django-fusionbox Documentation

Release 0.0.1

Fusionbox Programmers

February 14, 2017

Contents

Contents:

CHAPTER 1

Introduction

Django app of useful, reusable tools for use in Django projects. Primarily used by the development team at Fusionbox, however contains many tools that any Django developer would find useful.

CHAPTER 2

Auth

Authentication backend to allow you to insert your own user model into the built in django authentication backend.

2.1 Installation and Usage

- Add `'fusionbox.auth.backends.CustomModelBackend'` to your `AUTHENTICATION_BACKENDS` setting in `settings.py`
- Set `CUSTOM_USER_MODEL` in your `settings.py` file to a fully qualified path to your custom user model.

CHAPTER 3

The Middlewares

class fusionbox.middleware.**GenericTemplateFinderMiddleware**
Response middleware that uses *generic_template_finder_view()* to attempt to autolocate a template for otherwise 404 responses.

process_response(*request*, *response*)
Ensures that 404 raised from view functions are not caught by GenericTemplateFinderMiddleware.

process_view(*request*, *view_func*, *view_args*, *view_kwargs*)
Informs *process_response()* that there was a view for this url and that it threw a real 404.

class fusionbox.middleware.**Redirect**(*source*, *target*, *status_code*, *filename*, *line_number*)
Encapulates all of the information about a redirect.

class fusionbox.middleware.**RedirectFallbackMiddleware**(**args*, ***kwargs*)
This middleware handles 3xx redirects and 410s.

Only 404 responses will be redirected, so if something else is returning a non 404 error, this middleware will not produce a redirect

Redirects should be formatted in CSV files located in either <project_path>/redirects/ or an absolute path declared in settings.REDIRECTS_DIRECTORY.

CSV files should not contain any headers, and be in the format source_url, target_url, status_code where status_code is optional and defaults to 301. To issue a 410, leave off target url and status code.

fusionbox.middleware.**generic_template_finder_view**(*request*, **args*, ***kwargs*)
Find a template based on the request url and render it.

- / -> index.html
- /foo/ -> foo.html OR foo/index.html

fusionbox.middleware.**preprocess_redirects**(*lines*, *raise_errors=True*)
Takes a list of dictionaries read from the csv redirect files, creates Redirect objects from them, and validates the redirects, returning a dictionary of Redirect objects.

CHAPTER 4

Template Tags

The `fusionbox_tags` library contains several useful template tags. Use `{% load fusionbox_tags %}` to use these.

4.1 highlight_here

Filter the subnode's output to add a class to every anchor where appropriate, based on startswith matching. By default the class is `here`, but you can override by passing an argument to the tag.

Note: This requires BeautifulSoup to do the HTML parsing

4.2 Examples

Given:

```
{% highlight_here %}
    <a href="/" class="home">/</a>
    <a href="/blog/">blog</a>
{% endhighlight %}
```

If request.url is `/`, the output is:

```
<a href="/" class="home here">/</a>
<a href="/blog/">blog</a>
```

On `/blog/`, it is:

```
<a href="/" class="home">/</a>
<a href="/blog/" class="here">blog</a>
```

4.3 Code

CHAPTER 5

Behaviors

5.1 Usage Notes

Because behaviors are ultimately implemented using a special metaclass, any model inheritance involving behaviors must come before any other parent class which inherits from Django's built in metaclass.

Example Correct and Incorrect Usage:

```
class MyBaseModel(models.Model):
    pass

# Incorrect Usage
class MyChildModel(MyBaseModel, Timestampable):
    pass

# Correct Usage
class MyChildModel(Timestampable, MyBaseModel):
    pass
```

5.2 Built in Behaviors

class `fusionbox.behaviors.`**`Behavior`**(**args*, ***kwargs*)

Base class for all Behaviors

Behaviors are implemented through model inheritance, and support multi-inheritance as well. Each behavior adds a set of default fields and/or methods to the model. Field names can be customized like example B.

EXAMPLE A:

```
    class MyModel(FooBehavior):
        pass
```

`MyModel` will have whatever fields `FooBehavior` adds with default field names.

