flask-peewee Documentation

Release 0.6.7

charles leifer

Sep 21, 2018

Contents

Warning: I'm sorry to announce that flask-peewee will now be in maintenance-only mode. This decision is motivated by a number of factors:

- Flask-Admin provides a superior admin interface and has support for peewee models.
- Flask-Security and Flask-Login both provide authentication functionality, and work well with Peewee.
- Most importantly, though, I do not find myself wanting to work on flask-peewee.

I plan on rewriting the `Database` and `REST API` portions of flask-peewee and repackaging them as a new library, but flask-peewee as it stands currently will be in maintenance-only mode.

Welcome to the flask-peewee documentation!

provides a layer of integration between the flask web framework and the peewee orm.

Contents:

CHAPTER 1

Installing

flask-peewee can be installed very easily using pip.

```
pip install flask-peewee
```

If you do not have the dependencies installed already, pip will install them for you, but for reference they are:

- flask
- peewee
- wtforms
- wtf-peewee
- python 2.5 or greater

1.1 Using git

If you want to run the very latest, feel free to pull down the repo from github and install by hand.

```
git clone https://github.com/coleifer/flask-peewee.git
cd flask-peewee
python setup.py install
```

You can run the tests using the test-runner:

```
python setup.py test
```

CHAPTER 2

Getting Started

The goal of this document is to help get you up and running quickly. So without further ado, let's get started.

Note: Hopefully you have some familiarity with the flask framework and the peewee orm, but if not those links should help you get started.

Note: For a complete example project, check the example app that ships with flask-peewee.

2.1 Creating a flask app

First, be sure you have *installed flask-peewee and its dependencies*. You can verify by running the test suite: `python setup.py test`.

After ensuring things are installed, open a new file called "app.py" and enter the following code:

```
from flask import Flask

app = Flask(__name__)
app.config.from_object(__name__)

if __name__ == '__main__':
    app.run()
```

This isn't very exciting, but we can check out our project by running the app:

```
$ python app.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

Navigating to the url listed will show a simple 404 page, because we haven't configured any templates or views yet.

2.2 Creating a simple model

Let's add a simple model. Before we can do that, though, it is necessary to initialize the peewee database wrapper and configure the database:

```python
from flask import Flask

# flask-peewee bindings
from flask_peewee.db import Database

# configure our database
DATABASE = {
    'name': 'example.db',
    'engine': 'peewee.SqliteDatabase',
}
DEBUG = True
SECRET_KEY = 'ssshhhh'

app = Flask(__name__)
app.config.from_object(__name__)

# instantiate the db wrapper
db = Database(app)

if __name__ == '__main__':
    app.run()
```

What this does is provides us with request handlers which connect to the database on each request and close it when the request is finished. It also provides a base model class which is configured to work with the database specified in the configuration.

Now we can create a model:

```python
import datetime
from peewee import *


class Note(db.Model):
    message = TextField()
    created = DateTimeField(default=datetime.datetime.now)
```

Note: The model we created, `Note`, subclasses `db.Model`, which in turn is a subclass of `peewee.Model` that is pre-configured to talk to our database.

2.3 Setting up a simple base template

We'll need a simple template to serve as the base template for our app, so create a folder named `templates`. In the `templates` folder create a file `base.html` and add the following:

```html
<!doctype html>
<html>
<title>Test site</title>
<body>
```

(continues on next page)

(continued from previous page)

```
  <h2>{% block content_title %}{% endblock %}</h2>
  {% block content %}{% endblock %}
</body>
</html>
```

2.4 Adding users to the site

Before we can edit these Note models in the admin, we'll need to have some way of authenticating users on the site. This is where *Auth* comes in. *Auth* provides a User model and views for logging in and logging out, among other things, and is required by the *Admin*.

```
from flask_peewee.auth import Auth

# create an Auth object for use with our flask app and database wrapper
auth = Auth(app, db)
```

Let's also modify the code that runs our app to ensure our tables get created if need be:

```
if __name__ == '__main__':
    auth.User.create_table(fail_silently=True)
    Note.create_table(fail_silently=True)

    app.run()
```

After cleaning up the imports and declarations, we have something like the following:

```
import datetime
from flask import Flask
from flask_peewee.auth import Auth
from flask_peewee.db import Database
from peewee import *

# configure our database
DATABASE = {
    'name': 'example.db',
    'engine': 'peewee.SqliteDatabase',
}
DEBUG = True
SECRET_KEY = 'ssshhhh'

app = Flask(__name__)
app.config.from_object(__name__)

# instantiate the db wrapper
db = Database(app)


class Note(db.Model):
    message = TextField()
    created = DateTimeField(default=datetime.datetime.now)


# create an Auth object for use with our flask app and database wrapper
auth = Auth(app, db)
```

(continues on next page)

(continued from previous page)

```python
if __name__ == '__main__':
    auth.User.create_table(fail_silently=True)
    Note.create_table(fail_silently=True)

    app.run()
```

2.5 Managing content using the admin area

Now we're ready to add the admin. Place the following lines of code after the initialization of the `Auth` class:

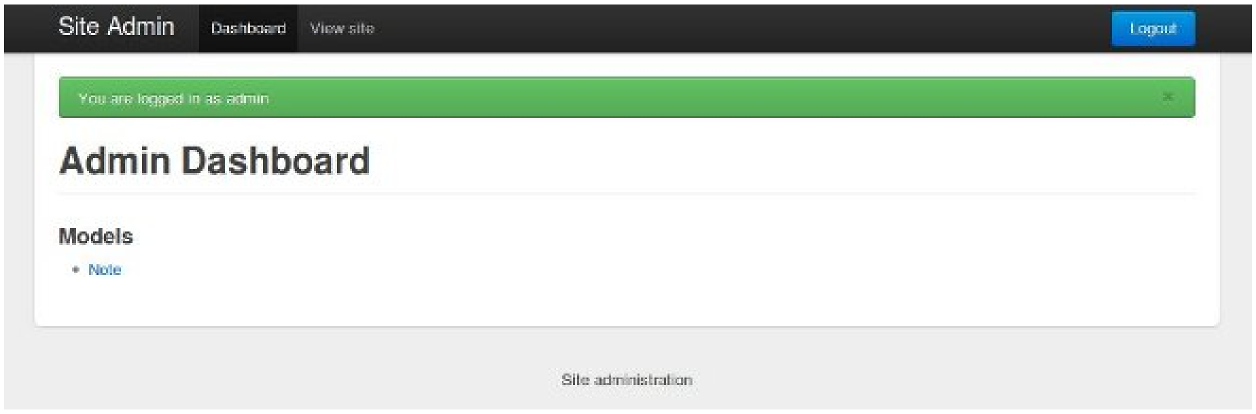
```python
from flask_peewee.admin import Admin

admin = Admin(app, auth)
admin.register(Note)

admin.setup()
```

We now have a functioning admin site! Of course, we'll need a user log in with, so open up an interactive python shell in the directory alongside the app and run the following:

```python
from app import auth
auth.User.create_table(fail_silently=True)  # make sure table created.
admin = auth.User(username='admin', email='', admin=True, active=True)
admin.set_password('admin')
admin.save()
```

It should now be possible to:

1. navigate to http://127.0.0.1:5000/admin/
2. enter in the username and password ("admin", "admin")
3. be redirected to the admin dashboard



You are logged in as admin

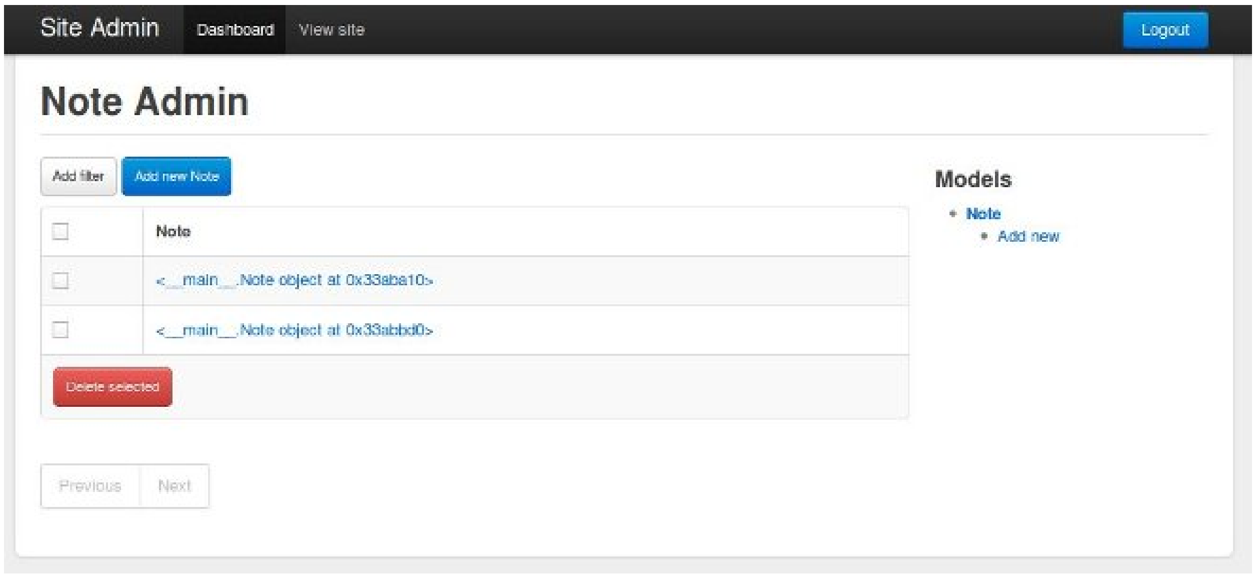
Admin Dashboard

Models

- Note

Site administration

The dashboard is pretty empty right now. Go ahead and add a few notes (http://127.0.0.1:5000/admin/note/). If you navigate now to the note modeladmin you will see something like this:



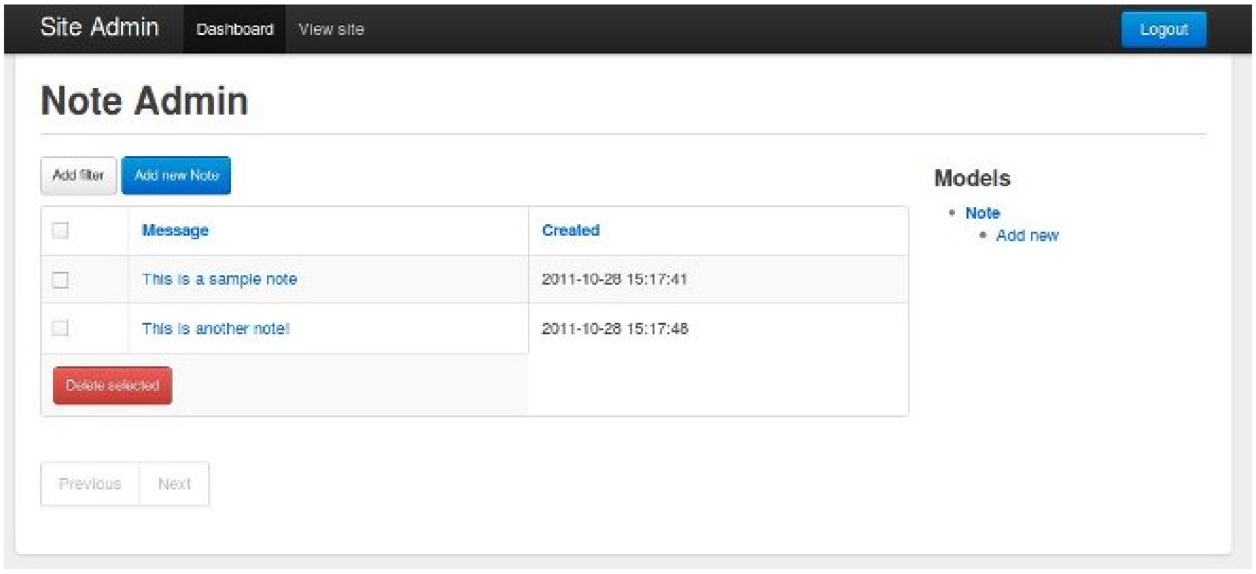
Note Admin

	Note
	<__main__.Note object at 0x33aba10>
	<__main__.Note object at 0x33abbd0>

This is pretty lousy so let's clean it up to display the message and when it was published. We can do that by customizing the columns displayed. Edit the app with the following changes:

