django-admin2 Documentation

Release 0.5.2

Daniel Greenfeld

Sep 27, 2017

Contents

1 Features **3**

2 Basic API **5**
2.1 Content 5
2.2 Reference 15

3 Indices and tables **33**

Warning: This project is currently in an **alpha** state and currently not meant for real projects.

One of the most useful parts of `django.contrib.admin` is the ability to configure various views that touch and alter data. django-admin2 is a complete rewrite of that library using modern Class-Based Views and enjoying a design focused on extendibility and adaptability. By starting over, we can avoid the legacy code and make it easier to write extensions and themes.

django-admin2 aims to replace django's built-in admin that lives in `django.contrib.admin`. Come and help us, read the *Design* and *Contributing* pages, and visit the GitHub project.

This project is intentionally backwards-incompatible with `django.contrib.admin`.

CHAPTER 1

Features

- Rewrite of the Django Admin backend
- Drop-in themes
- Built-in RESTful API

CHAPTER 2

Basic API

If you've worked with Django, this implementation should look familiar:

```
# myapp/admin2.py
# Import your custom models
from django.contrib.auth.forms import UserCreationForm, UserChangeForm
from django.contrib.auth.models import User

from .models import Post, Comment

from djadmin2.site import djadmin2_site
from djadmin2.types import ModelAdmin2


class UserAdmin2(ModelAdmin2):
    create_form_class = UserCreationForm
    update_form_class = UserChangeForm


#  Register each model with the admin
djadmin2_site.register(Post)
djadmin2_site.register(Comment)
djadmin2_site.register(User, UserAdmin2)
```

Content

Installation

Adding django-admin2 to your project

Use pip to install from PyPI:

```
pip install django-admin2
```

Add djadmin2 and rest_framework to your settings file:

```
INSTALLED_APPS = (
    ...
    'djadmin2',
    'djadmin2.themes.djadmin2theme_bootstrap3', # for the default theme
    'rest_framework', # for the browsable API templates
    ...
)

REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 10
}
ADMIN2_THEME_DIRECTORY = "djadmin2theme_bootstrap3"
```

Add djadmin2 urls to your URLconf:

```
# urls.py
from django.conf.urls import include

from djadmin2.site import djadmin2_site

djadmin2_site.autodiscover()

urlpatterns = [
  ...
  url(r'^admin2/', include(djadmin2_site.urls)),
]
```

Development Installation

See *Contributing*.

Migrating from 0.6.x

- The default theme has been updated to bootstrap3, be sure to replace your reference to the new one.
- Django rest framework also include multiple pagination system, the only one supported now is the PageNumberPagination.

Therefore, your *settings* need to include this:

```
# In settings.py
INSTALLED_APPS += ('djadmin2.themes.djadmin2theme_bootstrap3',)
ADMIN2_THEME_DIRECTORY = "djadmin2theme_bootstrap3"

REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 10
}
```

The default admin2 site has move into djadmin2.site make sure your use the news djadmin2_site in your urls.py:

```
# urls.py
from django.conf.urls import include

from djadmin2.site import djadmin2_site

djadmin2_site.autodiscover()

urlpatterns = [
  ...
  url(r'^admin2/', include(djadmin2_site.urls)),
]
```

Migrating from 0.5.x

Themes are now defined explicitly, including the default theme. Therefore, your *settings* need to include this:

```
# In settings.py
INSTALLED_APPS += ('djadmin2.themes.djadmin2theme_default',)
ADMIN2_THEME_DIRECTORY = "djadmin2theme_default"
```

Contributing

> **Warning:** Before you begin working on your contribution, please read and become familiar with the design of `django-admin2`. The design document should hopefully make it clear what our constraints and goals are for the project.

Setup

Fork on GitHub

Before you do anything else, login/signup on GitHub and fork **django-admin2** from the GitHub project.

Clone your fork locally

If you have git-scm installed, you now clone your git repo using the following command-line argument where <my-github-name> is your account name on GitHub:

```
git clone git@github.com:<my-github-name>/django-admin2.git
```

Local Installation

1. Create a virtualenv (or use virtualenvwrapper). Activate it.
2. cd into django-admin2
3. type `$ pip install -r requirements.txt`
4. type `$ python setup.py develop`

Try the example projects

1. cd into example/
2. create the database: `$ python manage.py migrate`
3. run the dev server: `$ python manage.py runserver`

Issues!

The list of outstanding **django-admin2** feature requests and bugs can be found on our on our GitHub issue tracker. Pick an unassigned issue that you think you can accomplish, add a comment that you are attempting to do it, and shortly your own personal label matching your GitHub ID will be assigned to that issue.

Feel free to propose issues that aren't described!

Tips

1. **starter** labeled issues are deemed to be good low-hanging fruit for newcomers to the project, Django, or even Python.
2. **doc** labeled issues must only touch content in the docs folder.
3. Since this project will live on inheritance, all views are Class-Based.
4. Familiarize yourself with the project design document.

Setting up topic branches and generating pull requests

Note: This is our way of describing our version of git-flow.

While it's handy to provide useful code snippets in an issue, it is better for you as a developer to submit pull requests. By submitting pull request your contribution to django-admin2 will be recorded by Github.

In git it is best to isolate each topic or feature into a "topic branch". While individual commits allow you control over how small individual changes are made to the code, branches are a great way to group a set of commits all related to one feature together, or to isolate different efforts when you might be working on multiple topics at the same time.

While it takes some experience to get the right feel about how to break up commits, a topic branch should be limited in scope to a single `issue` as submitted to an issue tracker.

Also since GitHub pegs and syncs a pull request to a specific branch, it is the **ONLY** way that you can submit more than one fix at a time. If you submit a pull from your master branch, you can't make any more commits to your master without those getting added to the pull.

