django-is-core Documentation

Release 1.4

Luboš Mátl

Nov 20, 2018

Contents

Django-is-core is application/framework for simple development of a Information System. You will find that it is very simlar to django admin but there si several differences that justifies why we created own implementation.

CHAPTER 1

Features

- Django-is-core has same detail/add/table views as admin, but it uses REST and AJAX call to achieve it. It adds easier usage and broaden usability.
- Django-is-core can be used for creation only REST resources without UI.
- Models UI (add/detail) is more linked together. Links between foreign keys are automatically added.
- Django-is-core provides more posibilities for readonly fields. For example the fields defined only inside form can be readonly too.
- Exports to xlsx, pdf, csv can be very simply add to table view.
- Better permissions, for example link between objects is not added to UI if user does not have permission to see the object.
- Add new custom view is for django admin nightmare. With django-is-core it is very easy.
- Django-is-core views as implemented with using generic views not as method. It is cleaner and changes are simplier.
- Add new model administration without its registration.
- Better objects filters from UI (automatically respond to user typing) and coding (easier new filter implementation) perspective too.
- Token authorization.
- And much much more.

1.1 Project Home

https://github.com/matllubos/django-is-core

1.2 Documentation

https://django-is-core.readthedocs.org/en/latest

CHAPTER 2

Content

2.1 Installation

2.1.1 Requirements

Python/Django versions

Modeltranslation	Python	Django
>=1.4	3.2 - 3.4	1.8 - 1.9
	2.7	1.8 - 1.9
<=1.3	2.7	1.6 - 1.8

Libraries

- **django-class-based-auth-views** - Login/Logkout views as generic view structure
- **django-piston** - not original pistion library, but improved. You can find it here https://github.com/matllubos/django-piston
- **django-block-snippets** - library providing block snippets of html code for easier development webpages with ajax. You can find it here https://github.com/matllubos/django-block-snippets
- **django-chamber** - several helpers removing code duplication. You can find it here https://github.com/matllubos/django-chamber
- **python-dateutil** - provides powerful extensions to the datetime module available in the Python standard library
- **django-apptemplates** - Django template loader that allows you to load a template from a specific application
- **django-project-info** - small library getting project version to django context data
- **pillow** - Python imaging library (optional)
- **sorl-thumbnail** - thumbnails for django (optional)

- **germanium** - framework for testing purposes (optional)
- **factory-boy** - testing helper for creating model data for tests (optional)

All optional libraries is not instaled automatically. Other libraries are dependecies of django-is-core.

2.1.2 Using Pip

Django is core is not currently inside *PyPE* but in the future you will be able to use:

```
$ pip install django-is-core
```

Because *django-is-core* is rapidly evolving framework the best way how to install it is use source from github

```
$ pip install https://github.com/matllubos/django-is-core/tarball/{{ version }}
↪#egg=django-is-core-{{ version }}
```

2.2 Configuration

After instalation you must go throught these steps to use django-is-core:

2.2.1 Required Settings

The following variables have to be added to or edited in the project's `settings.py`:

INSTALLED_APPS

For using is-core you just add add `is_core` and `block_snippets` to `INSTALLED_APPS` variable:

```
INSTALLED_APPS = (
    ...
    'is_core',
    'block_snippets',
    ...
)
```

MIDDLEWARE_CLASSES

Next add two middlewares to end of `MIDDLEWARE_CLASSES` variable:

```
MIDDLEWARE_CLASSES = (
    ...
    'is_core.middleware.RequestKwargsMiddleware',
    'is_core.middleware.HttpExceptionsMiddleware',
)
```

2.3 Setup

To finally setup the application please follow these steps:

1. Collect static files from django-is-core with command `python manage.py collectstatic`
2. Sync database with command `python manage.py syncdb` or `python manage.py migrate`

2.4 Advanced Settings

2.4.1 Token authentification

Because django-is-core provides simple way how to create Information Systems based on REST the standard django session authentification is not ideal for this purpose.

Django-is-core provides token authentification. The advantages of this method are: 1. You can use fat client that can not use cookies. 2. Every token conains information about connected device. So you can watch user activity. 3. You can lead connected users by expiration time or deactivate user token to logout authentificated user.

If you want to use token authentification follow these steps:

INSTALLED_APPS

Add `is_core.auth_token` right after `is_core` inside `INSTALLED_APPS` variable:

```
INSTALLED_APPS = (
    ...
    'is_core',
    'is_core.auth_token',
    'block_snippets',
    ...
)
```

MIDDLEWARE_CLASSES

Replace `django.contrib.auth.middleware.AuthenticationMiddleware` with `is_core.auth_token.middleware.TokenAuthenticationMiddlewares` inside `MIDDLEWARE_CLASSES`

Setup

Finally again sync database models, because auth_token adds new django models (`python manage.py syncdb` or `python manage.py migrate`)

2.5 Cores

2.5.1 UIRESTModelISCore

The purpose of `ISCore` is to get a shared functionality of views into a one source. The `UIRESTModelISCore` class is the representation of a model in the **django-is-core** interface. These representations are stored in a file named

`cores.py` in your application. We will start with the most common case when you want to create three typical views for information system:

- table view for printing objects
- add view for creating new objects
- detail view for reading or editing objects

As example project we use Issue tracker. Firstly for every application you need management of users. We use default **Django** user model.