```python
from flask_peewee.admin import Admin, ModelAdmin

class NoteAdmin(ModelAdmin):
    columns = ('message', 'created',)

admin = Admin(app, auth)

admin.register(Note, NoteAdmin)

admin.setup()
```

Now our modeladmin should look more like this:



Note Admin

	Message	Created
	This is a sample note	2011-10-28 15:17:41
	This is another note!	2011-10-28 15:17:48

Let's go ahead and add the `auth.User` model to the admin as well:

```
admin.register(Note, NoteAdmin)
auth.register_admin(admin)

admin.setup()
```

2.6 Exposing content using a REST API

Adding a REST API is very similar to how we added the *Admin* interface. We will create a *RestAPI* object, and then register our project's models with it. If we want to customize things, we can subclass *RestResource*.

The first step, then, is to create the *RestAPI* object:

```
from flask_peewee.rest import RestAPI

# create a RestAPI container
api = RestAPI(app)

api.setup()
```

This doesn't do anything yet, we need to register models with it first. Let's register the `Note` model from earlier:

```
# create a RestAPI container
api = RestAPI(app)

# register the Note model
api.register(Note)

api.setup()
```

Assuming your project is still running, try executing the following command (or just browse to the url listed):

```
$ curl http://127.0.0.1:5000/api/note/
```

You should see something like the following:

```
{
  "meta": {
    "model": "note",
    "next": "",
    "page": 1,
    "previous": ""
  },
  "objects": [
    {
      "message": "blah blah blah this is a note",
      "id": 1,
      "created": "2011-09-23 09:07:39"
    },
    {
      "message": "this is another note!",
      "id": 2,
      "created": "2011-09-23 09:07:54"
    }
  ]
}
```

Suppose we want it to also be possible for registered users to be able to POST messages using the API. If you try and make a POST right now, you will get a `401` response:

```
$ curl -i -d '' http://127.0.0.1:5000/api/note/

HTTP/1.0 401 UNAUTHORIZED
WWW-Authenticate: Basic realm="Login Required"
Content-Type: text/html; charset=utf-8
Content-Length: 21
Server: Werkzeug/0.8-dev Python/2.6.6
Date: Fri, 23 Sep 2011 14:45:38 GMT

Authentication failed
```

This is because we have not configured any *`Authentication`* method for our *`RestAPI`*.

Note: The default authentication mechanism for the API only accepts GET requests. In order to handle POST/PUT/DELETE you will need to use a subclass of the *`Authentication`* class.

In order to allow users of the site to post notes, we will use the *`UserAuthentication`* subclass, which requires that API requests be made with HTTP Basic auth and that the auth credentials match those of one of the `auth.User` models.

```
from flask_peewee.rest import RestAPI, UserAuthentication

# instantiate the user auth
user_auth = UserAuthentication(auth)

# create a RestAPI container
api = RestAPI(app, default_auth=user_auth)
```

Now we can post new notes using a command-line tool like curl:

```
$ curl -u admin:admin -d data='{"message": "hello api"}' http://127.0.0.1:5000/api/
↪note/

{
  "message": "hello api",
  "id": 3,
  "created": "2011-09-23 13:14:56"
}
```

You can see that it returns a serialized copy of the new `Note` object.

Note: This is just a small example of what you can do with the Rest API – refer to the *Rest API docs* for more detailed information, including

- limiting access on a per-model basis
- customizing which fields are returned by the API
- filtering and querying using GET parameters

CHAPTER 3

Database Wrapper

The Peewee database wrapper provides a thin layer of integration between flask apps and the peewee orm.

The database wrapper is important because it ensures that a database connection is created for every incoming request, and closed upon request completion. It also provides a subclass of `Model` which works with the database specified in your app's configuration.

Most features of `flask-peewee` require a database wrapper, so you very likely always create one.

The database wrapper reads its configuration from the Flask application. The configuration requires only two arguments, but any additional arguments will be passed to the database driver when connecting:

name The name of the database to connect to (or filename if using sqlite3)

engine The database driver to use, must be a subclass of `peewee.Database`.

```
from flask import Flask
from peewee import *

from flask_peewee.db import Database

DATABASE = {
    'name': 'example.db',
    'engine': 'peewee.SqliteDatabase',
}

app = Flask(__name__)
app.config.from_object(__name__) # load database configuration from this module

# instantiate the db wrapper
db = Database(app)

# start creating models
class Blog(db.Model):
    name = CharField()
    # .. etc
```

3.1 Other examples

To connect to MySQL using authentication:

```
DATABASE = {
    'name': 'my_database',
    'engine': 'peewee.MySQLDatabase',
    'user': 'db_user',
    'passwd': 'secret password',
}
```

If using a multi-threaded WSGI server:

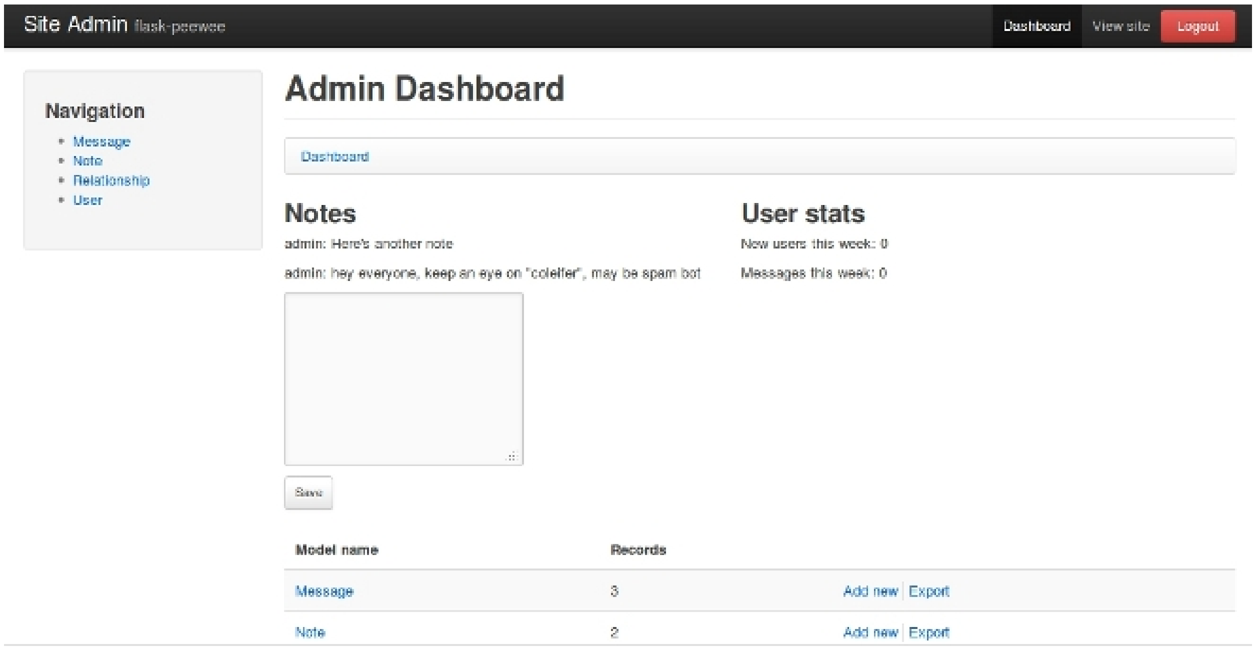
```
DATABASE = {
    'name': 'foo.db',
    'engine': 'peewee.SqliteDatabase',
    'threadlocals': True,
}
```

CHAPTER 4

Admin Interface

Many web applications ship with an “admin area”, where priveleged users can view and modify content. By introspecting your application’s models, flask-peewee can provide you with straightforward, easily-extensible forms for managing your application content.

Here’s a screen-shot of the admin dashboard:



Notes

admin: Here's another note

admin: hey everyone, keep an eye on "coleifer", may be spam bot

User stats

New users this week: 0

Messages this week: 0

Model name	Records	
Message	3	Add new | Export
Note	2	Add new | Export

4.1 Getting started

To get started with the admin, there are just a couple steps:

1. Instantiate an `Auth` backend for your project – this component is responsible for providing the security for the admin area

    ```
    from flask import Flask

    from flask_peewee.auth import Auth
    from flask_peewee.db import Database

    app = Flask(__name__)
    db = Database(app)

    # needed for authentication
    auth = Auth(app, db)
    ```

2. Instantiate an `Admin` object

    ```
    # continued from above...
    from flask_peewee.admin import Admin

    admin = Admin(app, auth)
    ```

3. Register any `ModelAdmin` or `AdminPanel` objects you would like to expose via the admin

    ```
    # continuing... assuming "Blog" and "Entry" models
    admin.register(Blog) # register "Blog" with vanilla ModelAdmin
    admin.register(Entry, EntryAdmin) # register "Entry" with a custom␣
    ↪ModelAdmin subclass

    # assume we have an "AdminPanel" called "NotePanel"
    admin.register_panel('Notes', NotePanel)
    ```

4. Call `Admin.setup()`, which registers the admin blueprint and configures the urls

    ```
    # after all models and panels are registered, configure the urls
    admin.setup()
    ```

Note: For a complete example, check the example which ships with the project.

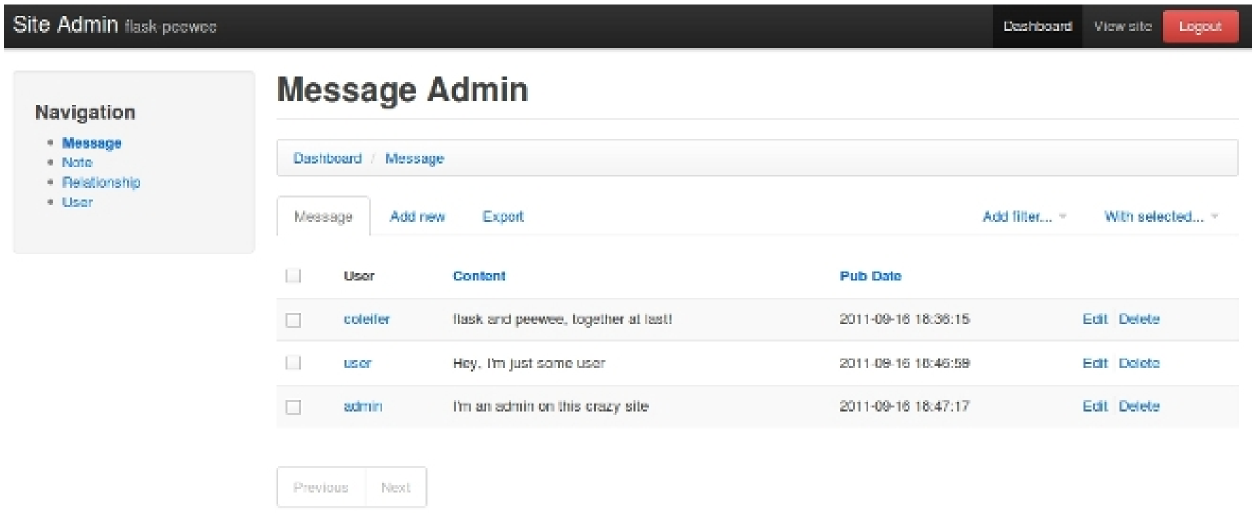
4.2 Customizing how models are displayed

We'll use the "Message" model taken from the example app, which looks like this:

```
class Message(db.Model):
    user = ForeignKeyField(User)
    content = TextField()
    pub_date = DateTimeField(default=datetime.datetime.now)

    def __unicode__(self):
        return '%s: %s' % (self.user, self.content)
```

If we were to simply register this model with the admin, it would look something like this:

```
admin = Admin(app, auth)
admin.register(Message)

admin.setup()
```



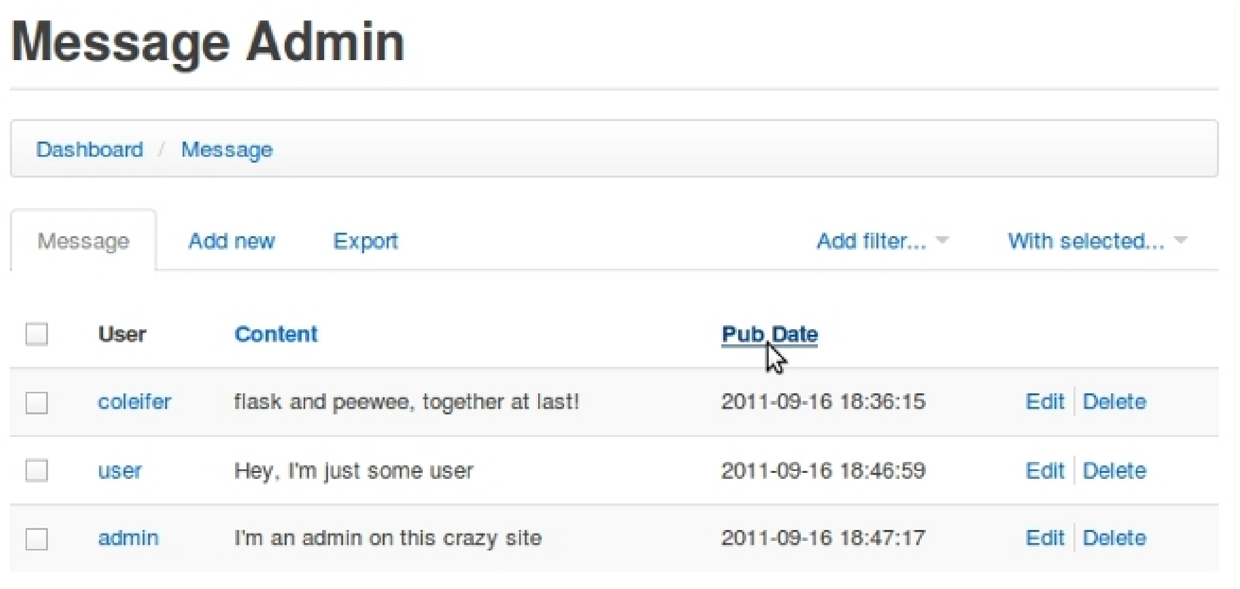
Message Admin

	User	Content	Pub Date	
☐	coleifer	flask and peewee, together at last!	2011-09-16 18:36:15	Edit Delete
☐	user	Hey, I'm just some user	2011-09-16 18:46:59	Edit Delete
☐	admin	I'm an admin on this crazy site	2011-09-16 18:47:17	Edit Delete

A quick way to improve the appearance of this view is to specify which columns to display. To start customizing how the `Message` model is displayed in the admin, we'll subclass *`ModelAdmin`*.

```python
from flask_peewee.admin import ModelAdmin

class MessageAdmin(ModelAdmin):
    columns = ('user', 'content', 'pub_date',)

admin.register(Message, MessageAdmin)

admin.setup()
```

Now the admin shows all the columns and they can be clicked to sort the data:



Message Admin

	User	Content	Pub Date	
☐	coleifer	flask and peewee, together at last!	2011-09-16 18:36:15	Edit Delete
☐	user	Hey, I'm just some user	2011-09-16 18:46:59	Edit Delete
☐	admin	I'm an admin on this crazy site	2011-09-16 18:47:17	Edit Delete

Suppose privacy is a big concern, and under no circumstances should a user be able to see another user's messages – even in the admin. This can be done by overriding the `get_query()` method:

```
def get_query(self):
    return self.model.select().where(self.model.user == g.user)
```

Now a user will only be able to see and edit their own messages.

4.2.1 Overriding Admin Templates

Use the `ModelAdmin.get_template_overrides()` method to override templates for an individual `Model`:

```
class MessageAdmin(ModelAdmin):
    columns = ('user', 'content', 'pub_date',)

    def get_template_overrides(self):
        # override the edit template with a custom one
        return {'edit': 'messages/admin/edit.html'}

admin.register(Message, MessageAdmin)
```

This instructs the admin to use a custom template for the edit page in the Message admin. That template is stored in the application's templates. It might look something like this:

```
{% extends "admin/models/edit.html" %} {# override the default edit template #}

{# override any blocks here #}
```

There are five templates that can be overridden:

- index
- add
- edit
- delete
- export

4.3 Nicer display for Foreign Key fields

If you have a model that foreign keys to another, by default the related model instances are displayed in a <select> input.

This can be problematic if you have a large list of models to search (causes slow load time, hurts the database). To mitigate this pain, foreign key lookups can be done using a paginated widget that supports type-ahead searching.

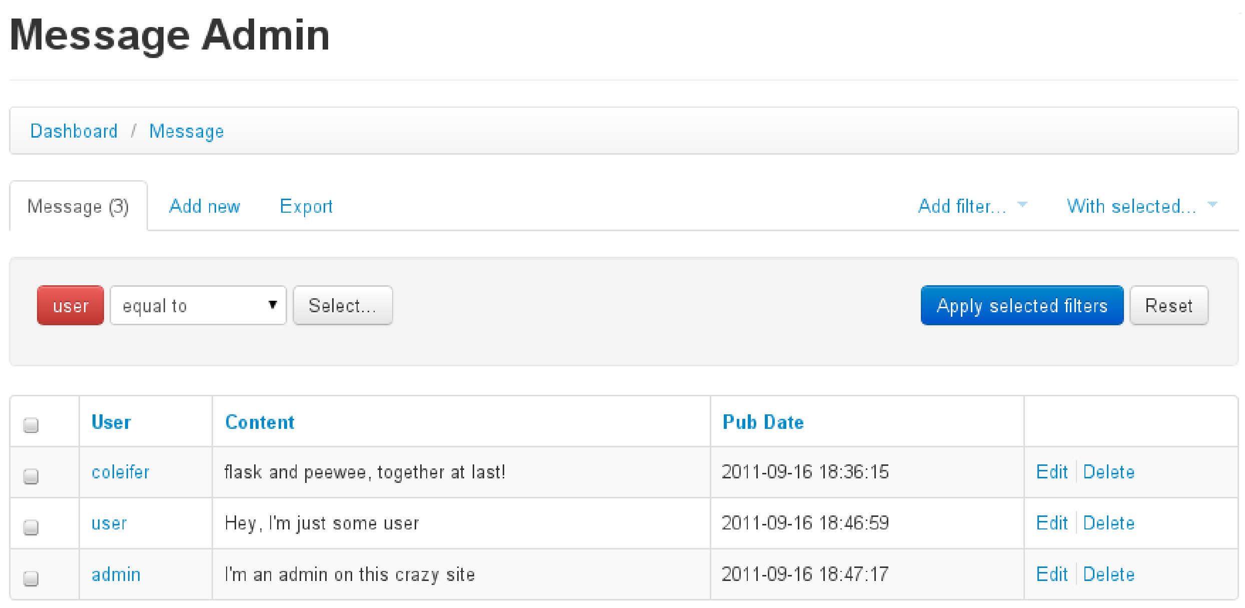
Setting this up is very easy:

```
class MessageAdmin(ModelAdmin):
    columns = ('user', 'content', 'pub_date',)
    foreign_key_lookups = {'user': 'username'}
```

When flask-peewee sees the `foreign_key_lookups` it will use the special modal window to select instances. This applies to both filters and model forms:

4.3.1 Filters

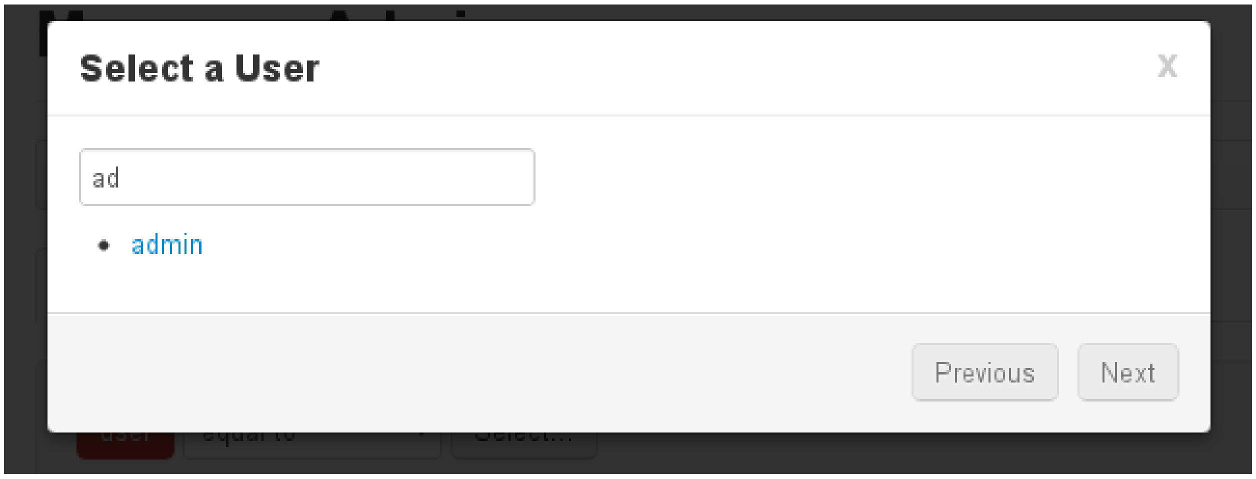
1. Select a user by clicking the "Select. . . " button


Message Admin

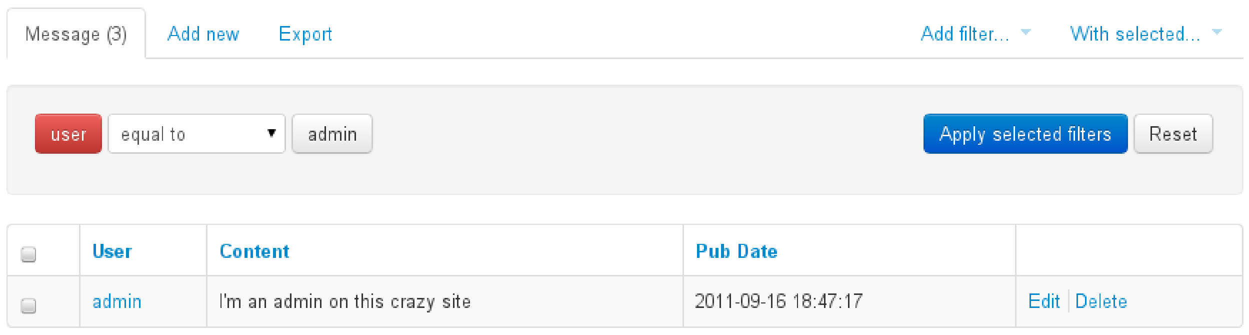
Dashboard / Message

	User	Content	Pub Date	
	coleifer	flask and peewee, together at last!	2011-09-16 18:36:15	Edit | Delete
	user	Hey, I'm just some user	2011-09-16 18:46:59	Edit | Delete
	admin	I'm an admin on this crazy site	2011-09-16 18:47:17	Edit | Delete

2. A modal window with a paginated list and typeahead search appers:


Select a User

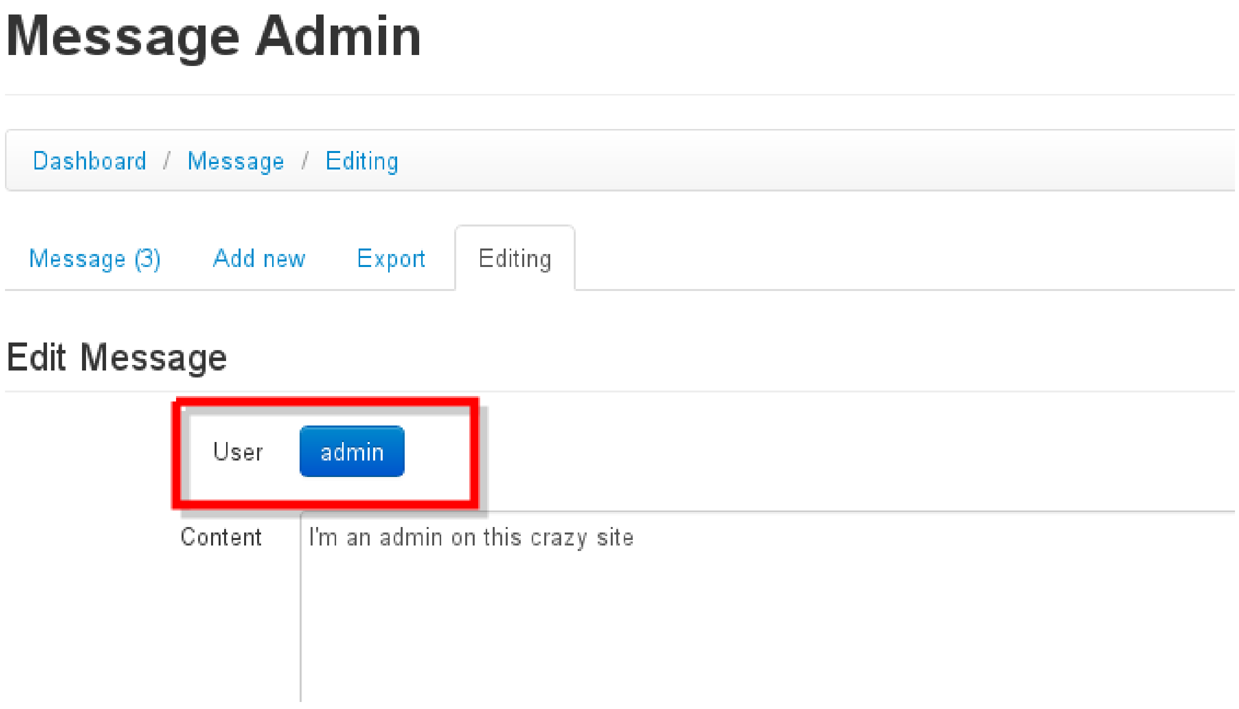
3. The button now indicates the selected user, clicking again will reload the dialog:



	User	Content	Pub Date	
	admin	I'm an admin on this crazy site	2011-09-16 18:47:17	Edit | Delete

4.3.2 Admin ModelForms

The interface is the same as with the filters, except the foreign key field is replaced by a simple button:



I'm an admin on this crazy site

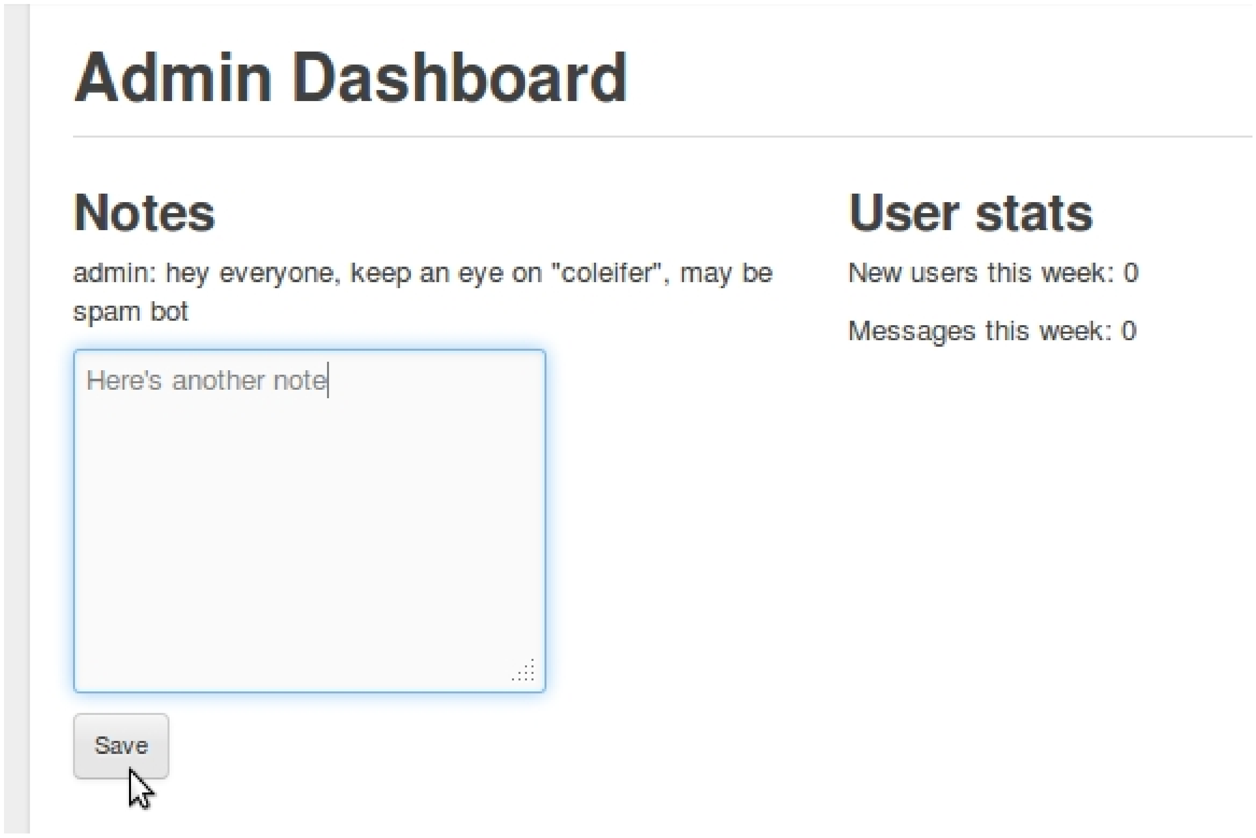
4.4 Creating admin panels

AdminPanel classes provide a way of extending the admin dashboard with arbitrary functionality. These are displayed as "panels" on the admin dashboard with a customizable template. They may additionally, however, define any views and urls. These views will automatically be protected by the same authentication used throughout the admin area.

Some example use-cases for AdminPanels might be:

- Display some at-a-glance functionality in the dashboard, like stats on new user signups.
- Provide a set of views that should only be visible to site administrators, for example a mailing-list app.
- Control global site settings, turn on and off features, etc.

Referring to the example app, we'll look at a simple panel that allows administrators to leave "notes" in the admin area:



Admin Dashboard

Notes

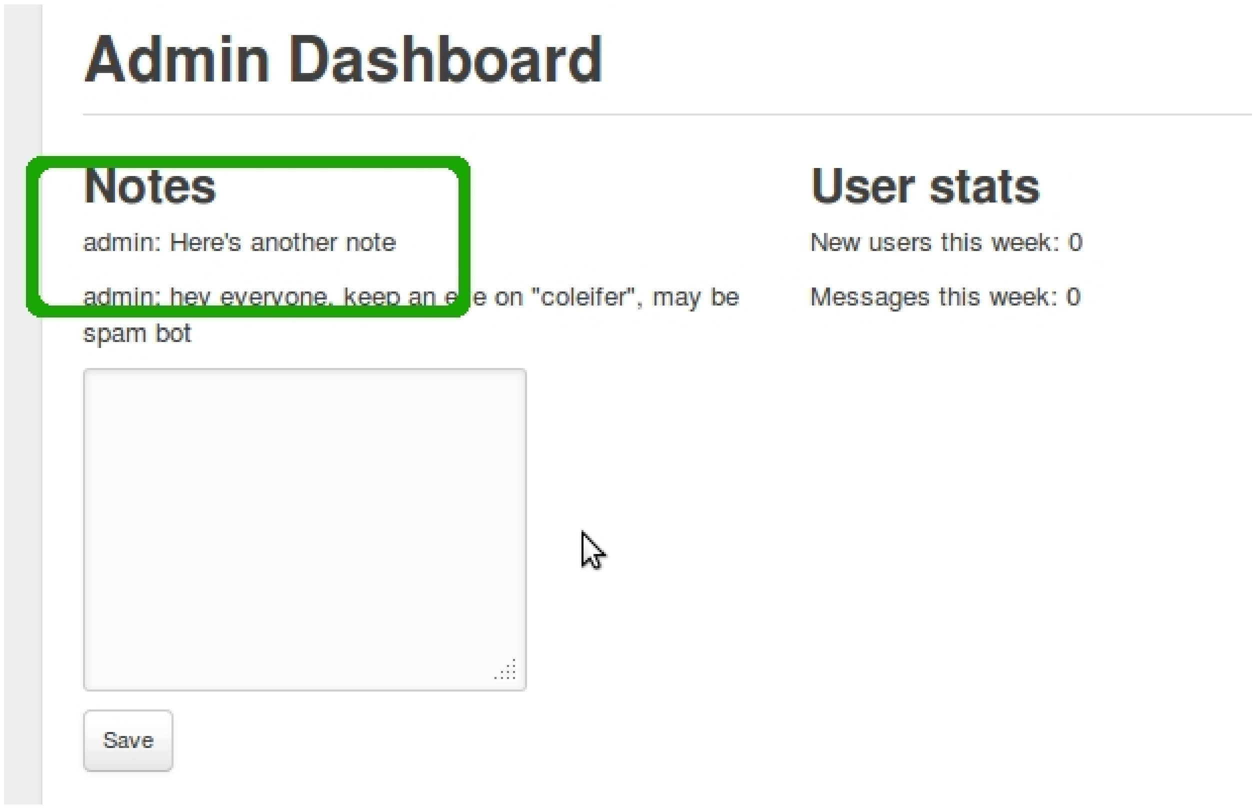
admin: hey everyone, keep an eye on "coleifer", may be spam bot

Here's another note

User stats

New users this week: 0

Messages this week: 0



Admin Dashboard

Notes

admin: Here's another note

admin: hey everyone, keep an eye on "coleifer", may be spam bot

User stats

New users this week: 0

Messages this week: 0

Here's what the panel class looks like:

```
class NotePanel(AdminPanel):
    template_name = 'admin/notes.html'

    def get_urls(self):
        return (
            ('/create/', self.create),
        )

    def create(self):
        if request.method == 'POST':
            if request.form.get('message'):
                Note.create(
                    user=auth.get_logged_in_user(),
                    message=request.form['message'],
                )
        next = request.form.get('next') or self.dashboard_url()
        return redirect(next)

    def get_context(self):
        return {
            'note_list': Note.select().order_by(Note.created_date.desc()).limit(3)
        }
```

When the admin dashboard is rendered (`/admin/`), all panels are rendered using the templates the specify. The template is rendered with the context provided by the panel's `get_context` method.

And the template:

```
{% extends "admin/panels/default.html" %}

{% block panel_content %}
  {% for note in note_list %}
    <p>{{ note.user.username }}: {{ note.message }}</p>
  {% endfor %}
  <form method="post" action="{{ url_for(panel.get_url_name('create')) }}">
    <input type="hidden" value="{{ request.url }}" />
    <p><textarea name="message"></textarea></p>
    <p><button type="submit" class="btn small">Save</button></p>
  </form>
{% endblock %}
```

A panel can provide as many urls and views as you like. These views will all be protected by the same authentication as other parts of the admin area.

4.5 Handling File Uploads

Flask and wtforms both provide support for handling file uploads (on the server and generating form fields). Peewee, however, does not have a "file field" – generally I store a path to a file on disk and thus use a `CharField` for the storage.

Here's a very simple example of a "photo" model and a `ModelAdmin` that enables file uploads.