To create a topic branch, its easiest to use the convenient `-b` argument to `git checkout`:

```
git checkout -b fix-broken-thing
Switched to a new branch 'fix-broken-thing'
```

You should use a verbose enough name for your branch so it is clear what it is about. Now you can commit your changes and regularly merge in the upstream develop as described below.

When you are ready to generate a pull request, either for preliminary review, or for consideration of merging into the project you must first push your local topic branch back up to GitHub:

```
git push origin fix-broken-thing
```

Now when you go to your fork on GitHub, you will see this branch listed under the "Source" tab where it says "Switch Branches". Go ahead and select your topic branch from this list, and then click the "Pull request" button.

Your pull request should be applied to the **develop** branch of django-admin2. Be sure to change from the default of `master` to `develop`.

Next, you can add a comment about your branch. If this in response to a submitted issue, it is good to put a link to that issue in this initial comment. The repo managers will be notified of your pull request and it will be reviewed (see below for best practices). Note that you can continue to add commits to your topic branch (and push them up to GitHub) either if you see something that needs changing, or in response to a reviewer's comments. If a reviewer asks for changes, you do not need to close the pull and reissue it after making changes. Just make the changes locally, push them to GitHub, then add a comment to the discussion section of the pull request.

Pull upstream changes into your fork regularly

django-admin2 is advancing quickly. It is therefore critical that you pull upstream changes from master into your fork on a regular basis. Nothing is worse than putting in a day of hard work into a pull request only to have it rejected because it has diverged too far from master.

To pull in upstream changes:

```
git remote add upstream https://github.com/twoscoops/django-admin2.git
git pull upstream develop
```

For more info, see http://help.github.com/fork-a-repo/

Advanced git users: Pull with rebase

This will pull and then reapply your work on top of the upcoming changes:

```
git pull --rebase upstream develop
```

It saves you from an extra merge, keeping the history cleaner, but it's potentially dangerous because you're rewriting history. For more info, see http://gitready.com/advanced/2009/02/11/pull-with-rebase.html

How to get your pull request accepted

We want your submission. But we also want to provide a stable experience for our users and the community. Follow these rules and you should succeed without a problem!

Run the tests!

Before you submit a pull request, please run the entire django-admin2 test suite via:

```
python runtests.py
```

The first thing the core committers will do is run this command. Any pull request that fails this test suite will be **immediately rejected**.

If you add code/views you need to add tests!

We've learned the hard way that code without tests is undependable. If your pull request reduces our test coverage because it lacks tests then it will be **rejected**.

For now, we use the Django Test framework (based on unittest).

Also, keep your tests as simple as possible. Complex tests end up requiring their own tests. We would rather see duplicated assertions across test methods then cunning utility methods that magically determine which assertions are needed at a particular stage. Remember: *Explicit is better than implicit*.

You don't need to run the whole test suite during development in order to make the test cycles a bit faster. Just pass in the specific tests you want to run to `runtests.py` as you would do with the `django-admin.py test` command. Examples:

```
# only run the tests from application ``blog``
python runtests.py blog

# only run testcase class ``Admin2Test`` from app ``djadmin2``
python runtests.py djadmin2.Admin2Test

# run all tests from application ``blog`` and the test named
# ``test_register`` on the ``djadmin2.Admin2Test`` testcase.
python runtests.py djadmin2.Admin2Test.test_register blog
```

Don't mix code changes with whitespace cleanup

If you change two lines of code and correct 200 lines of whitespace issues in a file the diff on that pull request is functionally unreadable and will be **immediately rejected**. Whitespace cleanups need to be in their own pull request.

Keep your pull requests limited to a single issue

django-admin2 pull requests should be as small/atomic as possible. Large, wide-sweeping changes in a pull request will be **rejected**, with comments to isolate the specific code in your pull request. Some examples:

1. If you are making spelling corrections in the docs, don't modify the settings.py file (pydanny is guilty of this mistake).
2. If you are fixing a view don't '*cleanup*' unrelated views. That cleanup belongs in another pull request.
3. Changing permissions on a file should be in its own pull request with explicit reasons why.

Best Practices

As much as possible, we follow the advice of the Two Scoops of Django book. Periodically the book will be referenced either for best practices or as a blunt object by the project lead in order to end bike-shedding.

Python

Follow PEP-0008 and memorize the Zen of Python:

```
>>> import this
```

Please keep your code as clean and straightforward as possible. When we see more than one or two functions/methods starting with *_my_special_function* or things like *__builtins__.object = str* we start to get worried. Rather than try and figure out your brilliant work we'll just **reject** it and send along a request for simplification.

Furthermore, the pixel shortage is over. We want to see:

- *options* instead of *opts*
- *model_name* instead of *model*
- *my_function_that_does_things* instead of *mftdt*

Templates

Follow bootstrap's coding standards for HTML and CSS. Use two spaces for indentation, and write so the templates are readable (not for the generated html).

Internationalize

Any new text visible to the user must be internationalized.

How pull requests are checked, tested, and done

First we pull the code into a local branch:

```
git checkout develop
git checkout -b <submitter-github-name>-<submitter-branch> develop
git pull git://github.com/<submitter-github-name>/django-admin2.git <submitter-branch>
↪ <branch-name>
```

Then we run the tests:

```
coverage run runtests.py
coverage report
```

We do the following:

1. Any test failures or the code coverage drops and the pull request is rejected.
2. We open up a browser and make sure it looks okay.
3. We check the commit's code changes and make sure that they follow our rules.

