django-scaffold Documentation

Release 1.1.1

James Stevenson

May 27, 2015

Contents

A guide to using and extending scaffold.

Contents:

CHAPTER 1

Installation

Scaffold is installed like any other Django app:

```
?>pip install django-scaffold
```

or, if you can't use pip:

```
?>easy_install django-scaffold
```

or, if all else fails, place scaffold package where your python interpreter can find it and then

When you have the package installed, add it to the list of apps in the INSTALLED_APPS setting of your *settings.py* file.

CHAPTER 2

Creating an app to extend scaffold

Although you've installed it, scaffold won't do much by itself. Think of it as a kind of *abstract* application, akin to the notion of an abstract class in python. In other words, scaffold is meant to be extended by an application that you create. We'll call this the **concrete app** from here on out.

This is not to say scaffold doesn't have a lot going on under the hood; like any Django app, scaffold has views, models, templates and media files. However, any one of these elements can–and should be–extended or overridden as needed. Let's walk through the steps we'll need to get a basic **concrete app** working using scaffold.

A typical use case for scaffolding is creating a tree of sections and subsections for a web site. Let's say we're putting together a simple news site, which will have sections for news, weather, entertainment and shopping. Some of these sections–entertainment–will have sub-sections (say, movies, theater, music, and art). Content creators will be able to create articles which can be attached to any one of these sections and subsections. All in all, a simple, common task for a web developer and one that scaffold can help with.

2.1 1. Create a new application.

Let's start by creating an application for handling sections in the site. We'll even call the application "sections":

```
python manage.py startapp sections
```

2.2 2. Create a model which extends scaffold

We decide that a section should have a title, a description (which we'll use in meta tags for SEO purposes), and a photo. We'll start by creating a model in the models.py file that extends the `scaffold.models.BaseSection` model. Here's some of what's in that `BaseSection` model:

```
class BaseSection(MP_Node):

    slug = models.SlugField(_("Slug"), help_text=_("Used to construct URL"))
    title =  models.CharField(_("Title"), max_length=255)
    order = models.IntegerField(_("Order of section"), blank=True, default=0)
```

Notice that the model only defines 3 fields. Let's ignore "order" for the moment; scaffold assumes that anything that extends `BaseSection` will have at least a slug (for constructing the url of the section) and a title.

Now we can create a model which adds the fields we need. In the `models.py` for your new app, add the following:

```
from scaffold.models import BaseSection

class Section(BaseSection):
    description = models.TextField("Description", help_text="For SEO.")
    photo = models.ImageField("Photo", upload_to="section_images")
```

...and that's it, we're done. BaseSection provides a number of powerful methods that we'll get into later.

2.3 3. Setup your URL Configuration

Change the default urls.py file for your Django project to the following:

```
try:
    from django.conf.urls import *
except ImportError:
    from django.conf.urls.defaults import *

from django.contrib import admin
admin.autodiscover()
urlpatterns = patterns('',
    (r'^admin/', include(admin.site.urls)),
    url(r'^(?P<section_path>.+)/$', 'scaffold.views.section', name="section"),
)
```

We've done a couple things here. First, we've enabled the admin app by uncommenting the lines which turn on autodiscover and route `/admin/` urls to the admin app. That takes care of the admin interface and allows us to manage a sections/subsections tree in the admin (Scaffold provides a number of admin views to manage your models, but these are all handled in a special `ModelAdmin` class called `SectionAdmin` and do not need to be specially referenced in your URL conf.)

But how will we actually view a section or subsection on the website? The second url pattern handles this:

```
url(r'^(?P<section_path>.+)/$', 'scaffold.views.section', name="section")
```

This line works for a very specific, but common URL addressing schema: Top level sections will have root-level slugs in the url. Our site has an "Entertainment" section with the slug `entertainment`. The URL will therefore be **http://www.mysite.com/entertainment/**. There is also a subsection of entertainment, called "Dining Out" with the slug `dining`. It's URL would be **http://www.mysite.com/entertainment/dining/**.

