', {
            'pageRoute': 'profile_fk_widget'
        }),
        # When 'club_grid_simple' grid view has it's own foreign key filter fields,␣
↪these will be automatically
        # detected - no need to specify these in .get_grid_options() as nested dict.
        ('club', {
            'pageRoute': 'club_grid_simple',
            # Optional setting for BootstrapDialog:
            'dialogOptions': {'size': 'size-wide'},
        }),
        ('last_visit', None),
        ('plays', None),
        ('role', None),
        ('is_endorsed', None),
    ])
    grid_options = {
        # Note: 'classPath' is not required for standard Grid.
        'classPath': 'MemberGrid',
        'searchPlaceholder': 'Search for club or member profile',
    }
```

See *Component IoC* how to register custom Javascript `classPath`, like `MemberGrid` mentioned above.

Dynamic generation of filter fields

There are many cases when datatables require dynamic generation of filter fields and their values:

- Different types of filters for end-users depending on their permissions.
- Implementing base grid pattern, when there is a base grid class defining base filters, and few child classes, which may alter / add / delete some of the filters.
- `'choices' filter` values might be provided via Django database queryset.
- `'choices' filter` values might be generated as foreign key id's for Django contenttypes framework generic models relationships.

Let's explain the last case as the most advanced one.

Generation of `'choices' filter` list of choice values for Django contenttypes framework is implemented via `BaseFilterView.get_contenttype_filter()` method, whose class is a base class to both `KoGridView` and it's traditional request counterpart `ListSortingView` (see views for details).

We want to implement generic action logging, similar to `django.admin` logging but visually displayed as AJAX grid. Our `Action` model, defined in event_app.models looks like this:

```
from collections import OrderedDict

from django.utils import timezone
from django.db import models
from django.db import transaction
from django.contrib.auth.models import User
from django.contrib.contenttypes.fields import GenericForeignKey
from django.contrib.contenttypes.models import ContentType

from django_jinja_knockout.tpl import format_local_date
from django_jinja_knockout.utils.sdv import flatten_dict, str_dict

class Action(models.Model):

    TYPE_CREATED = 0
    TYPE_MODIFIED = 1
    TYPES = (
        (TYPE_CREATED, 'Created'),
        (TYPE_MODIFIED, 'Modified'),
    )

    performer = models.ForeignKey(User, related_name='+', verbose_name='Performer')
    date = models.DateTimeField(verbose_name='Date', db_index=True)
    action_type = models.IntegerField(choices=TYPES, verbose_name='Type of action')
    content_type = models.ForeignKey(ContentType, related_name='related_content',␣
↪blank=True, null=True,
                                     verbose_name='Related object')
    object_id = models.PositiveIntegerField(blank=True, null=True, verbose_name=
↪'Object link')
    content_object = GenericForeignKey('content_type', 'object_id')

    class Meta:
        verbose_name = 'Action'
        verbose_name_plural = 'Actions'
        ordering = ('-date',)

    # ... skipped ...
```

To allow queryset filtering via 'content_object' field `'choices' filter` (*Choices filter*), `ActionGrid` overrides `get_allowed_filter_fields()` method to generate `'choices' filter` values from contenttypes framework by calling `get_contenttype_filter()` method:

```
from collections import OrderedDict
from django.utils.html import format_html
from django_jinja_knockout.views import KoGridView
from .models import Action

class ActionGrid(KoGridView):

    model = Action
    grid_fields = [
        'performer',
        'date',
        'action_type',
        # Note that generic object relationship field is treated as virtual field␣
↪because Django ORM does not
```

(continues on next page)

(continued from previous page)

```
        # allow to perform values() method on querysets which have such fields.
        'content_object'
    ]
    allowed_sort_orders = [
        'performer',
        'date',
        'action_type',
    ]
    mark_safe_fields = [
        'content_object'
    ]
    enable_deletion = True

    def get_allowed_filter_fields(self):
        allowed_filter_fields = OrderedDict([
            ('action_type', None),
            # Get names / ids of 'content_type' choices filter.
            ('content_type', self.get_contenttype_filter(
                ('club_app', 'club'),
                ('club_app', 'equipment'),
                ('club_app', 'member'),
            ))
        ])
        return allowed_filter_fields

    def get_related_fields(self, query_fields=None):
        query_fields = super().get_related_fields(query_fields)
        # Remove virtual field from queryset values().
        query_fields.remove('content_object')
        return query_fields

    def postprocess_row(self, row, obj):
        # Add virtual field value.
        content_object = obj.content_object
        row['content_object'] = content_object.get_str_fields() \
            if hasattr(content_object, 'get_str_fields') \
            else str(content_object)
        row = super().postprocess_row(row, obj)
        return row

    # Optional formatting of virtual field (not required).
    def get_row_str_fields(self, obj, row=None):
        str_fields = super().get_row_str_fields(obj, row)
        if str_fields is None:
            str_fields = {}
        # Add formatted display of virtual field.
        if hasattr(obj.content_object, 'get_absolute_url'):
            link = obj.content_object.get_absolute_url()
            str_fields['content_type'] = format_html(
                '<a href="{}" target="_blank">{}</a>',
                link,
                str_fields['content_type']
            )
        return str_fields
```

See event_app.views_ajax for the complete example.

3.4.3 Modifying visual layout of grid

Top DOM nodes of grid component can be overridden by using Jinja2 `{% call(kwargs) ko_grid() %}` statement, then implementing a caller section with custom DOM nodes. See the source code of ko_grid.htm template for original DOM nodes of Grid component. This feature is rarely used since version 0.5.0 rewritten template processor offers more simpler ways to override root `ko_grid_body` underscore.js template at client-side.

It is possible to override some or all underscore.js templates of Grid component. `ko_grid()` macro allows to override built-in grid templates with custom ones by providing `dom_attrs` argument with `'data-template-options'` attribute key / values. In the example just below `'member_ko_grid_filter_choices'` and `'member_ko_grid_body'` will be called instead of default templates.

When custom grid templates are defined, one may wish not to include unused standard grid templates. To include only selected standard grid templates, there are optional arguments of `ko_grid_body()` Jinja2 macro with the lists of template names.

- Optional `'include_ids' argument` list of built-in nested templates DOM ids that will be included into generated html page.
- Optional `'exclude_ids' argument` list of built-in nested templates DOM ids to be skipped from generated html page.

Here is the example of overriding visual display of GridFilter that is used to select filter field from the list of specified choices. `ko_grid_body` underscore.js template is overridden to `member_ko_grid_body` template with button inserted that has knockout.js custom binding:

```
"click: onChangeEndorsement"
```

Full code:

```
{% from 'ko_grid.htm' import ko_grid with context %}
{% from 'ko_grid_body.htm' import ko_grid_body with context %}
{% extends 'base.htm' %}

{% block main %}
    {#
        'separateMeta' is required because Django grid specifies 'active_choices'␣
↪field filter value.
    #}
    {#
        Overwrites templates for custom display of MemberGrid.
    #}
    {{ ko_grid(
        grid_options={
            'pageRoute': view.request.resolver_match.url_name,
            'separateMeta': True,
        },
        template_args={
            'vscroll': True
        },
        dom_attrs={
            'id': 'member_grid',
            'data-template-options': {
                'templates': {
                    'ko_grid_body': 'member_ko_grid_body',
                    'member_ko_grid_nav': 'ko_grid_nav',
                    'ko_grid_filter_choices': 'member_ko_grid_filter_choices',
                }
```

(continues on next page)

(continued from previous page)

```
            },
        }
    ) }}

{% do page_context.set_custom_scripts(
    'sample/js/member-grid.js',
) -%}

{% endblock main %}

{% block bottom_scripts %}
    {# Generate standard grid templates for KoGridWidget #}
    {{ ko_grid_body() }}

    <script type="text/template" id="member_ko_grid_body">
        <card-primary data-bind="using: $root, as: 'grid'">
            <card-header data-bind="text: meta.verboseNamePlural"></card-header>
            <card-body>
                <!-- ko if: meta.hasSearch() || gridFilters().length > 0 -->
                <div data-template-id="member_ko_grid_nav"></div>
                <!-- /ko -->
                <div data-template-id="ko_grid_table"></div>
                <div class="default-padding">
                    <button
                            data-bind="click: onChangeEndorsement" type="button"␣
↪class="btn btn-warning">
                        Change endorsement
                    </button>
                </div>
            </card-body>
            <div data-template-id="ko_grid_pagination"></div>
        </card-primary>
    </script>

    <script type="text/template" id="member_ko_grid_filter_choices">
        <li data-bind="grid_filter">
            <nav class="navbar navbar-default">
                <div class="container-fluid">
                    <div class="navbar-header"><a class="navbar-brand" href="##" data-
↪bind="text: name"></a></div>
                    <ul class="nav navbar-nav">
                        <!-- ko foreach: {data: choices, as: 'filterChoice'} -->
                        <li data-bind="css: {active: is_active()}">
                            <a data-bind="css: {bold: is_active()}, text: name, grid_
↪filter_choice, click: onLoadFilter.bind(filterChoice)" name="#"></a>
                        </li>
                        <!-- /ko -->
                    </ul>
                </div>
            </nav>
        </li>
    </script>

{% endblock bottom_scripts %}
```

See member_grid_tabs.htm, member-grid.js, club_app.views_ajax for the complete example.

It's also possible to use different layout for the different cells of datatable row via custom `ko_grid_table` template.

Use `val()` method of grid row to access raw data values (eg. html attributes) and `grid_cell` binding to render individual (non-compound) row cells:

```
<script type="text/template" id="agenda_ko_grid_table">
    <div class="agenda-wrapper" data-top="true">
        <div data-bind="foreach: {data: gridRows, as: 'gridRow', afterRender:␣
↪afterRowRender.bind(grid)}">
            <div data-bind="grid_row">
                <div class="agenda-image">
                    <a data-bind="attr: {href: gridRow.val('document').href}" class=
↪"link-preview" target="_blank" data-tip-css='{"z-index": 2000}'>
                        <img data-bind="attr: {src: gridRow.val('document').icon,␣
↪alt: gridRow.val('document').text}" class="agenda-image">
                    </a>
                </div>
                <div class="agenda-description">
                    <span data-bind="grid_cell: 'upload_date'"></span> /
                    <span data-bind="grid_cell: 'is_latest'"></span>
                </div>
            </div>
        </div>
        <div class="jumbotron default-padding" data-bind="visible: gridRows().length␣
↪=== 0">
            <div data-template-id="ko_grid_no_results"></div>
        </div>
    </div>
</script>
```

Where `document.href` / `document.text` display values (str_fields) are generated at server-side in `AgendaGrid` Python class `get_row_str_fields()` method:

```
class AgendaGrid(KoGridView):

    model = AgendaFileRevision
    enable_switch_highlight = False
    grid_fields = [
        'document',
        'upload_date',
        'is_latest',
    ]
    allowed_sort_orders = [
        'upload_date',
    ]
    allowed_filter_fields = OrderedDict([
        ('upload_date', None),
        ('is_latest', None),
    ])

    def get_row_str_fields(self, obj, row=None):
        str_fields = super().get_row_str_fields(obj, row)
        str_fields['document'] = {
            'href': obj.document.url,
            'text': obj.file.basename
        }
        return str_fields
```

3.5 Action routing

Datatables (grids) support arbitrary number of built-in and custom actions besides standard CRUD. Thus grid requests do not use HTTP method routing such as PUT DELETE, which would be too limiting approach. All of grid actions are performed as HTTP POST; Django class-based view kwarg `action` value automatically generated by *UrlPath* is used to determine the current action:

```
from django_jinja_knockout.urls import UrlPath
from my_app.views import Model1Grid

# ... skipped ...
UrlPath(Model1Grid)(
    name='model1_grid',
    kwargs={'permission_required': 'my_app.change_model1'}
),
# ... skipped ...
```

Value of `action` kwarg is normalized (leading '/' are stripped) and is stored in `self.current_action_name` property of grid class instance at server-side. Key name of view kwargs dict used for grid action url name may be changed via Django grid class static property `action_kwarg`:

```
from django_jinja_knockout.views import KoGridView
from .models import Model1

class Model1Grid(KoGridView):

    action_kwarg = 'action'
    model = Model1
    # ... skipped ...
```

3.5.1 Server-side action routing

Django class-based view derived from views.KoGridView defines the list of available actions via `get_actions()` method. Defined actions are implemented via grid `action_NAME` method, where `NAME` is actual name of defined action, for example built-in action `'list'` is mapped to `GridActionsMixin.action_list()` method.

Django grid action method is called via AJAX so it is supposed to return one or more viewmodels via AJAX response, see *Client-side viewmodels and AJAX response routing*.

It might be either one of pre-defined viewmodels, like `{'view': 'alert'}` (see ioc.js for the basic list of viewmodels), or a grid viewmodel, which is routed to GridActions class (or it's child class) at client-side. Here is the example of action implementation:

```
from django_jinja_knockout.views import KoGridView
# ... skipped ...

class MemberGridCustomActions(KoGridView):

    # ... skipped ...
    def action_edit_note(self):
        member = self.get_object_for_action()
        note = self.request_get('note')
        modified_members = []
        if member.note != note:
            member.note = note
```

(continues on next page)

(continued from previous page)

```
            member.save()
            modified_members.append(member)
        if len(modified_members) == 0:
            return vm_list({
                'view': 'alert',
                'title': str(member.profile),
                'message': 'Note was not changed.'
            })
        else:
            return vm_list({
                'view': self.__class__.viewmodel_name,
                'update_rows': self.postprocess_qs(modified_members),
            })
```

views module has many built-in actions implemented, while club_app.views_ajax has some examples of custom actions code.

3.5.2 Client-side action routing

GridActions class is used both to invoke grid actions and to process their results.

GridActions class uses Actions as the base class for client-side viewmodel routing.

See *AJAX actions* for general introduction.

Invocation of action

Actions are invoked via Javascript `Actions.perform()` method:

```
Actions.perform = function(action, actionOptions, ajaxCallback)
```

- `'action' argument`: mandatory name of action as it is returned by Django grid `get_actions()` method;
- `'actionOptions' argument`: optional, custom parameters of action (usually Javascript object). These are passed to AJAX query request data. To add queryargs to some action, implement `queryargs_NAME` method, where `NAME` is actual name of action.
- `'ajaxCallback' argument`: optional function closure that will be executed when action is complete;

Interactive actions (action types `'button'` / `'iconui'`) are also represented by instances of KoGridAction Javascript class, which is used to setup CSS classes of bound DOM element button or iconui in ko_grid_body.htm.

When bound DOM element is clicked, these interactive actions invoke `Action.doAction()` method for particular visual action Knockout.js viewmodel, which calls chain of Grid / GridActions methods, finally issuing the same `Actions.perform()` method:

```
Actions.doAction = function(options, actionOptions)
```

- `'options' argument` of object type may pass key `'gridRow'` which value is the instance of GridRow class that will be used as interactive action target row. It is used by interactive actions that are related to specified grid row, such as *'edit_form' action*. Target row instance of GridRow will be stored in Grid instance `lastClickedKoRow` property, accessible in GridActions derived instance `this.grid.lastClickedKoRow` property in every `perform_NAME` method, eg.:

```
Model1GridActions.perform_my_action = function(queryArgs, ajaxCallback) {
    // Get raw value of last clicked grid row 'role' field.
    this.grid.lastClickedKoRow.getValue('role');
};
```

Javascript invocation of interactive action with specified target grid row when grid just loaded first time:

```
Model1Grid.onFirstLoad = function() {
    // Get instance of Action for specified action name:
    var editFormAction = this.getKoAction('edit_form');
    // Find row with pk value === 3, if any, in current page queryset:
    var targetKoRow = this.findKoRowByPkVal(3);
    // Check whether the row with pk value === 3 is in current page queryset:
    if (targetKoRow !== null) {
      // Execute 'edit_form' action for row with pk value === 3.
        editFormAction.doAction({gridRow: targetKoRow});
    }
};
```

- `'actionOptions' argument`: optional Javascript object that is passed to `Actions.perform()` as `actionOptions` argument, with the following optional keys:
  - `queryArgs`: extended action AJAX POST request arguments
  - `ajaxIndicator`: boolean, when the value is true, enables action target AJAX request `ladda` progress indicator (since v2.0)
  - custom keys may be used to pass data to alter the logic of the custom client-side actions

Grid class `.performAction()` method is used to invoke the datatable action:

```
Grid.performAction = function(actionName, actionType, actionOptions)
```

To bind the action invocation to datatable template button:

```
<button class="btn-choice btn-info club-edit-grid" data-bind="click: function() {␣
↪this.performAction('create_inline'); }">
    <span class="iconui iconui-plus"></span> Add row
</button>
```

Action queryargs

Here is the example of `'list'` action AJAX request queryargs population:

```
GridActions.queryargs_list = function(options) {
    return this.grid.getListQueryArgs();
};

// ... skipped ...

Grid.getListQueryArgs = function() {
    this.queryArgs['list_search'] = this.gridSearchStr();
    this.queryArgs['list_filter'] = JSON.stringify(this.queryFilters);
    return this.queryArgs;
};

// ... skipped ...
```

(continues on next page)

(continued from previous page)

```
Grid.listAction = function(callback) {
    this.actions.perform('list', {}, callback);
};

// ... skipped ...

Grid.searchSubstring = function(s) {
    if (typeof s !== 'undefined') {
        this.gridSearchStr(s);
    }
    this.queryArgs.page = 1;
    this.listAction();
};
```

Note that some keys of `queryArgs` object are populated in grid class own methods, while only the `'list_search'` and `'list_filter'` entries are set by `GridActions.queryargs_list()` method. It's easier and more convenient to implement `queryargs_NAME` method for that purpose.

For the reverse url of `Model1Grid` class-based view action `'list'`:

```
http://127.0.0.1:8000/model1-grid/list/
```

it will generate AJAX request queryargs similar to these:

```
page: 2
list_search: test
list_filter: {"role": 2}
csrfmiddlewaretoken: JqkaCTUzwpl7katgKiKnYCjcMpNYfjQc
```

which will be parsed by `KoGridView` derived instance `action_list()` method.

it is also possible to execute actions interactively with custom options, including custom queryArgs:

```
Model1Grid.onFirstLoad = function() {
    var myAction = this.getKoAction('my_custom_action');
    var targetKoRow = this.findKoRowByPkVal(10);
    myAction.doAction({
        myKoProp: 123,
        queryArgs: {rowId: targetKoRow.getPkVal()},
    });
};
```

When action is a purely client-side one implemented via GridActions derived instance `perform_NAME()` method, `actionOptions` may be used as client-side options, for example to pass initial values of Knockout.js custom template viewmodel properties.

Action AJAX response handler

To process AJAX response data returned from Django grid `action_NAME()` method, one has to implement GridActions derived class, where `callback_NAME()` method will be used to update client-side of grid. For example, AJAX `ModelForm`, generated by standard *'create_form' action* is displayed with:

```
import { ModelFormDialog } from '../../djk/js/modelform.js';
```

(continues on next page)

(continued from previous page)

```
GridActions.callback_create_form = function(viewModel) {
    viewModel.grid = this.grid;
    var dialog = new ModelFormDialog(viewModel);
    dialog.show();
};
```

grid meta-data (verbose names, field filters) are updated via:

```
GridActions.callback_meta = function(data) {
    if (typeof data.action_kwarg !== 'undefined') {
        this.setActionKwarg(data.action_kwarg);
    }
    this.grid.loadMetaCallback(data);
};
```

See standard `callback_*()` methods in GridActions class code and custom `callback_*()` methods in member-grid.js for more examples.

Client-side actions

It is also possible to perform actions partially or entirely at client-side. To implement this, one should define `perform_NAME()` method of GridActions derived class. It's used to display client-side BootstrapDialogs via ActionTemplateDialog -derived instances with underscore.js / knockout.js templates bound to current Grid derived instance:

```
import { inherit } from '../../djk/js/dash.js';
import { ActionTemplateDialog } from '../../djk/js/modelform.js';
import { Grid } from '../../djk/js/grid.js';
import { GridActions } from '../../djk/js/grid/actions.js';

MemberGridActions = function(options) {
    inherit(GridActions.prototype, this);
    this.init(options);
};

(function(MemberGridActions) {

    // Client-side invocation of the action.
    MemberGridActions.perform_edit_note = function(queryArgs, ajaxCallback) {
        var actionDialog = new ActionTemplateDialog({
            template: 'member_note_form',
            owner: this.grid,
            meta: {
                noteLabel: 'Member note',
                note: this.grid.lastClickedKoRow.getValue('note')
            },
        });
        actionDialog.show();
    };

    MemberGridActions.callback_edit_note = function(viewModel) {
        this.grid.updatePage(viewModel);
    };

})(MemberGridActions.prototype);
```

(continues on next page)

(continued from previous page)

```
MemberGrid = function(options) {
    inherit(Grid.prototype, this);
    this.init(options);
};

(function(MemberGrid) {

    MemberGrid.iocGridActions = function(options) {
        return new MemberGridActions(options);
    };

})(MemberGrid.prototype);
```

Where the `'member_note_form'` template could be like this, based on `ko_action_form` template located in ko_grid_body.htm:

```
<script type="text/template" id="member_note_form">
    <card-default">
        <card-body>
            <form class="ajax-form" enctype="multipart/form-data" method="post" role=
↪"form" data-bind="attr: {'data-url': gridActions.getLastActionUrl()}">
                <input type="hidden" name="csrfmiddlewaretoken" data-bind="value:␣
↪getCsrfToken()">
                <input type="hidden" name="pk_val" data-bind="value: getLastPkVal()">
                <div class="row form-group">
                    <label data-bind="text: meta.noteLabel" class="control-label col-
↪md-4" for="id_note"></label>
                    <div class="field col-md-6">
                        <textarea data-bind="textInput: meta.note" id="id_note" class=
↪"form-control autogrow" name="note" type="text"></textarea>
                    </div>
                </div>
            </form>
        </card-body>
    </card-default>
</script>
```

which may include any custom Knockout.js properties / observables bound to current grid instance. That allows to produce interactive client-side forms without extra AJAX requests.

See club_app.views_ajax, member_grid_custom_actions.htm and member-grid.js for full example of 'edit_note' action implementation.

3.5.3 Custom view kwargs

In some cases a grid may require additional kwargs to alter base queryset of grid. For example, if Django app has `Member` model related as many to one to `Club` model, grid that displays members of specified club id (foreign key value) requires additional `club_id` view kwarg in `urls.py`:

```
# ... skipped ...
UrlPath(ClubMemberGrid)(
    name='club_member_grid',
    # Note that 'action' arg will be appended automatically,
    # thus we have not specified it.
```

(continues on next page)

(continued from previous page)

```
    # However one may specify it to re-order capture patterns:
    # args=['action', 'club_id'],
    args=['club_id],
    kwargs={'permission_required': 'my_app.change_member'}
),
# ... skipped ...
```

Then, grid class may filter base queryset according to received `club_id` view kwargs value:

```
class ClubMemberGrid(KoGridView):

    model = Member
    # ... skipped ...
    def get_base_queryset(self):
        return super().get_base_queryset().filter(club_id=self.kwargs['club_id'])
```

The component template should provide the options with specified view kwargs values. One have to pass proper initial `pageRouteKwargs` `club_id` key / value when rendering the template:

```
{{ ko_grid(
    grid_options={
        'pageRoute': 'club_member_grid',
        'pageRouteKwargs': {'club_id': club_id},
    },
    dom_attrs={
        'id': 'club_member_grid'
    }
) }}
```

This way grid will have custom list of club members according to `club_id` view kwarg value.

- Version 2.0 has the improved support for automatic filling of related grid `pageRouteKwargs` values from the current grid view via pageRouteKwargsKeys. See pageRouteKwargsKeys example.

Because foreign key widgets utilize views.KoGridView and Grid classes, base querysets of foreign key widgets may be filtered as well:

```
class Model1Grid(KoGridView):

    allowed_filter_fields = OrderedDict([
        # Autodetect filter type.
        ('field_1', None),
        ('model2_fk', {
            # optional classPath
            # 'classPath': 'Model2Grid',
            'pageRoute': 'model2_fk_grid',
            'pageRouteKwargs': {'type': 'custom'},
            'searchPlaceholder': 'Search for Model2 values',
        }),
    ])
```

3.6 Standard actions

Datatables (grids) views.KoGridView are based on generic views.ActionsView class which allows to interact with any client-side AJAX component. See *AJAX actions* for more info.

By default views.KoGridView and GridActions offer many actions that can be applied either to the whole grid or to one / few columns of grid. Actions can be interactive (represented as UI elements) and non-interactive. Actions can be executed as one or multiple AJAX requests or be partially / purely client-side.

views.ActionsView / views.GridActionsMixin .get_actions() method returns dict defining built-in actions available. Top level of that dict is current `action type`.

Action definitions do not require to have `'enabled': True` to be set explicitly. The action is considered to be enabled by default. That shortens the list of action definitions. To conditionally disable action, set `enabled` key of action definition dict to `False` value. See built-in .get_actions() method for the example.

Let's see which action types are available and their associated actions.

3.6.1 Action type 'built_in'

Actions that are supposed to be used internally without generation of associated invocation elements (buttons, iconuis).

'meta' action

Returns AJAX response data:

- the list of allowed sort orders for grid fields (`'sortOrders'`);
- flag whether search field should be displayed (`'meta.hasSearch'`);
- verbose name of associated Django model (`'meta.verboseName'` / `'meta.verboseNamePlural'`);
- verbose names of associated Django model fields and related models verbose field names, see *Nested verbose field names* (`'meta.listOptions'` / `'meta.fkNestedListOptions'`);
- name of primary key field `'meta.pkField'` that is used in different parts of Grid to address grid rows;
- list of defined grid actions, See *Standard actions*, *Action routing*, *Custom action types*;
- allowed grid fields (list of grid columns), see *Grid configuration*;
- field filters which will be displayed in top navigation bar of grid client-side component via `'ko_grid_nav'` underscore.js template, see *Filter fields*;

Custom Django grid class-based views derived from `KoGridView` may return extra meta properties for custom client-side templates. These will be updated "on the fly" automatically with standard client-side GridActions class `callback_meta()` method.

Custom actions also can update grid meta by calling client-side Grid class `updateMeta()` method directly:

```
Model1GridActions.callback_approve_user = function(viewModel) {
    this.grid.updateMeta(viewModel.meta);
    // Do something more...
};
```

See *Action AJAX response handler* how meta is updated in client-side AJAX callback.

See *Modifying visual layout of grid* how to override client-side underscore.js / Knockout.js templates.

'list' action

Returns AJAX response data with the list of currently paginated grid rows, both "raw" database field `values` list and their optional `str_fields` formatted list counterparts. While some grids datatables may do not use `str_fields` at all, complex formatting of local date / time / financial currency Django model field values requires `str_fields` to be generated at server-side.

`str_fields` also are used for nested representation of fields (displaying foreign related models fields list in one grid cell).

`str_fields` are populated at server-side for each grid row via views.KoGridView class . `get_row_str_fields()` method and are converted to client-side `display values` in GridRow class `display()` method.

Both methods can be overridden in ancestor classes to customize field values output. When associated Django model has *get_str_fields()* method defined, it will be used to get `str_fields` for each row by default.

'meta_list' action

By default `meta` action is not performed in separate AJAX query, rather it's combined with `list` action into one AJAX request via `meta_list` action. Such way it saves HTTP traffic and reduces server load. However, in some cases, grid filters or sorting orders has to be set up with specific choices before `'list'` action is performed. That is required to load grid with initially selected field filter choices or to change default sorting.

'meta_list' action and custom initial field filters

If Django grid class specifies the list of initially selected field filter choices as `active_choices`:

```
class MemberGridTabs(MemberGrid):

    template_name = 'member_grid_tabs.htm'

    allowed_filter_fields = OrderedDict([
        ('profile', None),
        ('last_visit', None),
        # Next choices of 'plays' field filter will be set when grid loads.
        ('plays', {'active_choices': [Member.SPORT_BADMINTON, Member.SPORT_SQUASH]}),
        ('role', None),
        ('is_endorsed', None),
    ])
```

To make sure `ClubMemberGrid` action `'list'` respects `allowed_filter_fields` definition of `['plays']['active_choices']` default choices values, one has to turn on client-side Grid class `options.separateMeta` value to `true` either with *ko_grid() macro* grid_options:

```
{{ ko_grid(
    grid_options={
        'pageRoute': 'club_member_grid',
        'separateMeta': True,
    },
    dom_attrs={
        'id': 'club_member_grid'
    }
) }}
```

by setting Django grid class `grid_options` dict `separateMeta` key value:

```
class ClubMemberGrid(KoGridView):

    model = ClubMember
    # ... skipped ...

    grid_options = {
        'classPath': 'ClubMemberGrid',
        'separateMeta': True,
    }
```

by overriding Django grid class `get_grid_options()` method:

```
class ClubMemberGrid(KoGridView):

    model = ClubMember
    # ... skipped ...

