Django Smartmin Documentation

Release 1.8.1

Nyaruka Ltd

Nov 28, 2018

Contents

1 Installation **3**

2 Configuration **5**

3 Versioning: **7**

4 Contents: **9**

4.1 QuickStart . . . 9
4.2 Adding Search . . . 12
4.3 Permissions . . . 13
4.4 Smartmin Views . . . 14
4.5 SmartCreateView . . . 14
4.6 SmartReadView . . . 15
4.7 SmartUpdateView . . . 15
4.8 SmartDeleteView . . . 16
4.9 SmartListView . . . 17
4.10 HTML5 Boilerplate . . . 18
4.11 File Layout . . . 18
4.12 Blocks . . . 18
4.13 Customizing . . . 18
4.14 Group Creation . . . 19
4.15 Defining Permissions . . . 19
4.16 Assigning Permissions for Groups . . . 19
4.17 Permissions on Views . . . 20
4.18 Users . . . 20
4.19 Configuration . . . 20
4.20 Miscellaneous Utilities . . . 21

5 Indices and tables **23**

Smartmin was born out of the frustration of the Django admin site not being well suited to being exposed to clients. Smartmin aims to allow you to quickly build scaffolding which you can customize by using Django class based views.

It is very opininated in how it works, if you don't agree, Smartmin may not be for you:

- Permissions are used to gate access to each page, embrace permissions throughout and you'll love this
- CRUDL operations at the object level, that is, Create, Read, Update, Delete and List, permissions and views are based around this
- URL automapping via the the CRUDL objects, this should keep things very very DRY

The full documentation can be found at: http://readthedocs.org/docs/smartmin/en/latest/

The official source code repository is: http://www.github.com/nyaruka/smartmin/

Built in Rwanda by Nyaruka Ltd: http://www.nyaruka.com

CHAPTER 1

Installation

The easiest and fastest way of downloading smartmin is from the cheeseshop:

```
% pip install smartmin
```

This will take care of installing all the appropriate dependencies as well.

CHAPTER 2

Configuration

To get started with smartmin, the following changes to your `settings.py` are needed:

```
# create the smartmin CRUDL permissions on all objects
PERMISSIONS = {
  '*': ('create', # can create an object
        'read',   # can read an object, viewing it's details
        'update', # can update an object
        'delete', # can delete an object,
        'list'),  # can view a list of the objects
}

# assigns the permissions that each group should have, here creating an Administrator␣
↪group with
# authority to create and change users
GROUP_PERMISSIONS = {
    "Administrator": ('auth.user.*',)
}

# set this if you want to use smartmin's user login
LOGIN_URL = '/users/login'
```

You'll also need to add smartmin to your installed apps:

```
INSTALLED_APPS = (
  # .. other apps ..

  'smartmin',
)
```

Finally, if you want to use the default smartmin views for managing users and logging in, you'll want to add the smartmin.users app to your `urls.py`:

```
urlpatterns = [
  # .. other patterns ..
```

(continues on next page)

(continued from previous page)

```
  url(r'^users/', include('smartmin.users.urls')),
]
```

You can now sync your database and start the server:

```
% python manage.py migrate
% python manage.py runserver
```

And if you want to see a Smartmin view in action, check out smartmin's user management pages for a demo that functionality by pointing your browser to:

```
http://localhost:8000/users/user
```

From here you can create, update and list users on the system, all using standard smartmin views. The total code to create all this functionality is less than 30 lines of Python.

CHAPTER 3

Versioning:

Smartmin will release major versions in step (or rather a bit behind) Django's major releases. Version 1.11 actually works against Django 1.11, 1.10 and 1.9 - we hope to support the 3 most recent versions in each release. Smartmin is used in quite a few of our projects, so we don't rock the boat too much, even in major releases. That said, we don't guarantee that major releases always be backwards compatible.

At the onset of each new Django version we will upgrade Twitter Bootstrap to the current version. Currently for 1.11, which targets Django 1.11, that means Twitter Bootstrap 3. Note that some of our screenshots are a bit outdated, our standard views now use Bootstrap styling, not the more Django admin looking pages shown in our docs. (PRs accepted to fix this!)

CHAPTER 4

Contents:

4.1 QuickStart

Here's a simple example of Smartmin with a very simple model to show you how it works. You can find out more details on each of these features in the appropriate sections but this is a good overview given a simple model.

For this example, we are going to use the ubiquitous blog example. We'll simply add a way for authenticated users to create a blog post, list them, edit them and remove them.