EXAMPLE B:

```
    class MyModel(FooBehavior):
        class FooBehavior:
            bar = 'qux'
            baz = 'quux'
```

MyModel will have the fields from FooBehavior added, but the field names will be 'qux' and 'quux' respectively.

EXAMPLE C:

```
class MyModel(FooBehavior, BarBehavior):
    pass
```

MyModel will have the fields from both FooBehavior and BarBehavior, each with default field names. To customizing field names can be done just like it was in example B.

classmethod merge_parent_settings()
Every behavior's settings are stored in an inner class whose name matches its behavior's name. This method implements inheritance for those inner classes.

classmethod modify_schema()
Hook for behaviors to modify their model class just after it's created

fusionbox.behaviors.**ManagedQuerySet**
alias of *QuerySetManagerModel*

class fusionbox.behaviors.**MetaBehavior**
Base Metaclass for Behaviors

class fusionbox.behaviors.**Publishable**(**args*, ***kwargs*)
Base class for adding publishable behavior to a model.

Added Fields:

Field 1: field: DateTimeField(default=datetime.datetime.now, help_text='Selecting a future date will automatically publish to the live site on that date.') description: The date that the model instance will be made available to the PublishableManager's query set default_name: publish_at

Field 2: field: DateTimeField(default=datetime.datetime.now, help_text='Selecting a future date will automatically publish to the live site on that date.') description: setting to False will automatically draft the instance, making it unavailable to the PublishableManager's query set default_name: is_published

Added Managers:

PublishableManager: description: overwritten get_queryset() function to only fetch published instances. name: published usage:

class Blog(Publishable): ...

all_blogs = Blog.objects.all() published_blogs = Blog.published.all()

class fusionbox.behaviors.**PublishableManager**
Manager for publishable behavior

class fusionbox.behaviors.**QuerySetManagerModel**(**args*, ***kwargs*)
This behavior is meant to be used in conjunction with *fusionbox.db.models.QuerySetManager*

A class which inherits from this class will any inner QuerySet classes found in the *mro* merged into a single class.

Given the following Parent class:

```
class Parent(models.Model):
    class QuerySet(QuerySet):
        def get_active(self):
            ...
```

The following two Child classes are equivalent:

```
    class Child(Parent):
        class QuerySet(Parent.QuerySet):
            def get_inactive(self):
                ...

    class Child(QuerySetManagerModel, Parent):
        class QuerySet(QuerySet):
            def get_inactive(self):
                ...
```

classmethod merge_parent_settings()
Automatically merges all parent QuerySet classes to preserve custom defined QuerySet methods

class fusionbox.behaviors.**SEO**(**args*, ***kwargs*)
Base class for adding seo behavior to a model.

Added Fields:

Field 1: field: CharField(max_length = 255) description: Char field intended for use in html <title> tag. validation: Max Length 255 Characters default_name: seo_title

Field 2: field: TextField() description: Text field intended for use in html <meta name='description'> tag. default_name: seo_description

Field 3: field: TextField() description: Text field intended for use in html <meta name='keywords'> tag. validation: comma separated text strings default_name: seo_keywords

formatted_seo_data(*title=''*, *description=''*, *keywords=''*)
A string containing the model's SEO data marked up and ready for output in HTML.

class fusionbox.behaviors.**Timestampable**(**args*, ***kwargs*)
Base class for adding timestamping behavior to a model.

Added Fields:

Field 1: field: DateTimeField(default=now) description: Timestamps set at the creation of the instance default_name: created_at

Field 2: field: DateTimeField(auto_now=True) description: Timestamps set each time the save method is called on the instance default_name: updated_at

class fusionbox.behaviors.**Validation**(**args*, ***kwargs*)
Base class for adding complex validation behavior to a model.

By inheriting from Validation, your model can have `validate` and `validate_<field>` methods.

`validate()` is for generic validations, and for `NON_FIELD_ERRORS`, errors that do not belong to any one field. In this method you can raise a ValidationError that contains a single error message, a list of errors, or - if the messages **are** associated with a field - a dictionary of field-names to message-list.

You can also write `validate_<field>` methods for any columns that need custom validation. This is for convience, since it is easier and more intuitive to raise an 'invalid value' from within one of these methods, and have it automatically associated with the correct field.