Like almost everything about scaffold, you are not required to use this pattern. You can write your own url conf, or completely override the `scaffold.views.section` view if you like.

Note

The positioning of the url patterns here is very deliberate. The regular expression '^(?P<section_path>.+)/$' is rather greedy and will match anything, therefore we put it last.

2.4 4. Register your Section model in the admin site

Create an admin.py file in your concrete application and register your new `Section` model there:

```
from django.contrib import admin
from models import Section
from scaffold.admin import SectionAdmin
```

```
admin.site.register(Section, SectionAdmin)
```

You'll notice that we're registering our concrete model with the admin site using the `SectionAdmin` class in django-scaffold. This step is crucial if you want scaffold to work properly in the admin interface. The standard `admin.ModelAdmin` class does not provide the special properties and views needed to manage scaffold's concrete models.

2.5 5. Add the necessary project settings

All that's left to do is add a single setting to your Django project. In your settings.py file, place the following:

```
SCAFFOLD_EXTENDING_APP_NAME = 'sections'
```

Note: this example assumes your concrete app is called *sections*. Use whatever you've named your app as the *SCAFFOLD_EXTENDING_APP_NAME* setting.

2.6 6. Make the the scaffold media available.

Django-scaffold has a number of CSS, JavaScript and image files which it uses in the admin interface. These are stored in media/scaffold in the scaffold application directory. You can copy the `scaffold` folder from the scaffold media directory to your own project's media directory, but it's best to simply create a symlink instead. (Make sure, if you're using apache to server this, you have the `Options FollowSymLinks` directive in place.)

At this point, you should be able to start up your Django project, browse to the admin interface and start creating sections.

CHAPTER 3

Customizing scaffold

In the previous section (Creating an app to extend scaffold) we created an app that used scaffold, but the only thing we customized was the model the app used. We used the URL conf, views, and templates provided by scaffold, but we don't have to.

Almost any piece of scaffold can be overridden in your concrete app. For example, let's say we want to create our own view of our `Section` model, rather than using scaffold's. And while we're at it, we want to change how the url addressing of sections works.

3.1 Customizing URLs

By default, scaffold uses a common URL addressing scheme for sections and subsections. A url like `"/projects/local/water/"` means give me the section with the slug "water", which is a child of the "local" section, which in turn is a child of the "projects" section. This is a common–and useful way–of orienting the user within the IA of the site using the URL.

But, let's say you want a simpler scheme, with URLs like `"/sections/local/"` or `"/sections/water/"`. Heres how our URL conf file looked at the end of the last section:

```
try:
    from django.conf.urls import *
except ImportError:
    from django.conf.urls.defaults import *

from django.contrib import admin
admin.autodiscover()
urlpatterns = patterns('',
    (r'^admin/', include(admin.site.urls)),
    url(r'^(?P<section_path>.+)/$', 'scaffold.views.section', name="section"),
)
```

Now we can make the urlpatterns look like this:

```
urlpatterns = patterns('',
    url(r'^sections/(?P<slug>[\w-]+)/?$', 'scaffold.views.section', name="section"),
    (r'^admin/', include(admin.site.urls)),
)
```

3.2 Customizing Views

The one problem is that we aren't passing scaffold.views.section the arguments it wants anymore, so we'll need to create our own view function:

```
urlpatterns = patterns('',
    url(r'^sections/(?P<slug>[\w-]+)/?$', 'sections.views.section'),
    (r'^admin/', include(admin.site.urls)),
)
```

Then we create a "section" view in our app's *views.py* file:

```
from django.shortcuts import get_object_or_404
from django.views.generic import simple
from models import Section

def section(request, slug=None):
    section = get_object_or_404(Section, slug=slug)
    return simple.direct_to_template(request,
        template="scaffold/section.html",
        extra_context={
            'section': section,
        }
    )
```