    @classmethod
    def get_grid_options(cls):
        return {
            'classPath': 'ClubMemberGrid',
            'separateMeta': True,
        }
```

via overloading of client-side Grid by custom class:

```
import { inherit } from '../../djk/js/dash.js';
import { Grid } from '../../djk/js/grid.js';

ClubMemberGrid = function(options) {
    inherit(Grid.prototype, this);
    /**
     * This grid has selected choices for query filter 'plays' by default,
     * thus it requires separate 'list' action after 'meta' action,
     * instead of joint 'meta_list' action.
     */
    options.separateMeta = true;
    this.init(options);
};
```

When `ClubMemberGrid options.separateMeta` is `true`, `meta` action will be issued first, setting `'plays'` filter selected choices, then `'list'` action will be performed separately, respecting these filter choices.

Without `options.separateMeta`, `ClubMemberGrid plays` filter will be visually highlighted as selected, but the first (initial) `list` action will incorrectly return unfiltered rows.

'meta_list' action and custom initial ordering

When one supplies custom initial ordering of rows that does not match default Django model ordering:

```
{{ ko_grid(
    grid_options={
        'pageRoute': 'club_grid_with_action_logging',
        'defaultOrderBy': {'foundation_date': '-'},
    },
    dom_attrs={
```

(continues on next page)

(continued from previous page)

```
        'id': 'club_grid'
    }
) }}
```

Grid `options.separateMeta` will be enabled automatically and does not require to be explicitly passed in.

See club_app.views_ajax, club_grid_with_action_logging.htm for fully featured example.

'meta list' action preload

Sometimes one html page may include large number of Grid components. When loaded, it would cause large number of simultaneous AJAX requests, slowing the initial load performance and causing increased server load. One may preload the initial *'meta_list' action* request at server-side by setting views.KoGridView `grid_options` dictionary attribute `preload_meta_list` to `True`:

```
class ClubMemberGrid(KoGridView):

    model = ClubMember
    # ... skipped ...

    grid_options = {
        'preload_meta_list': True,
    }
```

Server-side preloaded result of *'meta_list' action* later will be passed to client-side datatable (grid) via *ko_grid() macro* `preloadedMetaList` option.

'meta list' action preload may fail in the following cases:

- views.KoGridView which use `view.kwargs` keys / values different from embedding *ko_grid() macro* `view.kwargs`

Thus it is disabled by default for the compatibility purposes.

'update' action

This action is not called directly internally but is implemented for user convenience. It performs the same ORM query as *'list' action*, but instead of removing all existing rows and replacing them with new ones, it compares old rows and new rows, deletes non-existing rows, keeps unchanged rows intact, adding new rows while highlighting them.

This action is useful to update related grid rows after current grid performed some actions that changed related models of the related grid.

Open club-grid.js to see the example of manually executing `ActionGrid` *'update' action* on the completion of `ClubGrid` *'save_inline' action* and *'delete_confirmed' action*:

```
(function(ClubGridActions) {

    ClubGridActions.updateActionGrid = function() {
        // Get instance of ActionGrid.
        var actionGrid = $('#action_grid').component();
        if (actionGrid !== null) {
            // Update ActionGrid.
            actionGrid.actions.perform('update');
        }
    };
```

(continues on next page)

(continued from previous page)

```
    ClubGridActions.callback_save_inline = function(viewModel) {
        this._super._call('callback_save_inline', viewModel);
        this.updateActionGrid();
    };

    ClubGridActions.callback_delete_confirmed = function(viewModel) {
        this._super._call('callback_delete_confirmed', viewModel);
        this.updateActionGrid();
    };

})(ClubGridActions.prototype);
```

'save_form' action

Performs validation of AJAX submitted form previously created via *'create_form' action* / *'edit_form' action*, which will either create new grid row or edit existing grid row.

Each grid row represents an instance of associated Django model. Form rows are bound to specified Django `ModelForm` automatically, one has to set value of grid class `form` static property:

```
from django_jinja_knockout.views import KoGridView
from .models import Model1
from .forms import Model1Form

class Model1Grid(KoGridView):

    model = Model1
    form = Model1Form
    # ... skipped ...
```

Alternatively, one may define factory methods, which would bind different Django `ModelForm` classes to *'create_form' action* and *'edit_form' action*. That allows to have different set of bound model fields when creating and editing grid row Django models:

```
from django_jinja_knockout.views import KoGridView
from .models import Model1
from .forms import Model1CreateForm, Model1EditForm

class Model1Grid(KoGridView):

    model = Model1

    def get_create_form(self):
        return Model1CreateForm

    def get_edit_form(self):
        return Model1EditForm
```

'save_form' action will:

- Display AJAX form errors in case there are `ModelForm` validation errors.
- Create new model instance / add new row to grid when invoked via *'create_form' action*.
- Update existing model instance / grid row, when invoked via *'edit_form' action*.

Grid.updatePage() method

To automatize grid update for AJAX submitted action, the following optional JSON properties could be set in AJAX viewmodel response:

- `'append_rows'`: list of rows which should be appended to current grid page to the bottom;
- `'prepend_rows'`: list of rows which should be prepended to current grid page from the top;
- `'update_rows'`: list of rows that are updated, so their display needs to be refreshed;
- `'deleted_pks'`: list of primary key values of Django models that were deleted in the database thus their rows have to be visually removed from current grid page;

Standard grid action handlers (as well as custom action handlers) may return AJAX viewmodel responses with these JSON keys to client-side action viewmodel response handler, issuing multiple CRUD operations at once. For example GridActions class `callback_save_form()` method:

```
GridActions.callback_save_form = function(viewModel) {
    this.grid.updatePage(viewModel);
};
```

See also views.ModelFormActionsView class `action_save_form()` and views.GridActionsMixin class `action_delete_confirmed()` methods for server-side part example.

Client-side part of multiple CRUD operation is implemented in grid.js Grid class `updatePage()` method.

`'update_rows'` response processing internally uses GridRow class `.matchesPk()` method to check whether two grid rows match the same Django model instance, instead of direct `pkVal` comparison.

It is possible to override `.matchesPk()` method in child class for custom grid rows matching - for example in grids datatables with RAW query `LEFT JOIN` which may have multiple rows with the same `pkVal` == `null`, while being distinguished by another field values.

'save_inline' action

Similar to *'save_form' action* described above, this action is an AJAX form submit handler for *'create_inline' action* / *'edit_inline' action*. These actions generate BootstrapDialog with `FormWithInlineFormsets` AJAX submittable form instance bound to current grid row via views.KoGridView class `form_with_inline_formsets` static property:

```
from django_jinja_knockout.views import KoGridView
from .models import Model1
from .forms import Model1FormWithInlineFormsets

class Model1Grid(KoGridView):

    model = Model1
    form_with_inline_formsets = Model1FormWithInlineFormsets
    # ... skipped ...
```

Alternatively, one may define factory methods, which allows to bind different `FormWithInlineFormsets` classes to *'create_inline' action* / *'edit_inline' action* target grid row (Django model):

```
from django_jinja_knockout.views import KoGridView
from .models import Model1
from .forms import Model1CreateFormWithInlineFormsets,␣
↪Model1EditFormWithInlineFormsets
```

(continues on next page)

(continued from previous page)

```
class Model1Grid(KoGridView):

    model = Model1

    def get_create_form_with_inline_formsets(self):
        return Model1CreateFormWithInlineFormsets

    def get_edit_form_with_inline_formsets(self):
        return Model1EditFormWithInlineFormsets
```

These methods should return classes derived from `django_jinja_knockout.forms.FormWithInlineFormsets` class (see *Forms*).

'delete_confirmed' action

Deletes one or more grid rows via their pk values previously submitted by *'delete' action*. To selectively disable deletion of some grid rows, one may implement custom `action_delete_is_allowed` method in the Django grid class:

```
class MemberGridTabs(MemberGrid):

    template_name = 'member_grid_tabs.htm'
    enable_deletion = True

    allowed_filter_fields = OrderedDict([
        ('profile', None),
        ('last_visit', None),
        # Next choices of 'plays' field filter will be set when grid loads.
        ('plays', {'active_choices': [Member.SPORT_BADMINTON, Member.SPORT_SQUASH]}),
        ('role', None),
        ('is_endorsed', None),
    ])

    # Do not allow to delete Member instances with role=Member.ROLE_FOUNDER:
    def action_delete_is_allowed(self, objects):
        # ._clone() is required because original pagination queryset is passed as␣
↪objects argument.
        qs = objects._clone()
        return not qs.filter(role=Member.ROLE_FOUNDER).exists()
```

See club_app.views_ajax for full-featured example.

3.6.2 Action type 'button'

These actions are visually displayed as buttons and manually invoked via button click. With the default underscore.js templates these buttons are located at top navbar of the grid (datatable). Usually type `'button'` actions are not targeted to the single row, but are supposed either to create new rows or to process the whole queryset / list of rows.

However, when Grid -derived class instance has visible row selection enabled via `init()` method `options.showSelection` = `true` and / or `options.selectMultipleRows` = `true`, the button action could be applied to the selected row(s) as well.

New actions of `button` type may be added by overriding .get_actions() method of views.KoGridView derived class, then extending client-side GridActions class to implement custom `'callback_'` method (see *Client-side actions*

for more info).

'create_form' action

Server-side part of this action renders AJAX-powered Django `ModelForm` instance bound to new Django grid model.

Client-side part of this action displays rendered `ModelForm` as `BootstrapDialog` modal dialog. Together with *'save_form' action*, which serves as callback for this action, it allows to create new grid rows (new Django model instances).

This action is enabled (and thus UI button will be displayed in grid component navbar) when Django grid class-based view has assigned `ModelForm` class specified as:

```
from django_jinja_knockout.views import KoGridView
from .models import Model1
from .forms import Model1Form

class Model1Grid(KoGridView):

    model = Model1
    form = Model1Form
    # ... skipped ...
```

Alternatively, one may define factory methods, which would bind different Django `ModelForm` classes to *'create_form' action* and *'edit_form' action*. That allows to have different set of bound model fields when creating and editing grid row Django models:

```
from django_jinja_knockout.views import KoGridView
from .models import Model1
from .forms import Model1CreateForm, Model1EditForm

class Model1Grid(KoGridView):

    model = Model1

    def get_create_form(self):
        return Model1CreateForm

    def get_edit_form(self):
        return Model1EditForm
```

When one would look at server-side part of views.GridActionsMixin class `action_create_form()` method source code, there is last_action viewmodel key with value `'save_form'` returned to Javascript client-side:

```
# ... skipped ...
return vm_list({
    'view': self.__class__.viewmodel_name,
    'last_action': 'save_form',
    'title': format_html('{}: {}',
        self.get_action_local_name(),
        self.get_model_meta('verbose_name')
    ),
    'message': form_html
})
```

Viewmodel action overrides

Viewmodel's last_action optional key is used in client-side Javascript GridActions class `respond()` method to override the name of last executed action from current `'create_form'` to `'save_form'`. The name of last executed action is used to generate last action url value in grid templates (ko_grid_body.htm) / component templates via Actions class getLastActionUrl() method.

It is then used in client-side Javascript ModelFormDialog class `getButtons()` method `submit` button event handler to perform *'save_form' action* when that button is clicked by end-user, instead of already executed *'create_form' action*, which already generated AJAX model form and displayed it using ModelFormDialog instance.

Viewmodel's callback_action optional key is used in client-side Javascript to override the action callback method. Some viewmodel callbacks may share the same action callback method (handler) to reduce duplication of code. Since v2.0 views.ModelFormActionsView .vm_form() also supports optional specifying ot callback_action value.

See *Action AJAX response handler* for more info on action client-side AJAX callbacks.

'create_inline' action

Server-side part of this action renders AJAX-powered forms.FormWithInlineFormsets instance bound to new Django grid model.

Client-side part of this action displays rendered `FormWithInlineFormsets` as `BootstrapDialog` modal. Together with *'save_inline' action*, which serves as callback for this action, it allows to create new grid rows (new Django model instances) while also adding one to many related models instances via one or multiple inline formsets.

This action is enabled (and thus UI button will be displayed in grid component navbar) when Django grid class-based view has assigned forms.FormWithInlineFormsets derived class (see *Forms* for more info about that class). It should be specified as:

```
from django_jinja_knockout.views import KoGridView
from .models import Model1
from .forms import Model1FormWithInlineFormsets

class Model1Grid(KoGridView):

    model = Model1
    form_with_inline_formsets = Model1FormWithInlineFormsets
    # ... skipped ...
```

Alternatively, one may define factory methods, which allows to bind different `FormWithInlineFormsets` derived classes to *'create_inline' action* new row and *'edit_inline' action* existing grid row (Django model):

```
from django_jinja_knockout.views import KoGridView
from .models import Model1
from .forms import Model1CreateFormWithInlineFormsets,␣
↪Model1EditFormWithInlineFormsets

class Model1Grid(KoGridView):

    model = Model1

    def get_create_form_with_inline_formsets(self):
        return Model1CreateFormWithInlineFormsets

    def get_edit_form_with_inline_formsets(self):
        return Model1EditFormWithInlineFormsets
```

- Server-side part of this action overrides the name of last executed action by setting AJAX response viewmodel `last_action` key to `save_inline` value, which specifies the action of BootstrapDialog form modal button. See *'create_form' action* description for more info about `last_action` key.
- views.KoGridInline class is the same views.KoGridView class only using different value of `template_name` class property pointing to Jinja2 template which includes formsets.js by default.
- See club_app.views_ajax for fully featured example of `KoGridView form_with_inline_formsets` usage.

3.6.3 Action type 'button_footer'

Works exactly like *Action type 'button'*, however it displays grid action buttons below the grid rows, instead of the grid navigation bar.

There is no built-in actions of this type. Custom actions of this type may be implemented in `KoGridView` inherited classes to change button display layout.

3.6.4 Action type 'click'

These actions are designed to process already displayed grid row, associated to existing Django model.

- By default there is no active click actions, so clicking grid row does nothing.
- When there is only one click action enabled, it will be executed immediately after end-user clicking of target row.
- When there is more than one click actions enabled, Grid will use special version of BootstrapDialog wrapper ActionsMenuDialog to display menu with clickable buttons to select one action from the list of available ones.

Cell actions

Since v2.0, `click` type of action optionally supports specifying Grid row cells as action target, which makes possible to define separate `click` actions for each / multiple cells of Grid row:

```
from django_jinja_knockout.views import KoGridView

class PriceGrid(KoGridView):

grid_fields = [
    [
        'step_price_percent',
        'min_growth_percent',
        'max_growth_percent',
    ],
    [
        'decline_threshold',
        'growth_threshold',
    ]
]

def get_actions(self):
    actions = super().get_actions()
    nested_update(actions, {
        'click': {
            'edit_price_percent_form': {
```

(continues on next page)

(continued from previous page)

```
                'localName': 'Edit price percentage change',
                'css': 'btn-default',
                'cells': [
                    'step_price_percent',
                    'min_growth_percent',
                    'max_growth_percent',
                ],
            },
            'edit_threshold_form': {
                'localName': 'Edit price threshold',
                'css': 'btn-default',
                'cells': [
                    'decline_threshold',
                    'growth_threshold',
                ],
            },
        }
    })
    return actions
```

Such way, `edit_price_percent_form` and `edit_threshold_form` actions will be performed only for the selected `grid_fields` cells names.

See Grid.getCellActions() method for the details of the client-side cell actions implementation.

'edit_form' action

This action is enabled when current Django grid class inherited from views.KoGridView class has class property `form` set to specified Django `ModelForm` class used to edit grid row via associated Django model:

```
from django_jinja_knockout.views import KoGridView
from .models import Model1
from .forms import Model1Form

class Model1Grid(KoGridView):

    model = Model1
    form = Model1Form
```

Alternatively, one may define `get_edit_form()` Django grid method to return `ModelForm` class dynamically.

Server-side of this action is implemented via views.GridActionsMixin class `action_edit_form()` method. It returns AJAX response with generated HTML of `ModelForm` instance bound to target grid row Django model instance. Returned viewmodel `last_action` property value is set to `'save_form'`, to override GridActions class `lastActionName` property.

Client-side of this action uses ModelFormDialog to display generated `ModelForm` html and to submit AJAX form to *'save_form' action*.

'edit_inline' action

This action is enabled when current Django grid class has defined class property `form_with_inline_formsets` set to forms.FormWithInlineFormsets derived class used to edit grid row and it's foreign relationships via Django inline formsets (see *Forms*):

```
from django_jinja_knockout.views import KoGridView
from .models import Model1
from .forms import Model1FormWithInlineFormsets

class Model1Grid(KoGridView):

    model = Model1
    form_with_inline_formsets = Model1FormWithInlineFormsets
```

Alternatively, one may define `get_edit_form_with_inline_formsets()` Django grid method to return `FormWithInlineFormsets` derived class dynamically.

Server-side of this action is implemented in views.GridActionsMixin class `action_edit_inline()` method. It returns AJAX response with generated HTML of `FormWithInlineFormsets` instance bound to target grid row Django model instance. Returned viewmodel `last_action` property value is set to `'save_inline'`, to override GridActions class `lastActionName` property.

Client-side of this action uses ModelFormDialog to display generated `FormWithInlineFormsets` html and to submit AJAX form to *'save_inline' action*.

See *Implementing custom grid row actions* section how to implement custom actions of `'click'` and `'iconui'` types.

3.6.5 Action type 'pagination'

Actions of `pagination` type adds iconui buttons directly to pagination control of the current grid (datatable). These actions may be applied to the whole grid or to the selected grid rows, similarly to *Action type 'button'*.

The following built-in actions of this type are implemented:

'rows_per_page' action

Allows to select the number of rows per grid (datatable) page via Bootstrap dialog. This may be useful when one wants to observe more rows or to select more rows to perform subsequent mass-rows actions. When number of displayed rows is changed, it tries to keep the current top row visible.

By default it allows to chose 1x to 5x steps from the current *OBJECTS_PER_PAGE*. It can be overridden in child class by changing default `'range'` settings of action definition:

```
from django_jinja_knockout.views import KoGridView
from django_jinja_knockout.utils.sdv import nested_update
from my_app.models import Member
# ... skipped ...

class MemberGrid(KoGridView):

    model = Member
    # ... skipped ...

    def get_actions(self):
        actions = super().get_actions()
        nested_update(actions, {
            'pagination': {
                'rows_per_page': {
                    'range': {
                        'min': 10,
```

(continues on next page)

(continued from previous page)

```
                        'max': 100,
                        'step': 10,
                    },
                },
            }
        })
        return actions
```

Be aware that enabling large `'rows_per_page'` value may greatly increase server load. For high-load sites this action could be conditionally disabled (eg. for anonymous users), by setting key `'enabled'` to `False`, such as for every another action out there.

'switch_highlight' action

Cycles between the defined highlight modes of grid. The following built-in highlight modes are available:

```
'none', 'cycleColumns', 'cycleRows', 'linearRows'
```

Default highlight mode is set via overriding current grid (datatable) like this:

```
class MemberGrid(KoGridView):

    grid_options = {
        'highlightMode': 'cycleColumns',
    }

    # or, like this:
    @classmethod
    def get_grid_options(cls):
        grid_options = super().get_grid_options()
        grid_options['highlightMode'] = 'cycleColumns'
        return grid_options
```

It is possible to disable some of highlight modes or to define new ones via *Client-side class overriding* and providing custom list of `highlightModeRules` values in overridden (inherited) grid (datatable) class.

Traditional (non-AJAX) request views.list.ListSortingView also supports `highlight_mode` attribute with similar highlighting settings, but no dynamical change of current highlight mode.

3.6.6 Action type 'iconui'

These actions are designed to process already displayed grid (datatable) row, associated to existing Django model. Their implementation is very similar to *Action type 'button'*, but instead of clicking at any place of row, these actions are visually displayed as iconui links in separate columns of grid.

iconui actions are rendered in the single column of datatable, instead of each action per column for better utilization of the display space.

By default there is no `iconui` type actions enabled. But there is one standard action of such type implemented for `KoGridView`: *'delete' action*.

'delete' action

This action deletes grid row (Django model instance) but is disabled by default. To enable grid row deletion, one has to set Django grid class property `enable_deletion` value to `True`:

```
from django_jinja_knockout.views import KoGridView
from .models import Manufacturer
from .forms import ManufacturerForm

class ManufacturerGrid(KoGridView):

    model = Manufacturer
    form = ManufacturerForm
    enable_deletion = True
    grid_fields = '__all__'
    allowed_sort_orders = '__all__'
    allowed_filter_fields = OrderedDict([
        ('direct_shipping', None)
    ])
    search_fields = [
        ('company_name', 'icontains'),
    ]
```

This grid also specifies `form` class property, which enables all CRUD operations with `Manufacturer` Django model.

Note that *'delete_confirmed' action* is used as success callback for *'delete' action* and usually both are enabled or disabled per grid class - if one considers to check the user permissions:

```
from django_jinja_knockout.views import KoGridView
from .models import Manufacturer
from .forms import ManufacturerForm

class ManufacturerGrid(KoGridView):

    model = Manufacturer
    form = ManufacturerForm

    def get_actions(self):
        enable_deletion = self.request.user.has_perm('club_app.delete_manufacturer')
        actions = super().get_actions()
        actions['iconui']['delete']['enabled'] = enable_deletion
        actions['built_in']['delete_confirmed']['enabled'] = enable_deletion
        return actions
```

views.KoGridView has built-in support permission checking of deletion rights for selected rows lists / querysets. See *'delete_confirmed' action* for the primer of checking delete permissions per row / queryset.

The action itself is defined in views.GridActionsMixin class:

```
OrderedDict([
    # Delete one or many model object.
    ('delete', {
        'localName': _('Remove'),
        'css': 'iconui-remove',
        'enabled': self.enable_deletion
    })
])
```

See *Implementing custom grid row actions* section how to implement custom actions of `'click'` and `'iconui'` types.

Imagine one grid having custom iconui action defined like this:

```
class MemberGrid(KoGridView):
    model = Member
    form = MemberFormForGrid

    def get_actions(self):
        actions = super().get_actions()
        actions['iconui']['quick_endorse'] = {
            'localName': _('Quick endorsement'),
            'css': 'iconui-cloud-upload',
        }
        return actions
```

Grid rows may selectively enable / disable their actions on the fly with visual updates. It is especially important to actions of type `'iconui'`, because these are always visible in grid columns.

To implement online visibility update of grid row actions one should override client-side GridRow class `hasEnabledAction()` method like this:

```
import { inherit } from '../../djk/js/dash.js';
import { Grid } from '../../djk/js/grid.js';
import { GridRow } from '../../djk/js/grid/row.js';

MemberGridRow = function(options) {
    inherit(GridRow.prototype, this);
    this.init(options);
};

(function(MemberGridRow) {

    // .. skipped ...

    MemberGridRow.hasEnabledAction = function(action) {
        if (action.name === 'quick_endorse' && this.values['is_endorsed'] === true) {
            return false;
        }
        return true;
    };

})(MemberGridRow.prototype);

MemberGrid = function(options) {
    inherit(Grid.prototype, this);
    this.init(options);
};

(function(MemberGrid) {

    // .. skipped ...

    MemberGrid.iocRow = function(options) {
        return new MemberGridRow(options);
    };

})(MemberGrid.prototype);
```

This way action of `iconui` type with `'quick_endorse'` name will be displayed as link only when associated Django model instance field name `is_endorsed` has the value `true`. Otherwise the link to action will be hidden. Updating grid rows via Grid class `updatePage()` method will cause visual re-draw of available grid rows actions display.

In case action is not purely client-side (has `callback_NAME`), additional permission check should also be performed with server-side Django grid `action_NAME` method.

- See *'save_form' action* and *Grid.updatePage() method* how to use `updatePage()` in your grids datatables.
- See *Action AJAX response handler* for explanation of server-side actions vs pure client-side actions.
- See fully-featured example in member-grid.js / club_app.views_ajax.

Implementing custom grid row actions

First step to add new action is to override `get_actions()` method in Django grid class. Let's create new action `'ask_user'` of `'click'` type:

```
from django_jinja_knockout.views import KoGridView
from .models import Profile
from django.utils.translation import ugettext as _

class ProfileGrid(KoGridView):

    model = Profile
    # ... skipped ...

    def get_actions(self):
        actions = super().get_actions()
        action_type = 'click'
        actions[action_type]['ask_user'] = {
            'localName': _('Ask user'),
            'css': 'btn-warning',
        }
        return actions
```

To create new action `'ask_user'` of `'iconui'` type instead:

```
from django_jinja_knockout.views import KoGridView
from .models import Profile
from django.utils.translation import ugettext as _

class ProfileGrid(KoGridView):

    model = Profile
    # ... skipped ...

    def get_actions(self):
        actions = super().get_actions()
        action_type = 'iconui'
        actions[action_type]['ask_user'] = {
            'localName': _('Ask user'),
            'css': 'iconui-user',
        }
        return actions
```

Next step is to implement newly defined action server-side and / or it's client-side parts.

If one wants to bind multiple different Django `ModelForm` edit actions to grid, the server-side of the custom action might be implemented like this:

```
from django_jinja_knockout.views import KoGridView
from .models import Profile
from .forms import ProfileForm, ProfileAskUserForm

class ProfileGrid(KoGridView):

    model = Profile
    # This form will be used for actions 'create_form' / 'edit_form' / 'save_form'.
    form = ProfileForm
    # ... skipped ...