To start out with, let's create our app:

```
python manage.py startappp blog
```

Don't forget to add the app to your `INSTALLED_APPS` in `settings.py`.

Now, let's add a (deliberately simple) Post object to our newly created `models.py`:

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=128,
                             help_text="The title of this blog post, keep it relevant
↪")
    body = models.TextField(help_text="The body of the post, go crazy")
    tags = models.CharField(max_length=128,
                            help_text="Any tags for this post")
```

Ok, so far, this all normal Django stuff. Now let's create some views to manage these Post objects. This is where Smartmin comes in.

Smartmin provides a controller that lets you easily hook up Create, Read, Update, Delete and List views for your object. In Smartmin, we call this CRUDL, and we call the helper class that defines it a SmartCRUDL. So in our `views.py` let's create a CRUDL for our Post object:

```python
from smartmin.views import *
from .models import *

class PostCRUDL(SmartCRUDL):
    model = Post
```

You'll see that right now, all we are doing is defining the model that our CRUDL is working with. Everything else is using defaults.

Finally, we'll have to create a `urls.py` for our app, and hook in this CRUDL:

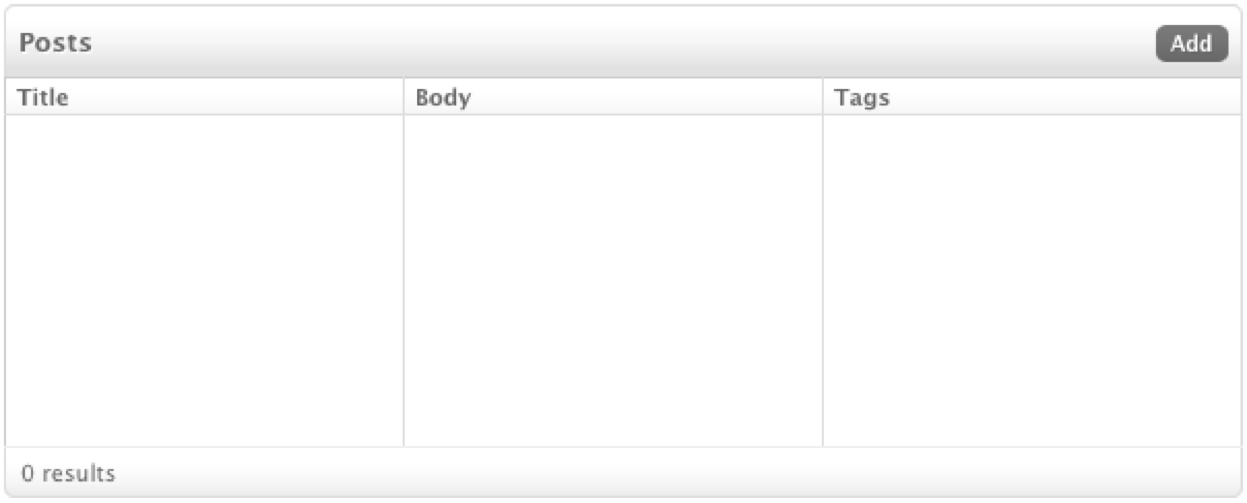
```python
from .views import *

