

CHAPTER 8

Example: A Social Blogging Application

User Authentication

Most applications need to keep track of who their users are. When users connect with an application, they *authenticate* with it, a process by which they make their identity known. Once the application knows who the user is, it can offer a customized experience.

Authentication Extensions for Flask

This is the list of packages that will be used, and what they're used for:

- Flask-Login: Management of user sessions for logged-in users
- Werkzeug: Password hashing and verification
- itsdangerous: Cryptographically secure token generation and verification

In addition to authentication-specific packages, the following general-purpose extensions will be used:

- Flask-Mail: Sending of authentication-related emails
 - Flask-Bootstrap: HTML templates
 - Flask-WTF: Web forms
-

Hashing Passwords with Werkzeug

Werkzeug's *security* module conveniently implements secure password hashing. This functionality is exposed with just two functions, used in the registration and verification phases, respectively:

`generate_password_hash(password, method='pbkdf2:sha256', salt_length=8)` This function takes a plain-text password and returns the password hash as a string that can be stored in the user database. The default values for `method` and `salt_length` are sufficient for most use cases.

`check_password_hash(hash, password)`
This function takes a password hash previously stored in the database and the password entered by the user. A return value of `True` indicates that the user password is correct.

Example 8-1 shows the changes to the User model created in **Chapter 5** to accommodate password hashing.

Example 8-1. app/models.py: password hashing in the User model

```
from werkzeug.security import generate_password_hash, check_password_hash

class User(db.Model):
    # ...
    password_hash = db.Column(db.String(128))

    @property
    def password(self):
        raise AttributeError('password is not a readable attribute')

    @password.setter
    def password(self, password):
        self.password_hash = generate_password_hash(password)

    def verify_password(self, password):
        return check_password_hash(self.password_hash, password)
```

The password hashing functionality is now complete and can be tested in the shell:

```
(venv) $ flask shell
```

```
>>> u = User()
```

```
>>> u.password = 'cat'
```

```
>>> u.password
```

```
Traceback (most recent call last):
```

```
File "<console>", line 1, in <module>
```

```
File "/home/flask/flasky/app/models.py", line 24, in password raise
```

```
AttributeError('password is not a readable attribute')
```

```
AttributeError: password is not a readable attribute
```

```
>>> u.password_hash
```

```
'pbkdf2:sha256:50000$moHwFH1B$ef1574909f9c549285e8547cad181c5e0213cfa44a4aba4349 fa830aalfd227f'
```

```
>>> u.verify_password('cat')
```

```
True
```

```
>>> u.verify_password('dog')
```

```
False
```

```
>>> u2 = User()
```

```
>>> u2.password = 'cat'
```

```
>>> u2.password_hash
```

```
'pbkdf2:sha256:50000$Pfz0m0KU$27be930b7f0e0119d38e8d8a62f7f5e75c0a7db61ae16709bc aa6cfd60c44b74'
```

Example 8-2. tests/test_user_model.py: password hashing tests

```
import unittest
from app.models import User

class UserModelTestCase(unittest.TestCase):
    def test_password_setter(self):
        u = User(password = 'cat' )
        self.assertTrue(u.password_hash is not None)

    def test_no_password_getter(self):
        u = User(password = 'cat')
        with self.assertRaises(AttributeError):
            u.password

    def test_password_verification(self):
        u = User(password = 'cat')
        self.assertTrue(u.verify_password('cat'))
        self.assertFalse(u.verify_password('dog' ))
    def test_password_salts_are_random(self):
        u = User(password='cat')
        u2 = User(password='cat' )
        self.assertTrue(u.password_hash != u2.password_hash)
```

To run these new unit tests, use the following command:

```
(venv) $ flask test
```

Creating an Authentication Blueprint

The `auth` blueprint will be hosted in a Python package with the same name. The blueprint's package constructor creates the blueprint object and imports routes from a `views.py` module. This is shown in [Example 8-3](#).

Example 8-3. `app/auth/_init_.py`: authentication blueprint creation

```
from flask import Blueprint

auth = Blueprint('auth', __name__) from . import views
```

The `app/auth/views.py` module, shown in [Example 8-4](#), imports the blueprint and defines the routes associated with authentication using its route decorator. For now, a `/login` route is added, which renders a placeholder template of the same name.

Example 8-4. app/auth/views.py: authentication blueprint routes and view functions

```
from flask import render_template
from . import auth

@auth.route('/login')
def login():
    return render_template('auth/login.html')
```

Example 8-5. app/_init_.py: authentication blueprint registration

```
def create_app(config_name):
    # ...
    from .auth import auth as auth_blueprint
    app.register_blueprint(auth_blueprint, url_prefix='/auth')

    return app
```

User Authentication with Flask-Login

To begin, the extension needs to be installed in the virtual environment:

```
(venv) $ pip install flask-login
```

Preparing the User Model for Logins

Example 8-6. app/models.py: updates to the User model to support user logins

```
from flask_login import UserMixin

class User(UserMixin, db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key = True)
    email = db.Column(db.String(64), unique=True, index=True)
    username = db.Column(db.String(64), unique=True, index=True)
    password_hash = db.Column(db.String(128))
    role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))
```

Example 8-7. app/_init_.py: Flask-Login initialization

```
from flask_login import LoginManager

login_manager = LoginManager() login_manager.login_view =
'auth.login'

def create_app(config_name):
    # ...
    login_manager.init_app(app)
    # ...
```

Example 8-8. app/models.py: user loader function `from . import`

```
login_manager @login_manager.user_loader

def load_user(user_id):
    return User.query.get(int(user_id))
```

Protecting Routes

To protect a route so that it can only be accessed by authenticated users, Flask-Login provides a `login_required` decorator. An example of its usage follows:

```

from flask_login import login_required

@app.route('/secret') @login_required
def secret():
    return 'Only authenticated users are allowed!'

```

You can see from this example that it is possible to “chain” multiple function decorators. When two or more decorators are added to a function, each decorator only affects those that are below it, in addition to the target function.

Adding a Login Form

Example 8-9. app/auth/forms.py: login form

```

from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import DataRequired, Length, Email

class LoginForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Length(1, 64),
                                           Email()])
    password = PasswordField('Password', validators=[DataRequired()])
    remember_me = BooleanField('Keep me logged in')
    submit = SubmitField('Log In')

```

The `PasswordField` class represents an `<input>` element with `type="password"`. The `BooleanField` class represents a checkbox.

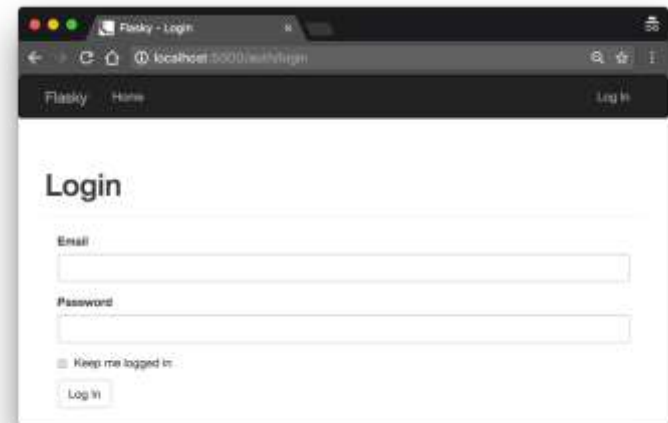
The email field uses the `Length()` and `Email()` validators from WTForms in addition to `DataRequired()`, to ensure that the user not only provides a value for this field, but that it is valid.

Figure 8-1. Login form

The navigation bar in the *base.html* template uses a Jinja2 conditional to display “Log In” or “Log Out” links depending on the logged-in state of the current user. The conditional is shown in **Example 8-10**.

Example 8-10. *app/templates/base.html*: Log In and Log Out navigation bar links

```
<ul class="nav navbar-nav navbar-right">
  {% if current_user.is_authenticated %}
  <li><a href="{{ url_for('auth.logout') }}">Log Out</a></li>
  {% else %}
  <li><a href="{{ url_for('auth.login') }}">Log In</a></li>
  {% endif %}
</ul>
```



Signing Users In

Example 8-11. app/auth/views.py: login route

```
from flask import render_template, redirect, request, url_for, flash
from flask_login import login_user
from . import auth
from ..models import User
from .forms import LoginForm

@auth.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        if user is not None and user.verify_password(form.password.data):
            login_user(user, form.remember_me.data)
            next = request.args.get('next')
            if next is None or not next.startswith('/'): next =
                url_for('main.index')
            return redirect(next) flash('Invalid username or
password.')
    return render_template('auth/login.html', form=form)
```

Example 8-12. app/templates/auth/login.html: login form template

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}

{% block title %}Flasky - Login{% endblock %}

{% block page_content %}
<div class="page-header">
  <h1>Login</h1>
</div>
<div class="col-md-4">
  {{ wtf.quick_form(form) }}
</div>
{% endblock %}
```

Signing Users Out

Example 8-13. app/auth/views.py: logout route

```
from flask_login import logout_user, login_required

@auth.route('/logout')
```

```
@login_required
def logout():
    logout_user()
    flash('You have been logged out.')
    return redirect(url_for('main.index'))
```

Testing Logins

To verify that the login functionality is working, the home page can be updated to greet the logged-in user by name.

Example 8-14. app/templates/index.html: greeting the logged-in user

```
Hello,
{% if current_user.is_authenticated %}
    {{ current_user.username }}
{% else %} Stranger {% endif %}!
```

Because no user registration functionality has been built, a new user can only be registered from the shell at this time:

```
(venv) $ flask shell
>>> u = User(email='john@example.com', username='john', password='cat')
>>> db.session.add(u)
>>> db.session.commit()
```

The user created previously can now log in. **Figure 8-2** shows the application home page with the user logged in.

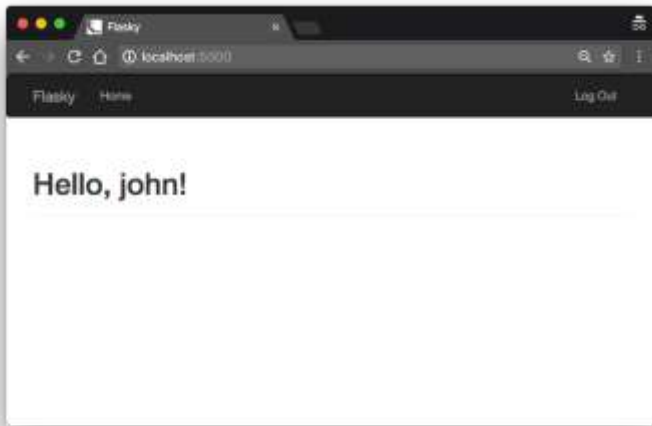


Figure 8-2. Home page after successful login

New User Registration

When new users want to become members of the application, they must register with it so that they are known and can log in. A link in the login page will send them to a registration page, where they can enter their email address, username, and password.

Adding a User Registration Form

The form that will be used in the registration page asks the user to enter an email address, username, and password.

Example 8-15. *app/auth/forms.py*: user registration form

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import DataRequired, Length, Email, Regexp, EqualTo
from wtforms import ValidationError
from ..models import User

class RegistrationForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Length(1, 64),
                                             Email()])

    username = StringField('Username', validators=[DataRequired(), Length(1, 64),
                                                  Regexp('^[A-Za-z][A-Za-z0-9_]*$', 0,
                                                  'Usernames must have only letters, numbers, dots or underscores')])

    password = PasswordField('Password', validators=[DataRequired(), EqualTo('password2', message='Passwords must match.')])
    password2 = PasswordField('Confirm password', validators=[DataRequired()])
    submit = SubmitField('Register')

    def validate_email(self, field):
        if User.query.filter_by(email=field.data).first():
            raise ValidationError('Email already registered.')

    def validate_username(self, field):
        if User.query.filter_by(username=field.data).first():
            raise ValidationError('Username already in use.')
```

The template that presents this form is called */templates/auth/register.html*. Like the login template, this one also renders the form with `wtf.quick_form()`. The registration page is shown in **Figure 8-3**.

Figure 8-3. New user registration form

The registration page needs to be linked from the login page so that users who don't have an account can easily find it. This change is shown in **Example 8-16**.

Example 8-16. app/templates/auth/login.html: link to the registration page

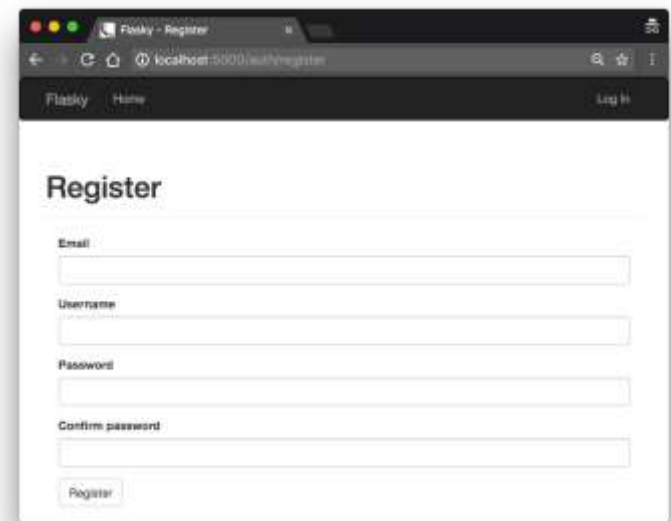
<p>

New user?

 Click

here to register

</p>

A screenshot of a web browser window titled 'Flasky - Register'. The address bar shows 'localhost:5000/auth/register'. The page has a dark header with 'Flasky' and 'Home' links, and a 'Log In' link on the right. The main content area is titled 'Register' and contains four input fields: 'Email', 'Username', 'Password', and 'Confirm password'. Below these fields is a 'Register' button.

Registering New Users

Handling user registrations does not present any big surprises. When the registration form is submitted and validated, a new user is added to the database using the information provided by the user. The view function that performs this task is shown in **Example 8-17**.

Example 8-17. app/auth/views.py: user registration route

```
@auth.route('/register', methods=['GET', 'POST'])
def register():
    form = RegistrationForm()
    if form.validate_on_submit():
        user = User(email=form.email.data,
                    username=form.username.data,
                    password=form.password.data)
        db.session.add(user)
        db.session.commit()
        flash('You can now login.')
        return redirect(url_for('auth.login'))
    return render_template('auth/register.html', form=form)
```

Generating Confirmation Tokens with itsdangerous

The following is a short shell session that shows how `itsdangerous` can generate a signed token that contains a user id inside:

```
(venv) $ flask shell
>>> from itsdangerous import TimedJSONWebSignatureSerializer as Serializer
>>> s = Serializer(app.config['SECRET_KEY'], expires_in=3600)
>>> token = s.dumps({ 'confirm': 23 })
>>> token
'eyJhbGciOiJIUzI1NiIsImV4cCI6MTM4MTcxODU1OCwiaWF0IjoxMzgxNzE0OTU4fQ.eyJ...'
```

```
>>> data = s.loads(token)
>>> data
{'confirm': 23}
```

Token generation and verification using this functionality can be added to the User model.

Example 8-18. app/models.py: user account confirmation

```
from itsdangerous import TimedJSONWebSignatureSerializer as Serializer
from flask import current_app
from . import db

class User(UserMixin, db.Model):
    # ...
    confirmed = db.Column(db.Boolean, default=False)
```

```

def generate_confirmation_token(self, expiration=3600):
    s = Serializer(current_app.config['SECRET_KEY'], expiration)
    return s.dumps({'confirm': self.id}).decode('utf-8')

def confirm(self, token):
    s = Serializer(current_app.config['SECRET_KEY'])
    try:
        data = s.loads(token.encode('utf-8'))
    except:
        return False
    if data.get('confirm') != self.id:
        return False
    self.confirmed = True
    db.session.add(self)
    return True

```

Sending Confirmation Emails

The current `/register` route redirects to `/index` after adding the new user to the database. Before redirecting, this route now needs to send the confirmation email.

Example 8-19. `app/auth/views.py`: registration route with confirmation email

```

from ..email import send_email @auth.route('/register',

methods=['GET', 'POST'])
def register():
    form = RegistrationForm()
    if form.validate_on_submit():
        # ...

```

```
db.session.add(user) db.session.commit()
token = user.generate_confirmation_token()
send_email(user.email, 'Confirm Your Account',
            'auth/email/confirm', user=user, token=token) flash('A
confirmation email has been sent to you by email.') return
redirect(url_for('main.index'))
return render_template('auth/register.html', form=form)
```

Example 8-20. app/templates/auth/email/confirm.txt: text body of confirmation email

Dear {{ user.username }}, Welcome to Flasky!

To confirm your account please click on the following link:

{{ url_for('auth.confirm', token=token, _external=True) }}

Sincerely,

The Flasky Team

Note: replies to this email address are not monitored.

Example 8-21. app/auth/views.py: confirming a user account

```
from flask_login import current_user

@auth.route('/confirm/<token>')
@login_required
def confirm(token):
    if current_user.confirmed:
        return redirect(url_for('main.index'))
    if current_user.confirm(token):
        db.session.commit()
        flash('You have confirmed your account. Thanks!')
    else:
        flash('The confirmation link is invalid or has expired.')
    return redirect(url_for('main.index'))
```

This route is protected with the `login_required` decorator from Flask-Login, so that when the users click on the link from the confirmation email they are asked to log in before they reach this view function.

Example 8-22. app/auth/views.py: filtering unconfirmed accounts with the before_app_request handler

```
@auth.before_app_request
def before_request():
    if current_user.is_authenticated \
        and not current_user.confirmed \
        and request.blueprint != 'auth' \
        and request.endpoint != 'static':
        return redirect(url_for('auth.unconfirmed'))

@auth.route('/unconfirmed')
def unconfirmed():
    if current_user.is_anonymous or current_user.confirmed:
        return redirect(url_for('main.index'))
    return render_template('auth/unconfirmed.html')
```

The page that is presented to unconfirmed users (shown in **Figure 8-4**) just renders a template that gives users instructions for how to confirm their accounts and offers a link to request a new confirmation email, in case the original email was lost. The route that resends the confirmation email is shown in **Example 8-23**.

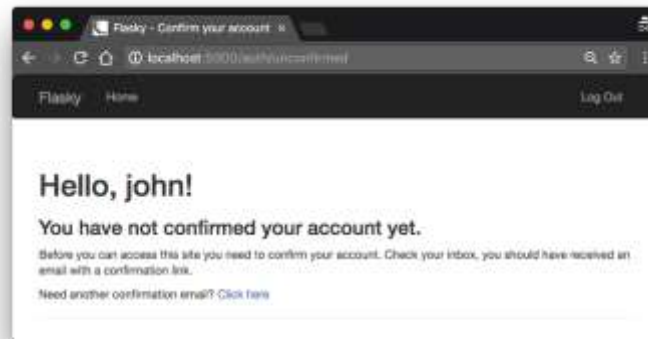


Figure 8-4. Unconfirmed account page

Example 8-23. app/auth/views.py: resending the account confirmation email

```
@auth.route('/confirm') @login_required
def resend_confirmation():
    token = current_user.generate_confirmation_token()
    send_email(current_user.email, 'Confirm Your Account',
               'auth/email/confirm', user=current_user, token=token)
    flash('A new confirmation email has been sent to you by email.')
    return redirect(url_for('main.index'))
```

Database Representation of Roles

A simple roles table was created in **Chapter 5** as a vehicle to demonstrate one-to-many relationships. **Example 9-1** shows an improved Role model with some additions.

Example 9-1. app/models.py: role database model

```
class Role(db.Model):
    __tablename__ = 'roles'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)
    default = db.Column(db.Boolean, default=False, index=True)
    permissions = db.Column(db.Integer)
    users = db.relationship('User', backref='role', lazy='dynamic')

    def __init__(self, **kwargs):
        super(Role, self).__init__(**kwargs)
        if self.permissions is None:
            self.permissions = 0
```

The default field is one of the additions to this model. This field should be set to True for only one role and False for all the others.