    # Based on GridActionsMixin.action_edit_form() implementation.
    def action_ask_user(self):
        # Works with single row, thus we are getting single model instance.
        obj = self.get_object_for_action()
        # This form will be used for actions 'ask_user' / 'save_form'.
        form = ProfileAskUserForm(instance=obj)
        return self.vm_form(
            form, self.render_object_desc(obj), {'pk_val': obj.pk}
        )
```

- Actions which work with single objects (single row) should use `get_object_for_action()` method to obtain Django model object instance of target grid row.
- Actions which work with lists / querysets of objects (multiple rows) should use `get_queryset_for_action()` method to obtain the whole queryset of selected grid rows. See `action_delete()` / `action_delete_confirmed()` methods code in views.GridActionsMixin class for example.

ModelFormDialog class will be used to render AJAX-generated Django `ModelForm` at client-side. One has to inherit `ProfileGridActions` from `GridActions` and define custom action's own `callback_NAME`:

```
import { ModelFormDialog } from '../../djk/js/modelform.js';
import { GridActions } from '../../djk/js/grid/actions.js';

ProfileGridActions.callback_ask_user = function(viewModel) {
    viewModel.grid = this.grid;
    var dialog = new ModelFormDialog(viewModel);
    dialog.show();
};
```

- see *Action AJAX response handler* for more info on action client-side AJAX callbacks.

Completely different way of generating form with pure client-side underscore.js / Knockout.js templates for custom action (no AJAX callback is required to generate form HTML) is implemented in *Client-side actions* section of the documentation.

3.7 ForeignKeyGridWidget

widgets.ForeignKeyGridWidget is similar to `ForeignKeyRawIdWidget` implemented in django.contrib.admin.widgets, but is easier to integrate into non-admin views. It provides built-in sorting / filters and optional CRUD editing of related model rows, because it is based on the code of views.KoGridView and grid.js.

Let's imagine we have two Django models with one to many relationships:

```
from django.db import models

class Profile(models.Model):
    first_name = models.CharField(max_length=30, verbose_name='First name')
    last_name = models.CharField(max_length=30, verbose_name='Last name')
    birth_date = models.DateField(db_index=True, verbose_name='Birth date')
    # ... skipped ...

class Member(models.Model):
    profile = models.ForeignKey(Profile, verbose_name='Sportsman')
    # ... skipped ...
```

- See club_app.models for complete definitions of models.

Now we will define `MemberForm` bound to `Member` model:

```
from django import forms
from django_jinja_knockout.widgets import ForeignKeyGridWidget
from django_jinja_knockout.forms BootstrapModelForm
from .models import Member

class MemberForm(BootstrapModelForm):

    class Meta:
        model = Member
        fields = '__all__'
        widgets = {
            'profile': ForeignKeyGridWidget(model=Profile, grid_options={
                'pageRoute': 'profile_fk_widget',
                'dialogOptions': {'size': 'size-wide'},
                # Foreign key filter options will be auto-detected, but these could
                # have been defined explicitly when needed:
                # 'fkGridOptions': {
                #     'user': {
                #         'pageRoute': 'user_fk_widget',
                #     },
                # },
                # Override default search field label (optional):
                'searchPlaceholder': 'Search user profiles',
            }),
            'plays': forms.RadioSelect(),
            'role': forms.RadioSelect()
        }
```

Any valid Grid constructor option can be specified as `grid_options` argument of `ForeignKeyGridWidget`, including nested foreign key widgets and filters (see commented `fkGridOptions` section).

- See club_app.forms for complete definitions of forms.

To bind `MemberForm` `profile` field widget to actual `Profile` model grid, we have specified class-based view url name (route) of our widget as `'pageRoute'` argument value `'profile_fk_widget'`.

Now to implement the class-based grid view once for any possible ModelForm with `'profile'` foreign field:

```
from django_jinja_knockout import KoGridView
from .models import Profile
```

(continues on next page)

(continued from previous page)

```
class ProfileFkWidgetGrid(KoGridView):

    model = Profile
    grid_fields = ['first_name', 'last_name']
    allowed_sort_orders = '__all__'
    search_fields = [
        ('first_name', 'icontains'),
        ('last_name', 'icontains'),
    ]
```

We can set `ProfileFkWidgetGrid` attribute `form = ProfileForm`, so `Profile` foreign key widget will support in-place CRUD AJAX actions, allowing to create new Profiles just in place before the related `MemberForm` instance is saved:

```
from django_jinja_knockout import KoGridView
from .models import Profile
from .forms import ProfileForm

class ProfileFkWidgetGrid(KoGridView):

    model = Profile
    form = ProfileForm
    enable_deletion = True
    grid_fields = ['first_name', 'last_name']
    allowed_sort_orders = '__all__'
    search_fields = [
        ('first_name', 'icontains'),
        ('last_name', 'icontains'),
    ]
```

and finally to define `'profile_fk_widget'` url name in `urls.py`:

```
from django_jinja_knockout.urls import UrlPath
from club_app.views_ajax import ProfileFkWidgetGrid

# ... skipped ...
UrlPath(ProfileFkWidgetGrid)(
    name='profile_fk_widget',
    # kwargs={'permission_required': 'club_app.change_profile'}
),
UrlPath(UserFkWidgetGrid)(
    name='user_fk_widget',
    # kwargs={'permission_required': 'auth.change_user'}
),
```

Typical usage of ModelForm such as `MemberForm` is to perform CRUD actions in views or in grids datatables with Django model instances. In such case do not forget to inject url name of `'profile_fk_widget'` to client-side for AJAX requests to work automatically.

In your class-based view that handlers `MemberForm` inject `'profile_fk_widget'` url name (route) at client-side (see *Installation* and *context_processors.py* for details about injecting url names to client-side via `client_routes`):

```
from django.views.generic.edit import CreateView
from .forms import MemberForm
```

(continues on next page)

(continued from previous page)

```
class MemberCreate(CreateView):
    # Next line is required for ``ProfileFkWidgetGrid`` to be callable from client-
↪side:
    client_routes = {
        'profile_fk_widget'
    }
    form = MemberForm
```

- See club_app.views_ajax and urls.py code for fully featured example.

The same widget can be used with `MemberForm` bound to datatables via *'create_form' action* / *'edit_form' action*, or with any custom action, both AJAX requests and traditional requests.

When widget is used in many different views, it could be more convenient to register client-side route (url name) globally in project's `settings.py`. Such client-side routes will be injected into every generated page via base_bottom_scripts.htm:

```
# Second element of each tuple defines whether the client-side route should be␣
↪available to anonymous users.
DJK_CLIENT_ROUTES = {
    ('equipment_grid', True),
    ('profile_fk_widget', False),
    ('user_fk_widget', False),
    ('user_change', False),
}
```

3.7.1 ForeignKeyGridWidget implementation notes

Both *ForeignKeyGridWidget* and *MultipleKeyGridWidget* inherit from base class widgets.BaseGridWidget.

Client-side part of `ForeignKeyGridWidget` is implemented in FkGridWidget class. It uses the instance of GridDialog class to browse and to select foreign key field value(s) for the related `ModelForm`.

views.KoGridView class `postprocess_row()` method is used to generate `str()` representation for each Django model instance associated to each grid row, in case there is neither Django model *get_str_fields()* method nor grid class custom method `get_row_str_fields()` defined:

```
def postprocess_row(self, row, obj):
    str_fields = self.get_row_str_fields(obj, row)
    if str_fields is None or self.__class__.force_str_desc:
        row['__str'] = str(obj)
    if str_fields is not None:
        row['__str_fields'] = str_fields
    return row
```

views.KoGridRelationView overrides `postprocess_row` method so the row also includes `__perm` key, which is then stored to GridRow instance `.perm` attribute to determine additional grid row permissions, such as `canDelete` (foreign key deletion per row) in FkGridWidget inputRow.

In case `str_fields` representation of row is too verbose for `ForeignKeyGridWidget` display value, one may define grid class property `force_str_desc` = `True` to always use `str()` representation instead:

```
class ProfileFkWidgetGrid(KoGridView):

    model = Profile
    form = ProfileForm
```

(continues on next page)

(continued from previous page)

```
    enable_deletion = True
    force_str_desc = True
    grid_fields = ['first_name', 'last_name']
    allowed_sort_orders = '__all__'
    search_fields = [
        ('first_name', 'icontains'),
        ('last_name', 'icontains'),
    ]
```

`str_fields` still will be used to automatically format or localize row field values in grid, when available.

Client-side of widget is dependent either on cbv_grid.htm or cbv_grid_inline.htm Jinja2 templates, which generate grid underscore.js client-side templates via *ko_grid_body() macro* call.

One has to use these templates in his project, or to develop separate templates with these client-side scripts included. It's possible to include Jinja2 templates from Django templates using custom template tags library:

```
{% load %jinja %}
{% jinja 'ko_grid_body.htm' with _render_=1 %}
```

- See club_grid.html for example of grid templates generation in Django Template Language.

ko_grid_body() macro contains `ko_fk_grid_widget` / `ko_fk_grid_widget_row` / `ko_fk_grid_widget_controls` templates, used by widgets.BaseGridWidget and it's ancestors. To customize visual layout of widget / selected foreign key rows, one may use `attrs` and `grid_options` widget kwargs with Javascript class and / or template names like this:

```
self.fields['tag_set'] = forms.ModelMultipleChoiceField(
    widget=MultipleKeyGridWidget(
        attrs={
            # Override widget Javascript class name (optional)
            'classPath': 'TagWidget',
            # Override widget templates
            'data-template-options': {
                'templates': {
                    'ko_fk_grid_widget_row': 'ko_tag_widget_row',
                    'ko_fk_grid_widget_controls': 'ko_tag_widget_controls',
                }
            }
        },
        grid_options={
            # Override foreign key grid Javascript class name (optional)
            'classPath': 'TagGrid',
            # Set url 'club_id` kwargs initial value (when needed)
            'pageRoute': 'tag_fk_widget',
            'pageRouteKwargs': {
                'club_id': 0 if self.instance.club is None else self.instance.club.pk,
            },
        },
    )
}
```

Then to define actual templates in html code:

```
<script type="text/template" id="ko_tag_widget_row">
    <div class="container col-auto" data-bind="css: inputRow.css, click: inputRow.
↪onClick">
```

(continues on next page)

(continued from previous page)

```
        <div class="row well well-sm default-margin">
            <div class="col-sm-6">
                <div class="badge preformatted" data-bind="text: inputRow.desc().name
↪"></div>
            </div>
            <div class="col-sm-1">
                <a class="close" data-bind="visible: inputRow.canDelete, click:␣
↪inputRow.remove">×</a>
            </div>
        </div>
    </div>
</script>

<script type="text/template" id="ko_tag_widget_controls">
<div data-top="true">
    <button class="btn btn-info default-margin" data-bind="click: onFkButtonClick,␣
↪clickBubble: false">{{ _('Change') }}</button>
    <button class="btn btn-success default-margin">Custom action</button>
</div>
</script>
```

`inputRow .desc()` attribute is generated by GridRow.getDescParts() method, which uses *get_str_fields()*, when available and falls down to `str()` representation otherwise.

The value of `attrs` argument of `ForeignKeyGridWidget` defines widget DOM attrs which may optionally include the following special DOM attributes:

- `classPath` - override widget component Javascript class name (FkGridWidget)
- `data-template-id` - override widget html template ('ko_fk_grid_widget')
- `data-template-options` - specify *Underscore.js templates* template options, eg. to override widget's nested template names (like `ko_tag_widget_row` in the example above)

The value of `grid_options` argument of `ForeignKeyGridWidget()` is very much similar to the definition of `'fkGridOptions'` value for *Foreign key filter*. Both embed datatables (grids) inside BootstrapDialog, with the following differences:

- `'fk' filter` limits grid queryset.
- `ForeignKeyGridWidget` is used to set foreign key value, to be later submitted via `ModelForm`, including traditional HTML request / response and AJAX request. See *Client-side viewmodels and AJAX response routing* / *Built-in views*.

Widget's Python code generates client-side component similar to *ko_grid() macro*, but it uses FkGridWidget component class instead of Grid component class.

3.8 MultipleKeyGridWidget

Since version 1.0.0, editing of many to many relations is supported via `MultipleKeyGridWidget`. For example, each `Club` has many to many relation to `Tag`:

```
class Tag(models.Model):
    name = models.CharField(max_length=32, verbose_name='Tag')
    clubs = models.ManyToManyField(Club, blank=True, verbose_name='Clubs')
```

(continues on next page)

(continued from previous page)

```
    def get_str_fields(self):
        str_fields = OrderedDict([
            ('name, self.name)
        ])
        return str_fields
```

Then to edit multiple relations of `Club` in `ClubForm`:

```
from django_jinja_knockout.forms import RendererModelForm

class TagForm(RendererModelForm):

    class Meta:
        model = Tag
        fields = ['name']

class ClubForm(RendererModelForm):

    def add_tag_set_fk_grid(self):
        # https://kite.com/python/docs/django.forms.ModelMultipleChoiceField
        # value = field.widget.value_from_datadict(self.data, self.files, self.add_
↪prefix(name))
        self.fields['tag_set'] = forms.ModelMultipleChoiceField(
            widget=MultipleKeyGridWidget(grid_options={
                'pageRoute': 'tag_fk_widget',
            }),
            queryset=Tag.objects.all(),
            required=False,
        )
        if self.instance.pk is not None:
            self.fields['tag_set'].initial = self.instance.tag_set.all()

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.add_tag_set_fk_grid()
```

Now define the widget grid for `Tag` model:

```
class TagFkWidgetGrid(KoGridRelationView):

    form = TagForm
    grid_fields = ['name']
```

it's url (*UrlPath*):

```
from django_jinja_knockout.urls import UrlPath

UrlPath(TagFkWidgetGrid)(
    name='tag_fk_widget',
    # kwargs={'permission_required': 'club_app.change_tag'},
),
```

and it's client-route:

```
class ClubEditMixin(ClubNavsMixin):
```

(continues on next page)

(continued from previous page)

```
    client_routes = {
        'manufacturer_fk_widget',
        'profile_fk_widget',
        'tag_fk_widget',
    }
    template_name = 'club_edit.htm'
    form_with_inline_formsets = ClubFormWithInlineFormsets
```

The definition is very similar to *ForeignKeyGridWidget* with the exception that multiple keys are allowed to add / edit / delete.

`KoGridRelationView` selects which relation rows are allowed to remove for the current user via overriding it's can_delete_relation() method.

- See club_app.models for complete definitions of models.
- See club_app.forms for complete definitions of forms.
- See club_app.views_ajax and urls.py code for fully featured example.

3.9 Grids interaction

Multiple grid components can be rendered at single html page via multiple Jinja2 *ko_grid() macro* calls. Each grid will have it's own sorting, filters, pagination and actions. Sometimes it's desirable to update one grid state depending on the results of action performed in another grid.

3.9.1 Server-side interaction between grids

Let's see `ClubEquipmentGrid` which allows to add instances of `Equipment` model to particular instance of `Club` model. It's quite similar to inline formsets but saves the relation during each related form / model adding. Such way it provides a kind of "quick save" feature and also allows to edit large set of related `Equipment` models with pagination and optional search / filtering - not having to load the whole queryset as inline formset.

`ClubEquipmentGrid` has two custom actions `'add_equipment'` / `'save_equipment'` implemented as:

```
class ClubEquipmentGrid(EditableClubGrid):

    client_routes = {
        'equipment_grid',
        'club_grid_simple',
        'manufacturer_fk_widget',
    }
    template_name = 'club_equipment.htm'
    form = ClubForm
    form_with_inline_formsets = None

    def get_actions(self):
        actions = super().get_actions()
        actions['built_in']['save_equipment'] = {}
        actions['iconui']['add_equipment'] = {
            'localName': _('Add club equipment'),
            'css': 'iconui-wrench',
        }
        return actions
```

(continues on next page)

(continued from previous page)

```
    # Creates AJAX ClubEquipmentForm bound to particular Club instance.
    def action_add_equipment(self):
        club = self.get_object_for_action()
        if club is None:
            return vm_list({
                'view': 'alert_error',
                'title': 'Error',
                'message': 'Unknown instance of Club'
            })
        equipment_form = ClubEquipmentForm(initial={'club': club.pk})
        # Generate equipment_form viewmodel
        vms = self.vm_form(
            equipment_form, form_action='save_equipment'
        )
        return vms

    # Validates and saves the Equipment model instance via bound ClubEquipmentForm.
    def action_save_equipment(self):
        form = ClubEquipmentForm(self.request.POST)
        if not form.is_valid():
            form_vms = vm_list()
            self.add_form_viewmodels(form, form_vms)
            return form_vms
        equipment = form.save()
        club = equipment.club
        club.last_update = timezone.now()
        club.save()
        # Instantiate related EquipmentGrid to use it's .postprocess_qs() method
        # to update it's row via grid viewmodel 'prepend_rows' key value.
        equipment_grid = EquipmentGrid()
        equipment_grid.request = self.request
        equipment_grid.init_class(equipment_grid)
        return vm_list({
            'view': self.__class__.viewmodel_name,
            'update_rows': self.postprocess_qs([club]),
            # return grid rows for client-side EquipmentGrid component .updatePage(),
            'equipment_grid_view': {
                'prepend_rows': equipment_grid.postprocess_qs([equipment])
            }
        })
```

- `'add_equipment' action` creates `ClubEquipmentForm` bound to particular `Club` foreign key instance.
- `'save_equipment' action` validates and saves `Equipment` model instance related to target row `Club` instance via bound `ClubEquipmentForm`.
- Because both `ClubEquipmentGrid` and `EquipmentGrid` are sharing single club_equipment.htm template, `ClubEquipmentGrid` defines all client-side url names (client routes), required for `EquipmentGrid` and it's foreign key filters to work as `client_routes` class property.

Note that grid viewmodel returned by `ClubEquipmentGrid` class `action_save_equipment()` method has `'equipment_grid_view'` subproperty which will be used to update rows of `EquipmentGrid` at client-side (see below). Two lists of rows are returned to be updated via *Grid.updatePage() method*:

- vm_list `'update_rows': self.postprocess_qs([club])` list of rows to be updated for `ClubEquipmentGrid`

- vm_list `'equipment_grid_view': {'prepend_rows': ...}` list of rows to be updated for `EquipmentGrid`

`EquipmentGrid` is much simpler because it does not define custom actions. It's just used to display both already existing and newly added values of particular `Club` related `Equipment` model instances:

```
class EquipmentGrid(KoGridView):
    model = Equipment
    grid_fields = [
        'club',
        'manufacturer',
        'inventory_name',
        'category',
    ]
    search_fields = [
        ('inventory_name', 'icontains')
    ]
    allowed_filter_fields = OrderedDict([
        ('club', {
            'pageRoute': 'club_grid_simple',
            # Optional setting for BootstrapDialog:
            'dialogOptions': {'size': 'size-wide'},
        }),
        ('manufacturer', {
            'pageRoute': 'manufacturer_fk_widget'
        }),
        ('category', None)
    ])
    grid_options = {
        'searchPlaceholder': 'Search inventory name',
    }
```

To see full-size example:

- Jinja2 template club_equipment.htm for these grids datatables
- club_app.views_ajax class-based views
- club-grid.js client-side part, which implements `ClubGrid` and `ClubGridActions` classes, shared between club_equipment.htm and club_grid_with_action_logging.htm templates.

3.9.2 Client-side interaction between grids

At client-side `ClubEquipmentGrid` is instantiated as `ClubGrid` via club-grid.js script. It implements custom `callback_*` methods via `ClubGridActions` class for the following actions:

- `callback_save_form` / `callback_save_inline` / `callback_delete_confirmed` updates related grid(s) via looking up for their `.component()` then performing `.perform('update')` on component class, when available.
- `callback_add_equipment` displays BootstrapDialog with AJAX-submittable `ClubEquipmentForm` via `.callback_create_form(viewModel)` built-in method call.
- `callback_save_equipment` updates row(s) of both `ClubGrid`, which is bound to `ClubEquipmentGrid` class-based view in club_equipment.htm and `EquipmentGrid`, which does not define custom client-side grid class.

Here is the code of grid AJAX callbacks:

```
(function(ClubGridActions) {

    ClubGridActions.updateDependentGrid = function(selector) {
        // Get instance of dependent grid.
        var grid = $(selector).component();
        if (grid !== null) {
            // Update dependent grid.
            grid.actions.perform('update');
        }
    };

    // Used in club_app.views_ajax.ClubGridWithActionLogging.
    ClubGridActions.callback_save_inline = function(viewModel) {
        this._super._call('callback_save_form', viewModel);
        this.updateDependentGrid('#action_grid');
        this.updateDependentGrid('#equipment_grid');
    };

    // Used in club_app.views_ajax.ClubEquipmentGrid.
    ClubGridActions.callback_save_form = function(viewModel) {
        this._super._call('callback_save_form', viewModel);
        this.updateDependentGrid('#action_grid');
        this.updateDependentGrid('#equipment_grid');
    };

    ClubGridActions.callback_delete_confirmed = function(viewModel) {
        this._super._call('callback_delete_confirmed', viewModel);
        this.updateDependentGrid('#action_grid');
        this.updateDependentGrid('#equipment_grid');
    };

    ClubGridActions.callback_add_equipment = function(viewModel) {
        this.callback_create_form(viewModel);
    };

    ClubGridActions.callback_save_equipment = function(viewModel) {
        var equipmentGridView = viewModel.equipment_grid_view;
        delete viewModel.equipment_grid_view;
        this.grid.updatePage(viewModel);
        // Get client-side class of EquipmentGrid component by id (instance of Grid␣
↪or derived class).
        var equipmentGrid = $('#equipment_grid').component();
        if (equipmentGrid !== null) {
            // Update rows of MemberGrid component (instance of Grid or derived␣
↪class).
            equipmentGrid.updatePage(equipmentGridView);
        }
    };

})(ClubGridActions.prototype);
```

- `callback_save_equipment` uses jQuery selector `$('#equipment_grid')` to find root DOM element for `EquipmentGrid` component.
- Because there is no custom client-side grid class for `EquipmentGrid` defined in club_equipment.htm Jinja2 template, it uses built-in Grid instance from grid.js which is retrieved with `.component()` call on `$('#equipment_grid')` jQuery selector.

- When grid class instance is available in local `equipmentGrid` variable, it's rows are updated by calling `equipmentGrid` instance `.updatePage(equipmentGridView)` method.
- Full code of client-side part is available in club-grid.js script.
- See also `dom_attrs` argument of *ko_grid() macro* for explanation how grid component DOM id is set.

3.10 Custom action types

It is possible to define new grid action types. However to display these at client-side one has to use custom templates, which is explained in *Modifying visual layout of grid* section.

Let's define new action type `'button_bottom'`, which will be displayed as grid action buttons below the grid rows, not above as standard `'button'` action type actions.

First step is to override `KoGridView` class `get_actions()` method to return new grid action type with action definition(s):

```
class Model1Grid(KoGridView):

    model = Model1

    # ... skipped ...

    def get_actions(self):
        actions = super().get_actions()
        # Custom type UI actions.
        actions['button_bottom'] = OrderedDict([
            ('approve_user', {
                'localName': _('Approve user'),
                'css': {
                    'button': 'btn-warning',
                    'iconui': 'iconui-user'
                },
            })
        ])
        return actions

    def get_custom_meta(self):
        return {
            'user_name': str(self.user),
        }

    def get_ko_meta(self):
        meta = super().get_ko_meta()
        meta.update(self.get_custom_meta())
        return meta

    def action_approve_user(self):
        role = self.request.POST.get('role_str')
        self.user = self.request.POST.get('user_id')
        self.user.set_role(role)
        # Implement custom logic in user model:
        user.approve()
        return vm_list({
            'view': self.__class__.viewmodel_name,
            'title': format_html('User was approved {0}', self.user.username),
```

(continues on next page)

(continued from previous page)

```
            'message': 'Congratulations, you were approved!',
            'meta': self.get_custom_meta(),
            'update_rows': [self.user]
        })
```

Note that we override `get_ko_meta()` method to automatically set the value of `meta.user_name` observable in `Model1Grid` Knockout.js bindings via Grid class built-in `.loadMetaCallback()` method.

Second step is to override `uiActionTypes` property of client-side Grid class to add `'button_bottom'` to the list of interactive action types.

One also has to implement client-side handling methods for newly defined `approve_user` action. The following example assumes that the action will be performed as AJAX query / response with `Model1Grid` class `action_approve_user()` method:

```
import { inherit } from '../../djk/js/dash.js';
import { Grid } from '../../djk/js/grid.js';

Model1Grid = function(options) {
    inherit(Grid.prototype, this);
    this.init(options);
};

(function(Model1Grid) {

    Model1Grid.init = function(options) {
        this._super._call('init', options);
        this.meta.user_name = ko.observable();
    };

    Model1Grid.uiActionTypes = ['button', 'button_footer', 'pagination', 'click',
↪'iconui', 'button_bottom'];

    Model1Grid.iocGridActions = function(options) {
        return new Model1GridActions(options);
    };

    Model1Grid.getRoleFilterChoice = function() {
        return this.getKoFilter('role').getActiveChoices()[0];
    };

})(Model1Grid.prototype);
```

Mandatory (for server-side AJAX actions only) `callback_approve_user` method and optional `queryargs_approve_user` method are implemented to perform custom action (see *Action AJAX response handler*, *Action queryargs*):

```
import { inherit } from '../../djk/js/dash.js';
import { Dialog } from '../../djk/js/dialog.js';
import { GridActions } from '../../djk/js/grid/actions.js';

Model1GridActions = function(options) {
    inherit(GridActions.prototype, this);
    this.init(options);
};

(function(Model1GridActions) {
```

(continues on next page)

(continued from previous page)

```
    Model1GridActions.queryargs_approve_user = function(options) {
        var roleFilterChoice = this.grid.getRoleFilterChoice();
        options['role_str'] = roleFilterChoice.value;
        return options;
    };

    Model1GridActions.callback_approve_user = function(viewModel) {
        // Update grid meta (visual appearance).
        this.grid.updateMeta(viewModel.meta);
        // Update grid rows.
        this.grid.updatePage(viewModel);
        // Display dialog with server-side title / message generated in Model1Grid.
↪action_approve_user.
        var dialog = new Dialog(viewModel);
        dialog.alert();
    };

})(Model1GridActions.prototype);
```

And the final step is to generate client-side component in Jinja2 template with overridden ko_grid_body template

```
{% extends 'base_min.htm' %}
{% from 'bs_navs.htm' import bs_navs with context %}
{% from 'ko_grid.htm' import ko_grid with context %}
{% from 'ko_grid_body.htm' import ko_grid_body with context %}


{% block main %}

{{ bs_navs(main_navs) }}

{{ ko_grid(
    grid_options={
        'pageRoute': 'model1_grid',
    },
    dom_attrs={
        'id': 'model1_grid',
        'data-template-options': {
            'templates': {
                'ko_grid_body': 'model1_ko_grid_body',
            }
        },
    }
) }}

{% do page_context.set_custom_scripts(
    'sample/js/model1-grid.js',
) -%}

{% endblock main %}

{% block bottom_scripts %}
    {{ ko_grid_body() }}

    <script type="text/template" id="model1_ko_grid_body">
        <card-primary data-bind="using: $root, as: 'grid'">
```

(continues on next page)

(continued from previous page)

```
            <card-header data-bind="text: meta.verboseNamePlural"></card-header>
            <card-body>
                <!-- ko if: meta.hasSearch() || gridFilters().length > 0 -->
                <div data-template-id="model1_ko_grid_nav"></div>
                <!-- /ko -->
                <div data-template-id="model1_ko_grid_table"></div>
                <!-- ko foreach: {data: actionTypes['button_bottom'], as: 'koAction'}␣
↪-->
                    <button class="btn" data-bind="css: getKoCss('button'), click:␣
↪function() { doAction({}); }">
                        <span class="iconui" data-bind="css: getKoCss('iconui')"></
↪span>
                        <span data-bind="text: koAction.localName"></span>
                    </button>
                <!-- /ko -->
            </card-body>
        </card-primary>
    </script>