3.3 Customizing Templates

We're still using scaffold's template, but this is probably one of the first thing's you'd want to override. You can do this in two ways: create a `scaffold` directory in your own project's templates folder, then create a `section.html` file where you template the object yourself. Or, if you've written your own view function like we have, then you can call the template whatever you want:

```
from django.shortcuts import get_object_or_404
from django.views.generic import simple
from models import Section

def section(request, slug=None):
    section = get_object_or_404(Section, slug=slug)
    return simple.direct_to_template(request,
        template="sections/section.html",
        extra_context={
            'section': section,
        }
    )
```

3.4 Customizing the Admin

One of scaffold's best features is it's integration with the Django admin. Even this, however, is customizable. All scaffold admin views are located in the `scaffold.admin.SectionAdmin` class. Besides custom versions of the usual admin views, (change list, change object, add object, and delete object) scaffold provides views to move nodes in the tree, view all content attached to a node, and and order content attached to a node. Read the Django documentation to find out more about how to customize a model admin.

Note that most customizations possible for the `ModelAdmin` class are possible for `SectionsAdmin`, although a few are ignore because of differences in the UI.

3.4.1 Unspported `ModelAdmin` Options

- `date_hierarchy`
- `list_display`
- `list_editable`
- `list_filter`
- `list_per_page`
- `list_select_related`
- `ordering`
- `search_fields`
- `actions`

CHAPTER 4

Available settings

Here's a full list of all available settings for the django-scaffold application, in alphabetical order, and their default values.

4.1 SCAFFOLD_ALLOW_ASSOCIATED_ORDERING

Default: `True`

One of scaffold's features is that you can order multiple types of content that is attached to a scaffold item. For example, lets say you extend `scaffold.models.BaseSection` with a model called Section. By it's very nature, one section can be the child of another. However, you might also create a model called `Article` which has a Foreign-key relationship with a section, and thus is it's child too. In fact you might even establish a generic foreign key relationship between a model and your `Section` model. When this property is set to True, you can order all items relative to each other via the admin interface.

Note that for this to work, all models must share a common field were the order, relative to each other, can be stored as an integer. By default, models that inherit from `scaffold.models.BaseSection` assume this field is called 'order'.

If you don't want this ordering option to be available in the admin interface for associated content, set this to False.

4.2 SCAFFOLD_EXTENDING_APP_NAME

Default: Not defined

The name of the concrete application which is extending scaffold. Note that this setting is required: scaffold will not work without it.

4.3 SCAFFOLD_EXTENDING_MODEL_PATH

Default: `'{SCAFFOLD_EXTENDING_APP_NAME}.models.Section'`

The location of the model which extends `scaffold.models.BaseSection`. By default, it assumes this model is called `Section`, thus if you create an app named "pages", scaffold will try to import `pages.models.Section` unless this setting is provided.

4.4 SCAFFOLD_LINK_HTML

Default:

```
(
    ('edit_link', (
        "<a class=\"changelink\" href=\"%s/\">"
        "edit</a>"
    ),),
    ('add_link', (
        "<a class=\"addlink\" href=\"%s/create/\">"
        "add child</a>"
    ),),
    ('del_link', (
        "<a class=\"deletelink\" href=\"%s/delete/\">"
        "delete</a>"
    ),),
    ('list_link', (
        "<a class=\"listlink\" href=\"%s/related/\">"
        "list content</a>"
    ),)
)
```

These are the four links which are added to every item in the tree in the scaffold admin view. You can override this tuple of tuples with your own links, or reorder this one.

4.5 SCAFFOLD_PATH_CACHE_KEY

Default: `'scaffold-path-map'`

The key name under which scaffold stores it's path cache values. This should only be changed to avoid key collisions in the cache

4.6 SCAFFOLD_PATH_CACHE_TTL

Default: `43200` (that's equal to 12 hours)

The length of time (in seconds) an item persists in the path cache. The path cache is a way of very quickly (and without a DB call) looking up scaffold items from a url. Note that that adding, editing the slug of, or removing a scaffold item automatically refreshes the cache.