```
# models.py
import datetime
import os
```

(continues on next page)

(continued from previous page)

```
from flask import Markup
from peewee import *
from werkzeug import secure_filename

from app import app, db


class Photo(db.Model):
    image = CharField()

    def __unicode__(self):
        return self.image

    def save_image(self, file_obj):
        self.image = secure_filename(file_obj.filename)
        full_path = os.path.join(app.config['MEDIA_ROOT'], self.image)
        file_obj.save(full_path)
        self.save()

    def url(self):
        return os.path.join(app.config['MEDIA_URL'], self.image)

    def thumb(self):
        return Markup('<img src="%s" style="height: 80px;" />' % self.url())
```

```
# admin.py
from flask import request
from flask_peewee.admin import Admin, ModelAdmin
from wtforms.fields import FileField, HiddenField
from wtforms.form import Form

from app import app, db
from auth import auth
from models import Photo


admin = Admin(app, auth)


class PhotoAdmin(ModelAdmin):
    columns = ['image', 'thumb']

    def get_form(self, adding=False):
        class PhotoForm(Form):
            image = HiddenField()
            image_file = FileField(u'Image file')

        return PhotoForm

    def save_model(self, instance, form, adding=False):
        instance = super(PhotoAdmin, self).save_model(instance, form, adding)
        if 'image_file' in request.files:
            file = request.files['image_file']
            instance.save_image(file)
        return instance
```

(continues on next page)

(continued from previous page)

```
admin.register(Photo, PhotoAdmin)
```

CHAPTER 5

Authentication

The *Authentication* class provides a means of authenticating users of the site. It is designed to work out-of-the-box with a simple `User` model, but can be heavily customized.

The *Auth* system is comprised of a single class which is responsible for coordinating incoming requests to your project with known users. It provides the following:

- views for login and logout
- model to store user data (or you can provide your own)
- mechanism for identifying users across requests (uses session storage)

All of these pieces can be customized, but the default out-of-box implementation aims to provide a good starting place.

The auth system is also designed to work closely with the *Admin Interface*.

5.1 Getting started

In order to provide a method for users to authenticate with your site, instantiate an *Auth* backend for your project:

```
from flask import Flask

from flask_peewee.auth import Auth
from flask_peewee.db import Database

app = Flask(__name__)
db = Database(app)

# needed for authentication
auth = Auth(app, db)
```

Note: `user` is reserverd keyword in Postgres. Pass db_table to Auth to override db table.

5.2 Marking areas of the site as login required

If you want to mark specific areas of your site as requiring auth, you can decorate views using the *Auth.login_required()* decorator:

```
@app.route('/private/')
@auth.login_required
def private_timeline():
    user = auth.get_logged_in_user()

    # ... display the private timeline for the logged-in user
```

If the request comes from someone who has not logged-in with the site, they are redirected to the `Auth.login()` view, which allows the user to authenticate. After successfully logging-in, they will be redirected to the page they requested initially.

5.3 Retrieving the current user

Whenever in a request context, the currently logged-in user is available by calling *Auth.get_logged_in_user()*, which will return `None` if the requesting user is not logged in.

The auth system also registers a pre-request hook that stores the currently logged-in user in the special flask variable `g`.

5.4 Accessing the user in the templates

The auth system registers a template context processor which makes the logged-in user available in any template:

```
{% if user %}
  <p>Hello {{ user.username }}</p>
{% else %}
  <p>Please <a href="{{ url_for('auth.login') }}?next={{ request.path }}">log in</a></
↪p>
{% endif %}
```

5.5 Using a custom "User" model

It is easy to use your own model for the `User`, though depending on the amount of changes it may be necessary to override methods in both the *Auth* and *Admin* classes.

Unless you want to override the default behavior of the *Auth* class' mechanism for actually authenticating users (which you may want to do if relying on a 3rd-party for auth) – you will want to be sure your `User` model implements two methods:

- `set_password(password)` – takes a raw password and stores an encrypted version on model
- `check_password(password)` – returns whether or not the supplied password matches the one stored on the model instance

Note: The *BaseUser* mixin provides default implementations of these two methods.

Here’s a simple example of extending the auth system to use a custom user model:

```python
from flask_peewee.auth import BaseUser # <-- implements set_password and check_
↪password

app = Flask(__name__)
db = Database(app)

# create our custom user model. note that we're mixing in BaseUser in order to
# use the default auth methods it implements, "set_password" and "check_password"
class User(db.Model, BaseUser):
    username = CharField()
    password = CharField()
    email = CharField()

    # ... our custom fields ...
    is_superuser = BooleanField()


# create a modeladmin for it
class UserAdmin(ModelAdmin):
    columns = ('username', 'email', 'is_superuser',)

    # Make sure the user's password is hashed, after it's been changed in
    # the admin interface. If we don't do this, the password will be saved
    # in clear text inside the database and login will be impossible.
    def save_model(self, instance, form, adding=False):
        orig_password = instance.password

        user = super(UserAdmin, self).save_model(instance, form, adding)

        if orig_password != form.password.data:
            user.set_password(form.password.data)
            user.save()

        return user


# subclass Auth so we can return our custom classes
class CustomAuth(Auth):
    def get_user_model(self):
        return User

    def get_model_admin(self):
        return UserAdmin

# instantiate the auth
auth = CustomAuth(app, db)
```

Here’s how you might integrate the custom auth with the admin area of your site:

```python
# subclass Admin to check for whether the user is a superuser
class CustomAdmin(Admin):
    def check_user_permission(self, user):
        return user.is_superuser

# instantiate the admin
admin = CustomAdmin(app, auth)
```

(continues on next page)

(continued from previous page)

```

admin.register(User, UserAdmin)
admin.setup()
```

CHAPTER 6

REST Api

flask-peewee comes with some tools for exposing your project's models via a RESTful API. There are several components to the `rest` module, but the basic setup is to create an instance of `RestAPI` and then register your project's models with subclasses of `RestResource`.

Each `RestResource` you expose via the API will support, by default, the following:

- */api/<model name>/* – GET and POST requests
- */api/<model name>/<primary key>/* – GET, PUT and DELETE requests

Also, you can filter results by columns on the model using django-style syntax, for example:

- */api/blog/?name=Some%20Blog*
- */api/blog/?author__username=some_blogger*

6.1 Getting started with the API

In this documentation we'll start with a very simple API and build it out. The complete version of this API is included in the example-app, so feel free to refer there.

The project will be a simple 'twitter-like' app where users can post short messages and "follow" other users.

Note: If you're using apache with mod_wsgi and would like to use any of the auth backends that use basic auth, you will need to add the following directive: `WSGIPassAuthorization On`

6.1.1 Project models

There are three main models - `User`, `Relationship` and `Message` - which we will expose via the API. Here is a truncated version of what they look like:

```
from flask_peewee.auth import BaseUser

class User(db.Model, BaseUser):
    username = CharField()
    password = CharField()
    email = CharField()
    join_date = DateTimeField(default=datetime.datetime.now)
    active = BooleanField(default=True)
    admin = BooleanField(default=False)

class Relationship(db.Model):
    from_user = ForeignKeyField(User, related_name='relationships')
    to_user = ForeignKeyField(User, related_name='related_to')

class Message(db.Model):
    user = ForeignKeyField(User)
    content = TextField()
    pub_date = DateTimeField(default=datetime.datetime.now)
```

6.2 Creating a RestAPI

The `RestAPI` acts as a container for the various `RestResource` objects we will expose. By default it binds all resources to `/api/<model-name>/`.

Here we'll create a simple api and register our models:

```
from flask_peewee.rest import RestAPI

from app import app # our project's Flask app

# instantiate our api wrapper
api = RestAPI(app)

# register our models so they are exposed via /api/<model>/
api.register(User)
api.register(Relationship)
api.register(Message)

# configure the urls
api.setup()
```

Now if we hit our project at `/api/message/` we should get something like the following:

```
{
  "meta": {
    "model": "message",
    "next": "",
    "page": 1,
    "previous": ""
  },
  "objects": [
    {
      "content": "flask and peewee, together at last!",
      "pub_date": "2011-09-16 18:36:15",
      "user_id": 1,
```

(continues on next page)

(continued from previous page)

```
      "id": 1
    },
    {
      "content": "Hey, I'm just some user",
      "pub_date": "2011-09-16 18:46:59",
      "user_id": 2,
      "id": 2
    }
  ]
}
```

Say we're interested in the first message, we can hit `/api/message/1/` to view just the details on that object:

```
{
  content: "flask and peewee, together at last!"
  pub_date: "2011-09-16 18:36:15"
  user_id: 1
  id: 1
}
```

6.3 Customizing what is returned

If you access the `User` API endpoint, we quickly notice a problem:

```
$ curl http://127.0.0.1:5000/api/user/

{
  "meta": {
    "model": "user",
    "next": "",
    "page": 1,
    "previous": ""
  },
  "objects": [
    {
      "username": "admin",
      "admin": true,
      "email": "",
      "join_date": "2011-09-16 18:34:49",
      "active": true,
      "password": "d033e22ae348aeb5660fc2140aec35850c4da997",
      "id": 1
    },
    {
      "username": "coleifer",
      "admin": false,
      "email": "coleifer@gmail.com",
      "join_date": "2011-09-16 18:35:56",
      "active": true,
      "password": "a94a8fe5ccb19ba61c4c0873d391e987982fbbd3",
      "id": 2
    }
  ]
}
```

Passwords and email addresses are being exposed. In order to exclude these fields from serialization, subclass *RestResource*:

```
from flask_peewee.rest import RestAPI, RestResource

from app import app # our project's Flask app

# instantiate our api wrapper
api = RestAPI(app)

# create a special resource for users that excludes email and password
class UserResource(RestResource):
    exclude = ('password', 'email',)

# register our models so they are exposed via /api/<model>/
api.register(User, UserResource) # specify the UserResource
api.register(Relationship)
api.register(Message)
```

Now emails and passwords are no longer returned by the API.

6.4 Allowing users to post objects

What if we want to create new messages via the Api? Or modify/delete existing messages?

```
$ curl -i -d '' http://127.0.0.1:5000/api/message/

HTTP/1.0 401 UNAUTHORIZED
WWW-Authenticate: Basic realm="Login Required"
Content-Type: text/html; charset=utf-8
Content-Length: 21
Server: Werkzeug/0.8-dev Python/2.6.6
Date: Thu, 22 Sep 2011 16:14:21 GMT

Authentication failed
```

The authentication failed because the default authentication mechanism only allows read-only access.

In order to allow users to create messages via the API, we need to use a subclass of *Authentication* that allows `POST` requests. We also want to ensure that the requesting user is a member of the site.

For this we will use the *UserAuthentication* class as the default auth mechanism.