{% endblock bottom_scripts %}
```

Knockout.js `foreach: {data: actionTypes['button_bottom'], as: 'koAction'}` binding is very similar to standard `'button'` type actions binding, defined in ko_grid_body.htm, with the exception that the buttons are placed below the grid table, not the above.

There is built-in *Action type 'button_footer'* available, which displays grid action buttons below the grid rows, so this code is not requited in recent versions of the framework, but still it provides an useful example to someone who wants to implement custom action types and their templates.

There is built-in *Action type 'pagination'* which allows to add iconui buttons with grid actions attached directly to datatable pagination list.

3.10.1 Grids API

See the source code and the sample project code.

- FilterDialog - *todo*
- GridDialog - *ForeignKeyGridWidget implementation notes*
- **ModelFormDialog** - ***Action AJAX response handler***, ***'create_form' action***, ***'edit_form' action***, *'create_inline' action*, *'edit_inline' action*, *Implementing custom grid row actions*
- ActionsMenuDialog - *Action type 'click'*
- ActionTemplateDialog - *Client-side actions*
- Grid init options - *ko_grid() macro*
- ioc methods - *todo*
- methods to get actions / filters / rows / row field values - *todo*

3.11 djk_ui

django-jinja-knockout supports Bootstrap 3 / Bootstrap 4 / Bootstrap 5 via the `djk_ui` Django application module.

djk_ui module is installed from djk-bootstrap3 / djk-bootstrap4 / djk-bootstrap5 packages, respectively. This means that djk-bootstrap3 / djk-bootstrap4 / djk-bootstrap5 packages are mutually exclusive and only one has to be installed in the project virtualenv at the same time.

Unfortunately pip does not support requirements.txt files with de-installation directives. Thus one has to use pip with separate requirements-bs3.txt / requirements-bs4.txt / requirements-bs5.txt files, to install the current stable version, or to copy and then run 3bs.sh / 4bs.sh / 5bs.sh shell scripts, to switch between current master (possibly unstable) versions of djk_ui. Usually most of projects does not require changing Bootstrap version on the fly, so that's not much of problem.

3.11.1 conf.py

Contains the default layout_classes values, for example for Bootstrap 5 (version 2.1.0):

```
LAYOUT_CLASSES = {
    '': {
        'label': 'col-md-3',
        'field': 'col-md-7',
    },
    'display': {
        'label': 'w-25 table-info',
        'field': 'w-100 table-light',
    },
}
```

where '' key specifies the layout classes of editable model forms, 'display' key specifies the layout classes of the display-only model forms.

These default values can be overridden via the project *settings* module LAYOUT_CLASSES variable. See also *opts argument* for more info how layout classes are applied to form / formset renderers; see *Changing bootstrap grid layout* how layout classes are used in form / formset macros.

3.11.2 tpl.py

Contains nested list / dict formatters, specific to used Bootstrap version. See *tpl.py* for more info.

3.11.3 Customization

This module implements both server-side (Python) and client-side (Javascript) parts of the code that differs between Bootstrap 3 / Bootstrap 4 / Bootstrap 5. While it's possible to implement much larger djk_ui wrappers for more generic non-Bootstrap based UIs, currently I do not have enough of time / resources for that.

3.12 Forms

forms module is refactored into forms package with base module / renderers module / and validators module.

3.12.1 Renderers

See renderers module for source code.

django-jinja-knockout uses Renderer derived classes to display Django model forms and inline formsets. Recent versions of Django utilize renderers with templates to display form field widgets. There are some packages that use renderers with templates to generate the whole forms. In addition to that, django-jinja-knockout uses renderers to generate the formsets with the related forms, which follows Django DRY approach. It's possible to override the displayed HTML partially or completely.

The base Renderer class is located in *tpl* module and is not tied to any field / form / formset. It may be used in any template context.

The instance of Renderer holds the following related data:

- `self.obj` - a Python object that should be displayed (converted to HTML / string);
- `.get_template_name()` - method to obtain a DTL or Jinja2 template name, which could be hardcoded via `.template` class attribute, or can be dynamically generated depending on the current value of `self.obj`.

  See FieldRenderer for example of dynamic template name generation based on `self.obj` value, where `self.obj` is an instance of model form field;
- `self.context` - a Python dict with template context that will be passed to the rendered template;

The instance of Renderer encapsulates the object and the template with it's context to convert `self.obj` to string / HTML representation. Basically, it's an extensible string formatter. See `.__str__()` and `.__call__()` methods of Renderer class for the implementation.

The built-in renderers support both ordinary input fields POST forms (non-AJAX and AJAX versions) and the display forms (read-only forms): *Displaying read-only "forms"*.

Multiple objects should not re-use the same Renderer derived instance: instead the renderers of nested objects are nested into each other (object composition). Here is the complete list of nested hierarchies of the built-in renderers:

FieldRenderer

Renders model form field.

Default FieldRenderer attached to each form field will apply bootstrap HTML to the form fields / form field labels. It supports both input fields POST forms (non-AJAX and AJAX versions) and the display forms (read-only forms): *Displaying read-only "forms"*. Templates are chosen dynamically depending on the field type.

The instance of FieldRenderer is attached to each visible form field. By default the form fields are rendered this way:

```
{% for field in form.visible_fields() -%}
    {{ field.renderer() }}
{% endfor -%}
```

It's possible to render "raw" field with:

```
{{ field }}
```

and to render the list of selected fields with:

```
{{ render_fields(form, 'field1', 'fieldN') }}
```

FormBodyRenderer

Renders only the form body, no `<form>` tag, similar to how Django converts form to string.

- FormBodyRenderer
  - [FieldRenderer (1), ... FieldRenderer (n)]

In Jinja2 template call render_form() template context function:

```
{{ render_form(request, 'body', form, opts) }}
```

StandaloneFormRenderer

Standalone form renderer includes the whole form with the body (fields, field labels), `<form>` tag, wrapped into bootstrap card tags. It's a complete HTML form with separate visual look which could be directly submitted to view.

Renders the instance of model form:

- StandaloneFormRenderer
  - FormBodyRenderer
    - [FieldRenderer (1), ... FieldRenderer (n)]

In Jinja2 template call *bs_form()* macro or call render_form() template context function:

```
{{ render_form(request, 'standalone', form, {
    'action': action,
    'opts': opts,
    'method': method,
}) }}
```

Rendering FormWithInlineFormsets

To render *FormWithInlineFormsets class*, in Jinja2 template call *bs_inline_formsets()* macro, which calls the following hierarchy of renderers:

- RelatedFormRenderer
  - FormBodyRenderer
    - [FieldRenderer (1), ... FieldRenderer (n)]
- FormsetRenderer (1)
  - InlineFormRenderer (1)
    - FormBodyRenderer (1)
      - [FieldRenderer (1), ... FieldRenderer (n)]
  - InlineFormRenderer (n)
    - FormBodyRenderer (n)
      - [FieldRenderer (1), ... FieldRenderer (n)]
- FormsetRenderer (n)
  - InlineFormRenderer (n)
    - FormBodyRenderer (n)
      - [FieldRenderer (1), ... FieldRenderer (n)]

Note that is the composition hierarchy of instances, not a class inheritance hierarchy.

Single formset is rendered with the following call:

```
{{ formset.renderer() }}
```

opts argument

`opts` dict argument optionally passed to *bs_form()* / *bs_inline_formsets()* macros / render_form() template context function / form renderers support the following keys:

- `class` - CSS class of bootstrap panel form wrapper;
- `is_ajax` - bool, whether the form should be submitted via AJAX - by default is *False*; see *AJAX forms processing* for more info;
- layout_classes - change default Bootstrap grid layout width for field labels / field inputs. See *Changing bootstrap grid layout* for more details;
- `submit_text` - text of form submit button; if not defined, no button will be displayed;
- `title` - text of bootstrap panel title form wrapper; if not defined, no title will be displayed;

Some attributes are used only by some renderers:

- `inline_title` - the title of inline form, which could be different from `title` of related / standalone form;
- `table_classes` - CSS classes of form table wrapper for *Displaying read-only "forms"*;

Rendering customization

The most simplest way to customize form is to override / extend one of the default model form templates via overriding RendererModelForm template attributes, for example to change inline form wrapper:

```
class EquipmentForm(RendererModelForm):

    inline_template = 'inline_equipment_form.htm'
```

To change field templates one should override RendererModelForm `Meta` class `field_templates` dict attribute:

```
class ClubMemberDisplayForm(WidgetInstancesMixin, RendererModelForm,␣
↪metaclass=DisplayModelMetaclass):

    inline_template = 'inline_form_chevron.htm'
    body_template = 'form_body_club_group_member_display.htm'

    class Meta:

        model = ClubMember

        fields = [
            'role',
            'profile',
            'note',
        ]
        field_templates = {
            'role': 'field_items.htm',
            'note': 'field_items.htm',
        }
```

To change formset template, one should set the value of formset class attribute like this:

```
ClubEquipmentFormSet = ko_inlineformset_factory(
    Club, Equipment, form=EquipmentForm, extra=0, min_num=1, max_num=5, can_
↪delete=True
)
ClubEquipmentFormSet.template = 'club_equipment_formset.htm'
```

It's also possible to use raw built-in rendering, which does not uses Jinja2 templates. To achieve that, set the template name value to empty string ''. In such case renderer instance .render_raw() method will be called to convert `self.obj` with it's current context to the string. For more complex cases one may override .render_raw() method via inherited renderer class.

To use custom renderer classes with model forms, one may override BootstrapModelForm `Meta` class default renderer attributes with the extended classes:

```
class MyModelForm(BootstrapModelForm):

    class Meta(BootstrapModelForm.Meta):
        render_body_cls = MyFormBodyRenderer
        # render_inline_cls = MyInlineFormRenderer
        # render_related_cls = MyRelatedFormRenderer
        render_standalone_cls = MyStandaloneFormRenderer
```

but in most of the cases overriding the template names is enough.

See renderer template samples in `djk-sample` project for the example of simple customization of default templates.

3.12.2 Forms base module

See base module source code.

RendererModelForm

While it's possible to use renderers with ordinary Django `ModelForm` class, the recommended way is to derive model form class from RendererModelForm class:

```
from django_jinja_knockout.forms import RendererModelForm

class ProfileForm(RendererModelForm):

    class Meta:
        model = Profile
        exclude = ('age',)
        fields = '__all__'
```

By default, in case there are no custom templates / no custom renderers specified, render_form() will use the default renderers from BootstrapModelForm `Meta` class, which would stylize model form with Bootstrap attributes.

RendererModelForm class `.has_saved_instance()` method used to check whether current Django ModelForm has the bound and saved instance.

3.12.3 AJAX forms processing

`django_jinja_knockout` includes `bs_form()` and `bs_inline_formsets()` Jinja2 macros, which generate Bootstrap styled Django ModelForms. Usual form generation syntax is:

```
{% extends 'base_min.htm' %}
{% from 'bs_form.htm' import bs_form with context %}

{% block main %}

{{ bs_form(form=form, action=url('my_url_name'), opts={
    'class': 'form_css_class',
    'title': page_context.get_view_title(),
    'submit_text': 'My button'
}) }}

{% endblock main %}
```

If your class-based views extends one of the following view classes:

```
django_jinja_knockout.views.FormWithInlineFormsetsMixin
django_jinja_knockout.views.InlineCreateView
# Next view is suitable both for updating ModelForms with inline formsets
# as well for displaying read-only forms with forms.DisplayModelMetaclass.
django_jinja_knockout.views.InlineCrudView
```

then, in order to have the form processed as AJAX form, add `'is_ajax': True` key to `bs_form()` / `bs_inline_formsets()` Jinja2 macro call:

```
{{ bs_form(form=form, action=url('my_url_name'), opts={
    'class': 'form_css_class',
    'is_ajax': True,
    'title': page_context.get_view_title(),
    'submit_text': 'My button'
}) }}
```

AJAX response and success URL redirection will be automatically generated. Form errors will be displayed in case there is any. Such form will behave very similarly to usual non-AJAX submitted form with the following advantages:

1. AJAX response saves HTTP traffic.
2. Instead of just redirecting to `success_url`, one may perform custom actions, including displaying Bootstrap-Dialog alerts and confirmations.
3. ajaxform.js includes Bootstrap progress bar when the form has file inputs. So when large files are uploaded, the progress indicator will be updated, instead of just waiting until the request completes.

At client-side both successful submission of form and form errors are handled by lists of client-side viewmodels. See *Client-side viewmodels and AJAX response routing* for more detail.

At server-side (Django), the following code of `FormWithInlineFormsetsMixin` is used to process AJAX-submitted form errors:

```
def get_form_error_viewmodel(self, form):
    for bound_field in form:
        return {
            'view': 'form_error',
            'class': 'danger',
            'id': bound_field.auto_id,
            'messages': list((escape(message) for message in form.errors['__all__']))
        }
    return None
```

(continues on next page)

(continued from previous page)

```
def get_field_error_viewmodel(self, bound_field):
    return {
        'view': 'form_error',
        'id': bound_field.auto_id,
        'messages': list((escape(message) for message in bound_field.errors))
    }
```

and the following code returns success viewmodels:

```
def get_success_viewmodels(self):
    # @note: Do not just remove 'redirect_to', otherwise deleted forms will not be␣
↪refreshed
    # after successful submission. Use as callback for view: 'alert' or make your own␣
↪view.
    return vm_list({
        'view': 'redirect_to',
        'url': self.get_success_url()
    })
```

In instance of `FormWithInlineFormsetsMixin`, `self.forms_vms` and `self.fields_vms` are the instances of `vm_list()` defined in `viewmodels.py`. These instances accumulate viewmodels (each one is a simple Python dict with `'view'` key) during ModelForm / inline formsets validation.

Actual AJAX ModelForm response success / error viewmodels can be overridden in child class, if needed.

These examples shows how to generate dynamic lists of client-side viewmodels at server-side. `viewmodels.py` defines methods to alter viewmodels in already existing `vm_list()` instances.

3.12.4 Displaying read-only "forms"

If form instance was instantiated from `ModelForm` class with `DisplayModelMetaclass` metaclass:

```
from django_jinja_knockout.forms import BootstrapModelForm, DisplayModelMetaclass

from my_app.models import Profile

class ProfileDisplayForm(BootstrapModelForm, metaclass=DisplayModelMetaclass):

    class Meta:
        model = Profile
        exclude = ('age',)
        fields = '__all__'
```

one may use empty string as submit url value of `action=''` argument, to display ModelForm instance as read-only Bootstrap table:

```
{% extends 'base_min.htm' %}
{% from 'bs_inline_formsets.htm' import bs_inline_formsets with context %}

{{
    bs_inline_formsets(related_form=form, formsets=[], action='', opts={
        'class': 'project',
        'title': form.get_title(),
    })
}}
```

Such "forms" do not contain `<input>` elements and thus cannot be submitted. Additionally you may inherit from `UnchangeableModelMixin`:

```
from django_jinja_knockout.forms import UnchangeableModelMixin
```

to make sure bound model instances cannot be updated via custom script submission (eg. Greasemonkey).

In case related many to one inline formset ModelForms should be included into read-only "form", define their `ModelForm` class with `metaclass=DisplayModelMetaclass` and specify that class as `form` kwarg of `inlineformset_factory()`:

```
from django_jinja_knockout.forms import BootstrapModelForm, DisplayModelMetaclass,␣
↪set_empty_template

from my_app.models import Profile

class MemberDisplayForm(BootstrapModelForm, metaclass=DisplayModelMetaclass):

    class Meta:
        model = Profile
        fields = '__all__'

MemberDisplayFormSet = inlineformset_factory(
    Project, Member,
    form=MemberDisplayForm, extra=0, min_num=1, max_num=2, can_delete=False
)
MemberDisplayFormSet.set_knockout_template = set_empty_template
```

`DisplayText` read-only field widget automatically supports lists as values of `models.ManyToManyField` fields, rendering these as Bootstrap "list groups".

Custom rendering of DisplayText form widgets

Sometimes read-only "form" fields contain complex values, such as dates, files and foreign keys. In such case default rendering of `DisplayText` form widgets, set up by `DisplayModelMetaclass`, can be customized via manual ModelForm field definition with `get_text_method` argument callback:

```
from django_jinja_knockout.forms import BootstrapModelForm, DisplayModelMetaclass,␣
↪WidgetInstancesMixin
from django_jinja_knockout.widgets import DisplayText
from django.utils.html import format_html
from django.forms.utils import flatatt

from my_app.models import ProjectMember

class ProjectMemberDisplayForm(WidgetInstancesMixin, BootstrapModelForm,␣
↪metaclass=DisplayModelMetaclass):

    class Meta:

        def get_profile(self, value):
            return format_html(
                '<a {}>{}</a>',
                flatatt({'href': reverse('profile_detail', profile_id=self.instance.
↪pk)}),
                self.instance.user
            )
```

(continues on next page)

(continued from previous page)

```
        model = ProjectMember
        fields = '__all__'
        widgets = {
            'profile': DisplayText(get_text_method=get_profile)
        }
```

`WidgetInstancesMixin` is used to make model `self.instance` available in `DisplayText` widget callbacks. It enables access to all fields of current model instance in `get_text_method` callback, in addition to `value` of the current field.

Note that `get_text_method` argument will be re-bound from form `Meta` class to instance of `DisplayText` widget.

`DisplayText` field widget supports selective skipping of table rows rendering via setting widget instance property `skip_output` to `True`:

```
# ... skipped imports ...
class ProjectMemberDisplayForm(WidgetInstancesMixin, BootstrapModelForm,␣
↪metaclass=DisplayModelMetaclass):

    class Meta:

        def get_profile(self, value):
            if self.instance.is_active:
                return format_html(
                    '<a {}>{}</a>',
                    flatatt({'href': reverse('profile_detail', profile_id=self.
↪instance.pk)}),
                    self.instance.user
                )
            else:
                # Do not display inactive user profile link in table form.
                self.skip_output = True
                return None

        model = ProjectMember
        fields = '__all__'
        widgets = {
            'profile': DisplayText(get_text_method=get_profile)
        }
```

Customizing string representation of scalar values is performed via `scalar_display` argument of `DisplayText` widget:

```
class ProjectMemberDisplayForm(WidgetInstancesMixin, BootstrapModelForm,␣
↪metaclass=DisplayModelMetaclass):

    class Meta:
        widgets = {
            'state': DisplayText(
                scalar_display={True: 'Allow', False: 'Deny', None: 'Unknown', 1: 'One
↪'}
            ),
        }
```

Optional `scalar_display` and `get_text_method` arguments of `DisplayText` widget can be used together.

Optional `get_text_fn` argument of `DisplayText` widget allows to use non-bound functions to generate text of the widget. It can be used with `scalar_display` argument, but not with `get_text_method` argument.

3.12.5 Dynamically adding new related formset forms

`bs_inline_formsets()` macro with conjunction of `django_jinja_knockout.forms.set_knockout_template()` monkey patching formset method and client-side `formsets.js` script supports dynamic adding / removing of new formset forms (so-called empty_form) via Knockout.js custom binding to Formset.

Instead of simply storing `formset.empty_form` value then cloning it via jQuery and performing `String.prototype.replace()` to set form index:

```
$('#form_set').append($('#empty_form').html().replace(/__prefix__/g, form_idx));
```

Knockout.js bindings offer the following advantages:

- Imagine unintentional or malicious content where `__prefix__` substring appears in empty_form representation outside form inputs DOM attribute values. `set_knockout_template()` of `django_jinja_knockout.forms` ensures that only `__prefix__` substring in specified DOM attributes is bound to be changed by using `lxml` to convert empty_form naive string prefixes to proper Knockout.js `data-bind` attribute values.
- Knockout.js automatically re-calculates form prefix index when one of newly dynamically added formset forms are deleted before submitting.
- Knockout.js translated version of empty_form template is stored in `bs_inline_formsets()` Jinja2 macro as the value of hidden textarea, which allows to dynamically add field widgets with inline scripts.

AFAIK it's the only solution to add client-side empty_form dynamically without possible XSS attacks. If there are another such solutions, please let me know.

To be able to add / remove new empty forms use monkey patching of inline formset class like this in `forms.py`:

```
from django.forms.models import BaseInlineFormSet, inlineformset_factory
from django_jinja_knockout.forms import BootstrapModelForm, set_knockout_template,␣
↪FormWithInlineFormsets

from my_app.models import Project

class ProjectForm(BootstrapModelForm):

    class Meta:
        model = Project
        fields = '__all__'

    def clean(self):
        super().clean()
        # Put form field validation here.

class ProjectMemberFormSetDef(BaseInlineFormSet):

    def clean(self):
        super().clean()
        for form in self.forms:
            if form.cleaned_data.get('DELETE'):
                continue
            # Put inline formset form field validation here.
```

(continues on next page)

(continued from previous page)

```
            # Warning! May be None, thus dict.get() is used.
            my_field_value = form.cleaned_data.get('my_field')

ProjectMemberFormSet = inlineformset_factory(
    Project, ProjectMember,
    form=ProjectForm, formset=ProjectMemberFormSetDef, extra=0, min_num=1, max_num=2,␣
↪can_delete=True
)
ProjectMemberFormSet.set_knockout_template = set_knockout_template

class ProjectFormWithInlineFormsets(FormWithInlineFormsets):

    FormClass = ProjectForm
    FormsetClasses = [ProjectMemberFormSet]
```

In your class-based views.py:

```
from django_jinja_knockout.views import InlineCreateView, InlineDetailView

class ProjectCreate(InlineCreateView):

    form_with_inline_formsets = ProjectFormWithInlineFormsets
    template_name = 'project_form.htm'

class ProjectUpdate(InlineDetailView):

    form_with_inline_formsets = ProjectFormWithInlineFormsets
    template_name = 'project_form.htm'
```

FormWithInlineFormsets class

There is extra step of deriving `ProjectFormWithInlineFormsets` from `forms.FormWithInlineFormsets` class because that class serves as intermediate layer between form with inline formsets and Django views. Besides class-based views (`InlineCreateView`, `InlineDetailView`, `FormWithInlineFormsetsMixin`) it can be used in traditional functional views as well:

```
ff = ProjectFormWithInlineFormsets(request, create=True)
if request.method == 'POST':
    if ff.save() is None:
        # Show form errors.
        return render(request, 'project_template.htm', {
            'form': ff.form,
            'formsets': ff.formsets
        })
    else:
        # Form with inline formsets was saved successfully.
        return redirect('project_save_success')
else:
    # Display initial form for project instance (project update form).
    project = Project.objects.filter(user=user).first()
    ff.get(project)
    return render(request, 'project_template.htm', {
        'form': ff.form,
        'formsets': ff.formsets,
    })
```

3.13 HTTP response

This module extends built-in Django response by providing immediate exception response and AJAX response:

- MockRequestFactory - allows to perform fully qualified name reverse url resolve in console management scripts:

```
from django_jinja_knockout.apps import DjkAppConfig
from django_jinja_knockout.tpl import reverseq

request = DjkAppConfig.get_context_middleware().get_request()
reverseq('member_detail', kwargs={'member_id': 1}, request=request, query={'users
↪': [1,2,3]})
```

- JsonResponse - HTTP response which automatically converts dicts / lists / mapping / sequence to JSON. It also has json_response shortcut function with the defaults.
- ImmediateHttpResponse - exception which allows to interrupt view code flow. It renders Django response provided as an exception's `__init__()` method agrument.
- ImmediateJsonResponse - exception which allows to interrupt view code flow. It renders JSON response provided as an exception's `__init__()` method argument:

```
from django.utils.html import format_html

# ... skipped ...

if not User.objects.filter(pk=user_id).exists():
    raise ImmediateJsonResponse({
        'view': 'alert_error',
        'message': format_html('Unknown used id: {}', user_id),
    })
```

- error_response() / exception_response() - wrappers around `django.http.HttpResponseBadRequest` to allow JSON viewmodel response in AJAX requests in case of error / exception occured.

3.14 Jinja2 macros

3.14.1 ModelForms

bs_form()

bs_form() macro allows to generate html representation of ModelForm:

```
{% extends 'base_min.htm' %}
{% from 'bs_form.htm' import bs_form with context %}

{% block main %}

{{ bs_form(form=form, action=tpl.url('my_url_name'), opts={
    'class': 'form_css_class',
    'title': page_context.get_view_title(),
    'submit_text': 'My button'
}) }}

{% endblock main %}
```

Since the introduction of form renderers in version 0.8.0, bs_form() macro become a simple compatibility wrapper, while the actual HTML code of form is generated with the following render_form() call:

```
{{ render_form(request, 'standalone', form, {
    'action': action,
    'opts': opts,
    'method': method,
}) }}
```

Note that the bs_form() macro generates html `<form>` tag and wraps the whole form into Bootstrap card with the heading / body. If you want to generate form body only (usual Django approach), call render_form() template context function instead:

```
{{ render_form(request, 'body', form) }}
```

To read more about render_form() template context function and the built-in form / inline formsets renderers, see *Forms*.

3.14.2 Inline formsets

bs_inline_formsets()

bs_inline_formsets() is a macro that supports html rendering of one or zero Django ModelForm with one or multiple related inline formsets. It also supports two types of rendering layouts:

- `<div>` layout for real changeable submittable forms.
- `<table>` layout primarily used to display read-only "forms" (see *Forms*).

Also it has support for inserting custom content between individual forms of formsets.

Example of form with inline formsets rendering:

```
{{
bs_inline_formsets(related_form=form, formsets=formsets, action=tpl.url('add_project',
↪ project_id=project.pk), opts={
    'class': 'project',
    'is_ajax': True,
    'title': page_context.get_view_title(),
    'submit_text': 'Add new project'
}) }}
```

- In this case form with formsets will be submitted and processed via AJAX POST request / response due to `is_ajax = True` argument.
- bs_inline_formsets() also supports `{% call() bs_inline_formsets() %}` syntax for complex formatting of formsets which is unused in this simplified example.

Changing bootstrap grid layout

One may use the custom layout_classes value as the key of the following macros `opts` dict argument:

- bs_form(form, action, opts, method='post')
- bs_inline_formsets(related_form, formsets, action, opts)

to alter default Bootstrap inline form grid width, for example:

```
{{
bs_inline_formsets(related_form=form, formsets=formsets, action=tpl.url('project_
↪candidate_add', project_id=project.pk), opts={
    'class': 'project',
    'is_ajax': True,
    'title': page_context.get_view_title(),
    'submit_text': 'Add candidate',
    'layout_classes': {
        '': {
            'label': 'col-md-4', 'field': 'col-md-6',
        }
    }
}) }}
```

Default value of Bootstrap inline grid layout classes is defined in `djk_ui` app *conf.py* module `LAYOUT_CLASSES` variable.

Inserting custom content

Calling bs_inline_formsets() macro with `kwargs` argument allows to insert custom blocks of html at the following points of form with related formsets rendering:

Begin of formset. `formset_begin` will hold the instance of formset, allowing to distinguish one formset from another one:

```
{{ caller({'formset_begin': formset, 'html': html}) }}
```

Begin of formset form:

```
{{ caller({'form_begin': form, 'html': html}) }}
```

End of formset form:

```
{{ caller({'form_end': form, 'html': html}) }}
```

End of formset. `formset_end` will hold the instance of formset, allowing to distinguish one formset from another one (see the example below):

```
{{ caller({'formset_end': formset, 'html': html}) }}
```

Adding custom buttons, for example many AJAX POST buttons each with different `data-url` or `data-route` html5 attributes. That allows to submit the same AJAX form to different Django views:

```
{{ caller({'buttons': True}) }}
```

The following example inserts custom submit button, which is supported when the `'is_ajax': True` parameter is specified:

```
{% extends 'base_min.htm' %}
{% from 'bs_inline_formsets.htm' import bs_inline_formsets with context %}

{% call(kwargs)
bs_inline_formsets(related_form=form, formsets=formsets, action=tpl.url('project_
↪update', project_id=project.pk), opts={
    'class': 'project',
    'is_ajax': True,
```

(continues on next page)

(continued from previous page)

```
    'title': page_context.get_view_title(),
    'submit_text': 'Update project'
}) %}