4.7 SCAFFOLD_VALIDATE_GLOBALLY_UNIQUE_SLUGS

Default: `False`

If set to `True` this setting will require all slugs to be globally unique. Otherwise, slugs can be reused **except** among objects with a common parent (in other words, an object cannot have two children with the same slug).

4.8 SCAFFOLD_TREEBEARD_NODE_TYPE

Default: `'treebeard.mp_tree.MP_Node'`

Allows the user to specify the tree model implementation to use. Allowed values are:

- `'treebeard.mp_tree.MP_Node'`
- `'treebeard.al_tree.AL_Node'`
- `'treebeard.ns_tree.NS_Node'`

Depending on the read/write profile of your site, some node types will be more efficient then others. Refer to the treebeard docs for an explanation of each type.

CHAPTER 5

The scaffold API

5.1 Model methods

The following methods are provided by `scaffold.models.BaseSection`:

class `scaffold.models.`**`BaseSection`**(**args*, ***kwargs*)

An abstract model of a section or subsection. This class provides a base level of functionality that should serve as scaffold for a custom section object.

`get_associated_content`(*only=[]*, *sort_key=None*)

This method returns an aggregation of all content that's associated with a section, including subsections, and other objects related via any type of foreign key. To restrict the types of objetcs that are returned from foreign-key relationships, the only argument takes a list of items with the signature:

```
{app name}.{model name}
```

For example, if you wanted to retrieve a list of all subsections and associated articles only, you could do the following:

```
section = Section.objects.all()[0]
section.get_associated_content(only=['articles.article'])
```

Furthermore, if all objects have a commone sort key, you can specify that with the sort_key parameter. So, since sections have an 'order' field, if articles had that field as well, you could do the following:

```
section = Section.objects.all()[0]
section.get_associated_content(
    only=['articles.article'],
    sort_key='order'
)
```

...and the list returned would be sorted by the 'order' field.

`get_first_populated_field`(*field_name*)

Returns the first non-empty instance of the given field in the sections tree. Will crawl from leaf to root, returning *None* if no non-empty field is encountered.

`get_related_content`(*sort_fields=[]*, *infer_sort=False*)

A method to access content associated with a section via a foreign-key relationship of any type. This includes content that's attached via a simple foreign key relationship, and content that's attached via a generic foreign key (for example, through a subclass of the SectionItem model).

This method returns a list of tuples:

```
        (object, app name, model name, relationship_type)
```

To sort associated content, pass a list of sort fields in via the sort_fields argument. For example, let's say we have two types of content we know could be attached to a section: articles and profiles Articles should be sorted by their 'headline' field, while profiles should be sorted by their 'title' field. We would call our method thusly:

```
        section = Section.objects.all()[0]
        section.get_related_content(sort_fields=['title', 'headline'])
```

This will create a common sort key on all assciated objects based on the first of these fields that are present on the object, then sort the entire set based on that sort key. (NB: This key is temporary and is removed from the items before they are returned.)

If 'infer_sort' is True, this will override the sort_fields options and select each content type's sort field based on the first item in the 'ordering' property of it's Meta class. Obviously, infer_sort will only work if the types of fields that are being compared are the same.

get_subsections()
This method return all subsections of the current section.

type
A property that returns the string 'section' if the section is at the root of the tree, 'subsection' otherwise.

5.2 Admin

The sections application contains the following views.

5.3 Middleware

Use the middleware if you need access to the section outside the view context.