Even if you don't implement custom validation methods, Validation changes the normal behavior of `save` so that validation **always** occurs. This makes it easy to write APIs without having to understand the `clean`, `full_clean`, and *`clean_fields()`* methods that must called in django. If a validation error occurs, the exception will **not** be caught, it is up to you to catch it in your view or API.

clean_fields(*exclude=None*)
Must be manually called in Django.

Calls any `validate_<field>` methods defined on the Model.

is_valid()
Returns `True` or `False`

validation_errors()
Returns a dictionary of errors.

fusionbox.behaviors.**now**()
[tz] -> new datetime with tz's local day and time.

class fusionbox.behaviors.**MetaBehavior**
Base Metaclass for Behaviors

CHAPTER 6

Panels

6.1 Intro

Django Debug Toolbar allows hooking in extra panels.

6.2 User Panel

The user panel is derived from https://github.com/playfire/django-debug-toolbar-user-panel. The user panel allows for quick swapping between users. This version has been modified to disable the panel unless the user logs in initially with a superuser.

6.2.1 Installation

- Add `'fusionbox.panels.user_panel'` to your `INSTALLED_APPS`:

```
INSTALLED_APPS = (
    ...

    'fusionbox.panels.user_panel',

    ...
)
```

- Add `fusionbox.panels.user_panel.panels.UserPanel` to the `DEBUG_TOOLBAR_PANELS` setting:

```
DEBUG_TOOLBAR_PANELS = (
    ...

    'fusionbox.panels.user_panel.panels.UserPanel',

    ...
)
```

- Include `fusionbox.panels.user_panel.urls` somewhere in your url conf:

```
urlpatterns = patterns('',
    ...

    url(r'', include('fusionbox.panels.user_panel.urls')),
```

```
        ...
    )
```

CHAPTER 7

Email

Markdown-templated email.

An email template looks like this:

```
---
subject: Hello, {{user.first_name}}
---
Welcome to the site.
```

When using `send_markdown_mail()`, its output is placed in a layout to produce a full html document::

```
<!DOCTYPE html>
<html>
    <body>
        {{content}}
    </body>
</html>
```

The default layout is specified in `settings.EMAIL_LAYOUT`, but can be overridden on a per-email basis.

`fusionbox.mail.`**`create_markdown_mail`**(*template*, *context*, *to=None*, *subject=None*, *from_address='settings.SERVER_EMAIL'*, *layout='settings.EMAIL_LAYOUT'*)

Creates a message from a markdown template and returns it as an `EmailMultiAlternatives` object.

`fusionbox.mail.`**`send_markdown_mail`**(**args*, ***kwargs*)

Wrapper around *`create_markdown_mail()`* that creates and sends the message in one step.

`fusionbox.mail.`**`render_template`**(*template*, *context*, *layout*)

With a markdown template, a context to render it in, and a layout to generate html from it, a 3-tuple of (metadata, markdown content, html content) is returned.

`fusionbox.mail.`**`extract_frontmatter`**(*str*)

Given a string with YAML style front matter, returns the parsed header and the rest of the string in a 2-tuple.

```
    >>> extract_frontmatter('---\nfoo: bar\n---\nbody content\n')
    ({'foo': 'bar'}, 'body content\n')
```

CHAPTER 8

Forms Module

8.1 Forms

class fusionbox.forms.**BaseChangeListForm**(**args*, ***kwargs*)
Base class for all ChangeListForms.

get_queryset()
If the form was initialized with a queryset, this method returns that queryset. Otherwise it returns Model.objects.all() for whatever model was defined for the form.

class fusionbox.forms.**SearchForm**(**args*, ***kwargs*)
Base form class for implementing searching on a model.

Example Usage

```
class UserSearchForm(SearchForm):
    SEARCH_FIELDS = ('username', 'email', 'profile__role')
    model = User
```

```
>>> form = UserSearchForm(request.GET, queryset=User.objects.filter(is_active=True))
>>> form
<accounts.forms.UserSearchForm object at 0x102ea18d0>
>>> form.get_queryset()
[<User: admin>, <User: test@test.com>, <User: test2@test.com>]
```

SEARCH_FIELDS should be an iterable of valid django queryset field lookups. Lookups can span foreign key relationships.