Example 9-2. app/models.py: permission constants

```
class Permission: FOLLOW = 1
    COMMENT = 2
    WRITE = 4
    MODERATE = 8
    ADMIN = 16
```

Example 9-3. app/models.py: permission management in the Role model

```
class Role(db.Model):
    # ...

    def add_permission(self, perm):
        if not self.has_permission(perm): self.permissions +=
            perm

    def remove_permission(self, perm):
        if self.has_permission(perm): self.permissions -=
            perm

    def reset_permissions(self): self.permissions = 0

    def has_permission(self, perm):
        return self.permissions & perm == perm
```

The `add_permission()`, `remove_permission()`, and `reset_permission()` methods all use basic arithmetic operations to update the permission list. The `has_permission()` method is the most complex of the set, as it relies on the [bitwise and operator &](#) to check if a combined permission value includes the given basic permission. You can play with these methods in a Python shell:

```
(venv) $ flask shell
>>> r = Role(name='User')
>>> r.add_permission(Permission.FOLLOW)
>>> r.add_permission(Permission.WRITE)
>>> r.has_permission(Permission.FOLLOW)
True
>>> r.has_permission(Permission.ADMIN)
False
>>> r.reset_permissions()
>>> r.has_permission(Permission.FOLLOW)
False
```

Example 9-4. app/models.py: creating roles in the database

```
class Role(db.Model):
    # ...
    @staticmethod
    def insert_roles():
        roles = {
            'User': [Permission.FOLLOW, Permission.COMMENT, Permission.WRITE],
            'Moderator': [Permission.FOLLOW, Permission.COMMENT,
                          Permission.WRITE, Permission.MODERATE],
            'Administrator':
                [Permission.FOLLOW, Permission.COMMENT,
                Permission.WRITE, Permission.MODERATE, Permission.ADMIN],
        }
        default_role = 'User'
        for r in roles:
            role = Role.query.filter_by(name=r).first()
            if role is None:
                role = Role(name=r)
                role.reset_permissions()
            for perm in roles[r]:
                role.add_permission(perm)
            role.default = (role.name == default_role)
            db.session.add(role)
            db.session.commit()
```

Role Assignment

Example 9-5. app/models.py: defining a default role for users

```
class User(UserMixin, db.Model):  
    # ...  
    def __init__(self, **kwargs): super(User,  
        self).__init__(**kwargs) if self.role is None:  
        if self.email == current_app.config['FLASKY_ADMIN']:  
            self.role = Role.query.filter_by(name='Administrator').first()  
        if self.role is None:  
            self.role = Role.query.filter_by(default=True).first()  
    # ...
```

The `User` constructor first invokes the constructors of the base classes, and if after that the object does not have a role defined, it sets the administrator or default role depending on the email address.

Role Verification

Example 9-6. app/models.py: evaluating whether a user has a given permission

```
from flask_login import UserMixin, AnonymousUserMixin

class User(UserMixin, db.Model):
    # ...

    def can(self, perm):
        return self.role is not None and self.role.has_permission(perm)

    def is_administrator(self):
        return self.can(Permission.ADMIN)

class AnonymousUser(AnonymousUserMixin):
    def can(self, permissions):
        return False

    def is_administrator(self):
        return False

login_manager.anonymous_user = AnonymousUser
```

Example 9-7. app/decorators.py: custom decorators that check user permissions

```
from functools import wraps
from flask import abort
from flask_login import current_user
from .models import Permission

def permission_required(permission):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            if not current_user.can(permission): abort(403)
            return f(*args, **kwargs)
        return decorated_function
    return decorator

def admin_required(f):
    return permission_required(Permission.ADMIN)(f)
```

The following are two examples that demonstrate the usage of these decorators:

```
from .decorators import admin_required, permission_required
```

```
@main.route('/admin') @login_required
```

```
@admin_required
```

```
def for_admins_only():
```

```
    return "For administrators!"
```

```
@main.route('/moderate') @login_required
```

```
@permission_required(Permission.MODERATE)
```

```
def for_moderators_only():
```

```
    return "For comment moderators!"
```

Example 9-8. app/main/_init_.py: adding the Permission class to the template context

```
@main.app_context_processor
def inject_permissions():
    return dict(Permission=Permission)
```

The new roles and permissions can be exercised in unit tests. **Example 9-9** shows two of the tests. The source code on GitHub includes one for each role.

Example 9-9. tests/test_user_model.py: unit tests for roles and permissions

```
class UserModelTestCase(unittest.TestCase):
    # ...
    def test_user_role(self):
        u = User(email='john@example.com', password='cat')
        self.assertTrue(u.can(Permission.FOLLOW))
        self.assertTrue(u.can(Permission.COMMENT))
        self.assertTrue(u.can(Permission.WRITE))
        self.assertFalse(u.can(Permission.MODERATE))
        self.assertFalse(u.can(Permission.ADMIN))

    def test_anonymous_user(self): u = AnonymousUser()
        self.assertFalse(u.can(Permission.FOLLOW))
        self.assertFalse(u.can(Permission.COMMENT))
        self.assertFalse(u.can(Permission.WRITE))
        self.assertFalse(u.can(Permission.MODERATE))
        self.assertFalse(u.can(Permission.ADMIN)).
```

Profile Information

Example 10-1. app/models.py: user information fields

```
class User(UserMixin, db.Model):  
    # ...  
    name = db.Column(db.String(64))    location =  
    db.Column(db.String(64))    about_me = db.Column(db.Text())  
    member_since = db.Column(db.DateTime(), default=datetime.utcnow)  
    last_seen = db.Column(db.DateTime(), default=datetime.utcnow)
```

The new fields store the user's real name, location, self-written bio, date of registration, and date of last visit. The `about_me` field is assigned the type `db.Text()`. The difference between `db.String` and `db.Text` is that `db.Text` is a variable-length field and as such does not need a maximum length.

Example 10-2. app/models.py: refreshing a user's last visit time

```
class User(UserMixin, db.Model):  
    # ...  
  
    def ping(self):  
        self.last_seen = datetime.utcnow()    db.session.add(self)  
        db.session.commit()
```

Example 10-3. app/auth/views.py: pinging the logged-in user

```
@auth.before_app_request
def before_request():
    if current_user.is_authenticated:
        current_user.ping()
        if not current_user.confirmed \
            and request.endpoint \
            and request.blueprint != 'auth' \
            and request.endpoint != 'static':
            return redirect(url_for('auth.unconfirmed'))
```

User Profile Page

Example 10-4. app/main/views.py: profile page route

```
@main.route('/user/<username>')
def user(username):
    user = User.query.filter_by(username=username).first_or_404()
    return render_template('user.html', user=user)
```

Example 10-5. `app/templates/user.html`: user profile template

```
{% extends "base.html" %}
{% block title %}Flasky - {{ user.username }}{% endblock %}

{% block page_content %}
<div class="page-header">
  <h1>{{ user.username }}</h1>
  {% if user.name or user.location %}
  <p>
    {% if user.name %} {{ user.name }} {% endif %}
    {% if user.location %}
    From <a href="http://maps.google.com/?q={{ user.location }}">
      {{ user.location }}
    </a>
    {% endif %}
  </p>
  {% endif %}
  {% if current_user.is_administrator() %}
  <p><a href="mailto:{{ user.email }}">{{ user.email }}</a></p>
  {% endif %}
  {% if user.about_me %}<p>{{ user.about_me }}</p>{% endif %}
  <p>
    Member since {{ moment(user.member_since).format('L') }}. Last seen {{
    moment(user.last_seen).fromNow() }}.
  </p>
</div>
{% endblock %}
```

Example 10-6. app/templates/base.html: add link to profile page in the navigation bar

```
{% if current_user.is_authenticated %}  
<li>  
  <a href="{{ url_for('main.user', username=current_user.username) }}">  
    Profile  
  </a>  
</li>  
{% endif %}
```

Using a conditional for the profile page link is necessary because the navigation bar is also rendered for unauthenticated users, in which case the profile link is skipped. **Figure 10-1** shows how the profile page looks in the browser. The new profile link in the navigation bar is also shown.



Figure 10-1. User profile page

User-Level Profile Editor

Example 10-7. *app/main/forms.py: edit profile form*

```
class EditProfileForm(FlaskForm):
    name = StringField('Real name', validators=[Length(0, 64)])
    location = StringField('Location', validators=[Length(0, 64)])
    about_me = TextAreaField('About me')
    submit = SubmitField('Submit')
```

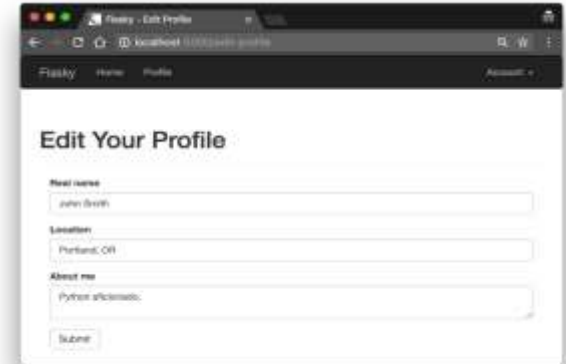


Figure 10-2. Profile editor

Example 10-8. *app/main/views.py: edit profile route*

```
@main.route('/edit-profile', methods=['GET', 'POST']) @login_required
def edit_profile():
    form = EditProfileForm()
    if form.validate_on_submit():
        current_user.name = form.name.data
        current_user.location = form.location.data
        current_user.about_me = form.about_me.data
        db.session.add(current_user._get_current_object())
        db.session.commit()
        flash('Your profile has been updated.')
        return redirect(url_for('.user', username=current_user.username))
    form.name.data = current_user.name
    form.location.data = current_user.location
    form.about_me.data = current_user.about_me
    return render_template('edit_profile.html', form=form)
```