```
from auth import auth # import the Auth object used by our project

from flask_peewee.rest import RestAPI, RestResource, UserAuthentication

# create an instance of UserAuthentication
user_auth = UserAuthentication(auth)

# instantiate our api wrapper, specifying user_auth as the default
api = RestAPI(app, default_auth=user_auth)

# create a special resource for users that excludes email and password
class UserResource(RestResource):
    exclude = ('password', 'email',)
```

(continues on next page)

(continued from previous page)

```
# register our models so they are exposed via /api/<model>/
api.register(User, UserResource) # specify the UserResource
api.register(Relationship)
api.register(Message)

# configure the urls
api.setup()
```

Now we should be able to POST new messages.

```
import json
import httplib2

sock = httplib2.Http()
sock.add_credentials('admin', 'admin') # use basic auth

message = {'user_id': 1, 'content': 'hello api'}
msg_json = json.dumps(message)

headers, resp = sock.request('http://localhost:5000/api/message/', 'POST', body=msg_
↪json)

response = json.loads(resp)
```

The response object will look something like this:

```
{
  'content': 'hello api',
  'user_id': 1,
  'pub_date': '2011-09-22 11:25:02',
  'id': 3
}
```

There is a problem with this, however. Notice how the `user_id` was passed in with the POST data? This effectively will let a user post a message as another user. It also means a user can use PUT requests to modify another user's message:

```
# continued from above script
update = {'content': 'haxed you, bro'}
update_json = json.dumps(update)

headers, resp = sock.request('http://127.0.0.1:5000/api/message/2/', 'PUT',␣
↪body=update_json)

response = json.loads(resp)
```

The response will look like this:

```
{
  'content': 'haxed you, bro',
  'pub_date': '2011-09-16 18:36:15',
  'user_id': 2,
  'id': 2
}
```

This is a problem – we need a way of ensuring that users can only edit their own messages. Furthermore, when they create messages we need to make sure the message is assigned to them.

6.5 Restricting API access on a per-model basis

flask-peewee comes with a special subclass of *RestResource* that restricts POST/PUT/DELETE requests to prevent users from modifying another user's content.

```
from flask_peewee.rest import RestrictOwnerResource


class MessageResource(RestrictOwnerResource):
    owner_field = 'user'

api.register(Message, MessageResource)
```

Now, if we try and modify the message, we get a 403 Forbidden:

```
headers, resp = sock.request('http://127.0.0.1:5000/api/message/2/', 'PUT', 
↪body=update_json)
print headers['status']

# prints 403
```

It is fine to modify our own message, though (message with id=1):

```
headers, resp = sock.request('http://127.0.0.1:5000/api/message/1/', 'PUT', 
↪body=update_json)
print headers['status']

# prints 200
```

Under-the-hood, the implementation of the *RestrictOwnerResource* is pretty simple.

- PUT / DELETE – verify the authenticated user is the owner of the object
- POST – assign the authenticated user as the owner of the new object

6.6 Locking down a resource

Suppose we want to restrict normal users from modifying `User` resources. For this we can use a special subclass of *UserAuthentication* that restricts access to administrators:

```
from flask_peewee.rest import AdminAuthentication

# instantiate our user-based auth
user_auth = UserAuthentication(auth)

# instantiate admin-only auth
admin_auth = AdminAuthentication(auth)

# instantiate our api wrapper, specifying user_auth as the default
api = RestAPI(app, default_auth=user_auth)

# register the UserResource with admin auth
api.register(User, UserResource, auth=admin_auth)
```

6.7 Filtering records and querying

A REST Api is not very useful if it cannot be queried in a meaningful fashion. To this end, the flask-peewee *RestResource* objects support "django-style" filtering:

```
$ curl http://127.0.0.1:5000/api/message/?user=2
```

This call will return only messages by the `User` with id=2:

```
{
  "meta": {
    "model": "message",
    "next": "",
    "page": 1,
    "previous": ""
  },
  "objects": [
    {
      "content": "haxed you, bro",
      "pub_date": "2011-09-16 18:36:15",
      "user_id": 2,
      "id": 2
    }
  ]
}
```

Joins can be traversed using the django double-underscore notation:

```
$ curl http://127.0.0.1:5000/api/message/?user__username=admin
```

```
{
  "meta": {
    "model": "message",
    "next": "",
    "page": 1,
    "previous": ""
  },
  "objects": [
    {
      "content": "flask and peewee, together at last!",
      "pub_date": "2011-09-16 18:36:15",
      "user_id": 1,
      "id": 1
    },
    {
      "content": "hello api",
      "pub_date": "2011-09-22 11:25:02",
      "user_id": 1,
      "id": 3
    }
  ]
}
```

It is also supported to use different comparison operators with the same double-underscore notation:

```
$ curl http://127.0.0.1:5000/api/user/?user__lt=2
```

```
{
  "meta": {
    "model": "user",
    "next": "",
    "page": 1,
    "previous": ""
    },
"objects": [{
    "username": "admin",
    "admin": true,
    "email": "admin@admin",
    "active": true,
    "password": "214de$25",
    "id": 1
    }]
}
```

Valid Comparison Operators are: 'eq', 'lt', 'lte', 'gt', 'gte', 'ne', 'in', 'is', 'like', 'ilike'

6.8 Sorting results

Results can be sorted by specifying an `ordering` as a GET argument. The ordering must be a column on the model.

/api/message/?ordering=pub_date

If you would like to order objects "descending", place a "-" (hyphen character) before the column name:

/api/message/?ordering=-pub_date

6.9 Limiting results and pagination

By default, resources are paginated 20 per-page. If you want to return less, you can specify a `limit` in the querystring.

/api/message/?limit=2

In the "meta" section of the response, URIs for the "next" and "previous" sets of results are available:

```
meta: {
  model: "message"
  next: "/api/message/?limit=1&page=3"
  page: 2
  previous: "/api/message/?limit=1&page=1"
}
```

CHAPTER 7

Utilities

flask-peewee ships with several useful utilities. If you're coming from the django world, some of these functions may look familiar to you.

7.1 Getting objects

get_object_or_404()

Provides a handy way of getting an object or 404ing if not found, useful for urls that match based on ID.

```
@app.route('/blog/<title>/')
def blog_detail(title):
    blog = get_object_or_404(Blog.select().where(Blog.active==True), Blog.
↪title==title)
    return render_template('blog/detail.html', blog=blog)
```

object_list()

Wraps the given query and handles pagination automatically. Pagination defaults to `20` but can be changed by passing in `paginate_by=XX`.

```
@app.route('/blog/')
def blog_list():
    active = Blog.select().where(Blog.active==True)
    return object_list('blog/index.html', active)
```

```
<!-- template -->
{% for blog in object_list %}
  {# render the blog here #}
{% endfor %}

{% if page > 1 %}
  <a href="./?page={{ page - 1 }}">Prev</a>
```

(continues on next page)

(continued from previous page)

```
{% endif %}
{% if page < pagination.get_pages() %}
  <a href="./?page={{ page + 1 }}">Next</a>
{% endif %}
```

PaginatedQuery

A wrapper around a query (or model class) that handles pagination.

Example:

```
query = Blog.select().where(Blog.active==True)
pq = PaginatedQuery(query)

# assume url was /?page=3
obj_list = pq.get_list()  # returns 3rd page of results

pq.get_page() # returns "3"

pq.get_pages() # returns total objects / objects-per-page
```

7.2 Misc

slugify(*string*)
Convert a string into something suitable for use as part of a URL, e.g. "This is a url" becomes "this-is-a-url"

```
from flask_peewee.utils import slugify


class Blog(db.Model):
    title = CharField()
    slug = CharField()

    def save(self, *args, **kwargs):
        self.slug = slugify(self.title)
        super(Blog, self).save(*args, **kwargs)
```

make_password(*raw_password*)
Create a salted hash for the given plain-text password

check_password(*raw_password*, *enc_password*)
Compare a plain-text password against a salted/hashed password

CHAPTER 8

Using gevent

If you would like to serve your flask application using gevent, there are two small settings you will need to add.

8.1 Database configuration

Instruct peewee to store connection information in a thread local:

```
# app configuration
DATABASE = {
    'name': 'my_db',
    'engine': 'peewee.PostgresqlDatabase',
    'user': 'postgres',
    'threadlocals': True, # <-- this
}
```

8.2 Monkey-patch the thread module

Some time before instantiating a `Database` object (and preferrably at the very “beginning” of your code) you will want to monkey-patch the standard library thread module:

```
from gevent import monkey; monkey.patch_thread()
```

If you want to patch everything (recommended):

```
from gevent import monkey; monkey.patch_all()
```

Note: Remember to monkey-patch before initializing your app

8.3 Rationale

flask-peewee opens a connection-per-request. Flask stores things, like “per-request” information, in a special object called a context local. Flask will ensure that this works even in a greened environment. Peewee does not automatically work in a “greened” environment, and stores connection state on the database instance in a local. Peewee can use a thread local instead, which ensures connections are not shared across threads. When using peewee with gevent, it is necessary to make this “threadlocal” a “greenlet local” by monkeypatching the thread module.

API in depth:

CHAPTER 9

API

9.1 Admin

class Admin(*app*, *auth*[, *blueprint_factory*[, *template_helper*[, *prefix*]]])

Class used to expose an admin area at a certain url in your application. The Admin object implements a flask blueprint and acts as the central registry for models and panels you wish to expose in the admin.

The Admin object coordinates the registration of models and panels and provides a method for ensuring a user has permission to access the admin area.

The Admin object requires an `Auth` instance when being instantiated, which in turn requires a Flask app and a py:class:*Database* wrapper.

Here is an example of how you might instantiate an Admin object:

```
from flask import Flask

from flask_peewee.admin import Admin
from flask_peewee.auth import Auth
from flask_peewee.db import Database

app = Flask(__name__)
db = Database(app)

# needed for authentication
auth = Auth(app, db)

# instantiate the Admin object for our project
admin = Admin(app, auth)
```

Parameters

- **app** – flask application to bind admin to
- **auth** – `Auth` instance which will provide authentication

- **blueprint_factory** – an object that will create the `BluePrint` used by the admin
- **template_helper** – a subclass of `AdminTemplateHelper` that provides helpers and context to used by the admin templates
- **prefix** – url to bind admin to, defaults to `/admin`

register (*model*[, *admin_class=ModelAdmin*])

Register a model to expose in the admin area. A *ModelAdmin* subclass can be provided along with the model, allowing for customization of the model's display and behavior.

Example usage:

```
# will use the default ModelAdmin subclass to display model
admin.register(BlogModel)

class EntryAdmin(ModelAdmin):
    columns = ('title', 'blog', 'pub_date',)

admin.register(EntryModel, EntryAdmin)
```

Warning: All models must be registered before calling *setup()*

Parameters

- **model** – peewee model to expose via the admin
- **admin_class** – *ModelAdmin* or subclass to use with given model

register_panel (*title*, *panel*)

Register a *AdminPanel* subclass for display in the admin dashboard.

Example usage:

```
class HelloWorldPanel(AdminPanel):
    template_name = 'admin/panels/hello.html'

    def get_context(self):
        return {
            'message': 'Hello world',
        }

admin.register_panel('Hello world', HelloWorldPanel)
```

Warning: All panels must be registered before calling *setup()*

Parameters

- **title** – identifier for panel, example might be "Site Stats"
- **panel** – subclass of *AdminPanel* to display

setup ()

Configures urls for models and panels, then registers blueprint with the Flask application. Use this method when you have finished registering all the models and panels with the admin object, but before starting the WSGI application. For a sample implementation, check out `example/main.py` in the example application supplied with flask-peewee.

```
# register all models, etc
admin.register(...)