For creating **add**, **detail** and **table** views you must only create file `cores.py` inside specific application that contains:

```
from django.contrib.auth.models import User

from is_core.main import UIRESTModelISCore

class UserISCore(UIRESTModelISCore):
    model = User
```

There is no obligation for registration. Cores are registered automatically. The result views with preview are:

Table/List

image 1

Add

image 2

Edit

image 3

REST

But there is created REST resource too. By default on URLs `/api/user/` and `/api/user/{obj_id}` that returns object in asked format (HTTP header `Content-type: application/json`).

2.5.2 RESTModelISCore

The `RESTModelISCore` is parent of `UIRESTModelISCore`. As the name suggests this class is used only for creating REST resources without UI HTML views. The usage is the same as `UIRESTModelISCore`:

```
from django.contrib.auth.models import User

from is_core.main import RESTModelISCore

class RESTUserISCore(RESTModelISCore):
    model = User
```

2.5.3 UIModelISCore

The `UIModelISCore` is the second parent of `UIRESTModelISCore`. It is used for creating only UI views. Because UI views needs some REST resources is necessary to specify on which URL is deployed REST resource of model (api_url_name is transformed to URL by Django resolve helper):

```
from django.contrib.auth.models import User

from is_core.main import UIModelISCore

class UIUserISCore(UIModelISCore):
    model = User
    api_url_name = 'api-user'
```

You can specify URL manually:

```
class UIUserISCore(UIModelISCore):
    model = User

    def get_api_url(self, request):
        return '/api/user/'
```

2.5.4 ISCore hiearchy

Now we provide detailed description of all ISCore objects. Firstly for better understanding you can see UML class diagram of core hierarchy.

TODO add diagram

2.5.5 ISCore

Following options and methods can be applied for all Core objects like `RESTModelISCore`, `UIModelISCore` or `UIRESTModelISCore` (all descendants of ISCore class).

Options

`ISCore.`**`abstract`**

The variable `abstract` provides way how to create core that is not registered but this class variable is not inherited. Let's show an example:

```
from django.contrib.auth.models import User

from is_core.main import RESTModelISCore

class AbstractUIRESTUserISCore(RESTModelISCore):
    model = User
    abstract = True
    verbose_name = 'example of abstract user core'

class UIRESTUserISCore(AbstractUIRESTUserISCore):
    pass
```

First core is not registered. Therefore views and REST resources are not created. But the second core that inherits from the abstract core is registered. All configuration from parent class is inhered (without abstract variable).

ISCore.verbose_name, ISCore.verbose_name_plural

These variables are used inside generic views. It can be added to `context_data` and rendered inside templates.

`ISCore.`**`menu_group`**

It is necessary have a slug that distinguish one core from another. For this purpose is used variable `menu_group`. This variable is used for example to generate URL patterns or menu. Value of the variable is generated automatically for cores that is connected to model.

Methods

`ISCore.`**`init_request`**`(`*request*`)`

Every core views/REST resources calls this method before calling dispatch. You can use it to change request its calling.

`ISCore.`**`get_url_prefix`**`()`

Every core must have unique URL. Therefore a method `get_url_prefix` is way how to achieve it. Method defines URL prefix for all views and rest resources. By default the URL prefix is value of attribute menu_group.

2.5.6 ModelISCore

The next class that extends `ISCore` is `ModelISCore`. All cores that inherits from ModelISCore works as controller over a model.

Options

`ModelISCore.`**`list_actions`**

Variable `list_action` contains actions that user can perform via REST or UI. More detailed explanation with example you find inside `UIRESTModelISCore options` part.

`ModelISCore.`**`form_fields`**

Use the `form_fields` option to make simple layout changes in the forms on the **add** and **detail** and REST resources pages such as showing only a subset of available fields, modifying their order, or grouping them into rows. We will show it on `UIRESTModelISCore`. If you want to restrict form fields to `username`, `first_name` and `last_name` the simpliest way is use:

```
from django.contrib.auth.models import User

from is_core.main import UIRESTModelISCore

class UserISCore(UIRESTModelISCore):
    model = User
    form_fields = ('username', 'fist_name', 'last_name')
```

`ModelISCore.`**`form_exclude`**

This attribute, if given, should be a list of field names to exclude from the form.:

```
from django.contrib.auth.models import User

from is_core.main import UIRESTModelISCore

```

(continues on next page)

(continued from previous page)

```
class UserISCore(UIRESTModelISCore):
    model = User
    form_exclude = ('password',)
```

ModelISCore.**form_class**

If you want to change default form class which is `SmartModelForm` you can change it with this option. The form is changed for **add**, **detail** views and REST resources too.