We finish with a merge and push to GitHub:

```
git checkout develop
git merge <branch-name>
git push origin develop
```

Design

Constraints

This section outlines the design constraints that django-admin2 follows:

1. There will be nothing imported from `django.contrib.admin`.

2. The original bootstrap/ theme shall contain no UI enhancements beyond the original `django.contrib.admin` UI. (However, future themes can and should be experimental.)
3. External package dependencies are allowed but should be very limited.
4. Building a django-admin2 theme cannot involve learning Python, which explains why we are not using tools like django-crispy-forms. (One of our goals is to make it easier for designers to explore theming django-admin2).

Backend Goals

Rather than creating yet another project that skins `django.contrib.admin`, our goal is to rewrite `django.contrib.admin` from the ground up using Class-Based Views, better state management, and attention to all the lessons learned from difficult admin customizations over the years.

While the internal API for the backend may be drastically different, the end goal is to achieve relative parity with existing functionality in an extendable way:

- Relative functional parity with `django.contrib.admin`. This is our desire to replicate much of the existing functionality, but not have to worry too much about coding ourselves into an overly-architected corner.
- Ability handle well under high load situations with many concurrent users. This is diametrically opposite from *django.contrib.admin* which doesn't work well in this regard.
- Extensible presentation and data views in such a way that it does not violate Constraint #4. To cover many cases, we will provide instructions on how to use the REST API to fetch data rather than create overly complex backend code.
- Create an architecture that follows the "*Principle of least surprise*". Things should behave as you expect them to, and you should be blocked from making dangerous mistakes. This is the reason for the ImmutableAdmin type.

Clean code with substantial documentation is also a goal:

1. Create a clearly understandable/testable code base.
2. All classes/methods/functions documented.
3. Provide a wealth of in-line code documentation.

REST API Goals

There are a lot of various cases that are hard to handle with pure HTML projects, but are trivial to resolve if a REST API is available. For example, using unmodified `django.contrib.admin` on projects with millions of database records combined with foreign key lookups. In order to handle these cases, rather than explore each edge case, `django-admin2` provides a RESTFUL API as of version 0.2.0.

Goals:

1. Provide a extendable self-documenting API (django-rest-framework).
2. Reuse components from the HTML view.
3. Backwards compatibility: Use a easily understood API versioning system so we can expand functionality of the API without breaking existing themes.

UI Goals

1. Replicate the old admin UI as closely as possible in the bootstrap/ theme. This helps us ensure that admin2/ functionality has parity with admin/.

2. Once (1) is complete and we have a stable underlying API, experiment with more interesting UI variations.

Frequently Asked Questions

Is this intended to go into Django contrib?

No.

Reasons why it won't be going into Django core:

1. We want to rely on external dependencies

We think certain packages can do a lot of the heavy lifting for us, and rewriting them is more time taken away from fixing bugs and implementing features. Since the Django core team isn't likely to accept external dependencies, especially ones that rely on Django itself, this alone is reason enough for django-admin2 to never make it into Django contrib.

2. We want increased Speed of Development

Django is a huge project with a lot of people relying on it. The conservative pace at which any change or enhancement is accepted is usually boon to the community of developers who work with it. Also, the committee-based management system means everyone gets a voice. This means things often happen at a slow and steady pace.

However, there are times when it's good to be outside of core, especially for experimental replacements for core functionality. Working outside of Django core means we can do what we want, when we want it.

What's wrong with the Django Admin?

The existing Django Admin is a powerful tool with pretty extensive extension capabilities. That said, it does have several significant issues.

Doesn't handle a million-record foreign key relation

Say you have a million users and a model with a foreign key relation to them. You go the model detail field in the admin and you know what happens? The Django admin tries to serve out a million option links to your browser. Django doesn't handle this well, and neither does your browser. You can fix this yourself, find a third-party package to do it for you, or use django-admin2.

Yes, before release 1.0 of django-admin2 it will handle this problem for you.

Uses an early version of Class-Based Views

TODO

Very Challenging to Theme

TODO

Internationalization and localization

Refer to the Django i18n documentation to get started with internationalization (i18n).

Enabling i18n in Django

Make sure you've activated translation for your project (the fastest way is to check in your `settings.py` file if `MIDDLEWARE_CLASSES` includes `django.middleware.locale.LocaleMiddleware`).

Then compile the messages so they can be used by Django.

```
python manage.py compilemessages
```

It should get you started !

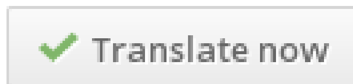
Translating django-admin2

The translation of the language files is handled using Transifex.

Improving existing translations

To check out what languages are currently being worked on, check out the Project page. If you want to help with one of the translations, open the team page by clicking on the language and request to join the team.

Now you can start translating. Open the language page, select a language resource (e.g. *djadmin2.po*).



Then select a string from the list on the left and enter a translation on the right side. Finally, click the *Save* button on the top right and you're done.

It is also possible to suggest better translations for existing ones with the *Suggest* button on the bottom.

Requesting a new language

If a language is not available on Transifex yet, you can request it with the *Request language* button on the Project page.

Using i18n in the django-admin2 project development

This section is mainly directed at

Marking strings for translation

Python code

Make sure to use ugettext or ugettext_lazy on strings that will be shown to the users, with string interpolation ("%(name_of_variable)s" instead of "%s") where needed.

Remember that all languages do not use the same word order, so try to provide flexible strings to translate !

Templates

Make sure to load the i18n tags and put `trans` tags and `blocktrans` blocks where needed.

Block variables are very useful to keep the strings simple.

Adding a new locale

```
cd djadmin2
django-admin.py makemessages -l $LOCALE_CODE
```

A new file will be created under `locale/$LOCALE_CODE/LC_MESSAGES/django.po`

Update the headers of the newly created file to match existing files and start the translation!

If you need help to adjust the *Plural-Forms* configuration in the .po file, refer to the gettext docs.