Administrator-Level Profile Editor

Example 10-10. app/main/forms.py: profile editing form for administrators

```
class EditProfileAdminForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Length(1, 64),
                                             Email()])
    username = StringField('Username', validators=[
        DataRequired(), Length(1, 64),
        Regexp('^[A-Za-z][A-Za-z0-9_]*$', 0,
            'Usernames must have only letters, numbers, dots or 'underscores')])
    confirmed = BooleanField('Confirmed')
    role = SelectField('Role', coerce=int)
    name = StringField('Real name', validators=[Length(0, 64)])
    location = StringField('Location', validators=[Length(0, 64)])
    about_me = TextAreaField('About me')
    submit = SubmitField('Submit')

    def __init__(self, user, *args, **kwargs):
        super(EditProfileAdminForm, self).__init__(*args, **kwargs)
        self.role.choices = [(role.id, role.name)
                              for role in Role.query.order_by(Role.name).all()]
        self.user = user

    def validate_email(self, field):
        if field.data != self.user.email and \
            User.query.filter_by(email=field.data).first():
            raise ValidationError('Email already registered.')

    def validate_username(self, field):
        if field.data != self.user.username and \
            User.query.filter_by(username=field.data).first():
            raise ValidationError('Username already in use.')
```

Example 10-11. app/main/views.py: edit profile route for administrators

```
from ..decorators import admin_required

@main.route('/edit-profile/<int:id>', methods=['GET', 'POST']) @login_required
@admin_required
def edit_profile_admin(id):
    user = User.query.get_or_404(id)
    form = EditProfileAdminForm(user=user)
    if form.validate_on_submit():
        user.email = form.email.data
        user.username = form.username.data
        user.confirmed = form.confirmed.data
        user.role = Role.query.get(form.role.data)
        user.name = form.name.data
        user.location = form.location.data
        user.about_me = form.about_me.data
        db.session.add(user)
        db.session.commit()
        flash('The profile has been updated.')
        return redirect(url_for('.user', username=user.username))
    form.email.data = user.email
    form.username.data = user.username
    form.confirmed.data = user.confirmed
    form.role.data = user.role_id
    form.name.data = user.name
    form.location.data = user.location
    form.about_me.data = user.about_me
    return render_template('edit_profile.html', form=form, user=user)
```

User Avatars

The service exposes the user's avatar through a specially crafted URL that includes the MD5 hash of the user's email address, which can be calculated as follows:

```
(venv) $ python
>>> import hashlib
>>> hashlib.md5('john@example.com'.encode('utf-8')).hexdigest()
'd4c74594d841139328695756648b6bd6'
```

Example 10-13. app/models.py: gravatar URL generation

```
import hashlib
from flask import request

class User(UserMixin, db.Model):
    # ...
    def gravatar(self, size=100, default='identicon', rating='g'):
        url = 'https://secure.gravatar.com/avatar'
        hash = hashlib.md5(self.email.lower().encode('utf-8')).hexdigest()
        return '{url}/{hash}?s={size}&d={default}&r={rating}'.format( url=url,
            hash=hash, size=size, default=default, rating=rating)
```

The avatar URL is generated from the base URL, the MD5 hash of the user's email address, and the arguments, all of which have default values. With this implementation it is easy to generate avatar URLs in the Python shell:

```
(venv) $ flask shell
>>> u = User(email='john@example.com')
>>> u.gravatar()
'https://secure.gravatar.com/avatar/d4c74594d841139328695756648b6bd6?s=100&d=identicon&r=g'
>>> u.gravatar(size=256)
'https://secure.gravatar.com/avatar/d4c74594d841139328695756648b6bd6?s=256&d=identicon&r=g'
```

Example 10-14. app/templates/user.html: adding an avatar to the profile page

```
...

<div class="profile-header">
...
</div>
...
```

The `profile-thumbnail` CSS class helps with the positioning of the image on the page. The `<div>` element that follows the image encapsulates the profile information and uses the `profile-header` CSS class to improve the formatting. You can see the definition of the CSS class in the GitHub repository for the application.

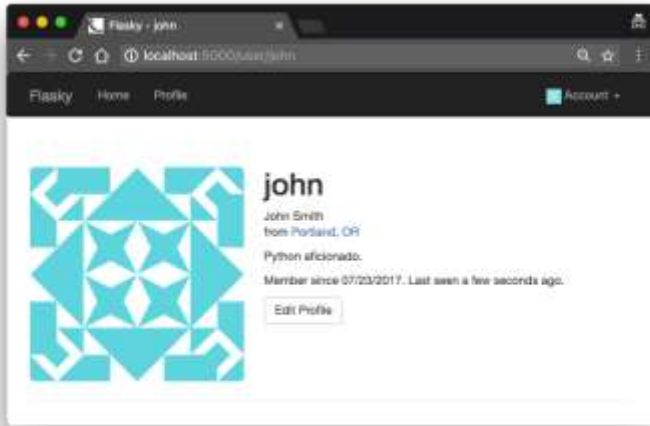


Figure 10-3. User profile page with avatar

If you have cloned the application's Git repository on GitHub, you can run `git checkout 10c` to check out this version of the application.

The generation of avatars requires an MD5 hash to be generated, which is a CPU- intensive operation. If a large number of avatars need to be generated for a page, then the computational work can add up and become significant. Since the MD5 hash for a user will remain constant for as long as the email address stays the same, it can be *cached* in the User model. **Example 10-15** shows the changes to the User model to store the MD5 hashes in the database.

Example 10-15. app/models.py: gravatar URL generation with caching of MD5 hashes

```
class User(UserMixin, db.Model):
    # ...
    avatar_hash = db.Column(db.String(32))

    def __init__(self, **kwargs):
        # ...
        if self.email is not None and self.avatar_hash is None:
```

```
self.avatar_hash = self.gravatar_hash()

def change_email(self, token):
    # ...
    self.email = new_email
    self.avatar_hash = self.gravatar_hash()
    db.session.add(self)
    return True

def gravatar_hash(self):
    return hashlib.md5(self.email.lower().encode('utf-8')).hexdigest()

def gravatar(self, size=100, default='identicon', rating='g'):
    if request.is_secure:
        url = 'https://secure.gravatar.com/avatar'
    else:
        url = 'http://www.gravatar.com/avatar'
    hash = self.avatar_hash or self.gravatar_hash()
    return '{url}/{hash}?s={size}&d={default}&r={rating}'.format(
        url=url, hash=hash, size=size, default=default, rating=rating)
```

Blog Post Submission and Display

Example 11-1. app/models.py: Post model

```
class Post(db.Model):
    __tablename__ = 'posts'
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.Text)
    timestamp = db.Column(db.DateTime, index=True, default=datetime.utcnow)
    author_id = db.Column(db.Integer, db.ForeignKey('users.id'))

class User(UserMixin, db.Model):
    # ...
    posts = db.relationship('Post', backref='author', lazy='dynamic')
```

A blog post is represented by a body, a timestamp, and a one-to-many relationship from the User model. The body field is defined with type `db.Text` so that there is no limitation on the length.

The form that will be shown in the main page of the application lets users write a blog post. This form is very simple; it contains just a text area where the blog post can be typed and a submit button. The form definition is shown in [Example 11-2](#).

Example 11-2. app/main/forms.py: blog post form

```
class PostForm(FlaskForm):  
    body = TextAreaField("What's on your mind?", validators=[DataRequired()])  
    submit = SubmitField('Submit')
```

The `index()` view function handles the form and passes the list of old blog posts to the template, as shown in **Example 11-3**.

Example 11-3. app/main/views.py: home page route with a blog post

```
@main.route('/', methods=['GET', 'POST'])  
def index():  
    form = PostForm()  
    if current_user.can(Permission.WRITE_ARTICLES) and  
        form.validate_on_submit():  
        post = Post(body=form.body.data,  
                    author=current_user._get_current_object())  
        db.session.add(post)  
        db.session.commit()  
        return redirect(url_for('.index'))  
    posts = Post.query.order_by(Post.timestamp.desc()).all()  
    return render_template('index.html', form=form, posts=posts)
```

Example 11-4. app/templates/index.html: home page template with blog posts

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}
...
<div>
    {% if current_user.can(Permission.WRITE_ARTICLES) %}
    {{ wtf.quick_form(form) }}
    {% endif %}
</div>
<ul class="posts">
    {% for post in posts %}
    <li class="post">
        <div class="profile-thumbnail">
            <a href="{{ url_for('.user', username=post.author.username) }}">
                
            </a>
        </div>
        <div class="post-date">{{ moment(post.timestamp).fromNow() }}</div>
        <div class="post-author">
            <a href="{{ url_for('.user', username=post.author.username) }}">
                {{ post.author.username }}
            </a>
        </div>
        <div class="post-body">{{ post.body }}</div>
    </li>
    {% endfor %}
</ul>
...
```

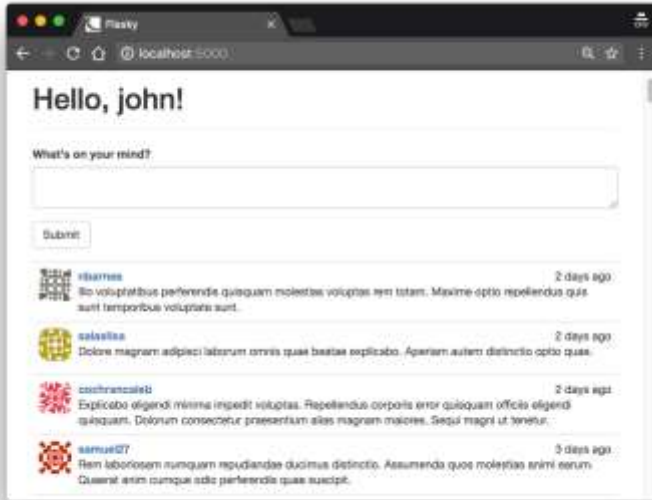


Figure 11-1. Home page with blog submission form and blog post list

Blog Posts on Profile Pages

Example 11-5. app/main/views.py: profile page route with blog posts

