Django Extras Documentation

Release 0.2.7.b1

Tim Savage

September 22, 2017

Contents

CHAPTER 1

Django Extras documentation

Project Status

This is a fairly new project, while this package is utilised in production on several projects the status will remain as beta until unit-test coverage has been expanded. Documentation is now fairly complete, in general areas of the code base that have not been documented should be considered unstable.

Getting help

- Report bugs with Django Extras with the issue tracker.

What's in Django Extras

Django Extras is a project that provides extensions for Django to solve common development situations not (or not yet) covered by the core Django framework.

Examples of this include:

- additional decorators
- model mixins to easily assign owners to a model
- additional model and form fields
- greatly expanded collection of default response classes

See *API Reference* full reference.

First steps

If you are new to Django it is recommended you visit the Django documentation as they have excellent documentation to get you up and running.

Other batteries included

- *Authentication*

Source code

Full source code is available on Git Hub. Have migrated away from BitBucket for Travis CI support.

CHAPTER 2

API Reference

`django_extras.contrib.auth`

See *User authentication extensions*.

Forms

Form API reference.

Form field reference

Field types

`ColorField`

class `ColorField`(*[allow_alpha=False, max_length=40, **options]*)

A `CharField` that checks that the value is a valid CSS color value. `allow_alpha` controls if colors can support alpha values.

Specifies that the default widget is the `JQueryColorPicker`.

`JsonField`

class `JsonField`(*[dump_options={'cls': DjangoJSONEncoder}, load_options={}, **options]*)

A `TextField` that handles serialisation/deserialization of JSON structures for display in a Text input.

Form widget reference

Widget types

HTML5 Input types

Html5EmailInput

class Html5EmailInput
HTML 5 email input.

Html5NumberInput

class Html5NumberInput
HTML 5 number input.

Html5DateInput

class Html5DateInput
HTML 5 date input.

Html5DateTimeInput

class Html5DateTimeInput
HTML 5 datetime input.

Html5TimeInput

class Html5TimeInput
HTML 5 time input.

JQuery enhanced widgets

JQueryColorPicker

class JQueryColorPicker
Widget that displays a color picker. This widget assumes jQuery has been included on the page. JavaScript and CSS is included and is defined using Django media definitions.

Middleware

Timing Middleware

class TimingMiddleware
Adds a header to all responses with the total time spent generating a response. This middleware component should be added as early as possible in the list of middleware classes to get best results.

The name of the header returned to the browser is `X-PROCESSING_TIME_MS`, time is in milliseconds.

Models

Model API reference.

Model field reference

Note: All fields registered with south (if used by your project).

Field types

ColorField

class ColorField([*allow_alpha=False*, *max_length=40*, ***options*])

A `CharField` that checks that the value is a valid CSS color value. `allow_alpha` controls if colors can support alpha values.

MoneyField

class MoneyField([*max_digits=40*, *decimal_places=4*, ***options*])

A `DecimalField` that sets up sensible defaults for monetary values, in addition the *`MoneyField`* will return values as instances of the `Money` type. The `Money` type is based on Pythons decimal object.

Note: The current implementation does not store the currency code with the money value.

PercentField

class PercentField(***options*)

A `FloatField` that represents a percentage value. Validates provided value to ensure it is within the range 0 to 100.

LatitudeField

class LatitudeField(***options*)

A `FloatField` that represents a latitude. Validates provided value to ensure it is within the range -90.0 to 90.0

LongitudeField

class LongitudeField(***options*)

A `FloatField` that represents a longitude. Values are validated to be within the range -180.0 to 180.0.

JsonField

class JsonField([*dump_options={'cls': DjangoJSONEncoder}*, *load_options={}*, ***options*])
A `TextField` that handles serialisation/deserialization of JSON structures into a database field.

Response objects

HttpResponse subclasses

Django extras includes a number of additional `HttpResponse` subclasses that handle different types of HTTP responses. These subclasses are defined in *`django_extras.http`*.

Common HTTP response types

Common response codes, most of these codes are defined in the W3C Protocol specification. This collection also includes status codes that are defined for Web DAV. This list does not include response types that are already defined in Django.

Successful 2xx responses

class HttpResponseCreated
Status code 201

class HttpResponseAccepted
Status code 202

class HttpResponseNonAuthoritative
Status code 203

class HttpResponseNoContent
Status code 204

class HttpResponseResetContent
Status code 205

class HttpResponsePartialContent
Status code 206

Redirection 3xx responses

All responses within the 3xx class inherit from `HttpResponseRedirect`.

class HttpResponseSeeOther
Status code 303

Client Error 4xx responses

class HttpResponseUnAuthorised
Status code 401

class HttpResponsePaymentRequired
Status code 402

class HttpResponseNotAcceptable
Status code 406

class HttpResponseRequestTimeout
Status code 408

class HttpResponseConflict
Status code 409

class HttpResponseLengthRequired
Status code 411

class HttpResponsePreconditionFailed
Status code 412

class HttpResponseRequestEntityTooLarge
Status code 413

class HttpResponseUnsupportedMediaType
Status code 415

class HttpResponseExpectationFailed
Status code 417

class HttpResponseUnprocessableEntity
Status code 422

class HttpResponseLocked
Status code 423

class HttpResponseFailedDependency
Status code 424

class HttpResponseUpgradeRequired
Status code 426

Server Error 5xx responses

class HttpResponseNotImplemented
Status code 501

class HttpResponseBadGateway
Status code 502

class HttpResponseServiceUnavailable
Status code 503

class HttpResponseGatewayTimeout
Status code 504

class HttpResponseInsufficientStorage
status_code = 507

Enhanced response types

class FileResponse
The constructor accepts the same `content` property as the default :class:`HttpResponse` class except it is

interpreted as a file name or file handle and `content_type`. The response object facilitates streaming the content of the file to the client. There is an optional parameter `include_last_modified` which defaults to *True* that supplies the last modified date of the specified file as an HTTP header.

class JsonResponse
Acts just like :class:`HttpResponse` except will encode the first parameter to JSON (using :class:`DjangoJSONEncoder`) and changes the default `content_type` to *application/json*.