ModelISCore.**ordering**

Option for changing default ordering of model for core.:

```
from django.contrib.auth.models import User

from is_core.main import UIRESTModelISCore

class UserISCore(UIRESTModelISCore):
    model = User
    ordering = ('last_name', 'fist_name', '-created_at')
```

Methods

ModelISCore.**get_form_fields**(*request*, *obj=None*)

Use this method to define form fields dynamically or if you want to define different form fields for **add**, **detail**, view of REST resources.

ModelISCore.**get_form_exclude**(*request*, *obj=None*)

The opposite to get_form_fields.

ModelISCore.**get_form_class**(*request*, *obj=None*)

Use this method to define form dynamically or if you want to define different form for **add**, **detail** view of REST resources.

ModelISCore.**pre_save_model**(*request*, *obj*, *form*, *change*)

Method `per_save_model` is called before saving object to database. Body is empty by default.

ModelISCore.**post_save_model**(*request*, *obj*, *form*, *change*)

Method `post_save_model` is called after saving object to database. Body is empty by default.

ModelISCore.**save_model**(*request*, *obj*, *form*, *change*)

You can rewrite this method if you want to change way how is object saved to database. Default body is:

```
def save_model(self, request, obj, form, change):
    obj.save()
```

ModelISCore.**pre_delete_model**(*request*, *obj*)

Method `pre_delete_model` is called before removing object from database. Body is empty by default.

ModelISCore.**post_delete_model**(*request*, *obj*)

Method `post_delete_model` is called after removing object from database. Body is empty by default.

ModelISCore.**delete_model**(*request*, *obj*)

You can rewrite this method if you want to change way how is object removed from database. Default body is:

```
def delete_model(self, request, obj):
    obj.delete()
```

ModelISCore.**verbose_name**(), *ModelISCore.verbose_name_plural*()

Default verbose names of ModelISCore is get from model meta options:

```
self.model._meta.verbose_name
self.model._meta.verbose_name_plural
```

ModelISCore.**menu_group**()

Default menu_group value is get from module name of model (self.model._meta.module_name)

ModelISCore.**get_ordering**(*request*)

Use this method if you want to change ordering dynamically.

ModelISCore.**get_queryset**(*request*)

Returns model queryset, ordered by defined ordering inside core. You can filter here objects according to user permissions.

ModelISCore.**preload_queryset**(*request*, *qs*)

The related objects of queryset should sometimes very slow down retrieving data from the database. If you want to improve a speed of your application use this function to create preloading of related objects.

ModelISCore.**get_list_actions**(*request*, *obj*)

Use this method if you want to change list_actions dynamically.

ModelISCore.**get_default_action**(*request*, *obj*)

Chose default action for object used inside UI and REST. For example default action is action that is performed if you select row inside table of objects. For table view default action is open **detail** view. If you return None no action is performed by default.

2.5.7 UIISCore

Options

UIISCore.**menu_url_name**

Every UI core has one place inside menu that addresses one of UI views of a core. This view is selected by option menu_url_name.

UIISCore.**show_in_menu**

Option *show_in_menu* is set to True by default. If you want to remove core view from menu set this option to False.

UIISCore.**view_classes**

Option contains view classes that are automatically added to Django urls. Use this option to add new views. Example you can see in section generic views (this is a declarative way if you want to register views dynamically see UIISCore.get_view_classes).:

```
from django.contrib.auth.models import User

from is_core.main import UIRESTModelISCore

from .views import MonthReportView


class UserISCore(UIRESTModelISCore):
    model = User

    view_classes = (
        ('reports', r'^/reports/$', MonthReportView),
    )
```

UIISCore.**default_ui_pattern_class**

Every view must have assigned is-core pattern class. This pattern is not the same patter that is used with **django** *urls*. This pattern has higher usability. You can use it to generate the url string or checking permissions. Option default_ui_pattern_class contains pattern class that is used with defined view classes. More about patterns you can find in section patterns. #TODO add link

Methods

UIISCore.**init_ui_request**(*request*)

Every view defined with option `view_classes` calls this method before calling dispatch. The default implementation of this method calls parent method `init_request`:

```
def init_ui_request(self, request):
    self.init_request(request)
```

UIISCore.**get_view_classes**()

Use this method if you want to change `view_classes` dynamically. A following example shows overriding **detail** view and registering a custom view:

```
from django.contrib.auth.models import User

from is_core.main import UIRESTModelISCore

from .views import UserDetailView, MonthReportView


class UserISCore(UIRESTModelISCore):
    model = User

    def get_view_classes(self):
        view_classes = super(UserISCore, self).get_view_classes()
        view_classes['detail'] = (r'^/(?P<pk>\d+)/$', UserDetailView)
        view_classes['reports'] = (r'^/reports/$', MonthReportView)
        return view_classes
```

UIISCore.**get_ui_patterns**()

Contains code that generates `ui_patterns` from view classes. Method returns ordered dict of pattern classes.