{% if 'buttons' in kwargs %}
    <button type="submit" data-url="{{ tpl.url('project_postpone', project_id=project.
↪pk) }}" class="btn btn-primary">
        Postpone project
    </button>
{% endif %}

{% endcall %}
```

Resulting html will have two form submit buttons:

- one is automatically generated with submit `tpl.url('project_update', ...)`
- another is manually inserted with submit `tpl.url('project_postpone', ...)`

Different views may be called from the same Django AJAX form with inline formsets, depending on which html button is pressed.

The following example will insert total project read-only "form" (see *Forms*) extra cost columns after the end of rendering related `projectmember_set` inline formset:

```
{% extends 'base_min.htm' %}
{% from 'bs_inline_formsets.htm' import bs_inline_formsets with context %}

{% call(kwargs)
bs_inline_formsets(related_form=form, formsets=formsets, action='', opts={
    'class': 'project',
    'title': form.instance,
    'submit_text': 'Review project'
}) %}

{% if 'formset_end' in kwargs and kwargs.formset_end.prefix == 'projectmember_set' %}
    {% set total_cost = form.project.get_total_cost() %}
    {% if total_cost > 0 %}
        <div class="default-padding">
            <table class="table">
                <colgroup>
                    <col class="{{ kwargs.html.layout_classes.label }}">
                    <col class="{{ kwargs.html.layout_classes.field }}">
                </colgroup>
                <tr>
                    <th class="success">Total cost</th>
                    <td class="info">{{ total_cost }}</td>
                </tr>
            </table>
        </div>
    {% endif %}
{% endif %}

{% endcall %}
```

Wrapping each form of formset with div with custom attributes (to process these in custom Javascript):

```
{% call(kwargs)
bs_inline_formsets(related_form=form, formsets=formsets, action=tpl.url('project_
↪update', project_id=project.pk), opts={
    'class': 'project',
    'is_ajax': True,
    'title': form.instance,
    'submit_text': 'Update project'
}) %}

{% if 'form_begin' in kwargs %}
<div id="revision-{{ kwargs.form_begin.instance.pk }}">
{% endif %}

{% if 'form_end' in kwargs %}
</div>
{% endif %}

{% endcall %}
```

Since version 0.8.0, the more flexible approach could be to override *Renderers* templates instead.

3.14.3 Bootstrap navigation

See .get_filter_kwargs() sample for working example of boostrap navigation macros.

bs_breadcrumbs()

bs_breadcrumbs macro generates bootstrap breadcrumbs of the current filter choices from the result of *ListSortingView* class .get_filter_kwargs() call:

```
{% for field in view.allowed_filter_fields -%}
    {{ bs_breadcrumbs(**view.get_filter_kwargs(field)) }}
{% endfor -%}
```

bs_choice_list()

bs_choice_list macro generates the flat list of the currently selected filter choices from the result of *ListSortingView* class .get_filter_kwargs() call:

```
{% for field in view.allowed_filter_fields -%}
    {{ bs_choice_list(**view.get_filter_kwargs(field)) }}
{% endfor -%}
```

bs_dropdown()

bs_dropdown macro generates bootstrap dropdown of the current filter choices from the result of *ListSortingView* class .get_filter_kwargs() call:

```
{% for field in view.allowed_filter_fields -%}
    {{ bs_dropdown(**view.get_filter_kwargs(field)) }}
{% endfor -%}
```

bs_filters()

Displays the list of *ListSortingView* filters which produce empty queryset:

```
{{ bs_filters(**view.get_no_match_kwargs()) }}
```

bs_list()

Displays current page of the supplied `ListView` / *ListSortingView* view instance `object_list` (queryset) with `page_obj` paginator links stylized for bootstrap. It has the optional call wrapper which supports three optional arguments to provide three optional caller sections: `has_filters`, `has_no_match`, `has_pagination`.

To override all of the caller sections:

```
{% call(has_filters, has_no_match, has_pagination) bs_list(view, object_list, is_
↪paginated, page_obj) -%}
    {% if has_filters -%}
        {# insert custom filters layout here #}
    {% elif has_no_match -%}
        {# insert custom filters layout here #}
    {% elif has_pagination -%}
        {# insert custom pagination layout here #}
    {% endif -%}
{% endcall -%}
```

To override just one `has_filters` caller section:

```
{% call(has_filters) bs_list(view, object_list, is_paginated, page_obj) -%}
    {% if has_filters -%}
        {# insert custom filters layout here #}
    {% endif -%}
{% endcall -%}
```

No override (no call) see cbv_list.htm for example:

```
{{ bs_list(view, object_list, is_paginated, page_obj) }}
```

For the example of customizing *ListSortingView* has_filters section / has_pagination section see the djk-sample club_list_with_component.htm Jinja2 template.

bs_navs()

This macro takes the result of prepare_bs_navs function or the result of *BsTabsMixin* template context `main_navs` variable to display automatically highlighted server-side bootstrap navigation tabs. Do not confuse to bs_tabs() macro, which is similar but switches between tabs at the client-side via TabPane Javascript class.

Since v0.9.0, *bs_navs()* macro is also argument-compatible to filter choices from the result of *ListSortingView* class .get_filter_kwargs() call (see also *bs_breadcrumbs()* / *bs_choice_list()* / *bs_dropdown()* / *bs_filters()* which are compatible):

```
{% for field in view.allowed_filter_fields -%}
    {{ bs_navs(**view.get_filter_kwargs(field)) }}
{% endfor -%}
```

bs_tabs()

bs_tabs() macro creates TabList Javascript component which manages client-side bootstrap tabs. Internally it uses TabPaneManager Javascript class. Let's explain some methods of TabPaneManager class:

- `.switchTo()` method enables automatic switching of bootstrap tab panes upon page load and via `window.location.hash` change. Hash change may occur programmatically from user script, or via clicking the anchor with matching hash name.
- `.highlight()` method provides permanent or temporary highlighting of displayed bootstrap tab, to indicate that it's contents was updated / changed. This is particularly useful when bs_tabs() is used together with AJAX dynamic components, such as datatables.
- `.loadTemplate()` method allows one-time filling of tab content from the specified `template_id` attribute of bs_tabs() macro. It allows to delay AJAX calls of the template components until the user actually clicked on the tab, instead of performing all AJAX calls even for the invisible tabs at once. Which is useful for the long lists of tabs with *ko_grid() macro* generated datatable component for example.

djk_sample demo project has bs_tabs() sample / TabPane sample which places grids into bootstrap tabs.

The first mandatory argument of bs_tabs() macro is the `tabs` list. Each element of the `tabs` list should be the dict that defines content of each tab. The following mandatory key-value pairs are required:

- `id` - the value of window.location.hash for current tab;
- `title` - title of current tab;

The following keys are mutually exclusive:

- `html` - html of tab pane. Use Jinja 2.8+ `{% set html %} {% endset %}` syntax to capture complex content, such as grid, ModelForm, inline formset and so on;
- `template_id` - one may specify underscore.js template id which will be expanded to tab pane when the user switches to that pane, instead of `html` which loads the content to the tab immediately. It's used to lazy load Javascript components, eg. datatables (grids). See *Underscore.js templates* and *Components* for more info.

Optional key-value pairs:

- `is_active` - set to `True` when current tab has to be selected by default;
- `tooltip` - optional tooltip for the tab link;

The second optional argument of bs_tabs() macro is `tabs_attrs` dict which defines tpl.json_flatatt() HTML attributes for the tabs wrapper tag, which is *ul.nav.nav-tabs* by default.

The third optional argument of bs_tabs() macro is `content_attrs` dict which defines tpl.json_flatatt() HTML attributes for the tabs content tag, which is *div.tab-content* by default.

3.15 Management commands

3.15.1 djk_seed

Implements optional `djk_seed` Django management command which may be used to seed initial data into managed models database tables after the migrations are complete. To enable model data seed after the migration, define `seed` method of the model like this:

```
class Specialization(models.Model):
    BUILTIN_SPECIALIZATIONS = (
        ('Administrator', False),
```

(continues on next page)

(continued from previous page)

```
        ('Manager', True),
        ('Contractor', True),
    )

    @classmethod
    def seed(cls, recreate=False):
        if recreate or cls.objects.count() == 0:
            # Setup default  list (only once).
            for name, is_anon in cls.BUILTIN_SPECIALIZATIONS:
                cls.objects.update_or_create(name=name, defaults={
                    'is_builtin': True,
                    'is_anon': is_anon
                })
```

then add app which has `Specialization` model into settings.DJK_APPS list. See installation for more info about `DJK_APPS` list.

After that run the console command:

```
./manage.py djk_seed
```

`djk_seed` management command has `--help` option which describes possible use cases. For example it may create models content types for the selected Django apps, not running any post-migration seed:

```
./manage.py djk_seed --create-content-types --skip-seeds
```

This is often an pre-requisite to have contenttypes framework running correctly.

3.16 middleware.py

3.16.1 Middleware installation

The built-in middleware is compatible both to the old type of middleware and to the new type of middleware.

To add the built-in middleware to the project `settings.py`, define `DJK_MIDDLEWARE` value, then add it to the `MIDDLEWARE` list:

```
DJK_MIDDLEWARE = 'django_jinja_knockout.middleware.ContextMiddleware'

MIDDLEWARE = [
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    'django.middleware.security.SecurityMiddleware',
    DJK_MIDDLEWARE,
]
```

The built-in middleware is applied only to Django apps which are registered in `settings.py` variable `DJK_APPS` list:

```
DJK_APPS = (
    'djk_sample',
    'club_app',
    'event_app',
)

INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.sites',
    'django_jinja',
    'django_jinja.contrib._humanize',
    'djk_ui',
    'django_jinja_knockout',
) + DJK_APPS
```

Such apps has to be both in `DJK_APPS` list and in `INSTALLED_APPS` list. See sample settings.py for the complete example.

Since v0.9.0 the dependency on `DJK_MIDDLEWARE` was sufficiently reduced. The code which does not call `DjkAppConfig` class .get_context_middleware() method and does not use middleware features described here (like permission checks), should work without `DJK_MIDDLEWARE` defined in settings.py at all. However beware that `RendererModelForm` and `ForeignKeyGridWidget` still require it, so it's not the recommended settings to run.

3.16.2 Extending built-in middleware

Middleware is extendable (inheritable), which allows to implement your own features by overloading it's methods. See the example of extending middleware.

`DjkAppConfig` class .get_context_middleware() method should be used to resolve the installed `ContextMiddleware` class instead of direct import. Such way the extended `ContextMiddleware` class specified in `settings.py` `DJK_MIDDLEWARE` will be used instead of the original version:

```
from django_jinja_knockout.apps import DjkAppConfig

ContextMiddleware = DjkAppConfig.get_context_middleware()
```

Direct import from `django_jinja_knockout.middleware` or from `my_project.middleware` is possible but is discouraged as wrong version of middleware may be used.

The instance of middleware provides the access to current HTTP request instance anywhere in form / formset / field widget code:

```
request = ContextMiddleware.get_request()
```

- Real HTTP request instance will be loaded when running as web server.
- Fake request will be created when running in console (for example in the management commands). Fake request HTTP GET / POST arguments can be initialized via `ContextMiddleware` class `.mock_request()` method, before calling `.get_request()`.

Still it's wise to restrict `.get_request()` usage to forms / formsets / widgets mostly, avoiding usage at the model / database / console management command level, although the mocking requests makes that possible.

3.16.3 Automatic timezone detection

Automatic timezone detection and activation from the browser, which should be faster than using maxmind geoip database. It's possible to get timezone name string from current browser http request to use in the application (for example to pass it to celery task):

```
ContextMiddleware.get_request_timezone()
```

3.16.4 Middleware security

The views that belong to modules defined in `DJK_APPS` are checked for permissions, specified in urls.py url() call `kwargs`.

`DJK_APPS` views are secured by the middleware with urls that deny access to anonymous / inactive users by default. Anonymous views require explicit permission defined as `url()` extra kwargs per each view in `urls.py`:

```
from my_app.views import signup
# ...
url(r'^signup/$', signup, name='signup', kwargs={'allow_anonymous': True})
```

Optional check for specific Django permission:

```
from my_app.views import check_project
# ...
url(r'^check-project/$', check_project, name='check_project', kwargs={
    'permission_required': 'my_app.project_can_add'
})
```

3.16.5 Request mock-up

It's possible to mock-up requests in console mode (management commands) to resolve reverse URLs fully qualified names:

```
from django_jinja_knockout.apps import DjkAppConfig
from django_jinja_knockout import tpl

request = DjkAppConfig.get_context_middleware().get_request()
# Will return fully-qualified URL for the specified route with query string appended:
tpl.reverseq('profile_detail', kwargs={'profile_id': 1}, request=request, query={
↪'users': [1,2,3]})
```

By default domain name is taken from current configured Django site. Otherwise either `settings.DOMAIN_NAME` or `settings.ALLOWED_HOSTS` should be set to autodetect current domain name.

3.16.6 Mini-router

Inherited middleware classes (see *DJK_MIDDLEWARE* settings) support built-in mini router, which could be used to implement CBV-like logic in the middleware class itself, either via request path string match or via the regexp match:

```
class ContextMiddleware(RouterMiddleware):

    routes_str = {
        '/-djk-js-error-/': 'log_js_error',
    }
    routes_re = [
        # (r'^/-djk-js-(?P<action>/?\w*)-/', 'log_js_error'),
    ]

    def log_js_error(self, **kwargs):
        from .log import send_admin_mail_delay
        vms = vm_list()
        # ... skipped ...
        return JsonResponse(vms)
```

3.16.7 Our request

Only views that belong to settings.py `DJK_APPS` (see *Middleware installation*) will be processed by djk middleware. One should override is_our_module() method in the extended middleware (see *Extending built-in middleware*) to implement custom middleware applying filter. The views, which have djk middleware applied, will have `request` object is_djk attribute set.

3.17 Models

This module contains the functions / classes to get information about Django models or to manipulate them.

- get_users_with_permission() - return the queryset of all users who have specified permission string, including all three possible sources of such users (user permissions, group permissions and superusers).
- Next functions allow to use parts of queryset functionality on single Django model object instances, supporting spanned relationships *without* the field lookups:
  - get_related_field_val() / get_related_field() get related field properties from the supplied model instance via `fieldname` argument string.
  - model_values() - get the dict of model fields name / value pairs like queryset `.values()` for the single model instance supplied.
- get_meta() / get_verbose_name() - get meta property of Django model field, including spanned relationships with the related (foreign) and reverse-related fields:

  ```
  get_verbose_name(profile, 'user__username')
  get_meta(profile, 'verbose_name_plural', 'user__username')
  ```

- file_exists() - checks whether Diango file field object exists in the related filesystem.
- get_object_description() - returns the possibly nested list / dict of django model fields. Uses Model.get_str_fields(), when available, otherwise fallback to Model.__str()__. See *get_str_fields model formatting / serialization* for more info.
- get_app_label_model() - parses dot-separated name of app_label / model (natural key) as returned by content-types framework. Can be used to parse request content type argument like this:

  ```
  app_label, model = get_app_label_model(self.request_get('model', ''))
  ```

- get_content_object() - returns content type / content object via contenttypes framework with any valid combination of arguments: `object_id`, `content_type_id`, `app_label`, `model`. For example, to get the Model instance from the request:

```
app_label, model = get_app_label_model(self.request_get('model', ''))
object_id = self.request_get('content_object_id', None)
content_type, content_object = get_content_object(object_id=object_id, app_
↪label=app_label, model=model)
```

3.18 query.py

3.18.1 FilteredRawQuerySet

`FilteredRawQuerySet` inherits Django `RawQuerySet` class whose instances are returned by Django model object manager `.raw()` calls.

It supports `.filter()` / `.exclude()` / `.order_by()` / `values()` / `values_list()` queryset methods and also SQL-level slicing which is much more efficient than Python slicing of `RawQuerySet`.

These methods are used by filtering / ordering code in ListSortingView and KoGridView class-based views.

See FilteredRawQuerySet sample in `djk-sample` project source code for a complete example of AJAX grid with raw query which has `LEFT JOIN` statement.

Since version 0.4.0 it supports args with Q objects via `relation_map` argument:

```
raw_qs = Profile.objects.raw(
    'SELECT club_app_profile.*, club_app_member.is_endorsed, '
    'auth_user.username AS user__username, '
    'CONCAT_WS(\' \', auth_user.last_name, auth_user.first_name) AS fio '
    'FROM club_app_profile '
    'LEFT JOIN club_app_member ON club_app_profile.user_id = club_app_member.profile_
↪id AND '
    'club_app_member.project_id=%s AND club_app_member.role=%s '
    'JOIN auth_user ON auth_user.id = club_app_profile.user_id ',
    params=[self.project.pk, 'watch'],
)
fqs = FilteredRawQuerySet.clone_raw_queryset(
    raw_qs=raw_qs, relation_map={'is_endorsed': 'member'}
)
```

3.18.2 ListQuerySet

`ListQuerySet` implements large part of Django queryset functionality for Python lists of Django model instances. Such lists are returned by Django queryset `.prefetch_related()` method.

This allows to have the same logic of processing queries with both `.prefetch_related()` applied results and without them. For example, imagine one have two querysets:

```
from django.db import models
from django.db.models import Prefetch
from django_jinja_knockout.query import ListQuerySet

def process_related():
```

(continues on next page)

(continued from previous page)

```
    qs1 = Project.objects.all()[:10]
    qs2 = Project.objects.all()[:10].prefetch_related(
        Prefetch(
            'projectmember_set',
            to_attr='projectmember_list'
        )
    )
    (obj.process_members() for obj in qs1)
    (obj.process_members() for obj in qs2)

class Project(models.Model):

    # ... skipped ...

    def process_members(self):
        # Detect Prefetch().
        if hasattr(self, 'projectmember_list'):
            qs = ListQuerySet(self.projectmember_list)
        else:
            qs = self.projectmember_set
        # ... Do .filter() / .order_by() / slice operation with qs
        qs_subset = qs.filter(is_approved=False)
        # ... Do some more operations with qs_subset or it's members.
        for obj in qs_subset:
            obj.approve()

class ProjectMember(models.Model):

    project = models.ForeignKey(Project, verbose_name='Project')
    is_approved = models.BooleanField(default=False, verbose_name='Approved member')
    # ... skipped ...

    def approve(self):
        self.is_approved = True
```

- Version 0.3.0 implemented `.filter()` / `.exclude()` / slicing / `.order_by()` / `.first()` / `.values()` / `.values_list()` methods. Many but not all of the field lookups are supported. Feel free to submit a pull request if you need more functionality.
- Version 0.8.0 implemented spanned relationships for `.order_by()` method.
- Version 0.8.1 implemented `|` and `+` operators for *ListQuerySet*. Note that the operation does not ensure the uniqueness of the resulting queryset. In case unique rows are required, call `.distinct('pk')` on the result.

3.19 tpl.py

Various formatting functions, primarily to be used in:

- `admin.ModelAdmin` classes readonly_fields
- *Jinja2 macros* and templates
- *Components*
- `DisplayText` *widgets.py*

Since version 0.8.0, the significant part of the module is implemented via *djk_ui* package.

3.19.1 Renderer

Since version 0.8.0, Renderer class is implemented which is internally used to render formsets / forms / fields of Django modelforms. See *Renderers* and *Forms base module* for more detail. It's usage is not limited to forms as it supports rendering of any object with the related context data and template with possible nesting of renderers.

3.19.2 Contenttypes framework helpers

- ContentTypeLinker - class to simplify generation of contenttypes framework object links:

```
{% set ctl = tpl.ContentTypeLinker(request.user, object, 'content_type', 'object_
↪id') %}
{% if ctl.url is not none %}
    <a href="{{ ctl.url }}" title="{{ str(ctl.obj_type) }}" target="_blank">
{% endif %}
    {{ ctl.desc }}
{% if ctl.url is not none %}
    </a>
{% endif %}
```

3.19.3 Manipulation with css classes

- `escape_css_selector()` - can be used with server-generated AJAX viewmodels or in Selenium tests.
- `add_css_classes()` - similar to client-side `jQuery` .addClass();
- `has_css_classes()` - similar to client-side `jQuery` .hasClass();
- `remove_css_classes()` - similar to client-side `jQuery` .removeClass();

Optimized for usage as argument of `Django` flatatt():

- `add_css_classes_to_dict()` - adds CSS classes to the end of the string
- `has_css_classes_in_dict()`
- `prepend_css_classes_to_dict()` - adds CSS classes to the begin of the string
- `remove_css_classes_from_dict()`

3.19.4 Objects rendering

- `Str` - string with may have extra attributes. It's used with `get_absolute_url()` Django model method. See get_absolute_url() documentation and get_absolute_url() sample:

```
class Manufacturer(models.Model):

    # ... skipped ...

    title = models.CharField(max_length=64, unique=True, verbose_name='Title')

    def get_absolute_url(self):
        url = Str(reverse('club_detail', kwargs={'club_id': self.pk}))
        url.text = str(self.title)
        return url
```

- ModelLinker - render Model links with descriptions which supports `get_absolute_url()` and *get_str_fields model formatting / serialization*. Since v2.1.0, optional `request_user` argument can be defined for custom `get_absolute_url()` Django model method which then may be used to hide part of `url.text` per user permissions. In such case ModelLinker / ContentTypeLinker instance should be initialized with `request_user` value, when available:

```
from django_jinja_knockout import tpl

obj = Model.objects.get(pk=1)
ctl = tpl.ContentTypeLinker(request.user, obj.content, 'content_type', 'object_id
↪')
content_type_str = ctl.get_str_obj_type()
# nested serialization of generic relation optional check of request.user␣
↪permissions
# see serializers.py
content_tree = ctl.get_nested_data()
# render nested serialization to str
content_tree_str = tpl.print_list(content_tree)
# get url / description text of content_object, when available
content_url = obj.content.content_object.get_absolute_url(request_user)
content_desc = ctl.desc if content_url is none else content_url.text
```

- PrintList class supports custom formatting of nested Python structures, including the mix of dicts and lists. There are some already setup function helpers which convert nested content to various (HTML) string representations, using PrintList class instances:
  - `print_list()` - print nested HTML list. Used to format HTML in JSON responses and in custom `DisplayText` widgets.
  - `print_brackets()` - print nested brackets list.
  - `print_table()` - print uniform 2D table (no colspan / rowspan yet).
  - `print_bs_labels()` - print HTML list as Bootstrap labels.
  - `reverseq()` - construct url with query parameters from url name. When request instance is supplied, absolute url will be returned.
- str_dict() - Django models could define *get_str_fields model formatting / serialization* method which maps model instance field values to their formatted string values, similar to `Model __str()__` method, but for each or to some selected separate fields. If these models have foreign keys pointing to another models which also have *get_str_fields model formatting / serialization* defined, str_dict() can be used to convert nested dict *get_str_fields model formatting / serialization* values to flat strings in `__str__()` method:

```
class Member(models.Model):

    # ... skipped ...

    def get_str_fields(self):
        parts = OrderedDict([
            ('profile', self.profile.get_str_fields()),
            ('club', self.club.get_str_fields()),
            ('last_visit', format_local_date(timezone.localtime(self.last_
↪visit))),
            ('plays', self.get_plays_display()),
            ('role', self.get_role_display()),
            ('is_endorsed', 'endorsed' if self.is_endorsed else 'unofficial')
        ])
        return parts
```

(continues on next page)

(continued from previous page)

```
    def __str__(self):
        # Will flatten 'profile' and 'club' str_fields dict keys values
        # and convert the whole str_fields dict values into str.
        str_fields = self.get_str_fields()
        return str_dict(str_fields)
```

Internally str_dict() uses lower level `flatten_dict()` function which is defined in the same module.

3.19.5 String manipulation

- `limitstr()` - cut string after specified length.
- `repeat_insert()` - separate string every nth character with specified separator characters.

3.19.6 String formatting

- json_flatatt() - similar to Django flatatt(), but converts dict / list / tuple / bool HTML attribute values to JSON string. Used in *Jinja2 macros*.
- format_html_attrs() - similar to Django format_html(), but converts dict / list / tuple / bool HTML attribute values to JSON string. Used to generate *Components*.
- `format_local_date()` - output localized `Date` / `DateTime`.
- `html_to_text()` - convert HTML fragment with anchor links into plain text with text links. It's used in *utils/mail.py* `SendmailQueue` to convert HTML body of email message to text-only body.
- `to_json()` - converts Python structures to JSON utf-8 string.

3.19.7 URL resolution

- `get_formatted_url()` converts url with supplied `url_name` from regex named parameters eg. `(?P<arg>\w+)` to `sprintf()` named formatters eg. `%(arg)s`. Such urls are injected into client-side as *Client-side routes* and then are resolved via the bundled sprintf.js library.
- `resolve_cbv()` takes `url_name` and it's `kwargs` and returns a function view or a class-based view for these arguments, when available:

```
tpl.resolve_cbv(url_name, view_kwargs)
```

Current request's `url_name` can be obtained from the `request` .resolver_match `.url_name`, or `.view_name` for namespaced urls.

3.20 urls.py

3.20.1 UrlPath

Since the version 1.0.0, auto-generation of Django re_path Class-Based views urls is supported via UrlPath class:

```
from django_jinja_knockout.urls import UrlPath
```

- `action` class-based view kwarg is used with *AJAX actions*.

Here are some examples of UrlPath calls and their equivalent via re_path.

3.20.2 Simplest example

view.action_kwarg = None

UrlPath:

```
UrlPath(ClubCreate)(name='club_create'),
```

is equivalent to re_path:

```
re_path(r'^club-create/$', ClubCreate.as_view(), name='club_create'),
```

ActionsView / KoGridView (view.action_kwarg = 'action')

UrlPath:

```
UrlPath(MyActionsView)(name='actions_view'),
```

is equivalent to re_path:

```
re_path(r'^actions-view(?P<action>/?\w*)/$', MyActionsView.as_view(), name='actions_
↪view'),
```

3.20.3 Extra kwargs for view_title and permissions checking

UrlPath:

```
UrlPath(EquipmentGrid)(
    name='equipment_grid',
    kwargs={
        'view_title': 'Grid with the available equipment',
        'permission_required': 'club_app.change_manufacturer'
    }
),
```

is equivalent to re_path:

```
re_path(r'^equipment-grid(?P<action>/?\w*)/$', EquipmentGrid.as_view(),
    name='equipment_grid', kwargs={
        'view_title': 'Grid with the available equipment',
        'permission_required': 'club_app.change_manufacturer'
    }),
```

3.20.4 Override url base path

UrlPath:

```
UrlPath(MyActionsView)(
    name='actions_view_url_name',
    base='my-actions-view',
    kwargs={'view_title': 'Sample ActionsView'}
),
```

is equivalent to re_path:

```
re_path(r'^my-actions-view(?P<action>/?\w*)/$', MyActionsView.as_view(),
    name='actions_view_url_name',
    kwargs={'view_title': 'Sample ActionsView'}),
```

3.20.5 Extra view named kwarg

view.action_kwarg = None

UrlPath:

```
UrlPath(ClubDetail)(
    name='club_detail',
    args=['club_id'],
    kwargs={'view_title': '{}'}
),
```

is equivalent to re_path:

```
re_path(r'^club-detail-(?P<club_id>\d+)/$', ClubDetail.as_view(),
    name='club_detail', kwargs={'view_title': '{}'}),
```

Note that `_id` suffix in `club_id` name of `UrlPath` call `args` causes `\d+` pattern to be generated instead of `\w*`.

ActionsView / KoGridView (view.action_kwarg = 'action')

UrlPath:

```
UrlPath(ClubMemberGrid)(
    name='club_member_grid',
    args=['club_id'],
    kwargs={'view_title': '"{}" members'}
),
```

is equivalent to re_path:

```
re_path(r'^club-member-grid-(?P<club_id>\w*)(?P<action>/?\w*)/$', ClubMemberGrid.as_
↪view(),
    name='club_member_grid',
    kwargs={'view_title': '"{}" members'}),
```

3.20.6 Change view named kwargs order

UrlPath:

```
UrlPath(MyActionsView)(
    name='actions_view',
    args=['action', 'club_id'],
    kwargs={
        'view_title': 'Club actions',
    }
),
```

is equivalent to re_path:

```
re_path(r'^actions-view(?P<action>/?\w*)-(?P<club_id>\d+)/$', MyActionsView.as_view(),
        name='actions_view', kwargs={
        'view_title': 'Club actions',
    }),
```

UrlPath:

```
UrlPath(ClubGrid)(
    name='club_grid',
    base='clubs',
    args=['club_id', 'type'],
    kwargs={'view_title': 'Club list',
            'permission_required': 'club_app.view_club'}
),
```

is equivalent to re_path:

```
re_path(r'^clubs-(?P<club_id>\d+)-(?P<type>\w*)(?P<action>/?\w*)/$', ClubGrid.as_
↪view(),
        name='club_grid',
        kwargs={'view_title': 'Club list',
        'permission_required': 'club_app.view_club'}),
```

- See urls.py from djk_sample for the complete working example of UrlPath usage.

3.21 utils/mail.py

class `SendmailQueue`, which instance is available globally as `EmailQueue`, allows to send multiple HTML emails with attachments. In case sendmail error is occured, error message can be converted to form non-field errors with `form` named argument of `.flush()` method (works with AJAX and non-AJAX forms):

```
from django_jinja_knockout.utils.mail import EmailQueue

EmailQueue.add(
    subject='Thank you for registration at our site!',
    html_body=body,
    to=destination_emails,
).flush(
    form=self.form
)
```

When there is no form submitted or it's undesirable to add form's non-field error, `request` named argument of `.flush()` may be supplied instead. It also works with both AJAX and non-AJAX views. AJAX views would use client-side *Client-side viewmodels and AJAX response routing*, displaying error messages in BootstrapDialog window. Non-AJAX views would use Django messaging framework to display sendmail errors:

```
from django_jinja_knockout.utils.mail import EmailQueue

EmailQueue.add(
    subject='Thank you for registration at our site!',
    html_body=body,
    to=destination_emails,
).flush(
    request=self.request
)
```

`SendmailQueue` class functionality could be extended by injecting ioc class. It allows to use database backend or non-SQL store to process emails in background, for example as Celery task. `SendmailQueue` class `.add()` and `.flush()` methods could be overridden in `self.ioc` and new methods can be added as well.

`uncaught_exception_email` function can be used to monkey patch Django exception `BaseHandler` to use `SendmailQueue` to send the uncaught exception reports to selected email addresses.

Here is the example of extending `EmailQueue` instance of `SendmailQueue` via custom ioc class (`EmailQueueIoc`) and monkey patching Django exception `BaseHandler`. This code should be placed in the project's `apps.py`:

```
class MyAppConfig(AppConfig):
    name = 'my_app'
    verbose_name = "Verbose name of my application"

    def ready(self):
        from django_jinja_knockout.utils.mail import EmailQueue
        # EmailQueueIoc should have custom .add() and / or .flush() methods␣
↪implemented.
        # Original .add() / .flush() methods may be called via ._add() / ._flush().
        from my_app.tasks import EmailQueueIoc

        EmailQueueIoc(EmailQueue)

        # Save uncaught exception handler.
        BaseHandler.original_handle_uncaught_exception = BaseHandler.handle_uncaught_
↪exception
        # Override uncaught exception handler.
        BaseHandler.handle_uncaught_exception = uncaught_exception_email
        BaseHandler.developers_emails = ['user@host.org']
        BaseHandler.uncaught_exception_subject = 'Django exception stack trace for my␣
↪project'
```

`my_app.tasks.py`:

```
class EmailQueueIoc:

    def __init__(self, email_queue):
        self.queue = email_queue
        self.instances = []
        # Maximum count of messages to send in one batch.
        self.batch_limit = 10
        self.max_total_errors = 3
        email_queue.set_ioc(self)

    def add(self, **kwargs):
        # Insert your code here.
        # Call original _add():
```

(continues on next page)

(continued from previous page)

```
        return self.queue._add(**kwargs)

    def flush(self, **kwargs):
        # Insert your code here.
        # Call original _flush():
        return self.queue._flush(**kwargs)

    def celery_task():
        # Insert your code here.

@app.task
def email_send_batch():
    EmailQueue.celery_task()
```

3.22 utils/sdv.py

Contains helper functions internally used by django-jinja-knockout. Some of these might be useful in Django project modules.

3.22.1 Class / model helpers

- `get_object_members()`
- `get_class_that_defined_method()`
- `extend_instance()` - allows to dynamically add mixin class to class instance. Can be used to dynamically add different *BsTabsMixin* ancestors to create context-aware navbar menus.
- `FuncArgs` - class which instance may hold args / kwargs which then may be applied to the specified method.
- `get_str_type()` - get string of type for the specified object.
- get_choice_str() - Similar to Django model built-in magic method get_FOO_display() but does not require to have an instance of particular Django model object.

For example:

```
class Member(models.Model):

    # ... skipped ...
    role = models.IntegerField(choices=ROLES, default=ROLE_MEMBER, verbose_name=
↪'Member role')

from .models import Member
from django_jinja_knockout.models import get_choice_str

# ... skipped ...
role_str = sdv.get_choice_str(Member.ROLES, role_val)
```

3.22.2 Debug logging

`dbg()` - dumps `value` into text log file *'sdv_out.py3'* under `name` label. To setup log file path specify the `LOGPATH` value in Django project `settings.py` like that:

```
import os
from django_jinja_knockout.utils import sdv

# create log file inside active virtualenv path
sdv.LOGPATH = [os.environ['VIRTUAL_ENV'], 'djk-sample', 'logs']
```

Then one may use it to log variables in Python code:

```
from django_jinja_knockout.utils import sdv

class Project(models.Model):

  # ... skipped ...

  def save(self, *args, **kwargs):
      sdv.dbg('self.pk', self.pk)
      # ... skipped ...
```

When Project.save() method will be executed, *'sdv_out.py3'* log file will contain lines like this:

```
# /home/user/work/djk_sample/djk-sample/club-app/models.py::save()::251
# self.pk
9
```

Where `9` was the value of `self.pk`.

3.22.3 Iteration

- `reverse_enumerate()`
- `iter_enumerate()` - enumerates both dicts, lists and tuples of lists (dict-like structures with repeated keys) in unified way.
- `yield_ordered()` - ordered enumeration of dicts (Python 3.6+) / OrderedDict / lists.

3.22.4 Nested data structures access

- `get_nested()` / `set_nested()` / `nested_values()` for nested data with mixed lists / dicts.
- nested_update() recursive update of Python dict. Used in *Datatables* extended classes to update `super().get_actions()` action dict.

3.22.5 String helpers

- `str_to_numeric` - convert string to numeric value, when possible.

3.23 Client-side viewmodels and AJAX response routing

3.23.1 Client-side viewmodels

`django_jinja_knockout` implements AJAX response routing with client-side viewmodels.

Viewmodels are defined as an array of simple objects in Javascript:

```
var viewmodels = [
    {
        'view': 'prepend',
        'selector': '#infobar',
        'html': '<div class="alert alert-info">Welcome to our site!</div>'
    },
    {
        'view': 'confirm',
        'title': 'Please enter <i>your</i> personal data.',
        'message': 'After the registration our manager will contact <b>you</b> to␣
↪validate your personal data.',
        'callback': [{
            'view': 'redirect_to',
            'url': '/homepage/'
        }],
        'cb_cancel': [{
            'view': 'redirect_to',
            'url': '/logout/'
        }]
    }
];
```

and as the special list (vm_list) of ordinary dicts in Python:

```
from django_jinja_knockout.viewmodels import vm_list

viewmodels = vm_list(
    {
        'view': 'prepend',
        'selector': '#infobar',
        'html': '<div class="alert alert-info">Welcome to our site!</div>'
    },
    {
        'view': 'confirm',
        'title': 'Please enter <i>your</i> personal data.',
        'message': 'After the registration our manager will contact <b>you</b> to␣
↪validate your personal data.',
        'callback': vm_list({
            'view': 'redirect_to',
            'url': '/homepage/'
        }),
        'cb_cancel': vm_list({
            'view': 'redirect_to',
            'url': '/logout/'
        })
    }
)
```

When executed, each viewmodel object (dict) from the `viewmodels` variable defined above, will be used as the function argument of their particular handler:

- `'view': 'prepend'`: executes `jQuery.prepend(viewmodel.html)` function for specified selector `#infobar`;
- `'view': 'confirm'`: shows `BootstrapDialog` confirmation window with specified `title` and `message`;
  - `'callback'`: when user hits `Ok` button of `BootstrapDialog`, nested `callback` list of client-side viewmodels will be executed, which defines just one command: `redirect_to` with the specified url

/homepage/;

- 'cb_cancel: when user cancels confirmation dialog, redirect to /logout/ url will be performed.

Now, how to execute these viewmodels we defined actually? At Javascript side it's a simple call:

```
import { vmRouter } from '../../djk/js/ioc.js';

vmRouter.respond(viewmodels);
```

While single viewmodel may be executed via the following call:

```
import { vmRouter } from '../../djk/js/ioc.js';

vmRouter.show({
    'view': 'form_error',
    'id': $formFiles[i].id,
    'messages': [message]
});
```

However, it does not provide much advantage over performing jQuery.prepend() and instantiating BootstrapDialog() manually. Then why is all of that?

First reason: one rarely should execute viewmodels from client-side directly. It's not the key point of their introduction. They are most useful as foundation of interaction between server-side Django and client-side Javascript via AJAX requests where the AJAX response is the list of viewmodels generated at server-side, and in few other special cases, such as sessions and document.onload viewmodels injecting.

Second reason: It is possible to setup multiple viewmodel handlers and then to remove these. One handler also could call another handler. Think of event subscription: these are very similar, however not only plain functions are supported, but also functions bound to particular instance (methods) and classpath strings to instantiate new Javascript classes:

```
import { vmRouter } from '../../djk/js/ioc.js';

// viewmodel bind context with method
var handler = {
    fn: MyClass.prototype.myMethod,
    context: myClassInstance
};
// Subscribe to bound method:
vmRouter.addHandler('my_view', handler)
// Subscribe to bound method:
    .add('my_view', MyClass.prototype.myMethod2, myClassInstance)
// Subscribe to unbound function:
    .add('my_view', myFunc)
// Subscribe to instantiate a new class via classpath specified:
    .addHandler('my_view', 'MyClass');
// ...
// Will execute all four handlers attached above with passed viewmodel argument:
vmRouter.exec('my_view', {'a': 1, 'b': 2});
// ...
// Unsubscribe handlers. The order is arbitrary.
vmRouter.removeHandler('my_view', {fn: MyClass.prototype.myMethod2, context:␣
↪myClassInstance})
    .removeHandler('my_view', myFunc)
    .removeHandler('my_view', handler)
    .removeHandler('my_view', 'MyClass');
```

Javascript bind context

The bind context is used when the viewmodel response is processed. It is used by `add()` / `addHandler()` viewmodel router methods and as well as *AJAX actions* callback.

The following types of context arguments of are available:

- unbound function: subscribe viewmodel to that function;
- plain object with optional `fn` and `context` arguments: to subscribe to bound method;
- string: Javascript class name to instantiate;

See ViewModelRouter.applyHandler() for the implementation details.

Viewmodel data format

Key `'view'` of each Javascript object / Python dict in the list specifies the value of `viewmodel name`, that is bound to particular Javascript `viewmodel handler`. The viewmodel itself is used as the Javascript object argument of each particular `viewmodel handler` with the corresponding keys and their values. The following built-in viewmodel names currently are available in ioc.js:

```
[
    'redirect_to',
    'post',
    'alert',
    'alert_error',
    'confirm',
    'trigger',
    'append',
    'prepend',
    'after',
    'before',
    'remove',
    'text',
    'html',
    'replaceWith',
    'replace_data_url'
]
```

If your AJAX code just needs to perform one of these standard actions, such as display alert / confirm window, trigger an event, redirect to some url or to perform series of jQuery DOM manipulation, then you may just use the list of viewmodels that map to these already pre-defined handlers.

Automatic AJAX POST is available with `post` viewmodel and even an AJAX callback is not required for POST because each `post` viewmodel AJAX response will be interpreted (routed) as the list of viewmodels - making chaining / nesting of HTTP POSTs easily possible.

There are class-based *AJAX actions* available, which allow to bind multiple methods of the Javascript class instance to single viewmodel handler: to perform multiple actions bound to the one viewmodel name.

Defining custom viewmodel handlers

One may add custom viewmodel handlers via Javascript plugins to define new actions. See tooltips.js for the additional bundled viewmodel names and their viewmodel handlers:

```
'tooltip_error', 'popover_error', 'form_error'
```

which are primarily used to display errors for AJAX submitted forms via viewmodels AJAX response.

The following methods allows to attach one or multiple handlers to one viewmodel name:

```
import { vmRouter } from '../../djk/js/ioc.js';

vmRouter.add('my_view', function(viewModel, vmRouter) {
    // execute viewmodel here...
})
    .add('my_view2', {fn: MyClass.prototype.method, context: MyClassInstance})
    .add('my_view3', 'MyClass');
// or
vmRouter.add({
    'my_view': function(viewModel, vmRouter) {
        // execute viewmodel here...
    },
    'my_view2': {fn: MyClass.prototype.method, context: MyClassInstance},
    'my_view3': 'MyClass'
});
```

The following syntax allows to reset previous handlers with the names specified (if any):

```
import { vmRouter } from '../../djk/js/ioc.js';

vmRouter.removeAll('my_view', 'my_view2', 'my_view3')
    .add({
        'my_view': function(viewModel, vmRouter) {
            // execute viewmodel here...
        },
        'my_view2': {fn: MyClass.prototype.method, context: MyClassInstance},
        'my_view3': 'MyClass'
    });
```

When `function` handler is called, it's `viewModel` argument receives the actual instance of `viewmodel`. Second optional argument `vmRouter` points to the instance of vmRouter that was used to process current `viewmodel`. This instance of vmRouter could be used to call another viewmodel handler inside the current handler, or to add / remove handlers via calling vmRouter instance methods:

```
import { vmRouter } from '../../djk/js/ioc.js';

vmRouter.add('my_view1', function(viewModel, currentVmRouter) {
    // dynamically add 'my_view2' viewmodel handler when 'my_view1' handler is␣
↪executed:
    currentVmRouter.add('my_view2', function(viewModelNested, vmRouter) {
        // will receive argument viewModelNested == {'a': 1, 'b': 2}}
        // execute viewModelNested here...
    });
    // ... skipped ...
    // nested execution of 'my_view2' viewmodel from 'my_view1' handler:
    currentVmRouter.exec('my_view2', {'a': 1, 'b': 2});
});
```

New properties might be added to viewmodel for further access, like `.instance` property which holds an instance of `FieldPopover` in the following code:

```
import { vmRouter } from '../../djk/js/ioc.js';
import { FieldPopover } from '../../djk/js/tooltips.js';
```

(continues on next page)

(continued from previous page)

```
vmRouter.add('tooltip_error', function(viewModel) {
    // Adding .instance property at the client-side to server-side generated␣
↪viewModel:
    viewModel.instance = new FieldPopover(viewModel);
});
```

Every already executed viewmodel is stored in `.executedViewModels` property of vmRouter instance, which may be processed later. An example of such processing is destroyFormErrors static method, which clears form input Bootstrap tooltips previously set by `'tooltip_error'` viewmodel handler then removes these viewmodels from `.executedViewModels` list via ViewModelRouter.filterExecuted() method:

```
AjaxForm.destroyFormErrors = function() {
    var form = this.$form.get(0);
    vmRouter.filterExecuted(
        function(viewModel) {
            if (viewModel.view === 'form_error' && typeof viewModel.instance !==
↪'undefined') {
                viewModel.instance.destroy(form);
                return false;
            }
            return true;
        }
    );
};
```

It is possible to chain viewmodel handlers, implementing a code-reuse and a pseudo-inheritance of viewmodels:

```
import { vmRouter } from '../../djk/js/ioc.js';
import { FieldPopover } from '../../djk/js/tooltips.js';

vmRouter.add('popover_error', function(viewModel, vmRouter) {
    viewModel.instance = new FieldPopover(viewModel);
    // Override viewModel.name without altering it:
    vmRouter.exec('tooltip_error', viewModel);
    // or, to preserve the bound context (if any):
    vmRouter.exec('tooltip_error', viewModel, this);
});
```

where newly defined handler `popover_error` executes already existing `tooltip_error` viewmodel handler to re-use it's code.

The purpose of passing `this` bind context as an optional third argument of `vmRouter.exec()` call is to preserve currently passed Javascript bind context.

3.23.2 AJAX response routing

When one develops mixed web application with traditional server-side generated html responses but also having lots of AJAX interaction, with traditional approach, the developer would have to write a lot of boilerplate code, like this, html:

```
<button id="my_button" class="button btn btn-default">Save your form template</button>
```

Javascript:

```
import { AppConf } from '../../djk/js/conf.js';

$('#my_button').on('click', function(ev) {
    $.post(
        '/url_to_ajax_handler',
        {csrfmiddlewaretoken: AppConf('csrfToken')},
        function(response) {
            BootstrapDialog.confirm('After the registration our manager will contact
↪<b>you</b> ' +
                    'to validate your personal data.',
                function(result) {
                    if (result) {
                        window.location.href = '/another_url';
                    }
                }
            );
        },
        'json'
    )
});
```

Such code have many disadvantages:

1. Too much of callback nesting.
2. Repeated boilerplate code with `$.post()` numerous arguments, including manual specification `$.post()` arguments.
3. Route url names are hardcoded into client-side Javascript, instead of being supplied from Django server-side. If one changes an url of route in `urls.py`, and forgets to update url path in Javascript code, AJAX POST will fail.
4. What if the AJAX response should have finer control over client-side response? For example, sometimes you need to open `BootstrapDialog`, sometimes to redirect instead, sometimes to perform a custom client-side action for the same HTTP POST url?

Enter client-side viewmodels response routing: to execute AJAX post via button click, the following Jinja2 template code will be enough:

```
<button class="button btn btn-default" data-route="button-click">
    Save your form template
</button>
```

ajaxform.js AjaxButton class will care itself of setting Javascript event handler, performing AJAX request POST, then AJAX response routing will execute viewmodels returned from Django view. Define the view path in project `urls.py`:

```
from my_app.views import button_click
# ...
url(r'^button-click/$', button_click, name='button-click', kwargs={'is_anonymous':␣
↪True}),
```

Client-side routes

Let's implement the view. Return the list of viewmodels which will be returned via button click in my_app/views.py:

```
from django_jinja_knockout.decorators import ajax_required
from django_jinja_knockout.viewmodels import vm_list

@ajax_required
def button_click(request):
    return vm_list({
            'view': 'confirm',
            'title': 'Please enter <i>your</i> personal data.',
            'message': 'After the registration our manager will contact <b>you</b> to␣
↪validate your personal data.',
            'callback': vm_list({
                'view': 'redirect_to',
                'url': '/homepage'
            })
    })
```

Register AJAX client-side route (url name) in `settings.py`, to make url available in Javascript application:

```
DJK_CLIENT_ROUTES = {
    # True means that the 'button-click' url will be available to anonymous users:
    ('button-click', True),
}
```

Register `button-click` url mapped to my_app.views.button_click in your `urls.py`:

```
from my_app.views import button_click
# ...
url(r'^button-click/$', button_click, name='button-click', 'allow_anonymous': True,
↪'is_ajax': True}),
```

That's all.

Django view that processes `button-click` url (route) returns standard client-side viewmodels only, so it does not even require to modify a single bit of built-in Javascript code. To execute custom viewmodels, one would have to register their handlers in Javascript (see *Defining custom viewmodel handlers*).

It is possible to specify client-side routes per view, not having to define them globally in template context processor:

```
from django_jinja_knockout.views import create_page_context

def my_view(request):
    create_page_context(request).add_client_routes({
        'club_detail',
        'member_grid',
    })
```

or via decorator:

```
from django.shortcuts import render
from django_jinja_knockout.views import page_context_decorator

@page_context_decorator(client_routes={
        'club_detail',
        'member_grid',
})
def my_view(request):
    # .. skipped ..
    return render(request, 'sample_template.htm', {'sample': 1})
```

and per class-based view:

```
class MyGridView(KoGridView):

    client_routes = {
        'my_grid_url_name'
    }
```

It is possible to specify view handler function bind context via `.add()` method optional argument:

```
import { vmRouter } from '../../djk/js/ioc.js';

vmRouter.add({
    'set_context_title': {
        fn: function(viewModel) {
            // this == bindContext1
            this.setTitle(viewModel.title);
        },
        context: bindContext1
    },
    'set_context_name': {
        fn: function(viewModel) {
            // this == bindContext2
            this.setName(viewModel.name);
        },
        context: bindContext2
    }
});
```

It is also possible to override the value of context for viewmodel handler dynamically with AppPost() optional `bindContext` argument:

```
import { AppPost } from '../../djk/js/url.js';

AppPost('button-click', postData, bindContext);
```

That allows to use method prototypes bound to different instances of the same Javascript class:

```
import { inherit } from '../../djk/js/dash.js';
import { vmRouter } from '../../djk/js/ioc.js';
import { AppPost } from '../../djk/js/url.js';
import { Dialog } from '../../djk/js/dialog.js';

AjaxDialog = function(options) {
    inherit(Dialog.prototype, this);
    this.create(options);
};

(function(AjaxDialog) {

    AjaxDialog.receivedMessages = [];
    AjaxDialog.sentMessages = [];

    AjaxDialog.vm_addReceivedMessage = function(viewModel, vmRouter) {
        this.receivedMessages.push(viewModel.text);
    };

    AjaxDialog.vm_addSentMessage = function(viewModel, vmRouter) {
```

(continues on next page)

(continued from previous page)

```
        this.sentMessages.push(viewModel.text);
    };

    AjaxDialog.receiveMessages = function() {
        /**
         * When AJAX response will contain one of 'add_received_message' / 'add_sent_
↪message' viewmodels,
         * currently bound instance of AjaxDialog passed via AppPost() this argument
         * methods .vm_addReceivedMessage() / .vm_addSentMessage() will be called:
         */
        AppPost('my_url_name', this.postData, this);
    };

    // Subscribe to 'add_received_message' / 'add_sent_message' custom viewmodel␣
↪handlers:
    vmRouter.add({
        'add_received_message': AjaxDialog.vm_addReceivedMessage,
        'add_sent_message': AjaxDialog.vm_addSentMessage,
    });

})(AjaxDialog.prototype);

var ajaxDialog = new AjaxDialog(options);
ajaxDialog.receiveMessages();
```

Django `MyView` mapped to `'my_url_name'` (see *Context processor*) should return vm_list () instance with one of it's elements having the structure like this:

```
from django.views import View
from django_jinja_knockout.viewmodels import vm_list
# skipped ...

class MyView(View):

    def post(self, request, *args, **kwargs):
        return vm_list([
            {
                # Would call .vm_addReceivedMessage() of Javascript ajaxDialog␣
↪instance with 'text' argument:
                'view': 'add_received_message',
                'text': 'Thanks, I am fine!'
            },
            {
                # Would call .vm_addSentMessage() of Javascript ajaxDialog instance␣
↪with 'text' argument:
                'view': 'add_sent_message',
                'text': 'How are you?'
            }
        ])
```

to have `ajaxDialog` instance `.vm_addReceivedMessage()` / `.vm_addSentMessage()` methods to be actually called. Note that with viewmodels the server-side Django view may dynamically decide which client-side viewmodels will be executed, the order of their execution and their arguments like the value of 'text' dict key in this example.

In case AJAX POST button route contains kwargs / query parameters, one may use `data-url` html5 attribute instead of `data-route`:

```
<button class="btn btn-sm btn-success" data-url="{{
    tpl.reverseq('post_like', kwargs={'feed_id': feed.id}, query={'type': 'upvote'})
}}">
```

3.23.3 Non-AJAX server-side invocation of client-side viewmodels

Besides direct client-side invocation of viewmodels via vmrouter.js `vmRouter.respond()` method, and AJAX POST / AJAX GET invocation via AJAX response routing, there are two additional ways to execute client-side viewmodels with server-side invocation:

Client-side viewmodels can be injected into generated HTML page and then executed when page DOM is loaded. It's useful to prepare page / form templates which may require automated Javascript code applying, or to display BootstrapDialog alerts / confirmations when the page is just loaded. For example to display confirmation dialog when the page is loaded, you can override class-based view `get()` method like this:

```
from django_jinja_knockout.views ViewmodelView

class MyView(ViewmodelView):

    def get(self, request, *args, **kwargs):
        load_vm_list = self.page_context.onload_vm_list('client_data')
        load_vm_list.append({
            'view': 'confirm',
            'title': 'Please enter <i>your</i> personal data.',
            'message': 'After the registration our manager will contact <b>you</b> to␣
↪validate your personal data.',
            'callback': [{
                'view': 'redirect_to',
                'url': '/homepage'
            }]
        })
        return super().get(self, request, *args, **kwargs)
```

Read more about *PageContext (page_context)*.

The second way of server-side viewmodels invocation is similar to just explained one. It stores client-side viewmodels in the current user session, making them persistent across requests. This allows to set initial page viewmodels after HTTP POST or after redirect to another page (for example after login redirect), to display required viewmodels in the next request:

```
def set_session_viewmodels(request):
    last_message = Message.objects.last()
    # Custom viewmodel. Define it's handler at client-side with .add() method::
    # vmRouter.add('session_view', function(viewModel) { ... });
    # // or:
    # vmRouter.add({'session_view': {fn: myMethod, context: myClass}});
    view_model = {
        'view': 'session_view'
    }
    if last_message is not None:
        view_model['message'] = {
            'title': last_message.title,
            'text': last_message.text
        }
    page_context = create_page_context(request)
    session_vm_list = page_context.onload_vm_list(request.session)
```

(continues on next page)

(continued from previous page)

```
    # Find whether 'session_view' viewmodel is already stored in HTTP session vm_list:
    idx, old_view_model = session_vm_list.find_by_kw(view='session_view')
    if idx is not False:
        # Remove already existing 'session_view' viewmodel, otherwise they will␣
↪accumulate.
        # Normally it should not happen, but it's better to be careful.
        session_vm_list.pop(idx)
    if len(view_model) > 1:
        session_vm_list.append(view_model)
```

To inject client-side viewmodel when page DOM loads just once (function view):

```
onload_vm_list = create_page_context(request).onload_vm_list('client_data')
onload_vm_list.append({'view': 'my_view'})
```

In CBV view, inherited from ViewmodelView:

```
onload_vm_list = self.page_context.onload_vm_list('client_data')
onload_vm_list.append({'view': 'my_view'})
```

To inject client-side viewmodel when page DOM loads persistently in user session (function view):

```
session_vm_list = create_page_context(request).onload_vm_list(request.session)
session_vm_list.append({'view': 'my_view'})
```

In CBV view, inherited from ViewmodelView:

```
session_vm_list = self.page_context.onload_vm_list(request.session)
session_vm_list.append({'view': 'my_view'})
```

See PageContext.onload_vm_list() and vm_list.find_by_kw() for the implementation details.

3.23.4 Require viewmodels handlers

Sometimes there are many separate Javascript source files which define different viewmodel handlers. To assure that required external source viewmodel handlers are immediately available, use vmRouter instance `.req()` method:

```
import { vmRouter } from '../../djk/js/ioc.js';

vmRouter.req('field_error', 'carousel_images');
```

3.23.5 Nested / conditional execution of client-side viewmodels

Nesting viewmodels via callbacks is available for automated conditional / event subscribe viewmodels execution. Example of such approach is the implementation of `'confirm'` viewmodel in ioc.js `Dialog` callback via `vmRouter.respond()` method conditionally processing returned viewmodels:

```
import { vmRouter } from '../../djk/js/ioc.js';

var self = this;
var cbViewModel = this.dialogOptions.callback;
this.dialogOptions.callback = function(result) {
    // @note: Do not use alert view as callback, it will cause stack overflow.
```

(continues on next page)

(continued from previous page)

```
    if (result) {
        vmRouter.respond(cbViewModel);
    } else if (typeof self.dialogOptions.cb_cancel === 'object') {
        vmRouter.respond(self.dialogOptions.cb_cancel);
    }
};
```

3.23.6 Asynchronous execution of client-side viewmodels

There is one drawback of using vm_list: it is execution is synchronous and does not support promises by default. In some complex cases, for example when one needs to wait for some DOM loaded first, then to execute viewmodels, one may "save" viewmodels received from AJAX response, then "restore" (execute) these later in another DOM event / promise handler.

vmRouter method `.saveResponse()` saves received viewmodels:

```
import { vmRouter } from '../../djk/js/ioc.js';

vmRouter.add('popup_modal_error', function(viewModel, currentVmRouter) {
    // Save received response to execute it in the 'shown.bs.modal' event handler␣
↪(see just below).
    currentVmRouter.saveResponse('popupModal', viewModel);
    // Open modal popup to show actual errors (received as viewModel from server-
↪side).
    $popupModal.modal('show');
});
```

vmRouter method `loadResponse()` executes viewmodels previously saved with `.saveResponse()` call:

```
import { vmRouter } from '../../djk/js/ioc.js';

// Open modal popup.
$popupModal.on('shown.bs.modal', function (ev) {
    // Execute viewmodels previously received in 'popup_modal_error' viewmodel␣
↪handler.
    vmRouter.loadResponse('popupModal');
});
```

Multiple save points might be set by calling vmRouter `.saveResponse()` with the particular `name` argument value, then calling vmRouter `.loadResponse()` with the matching `name` argument value.

3.23.7 AJAX actions

Large classes of AJAX viewmodel handlers inherit from ActionsView at server-side and from Actions at client-side, which utilize the same viewmodel handler for multiple actions. It allows to structurize AJAX code and to build the client-server AJAX interaction more easily.

ModelFormActionsView and KoGridView inherit from ActionsView, while client-side ModelFormActions and GridActions inherit from Actions. See *Datatables* for more info.

Viewmodel router defines own (our) viewmodel name as Python ActionsView class viewmodel_name attribute / Javascript Actions class `.viewModelName` property. By default it has the value `action` but the derived classes may change it's name; for example grid datatables use `grid_page` as the viewmodel name.

Viewmodels which have non-matching names are not processed by Actions directly. Instead, they are routed to standard viewmodel handlers, added via vmRouter methods - see *Defining custom viewmodel handlers* section. Such way standard built-in viewmodel handlers are not ignored. For example server-side exception reporting is done with `alert_error` viewmodel handler (see ioc.js), while AJAX form validation errors are processed via `form_error` viewmodel handler (see tooltips.js).

The difference between handling AJAX viewmodels with vmRouter (see *Defining custom viewmodel handlers*) and AJAX actions is that the later shares the same viewmodel handler by routing multiple actions to methods of Actions class or it's descendant class.

Custom actions at the server-side

Server-side part of AJAX action with name `edit_form` is defined as ModelFormActionsView method `action_edit_form`:

```
def action_edit_form(self):
    obj = self.get_object_for_action()
    form_class = self.get_edit_form()
    form = form_class(instance=obj, **self.get_form_kwargs(form_class))
    return self.vm_form(
        form, verbose_name=self.render_object_desc(obj), action_query={'pk_val': obj.
↪pk}
    )
```

This server-side action part generates AJAX html form, but it can be arbitrary AJAX data passed back to client-side via one or multiple viewmodels.

To implement custom server-side actions, one has to:

- Inherit class-based view class from ActionsView or it's descendants like ModelFormActionsView or KoGridView (see also *Datatables*)
- Define the action by overriding the view class `.get_actions()` method
- Implement `action_my_action` method of the view class, which usually would return action viewmodel(s).

Here is the example of defining two custom actions, `save_equipment` and `add_equipment` at the server-side:

```
class ClubEquipmentGrid(KoGridView):

    def get_actions(self):
        actions = super().get_actions()
        actions['built_in']['save_equipment'] = {}
        actions['iconui']['add_equipment'] = {
            'localName': _('Add club equipment'),
            'css': 'iconui-wrench',
        }
        return actions

    # Creates AJAX ClubEquipmentForm bound to particular Club instance.
    def action_add_equipment(self):
        club = self.get_object_for_action()
        if club is None:
            return vm_list({
                'view': 'alert_error',
                'title': 'Error',
                'message': 'Unknown instance of Club'
            })
```

(continues on next page)

(continued from previous page)

```
        equipment_form = ClubEquipmentForm(initial={'club': club.pk})
        # Generate equipment_form viewmodel
        vms = self.vm_form(
            equipment_form, form_action='save_equipment'
        )
        return vms

    # Validates and saves the Equipment model instance via bound ClubEquipmentForm.
    def action_save_equipment(self):
        form = ClubEquipmentForm(self.request.POST)
        if not form.is_valid():
            form_vms = vm_list()
            self.add_form_viewmodels(form, form_vms)
            return form_vms
        equipment = form.save()
        club = equipment.club
        club.last_update = timezone.now()
        club.save()
        # Instantiate related EquipmentGrid to use it's .postprocess_qs() method
        # to update it's row via grid viewmodel 'prepend_rows' key value.
        equipment_grid = EquipmentGrid()
        equipment_grid.request = self.request
        equipment_grid.init_class()
        return vm_list({
            'update_rows': self.postprocess_qs([club]),
            # return grid rows for client-side EquipmentGrid component .updatePage(),
            'equipment_grid_view': {
                'prepend_rows': equipment_grid.postprocess_qs([equipment])
            }
        })
```

Note that `form_action` argument of the `.vm_form()` method overrides default action name for the generated form.

See the complete example: https://github.com/Dmitri-Sintsov/djk-sample/blob/master/club_app/views_ajax.py

Separate action handlers for each HTTP method

Since v1.1.0 it's possible to define separate action handlers for each HTTP method:

```
from django_jinja_knockout import tpl
from django_jinja_knockout.views import ActionsView
from django_jinja_knockout.viewmodels import vm_list

class MemberActions(ActionsView):

    template_name = 'member_template.htm'

    def get_actions(self):
        return {
            # action type
            'built_in': {
                # action definition
                # empty value means the action has no options and is enabled by␣
↪default
                'reply': {}
```

(continues on next page)

(continued from previous page)

```
            }
        }

    # will be invoked for HTTP GET action 'reply':
    def get_action_reply(self):
        return tpl.Renderer(self.request, 'action_reply_template.htm', {
            'component_atts': {
                'class': 'component',
                'data-component-class': 'MemberReplyActions',
                'data-component-options': {
                    'route': self.request.resolver_match.view_name,
                    'routeKwargs': copy(self.kwargs),
                    'meta': {
                        'actions': self.vm_get_actions(),
                    },
                },
            }
        })()

    # will be invoked for HTTP POST action 'reply',
    # usually via Javascript MemberReplyActions.ajax('reply'):
    def post_action_reply(self):
        return vm_list({
            'members': Member.objects.filter(club=club, role=role)
        })

    def get(self, request, *args, **kwargs):
        reply = self.conditional_action('reply')
        if reply:
            return reply
        else:
            return super().get(request, *args, **kwargs)
```

However, by default automatic invocation of action handler is performed only for HTTP POST. To perform HTTP GET action, one has to invoke it manually by calling conditional_action method in `get` method view code (see above), or in `member_template.htm` Jinja2 template (in such case custom `get` method is not required):

```
{% set reply = view.conditional_action('reply') -%}
{% if reply %}
    {{ reply }}
{% endif -%}
```

See *Component IoC* how to register custom Javascript `data-component-class`, like `MemberReplyActions` mentioned in this example.

The execution path of the action

The execution of action usually is initiated in the browser via the *Components* DOM event / Knockout.js binding handler, or is programmatically invoked in Javascript via the Actions inherited class `.perform()` method:

```
import { inherit } from '../../djk/js/dash.js';
import { Actions } from '../../djk/js/actions.js';
// import { GridActions } from '../../djk/js/grid/actions.js';

ClubActions = function(options) {
```

(continues on next page)

(continued from previous page)

```
    // Comment out, when overriding Grid actions.
    // inherit(GridActions.prototype, this);
    inherit(Actions.prototype, this);
    this.init(options);
};

var clubActions = new ClubActions({
    route: 'club_actions_view',
    actions: {
        'review_club': {},
    }
});
var actionOptions = {'club_id': 1};
var ajaxCallback = function(viewmodel) {
    console.log(viewmodel);
    // process viewmodel...
};
clubActions.perform('review_club', actionOptions, ajaxCallback);
```

`actionOptions` and `ajaxCallback` arguments are the optional ones.

- In case there is `perform_review_club()` method defined in `ClubActions` Javascript class, it will be called first.
- If there is no `perform_review_club()` method defined, `.ajax()` method will be called, executing AJAX POST request with `actionOptions` value becoming the queryargs to the Django url `club_actions_view`.
  - In such case, Django `ClubActionsView` view class should have `review_club` action defined (see *Custom actions at the server-side*).
  - Since v0.9.0 `ajaxCallback` argument accepts *Javascript bind context* as well as viewmodel `before` and `after` callbacks, to define custom viewmodel handlers on the fly:

```
var self = this;
clubActions.ajax(
     'member_names',
     {
         club_id: this.club.id,
     },
     {
         // 'set_members' is a custom viewmodel handler defined on the fly:
         after: {
             set_members: function(viewModel) {
                 self.setMemberNames(viewModel.users);
             },
         }
     }
);

clubActions.ajax(
     'member_roles',
     {
         club_id: this.club.id,
     },
     // viewmodel response will be returned to the bound method␣
↪clubRolesEditor.updateMemberRoles():
     {
```

(continues on next page)

(continued from previous page)

```
            context: clubRolesEditor,
            fn: ClubRolesEditor.updateMemberRoles,
        }
);
```

- Note: `actionOptions` value may be dynamically altered / generated via optional `queryargs_review_club()` method in case it's defined in `ClubActions` class.
- Custom `perform_review_club()` method could execute some client-side Javascript code first then call `.ajax()` method manually to execute Django view code, or just perform a pure client-side action only.
- In case `ClubActions` class `.ajax()` method was called, the resulting viewmodel will be passed to `ClubActions` class `callback_review_club()` method, in case it's defined. That makes the execution chain of AJAX action complete.

See *Client-side routes* how to make `club_actions_view` Django view name (route) available in Javascript.

See club-grid.js for sample overriding of `Grid` actions. See *Datatables* for more info.

Overriding action callback

Possible interpretation of server-side ActionsView class `.action\*()` method (eg `.action_perform_review()`) result (AJAX response):

- `None` - client-side Actions class `.callback_perform_review()` method will be called, no arguments passed to it except the default viewmodel_name;
- `False` - client-side Actions class `.callback_perform_review()` will be suppressed, not called at all;
- `list` / `dict` - the result will be converted to vm_list
  - In case the viewmodel `view` key is omitted or contains the default Django view viewmodel_name attribute value, the default client-side Actions class `.callback_perform_review()` method will be called;
  - The rest of viewmodels (if any) will be processed by the vmRouter;
- *special case*: override callback method by routing to `another_action` Javascript Actions class `.callback_another_action()` method by providing callback_action key with the value `another_action` in the viewmodel dict response.

  For example to conditionally "redirect" to `show_readonly` action callback for `edit_inline` action in a KoGridView derived class:

```
from django_jinja_knockout import tpl
from django_jinja_knockout.views import KoGridView

class CustomGridView(KoGridView):

    # ... skipped...

    def action_edit_inline(self):
        # Use qs = self.get_queryset_for_action() in case multiple objects are␣
↪selected in the datatable.
        obj = self.get_object_for_action()
        if obj.is_editable:
            if obj.is_invalid:
                return {
                    'view': 'alert_error',
```

(continues on next page)

(continued from previous page)

```
                    'title': obj.get_str_fields(),
                    'message': tpl.format_html('<div>Invalid object={}</div>',␣
↪obj.pk)
                }
            else:
                title = obj.get_str_fields()
                # Action.callback_show_readonly() will be called instead of the␣
↪default
                # Action.callback_edit_inline() with the following viewmodel as␣
↪the argument.
                return {
                    'callback_action': 'show_readonly',
                    'title': title,
                }
        else:
            return super().action_edit_inline()
```

Custom actions at the client-side

To implement or to override client-side processing of AJAX action response, one should define custom Javascript class, inherited from Actions (or from GridActions in case of custom grid *Datatables*):

```
import { inherit } from '../../djk/js/dash.js';
import { Actions } from '../../djk/js/actions.js';

MyModelFormActions = function(options) {
    inherit(Actions.prototype, this);
    this.init(options);
};
```

Client-side part of `edit_form` action response, which receives AJAX viewmodel(s) response is defined as:

```
import { ModelFormDialog } from  '../../djk/js/modelform.js';

(function(MyModelFormActions) {

    MyModelFormActions.callback_edit_form = function(viewModel) {
        viewModel.owner = this.grid;
        var dialog = new ModelFormDialog(viewModel);
        dialog.show();
    };

    // ... See more sample methods below.

})(MyModelFormActions.prototype);
```

Client-side Actions descendant classes can optionally add queryargs to AJAX HTTP request in a custom `queryargs_ACTION_NAME` method:

```
MyFormActions.queryargs_edit_form = function(options) {
    // Add a custom queryarg to AJAX POST:
    options['myArg'] = 1;
};
```

Client-side Actions descendant classes can directly process actions without calling AJAX viewmodel server-side part (client-only actions) by defining `perform_ACTION_NAME` method:

```
import { ActionTemplateDialog } from '../../djk/js/modelform.js';

MyFormActions.perform_edit_form = function(queryArgs, ajaxCallback) {
    // this.owner may be instance of Grid or another class which implements proper␣
↪owner interface.
    new ActionTemplateDialog({
        template: 'my_form_template',
        owner: this.owner,
        meta: {
            user_id: queryArgs.user_id,
        },
    }).show();
};
```

For such client-only actions ActionTemplateDialog utilizes Underscore.js templates for one-way binding, or Knockout.js templates when two way binding is required. Here is the sample template

```
<script type="text/template" id="my_form_template">
    <card-default>
        <card-body>
            <form class="ajax-form" enctype="multipart/form-data" method="post" role=
↪"form" data-bind="attr: {'data-url': actions.getLastActionUrl()}">
                <input type="hidden" name="csrfmiddlewaretoken" data-bind="value:␣
↪getCsrfToken()">
                <div class="jumbotron">
                    <div class="default-padding">
                        The user id is <span data-bind="text: meta.user_id"></span>
                    </div>
                </div>
            </form>
        </card-body>
    </card-default>
</script>
```

Custom grid actions should inherit from both GridActions and it's base class Actions:

```
import { inherit } from '../../djk/js/dash.js';
import { Actions } from '../../djk/js/actions.js';
import { GridActions } from '../../djk/js/grid/actions.js';

MyGridActions = function(options) {
    inherit(GridActions.prototype, this);
    inherit(Actions.prototype, this);
    this.init(options);
};
```

For more detailed example of using viewmodel actions routing, see the documentation *Datatables* section *Client-side action routing*. Internally, AJAX actions are used by EditForm, EditInline and by Grid client-side components. See also EditForm usage in `djk-sample` project.

3.24 Built-in views

3.24.1 Inheritance hierarchy

Version 1.0.0:

- `PageContextMixin(TemplateResponseMixin, ContextMixin, View):` - provides *PageContext (page_context)*;
- `ViewmodelView(TemplateResponseMixin, ContextMixin, View)` - render component templates and process viewmodels response (see *Client-side viewmodels and AJAX response routing*);
  - `FormatTitleMixin(PageContextMixin)` - customizes *View title*;
  - `BsTabsMixin(PageContextMixin)` - *BsTabsMixin*;
  - `FormViewmodelsMixin(ViewmodelView)` - forms and forms fields AJAX viewmodel response;
  - `BaseFilterView(PageContextMixin)` - model queryset filtering / ordering base view, used by both *ListSortingView* and AJAX *Datatables*;
  - `ActionsView(FormatTitleMixin, ViewmodelView)` - generic actions for viewmodels (*AJAX actions*);
  - `ModelFormActionsView(ActionsView, FormViewmodelsMixin)` - AJAX actions to display / edit Django ModelForm / inline formsets;
    - `GridActionsMixin(ModelFormActionsView)` - AJAX actions to display / process ModelForm datatable (grid);
      - `KoGridView(BaseFilterView, GridActionsMixin)` - includes all the actions and functionality from the above classes and adds common code base for paginated *Datatables*;
      - `KoGridRelationView(KoGridView)` - used by BaseGridWidget, see *ForeignKeyGridWidget*;

3.24.2 Views kwargs

The built-in middleware is applied only to the views which belong to modules (Django apps) registered in project `settings` module `DJK_APPS` variable like this:

```
DJK_APPS = (
    'my_app',
)

INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.sites',
    'django_jinja',
    'django_jinja.contrib._humanize',
    'djk_ui',
    'django_jinja_knockout',
) + DJK_APPS
```

See `djk-sample` settings.py for the complete example.

See also *Middleware installation*, *Middleware security* for the description of additional permission check view kwargs.

View title

View title is optionally defined as url kwargs `'view_title'` key value:

```
from my_app.views import signup
# ...
url(r'^signup/$', signup, name='signup', kwargs={'view_title': 'Sign me up', 'allow_
↪anonymous': True})
```

to be used in generic Jinja2 templates (v0.8.x or older):

```
{{ request.resolver_match.view_title }}
```

or (v1.0.0 or newer), which uses *PageContext (page_context)*:

```
{{ page_context.get_view_title() }}
```

Django view kwargs are originally available in `request.resolver_match.kwargs` attribute to use in forms / templates, when needed.

3.24.3 FormWithInlineFormsetsMixin

The base class for the set of class-based views that create / edit the related form with the inline formsets with built-in support of `django_jinja_knockout.forms` module `FormWithInlineFormsets` class.

It supports both non-AJAX and AJAX form submission and validation. AJAX validation and AJAX success action is performed with built-in extensible *Client-side viewmodels and AJAX response routing*. By default AJAX supports class-based view .get_success_url() automatic client-side redirect on success which can be replaced to another AJAX viewmodel handler via overriding this method in derived view class.

Setting class attribute ajax_refresh value to `True` causes the successful AJAX submission of the form with the inline formsets to refresh the form HTML with just saved values instead of .get_success_url() redirect to another url. This is useful when the additional client-side processing is required, or when the form is the part of some component, like *bs_tabs()* tab.

Zero or one related form is supported and zero / one / many of inline formsets. Adding / removing inline forms is supported via Knockout.js custom bindings with XSS protection, which are generated via set_knockout_template function that uses InlineFormRenderer with formset empty_form. HTML rendering usually is performed with Jinja2 bs_inline_formsets() macro.

The following views inherit this class:

- `InlineCreateView` - CBV view to create new models with one to many related models.
- `InlineCrudView` - CBV view to create / edit models with one to many related models.
- `InlineDetailView` - CBV view to display or to update models with one to many related models. Suitable both for CREATE and for VIEW actions, last case via `ModelForm` with `metaclass=DisplayModelMetaclass`.

3.24.4 BsTabsMixin

- `BsTabsMixin` - automatic template context processor for CBV's, which uses `prepare_bs_navs()` function and *bs_navs()* jinja2 macro to navigate through the navbar list of visually grouped Django view links.
- `prepare_bs_navs()` - highlight current url of Bootstrap navbar. It's possible to override the highlighted navbar link by specifying navs[]['attrs']['class'] = 'active' value.

To implement server-side tabs navigation, one should define class inherited from *BsTabsMixin* with custom .get_main_navs() method of this class. For the example:

```
class ClubNavsMixin(BsTabsMixin):

    def get_main_navs(self, request, object_id=None):
        main_navs = [
            {'url': reverse('club_list'), 'text': 'List of clubs'},
            {'url': reverse('club_create'), 'text': 'Create new club'}
        ]
        if object_id is not None:
            main_navs.extend([
                {
                    'url': reverse('club_detail', kwargs={'club_id': object_id}),
                    'text': format_html('View "{}"', self.object.title)
                },
                {
                    'url': reverse('club_update', kwargs={'club_id': object_id}),
                    'text': format_html('Edit "{}"', self.object.title)
                }
            ])
        return main_navs
```

Then every class which uses the tabs should inherit (mix) from ClubNavsMixin:

```
class ClubEditMixin(ClubNavsMixin):

    client_routes = {
        'manufacturer_fk_widget',
        'profile_fk_widget'
    }
    template_name = 'club_edit.htm'
    form_with_inline_formsets = ClubFormWithInlineFormsets


class ClubCreate(ClubEditMixin, InlineCreateView):

    def get_bs_form_opts(self):
        return {
            'class': 'club',
            'title': 'Create sport club',
            'submit_text': 'Save sport club'
        }

    def get_success_url(self):
        return reverse('club_detail', kwargs={'club_id': self.object.pk})
```

`main_navs` may be the instance of NavsList type, which holds `props` dict attribute, allowing to pass extra data to Jinja2 template which then would call *bs_navs()* Jinja2 macro. That allows to set the navbar menu CSS styles dynamically via NavsList `props`.

3.24.5 ListSortingView

ListSortingView is a ListView with built-in support of sorting and field filtering.

Version 1.1.0 implements standard Django `range` / `date` / `datetime` filter fields, which could be extended by specifying custom `template` / component_class arguments of `allowed_filter_fields` dict items, see the sample ActionList:

```
from django_jinja_knockout.views import ListSortingView

from .models import Action


class ActionList(ListSortingView):
    # Enabled always visible paginator links because there could be many pages of␣
↪actions, potentially.
    always_visible_links = True
    model = Action
    grid_fields = [
        [
            'performer',
            'performer__is_superuser',
            'date',
        ],
        'action_type',
        'content_object'
    ]
    allowed_sort_orders = [
        'performer',
        'date',
        'action_type',
    ]

    def get_allowed_filter_fields(self):
        allowed_filter_fields = {
            # Override default templates for filter fields:
            'action_type': {'template': 'bs_navs.htm'},
            # Specify custom client-side Javascript component class to extend it's␣
↪functionality:
            'id': {
                'component_class': 'RangeFilter',
            },
            'date': None,
            # Generate widget choices for contenttypes framework:
            'content_type': self.get_contenttype_filter(
                ('club_app', 'club'),
                ('club_app', 'equipment'),
                ('club_app', 'member'),
            ),
        }
        return allowed_filter_fields
```

See *Component IoC* how to register custom Javascript `component_class`.

It's possible to specify `allowed_filter_fields` widget `choices`, `template` name and extra options at once:

```
allowed_filter_fields = {
    'club': {
        'choices': [(club.pk, club.title) for club in Club.objects.
↪filter(category=Club.CATEGORY_PROFESSIONAL)],
        'multiple_choices': False,
        'component_class': 'CustomChoicesFilter',
        # should generate client-side component which uses specified component_class:
        'template': 'custom_choices_widget.htm',
    },
}
```

It's also possible to override values of filter template kwargs or to add extra template kwargs with `template_kwargs` option:

```
allowed_filter_fields = {
    'content_type': self.get_contenttype_filter(
        'template': 'bs_dropdown.htm',
        'choices':
            ('club_app', 'club'),
            ('club_app', 'equipment'),
            ('club_app', 'member'),
        'template_kwargs': {
            'menu_attrs': {
                'class': 'dropdown-menu dropdown-menu-left'
            }
        },
    ),
}
```

- Some options, such as `template` / component_class are applicable to any filter, inherited from BaseFilter.
- `choices` / `multiple_choices` options are applicable only to ChoicesFilter.
- See ChoicesFilter / RangeFilter for the examples of widget.
- See *Components* how to create client-side components.
- `KoGridView` *Datatables* uses limited subset of filters, because it has it's widgets generated by ko_grid_body.htm macro. See ioc_field_filter implementations.

BaseFilterView interface (`grid_fields` / `allowed_sort_orders` / `allowed_filter_fields`) is inherited by both *ListSortingView* and by AJAX-based `KoGridView` (*Datatables*), which allows to switch between traditional full page server-rendered HTML views and dynamic AJAX views just via changing their parent class name.

- FoldingPaginationMixin - ListView / *ListSortingView* mixin that enables advanced pagination in bs_pagination / bs_list Jinja2 macros.

3.24.6 Viewmodels views and actions views

- ViewmodelView - base view; GET request usually generates html template, POST - returns AJAX viewmodels. It is the base class for the following built-in classes:
- ActionsView - implements AJAX actions router and their viewmodels responses. Actions allow to perform different AJAX POST requests to the same view. The responses are the AJAX viewmodels.
- ModelFormActionsView - implements AJAX actions specific to Django ModelForm / inline formsets handling: rendering form / validating / saving. It is also the base class for grids (datatables) actions, because the editing of datatables includes form editing via GridActionsMixin.

For introduction to viewmodels, see *Client-side viewmodels and AJAX response routing*.

For more detailed explanation of these views see *AJAX actions*.

3.24.7 Datatables

- KoGridView - together with `grid.js` allows to create AJAX powered django.admin-like datatables with filtering, sorting, search, CRUD actions and custom actions. See *Datatables* for more details.

3.24.8 Useful methods / classes of the views module

- `auth_redirect()` - authorization required response with redirect to login. Supports 'next' url query argument. Supports JSON viewmodel response.
- `cbv_decorator()` - may be used to check class-based views permissions.
- `ContextDataMixin` - allows to inject pre-defined dict of `extra_context_data` into template context of class-based view.

3.25 widgets.py

3.25.1 OptionalWidget

OptionalWidget - A two-component `MultiField`: a checkbox that indicates optional value and a field itself (`widget_class` = `Textarea` by default). The field itself is enabled / disabled according to the checkbox state via client-side $.optionalInput plugin, implemented in plugins.js:

```
from django_jinja_knockout.widgets import OptionalWidget

OptionalWidget(attrs={'class': 'autogrow vLargeTextField', 'cols': 40, 'rows': 2})
```

See also vLargeTextField usage, although it's optional and is not the requirement for *OptionalWidget*.

3.25.2 DisplayText

DisplayText - Read-only widget for existing `ModelForm` bound objects. Assign to `ModelForm.widgets` or to `ModelForm.fields.widget` to make selected form fields displayed as read-only text.

Use `DisplayModelMetaclass` from `django_jinja_knockout.forms` to set all field widgets of form as `DisplayText`, making the whole form read-only.

In last case the form will have special renderer with table like view. See *Displaying read-only "forms"* for more info.

Widget allows to specify custom formatting callback to display complex fields, including foreign relationships, predefined string mapping for scalar `True` / `False` / `None` and layout override for *bs_form()* / *bs_inline_formsets()* macros. Note that it's possible to call these macros from Django language templates like this:

```
{% jinja 'bs_form.htm' with _render_=1 form=form action=view_action opts=opts %}
```

For example, to override `Member` model `note` field *DisplayText* widget html output via `get_text_method()`:

```
class MemberDisplayForm(WidgetInstancesMixin, RendererModelForm,␣
↪metaclass=DisplayModelMetaclass):

    class Meta:

        def get_note(self, value):
            # self.instance.accepted_license.version
            if self.instance is None or self.instance.note.strip() == '':
                # Do not display empty row.
                self.skip_output = True
                return None
            return format_html_attrs(
                '<button {attrs}>Read</button>',
```

(continues on next page)

(continued from previous page)

```
                attrs={
                    'class': 'component btn btn-info',
                    'data-component-class': 'Dialog',
                    'data-event': 'click',
                    'data-component-options': {
                        'title': '<b>Note for </b> <i>{}</i>'.format(self.instance.
↪profile),
                        'message': format_html('<div class="preformatted">{}</div>',␣
↪self.instance.note),
                        'method': 'alert'
                    }
                }
            )

        model = Member
        fields = '__all__'
        widgets = {
            'note': DisplayText(get_text_method=get_note)
        }
```

See *Component IoC* how to register custom Javascript `data-component-class`.

See DisplayText sample for the complete example.

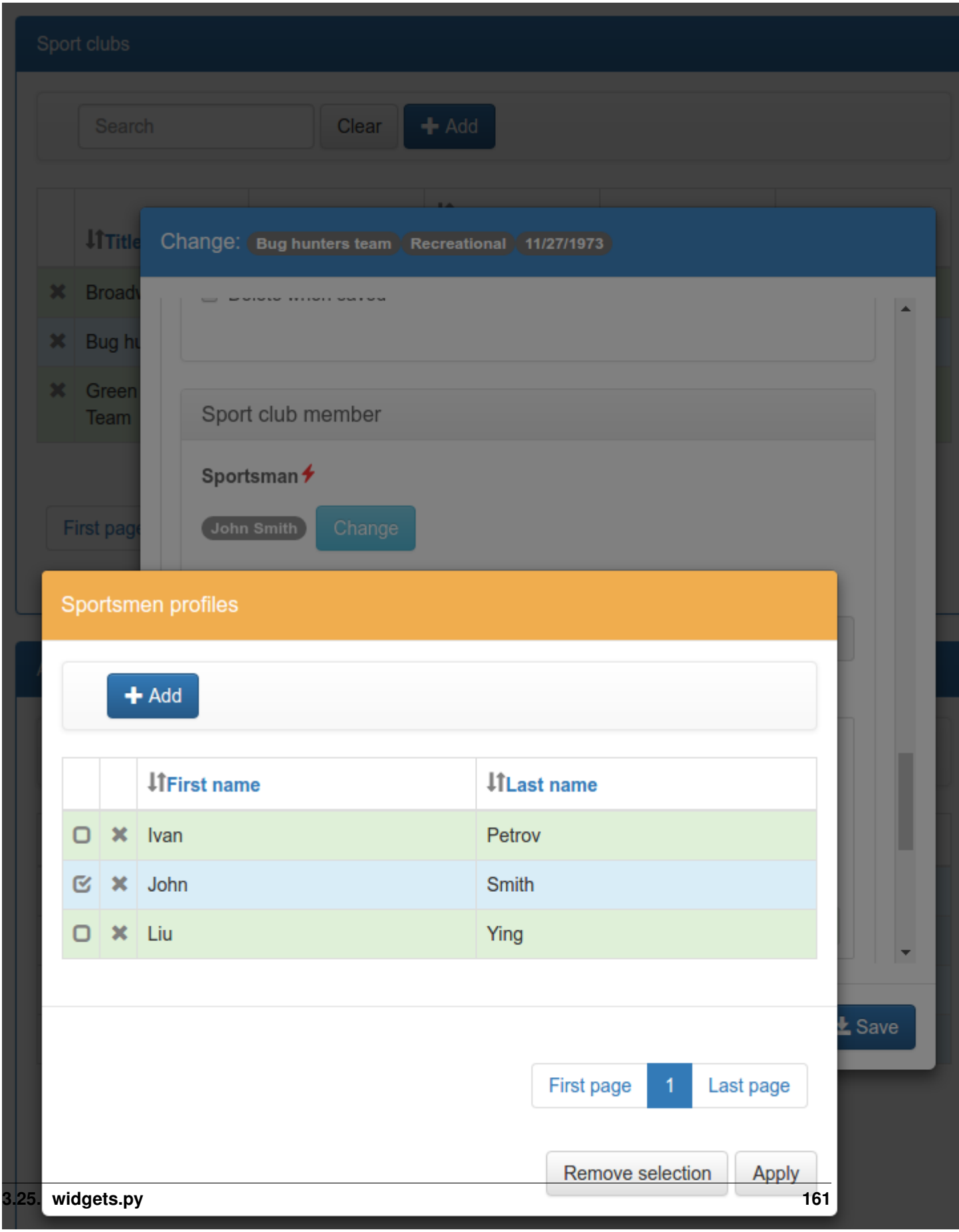
3.25.3 ForeignKeyGridWidget

Implements django.admin -like widget to select the foreign key value with the optional support of in-place CRUD editing of foreign key table rows.

- ForeignKeyGridWidget wiki

See *ForeignKeyGridWidget* section of *Datatables* for the detailed explanation.

Here is the screenshot of the *ForeignKeyGridWidget* running djk_sample project:


Sport clubs
Search
Clear
Add
Title
Broadv
Bug hu
Green Team
First page
Change: Bug hunters team Recreational 11/27/1973
Sport club member
Sportsman
John Smith
Change
Save
Sportsmen profiles
Add
First name
Last name
Ivan
Petrov
John
Smith
Liu
Ying
First page
1
Last page
Remove selection
Apply

3.25.4 MultipleKeyGridWidget

django.admin -like widget to select multiple foreign key values for the form relation.

See *MultipleKeyGridWidget* section of *Datatables* for the detailed explanation.

3.25.5 PrefillWidget

PrefillWidget - Django form input field which supports both free text and quick filling of input text value from the list of prefilled choices. ListQuerySet has `prefill_choices()` method, which allows to generate lists of choices for PrefillWidget initial values like this:

```
from django_jinja_knockout.widgets import PrefillWidget
from django_jinja_knockout.query import ListQuerySet

# ...

self.related_members_qs = ListQuerySet(
    Member.objects.filter(
        club__id=self.request.resolver_match.kwargs.get('club_id', None)
    )
)
if self.related_members_qs.count() > 1 and isinstance(form, MemberForm):
    # Replace standard Django CharField widget to PrefillWidget with incorporated␣
↪standard field widget:
    form.fields['note'].widget = PrefillWidget(
        data_widget=form.fields['note'].widget,
        choices=self.related_members_qs.prefill_choices('note')
    )
    # Replace one more field widget to PrefillWidget:
    form.fields['name'].widget = PrefillWidget(
        data_widget=form.fields['name'].widget,
        choices=self.related_members_qs.prefill_choices('name')
    )
```

See `djk-sample` project for the sample of PrefillWidget usage with inline formsets. It is even simpler to use this widget in single ModelForm without the inline formsets.

See widget_prefill_dropdown.htm macro for the default rendering of PrefillWidget.

CHAPTER 4

Datatable grids

See *Datatables*

CHAPTER 5

Contributing

django_jinja_knockout is an open source project originally written by very poor guy from Russia so feel free to support it either by contributing new features / fixes / unit tests or by hiring me remotely to develop additional required features.

Any non-trivial contribution will be recorded in authors list.

- Unit tests are partially implemented in djk-sample project which is used as showcase / testing project. Selenium is used to test client-side parts of `django-jinja-knockout`.
- The app is used in large enough project which is tested via actual manual work by real end-users.

You can contribute in many ways:

5.1 Types of Contributions

Any good quality contribution is welcome.

5.1.1 Report Bugs

Report bugs at https://github.com/Dmitri-Sintsov/django-jinja-knockout/issues

If you are reporting a bug, please include:

- Your operating system name and Python / Django version used.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.
- Feel free to fix bug or to suggest / implement a feature at github.

5.1.2 Translation / Localization

If you know one of the currently available languages, please contribute to localization of the project:

- django-jinja-knockout python localization
- django-jinja-knockout javascript localization

There are not so many strings to translate so it should not take too much of time. However even incomplete localization is better than none.

5.1.3 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/Dmitri-Sintsov/django-jinja-knockout/issues

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that good quality contributions are welcome!

5.2 Get Started!

Ready to contribute? Here's how to set up `django_jinja_knockout` for local development.

Fork the `django_jinja_knockout` repo on GitHub.

- Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ apt-get install python3-pip
$ python3 -m venv django-jinja-knockout
$ cd django-jinja-knockout/
# Clone your fork locally
$ git clone https://github.com/your_github_account/django-jinja-knockout.git
$ source bin/activate
# python setup.py develop
$ cd django-jinja-knockout
$ python3 -m pip install -U -r requirements.txt
```

Note that without Django installed, there is not much of usage for this pluggable app.

- Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

- Automated tests are partially implemented in djk-sample unit tests.
- Check that your changes passes flake8:

```
$ pip3 install flake8 flake8-bugbear
$ flake8 --ignore E501 django_jinja_knockout
```

Then run the tests in djk-sample unit tests

- Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

- Submit a pull request through the GitHub website.

5.2.1 Write Documentation

django_jinja_knockout reusable application could always use more documentation, whether as part of the official docs or in docstrings (but please not very long bloated ones).

Especially because I am not a native English speaker, though I try my best to avoid mistakes.

To check documentation changes install sphinx:

```
python3 -m pip install sphinx
```

then run in your active virtual environment:

```
$ cd $VIRTUAL_ENV/django-jinja-knockout/docs
$ make html
$ firefox _build/html/index.html &
```

5.3 Pull Request Guidelines

1. It would be great if the pull request included automated tests for djk-sample.
2. If the pull request adds functionality, the docs should be updated. Implement new functionality into a function / class / method with a docstring. Major and important features should be briefly described in the README.rst / QUICKSTART.rst. Detailed documentation is not required but is welcomed and should be implemented in separate rst file.
3. The pull request should work for Python 3 / Django LTS at least.

CHAPTER 6

Credits

6.1 Development Lead

- Dmitriy Sintsov <questpc256@gmail.com>

6.2 Contributors

- Melvyn Sopacua: Compatibility to 1.10+ new-style middleware; Dutch localization.
- kiwnix: Middleware import fixes;

Localization:

- Chinese: `goldmonkey`
- Dutch: Melvyn Sopacua
- Polish: `pawelkoston`
- Spanish: `Julio Cesar Cabrera Cabrera`, kiwnix

CHAPTER 7

History

7.1 0.1.0

- To be released on PyPI.

7.2 0.2.0

- Django 1.8 / 1.9 / 1.10, Python 3.4 / 3.5 support.
- `djk-sample` demo / automated testing project.
- "django.admin-like" AJAX functionality implemented via KoGridView class-based view.
- `$.inherit()` Javascript prototype inheritance function now supports multi-level inheritance with nested `._super._call()`.
- FilteredRawQuerySet supports Django raw querysets with `.filter()` / `.exclude()` / `.order_by()` / `.values()` / `.values_list()` and SQL level slicing.
- ForeignKeyGridWidget provides `ForeignKeyRawIdWidget` -like functionality via AJAX query / response in non-admin forms to select ModelForm foreign key values.
- Client-side generation of view urls with kwargs in Javascript client-side routes via `App.routeUrl()`.
- Nested autocompiled underscore.js client-side templates for Javascript components, primarily used with Knockout.js, but is not limited to.

7.3 0.3.0

- `ContentTypeLinker` - added method to return html representation of content types framework related object (html link with the description by default).
- FilteredRawQuerySet now supports more precise `.count()` method to calculate the length of raw queryset.

- ListQuerySet implements large part of queryset methods for the lists of Django model instances. Such lists are created by Django queryset `.prefetch_related()` method.
- Auto-highlight bootstrap navs which have 'auto-highlight' css class at client-side.
- `bs_tabs()` Jinja2 macro which simplifies generation of bootstrap tabs. Bootstrap tabs now support automatic switching via window.location.hash change.
- `ListSortingView` improvements:
  - Supports graceful handling of error reporting, producing in-place messages instead of just rising an exception.
  - `.get_filter_args()` / `.get_no_match_kwargs()` methods are implemented to generate macro arguments used in bs_list.htm Jinja2 template. This allows to override default messages for field filters / no match reports in the grid classes.
- KoGridView has multiple improvements:
  - `decimal` field filter is renamed to `number` as now it supports both Django model `DecimalField` and `IntegerField`.
  - Django model `IntegerField` is now bound either to `choices` type filter, when it has non-empty `choices` attribute, or to `number` type filter to select range of values, otherwise.
  - Action handlers do not require to return default viewmodel `view` name manually, now it's being done automatically (when viewmodel `view` name is not specified).
  - `get_default_grid_options()` method was renamed to shorter `get_grid_options()` method.
  - `grid_options` may be defined as class attribute, not having to always define `get_grid_options()` method which is more verbose (but is more flexible).
  - `discover_grid_options()` method was implemented to populate grid `fkGridOptions` which are used to setup foreign key filter fields automatically (when possible). That allows to reduce boilerplate data in `grid_options` / `get_grid_options()`, especially when using nested foreign key filters. `fkGridOptions` nested dictionary still can be specified manually as the part of `get_grid_options()` result, in complex cases (eg. DB or view kwargs based options).
  - Enable quick selection / deselection of currently displayed grid rows when `selectMultipleRows` is `true`.
- ForeignKeyGridWidget also autodetects foreign key filter `fkGridOptions`.
- `SendmailQueue` supports extension of `add()` / `flush()` methods via ioc class.
- `SendmailQueue` may be used to send uncaught exception emails when running in production mode.

7.4 0.4.0

- Improvements in testing support:
  - `AutomationCommands` now uses yield to generate the sequence of opcodes and their args, resulting in cleaner code.
  - `SeleniumCommands` is reworked into `BaseSeleniumCommands`. It supports:
    - Saving current database state to Django fixtures at the particular points of tests via dump_data command. That allows to skip already debugged parts of tests via .has_fixture() method, greatly reducing the time required to develop and debug long running Selenium tests. To make proper order (sequence) of stored / loaded fixtures, one has to define fixtures_order attribute of `DjkTestCase` derived class.

    * Automatic retry of the last Selenium commands execution in case current command is timed out when running at slow client due to DOM is not being updated in time.
    * css parsing / xpath string escaping.
  - `SeleniumQueryCommands` implements generic Selenium commands, including Django reverse url support for navigation bar, anchors and forms, which could be useful in any Django application.
  - `DjkSeleniumQueryCommands` implements additional Selenium commands related to django-jinja-knockout functionality, such as BootstrapDialog and Knockout.js grids / widgets support.

7.4.1 forms.py

- `BootstrapModelForm` always populates `.request` attribute for convenience.
- `CustomFullClean` / `StripWhitespaceMixin` mixins for Django forms.

7.4.2 middleware.py

- `ContextMiddleware` class:
  - Supports request mocking when running not under HTTP server, for example as shell command / celery task.
  - Supports request-time storage of lists / dicts of objects via add_instance / yield_out_instances methods.

7.4.3 query.py

- FilteredRawQuerySet supports Q expressions (Q objects) with relation mapping.

7.4.4 views submodule

- `BaseFilterView`
  - `filter_queryset()` now supports args in addition to kwargs. That allows to use Django `Q` objects in grids and lists, although actual generation of `Q` objects is still limited to `None` value filtering.
  - `None` can be valid value of field filter query. It is mapped to `is_null` field lookup, also it uses Django `Q.__or__` operation in case `None` is presented in the list of field filter values.
  - Query filters now support `in` clause for drop-down `choice` filter.

7.4.5 widgets.py

- `DisplayText` field widget `__init__()` method now supports two types of `get_text` callback arguments:
  - `get_text_method` which binds passed function to DisplayText widget instance (self as first argument)
  - `get_text_fn` which uses unbound function (no self).

If form that defined widget uses `WidgetInstancesMixin` and model field instance has `get_str_fields()` method implemented, such field will be auto-rendered via `print_list_group()` / `print_bs_well()` functions of `tpl` module to produce structured output.

7.4.6 ko_grid_body.htm

- Fixed `ko_grid_body()` macro not including underscore.js templates copied with different `template_id` when these templates were called from related underscore.js templates.

7.4.7 grid.js

- Reset filter now uses `undefined` value instead of `null` value because filtering by `None` value is now supported in KoGridView.
- `App.ko.GridRow` class `display()` method now automatically picks nested relation value from nested `strFields` value, when available. That allows to traverse nested `get_str_fields()` values automatically.

  See `getDisplayValue()` method for the implementation.
- Allow to click nested elements of row cells when these are enclosed into anchors.
- Allow to override grid callback action via viewmodel `callback_action` property.
- Query filters now support multi-value `in` clause for values of drop-down `choice` filter.
- Grid viewmodel `deleted_pks` key values are processed first in `App.ko.Grid.updatePage()`. That allows to delete old row and add new row with the same `pkVal` at once (forced update).
- `App.ko.Grid` class `.setFiltersChoices()` method simplifies programmatic filtering of grid at client-side, for example from the parsed querystring.

7.4.8 plugins.js

`$.linkPreview` now has separate inclusion filter for local urls and exclusion filter for remote urls, which minimizes the possibility of preview glitches due to wrong guess of resource type.

7.5 0.4.1

Support of the `'choices' filter` option `multiple_choices: True` in non-AJAX `ListSortingView`. That allows to perform `in` field lookups for the selected field filter which was previously available only in AJAX KoGridView.

Large monolithic `views.py` split into smaller parts with symbols exported via module `__init__.py` for the convenience and compatibility.

Alternative breadcrumbs layout of field filters widgets.

7.6 0.4.2

- Compatibility to 1.10+ new-style middleware (thanks to Melvyn Sopacua).
- Fixed pagination when multiple filter field choices are selected in views.ListSortingView.

7.7 0.4.3

- Django 1.11 / Python 3.6 support.
- Selenium testing commands fixes.

7.8 0.5.0

- Reworked recursive underscore.js template processor as `App.Tpl` class.
- Grid rows, grid row actions and ForeignKeyGridWidget placeholder now are displaying Django model instances verbose field names along with their values. Related model fields verbose names are displayed as well.
- Client-side components code now uses separate html5 data attribute `data-component-class` to bind DOM subtrees to Javascript component classes (for example grids), instead of placing everything into `data-component-options` attribute as in previous versions.
- Overridable method to check whether two grid rows match the same Django model instance, suitable for RAW query grids with LEFT JOIN, which could have multiple rows with the same `pkVal` === `null`.
- Automation commands now uses `SimpleNamespace` as chained context, which allows to use different nodes for relative search queries chaining. Currently implemented are relative Selenium queries for form, component, bootstrap dialog and grid. Much better tests coverage in djk-sample project. Many new Selenium commands are implemented, including `screenshot` command.
- `ko_generic_inlineformset_factory` supports dynamic adding / removal of generic inline formsets.
- FilteredRawQuerySet / ListQuerySet queryset classes `values()` and `values_list()` methods now support model relations in queried field names via __ separator, just like usual Django querysets.
- Numerous bugfixes.

7.9 0.6.0

- `ActionsView` with `App.Actions` client-side counterpart implements AJAX viewmodels routing to create generic AJAX actions / responses. It is now used as base foundation for `App.ModelFormDialog` / `ModelFormActionsView` and with knockout datatables actions (see modelFormAction method).
- `ModelFormActionsView` with `App.ModelFormActions` client-side counterpart allows to use Django forms / inline formsets with AJAX-powered BootstrapDialog via `App.EditForm` / `App.EditInline` client-side components.
- Selective skipping of `DisplayText` field widget rendering via setting `skip_output` property in `get_text_method` callback.
- Do not bind `App.ko.Formset` to display-only `bs_inline_formsets()` generated forms with inline formsets.
- Knockout grids (datatables) `'button_footer'` built-in action type.
- djk_seed Django management command.
- `App.renderNestedList` supports rendering of `jQuery` objects values.
- `App.TabPane` supports hiding / dynamic content loading of bootstrap 3 panes.
- `App.Dialog` is now closable by default. `App.Dialog` now can be run as component.

- `html` and `replaceWith` viewmodels applies `App.initClient` hooks, also works correctly with viewmodel `.html` content that is not wrapped into top tags.
- Implemented `App.propByPath` which is now used to load Javascript object specified for `App.renderNestedList` as `options.blockTags` string. That allows to pass Javascript path string as `options.blockTags` via server-side AJAX response. `App.Dialog` class, `'alert'` / `'alert_error'` viewmodels suppports this functionality when `message` option has `object` type value.
- `App.objByPath` / `App.newClassByPath` is used by `App.Tpl` class factories.
- `App.ko.Grid.iocKoFilter_*` methods now are orthogonal thus are easier to override.
- Grid dialogs default hotkeys (`Escape`, `Enter`).
- `widgets.PrefillWidget` - field widget to prefill form input value from bootstrap 3 dropdown menu. ListQuerySet now has `prefill_choices()` method, which may provide prefill values for the form field from db field list of values.
- `.badge.btn-*` CSS classes which can be used to wrap long text in bootstrap buttons.
- Separate `admin.js` script to enable client-side of `OptionalWidget` in django admin.
- `App.ko.Grid` actions `meta` / `list` / `meta_list` first requests passing HTTP POST `firstLoad` variable to detect the initial grid datatable action at server-side in KoGridView derived class.
- Fixed selection of all current page grid datatable rows at multiple grid datatable pages.
- plugins.js: `jQuery.id()` to get multiple DOM ids, `_.moveOptions()` to move options with possible default values. `highlightListUrl` jQuery function bugfixes.
- tooltips.js: `form_error` viewmodel handler, used to display AJAX forms validation errors now has the diagnostic for missing `auto_id` values and better support for multiple error messages per field.
- contenttypes: Create content types / user groups / user permissions / Django model migration seeds. For the example of seeds, see djk_seed Django management command.
- `FormWithInlineFormsets` supports form auto_id prefix and optional customizeable form / formset constructor kwargs.
- `json_validators` module is renamed into validators, which implements generic `ViewmodelValidator` class to validate AJAX submitted form input and to return error viewmodels when needed.
- `DjkJSONEncoder` serializes lazy strings to prevent json serialization errors.
- `BaseSeleniumCommands` logs browser errors.
- tpl module reworked and expanded. Nested lists use common class `PrintList`. Implemented `json_flatatt()` and `format_html_attrs()` functions which work like built-in Django `flatatt()` and `format_html()` but automatically convert list / dict types of arguments into html attributes and / or JSON strings.
- Many bugfixes.

7.10 0.7.0

- Grids (datatables)
  - New type of action `'pagination'`.
    - There are two built-in actions of this type implemented: `'rows_per_page'` and `'switch_highlight'`.
  - Support of compound columns.

  - `glyphicon` actions are rendered in the single column of datatable, instead of each action per column.
- Static assets are moved to '/djk' subdirectory, minimizing the risk of conflicts with third party assets.
- Updated to latest versions of Knockout.js / jQuery / Bootstrap 3 (should also work with not-too-old ones).
- viewmodels AJAX response routing is rewritten as `App.ViewModelRouter` class with default instance `App.vmRouter`. It now supports binding viewmodel handlers to Javascript class instances methods.
- Optional built-in Javascript error logger.
- `App.NestedList` internally used by `App.renderNestedList` for greater flexibility of client-side Javascript nested lists rendering. `App.NestedList` now supports ordered maps via `_.ODict` instances.
- Ajax forms submitting is refactored into `App.AjaxForm` class, while setting up the ajax forms is performed by `App.AjaxForms`.
- `App.readyInstances` introduced for global client-side IoC, available in custom user scripts as well.
- Knockout.js method subscription / unsubscription is placed into `App.ko.Subscriber` mixin class.
- `focus` binding is implemented for Knockout.js.
- Request mock-up when running without web server allows reverse resolving of FQN urls in console management commands and in background celery tasks via reverseq() calls when sites framework is correctly set up.
- `ast_eval` template tag.
- Headless Chrome Selenium webdriver support.

7.11 0.8.0

- Supports both Bootstrap 4 and Bootstrap 3 via pluggable djk_ui application.
- Default rendering layouts for fields / forms / related forms / inline formsets, which can be customized by providing custom template or via inheriting from Renderer class.
- Underscore.js templates support template attributes merging and custom tags.
- Nested components and Sparse components.
- Nested serializer.

7.12 0.8.1

- Dropped Django<=1.10 support. Added Django 2.2 support.
- Dropped IE9..10 support.
- Current request `.view_title` is stored in the `.resolver_match`.
- `bs_collapse()` Jinja2 macro supports setting the initial collapse state ('out' / 'in') and Bootstrap card type.
- Implemented `App.OrderedHooks` class used to execute `App.initClientHooks` in proper order.
- `grid.js`: cleaned up init / shutdown `.applyBindings()` / `.cleanBindings()` / `.runComponent()` / `.removeComponent()` code for `App.ko.Grid` and related classes.
- `grid.js`: Implemented action `meta_list` preload.
- Refactored views classes inheritance hierarchy.
- middleware: refactored middleware classes inheritance hierarchy.

- middleware: less intrusive, better compatibility with third party modules.
- middleware: `.djk_request()``_ ``.djk_view()` methods are called only for `DJK_APPS` views by default.
- middleware : `json_response()` shortcut method.
- `RendererModelForm.has_saved_instance()` method to check whether current Django ModelForm has the bound and saved instance.
- ListQuerySet: implemented `|` + operators.
- `DjkJSONEncoder`: moved to `tpl` module. Encoding improvements.
- Refactored forms module to forms package with base / renderers / validators modules.
- HTTP response related classes / methods are moved to `http` module.

7.13 0.8.2

- bdist wheel fix.
- PyPi readme fix.

7.14 0.9.0

- `django-jinja` dependency is off by default, may be removed in the future.
- `TemplateContext` class is used to manage client-side data injection.
- Less dependency on `DJK_MIDDLEWARE`.
- Templates / selenium test improvements.

7.15 1.0.0

- Django 3.1a1 / Bootstrap 4.5 / Knockout 3.5 support.
- MultipleKeyGridWidget allows to edit many to many relationships for Django models.
- `PageContext` to inject view title / client data / client routes / custom scripts to templates via `TemplateResponse`.
- `App.renderValue` supports jQuery elements / nested arrays / objects / strings HTML rendering.
- `App.renderNestedList` supports optional unwrapping of single top DOM node.
- Improved Bootstrap popovers support with jQuery `.getPopoverTip()` / `.getVisiblePopovers()` / `.closeVisiblePopovers()` plugins.
- Support for nested components in formsets.js (empty_form) 'anonymous_template' Knockout binding.
- `UrlPath` class for automatic `re_path()` generation with positional named keyword arguments.

7.16 2.0.0

- Django 3.2 / Django 4.0 support.
- `es6 modules` support for modern browsers.
- `SystemJS` loader support for IE11 via `django_deno`.
- `terser` bundles support both for `es6 modules` and for `SystemJS` loader via `django_deno`.
- `datatables` support separate cell click actions.
- Support for `datatables` annotated fields / virtual fields via `grid_fields` dicts.
- Optional lazy registration of client-side components.
- Improved related grid view kwargs auto-detection.
- `ListRangeFilter` for `ListSortingView` range fields.

7.17 2.1.0

- Built-in `custom elements` in es5 with IE11 polyfills.
- `Bootstrap 5` compatibility.
- `ObjDict` Django model serializer with built-in field permissions check.
- `get_absolute_url` with optional user permission check.