urlpatterns = PostCRUDL().as_urlpatterns()
```

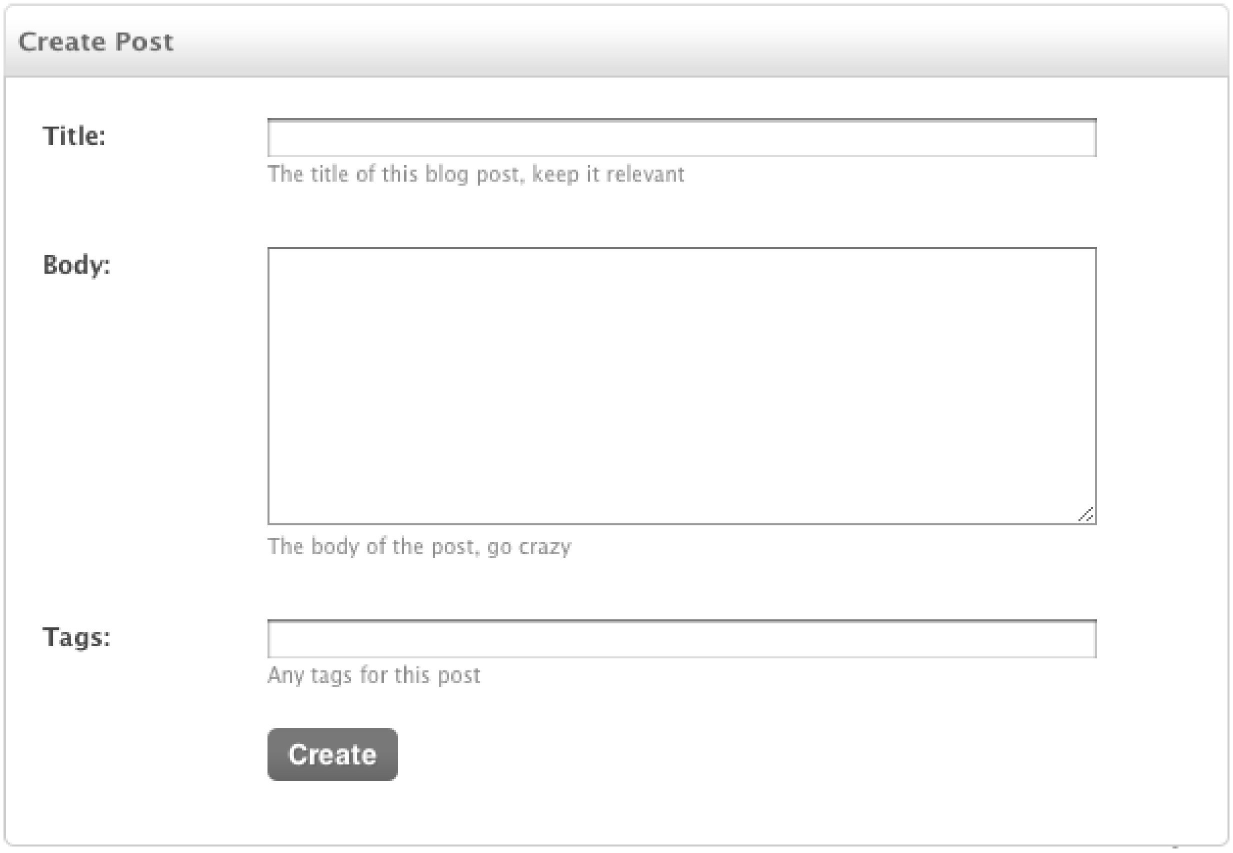
Again, Smartmin keeps this very, very, DRY. The urls (and reverse patterns) are created automatically using standard Django conventions and the name of our model. The last part is you'll need to add this urls.py to your global `urls.py`:

```python
urlpatterns = patterns('',
  # .. other url patterns
  url(r'^blog/', include('blog.urls')),
)
```

With a quick `python manage.py syncdb` that should be it. You should be able to start your server and hit the index for your CRUDL object at `http://localhost:8000/blog/post/` and get a view that looks like this:



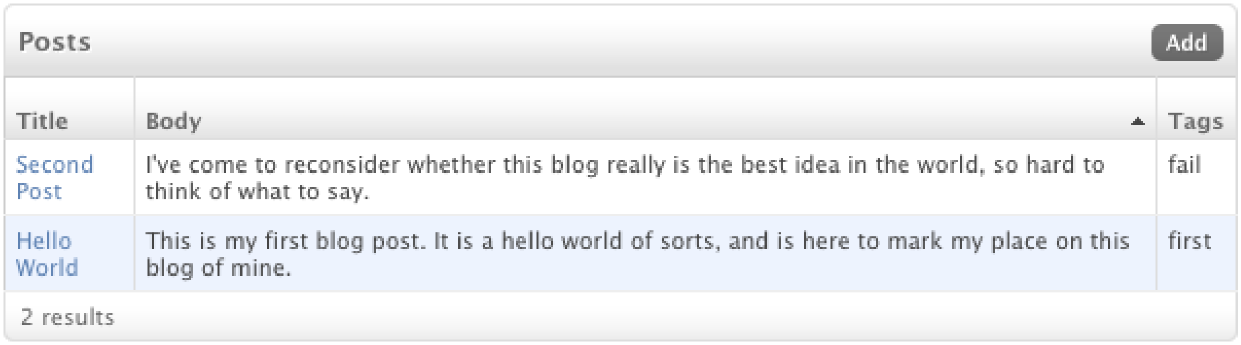
You'll notice that CRUDL has no only wired up the views, but given us a standard list view by default. If we click on the 'add' button, we'll get the default Create view for our object:



Create Post

Title:	The title of this blog post, keep it relevant
Body:	The body of the post, go crazy
Tags:	Any tags for this post

If we add a few items, we'll see our list displays things appropriately:

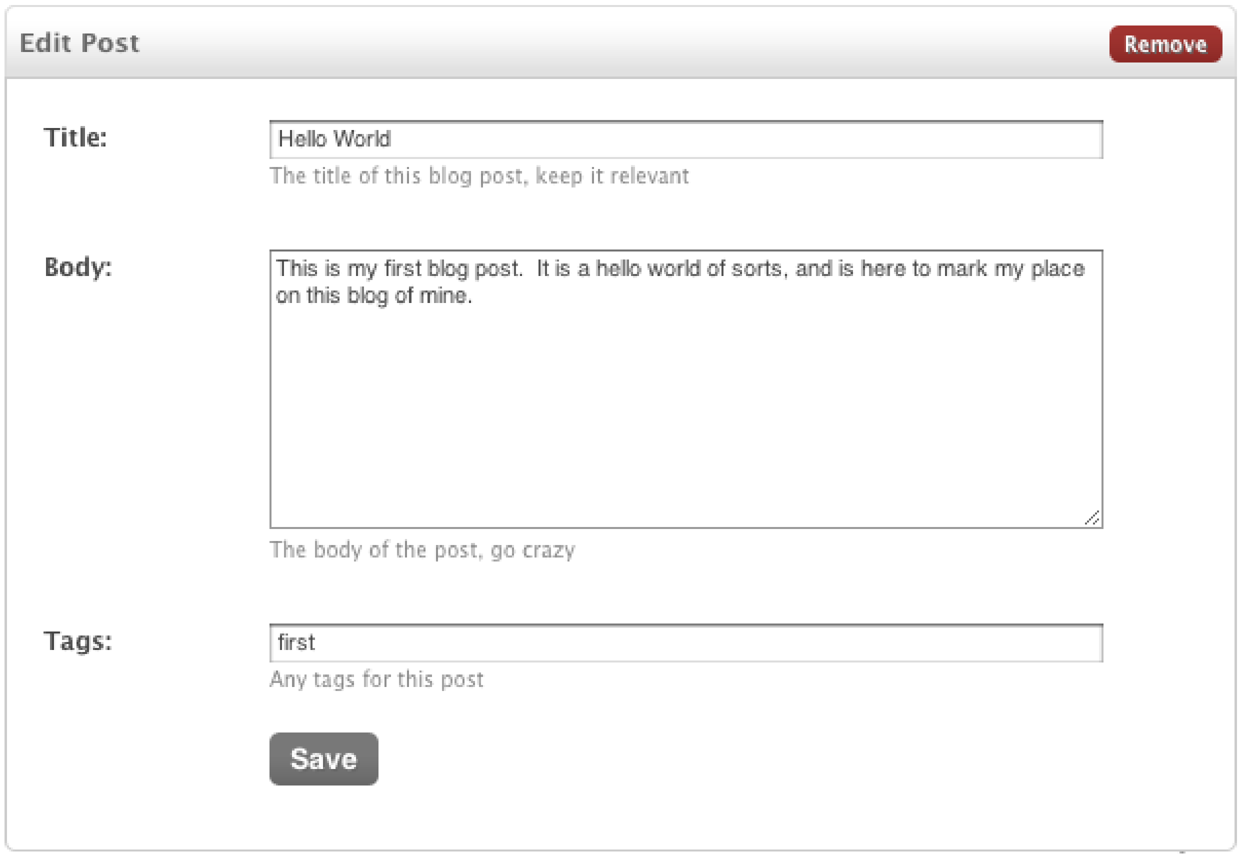


Posts

Title	Body	Tags
Second Post	I've come to reconsider whether this blog really is the best idea in the world, so hard to think of what to say.	fail
Hello World	This is my first blog post. It is a hello world of sorts, and is here to mark my place on this blog of mine.	first

2 results

We can click on any of the items to edit them, and when editing, and from there even remove them if needed:



Title: Hello World
The title of this blog post, keep it relevant

Body: This is my first blog post. It is a hello world of sorts, and is here to mark my place on this blog of mine.
The body of the post, go crazy

Tags: first
Any tags for this post

And that's it. You probably want to learn how to customize things, either at the CRUDL layer or on each individual view.

4.2 Adding Search

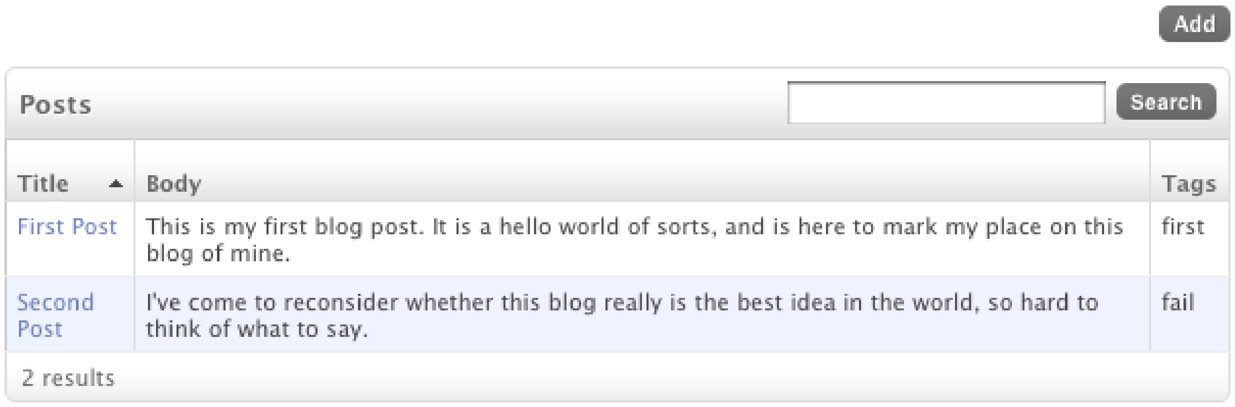
Ok, so now we have some basic views, but let's spruce up our list view by adding search. All we need to do to enable search is define which fields we want to be searchable. To do this we'll have to overload the default ListView on our PostCRUDL object:

```python
class PostCRUDL(SmartCRUDL):
  model = Post

  class List(SmartListView):
    search_fields = ('title__icontains', 'body__icontains')
    default_order = 'title'
```

So we are doing two things here. First, by defining `search_fields` we are telling smartmin to enable searching on this list view, and to search the contents of the title and body when doing searches. While we are at it, we have also set the default ordering for results to be by the title attribute.

Here's the result:



Posts

Title	Body	Tags
First Post	This is my first blog post. It is a hello world of sorts, and is here to mark my place on this blog of mine.	first
Second Post	I've come to reconsider whether this blog really is the best idea in the world, so hard to think of what to say.	fail

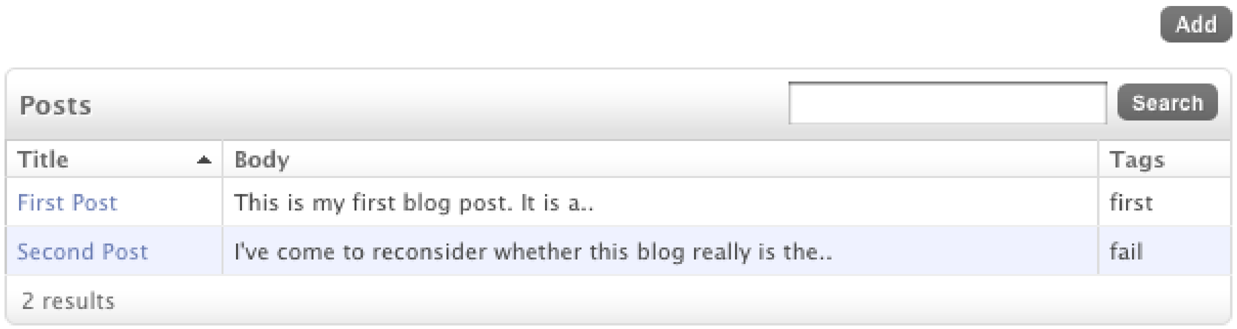
2 results

One thing that could still be better would be to only show the first few words of a post so we don't blow up the table. We can override what smartmin uses as the value of a column by just defining a `get_[fieldname]` method on our view:

```
class List(SmartListView):
  search_fields = ('title__icontains', 'body__icontains')
  default_order = 'title'

  def get_body(self, obj):
      """ Show only the first 10 words for long post bodies. """
      if len(obj.body) < 100:
          return obj.body
      else:
          return " ".join(obj.body.split(" ")[0:10]) + ".."
```

That gives us this:



Posts

Title	Body	Tags
First Post	This is my first blog post. It is a..	first
Second Post	I've come to reconsider whether this blog really is the..	fail

2 results

4.3 Permissions

Very few sites really want to allow just anybody to edit content, and the sanest way of managing who can do what is by using permissions. Smartmin uses permissions and groups throughout to help you manage this functionality easily.

So far we've enabled anybody to create Posts, so as a first step let's required that only authenticated users (and admins) who have the proper privileges can access our views.

Thankfully, that's a one line change, we just need to add the `permissions=True` attribute to our CRUDL object:

```
class PostCRUDL(SmartCRUDL):
    model = Post
```

(continues on next page)

(continued from previous page)

```
    permissions = True

    # .. view definitions ..
```

Now when we try to view any of the CRUDL pages for our Post object we are redirected to a login page.

4.4 Smartmin Views

Smartmin comes with five views defined. One each for, Create, Read, Update, Delete and List on your custom model objects. Each of these views can be used individually as you would any Django class based view, or they can be used together in a CRUDL object.

Whenever using a CRUDL object, you are implicitely creating views for each of the CRUDL actions. For example, if you define a CRUDL for a Post object, as so:

```
class PostCRUDL(SmartCRUDL):
  model = Post
```

You are implicitely creating views for the CRUDL operations. If you'd rather only include some actions, that is easily done by setting the `actions` tuple:

```
class PostCRUDL(SmartCRUDL):
  actions = ('create', 'update', 'list')
  model = Post
```

Now, only the views to create, update and list objects will be created and wired.

You can also choose to override any of the views for a CRUDL, without losing all the URL magic. The SmartCRUDL object will use any inner class of itself that is named the same as the action:

```
class PostCRUDL(SmartCRUDL):
  actions = ('create', 'update', 'list')
  model = Post

  class List(SmartListView):
    fields = ('title', 'body')
```

When created, the List class will be used instead of the default Smartmin generated list view. This let's you easily override behavior as you see fit.

4.5 SmartCreateView

The SmartCreateView provides a simple and quick way to create form pages for creating your objects. The following attributes are available.

fields

Defines which fields should be displayed in our form, and in what order.

Note that if you'd like to have this be set at runtime, you can do so by overriding the `derive_fields` method

permission

Let's you set what permission the user must have in order to view this page.

grant_permissions

A list or tuple which defines what object level permissions should be granted to the logged in user upon creating this object. This can let you use permissions at an entity level, letting smartmin automatically grant the privileges appropriately.

template_name

The name of the template used to render this view. By default, this is set to `smartmin/create.html` but you can override it to whatever you'd like.

4.5.1 Overriding

You can also extend your SmartCreateView to modify behavior at runtime, the most common methods to override are listed below.

pre_save

Called after our form has been validated and cleaned and our object created, but before the object has actually been saved. This can be a good place to add derived attributes to your model.

post_save

Called after our object has been saved. Sometimes used to add permissions.

get_success_url

Returns what URL the page should go to after the form has been successfully submitted.

4.6 SmartReadView

The SmartReadView provides a simple readonly view of your object. This is essentially just a detail view.

fields

Defines which fields should be displayed for the object, and in what order:

```
class PostCRUDL(SmartCRUDL):
  model = Post

  class Read(SmartReadView):
      fields = ('title', 'tags', 'body')
```

Note that if you'd like to have this be set at runtime, you can do so by overriding the `derive_fields` method.

permission

Let's you set what permission the user must have in order to view this page.

template_name

The name of the template used to render this view. By default, this is set to `smartmin/create.html` but you can override it to whatever you'd like.

4.7 SmartUpdateView

The SmartUpdateView provides a simple and quick way to create form pages for updating objects. The following attributes are available.

fields

Defines which fields should be displayed in our form, and in what order:

```
class PostCRUDL(SmartCRUDL):
  model = Post

  class Update(SmartUpdateView):
      fields = ('title', 'tags', 'body')
```

Note that if you'd like to have this be set at runtime, you can do so by overriding the `derive_fields` method.

readonly

A tuple of field names for fields which should be displayed in the form, but which should be not be editable:

```
class PostCRUDL(SmartCRUDL):
  model = Post

  class Update(SmartUpdateView):
      readonly = ('tags',)
```

permission

Let's you set what permission the user must have in order to view this page.

template_name

The name of the template used to render this view. By default, this is set to `smartmin/create.html` but you can override it to whatever you'd like.

4.7.1 Overriding

You can also extend your SmartCreateView to modify behavior at runtime, the most common methods to override are listed below.

pre_save

Called after our form has been validated and cleaned and our object created, but before the object has actually been saved. This can be a good place to add derived attributes to your model.

post_save

Called after our object has been saved. Sometimes used to add permissions.

get_success_url

Returns what URL the page should go to after the form has been successfully submitted.

4.8 SmartDeleteView

The SmartDeleteView provides a simple page for asking for confirmation and deleting an object.

cancel_url

What URL the user should be brought to if they choose to cancel deleting this object.

redirect_url

What URL the user should be brought to if they go through with deleting the object

4.9 SmartListView

The SmartListView provides the most bang for the buck, and was largely inspired by Django's own admin list API. It has the following options:

fields

Defines which fields should be displayed in the list, and in what order.

The order of precedence to get the field value is first the View, by calling `get_${field_name}`, then the object itself. This means you can easily define custom formatting of a field for a list view by simply declaring a new method:

```
class PostListView(SmartListView):
  model = Post
  fields = ('title', 'body')

  def get_body(self, obj):
    # only display first 50 characters of body
    return obj.body[:50]
```

Note that if you'd like to have this be set at runtime, you can do so by overriding the `derive_fields` method

link_fields

Defines which fields should be turned into links to the object itself. By default, this is just the first item in the field list, but you can change it as you wish, including having more than one field. By default Smartmin will generate a link to the 'read' view for the object.

You can modify what the link is by overriding `lookup_field_link`:

```
class List(SmartListView):
  model = Country
  link_fields = ('name', 'currency')

  def lookup_field_link(self, context, field, obj):
    # Link our name and currency fields, each going to their own place
    if field == 'currency':
      return reverse('locales.currency_update', args=[obj.currency.id])
    else:
      return reverse('locales.country_update', args=[obj.id])
```

Note that if you'd like to have this be set at runtime, you can do so by overriding the `derive_link_fields` method

search_fields

If set, then enables a search box which will search across the passed in fields. This should be a list or tuple. The values are used to build up a Q object, so you can specify standard Django manipulations if you'd like:

```
class List(SmartListView):
  model = User
  search_fields = ('username__icontains','first_name__icontains', 'last_name__
↪icontains')
```

Alternatively, if you want to customize the search even further, you can modify how the query is built by overriding the `derive_queryset` method.

template_name

The name of the template used to render this view. By default, this is set to `smartmin/list.html` but you can override it to whatever you'd like.

add_button

Whether an add button should be automatically added for this list view. Generally used with CRUDL.

4.10 HTML5 Boilerplate

Smartmin comes ready to go straight out the box, including using a recent version of the most excellent HTML5 boilerplate so you can build a standards compliant and optimized website.

4.11 File Layout

Again, Smartmin defines a layout for your files, things will just be easier if you agree:

```
/static/ - all static files

  /css/  - any css stylesheets
    reset.css - this is the HTML5 boilerplate reset
    smartmin_styles.css - styles specific to smartmin functionality
    styles.css - this can be any of your custom styles

  /img/ - any static images

  /js/ - your javascript files
    /libs/ - external javascript libraries you depend on
```

4.12 Blocks

All pages rendered by smartmin inherit from the `base.html`, which contains the following blocks:

title This is the title of the page displayed in the `<title>` tag

extrastyle Any extra stylesheets or CSS you want to include on your page. Surround either in `<style>` or `<link>`

login The login block, will either display a login link or the name of the logged in user with a logout link

messages Any messages, or 'flashes', pushed in by the view.

content The primary content block of the page, this is the main body.

footer Any footer treatment.

extrascript Any extra javascript you wanted included, this is put at the bottom of the page

4.13 Customizing

You can, and shoud customize the `base.html` to your needs. The only thing smartmin depends on is having the content, extrascript and extrastyle blocks available.

4.14 Group Creation

Smartmin believes in using groups and permissions to manage access to all your site resources. Managing these can be a bare in Django however as the permission and group ids can change out from under you, making fixtures ill suited.

Smartmin addresses this by letting you define your groups and the permissions for those groups within your `settings.py`. Every time you run `python manage.py syncdb`, smartmin will examine your settings and models and make sure all the permissions in sync.

4.15 Defining Permissions

You can define permissions per object or alternatively for all objects. Here we create the default smartmin 'create', 'read', 'update', 'list', 'delete' permissions on all objects:

```
PERMISSIONS = {
  '*': ('create', # can create an object
        'read',   # can read an object, viewing it's details
        'update', # can update an object
        'delete', # can delete an object,
        'list'),  # can view a list of the objects
}
```

You can also add specific permissions for particular objects if you'd like by specifying the path to the object and the verb you'd like to use for the permission:

```
PERMISSIONS = {
  'fruits.apple': ('pick',)
}
```

Smartmin will name this permission automatically in the form: `fruits.apple_pick`. Note that this is slightly different than standard Django convention, which usually uses the order of 'verb'->'object', but Smartmin does this on purpose so that URL reverse names and permissions are named identically.

4.16 Assigning Permissions for Groups

It is usually most convenient to assign users to particular groups, and assign permissions per group. Smartmin makes this easy by allowing you to define the groups that exist in your system, and the permissions granted to them via the settings file. Here's an example:

```
GROUP_PERMISSIONS = {
  "Administrator": ('auth.user_create', 'auth.user_read', 'auth.user_update',
                    'auth.user_delete', 'auth.user_list'),
  "Fruit Picker": ('fruits.apple_list', 'fruits.apple_pick'),
}
```

Again, these groups and permissions will automatically be created and granted when you run `python manage.py syncdb`

If you want a particular user to have *ALL* permissions on an object, you can do so by using a wildcard format. For example, to have the Administrator group above be able to perform any action on the user object, you could use: `auth.user.*`:

```
GROUP_PERMISSIONS = {
  "Administrator": ('auth.user.*', ),
  "Fruit Picker": ('fruits.apple_list', 'fruits.apple_pick'),
}
```

4.17 Permissions on Views

Smartmin supports gating any view using permissions. If you are using a CRUDL object, all you need to do is set `permissions = True`:

```
class FruitCRUDL(SmartCRUDL):
  model = Fruit
  permissions = True
```

But you can also customize permissions on a per view basis by setting the permission on the View itself:

```
class FruitListView(SmartListView):
  model = Fruit
  permission = 'fruits.apple_list'
```

The user will automatically be redirected to a login page if they try to access this view.

4.18 Users

Smartmin provides some views and utilities to facilitate managing users. This includes views to help users when they forget their passwords, functionality to force password expiration, complexity requirements and prevent users from repeating passwords.

4.19 Configuration

First and foremost, you'll want to include *smartmin.users* in your INSTALLED_APPS, and include smartmin.urls in your project urls.py.

If you intend to use the password expiration feature, you will also need to add the ChangePasswordMiddleware to your *MIDDLEWARE_CLASSES* setting *smartmin.users.middleware.ChangePasswordMiddleware*. (best if last)

The following variables can be set in your settings.py to change various behavior:

> USER_FAILED_LOGIN_LIMIT = The number of times a user can fail a login with an incorrect password before being locked out. (default value is 5)
>
> USER_LOCKOUT_TIMEOUT = The number of minutes that a user must wait before trying to log in again after reaching the limit above. If set to -1 or 0, the user is permanently locked out until an administrator resets the password. (default value is 10)
>
> USER_ALLOW_EMAIL_RECOVERY = Whether users are able to recover their password via a token sent to their email address. (default is True)
>
> USER_PASSWORD_EXPIRATION = How many days before a user's password expires and they need to choose a new one. If set to 0 or a negative value then there is no expiration. (default is -1)

USER_PASSWORD_REPEAT_WINDOW = The window whereby past passwords must not repeat. For example, if set to 365, users will not be able to set a password that has been used in the past year. If set to 0 or a negative value, then no enforcement of repetition is made. (default is -1)

4.20 Miscellaneous Utilities

We've included a few bonus features that we find useful when developing django apps.

4.20.1 Collect SQL Command

This is a management command to extract SQL operations from your Django migrations and organize them into several master SQL scripts:

```
python manage.py collect_sql
```

This will extract any SQL statements passed to RunSQL operations and write them to `current_indexes.sql`, `current_triggers.sql` and `current_functions.sql`.

4.20.2 Migrate Manual Command

This is a management command to make it easier to run long-running Django data migrations manually. To make a migration compatible with this command, include a function called `apply_manual` which takes no parameters:

```
python manage.py migrate_manual flows 0123
```

This will manually run the migration in the flows app with the prefix 0123.

4.20.3 Django Compressor

Smartmin already comes with django-compressor support. The default `base.html` template will wrap your CSS and JS in `{% compress %}` tags in order to optimize your page load times.

If you want to enable this, you'll just need to add `compressor` to your `INSTALLED_APPS` in `settings.py`:

```
INSTALLED_APPS = (
  # .. other apps ..
  'compressor',
)
```

And change the commented out `{# compress #}` tags in `base.html` to be valid, ie: `{% compress %}`.

4.20.4 PDB Template Tag

We all love `pdb.set_trace()` to help us debug problems, but sometimes you want to do the same thing in a template. The smartmin template tags include just that:

```
{% pdb %}
```

Will throw you into a pdb session when it hits that tag. You can examine variables in the session (including the request) and debug your template live.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search