UIISCore.**get_show_in_menu**(*request*)

Returns `boolean` if menu link is provided for the core, by default there are three rules:

- show_in_menu must be set to `True`
- menu_url_name need not be empty
- current user must have permissions to see the linked view

`UIISCore.`**`is_active_menu_item`**(*request*, *active_group*)

This method finds if a menu link of a core is active (if the view with `menu_url_name` is the current displayed page).

`UIISCore.`**`get_menu_item`**(*request*, *active_group*)

This method returns a menu item object that contains information about the link displayed inside menu.

`UIISCore.`**`menu_url`**(*request*, *active_group*)

Returns URL string of menu item.

2.5.8 RESTISCore

`RESTISCore` is very similar to `UIISCore`, but provides REST resources instead of UI views.

Options

`RESTISCore.`**`rest_classes`**

Option contains REST classes that are automatically added to django urls. Use this option to add new REST resources. Example you can see in section REST. #TODO add link

`RESTISCore.`**`default_rest_pattern_class`**

As UI views every resource must have assigned is-core pattern class. Default pattern for REST resources is *RESTPattern*. More about patterns you can find in section patterns. #TODO add link

Methods

`RESTISCore.`**`init_rest_request`**(*request*)

Every resource defined with option `rest_classes` calls this method before calling dispatch. The default implementation of this method calls parent method `init_request`.

`RESTISCore.`**`get_rest_classes`**()

Use this method if you want to change `rest_classes` dynamically.

`RESTISCore.`**`get_rest_patterns`**()

Contains code that generates `rest_patterns` from rest classes. Method returns an ordered dict of pattern classes.

2.5.9 HomeUIISCore

`HomeISCore` contains only one UI view which is index page. By default this page is empty and contains only menu because every information system has custom index. You can very simply change default view class by changing `settings` attribute `HOME_VIEW`, the default value is:

```
HOME_VIEW = 'is_core.generic_views.HomeView'
```

You can change whole is core too by attribute `HOME_IS_CORE`, default value:

```
HOME_IS_CORE = 'is_core.main.HomeUIISCore'
```

2.5.10 UIModelISCore

`UIModelISCore` represents core that provides standard views for model creation, editation and listing. The `UIModelISCore` will not work correctly without REST resource. Therefore you must set `api_url_name` option.

Options

`UIModelISCore.`**`default_model_view_classes`**

For the `UIModelISCore` default views are **add**, **detail** and **list**:

```
default_model_view_classes = (
    ('add', r'^/add/$', AddModelFormView),
    ('detail', r'^/(?P<pk>[-\w]+)/$', DetailModelFormView),
    ('list', r'^/?$', TableView),
)
```

`UIModelISCore.`**`api_url_name`**

The `api_url_name` is required attribute. The value is pattern name of REST resource.

`UIModelISCore.`**`list_display`**

Set `list_display` to control which fields are displayed on the list page.

`UIModelISCore.`**`export_display`**

Set `export_display` to control which fields are displayed inside exports (e.g. PDF, CSV, XLSX).

`UIModelISCore.`**`export_types`**

REST resources provide the ability to export output to several formats:

- XML
- JSON
- CSV
- XLSX (you must install library XlsxWriter)
- PDF (you must install library reportlab)

List view provides export buttons. Option `export_types` contains tripple:

- title
- type
- serialization format (content-type).

For example if you want to serialize users to CSV:

```
class UIRESTUserISCore(UIRESTISCore):
    export_types = (
        ('export to csv', 'csv', 'text/csv'),
    )
```

If you want to set `export_types` for all cores you can use `EXPORT_TYPES` attribute in your settings:

```
EXPORT_TYPES = (
    ('export to csv', 'csv', 'text/csv'),
)
```

`UIModelISCore.`**`default_list_filter`**

UI table view support filtering data from REST resource. There are situations where you need to set default values for filters. For example if you want to filter only superusers you can use:

```
class UIRESTUserISCore(UIRESTISCore):
    default_list_filter = {
        'filter': {
            'is_superuser': True
        }
    }
```

On the other hand if you want to filter all users that is not superusers:

```
class UIRESTUserISCore(UIRESTISCore):
    default_list_filter = {
        'exclude': {
            'is_superuser': True
        }
    }
```

Exclude and filter can be freely combined:

```
class UIRESTUserISCore(UIRESTISCore):
    default_list_filter = {
        'filter': {
            'is_superuser': True
        },
        'exclude': {
            'email__isnull': True
        }
    }
```