By default, searches will be case insensitive. Set CASE_SENSITIVE to True to make searches case sensitive.

get_queryset()
Constructs an '__contains' or '__icontains' filter across all of the fields listed in SEARCH_FIELDS.

post_search(*qs*)
Hook for modifying the queryset after the search. Will not be called on an invalid form.

Runs only if the form validates.

pre_search(*qs*)
Hook for modifying the queryset prior to the search

Runs prior to any searching and is run regardless form validation.

class fusionbox.forms.**FilterForm**(**args*, ***kwargs*)
Base class for implementing filtering on a model.

Example Usage

```
class UserFilterForm(FilterForm):
    FILTERS = {
        'active': 'is_active',
        'date_joined': 'date_joined__gte',
        'published': None, # Custom filtering
        }
    model = User

    PUBLISHED_CHOICES = (
            ('', 'All'),
            ('before', 'Before Today'),
            ('after', 'After Today'),
            )

    active = forms.BooleanField(required=False)
    date_joined = forms.DateTimeField(required=False)
    published = forms.ChoiceField(choices=PUBLISHED_CHOICES, widget=forms.HiddenInput())

    def pre_filter(self, queryset):
        published = self.cleaned_data.get('published')
        if published == '':
            return queryset
        elif published == 'before':
            return queryset.filter(published_at__lte=datetime.datetime.now())
        elif published == 'after':
            return queryset.filter(published_at__gte=datetime.datetime.now())
```

`FILTERS` defines a mapping of form fields to queryset filters.

When displaying in the template, this form also provides you with url querystrings for all of your filters.

`form.filters` is a dictionary of all of the filters defined on your form.

In the example above, you could do the following in the template for display links for the published filter

```
{% for choice in form.filters.published %}
    {% if choice.active %}
        {{ choice.display }} (<a href='?{{ choice.remove }}'>remove</a>)
    {% else %}
        <a href='?{{ choice.querystring }}'>{{ choice.display }}</a>
    {% endif %}
{% endfor %}
```

filters
: Generates a dictionary of filters with proper queryset links to maintian multiple filters.

get_queryset()

Performs the following steps:

- Returns the queryset if the form is invalid.
- Otherwise, filters the queryset based on the filters defined on the form.
- Returns the filtered queryset.

post_filter(*qs*)
: Hook for doing post-filter modification to the queryset. This is also the place where any custom filtering should take place.

  Runs only if the form validates.

pre_filter(*qs*)
Hook for doing pre-filter modification to the queryset

Runs prior to any form filtering and is run regardless form validation.

class fusionbox.forms.**SortForm**(**args*, ***kwargs*)
Base class for implementing sorting on a model.

Example Usage

```
class UserSortForm(SortForm):
    HEADERS = (
        {'column': 'username', 'title': 'Username', 'sortable': True},
        {'column': 'email', 'title': 'Email Address', 'sortable': True},
        {'column': 'is_active', 'title': 'Active', 'sortable': False},
    model = User
```

The sort field for this form defaults to a HiddenInput widget which should be output within your form to preserve sorting accross any form submissions.

get_queryset()
Returns an ordered queryset, sorted based on the values submitted in the sort parameter.

headers()
Returns an object with the following template variables:

{{ form.headers }}

- access to the header

{{ header.title }}

- title declared for this header

{{ header.sortable }}

- boolean for whether this header is sortable

{{ header.active }}

- boolean for whether the queryset is currently being sorted by this header

{{ header.classes }}

- list of css classes for this header. (active, ascending|descending)

{{ header.priority }}

- numeric index for which place this header is being used for ordering.