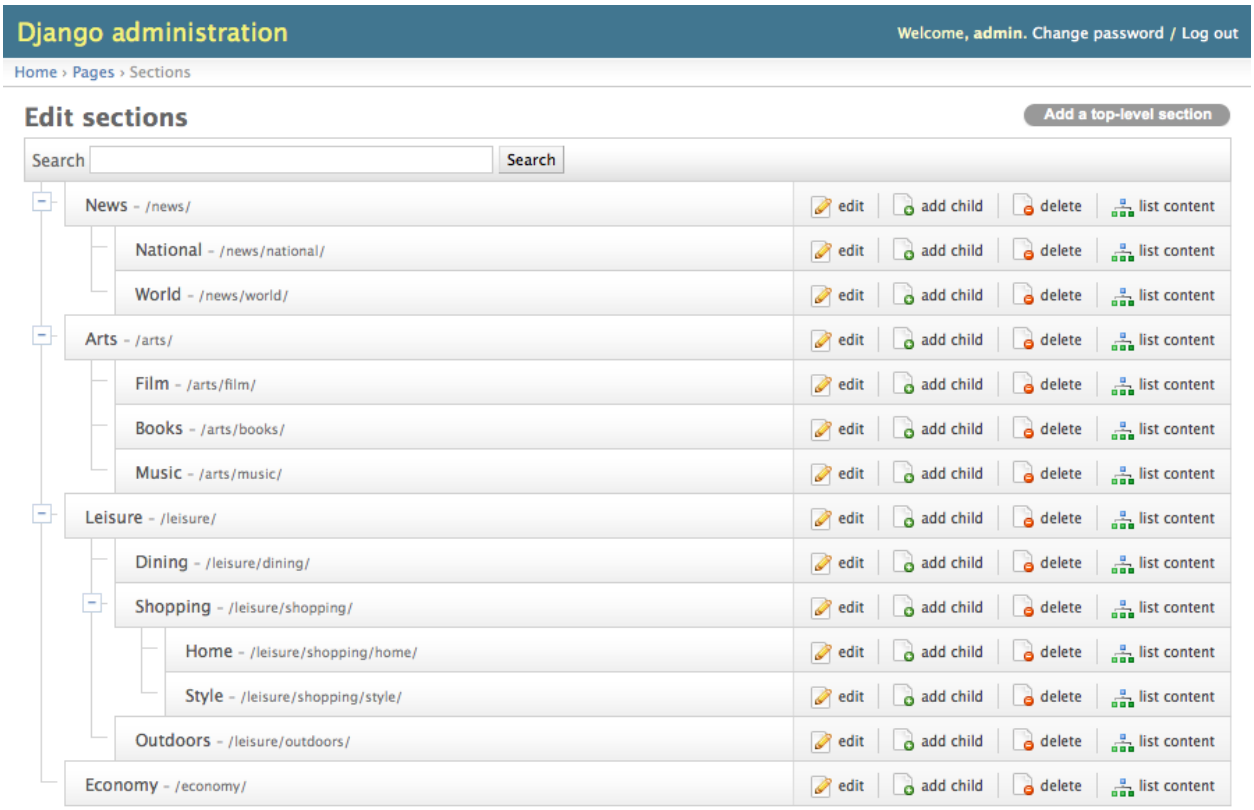
class scaffold.middleware.**SectionsMiddleware**
Middleware that stores the current section (if any) in the thread of the currently executing request

process_request(*request*)
Determine the section from the request and store it in the currently executing thread where anyone can grab it (remember, in Django, there's one request per thread).

scaffold.middleware.**get_current_section**()
Convenience function to get the current section from the thread of the currently executing request, assuming there is one. If not, returns None. NB: Make sure that the SectionsMiddleware is enabled before calling this function. If it is not enabled, this function will raise a MiddlewareNotUsed exception.

scaffold.middleware.**lookup_section**(*lookup_from*)
NB: *lookup_from* may either be an HTTP request, or a string representing an integer.

scaffold.middleware.**reset_section_path_map**(*sender*, ***kwargs*)


Django administration
Welcome, admin. Change password / Log out
Home › Pages › Sections
Edit sections
Add a top-level section
Search
Search
News - /news/
National - /news/national/
World - /news/world/
Arts - /arts/
Film - /arts/film/
Books - /arts/books/
Music - /arts/music/
Leisure - /leisure/
Dining - /leisure/dining/
Shopping - /leisure/shopping/
Home - /leisure/shopping/home/
Style - /leisure/shopping/style/
Outdoors - /leisure/outdoors/
Economy - /economy/
edit
add child
delete
list content

CHAPTER 6

Indices and tables

- search

S

B

G

L

P

R

S

T