`UIModelISCore.`**`form_inline_views`**

The **django-is-core** interface has the ability to edit models on the same page as a parent model. These are called inlines. We will use as example new model issue of issue tracker system:

```
class Issue(models.Model):
    name = models.CharField(max_length=100)
    watched_by = models.ManyToManyField(AUTH_USER_MODEL)
    created_by = models.ForeignKey(AUTH_USER_MODEL)
```

Now we want to add inline form view of all reported issues to user **add** and **detail** views:

```
class ReportedIssuesInlineView(TabularInlineFormView):
    model = Issue
    fk_name = 'created_by'

class UIRESTUserISCore(UIRESTISCore):
    form_inline_views = (ReportedIssuesInlineView,)
```

The `fk_name` is not required if there is only one relation between `User` and `Issue`. More about inline views you can find in generic views section # TODO add link.

UIModelISCore.**form_fieldsets**

Set form_fieldsets to control the layout of core **add** and **change** pages. Fieldset defines a list of form fields too. If you set form_fieldsets the form_fields is rewritten with a set of all fields from fieldsets. Therefore you should use only one of these attributes.

form_fieldsets is a list of two-tuples, in which each two-tuple represents a <fieldset> on the core form page. (a <fieldset> is a **section** of the form.).

The two-tuples are in the format (name, field_options), where name is a string representing the title of the form_fieldset and field_options is a dictionary of information about the fieldset, including a list of fields to be displayed in it.

As a example we will use User model again:

```
class UIRESTUserISCore(UIRESTISCore):
    form_fieldsets = (
        (None, {'fields': ('username', 'email')}),
        ('profile', {'fields': ('first_name', 'last_name'), 'classes': ('profile',)}),
    )
```

If neither form_fieldsets nor form_fields options are present, **Django** will default to displaying each field that isn't an AutoField and has editable=True, in a single fieldset, in the same order as the fields are defined in the model.

The field_options dictionary can have the following keys:

- fields

  A tuple of field names to display in this fieldset. This key is required.

  Example:

  ```
  {
      'fields': ('first_name', 'last_name'),
  }
  ```

  fields can contain values defined in form_readonly_fields to be displayed as read-only.

  If you add callable to fields its result will be displayed as read-only.

- classes

  A list or a tuple containing extra CSS classes to apply to the fieldset.

  Example:

  ```
  {
      'classes': ('profile',),
  }
  ```

- inline_view

  inline_view attribute can not be defined together with fields. This attribute is used for definig position of inline view inside form view. The value of the attribute is a string class name of the inline view.

  Example:

  ```
  {
      'inline_view': 'ReportedIssuesInlineView'
  }
  ```

UIModelISCore.**form_readonly_fields**

By default the **django-is-core** shows all fields as editable. Any fields in this option (which should be a list or a tuple) will display its data as-is and non-editable. Compare to **django-admin** fields defined in a form are used too (due SmartModelForm).

UIModelISCore.**menu_url_name**

menu_url_name is set to list by default, for all UIModelISCore and its descendants.

Methods

UIISCore.**get_form_fieldsets**(*request*, *obj=None*)

Use this method if you want to change form_fieldsets dynamically.

UIISCore.**get_form_readonly_fields**(*request*, *obj=None*)

Use this method if you want to change form_readonly_fields dynamically.

UIISCore.**get_ui_form_class**(*request*, *obj=None*)

Change this method to get a custom form only for UI. By default it uses get_ui_form_class(request, obj) method to obtain a form class.

UIISCore.**get_ui_form_fields**(*request*, *obj=None*)

Change this method to get a custom form fields only for UI. By default it uses get_form_fields(request, obj) method to obtain form fields.

UIISCore.**get_ui_form_exclude**(*request*, *obj=None*)

Change this method to get a custom form exclude fields only for UI. By default it uses get_form_exclude(request, obj) method to obtain excluded form fields.

UIISCore.**get_form_inline_views**(*request*, *obj=None*)

Use this method if you want to change form_inline_views dynamically.

UIISCore.**get_default_list_filter**(*request*)

Use this method if you want to change default_list_filter dynamically.

UIISCore.**get_list_display**(*request*)

Use this method if you want to change list_display dynamically.

UIISCore.**get_export_display**(*request*)

Method returns export_display if no export_display is set the output is result of method get_list_display(request).

UIISCore.**get_export_types**(*request*)

Use this method if you want to change export_types dynamically.

UIISCore.**get_api_url_name**(*request*)

Use this method if you want to change api_url_name dynamically.

UIISCore.**get_api_url**(*request*)

A result of this method is an URL string of REST API. The URL is generated with **Django** reverse function from api_url_name option.

UIISCore.**get_add_url**(*request*)

Returns an URL string of **add** view. Rewrite this method if you want to change a link of add button at the list view.

2.5.11 RESTModelISCore

`RESTModelISCore` represents core that provides a standard resource with default CRUD operations.