```
@main.route('/user/<username>')
def user(username):
    user = User.query.filter_by(username=username).first()
    if user is None: abort(404)
    posts = user.posts.order_by(Post.timestamp.desc()).all()
    return render_template('user.html', user=user, posts=posts)
```

The list of blog posts for a user is obtained from the `User.posts` relationship. This works like a query object, so filters such as `order_by()` can be used on it like in a regular query object.

Creating Fake Blog Post Data

To be able to work with multiple pages of blog posts, it is necessary to have a test database with a large volume of data. Manually adding new database entries is time consuming and tedious; an automated solution is more appropriate. There are several Python packages that can be used to generate fake information. A fairly complete one is *Faker*, which is installed with *pip*:

```
(venv) $ pip install faker
```

Example 11-7. requirements/dev.txt: development requirements file

```
-r common.txt faker==0.7.18
```

Example 11-8. app/fake.py: generating fake users and blog posts

```
from random import randint
from sqlalchemy.exc import IntegrityError
from faker import Faker
from . import db
from .models import User, Post

def users(count=100):
    fake = Faker()
    i = 0
    while i < count:
        u = User(email=fake.email(),
                 username=fake.user_name(), password='password', confirmed=True,
                 name=fake.name(), location=fake.city(), about_me=fake.text(),
                 member_since=fake.past_date())
        db.session.add(u)
        try:
            db.session.commit()
            i += 1
```

```

    except IntegrityError: db.session.rollback()
def posts(count=100): fake = Faker()
    user_count = User.query.count()
    for i in range(count):
        u = User.query.offset(randint(0, user_count - 1)).first() p = Post(body=fake.text(),
            timestamp=fake.past_date(), author=u)
        db.session.add(p) db.session.commit()

```

The new functions make it easy to create a large number of fake users and posts from the Python shell:

```

(venv) $ flask shell
>>> from app import fake
>>> fake.users(100)
>>> fake.posts(100)

```

If you run the application now, you will see a long list of random blog posts on the home page, by many different users.

Rendering in Pages

Example 11-9 shows the changes to the home page route to support pagination.

Example 11-9. app/main/views.py: paginating the blog post list

```
@main.route('/', methods=['GET', 'POST'])
def index():
    # ...
    page = request.args.get('page', 1, type=int)
    pagination = Post.query.order_by(Post.timestamp.desc()).paginate( page,
        per_page=current_app.config['FLASKY_POSTS_PER_PAGE'], error_out=False)
    posts = pagination.items
    return render_template('index.html', form=form, posts=posts,
        pagination=pagination)
```

Adding a Pagination Widget

Example 11-10. app/templates/_macros.html: pagination template macro

```
{% macro pagination_widget(pagination, endpoint) %}
<ul class="pagination">
    <li{% if not pagination.has_prev %} class="disabled"{% endif %}>
        <a href="{% if pagination.has_prev %}{{ url_for(endpoint,
            page = pagination.page - 1, **kwargs) }}{% else %}#{% endif %}"> &laquo;
    </a>
</li>
{% for p in pagination.iter_pages() %}
```

```

{% if p %}
  {% if p == pagination.page %}
    <li class="active">
      <a href="{{ url_for(endpoint, page = p, **kwargs) }}">{{ p }}</a>
    </li>
  {% else %}
    <li>
      <a href="{{ url_for(endpoint, page = p, **kwargs) }}">{{ p }}</a>
    </li>
  {% endif %}
{% else %}
  <li class="disabled"><a href="#">&hellip;</a></li>
{% endif %}
{% endfor %}
<li{% if not pagination.has_next %} class="disabled"{% endif %}>
  <a href="{% if pagination.has_next %}{{ url_for(endpoint,
    page = pagination.page + 1, **kwargs) }}{% else %}#{% endif %}"> &raquo;
  </a>
</li>
</ul>
{% endmacro %}

```

Example 11-11. app/templates/index.html: pagination footer for blog post lists

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}
{% import "_macros.html" as macros %}
...
{% include '_posts.html' %}
<div class="pagination">
    {{ macros.pagination_widget(pagination, '.index') }}
</div>
{% endif %}
```

Figure 11-2 shows how the pagination links appear in the page.

Figure 11-2. Blog post pagination



Rich-Text Posts with Markdown and Flask-PageDown

The Python packages can all be installed with *pip*:

```
(venv) $ pip install flask-pagedown markdown bleach
```

Using Flask-PageDown

Example 11-12. app/_init_.py: Flask-PageDown initialization

```
from flask_pagedown import PageDown
# ...
pagedown = PageDown()
# ...
def create_app(config_name):
    # ...
    pagedown.init_app(app)
    # ...
```

To convert the text area control in the home page to a Markdown rich-text editor, the `body` field of the `PostForm` must be changed to a `PageDownField` as shown in [Example 11-13](#).

Example 11-13. app/main/forms.py: Markdown-enabled post form

```
from flask_pagedown.fields import PageDownField

class PostForm(FlaskForm):
    body = PageDownField("What's on your mind?", validators=[Required()]) submit =
    SubmitField('Submit')
```

Example 11-14. app/templates/index.html: Flask-PageDown template declaration

```
{% block scripts %}
{{ super() }}
{{ pagedown.include_pagedown() }}
{% endblock %}
```

If you have cloned the application's Git repository on GitHub, you can run `git checkout 11e` to check out this version of the application. To ensure that you have all the dependencies installed also run `pip install -r requirements/dev.txt`.

With these changes, Markdown-formatted text typed in the text area field will be immediately rendered as HTML in the preview area below. **Figure 11-3** shows the blog submission form with rich text.

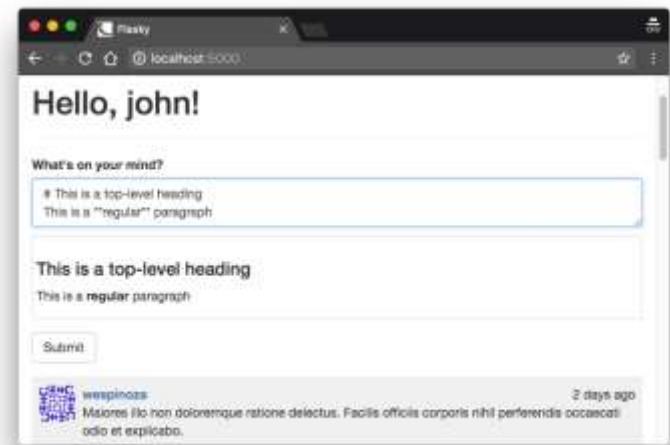


Figure 11-3. Rich-text blog post form

Handling Rich Text on the Server

Example 11-15. app/models.py: Markdown text handling in the Post model

```
from markdown import markdown
import bleach

class Post(db.Model):
    # ...
    body_html = db.Column(db.Text)
    # ...
    @staticmethod
    def on_changed_body(target, value, oldvalue, initiator):
        allowed_tags = ['a', 'abbr', 'acronym', 'b', 'blockquote', 'code',
                        'em', 'i', 'li', 'ol', 'pre', 'strong', 'ul', 'h1', 'h2', 'h3', 'p']
        target.body_html = bleach.linkify(bleach.clean(markdown(value,
                                                                output_format='html'),
                                                                tags=allowed_tags, strip=True))

db.event.listen(Post.body, 'set', Post.on_changed_body)
```

The `on_changed_body()` function is registered as a listener of SQLAlchemy's "set" event for `body`, which means that it will be automatically invoked whenever the `body` field is set to a new value. The handler function renders the HTML version of the `body` and stores it in `body_html`, effectively making the conversion of the Markdown text to HTML fully automatic.

Example 11-16. app/templates/_posts.html: use the HTML version of the post bodies in the template

```
...
<div class="post-body">
    {% if post.body_html %}
        {{ post.body_html | safe }}
    {% else %}
        {{ post.body }}
    {% endif %}
</div>
...
```

Permanent Links to Blog Posts

Users may want to share links to specific blog posts with friends on social networks. For this purpose, each post will be assigned a page with a unique URL that references it. The route and view function that support permanent links are shown in [Example 11-17](#).

Example 11-17. app/main/views.py: enabling permanent links to posts

```
@main.route('/post/<int:id>')
def post(id):
    post = Post.query.get_or_404(id)
    return render_template('post.html', posts=[post])
```

Example 11-18. app/templates/_posts.html: adding permanent links to posts

```
<ul class="posts">
  {% for post in posts %}
    <li class="post">
      ...
      <div class="post-content">
        ...
        <div class="post-footer">
          <a href="{{ url_for('.post', id=post.id) }}">
            <span class="label label-default">Permalink</span>
          </a>
        </div>
      </div>
    </li>
  {% endfor %}
</ul>
```

The new *post.html* template that renders the permanent link page is shown in **Example 11-19**. It includes the example template.

Example 11-19. app/templates/post.html: permanent link template

```
{% extends "base.html" %}

{% block title %}Flasky - Post{% endblock %}

{% block page_content %}
{% include '_posts.html' %}
{% endblock %}
```

Blog Post Editor

The last feature related to blog posts is a post editor that allows users to edit their own posts. The blog post editor lives in a standalone page and is also based on Flask-PageDown, so a text area where the Markdown text of the blog post can be edited is followed by a rendered preview. The *edit_post.html* template is shown in **Example 11-20**.