Updating existing locales

To update the language files with new strings in your .py files / templates:

```
cd djadmin2 # or any other package, for instance example/blog
django-admin.py makemessages -a
```

Then translate the files directly or upload them to Transifex.

When the translation is done, you need to recompile the new translations:

```
django-admin.py compilemessages
```

Tutorial

This is where the django-admin2 tutorial is in the process of being written. It will be analogous with Page 2 of the Django tutorial.

Reference

Themes

How To Create a Theme

A Django Admin 2 theme is merely a packaged Django app. Here are the necessary steps to create a theme called '*dandy*':

1. Make sure you have Django 1.8 or higher installed.

```
$ python -c 'import django; print(django.get_version())'
```

2. Create the package:

```
$ mkdir djadmin2theme-dandy
```

4. Create a `setup.py` module

```
$ cd djadmin2theme-dandy
$ touch setup.py
```

Then enter the following information (you will probably want to change the highlighted lines below to match your package name):

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from setuptools import setup
import re
import os
import sys

def get_packages(package):
    """
    Return root package and all sub-packages.
    """
    return [dirpath
            for dirpath, dirnames, filenames in os.walk(package)
            if os.path.exists(os.path.join(dirpath, '__init__.py'))]

if sys.argv[-1] == 'publish':
    os.system("python setup.py sdist upload")
    print("You probably want to also tag the version now:")
    print("  git tag -a %s -m 'version %s'" % (version, version))
    print("  git push --tags")
    sys.exit()

setup(
    name='djadmin2theme-dandy',
    version=0.1.0,
    description="A dandy theme for django-admin2.",
    long_description="A dandy theme for django-admin2.",
    classifiers=[
        "Environment :: Web Environment",
        "Framework :: Django",
        "License :: OSI Approved :: BSD License",
        "Operating System :: OS Independent",
        "Programming Language :: Python",
        "Topic :: Internet :: WWW/HTTP",
        "Topic :: Internet :: WWW/HTTP :: Dynamic Content",
        "Topic :: Software Development :: Libraries :: Python Modules",
```

```
    ],
    keywords='django,djadmin2',
    author="Your Name Here",
    author_email='Your Email Here',
    url='http://github.com/your-repo-here',
    license='MIT',
    packages=get_packages('djadmin2_dandy'),
    include_package_data=True,
    install_requires=[
        'django-admin2>=0.5.0',
        ],
    zip_safe=False,
)
```

5. Create a Django App called 'dandy' and go inside.

```
$ django-admin.py startapp djadmin2theme_dandy
$ cd djadmin2theme_dandy
```

Note: Why is the djadmin2theme prefix used everywhere?

Makes it easy to identify what is a theme and what is not.

Also we don't pollute our Python namespaces and Django app cache with names like 'foundation', 'storefront', or other useful names.

6. Add a `static/` file directory set:

```
$ mkdir -p static/djadmin2theme_dandy/{js,css,img}
```

These directories are where the dandy theme's custom CSS, JavaScript, and Image files are placed.

7. Add a `templates/djadmin2theme_dandy` directory:

```
$ mkdir -p templates/djadmin2theme_dandy
```

Inside of `templates/djadmin2theme_dandy` is where the templates for dandy are defined.

Now you can start working on templates and static files!

Installing the custom theme

In the settings module, place the theme right after djadmin2 (change the highlighted lines to your package's name):

```
########### DJANGO-ADMIN2 CONFIGURATION
ADMIN2_THEME_DIRECTORY = "djadmin2theme_dandy"
INSTALLED_APPS += (
    'djadmin2theme_dandy'
)
########### END DJANGO-ADMIN2 CONFIGURATION
```

Views and their Templates

See *Built-In Views*

Available Themes

Currently, only the default twitter bootstrap-powered "djadmin2.themes.djadmin2theme_bootstrap3" theme exists. The goal of this theme is to replicate the original Django admin UI functionality as closely as possible. This helps us ensure that we are not forgetting any functionality that Django users might be dependent on.

If you'd like to experiment with UI design that differs from the original Django admin UI, please create a new theme. It would be great to have at least 1 experimental theme!

Future

Keep in mind that this project is an experiment just to get our ideas down. We are looking at other similar projects to see if we can merge or borrow things.

RESTful API

django-admin2 comes with a builtin REST-API for accessing all the resources you can get from the frontend via JSON.

The API can be found at the URL you choose for the admin2 and then append `api/v0/`.

If the API has changed in a backwards-incompatible way we will increase the API version to the next number. So you can be sure that you're frontend code should keep working even between updates to more recent django-admin2 versions.

However currently we are still in heavy development, so we are using `v0` for the API, which means is subject to change and being broken at any time.

Actions

Actions are defined to work on a single view type. Currently, actions are only implemented against the `ModelListView`. This view contains the default `DeleteSelectedAction` method, which in end functionality mirrors `django.contrib.admin.delete_selected`.

However, under the hood, django-admin2's actions work very differently. Instead of functions with assigned attributes, they can either be functions or full fledged objects. Which means you can more easily extend them to suit your needs.

The documentation works off a simple set of models, as listed below:

```
# blog/models.py
from django.db import models

STATUS_CHOICES = (
    ('d', 'Draft'),
    ('p', 'Published'),
    ('w', 'Withdrawn'),
)


class Post(models.Model):
    title = models.CharField(max_length=255)
```

```
    body = models.TextField()
    status = models.CharField(max_length=1, choices=STATUS_CHOICES)

    def __unicode__(self):
        return self.title


class Comment(models.Model):
    post = models.ForeignKey(Post)
    body = models.TextField()

    def __unicode__(self):
        return self.body
```

Writing List Actions

The basic workflow of Django's admin is, in a nutshell, "select an object, then change it." This works well for a majority of use cases. However, if you need to make the same change to many objects at once, this workflow can be quite tedious.

In these cases, Django's admin lets you write and register "actions" – simple functions that get called with a list of objects selected on the change list page.

If you look at any change list in the admin, you'll see this feature in action; Django ships with a "delete selected objects" action available to all models. Using our sample models, let's pretend we wrote a blog article about Django and our mother put in a whole bunch of embarressing comments. Rather than cherry-pick the comments, we want to delete the whole batch.

In our blog/admin.py module we write:

```
from djadmin2.actions import BaseListAction
from djadmin2.site import djadmin2_site
from djadmin2.types import ModelAdmin2

from .models import Post, Comment

class DeleteAllComments(BaseListAction):

    description = 'Delete selected items'
    default_template_name = 'actions/delete_all_comments_confirmation.html'
    success_message = 'Successfully deleted %d %s' # first argument - items count,␣
↪second - verbose_name[_plural]

    def process_queryset(self):
        """Every action must provide this method"""
        self.get_queryset().delete()


def custom_function_action(request, queryset):
    print(queryset.count())

custom_function_action.description = 'Do other action'

class PostAdmin(ModelAdmin2):
    actions = [DeleteAllComments, custom_function_action]
```

```
djadmin2_site.register(Post, PostAdmin)
djadmin2_site.register(Comment)
```

> **Warning:** The "delete selected objects" action uses QuerySet.delete() for efficiency reasons, which has an important caveat: your model's delete() method will not be called.
>
> If you wish to override this behavior, simply write a custom action which accomplishes deletion in your preferred manner – for example, by calling `Model.delete()` for each of the selected items.
>
> For more background on bulk deletion, see the documentation on object deletion.

Read on to find out how to add your own actions to this list.

Forms

Replicating *django.contrib.admin*'s user management

If you have users, it's assumed you will have a Django app to manage them, called something like *accounts*, *users*, or *profiles*. For this exercise, we'll assume the app is called *accounts*.

Step 1 - The admin2.py module

In the *accounts* app, create an `admin2.py` module.

Step 2 - Web Integration

Enter the following code in `accounts/admin2.py`:

```
# Import the User and Group model from django.contrib.auth
from django.contrib.auth import get_user_model
from django.contrib.auth.models import Group

from djadmin2.site import djadmin2_site
from djadmin2.forms import UserCreationForm, UserChangeForm
from djadmin2.types import ModelAdmin2

# fetch the User model
User = get_user_model()

# Incorporate the
class UserAdmin2(ModelAdmin2):
    create_form_class = UserCreationForm
    update_form_class = UserChangeForm

djadmin2_site.register(User, UserAdmin2)
djadmin2_site.register(Group)
```

Done! The User and Group controls will appear in your django-admin2 dashboard.

Well... almost. We still need to incorporate the API components.

Step 3 - API Integration

Change `accounts/admin2.py` to the following:

```
# Import the User and Group model from django.contrib.auth
from django.contrib.auth import get_user_model
from django.contrib.auth.models import Group

from rest_framework.relations import PrimaryKeyRelatedField

import djadmin2

# fetch the User model
User = get_user_model()


# Serialize the groups
class GroupSerializer(Admin2APISerializer):
    permissions = PrimaryKeyRelatedField(many=True)

    class Meta:
        model = Group

# The GroupAdmin2 object is synonymous with GroupAdmin
class GroupAdmin2(djadmin2.ModelAdmin2):
    api_serializer_class = GroupSerializer


# Serialize the users, excluding password data
class UserSerializer(djadmin2.apiviews.Admin2APISerializer):
    user_permissions = PrimaryKeyRelatedField(many=True)

    class Meta:
        model = User
        exclude = ('passwords',)


# The UserAdmin2 object is synonymous with UserAdmin
class UserAdmin2(djadmin2.ModelAdmin2):
    create_form_class = UserCreationForm
    update_form_class = UserChangeForm

    api_serializer_class = UserSerializer

djadmin2.default.register(User, UserAdmin2)
djadmin2.default.register(Group, GroupAdmin2)
```

Things to Do

- Consider breaking the user management reference into more steps
- Create default UserAdmin2 and GroupAdmin2 classes
- Demonstrate how to easy it is to customize and HTML5-ize forms
- Demonstrate how easy it is to customize widgets

Permissions

Permissions are handled on a per view basis. So basically each admin view can hold its own permissions. That way you are very flexible in defining who is allowed to access which view. For example, the edit view might need some totally different permission checks then the delete view. However the add view has nearly the same requirements as the edit view, you just also need to have on extra permission. All those scenarios can be handled very easily in **django-admin2**.

Since the permission handling is centered around the specific views, this is the place where you attach the permission checking logic to. You can assign one or more permission backends to a view by setting the `permission_classes` attribute:

```
from django.views import generic
from djadmin2.viewmixins import Admin2Mixin
from djadmin2 import permissions


class MyView(Admin2Mixin, generic.TemplateView):
    permission_classes = (
        permissions.IsStaffPermission,
        permissions.ModelViewPermission)
```

See the following sections on which permission classes ship with **django-admin2**, ready to use and how you can roll your own.

Built-in permission classes

You can use the following permission classes directly in you views.

class `djadmin2.permissions.`**`IsStaffPermission`**
: It ensures that the user is authenticated and is a staff member.

class `djadmin2.permissions.`**`IsSuperuserPermission`**
: It ensures that the user is authenticated and is a superuser. However it does not check if the user is a staff member.

class `djadmin2.permissions.`**`ModelViewPermission`**
: Checks if the user has the `<app>.view_<model>` permission.

class `djadmin2.permissions.`**`ModelAddPermission`**
: Checks if the user has the `<app>.add_<model>` permission.

class `djadmin2.permissions.`**`ModelChangePermission`**
: Checks if the user has the `<app>.change_<model>` permission.

class `djadmin2.permissions.`**`ModelDeletePermission`**
: Checks if the user has the `<app>.delete_<model>` permission.

Writing your own permission class

If you need it, writing your own permission class is really easy. You just need to subclass the `djadmin2.permissions.BasePermission` class and overwrite the `has_permission()` method that implements the desired permission checking. The arguments that the method takes are pretty self explanatory:

`request` That is the request object that was sent to the server to access the current page. This will usually have the `request.user` attribute which you can use to check for user based permissions.

`view` The `view` argument is the instance of the class based view that the user wants to access.

obj This argument is optional and will only be given if an object-level permission check is performed. Take this into account if you want to support object-level permissions, or ignore it otherwise.

Based on these arguments should the `has_permission` method than return either `True` if the permission shall be granted or `False` if the access to the user shall be diened.

Here is an example implementation of a custom permission class:

```
from djadmin2.permissions import BasePermission

class HasAccessToSecretInformationPermission(BasePermission):
    '''
    Only allow superusers access to secret information.
    '''

    def has_permission(self, request, view, obj=None):
        if obj is not None:
            if 'secret' in obj.title.lower() and not request.user.is_superuser:
                return False
        return True
```

Permissions in Templates

Since the permission handling is designed around views, the permission checks in the template will also always access a view and return either `True` or `False` if the user has access to the given view. There is a `{{ permissions }}` variable available in the admin templates to perform these tests against a specific view.

At the moment you can check for view, add, change and delete permissions. To do so you use the provided `permissions` variable as seen below:

```
{% if permissions.has_change_permission %}
    <a href="... link to change form ...">Edit {{ object }}</a>
{% endif %}
```

This permission check will use the `ModelAdmin2` instance of the current view that was used to render the above template to find the view it should perform the permission check against. Since we test the change permission, it will use the `update_view` to check if the user has the permission to access the change page or not. If that's the case, we can safely display the link to the change page.

At the moment we can check for the following four basic permissions:

has_view_permission This will check the permissions against the current admin's `detail_view`.

has_add_permission This will check the permissions against the current admin's `create_view`.

has_change_permission This will check the permissions against the current admin's `update_view`.

has_delete_permission This will check the permissions against the current admin's `delete_view`.

Object-Level Permissions

The permission handling in templates also support checking for object-level permissions. To do so, you can use the `for_object` filter implemented in the `admin2_tags` templatetag library:

```
{% load admin2_tags %}

{% if permissions.has_change_permission|for_object:object %}
```

```
    <a href="... link to change form ...">Edit {{ object }}</a>
{% endif %}
```

Note: Please be aware, that the `django.contrib.auth.backends.ModelBackend` backend that ships with django and is used by default doesn't support object level permission. So unless you have implemented your own permission backend that supports it, the `{{ permissions.has_change_permission|for_object:object }}` will always return `False` and though will be useless.

Sometimes you have the need to perform all the permission checks in a block of template code to use one object. In that case you can *bind* an object to the permissions variable for easier handling:

```
{% load admin2_tags %}

{% with permissions|for_object:object as object_permissions %}
    {% if object_permissions.has_change_permission %}
        <a href="... link to change form ...">Edit {{ object }}</a>
    {% endif %}
    {% if object_permissions.has_delete_permission %}
        <a href="... link to delete page ...">Delete {{ object }}</a>
    {% endif %}
{% endwith %}
```

That also comes in handy if you have a rather generic template that performs some permission checks and you want it to use object-level permissions as well:

```
{% load admin2_tags %}

{% with permissions|for_object:object as object_permissions %}
    {% include "list_of_model_actions.html" with permissions=object_permissions %}
{% endwith %}
```

Checking for Permissions on Other Models

Sometimes you just need to check the permissions for that particular model. In that case, you can access its permissions like this:

```
{% if permissions.blog_post.has_view_permission %}
    <a href="...">View {{ post }}</a>
{% endif %}
```

So what we actually did here is that we just put the name of the `ModelAdmin2` that is used for the model you want to access between the `permissions` variable and the `has_view_permission`. This name will be the app label followed by the model name in lowercase with an underscore in between for ordinary django models. That way you can break free of beeing limitted to permission checks for the current `ModelAdmin2`. But that doesn't help you either if you don't know from the beginning on which model admin you want to check the permissions. Imagine the admin's index page that should show a list of all the available admin pages. To dynamically bind the permissions variable to a model admin, you can use the `for_admin` filter:

```
{% load admin2_tags %}

{% for admin in list_of_model_admins %}
    {% with permissions|for_admin:admin as permissions %}
        {% if permissions.has_add_permission %}Add another {{ admin.model_name }}{%␣
↪endif %}
```

```
    {% endwith %}
{% endfor %}
```

Dynamically Check for a Specific Permission Name

Just like you can bind a permission dynamically to a model admin, you can also specify the actual permission name on the fly. There is the `for_view` filter to do so.