Options

`RESTModelISCore.`**`rest_detailed_fields`**

Set `rest_detailed_fields` if you want to define fields that will be returned inside REST response for a request on concrete object (an URL contains an ID of a concrete model object. For example an URL of a request is `/api/user/1/`). This option rewrites settings inside `RESTMeta` (you can find more about it at section #TODO add link).

`RESTModelISCore.`**`rest_general_fields`**

Set `rest_general_fields` if you want to define fields that will be returned inside REST response for a request on more than one object (an URL does not contain an ID of a concrete objects, eq. `/api/user/`). This defined set of fields is used for generating result of a foreign key object. This option rewrites settings inside `RESTMeta` (you can find more about it at section #TODO add link).

`RESTModelISCore.`**`rest_extra_fields`**

Use `rest_extra_fields` to define extra fields that is not returned by default, but can be extra requested by a HTTP header `X-Fields` or a GET parameter `_fields`. More info you can find in **django-piston** library documentation. This option rewrites settings inside `RESTMeta` (you can find more about it at section #TODO add link).

`RESTModelISCore.`**`rest_default_guest_fields`**

`rest_guest_fields` contains list of fields that can be seen by user that has not permission to see the whole object data. In case that a user has permission to see an object that is related with other object that can not be seen. In this situation is returned only fields defined inside `rest_guest_fields`. This option rewrites settings inside `RESTMeta` (you can find more about it at section #TODO add link).

`RESTModelISCore.`**`rest_default_detailed_fields`**

The purpose of `rest_default_detailed_fields` is the same as `rest_detailed_fields` but this option does not rewrite settings inside `RESTMeta` but the result fields is intersection of `RESTMeta` options and this option.

`RESTModelISCore.`**`rest_default_general_fields`**

The purpose of `rest_default_general_fields` is the same as `rest_general_fields` but this option does not rewrite settings inside `RESTMeta` but the result fields is intersection of `RESTMeta` options and this option.

`RESTModelISCore.`**`rest_default_extra_fields`**

The purpose of `rest_default_extra_fields` is the same as `rest_extra_fields` but this option does not rewrite settings inside `RESTMeta` but the result fields is intersection of `RESTMeta` options and this option.

`RESTModelISCore.`**`rest_default_guest_fields`**

The purpose of `rest_default_guest_fields` is the same as `rest_guest_fields` but this option does not rewrite settings inside `RESTMeta` but the result fields is intersection of `RESTMeta` options and this option.

`RESTModelISCore.`**`rest_allowed_methods`**

A default value of `rest_allowed_methods` is:

```
rest_allowed_methods = ('get', 'delete', 'post', 'put')
```

Use this option to remove a REST operation from a model REST resource. For example if you remove `post`, the REST resource will not be able to create new model object:

```
rest_allowed_methods = ('get', 'delete', 'put')
```

`RESTModelISCore.`**`rest_obj_class_names`**

This option is used with `UIIScore`. A REST resource will return a list of defined class names inside a response. The atribute inside response has named `_class_names`.

`RESTModelISCore.`**`rest_resource_class`**

A default resource class is `RESTModelResource`. You can change it with this attribute.

2.6 Views

2.7 REST

2.8 Permissions

Main permissions goal is to often check if client has access to read/update/delete views. Implementation of *django-is-core* permissions system is very similar to DRF permissions. Base permission class is:

class `is_core.auth.permissions.`**`BasePermission`**
Objects define structure of all permissions instances.

has_permission(*name*, *request*, *view*, *obj=None*)
Method must be implemented for every permission object and should return True if all requirements was fulfilled to grant access to the client. First parameter name defines name of the wanted access, request is Django request object, view is Django view or REST resource and optional parameter obj is obj related with the given request.

2.8.1 Predefined permissions

class `is_core.auth.permissions.`**`PermissionsSet`**
`PermissionSet` contains a set of permissions identified by name. Permission is granted if permission with the given name grants the access. Finally if no permission with the given name is found `False` is returned.

class `is_core.auth.permissions.`**`IsAuthenticated`**
Grants permission if user is authenticated and is active.

class `is_core.auth.permissions.`**`IsSuperuser`**
Grants permission if user is authenticated, is active and is superuser.

class `is_core.auth.permissions.`**`IsAdminUser`**
Grants permission if user is authenticated, is active and is staff.

class `is_core.auth.permissions.`**`AllowAny`**
Grants permission every time.

class `is_core.auth.permissions.`**`CoreAllowed`**
Grants permission if core (related with the view) permission selected according to the name grants the access.

class `is_core.auth.permissions.`**`CoreReadAllowed`**
Grants permission if core read permission grant access.

class is_core.auth.permissions.**CoreCreateAllowed**
Grants permission if core create permission grant access.

class is_core.auth.permissions.**CoreUpdateAllowed**
Grants permission if core update permission grant access.

class is_core.auth.permissions.**CoreDeleteAllowed**
Grants permission if core delete permission grant access.