# finish up initialization of the admin object
admin.setup()

if __name__ == '__main__':
    # run the WSGI application
    app.run()
```

Note: call `setup()` **after** registering your models and panels

`check_user_permission`(*user*)

Check whether the given user has permission to access to the admin area. The default implementation simply checks whether the `admin` field is checked, but you can provide your own logic.

This method simply controls access to the admin area as a whole. In the event the user is **not** permitted to access the admin (this function returns `False`), they will receive a HTTP Response Forbidden (403).

Default implementation:

```
def check_user_permission(self, user):
    return user.admin
```

Parameters **`user`** – the currently logged-in user, exposed by the *`Auth`* instance

Return type Boolean

`auth_required`(*func*)

Decorator that ensures the requesting user has permission. The implementation first checks whether the requesting user is logged in, and if not redirects to the login view. If the user *is* logged in, it calls *`check_user_permission()`*. Only if this call returns `True` is the actual view function called.

`get_urls`()

Get a tuple of 2-tuples mapping urls to view functions that will be exposed by the admin. The default implementation looks like this:

```
def get_urls(self):
    return (
        ('/', self.auth_required(self.index)),
    )
```

This method provides an extension point for providing any additional "global" urls you would like to expose.

Note: Remember to decorate any additional urls you might add with *`auth_required()`* to ensure they are not accessible by unauthenticated users.

9.1.1 Exposing Models with the ModelAdmin

class ModelAdmin

Class that determines how a peewee `Model` is exposed in the admin area. Provides a way of encapsulating model-specific configuration and behaviors. Provided when registering a model with the *Admin* instance (see *Admin.register()*).

columns

List or tuple of columns should be displayed in the list index. By default if no columns are specified the `Model`'s `__unicode__()` will be used.

Note: Valid values for columns are the following:

- field on a model
- attribute on a model instance
- callable on a model instance (called with no parameters)

If a column is a model field, it will be sortable.

```
class EntryAdmin(ModelAdmin):
    columns = ['title', 'pub_date', 'blog']
```

filter_exclude

Exclude certain fields from being exposed as filters. Related fields can be excluded using "__" notation, e.g. `user__password`

filter_fields

Only allow filtering on the given fields

exclude

A list of field names to exclude from the "add" and "edit" forms

fields

Only display the given fields on the "add" and "edit" form

paginate_by = 20

Number of records to display on index pages

filter_paginate_by = 15

Default pagination when filtering in a modal dialog

delete_collect_objects = True

Collect and display a list of "dependencies" when deleting

delete_recursive = True

Delete "dependencies" recursively

get_query()

Determines the list of objects that will be exposed in the admin. By default this will be all objects, but you can use this method to further restrict the query.

This method is called within the context of a request, so you can access the `Flask.request` object or use the *Auth* instance to determine the currently-logged-in user.

Here's an example showing how the query is restricted based on whether the given user is a "super user" or not:

```python
class UserAdmin(ModelAdmin):
    def get_query():
        # ask the auth system for the currently logged-in user
        current_user = self.auth.get_logged_in_user()

        # if they are not a superuser, only show them their own
        # account in the admin
        if not current_user.is_superuser:
            return User.select().where(User.id==current_user.id)

        # otherwise, show them all users
        return User.select()
```

Return type A `SelectQuery` that represents the list of objects to expose

get_object(*pk*)
This method retrieves the object matching the given primary key. The implementation uses *get_query()* to retrieve the base list of objects, then queries within that for the given primary key.

Return type The model instance with the given pk, raising a `DoesNotExist` in the event the model instance does not exist.

get_form([*adding=False*])
Provides a useful extension point in the event you want to define custom fields or custom validation behavior.

Parameters **adding** (*boolean*) – indicates whether adding a new instance or editing existing

Return type A wtf-peewee Form subclass that will be used when adding or editing model instances in the admin.

get_add_form()
Allows you to specify a different form when adding new instances versus editing existing instances. The default implementation simply calls *get_form()*.

get_edit_form()
Allows you to specify a different form when editing existing instances versus adding new instances. The default implementation simply calls *get_form()*.

get_filter_form()
Provide a special form for use when filtering the list of objects in the model admin's index/export views. This form is slightly different in that it is tailored for use when filtering the list of models.

Return type A special Form instance (`FilterForm`) that will be used when filtering the list of objects in the index view.

save_model(*instance*, *form*, *adding=False*)
Method responsible for persisting changes to the database. Called by both the add and the edit views.

Here is an example from the default `auth.User` *ModelAdmin*, in which the password is displayed as a sha1, but if the user is adding or edits the existing password, it re-hashes:

```python
def save_model(self, instance, form, adding=False):
    orig_password = instance.password

    user = super(UserAdmin, self).save_model(instance, form, adding)

    if orig_password != form.password.data:
        user.set_password(form.password.data)
```

(continues on next page)

(continued from previous page)

```
        user.save()

    return user
```

Parameters

- **instance** – an unsaved model instance
- **form** – a validated form instance
- **adding** – boolean to indicate whether we are adding a new instance or saving an existing

get_template_overrides()

Hook for specifying template overrides. Should return a dictionary containing view names as keys and template names as values. Possible choices for keys are:

- index
- add
- edit
- delete
- export

```
class UserModelAdmin(ModelAdmin):
    def get_template_overrides(self):
        return {'index': 'users/admin/index_override.html'}
```

get_urls()

Useful as a hook for extending *ModelAdmin* functionality with additional urls.

Note: It is not necessary to decorate the views specified by this method since the *Admin* instance will handle this during registration and setup.

Return type tuple of 2-tuples consisting of a mapping between url and view

get_url_name(*name*)

Since urls are namespaced, this function provides an easy way to get full urls to views provided by this ModelAdmin

process_filters(*query*)

Applies any filters specified by the user to the given query, returning metadata about the filters.

Returns a 4-tuple containing:

- special `Form` instance containing fields for filtering
- filtered query
- a list containing the currently selected filters
- a tree-structure containing the fields available for filtering (`FieldTreeNode`)

Return type A tuple as described above

9.1.2 Extending admin functionality using AdminPanel

class AdminPanel

Class that provides a simple interface for providing arbitrary extensions to the admin. These are displayed as "panels" on the admin dashboard with a customizable template. They may additionally, however, define any views and urls. These views will automatically be protected by the same authentication used throughout the admin area.

Some example use-cases for AdminPanels might be:

- Display some at-a-glance functionality in the dashboard, like stats on new user signups.
- Provide a set of views that should only be visible to site administrators, for example a mailing-list app.
- Control global site settings, turn on and off features, etc.

template_name

What template to use to render the panel in the admin dashboard, defaults to `'admin/panels/default.html'`.

get_urls()

Useful as a hook for extending `AdminPanel` functionality with custom urls and views.

Note: It is not necessary to decorate the views specified by this method since the `Admin` instance will handle this during registration and setup.

Return type Returns a tuple of 2-tuples mapping url to view

get_url_name(*name*)

Since urls are namespaced, this function provides an easy way to get full urls to views provided by this panel

Parameters **name** – string representation of the view function whose url you want

Return type String representing url

```
<!-- taken from example -->
<!-- will return something like /admin/notes/create/ -->
{{ url_for(panel.get_url_name('create')) }}
```

get_template_name()

Return the template used to render this panel in the dashboard. By default simply returns the template stored under `AdminPanel.template_name`.

get_context()

Return the context to be used when rendering the dashboard template.

Return type Dictionary

render()

Render the panel template with the context – this is what gets displayed in the admin dashboard.

9.2 Auth

class Auth(*app*, *db*[, *user_model=None*[, *prefix='/accounts'*]], *db_table='user'*)

The class that provides methods for authenticating users and tracking users across requests. It also provides a model for persisting users to the database, though this can be customized.

The auth framework is used by the *Admin* and can also be integrated with the *RestAPI*.

Here is an example of how to use the Auth framework:

```python
from flask import Flask

from flask_peewee.auth import Auth
from flask_peewee.db import Database

app = Flask(__name__)
db = Database(app)

# needed for authentication
auth = Auth(app, db)

# mark a view as requiring login
@app.route('/private/')
@auth.login_required
def private_timeline():
    # get the currently-logged-in user
    user = auth.get_logged_in_user()
```

Unlike the *Admin* or the *RestAPI*, there is no explicit `setup()` method call when using the Auth system. Creation of the auth blueprint and registration with the Flask app happen automatically during instantiation.

Note: A context processor is automatically registered that provides the currently logged-in user across all templates, available as "user". If no user is logged in, the value of this will be `None`.

Note: A pre-request handler is automatically registered which attempts to retrieve the current logged-in user and store it on the global flask variable `g`.

Parameters

- **app** – flask application to bind admin to
- **db** – *Database* database wrapper for flask app
- **user_model** – `User` model to use
- **prefix** – url to bind authentication views to, defaults to /accounts/
- **db_table** – Create db table using db_table name. `user` is reserved keyword in postgres.

default_next_url = 'homepage'

The url to redirect to upon successful login in the event a `?next=<xxx>` is not provided.

get_logged_in_user()

Note: Since this method relies on the session storage to track users across requests, this method must be called while within a `RequestContext`.

Return type returns the currently logged-in `User`, or `None` if session is anonymous

login_required(*func*)

Function decorator that ensures a view is only accessible by authenticated users. If the user is not authed they are redirected to the login view.

Note: this decorator should be applied closest to the original view function

```
@app.route('/private/')
@auth.login_required
def private():
    # this view is only accessible by logged-in users
    return render_template('private.html')
```

Parameters **func** – a view function to be marked as login-required

Return type if the user is logged in, return the view as normal, otherwise returns a redirect to the login page

get_user_model()

Return type Peewee model to use for persisting user data and authentication

get_model_admin([*model_admin=None*])

Provide a *ModelAdmin* class suitable for use with the User model. Specifically addresses the need to re-hash passwords when changing them via the admin.

The default implementation includes an override of the *ModelAdmin.save_model()* method to intelligently hash passwords:

```
class UserAdmin(model_admin):
    columns = ['username', 'email', 'active', 'admin']

    def save_model(self, instance, form, adding=False):
        orig_password = instance.password

        user = super(UserAdmin, self).save_model(instance, form, adding)

        if orig_password != form.password.data:
            user.set_password(form.password.data)
            user.save()

        return user
```

Parameters **model_admin** – subclass of *ModelAdmin* to use as the base class

Return type a subclass of *ModelAdmin* suitable for use with the `User` model

get_urls()

A mapping of url to view. The default implementation provides views for login and logout only, but you might extend this to add registration and password change views.

Default implementation:

```
def get_urls(self):
    return (
        ('/logout/', self.logout),
        ('/login/', self.login),
    )
```

Return type a tuple of 2-tuples mapping url to view function.

get_login_form()

Return type a wtforms.Form subclass to use for retrieving any user info required for login

authenticate(*username*, *password*)
Given the username and password, retrieve the user with the matching credentials if they exist. No exceptions should be raised by this method.

Return type User model if successful, otherwise False

login_user(*user*)
Mark the given user as "logged-in". In the default implementation, this entails storing data in the Session to indicate the successful login.

Parameters **user** – User instance

logout_user(*user*)
Mark the requesting user as logged-out

Parameters **user** – User instance

9.2.1 The BaseUser mixin

class BaseUser
Provides default implementations for password hashing and validation. The auth framework requires two methods be implemented by the User model. A default implementation of these methods is provided by the BaseUser mixin.

set_password(*password*)
Encrypts the given password and stores the encrypted version on the model. This method is useful when registering a new user and storing the password, or modifying the password when a user elects to change.

check_password(*password*)
Verifies if the given plaintext password matches the encrypted version stored on the model. This method on the User model is called specifically by the *Auth.authenticate()* method.

Return type Boolean

9.3 Database

class Database(*app*)
The database wrapper provides integration between the peewee ORM and flask. It reads database configuration information from the flask app configuration and manages connections across requests.

The db wrapper also provides a Model subclass which is configured to work with the database specified by the application's config.

To configure the database specify a database engine and name:

```
DATABASE = {
    'name': 'example.db',
    'engine': 'peewee.SqliteDatabase',
}
```

Here is an example of how you might use the database wrapper:

```
# instantiate the db wrapper
db = Database(app)

# start creating models
class Blog(db.Model):
    # this model will automatically work with the database specified
    # in the application's config.
```

Parameters **app** – flask application to bind admin to

Model
Model subclass that works with the database specified by the app's config

9.4 REST API

class RestAPI(*app*[, *prefix='/api'*[, *default_auth=None*[, *name='api'*]]])

The *RestAPI* acts as a container for the various *RestResource* objects. By default it binds all resources to `/api/<model-name>/`. Much like the *Admin*, it is a centralized registry of resources.

Example of creating a `RestAPI` instance for a flask app:

```
from flask_peewee.rest import RestAPI

from app import app # our project's Flask app

# instantiate our api wrapper
api = RestAPI(app)

# register a model with the API
api.register(SomeModel)

# configure URLs
api.setup()
```

Note: Like the flask admin, the `RestAPI` has a `setup()` method which must be called after all resources have been registered.

Parameters

- **app** – flask application to bind API to
- **prefix** – url to serve REST API from
- **default_auth** – default *Authentication* type to use with registered resources
- **name** – the name for the API blueprint

register(*model*[, *provider=RestResource*[, *auth=None*[, *allowed_methods=None*]]])

Register a model to expose via the API.

Parameters

- **model** – `Model` to expose via API
- **provider** – subclass of *RestResource* to use for this model

- **auth** – authentication type to use for this resource, falling back to `RestAPI.default_auth`
- **allowed_methods** – `list` of HTTP verbs to allow, defaults to `['GET', 'POST', 'PUT', 'DELETE']`

setup()

Register the API `BluePrint` and configure urls.

Warning: This must be called **after** registering your resources.

9.4.1 RESTful Resources and their subclasses

class RestResource(*rest_api*, *model*, *authentication*[, *allowed_methods=None*])

Class that determines how a peewee `Model` is exposed by the Rest API. Provides a way of encapsulating model-specific configuration and behaviors. Provided when registering a model with the *RestAPI* instance (see *RestAPI.register()*).

Should not be instantiated directly in most cases. Instead should be "registered" with a `RestAPI` instance.

Example usage:

```
# instantiate our api wrapper, passing in a reference to the Flask app
api = RestAPI(app)

# create a RestResource subclass
class UserResource(RestResource):
    exclude = ('password', 'email',)

# assume we have a "User" model, register it with the custom resource
api.register(User, UserResource)
```

paginate_by = 20

Determines how many results to return for a given API query.

Note: *Fewer* results can be requested by specifying a `limit`, but `paginate_by` is the upper bound.

fields = None

A list or tuple of fields to expose when serializing

exclude = None

A list or tuple of fields to **not** expose when serializing

filter_exclude

A list of fields that **cannot** be used to filter API results

filter_fields

A list of fields that can be used to filter the API results

filter_recursive = True

Allow filtering on related resources

include_resources

A mapping of field name to resource class for handling of foreign-keys. When provided, foreign keys will be "nested".

```
class UserResource(RestResource):
    exclude = ('password', 'email')

class MessageResource(RestResource):
    include_resources = {'user': UserResource} # 'user' is a foreign key field
```

```
/* messages without "include_resources" */
{
  "content": "flask and peewee, together at last!",
  "pub_date": "2011-09-16 18:36:15",
  "id": 1,
  "user": 2
},

/* messages with "include_resources = {'user': UserResource} */
{
  "content": "flask and peewee, together at last!",
  "pub_date": "2011-09-16 18:36:15",
  "id": 1,
  "user": {
    "username": "coleifer",
    "active": true,
    "join_date": "2011-09-16 18:35:56",
    "admin": false,
    "id": 2
  }
}
```

delete_recursive = True
Recursively delete dependencies

get_query()
Returns the list of objects to be exposed by the API. Provides an easy hook for restricting objects:

```
class UserResource(RestResource):
    def get_query(self):
        # only return "active" users
        return self.model.select().where(active=True)
```

Return type a `SelectQuery` containing the model instances to expose

prepare_data(*obj*, *data*)
This method provides a hook for modifying outgoing data. The default implementation no-ops, but you could do any kind of munging here. The data returned by this method is passed to the serializer before being returned as a json response.

Parameters

- **obj** – the object being serialized
- **data** – the dictionary representation of a model returned by the `Serializer`

Return type a dictionary of data to hand off

save_object(*instance*, *raw_data*)
Persist the instance to the database. The raw data supplied by the request is also available, but at the time this method is called the instance has already been updated and populated with the incoming data.

Parameters

- **instance** – `Model` instance that has already been updated with the incoming `raw_data`
- **raw_data** – data provided in the request

Return type a saved instance

api_list()

A view that dispatches based on the HTTP verb to either:

- GET: *object_list()*
- POST: *create()*

Return type `Response`

api_detail(*pk*)

A view that dispatches based on the HTTP verb to either:

- GET: *object_detail()*
- PUT: *edit()*
- DELETE: *delete()*

Return type `Response`

object_list()

Returns a serialized list of `Model` instances. These objects may be filtered, ordered, and/or paginated.

Return type `Response`

object_detail()

Returns a serialized `Model` instance.

Return type `Response`

create()

Creates a new `Model` instance based on the deserialized POST body.

Return type `Response` containing serialized new object

edit()

Edits an existing `Model` instance, updating it with the deserialized PUT body.

Return type `Response` containing serialized edited object

delete()

Deletes an existing `Model` instance from the database.

Return type `Response` indicating number of objects deleted, i.e. `{'deleted': 1}`

get_api_name()

Return type URL-friendly name to expose this resource as, defaults to the model's name

check_get([*obj=None*])

A hook for pre-authorizing a GET request. By default returns `True`.

Return type Boolean indicating whether to allow the request to continue

check_post()

A hook for pre-authorizing a POST request. By default returns `True`.

Return type Boolean indicating whether to allow the request to continue

check_put (*obj*)
A hook for pre-authorizing a PUT request. By default returns `True`.

Return type Boolean indicating whether to allow the request to continue

check_delete (*obj*)
A hook for pre-authorizing a DELETE request. By default returns `True`.

Return type Boolean indicating whether to allow the request to continue

class RestrictOwnerResource (*RestResource*)
This subclass of `RestResource` allows only the "owner" of an object to make changes via the API. It works by verifying that the authenticated user matches the "owner" of the model instance, which is specified by setting `owner_field`.

Additionally, it sets the "owner" to the authenticated user whenever saving or creating new instances.

owner_field = 'user'
Field on the model to use to verify ownership of the given instance.

validate_owner (*user*, *obj*)

Parameters

- **user** – an authenticated `User` instance
- **obj** – the `Model` instance being accessed via the API

Return type Boolean indicating whether the user can modify the object

set_owner (*obj*, *user*)
Mark the object as being owned by the provided user. The default implementation simply calls `setattr`.

Parameters

- **obj** – the `Model` instance being accessed via the API
- **user** – an authenticated `User` instance

9.4.2 Authenticating requests to the API

class Authentication (*[protected_methods=None]*)
Not to be confused with the `auth.Authentication` class, this class provides a single method, `authorize`, which is used to determine whether to allow a given request to the API.

Parameters protected_methods – A list or tuple of HTTP verbs to require auth for

authorize ()
This single method is called per-API-request.

Return type Boolean indicating whether to allow the given request through or not

class UserAuthentication (*auth[, protected_methods=None]*)
Authenticates API requests by requiring the requesting user be a registered `auth.User`. Credentials are supplied using HTTP basic auth.

Example usage:

```
from auth import auth # import the Auth object used by our project

from flask_peewee.rest import RestAPI, RestResource, UserAuthentication

# create an instance of UserAuthentication
```

(continues on next page)

(continued from previous page)

```
user_auth = UserAuthentication(auth)

# instantiate our api wrapper, specifying user_auth as the default
api = RestAPI(app, default_auth=user_auth)

# create a special resource for users that excludes email and password
class UserResource(RestResource):
    exclude = ('password', 'email',)

# register our models so they are exposed via /api/<model>/
api.register(User, UserResource) # specify the UserResource

# configure the urls
api.setup()
```

Parameters

- **`auth`** – an *Authentication* instance
- **`protected_methods`** – A list or tuple of HTTP verbs to require auth for

`authorize`()

Verifies, using HTTP Basic auth, that the username and password match a valid `auth.User` model before allowing the request to continue.

Return type Boolean indicating whether to allow the given request through or not

class `AdminAuthentication`(*auth*[, *protected_methods=None*])

Subclass of the `UserAuthentication` that further restricts which users are allowed through. The default implementation checks whether the requesting user is an "admin" by checking whether the admin attribute is set to `True`.

Example usage:

```

```

Authenticates API requests by requiring the requesting user be a registered `auth.User`. Credentials are supplied using HTTP basic auth.

Example usage:

```
from auth import auth # import the Auth object used by our project

from flask_peewee.rest import RestAPI, RestResource, UserAuthentication,␣
↪AdminAuthentication

# create an instance of UserAuthentication and AdminAuthentication
user_auth = UserAuthentication(auth)
admin_auth = AdminAuthentication(auth)

# instantiate our api wrapper, specifying user_auth as the default
api = RestAPI(app, default_auth=user_auth)

# create a special resource for users that excludes email and password
class UserResource(RestResource):
    exclude = ('password', 'email',)

```

(continues on next page)

(continued from previous page)

```
# register our models so they are exposed via /api/<model>/
api.register(SomeModel)

# specify the UserResource and require the requesting user be an admin
api.register(User, UserResource, auth=admin_auth)

# configure the urls
api.setup()
```

verify_user(*user*)

Verifies whether the requesting user is an administrator

Parameters **user** – the `auth.User` instance of the requesting user

Return type Boolean indicating whether the user is an administrator

class APIKeyAuthentication(*model*, *protected_methods=None*)

Subclass that allows you to provide an API Key model to authenticate requests with.

Note: Must provide an API key model with at least the following two fields:

- key
- secret

```
# example API key model
class APIKey(db.Model):
    key = CharField()
    secret = CharField()
    user = ForeignKeyField(User)

# instantiating the auth
api_key_auth = APIKeyAuthentication(model=APIKey)
```

Parameters

- **model** – a `Database.Model` subclass to persist API keys.
- **protected_methods** – A list or tuple of HTTP verbs to require auth for

9.5 Utilities

get_object_or_404(*query_or_model*, **query*)

Provides a handy way of getting an object or 404ing if not found, useful for urls that match based on ID.

Parameters

- **query_or_model** – a query or model to filter using the given expressions
- **query** – a list of query expressions

```
@app.route('/blog/<title>/')
def blog_detail(title):
    blog = get_object_or_404(Blog.select().where(Blog.active==True), Blog.
↪title==title)
    return render_template('blog/detail.html', blog=blog)
```

object_list (*template_name*, *qr*[, *var_name='object_list'*[, ***kwargs*]])

Wraps the given query and handles pagination automatically. Pagination defaults to `20` but can be changed by passing in `paginate_by=XX`.

Parameters

- **template_name** – template to render
- **qr** – a select query
- **var_name** – the template variable name to use for the paginated query
- **kwargs** – arbitrary context to pass in to the template

```
@app.route('/blog/')
def blog_list():
    active = Blog.select().where(Blog.active==True)
    return object_list('blog/index.html', active)
```

```
<!-- template -->
{% for blog in object_list %}
  {# render the blog here #}
{% endfor %}

{% if page > 1 %}
  <a href="./?page={{ page - 1 }}">Prev</a>
{% endif %}
{% if page < pagination.get_pages() %}
  <a href="./?page={{ page + 1 }}">Next</a>
{% endif %}
```

get_next ()

Return type a URL suitable for redirecting to

slugify (*s*)

Use a regular expression to make arbitrary string `s` URL-friendly

Parameters **s** – any string to be slugified

Return type url-friendly version of string `s`

class PaginatedQuery (*query_or_model*, *paginate_by*)

A wrapper around a query (or model class) that handles pagination.

page_var = 'page'

The URL variable used to store the current page

Example:

```
query = Blog.select().where(Blog.active==True)
pq = PaginatedQuery(query)

# assume url was /?page=3
obj_list = pq.get_list()  # returns 3rd page of results

pq.get_page() # returns "3"

pq.get_pages() # returns total objects / objects-per-page
```

get_list ()

Return type a list of objects for the request page

get_page()

Return type an integer representing the currently requested page

get_pages()

Return type the number of pages in the entire result set

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Index