Example 11-20. app/templates/edit_post.html: edit blog post template

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}

{% block title %}Flasky - Edit Post{% endblock %}

{% block page_content %}
<div class="page-header">
  <h1>Edit Post</h1>
</div>
<div>
  {{ wtf.quick_form(form) }}
</div>
{% endblock %}

{% block scripts %}
{{ super() }}
{{ pagedown.include_pagedown() }}
{% endblock %}
```

Example 11-21. app/main/views.py: edit blog post route

```
@main.route('/edit/<int:id>', methods=['GET', 'POST'])
@login_required
def edit(id):
    post = Post.query.get_or_404(id)
    if current_user != post.author and \
        not current_user.can(Permission.ADMIN): abort(403)
    form = PostForm()
    if form.validate_on_submit(): post.body =
        form.body.data db.session.add(post)
        db.session.commit()
        flash('The post has been updated.')
        return redirect(url_for('.post', id=post.id))
    form.body.data = post.body
    return render_template('edit_post.html', form=form)
```

To complete the feature, a link to the blog post editor can be added below each blog post, next to the permanent link, as shown in **Example 11-22**.

Example 11-22. `app/templates/_posts.html`: adding the edit blog post link

```
<ul class="posts">
  {% for post in posts %}
    <li class="post">
      ...
      <div class="post-content">
        ...
        <div class="post-footer">
          ...
          {% if current_user == post.author %}
            <a href="{{ url_for('.edit', id=post.id) }}">
              <span class="label label-primary">Edit</span>
            </a>
          {% elif current_user.is_administrator() %}
            <a href="{{ url_for('.edit', id=post.id) }}">
              <span class="label label-danger">Edit [Admin]</span>
            </a>
          {% endif %}
        </div>
      </div>
    </li>
  {% endfor %}
</ul>
```

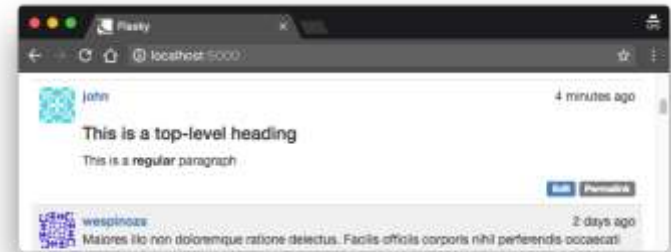


Figure 11-4. Edit and Permalink links in a blog post

Many-to-Many Relationships

The solution is to add a third table to the database, called an *association table*. Now the many-to-many relationship can be decomposed into two one-to-many relationships from each of the two original tables to the association table. **Figure 12-1** shows how the many-to-many relationship between students and classes is represented.

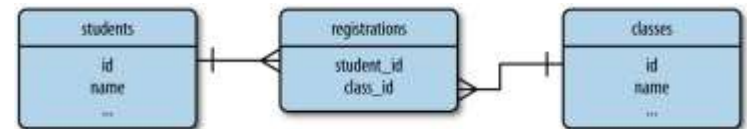
Figure 12-1. Many-to-many relationship example

Traversing two relationships to obtain query results sounds difficult, but for a simple relationship like the one in the previous example, SQLAlchemy does most of the work. Following is the code that represents the many-to-many relationship in **Figure 12-1**:

```
registrations = db.Table('registrations',
    db.Column('student_id', db.Integer, db.ForeignKey('students.id')),
    db.Column('class_id', db.Integer, db.ForeignKey('classes.id'))
)
```

```
class Student(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String)
    classes = db.relationship('Class',
        secondary=registrations, backref=db.backref('students',
            lazy='dynamic'), lazy='dynamic')
```

```
class Class(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String)
```



The `classes` relationship uses list semantics, which makes working with a many-to-many relationship configured in this way extremely easy. Given a student `s` and a class `c`, the code that registers the student for the class is:

```
>>> s.classes.append(c)
>>> db.session.add(s)
```

The queries that list the classes student `s` is registered for and the list of students registered for class `c` are also very simple:

```
>>> s.classes.all()
>>> c.students.all()
```

The `students` relationship available in the `Class` model is the one defined in the `db.backref()` argument. Note that in this relationship the `backref` argument was expanded to also have a `lazy='dynamic'` attribute, so both sides return a query that can accept additional filters.

If student `s` later decides to drop class `c`, you can update the database as follows:

```
>>> s.classes.remove(c)
```

Example 12-3. app/models.py: followers helper methods

```
class User(db.Model):
    # ...
    def follow(self, user):
        if not self.is_following(user):
            f = Follow(follower=self, followed=user) db.session.add(f)

    def unfollow(self, user):
        f = self.followed.filter_by(followed_id=user.id).first()
        if f:
            db.session.delete(f)

    def is_following(self, user):
        if user.id is None:
            return False
        return self.followed.filter_by( followed_id=user.id).first() is
            not None

    def is_followed_by(self, user):
        if user.id is None:
            return False
        return self.followers.filter_by( follower_id=user.id).first() is
            not None
```

Followers on the Profile Page

The profile page of a user needs to present a “Follow” button if the user viewing it is not a follower, or an “Unfollow” button if the user is a follower. It is also a nice addition to show the follower and followed counts, display the lists of followers and followed users, and show a “Follows you” sign when appropriate. The changes to the user profile template are shown in [Example 12-4](#). [Figure 12-3](#) shows how the additions look on the profile page.

Example 12-4. `app/templates/user.html`: follower enhancements to the user profile header

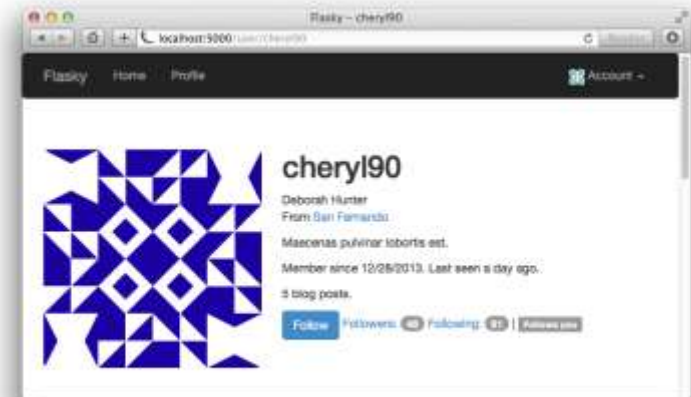
```
{% if current_user.can(Permission.FOLLOW) and user != current_user %}
    {% if not current_user.is_following(user) %}
        <a href="{{ url_for('.follow', username=user.username) }}" class="btn btn-
            primary">Follow</a>
    {% else %}
        <a href="{{ url_for('.unfollow', username=user.username) }}" class="btn btn-
            default">Unfollow</a>
    {% endif %}
{% endif %}
<a href="{{ url_for('.followers', username=user.username) }}"> Followers: <span
    class="badge">{{ user.followers.count() }}</span>
</a>
<a href="{{ url_for('.followed_by', username=user.username) }}"> Following: <span
    class="badge">{{ user.followed.count() }}</span>
</a>
{% if current_user.is_authenticated and user != current_user and
    user.is_following(current_user) %}
    | <span class="label label-default">Follows you</span>
{% endif %}
```

Figure 12-3. Followers on the profile page

There are four new endpoints defined in these template changes. The `/follow/<user-name>` route is invoked when a user clicks the “Follow” button on another user’s profile page. The implementation is shown in **Example 12-5**.

Example 12-5. `app/main/views.py`: follow route and view function

```
@main.route('/follow/<username>') @login_required
@permission_required(Permission.FOLLOW) def follow(username):
    user = User.query.filter_by(username=username).first()
    if user is None: flash('Invalid user.')
    return redirect(url_for('.index'))
    if current_user.is_following(user):
        flash('You are already following this user.')
        return redirect(url_for('.user', username=username))
    current_user.follow(user)
    db.session.commit()
    flash('You are now following %s.' % username)
    return redirect(url_for('.user', username=username))
```



This view function loads the requested user, verifies that it is valid and that it isn’t already followed by the logged-in user, and then calls the `follow()` helper function in

the User model to establish the link. The `/unfollow/<username>` route is implemented in a similar way.

The `/followers/<username>` route is invoked when a user clicks another user's follower count on the profile page. The implementation is shown in **Example 12-6**.

Example 12-6. `app/main/views.py`: followers route and view function

```
@main.route('/followers/<username>')
def followers(username):
    user = User.query.filter_by(username=username).first()
    if user is None: flash('Invalid user.')
    return redirect(url_for('.index'))
page = request.args.get('page', 1, type=int) pagination =
user.followers.paginate(
    page, per_page=current_app.config['FLASKY_FOLLOWERS_PER_PAGE'],
    error_out=False)
follows = [{'user': item.follower, 'timestamp': item.timestamp}
    for item in pagination.items]
return render_template('followers.html', user=user, title="Followers of",
    endpoint='.followers', pagination=pagination,
    follows=follows)
```

- `join(Post, Follow.followed_id == Post.author_id)` joins the results of `filter_by()` with the `Post` objects.

The query can be simplified by swapping the order of the filter and the join:

```
return Post.query.join(Follow, Follow.followed_id == Post.author_id)\
    .filter(Follow.follower_id == self.id)
```

Example 12-7. `app/models.py`: obtaining followed posts

```
class User(db.Model):
    # ...
    @property
    def followed_posts(self):
        return Post.query.join(Follow, Follow.followed_id == Post.author_id)\
            .filter(Follow.follower_id == self.id)
```

Note that the `followed_posts()` method is defined as a property so that it does not need the `()`. That way, all relationships have a consistent syntax.

Joins are extremely hard to wrap your head around; you may need to experiment with the example code in a shell before it all sinks in.

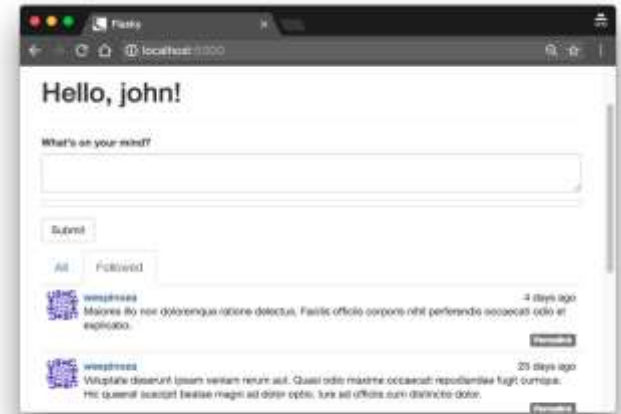
Showing Followed Posts on the Home Page

The home page can now give users the choice to view all blog posts or just those from followed users.

Example 12-8 shows how this choice is implemented.

Example 12-8. app/main/views.py: showing all or followed posts