class is_core.auth.permissions.**AndPermission**
`AndPermission` is only helper for joining more permissions with `AND` operator. `AndPermission` init method accepts any number of permission instances and returns `True` if every inner permission returns `True`:

```
AndPermission(IsAdminUser(), IsSuperuser(), IsAuthenticated())
```

Because this style of implementation is badly readable you can join permissions `&`, the result will be the same:

```
IsAdminUser() & IsSuperuser() & IsAuthenticated()
```

class is_core.auth.permissions.**OrPermission**
`OrPermission` is same as `AndPermission` but permissions are joined with `OR` operator. `OrPermission` returns `True` if any inner permission returns `True`. Again you can use joining with `|` operator:

```
OrPermission(IsAdminUser(), IsSuperuser(), IsAuthenticated())
IsAdminUser() | IsSuperuser() | IsAuthenticated()
```

Whole three permissions

2.8.2 Custom permission

If you want to implement custom permission, you only must create subclass of `is_core.auth.permissions.BasePermission` and implement `has_permission` method.

2.8.3 Core permissions

As an example of how to define core permissions we use model core of User object:

```
from django.contrib.auth.models import User

from is_core.auth.permissions import IsSuperuser
from is_core.main import UIRESTModelISCore

class UserISCore(UIRESTModelISCore):

    model = User
    permission = IsSuperuser()
```

Now only a superuser has access to the User core. But this solution is a little bit dangerous, because there is no validated permission name and we only want create,read, update and delete permission names. Better solution is to use `is_core.auth.permissions.PermissionsSet`:

```
from django.contrib.auth.models import User

from is_core.auth.permissions import PermissionsSet, IsSuperuser
from is_core.main import UIRESTModelISCore
```

(continues on next page)

(continued from previous page)

```
class UserISCore(UIRESTModelISCore):

    model = User
    permission = PermissionsSet(
        add=IsSuperuser(),
        read=IsSuperuser(),
        update=IsSuperuser(),
        delete=IsSuperuser(),
    )
```

Because writing too much code can lead to typos you can use `default_permission` attribute from which is automatically generated `permission` the result will be same as in previous example:

```
from django.contrib.auth.models import User

from is_core.auth.permissions import IsSuperuser
from is_core.main import UIRESTModelISCore

class UserISCore(UIRESTModelISCore):

    model = User
    default_permission = IsSuperuser()
```

But if you want to disable for example deleting model instances the delete permission will not be added to the permission set:

```
from django.contrib.auth.models import User

from is_core.auth.permissions import IsSuperuser
from is_core.main import UIRESTModelISCore

class UserISCore(UIRESTModelISCore):

    model = User
    default_permission = IsSuperuser()
    can_delete = False
```

the attribute permission will be now:

```
permission = PermissionsSet(
    add=IsSuperuser(),
    read=IsSuperuser(),
    update=IsSuperuser(),
)
```

You can use operator joining for using more permission types:

```
from django.contrib.auth.models import User

from is_core.auth.permissions import IsSuperuser, IsAdminUser
from is_core.main import UIRESTModelISCore

class UserISCore(UIRESTModelISCore):

    model = User
    default_permission = IsSuperuser() & IsAdminUser()
```

For some cases is necessary update permissions in a class mixin for this purpose you can use method ‘‘_init_permission(permission)‘:

```
from is_core.auth.permissions import IsSuperuser, IsAdminUser
from is_core.main import UIRESTModelISCore


class HistoryISCoreMixin:

    def _init_permission(self, permission):
        permission = super()._init_permission(permission)
        permission.set('history', IsSuperuser())
        return permission

class UserISCore(UIRESTModelISCore):

    model = User
    permission = PermissionsSet(
        add=IsAdminUser(),
        read=IsAdminUser(),
        update=IsAdminUser(),
        delete=IsAdminUser(),
    )
```

2.8.4 View permissions

View permissions are used in the same way as core permissions:

```
from is_core.auth.permissions import IsSuperuser
from is_core.generic_views.form_views import ReadonlyDetailModelFormView

class UserReadonlyDetailModelFormView(ReadonlyDetailModelFormView):

    permission = IsSuperuser()
```

Again you can set permissions according to names. For view permissions the names are HTTP method names:

```
from is_core.auth.permissions import PermissionsSet, IsSuperuser
from is_core.generic_views.form_views import DetailModelFormView

class UserDetailModelFormView(DetailModelFormView):

    permission = PermissionsSet(
        post=IsSuperuser(),
        get=IsSuperuser()
    )
```

By default core views get access permissions from core. For example detail view permissions are set this way:

```
from is_core.auth.permissions import PermissionsSet, CoreReadAllowed,␣
↪CoreUpdateAllowed
from is_core.generic_views.form_views import DetailModelFormView

class UserDetailModelFormView(DetailModelFormView):

    permission = PermissionsSet(
```