Validators

Additional validators

The *`django_extras.core.validators`* module contains a collection of callable validators for use with model and form fields. They're used internally but are available for use with your own fields, too. They can be used in addition to, or in lieu of custom `field.clean()` methods.

validate_color

validate_color
A `RegexValidator` instance that ensures a value looks like a CSS color value.

validate_alpha_color

validate_alpha_color
A `RegexValidator` instance that ensures a value looks like a CSS color value. Supports color definitions with alpha blending.

validate_json

validate_json
A `JsonValidator` instance that ensures a value is valid JSON.

CHAPTER 3

User authentication extensions

While Django comes with a built in user authentication system it leaves a couple of common use-cases incomplete. Django Extras fills in the missing pieces.

Installation

No installation is required, this extension does not require additional database models.

Convenience decorators

The staff_required decorator

staff_required([*include_superusers=True*, *login_url=None*, *raise_exception=False*])

This decorator provides a simple way of restricting access to a particular view to users who have the *is_staff* flag set. Rather than using the *~django.contrib.auth.decorators.user_passes_test()* decorator restricting a view to staff can be written as:

```
from django_extras.contrib.auth.decorators import staff_required
@staff_required
def my_view(request):
    ...
```

By default this decorator also includes users with the *is_superuser* flag set, these users can be excluded with the optional `include_superuser` parameter. Example:

```
from django_extras.contrib.auth.decorators import staff_required
@staff_required(include_superusers=False)
def my_view(request):
    ...
```

As in the `login_required()`, `login_url` defaults to `settings.LOGIN_URL`.

Mirroring the change in Django 1.4 this decorator also accepts the `raise_exception` parameter. If given, the decorator will raise `PermissionDenied`.

The superuser_required decorator

superuser_required(*[login_url=None, raise_exception=False]*)

This decorator provides a simple way of restricting access to a particular view to users who have the *is_superuser* flag set. Rather than using the *~django.contrib.auth.decorators.user_passes_test()* decorator restricting a view to super users can be written as:

```
from django_extras.contrib.auth.decorators import superuser_required
@superuser_required
def my_view(request):
    ...
```

As in the `login_required()`, `login_url` defaults to `settings.LOGIN_URL`.

Mirroring the change in Django 1.4 this decorator also accepts the `raise_exception` parameter. If given, the decorator will raise `PermissionDenied`.

Ownership Mixin Models

Two mixin classes are provided that provide a consistent API for assigning ownership of a model instance to a user.

Example:

```
class MyModel(SingleOwnerMixin, models.Model):
    name = models.CharField(max_length=50)
```

Many methods include *include_staff* and *include_superuser* parameters, these are used to treat staff and superuser users as if they are owners of the instance.

Both *SingleOwnerMixin* and *MultipleOwnerMixin* provide the following methods.

class OwnerMixinBase

OwnerMixinBase.**owned_by**(*user*[, *include_staff=False*, *include_superuser=False*])

Returns a boolean value to indicate if the supplied user is a valid owner of a model instance.

OwnerMixinBase.**owner_list**()

Returns a list of `User` models that are owners of the model instance. A list is returned by both single and multiple versions of the mixin.

SingleOwnerMixin

class SingleOwnerMixin

This class provides a simple way to assign ownership of a model instance to a single user.

owner

`User` object as provided by a `ForeignKey` model field.

Note: By default the `related_name` parameter is set to: `'%(app_label)s_%(class)s_owner'`

MultipleOwnerMixin

class MultipleOwnerMixin
This class provides a simple way to assign ownership of a model instance to multiple users.

owners
`RelatedManager` object as provided by a `ManyToManyField` model field.

Note: By default the `related_name` parameter is set to: `'%(app_label)s_%(class)s_owners'`

OwnerMixinManager

class OwnerMixinManager
Manager class used by *SingleOwnerMixin* and *MultipleOwnerMixin* to obtain the instances of a model that has ownership assigned to a particular user.

Fetching owned instances

`OwnerMixinManager.`**owned_by**(*user*[, *include_staff=False*, *include_superuser=False*])

Returns a `QuerySet` filtered by a user or users. The user parameter can be either a single `User` object or primary key, a sequence of `User` objects or primary keys.

Example:

```
# Single user
>>> MyModel.objects.owned_by(request.user)
[<MyModel: Foo>, <MyModel: Bar>]

# Multiple primary keys
>>> MyModel.objects.owned_by([1, 2, 3])
[<MyModel: Foo>, <MyModel: Bar>, <MyModel: Eek>]
```

Note: It is not possible to use the *include_staff* and *include_superuser* parameters when passing a sequence for the *user* parameter. A `TypeError` exception will be raised in this case.

Shortcuts

get_owned_object_or_40x([*klass*, *owner*, *include_staff=False*, *include_superuser=False*, **args*, ***kwargs*])

A convenience method that mirrors the Django shortcut *get_object_or_404*. If the object cannot be loaded a `Http404` exception is raised, if a user cannot be verified as an owner a `PermissionDenied` exception is raised.

As with the other extensions *include_staff* and *include_superuser* flags are provided. **args* and ***kwargs* work in the same way as *get_object_or_404*.

Python Module Index

d

Index

J

L

M

O

P

S

T

V