```
@main.route('/', methods = ['GET', 'POST'])
def index():
    # ...
    show_followed = False
    if current_user.is_authenticated:
        show_followed = bool(request.cookies.get('show_followed', ''))
    if show_followed:
        query = current_user.followed_posts
    else:
        query = Post.query
    pagination = query.order_by(Post.timestamp.desc()).paginate( page,
        per_page=current_app.config['FLASKY_POSTS_PER_PAGE'], error_out=False)
    posts = pagination.items
    return render_template('index.html', form=form, posts=posts,
        show_followed=show_followed, pagination=pagination)
```



*Figure 12-4.
Followed posts on
the home page*

Example 12-9. app/main/views.py: selection of all or followed posts

```
@main.route('/all') @login_required
def show_all():
    resp = make_response(redirect(url_for('.index'))) resp.set_cookie('show_followed', '',
        max_age=30*24*60*60) # 30 days return resp

@main.route('/followed') @login_required
def show_followed():
    resp = make_response(redirect(url_for('.index'))) resp.set_cookie('show_followed', '1',
        max_age=30*24*60*60) # 30 days return resp
```

Example 12-10. app/models.py: making users their own followers when they are created

```
class User(UserMixin, db.Model):
    # ...
    def __init__(self, **kwargs):
        # ...
        self.follow(self)
```

Unfortunately, you likely have several users in the database who are already created and are not following themselves. If the database is small and easy to regenerate, then it can be deleted and re-created, but if that is not an option, then adding an update function that fixes existing users is the proper solution. This is shown in **Example 12-11**.

Example 12-11. app/models.py: making users their own followers

```
class User(UserMixin, db.Model):
    # ...
    @staticmethod
    def add_self_follows():
        for user in User.query.all():
            if not user.is_following(user): user.follow(user)
            db.session.add(user) db.session.commit()
    # ...
```

Now the database can be updated by running the previous example function from the shell:

```
(venv) $ flask shell
>>> User.add_self_follows()
```

User Comments

Allowing users to interact is key to the success of a social blogging platform. In this chapter, you will learn how to implement user comments. The techniques presented are generic enough to be directly applicable to a large number of socially enabled applications.

Database Representation of Comments

Comments are not very different from blog posts. Both have a body, an author, and a timestamp, and in this particular implementation both are written with Markdown syntax. **Figure 13-1** shows a diagram of the `comments` table and its relationships with other tables in the database.



Figure 13-1. Database representation of blog post comments

Comments apply to specific blog posts, so a one-to-many relationship from the `posts` table is defined. This relationship can be used to obtain the list of comments associated with a particular blog post.

The `comments` table is also in a one-to-many relationship with the `users` table. This relationship gives access to all the comments made by a user, and indirectly how

Example 13-1. app/models.py: comment model

```
class Comment(db.Model):
    __tablename__ = 'comments'
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.Text)
    body_html = db.Column(db.Text)
    timestamp = db.Column(db.DateTime, index=True, default=datetime.utcnow)
    disabled = db.Column(db.Boolean)
    author_id = db.Column(db.Integer, db.ForeignKey('users.id'))
    post_id = db.Column(db.Integer, db.ForeignKey('posts.id'))

    @staticmethod
    def on_changed_body(target, value, oldvalue, initiator):
        allowed_tags = ['a', 'abbr', 'acronym', 'b', 'code', 'em', 'i',
                        'strong']
        target.body_html = bleach.linkify(bleach.clean( markdown(value,
                                                                output_format='html'),
                                                                tags=allowed_tags, strip=True))

db.event.listen(Comment.body, 'set', Comment.on_changed_body)
```

Example 13-2. app/models.py: one-to-many relationships from users and posts to comments

```
class User(db.Model):
    # ...
    comments = db.relationship('Comment', backref='author', lazy='dynamic')

class Post(db.Model):
    # ...
    comments = db.relationship('Comment', backref='post', lazy='dynamic')
```

Comment Submission and Display

Example 13-3. app/main/forms.py: comment input form

```
class CommentForm(FlaskForm):
    body = StringField('', validators=[DataRequired()]) submit =
    SubmitField('Submit')
```

Example 13-4. app/main/views.py: blog post comments support

```
@main.route('/post/<int:id>', methods=['GET', 'POST'])
def post(id):
    post = Post.query.get_or_404(id) form = CommentForm()
    if form.validate_on_submit():
        comment = Comment(body=form.body.data,
                           post=post, author=current_user._get_current_object())
        db.session.add(comment) db.session.commit()
        flash('Your comment has been published.')
        return redirect(url_for('.post', id=post.id, page=-1)) page =
request.args.get('page', 1, type=int)
if page == -1:
    page = (post.comments.count() - 1) // \
        current_app.config['FLASKY_COMMENTS_PER_PAGE'] + 1
    pagination = post.comments.order_by(Comment.timestamp.asc()).paginate( page,
        per_page=current_app.config['FLASKY_COMMENTS_PER_PAGE'], error_out=False)
    comments = pagination.items
    return render_template('post.html', posts=[post], form=form,
                           comments=comments, pagination=pagination)
```

Example 13-5. `_app/templates/_posts.html`: link to blog post comments

```
<a href="{{ url_for('.post', id=post.id) }}#comments">
  <span class="label label-primary">
    {{ post.comments.count() }} Comments
  </span>
</a>
```

Note how the text of the link includes the number of comments, which is easily obtained from the one-to-many relationship between the `posts` and `comments` tables using SQLAlchemy's `count()` filter.



Figure 13-2. Blog post comments

Comment Moderation

Example 13-6. app/templates/base.html: Moderate Comments link in navigation bar

```
...
{% if current_user.can(Permission.MODERATE) %}
<li><a href="{{ url_for('main.moderate') }}">Moderate Comments</a></li>
{% endif %}
...
```

Example 13-7. app/main/views.py: comment moderation route

```
@main.route('/moderate') @login_required
@permission_required(Permission.MODERATE)
def moderate():
    page = request.args.get('page', 1, type=int)
    pagination = Comment.query.order_by(Comment.timestamp.desc()).paginate( page,
        per_page=current_app.config['FLASKY_COMMENTS_PER_PAGE'], error_out=False)
    comments = pagination.items
    return render_template('moderate.html', comments=comments,
        pagination=pagination, page=page)
```

This is a very simple function that reads a page of comments from the database and passes them on to a template for rendering. Along with the comments, the template receives the pagination object and the current page number.

The *moderate.html* template, shown in **Example 13-8**, is also simple because it relies on the *_comments.html* subtemplate created earlier for the rendering of the comments.

Example 13-8. app/templates/moderate.html: comment moderation template

```
{% extends "base.html" %}
{% import "_macros.html" as macros %}

{% block title %}Flasky – Comment Moderation{% endblock %}

{% block page_content %}
<div class="page-header">
  <h1>Comment Moderation</h1>
</div>
{% set moderate = True %}
{% include '_comments.html' %}
{% if pagination %}
<div class="pagination">
  {{ macros.pagination_widget(pagination, '.moderate') }}
</div>
{% endif %}
{% endblock %}
```

Example 13-9. app/templates/_comments.html: rendering of the comment bodies

```
...
<div class="comment-body">
    {% if comment.disabled %}
    <p></p><i>This comment has been disabled by a moderator.</i></p>
    {% endif %}
    {% if moderate or not comment.disabled %}
        {% if comment.body_html %}
            {{ comment.body_html | safe }}
        {% else %}
            {{ comment.body }}
        {% endif %}
    {% endif %}
</div>

{% if moderate %}
    <br>
    {% if comment.disabled %}
    <a class="btn btn-default btn-xs" href="{{ url_for('.moderate_enable',
        id=comment.id, page=page) }}">Enable</a>
    {% else %}
    <a class="btn btn-danger btn-xs" href="{{ url_for('.moderate_disable',
        id=comment.id, page=page) }}">Disable</a>
    {% endif %}
{% endif %}
...
```

Example 13-10. app/main/views.py: comment moderation routes

```
@main.route('/moderate/enable/<int:id>') @login_required
@permission_required(Permission.MODERATE)

def moderate_enable(id):
    comment = Comment.query.get_or_404(id)
    comment.disabled = False db.session.add(comment)
    return redirect(url_for('.moderate',
                            page=request.args.get('page', 1, type=int)))

@main.route('/moderate/disable/<int:id>') @login_required
@permission_required(Permission.MODERATE)
def moderate_disable(id):
    comment = Comment.query.get_or_404(id)
    comment.disabled = True db.session.add(comment)
    return redirect(url_for('.moderate',
                            page=request.args.get('page', 1, type=int)))
```

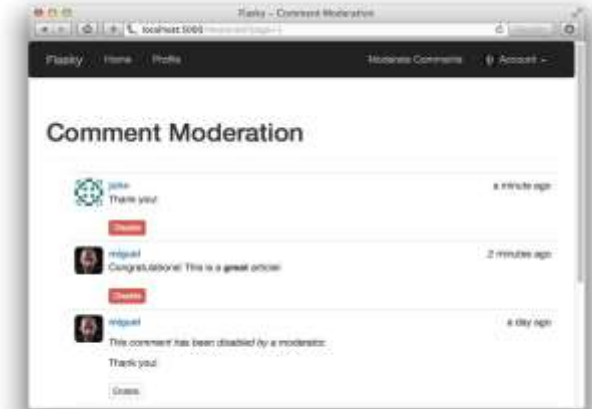


Figure 13-3. Comment moderation page

The comment enable and disable routes load the comment object, set the disabled field to the proper value, and write it back to the database. At the end, they redirect back to the comment moderation page (shown in **Figure 13-3**), and if a page argument was given in the query string, they include it in the redirect. The buttons in the `_comments.html` template are rendered with the page argument so that the redirect brings the user back to the same page.

Creating an API Blueprint

Example 14-2. app/api/__init__.py: API blueprint creation

```
from flask import Blueprint
```

```
api = Blueprint('api', __name__)
```

```
from . import authentication, posts, users, comments, errors
```

The structure of the blueprint package constructor is similar to that of the other blueprints. Importing all the components of the blueprint is necessary so that routes and other handlers are registered. Since many of these modules need to import the api blueprint referenced here, the imports are done at the bottom to help prevent errors due to circular dependencies.

Example 14-3. app/init.py: API blueprint registration