{{ header.querystring }}

- querystring for use with progressive sorting (sorting by multiple fields)

{{ header.remove }}

- querystring which can be used to remove this header from sorting

{{ header.singular }}

- querystring which can be used to sort only by this header

Example:

```
{% for header in form.headers %}
  {% if header.priority %}
  <th scope="col" class="active {{ form.prefix }}-{{ header.column }}">
```

```
            <div class="sortoptions {{ header.classes|join:' ' }}">
              <a class="sortremove" href="?{{ header.remove }}" title="Remove from sorting">X</a>
              <span class="sortpriority" title="Sorting priority: {{ header.priority }}">{{ header.p
              <a href="?{{ header.querystring }}" class="toggle" title="Toggle sorting"></a>
            </div>
          {% else %}
          <th scope="col" class="{{ form.prefix }}-{{ header.column }}">
          {% endif %}

          {% if header.sortable %}
             <div class="text"><a href="?{{ header.querystring }}">{{ header.title }}</a></div>
          {% else %}
             <div class="text">{{ header.title|safe }}</div>
          {% endif %}
          </th>
        {% endfor %}
```

post_sort(*qs*)
Hook for doing post-sort modification of the queryset. Will not be called on an invalid form.

pre_sort(*qs*)
Hook for doing pre-sort modification of the queryset. Runs regardless of whether the form is valid.

class fusionbox.forms.**CsvForm**(**args*, ***kwargs*)
Base class for implementing csv generation on a model.

Example:

Given this class...

```
    class UserFilterForm(FilterForm):
        model = User

        CSV_COLUMNS = (
                {'column': 'id', 'title': 'Id'},
                {'column': 'username', 'title': 'Username'},
                {'column': 'email__domain_name', 'title': 'Email Domain'},
                )

        FILTERS = {
            'active': 'is_active',
            'date_joined': 'date_joined__gte',
            'published': None, # Custom filtering
            }

        PUBLISHED_CHOICES = (
                ('', 'All'),
                ('before', 'Before Today'),
                ('after', 'After Today'),
                )

        active = forms.BooleanField(required=False)
        date_joined = forms.DateTimeField(required=False)
        published = forms.ChoiceField(choices=PUBLISHED_CHOICES, widget=forms.HiddenInput())

        def pre_filter(self, queryset):
            published = self.cleaned_data.get('published')
            if published == '':
                return queryset
            elif published == 'before':
```

```
            return queryset.filter(published_at__lte=datetime.datetime.now())
        elif published == 'after':
            return queryset.filter(published_at__gte=datetime.datetime.now())
```

```
>>> # This code in a repl will produce a string buffer with csv output for
>>> # the form's queryset
>>> form = UserFilterForm(request.GET, queryset=User.objects.all())
>>> form.csv_content()
<StringIO.StringO object at 0x102fd2f48>
>>>
```

`CSV_COLUMNS` defines a list of properties to fetch from each obj in the queryset which will be output in the csv content. The `column` key defines the lookup path for the property. This can lookup a field, property method, or method on the model which may span relationships. The `title` key defines the column header to use for that property in the csv content.

The *`csv_content()`* method returns a string buffer with csv content for the form's queryset.

`csv_content``()`
Returns the objects in the form's current queryset as csv content.

class `fusionbox.forms.`**`UncaptchaForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*)
Extension of `django.forms.Form` which adds an UncaptchaField to the form.

class `fusionbox.forms.`**`UncaptchaModelForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*, *instance=None*)
Extension of `django.forms.ModelForm` which adds an UncaptchaField to the form.

`fusionbox.forms.`**`csv_getattr`**(*obj*, *attr_name*)
Helper function for CsvForm class that gets an attribute from a model with a custom exception.

8.2 Fields

class `fusionbox.forms.`**`MonthField`**(**args*, ***kwargs*)
`MonthField` is a `TypedChoiceField` that selects a month. Its python value is a 1-indexed month number.

class `fusionbox.forms.`**`MultiFileField`**(**args*, ***kwargs*)
Implements a multifile field for multiple file uploads.

This class' *clean* method is implented by currying *super.clean* and running map over *data* which is a list of file upload objects received from the `MultiFileWidget`.