(continues on next page)

(continued from previous page)

```
        post=CoreUpdateAllowed(),
        get=CoreReadAllowed()
    )
```

If you want to have edit view accessible only if user is allowed to modify an object in core permissions. You can use very similar implementation:

```
from is_core.auth.permissions import PermissionsSet, CoreUpdateAllowed
from is_core.generic_views.form_views import DetailModelFormView

class UserDetailModelFormView(DetailModelFormView):

    permission = PermissionsSet(
        post=CoreUpdateAllowed(),
        get=CoreUpdateAllowed()
    )
```

2.8.5 REST permissions

For the REST classes permissions you can use the same rules. The only difference is that there are more types of permissions because REST resource fulfills two functions - serializer and view (HTTP requests):

```
from is_core.rest.resource import RESTObjectPermissionsMixin

class RESTModelCoreResourcePermissionsMixin(RESTObjectPermissionsMixin):

    permission = PermissionsSet(
        # HTTP permissions
        head=CoreReadAllowed(),
        options=CoreReadAllowed(),
        post=CoreCreateAllowed(),
        get=CoreReadAllowed(),
        put=CoreUpdateAllowed(),
        patch=CoreUpdateAllowed(),
        delete=CoreDeleteAllowed(),

        # Serializer permissions
        create_obj=CoreCreateAllowed(),
        read_obj=CoreReadAllowed(),
        update_obj=CoreUpdateAllowed(),
        delete_obj=CoreDeleteAllowed()
    )
```

2.8.6 Check permissions

View/resource

If you want to check your custom permission in view or REST resource you can use method `has_permission(name, obj=None)` as an example we can use method `is_readonly` in th form view (form is readonly only if post permission returns `False`):

```
def is_readonly(self):
    return not self.has_permission('post')
```

Because some permissions require obj parameter all views that inherit from `is_core.generic_views.mixins.GetCoreObjViewMixin` has automatically added objects to the permission check.

Core

Sometimes you need to check permission in the core. But there is no view instance and you will have to create it. For better usability you can check permissions via view patterns, as an example we can use method `get_list_actions` which return edit action only if user has permission to update an object:

```
def get_list_actions(self, request, obj):
    list_actions = super(UIRESTModelISCore, self).get_list_actions(request, obj)
    detail_pattern = self.ui_patterns.get('detail')
    if detail_pattern and detail_pattern.has_permission('get', request, obj=obj):
        return [
            WebAction(
                'detail-{}'.format(self.get_menu_group_pattern_name()), _('Detail'),
                'edit' if detail_pattern.has_permission('post', request, obj=obj)␣
↪else 'detail'
            )
        ] + list(list_actions)
    else:
        return list_actions
```

Pattern method `has_permission(name, request, obj=None, **view_kwargs)` can be used with more ways. By default is `view_kwargs` get from request kwargs. If you can change it you can use method kwargs parameters. Parameter `obj` can be used for save system performance because object needn't be loaded from database again:

```
detail_pattern = self.ui_patterns.get('detail')
detail_pattern.has_permission('get', request)  # object id is get from request.kwargs
detail_pattern.has_permission('get', request, id=obj.pk)  # request.kwargs "id" is␣
↪overridden with obj.pk
detail_pattern.has_permission('get', request, obj=obj)  # saves db queryes because␣
↪object needn't be loaded from database
```

2.9 Models

2.10 Forms

2.11 Filters

Filters are documented inside django-pyston. Is core uses Pyston filters to generate list view. Every column can contain form input which accepts inputu data. There is several options how is django widget which is used for rendering HTML of filter input:

2.11.1 UIFilterMixin

Django-is-core provedes special mixin for filters that adds posibility to change rendered widget inside filter class, example exclude `ForeignKeyFilter` with posibility to restrict field queryset choices:

```
class RestrictedFkFilter(UIFilterMixin, ForeignKeyFilter):

    def get_restricted_queryset(self, qs, request):
        # There can be foreign key queryset restricted
        return qs

    def get_widget(self, request):
        formfield = self.field.formfield()
        formfield.queryset = self.get_restricted_queryset(formfield.queryset, request)
        return formfield.widget
```

2.11.2 Field filter

There is two possibilities. If filter has set choices attribute, filter is always select box with filter choices. If not filter is obtained from model field by using method `formfield`.

2.11.3 Method/Resource filter

There is applied same rule as for field filter, but if choices is not defined is returned simple text input.

2.12 Auth

2.13 Utils

render_model_object_with_link(*request*, *obj*, *display_value=None*)

Returns clickable text representation of a model object, which leads to its detail. It can be for example used in model's property to display clickable name of a related object in detail view. Set `display_value` to override model's default text representation.

is_core.utils.**get_link_or_none**()

Helper that generates URL from pattern name and kwargs and checks if current request has permission to open the URL, if permission is not granted None is returned.

Index

L

M

O

P

R

S

V