```
{% load admin2_tags %}

{% with "add" as view_name %}
    {% if permissions|for_view:view_name %}
        <a href="...">{{ view_name|capfirst }} model</a>
    {% endif %}
{% endwith %}
```

That way you can avoid hardcoding the `has_add_permission` check and make the checking depended on a given template variable. The argument for the `for_view` filter must be one of the four strings: `view`, `add`, `change` or `delete`.

Views

TODO list

- Describe customization of model views
- Show how to use ModelAdmin2 inheritance so an entire project works off a custom base view.

Customizing the Dashboard view

When you first log into django-admin2, just like `django.contrib.admin` you are presented with a display of apps and models. While this is useful for developers, it isn't friendly for end-users. Fortunately, django-admin2 makes it trivial to switch out the standard dashboard view.

However, because this is the dashboard view, the method of customization and configuration is different than other django-admin2 views.

In your Django project's root URLconf module (`urls.py`) modify the code to include the commented code before the `djadmin2_site.autodiscover()`:

```
from django.conf.urls import include, url

from djadmin2.site import djadmin2_site
from djadmin2.views import IndexView


######### Begin django-admin2 customization code
# Create a new django-admin2 index view
class CustomIndexView(IndexView):

    # specify the template
    default_template_name = "custom_dashboard_template.html"

# override the default index_view
```

```
djadmin2_site.index_view = CustomIndexView
######### end django-admin2 customization code

djadmin2_site.autodiscover()

urlpatterns = [
    url(r'^admin2/', include(djadmin2_site.urls)),
    # ... Place the rest of the project URLs here
]
```

In real projects the new IndexView would likely be placed into a `views.py` module.

Note: Considering that dashboard is more intuitive of a name, perhaps the `IndexView` should be renamed `DashboardView`?

Customizing the Login view

The login view could also be customized.

In your Django project's root URLconf module (`urls.py`) modify the code to include the commented code before the `djadmin2.default.autodiscover()`:

```
from django.conf.urls import patterns, include, url

from djadmin2.site import djadmin2_site
from djadmin2.views import LoginView


######### Begin django-admin2 customization code
# Create a new django-admin2 index view
class CustomLoginView(LoginView):

    # specify the template
    default_template_name = "custom_login_template.html"

# override the default index_view
djadmin2_site.login_view = CustomLoginView
######### end django-admin2 customization code

djadmin2_site.autodiscover()

urlpatterns = patterns('',
    url(r'^admin2/', include(djadmin2_site.urls)),
    # ... Place the rest of the project URLs here
)
```

In real projects the new LoginView would likely be placed into a `views.py` module.

ModelAdmin2

The *ModelAdmin2* class is the representation of a model in the admin interface. These are stored in a file named *admin2.py* in your application. Let's take a look at a very simple example of the ModelAdmin2:

```
from .models import Post
from djadmin2.site import djadmin2_site
from djadmin2.types import ModelAdmin2

class PostAdmin(ModelAdmin2):
    pass

djadmin2_site.register(Post, PostAdmin)
```

Adding a new view

To add a new view to a ModelAdmin2, it's need add an attribute that is an instance of the *views.AdminView*.

The *view.AdminView* takes tree parameters: *url*, *view* and *name*. The *url* is expected a string for the url pattern for your view. The *view* is expected a view and *name* is an optional parameter and is expected a string that is the name of your view.

```
from .models import Post
from djadmin2 import views
from djadmin2.site import djadmin2_site
from djadmin2.types import ModelAdmin2

class PostAdmin(ModelAdmin2):
    preview_post = views.AdminView(r'^preview/$', views.PreviewPostView)

djadmin2_site.register(Post, PostAdmin)
```

Replacing an existing view

To replacing an existing admin view, it's need add an attribute with the same name that the view that you want replace:

```
from .models import Post
from djadmin2 import views
from djadmin2.site import djadmin2_site
from djadmin2.types import ModelAdmin2

class PostAdmin(ModelAdmin2):
    create_view = views.AdminView(r'^create/$', views.MyCustomCreateView)

djadmin2_site.register(Post, PostAdmin)
```

Built-In Views

Each of these views contains the list of context variables that are included in their templates.

Note: TODO: Fix the capitalization of context variables!

View Constants

The following are available in every view:

next The page to redirect the user to after login

MEDIA_URL Specify a directory where file uploads for users who use your site go

STATIC_URL Specify a directory for JavaScript, CSS and image files.

user Currently logged in user

View Descriptions

Custom Renderers

It is possible to create custom renderers for specific fields. Currently they are only used in the object list view, for example to render boolean values using icons. Another example would be to customize the rendering of dates.

Renderers

A renderer is a function that accepts a value and the field and returns a HTML representation of it. For example, the very simple builtin datetime renderer works like this:

```
def title_renderer(value, field):
    """Render a string in title case (capitalize every word)."""
    return unicode(value).title()
```

In this case the `field` argument is not used. Sometimes it useful though:

```
def number_renderer(value, field):
    """Format a number."""
    if isinstance(field, models.DecimalField):
        return formats.number_format(value, field.decimal_places)
    return formats.number_format(value)
```

You can create your renderers anywhere in your code, but it is recommended to put them in a file called `renderers.py` in your project.

Using Renderers

The renderers can be specified in the Admin2 class using the `field_renderers` attribute. The attribute contains a dictionary that maps a field name to a renderer function.

By default, some renderers are automatically applied, for example the boolean renderer when processing boolean values. If you want to suppress that renderer, you can assign `None` to the field in the `field_renderers` dictionary.

```
class PostAdmin(djadmin2.ModelAdmin2):
    list_display = ('title', 'body', 'published')
    field_renderers = {
        'title': renderers.title_renderer,
        'published': None,
    }
```

Builtin Renderers

Django's Model._meta

Currently django implements most of its behaviour that makes using models so nice using a metaclass. A metaclass is invoked when an actual class is created and can change that class' behaviour by adding or modifing its attributes and methods. This means that django is actually changing your model class at the moment when the `models.py` file of your app is loaded.

One of those changes is that the model's metaclass takes the specified `Meta` options that the model has attached to and uses it's attributes in conjunction with the model's fields to create a new attribute on the model that is called `_meta`. This is then responsible for providing an API to access all database related information like the name of the database table or a list of all available fields.

Django doesn't currently have an official documentation of the `_meta`'s semantics, however it is kind of a stable API since many projects and all the db related django internals depend on it. And this page is about documenting the semantics of `_meta`.

Trivia

The `Model._meta` attribute is an instance of the `django.db.models.options.Options` class. It gets attached to the model at that time when the model class is created by the `django.db.models.base.ModelBase` metaclass.

If `_meta` is mentioned we speak about the autogenerated `_meta` attribute that is attached to the model class.

Some clarifying: In the following text `Meta` is referring to the actual `class Meta:` definition that the app author has put inside the model class. Like here:

```
class Restaurant(models.Model):
    # ... some fields here ...

    class Meta:
        ordering = ('name',)
```

Attributes copied from `Meta`

Some of `_meta`'s attributes are just copied from the `Meta` options. The following attributes are those. Their behaviour is more detailed described in the django documentation.

`abstract` A boolean value.

See the django documentation on `abstract` for more information.

`app_label` By default it is the name of the app module that the model was created in. This can be overriden in `Meta` to make a model part of a specific app.

Also see the django documentation about `app_label`.

`db_table` Contains the name of the database table used for this model. This is either what was set on `Meta` or defaults to a string that is built from `app_label` and `model_name` seperated by an underscore. So for example the `db_table` for `django.contrib.auth.models.User` is `'auth_user'`.

Also see the django documentation about `db_table`.

`db_tablespace` See the django documentation on `db_tablespace` for more information.

get_latest_by The name of the field that should be used during ordering to make `latest()` and `earliest()` work.

Also see the django documentation about `get_latest_by`.

managed If `managed` is `True` then the `syncdb` management command will take care of creating the database tables. Defaults to `True`.

Also see the django documentation about `managed`.

order_with_respect_to See the django documentation on `order_with_respect_to` for more information.

ordering See the django documentation on `ordering` for more information.

permissions See the django documentation on `permissions` for more information.

proxy If set to `True` then this model will be treated a proxy model.

Also see the django documentation about `proxy`.

index_together See the django documentation on `index_together` for more information.

unique_together See the django documentation on `unique_together` for more information.

verbose_name A human-readable name of the models name, singular. If this is not set in `Meta`, django will try to guess a human readable name by using the `object_name` and inserting appropriate spaces for the CamelCased model name and then making everything lowercase.

See the django documentation on `verbose_name` for more information.

verbose_name_plural A human-readable name of the models name, plural. If this is not set in `Meta`, it will default to `verbose_name` + `"s"`.

See the django documentation on `verbose_name_plural` for more information.

Attributes

abstract_managers To handle various inheritance situations, we need to track where managers came from (concrete or abstract base classes).

auto_created TODO ...

auto_field TODO ...

concrete_managers TODO ...

concrete_model TODO ...

has_auto_field TODO ...

local_fields TODO ...

local_many_to_many TODO ...

model This is the actual `django.db.models.Model` that the `_meta` attribute is attached to.

model_name TODO ...

object_name It is the actual name of the model class.

parents TODO ...

pk TODO ...

proxy_for_model For any class that is a proxy (including automatically created classes for deferred object loading), `proxy_for_model` tells us which class this model is proxying. Note that `proxy_for_model` can create a chain of proxy models. For non-proxy models, the variable is always `None`.

related_fkey_lookups List of all lookups defined in ForeignKey 'limit_choices_to' options from *other* models. Needed for some admin checks. Internal use only.

swappable TODO ...

virtual_fields TODO ...

Methods

module_name(self) TODO ...

add_field(self, field) TODO ...

add_virtual_field(self, field) TODO ...

setup_pk(self, field) TODO ...

pk_index(self) TODO ...

setup_proxy(self, target) TODO ...

verbose_name_raw(self) TODO ...

fields(self) TODO ...

concrete_fields(self) TODO ...

local_concrete_fields(self) TODO ...

get_fields_with_model(self) TODO ...

get_concrete_fields_with_model(self) TODO ...

get_m2m_with_model(self) TODO ...

get_field(self, name, many_to_many=True) TODO ...

get_all_field_names(self) TODO ...

init_name_map(self) TODO ...

get_add_permission(self) TODO ...

get_change_permission(self) TODO ...

get_delete_permission(self) TODO ...

get_all_related_objects(self, local_only=False, include_hidden=False TODO ...

get_all_related_objects_with_model(self, local_only=False TODO ...

get_all_related_many_to_many_objects(self, local_only=False) TODO ...

get_all_related_m2m_objects_with_model(self) TODO ...

get_base_chain(self, model) TODO ...

get_parent_list(self) TODO ...

get_ancestor_link(self, ancestor) TODO ...

CHAPTER 3

Indices and tables

- genindex
- search

Index

A

Actions, 18
  Writing List Actions, 19

C

Contributing, 7
  Getting your Pull Requests Accepted, 9
  Issues, 8
  Pull Requests, 8
  Pulling Upstream Changes, 9
  Pulling with Rebase, 9
  Setup, 7
  Topic Branches, 8

D

Design, 11
  Backend Goals, 12
  Constraints, 11
  REST API Goals, 12
  UI Goals, 12

G

Getting your Pull Request Accepting, 9
  Don't mix code changes with whitespace cleanup, 10
  Don't reduce test coverage, 9
  Keep your pull requests limited to single issues, 10
  Run the tests, 9

I

installation, 5
internationalization, 14
IsStaffPermission (class in djadmin2.permissions), 22
IsSuperuserPermission (class in djadmin2.permissions), 22

M

ModelAddPermission (class in djadmin2.permissions), 22
ModelChangePermission (class in djadmin2.permissions), 22
ModelDeletePermission (class in djadmin2.permissions), 22
ModelViewPermission (class in djadmin2.permissions), 22

P

Permissions, 22
  Built-In Permission Classes, 22
  Checking for Permissions on Other Models, 24
  Custom Permission Classes, 22
  Dynamically Check for a Specific Permission Name, 25
  Object-Level Permissions, 23
  Permissions in Templates, 23