Using this field requires a little work on the programmer's part in order to use correctly. Like other Forms with fields that inherit from `FileField`, the programmer must pass in the kwarg *files* when creating the form instance. For example:

```
` form = MyFormWithFileField(data=request.POST,
files=request.FILES) `
```

After validation, the cleaned data will be a list of files. You might want to iterate over in a manner similar to this:

‘‘‘ if form.is_valid():

for media_file in form.cleaned_data['field_name']:

MyMedia.objects.create(name=media_file.name, file=media_file)

"""

widget
alias of `MultiFileWidget`

class `fusionbox.forms.`**`NoAutocompleteCharField`**(*max_length=None*, *min_length=None*, **args*, ***kwargs*)
`NoAutocompleteCharField` is a subclass of `CharField` that sets the `autocomplete` attribute to `off`. This is suitable for credit card numbers and other such sensitive information.

This should be used in conjunction with the *sensitive_post_parameters <https://docs.djangoproject.com/en/dev/howto/error-reporting/#sensitive_post_parameters>* decorator.

class `fusionbox.forms.fields.`**`FutureYearField`**(*number_of_years=6*, **args*, ***kwargs*)
`FutureYearField` is a `TypedChoiceField` that selects a year a defined period in the future. Useful for credit card expiration dates.

CHAPTER 9

Newsletter

CHAPTER 10

Fabric helpers

10.1 Usage

A typical `fabfile.py` using `fusionbox.fabric_helpers` looks like:

```
from fusionbox.fabric_helpers import *

env.roledefs = {
        'dev': ['dev.fusionbox.com'],
        }

env.project_name = 'foobar'
env.short_name = 'fb'

stage = roles('dev')(stage)
```

CHAPTER 11

Class based Views

11.1 REST Views

Fusionbox generic views.

class fusionbox.views.**RestView**(***kwargs*)
Inherit this base class to implement a REST view.

This view will handle:

- authentication (throuh the `auth` method)
- dispatching to the proper HTTP method function
- returning a proper error status code.

It also implements a default response for the OPTIONS HTTP request method.

auth(**args*, ***kwargs*)
Hook for implementing custom authentication.

Raises `NotImplementedError` by default. Subclasses must overwrite this.

dispatch(**args*, ***kwargs*)
Authenticates the request and dispatches to the correct HTTP method function (GET, POST, PUT,...).

Translates exceptions into proper JSON serialized HTTP responses:

- ValidationError: HTTP 409
- Http404: HTTP 404
- PermissionDenied: HTTP 403
- ValueError: HTTP 400

options(*request*, **args*, ***kwargs*)
Implements a OPTIONS HTTP method function returning all allowed HTTP methods.

11.2 REST View Mixins

View classes to help facilitate the creation of REST APIs

class fusionbox.views.rest.**JsonRequestMixin**
Adds a `data` method on the view instance. It returns the GET parameters if it is a GET request. It will return the python representation of the JSON sent with the request body.

data()
Helper class for parsing JSON POST data into a Python object.

class fusionbox.views.rest.**JsonResponseMixin**
Sets the response MIME type to application/json and serializes the context obj as a JSON string.

http_method_not_allowed(**args*, ***kwargs*)
Returns super after setting the Content-Type header to application/json

render_to_response(*obj*, ***response_kwargs*)
Returns an HttpResponse object instance with Content-Type: application/json.

The response body will be the return value of self.serialize(obj)

serialize(*obj*)
Returns a json serialized string object encoded using *fusionbox.core.serializers.FusionboxJSONEncoder*.

CHAPTER 12

Managers

class fusionbox.db.models.**QuerySetManager**
http://djangosnippets.org/snippets/734/

Easy extending of the base manager without needing to define multiple managers and queryset classes.

Example Usage

```
from django.db.models.query import QuerySet
from fusionbox.db.models import QuerySetManager

class Foo(models.Model):
    objects = QuerySetManager()
    class QuerySet(QuerySet):
        def published(self):
            return self.filter(is_published=True)
```

fusionbox.db.models.**generic_upload_to**(*instance*, *filename*)
Generic *upload_to* function for models.FileField and models.ImageField which uploads files to *<app_label>/<module_name>/<file_name>*.

CHAPTER 13

HTTP Helpers

class fusionbox.http.**JsonResponse**(*context*, **args*, ***kwargs*)

Takes a jsonable object and returns a response with its encoded value. Also sets the Content-Type to json.

Usage:

```
    def aview(request):
        return JsonResponse({'foo': 1})
```

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

f

O

P

Q

R

S

T

U

V

W