```
def create_app(config_name):  
    # ...  
    from .api import api as api_blueprint  
    app.register_blueprint(api_blueprint, url_prefix='/api/v1') # ...
```

The API blueprint is registered with a URL prefix, so that all its routes will have their URLs prefixed with `/api/v1`. Adding a prefix when registering the blueprint is a good idea because it eliminates the need to hardcode the version number in every blueprint route.

Error Handling

Example 14-4. app/api/errors.py: 404 error handler with HTTP content negotiation

```
@main.app_errorhandler(404)
def page_not_found(e):
    if request.accept_mimetypes.accept_json and \
        not request.accept_mimetypes.accept_html: response =
        jsonify({'error': 'not found'}) response.status_code = 404
    return response
return render_template('404.html'), 404
```

This new version of the error handler checks the `Accept` request header, which is decoded into `request.accept_mimetypes`, to determine what format the client wants the response in. Browsers generally do not specify any restrictions on response formats, but API clients typically do. The JSON response is generated only for clients that include JSON in their list of accepted formats, but not HTML.

The remaining status codes are generated explicitly by the web service, so they can be implemented as helper functions inside the blueprint in the `errors.py` module. **Example 14-5** shows the implementation of the 403 error; the others are similar.

Example 14-5. app/api/errors.py: API error handler for status code 403

```
def forbidden(message):  
    response = jsonify({'error': 'forbidden', 'message': message})  
    response.status_code = 403  
    return response
```

View functions in the API blueprint can invoke these auxiliary functions to generate error responses when necessary.

User Authentication with Flask-HTTPAuth

Web services, like regular web applications, need to protect information and ensure that it is not given to unauthorized parties. For this reason, RIAs must ask their users for login credentials and pass them to the server for verification. The HTTP authentication protocol is simple enough that it can be implemented directly, but the Flask-HTTPAuth extension provides a convenient wrapper that hides the protocol details in a decorator similar to Flask-Login's `login_required`.

Flask-HTTPAuth is installed with *pip*:

```
(venv) $ pip install flask-httpauth
```

To initialize the extension for HTTP Basic authentication, an object of class `HTTPBasicAuth` must be created. Like Flask-Login, Flask-HTTPAuth makes no assumptions about the procedure required to verify user credentials, so this information is given in a callback function. [Example 14-6](#) shows how the extension is initialized and provided with a verification callback.

Example 14-6. app/api/authentication.py: Flask-HTTPAuth initialization

```
from flask_httpauth import HTTPBasicAuth auth = HTTPBasicAuth()

@auth.verify_password
def verify_password(email, password):
    if email == '':
        return False

    user = User.query.filter_by(email = email).first()
    if not user:
        return False g.current_user = user
    return user.verify_password(password)
```

Because this type of user authentication will be used only in the API blueprint, the Flask-HTTPAuth extension is initialized in the blueprint package, and not in the application package like other extensions.

The email and password are verified using the existing support in the User model. The verification callback returns True when the login is valid and False otherwise. The Flask-HTTPAuth extension also will invoke the callback for requests that carry no authentication, setting both arguments to the empty string. In this case, when email is an empty string, the function immediately returns False to block the request; for certain applications it may be acceptable to allow the anonymous user by returning True. The authentication callback saves the authenticated user in Flask's g context variable so that the view function can access it later.

Example 14-7. _app/api/authentication.py: Flask-HTTPAuth error handler

```
from .errors import unauthorized @auth.error_handler
def auth_error():
    return unauthorized('Invalid credentials')
```

To protect a route, the `auth.login_required` decorator is used:

```
@api.route('/posts/') @auth.login_required def get_posts():
    pass
```

Example 14-8. app/api/authentication.py: before_request handler with authentication

```
from .errors import forbidden

@api.before_request @auth.login_required def before_request():
    if not g.current_user.is_anonymous and \
        not g.current_user.confirmed:
        return forbidden('Unconfirmed account')
```

Token-Based Authentication

Clients must send authentication credentials with every request. To avoid having to constantly transfer sensitive information such as a password, a token-based authentication solution can be used.

In token-based authentication, the client requests an access token by sending a request that includes the login credentials as authentication. The token can then be used in place of the login credentials to authenticate requests. For security reasons,

Example 14-9. app/models.py: token-based authentication support

```
class User(db.Model):
    # ...
    def generate_auth_token(self, expiration):
        s = Serializer(current_app.config['SECRET_KEY'],
                        expires_in=expiration)
        return s.dumps({'id': self.id}).decode('utf-8')

    @staticmethod
    def verify_auth_token(token):
        s = Serializer(current_app.config['SECRET_KEY'])
        try:
            data = s.loads(token)
        except:
            return None
        return User.query.get(data['id'])
```

Example 14-10. app/api/authentication.py: improved authentication verification with token support

```
@auth.verify_password
def verify_password(email_or_token, password):
    if email_or_token == '':
        return False
    if password == '':
        g.current_user = User.verify_auth_token(email_or_token)
        g.token_used = True
        return g.current_user is not None
    user = User.query.filter_by(email=email_or_token).first()
    if not user:
        return False
    g.current_user = user
    g.token_used = False
    return user.verify_password(password)
```

Example 14-11. app/api/authentication.py: authentication token generation

```
@api.route('/tokens/', methods=['POST'])
def get_token():
    if g.current_user.is_anonymous or g.token_used:
        return unauthorized('Invalid credentials')
    return jsonify({'token': g.current_user.generate_auth_token(
        expiration=3600), 'expiration': 3600})
```

Since this route is in the blueprint, the authentication mechanisms added to the `before_request` handler also apply to it. To prevent clients from authenticating to this route using a previously obtained token instead of an email address and password, the `g.token_used` variable is checked, and requests authenticated with a token are rejected. The purpose of this is to prevent users from bypassing the token expiration by requesting a new token using the old token as authentication. The function returns a token in the JSON response with a validity period of one hour. The period is also included in the JSON response.

Serializing Resources to and from JSON

A frequent need when writing a web service is to convert internal representations of resources to and from JSON, which is the transport format used in HTTP requests and responses. The process of converting an internal representation to a transport format such as JSON is called *serialization*. **Example 14-12** shows a new `to_json()` method added to the `Post` class.

Example 14-12. app/models.py: converting a post to a JSON serializable dictionary

```
class Post(db.Model):
    # ...
    def to_json(self):
        json_post = {
            'url': url_for('api.get_post', id=self.id), 'body': self.body,
            'body_html': self.body_html, 'timestamp': self.timestamp,
            'author_url': url_for('api.get_user', id=self.author_id), 'comments_url':
            url_for('api.get_post_comments', id=self.id), 'comment_count':
            self.comments.count()
        }
        return json_post
```

The `url`, `author_url`, and `comments_url` fields need to return the URLs for the respective resources, so these are generated with `url_for()` calls to other routes that will be defined in the API blueprint.

Example 14-13. app/models.py: converting a user to a JSON serializable dictionary

```
class User(UserMixin, db.Model):
    # ...
    def to_json(self):
        json_user = {
            'url': url_for('api.get_user', id=self.id), 'username': self.username,
            'member_since': self.member_since, 'last_seen': self.last_seen,
            'posts_url': url_for('api.get_user_posts', id=self.id), 'followed_posts_url':
            url_for('api.get_user_followed_posts',
                    id=self.id), 'post_count': self.posts.count()
        }
        return json_user
```

Example 14-14. app/models.py: creating a blog post from JSON

```
from app.exceptions import ValidationError

class Post(db.Model):
    # ...
    @staticmethod
    def from_json(json_post):
        body = json_post.get('body')
        if body is None or body == '':
            raise ValidationError('post does not have a body')
        return Post(body=body)
```

Example 14-15. app/exceptions.py: ValidationError exception

```
class ValidationError(ValueError): pass
```

The application now needs to handle this exception by providing the appropriate response to the client. To avoid having to add exception-catching code in view functions, a global exception handler can be installed using Flask's `errorhandler` decorator. A handler for the `ValidationError` exception is shown in [Example 14-16](#).

Example 14-16. app/api/errors.py: API error handler for ValidationError exceptions

```
@api.errorhandler(ValidationError)
def validation_error(e):
    return bad_request(e.args[0])
```

Using this technique, the code in view functions can be written very cleanly and concisely, without the need to include error checking. For example:

```
@api.route('/posts/', methods=['POST'])
def new_post():
    post = Post.from_json(request.json) post.author = g.current_user
    db.session.add(post) db.session.commit()
    return jsonify(post.to_json())
```

Implementing Resource Endpoints

Example 14-17. app/api/posts.py: GET resource handlers for posts

```
@api.route('/posts/')
def get_posts():
    posts = Post.query.all()
    return jsonify({ 'posts': [post.to_json() for post in posts] })

@api.route('/posts/<int:id>')
def get_post(id):
    post = Post.query.get_or_404(id)
    return jsonify(post.to_json())
```

Example 14-18. app/api/posts.py: POST resource handler for posts

```
@api.route('/posts/', methods=['POST'])
@permission_required(Permission.WRITE) def new_post():
    post = Post.from_json(request.json) post.author =
    g.current_user db.session.add(post) db.session.commit()
    return jsonify(post.to_json()), 201, \
        {'Location': url_for('api.get_post', id=post.id)}
```

Example 14-19. app/api/decorators.py: permission_required decorator

```
def permission_required(permission):
    def decorator(f): @wraps(f)
        def decorated_function(*args, **kwargs):
            if not g.current_user.can(permission):
                return forbidden('Insufficient permissions')
            return f(*args, **kwargs)
        return decorated_function
    return decorator
```

Example 14-20. app/api/posts.py: PUT resource handler for posts

```
@api.route('/posts/<int:id>', methods=['PUT'])
@permission_required(Permission.WRITE)
def edit_post(id):
    post = Post.query.get_or_404(id)
    if g.current_user != post.author and \
        not g.current_user.can(Permission.ADMIN):
        return forbidden('Insufficient permissions')
    post.body = request.json.get('body', post.body)
    db.session.add(post)
    db.session.commit()
    return jsonify(post.to_json())
```

The permission checks are more complex in this case. The standard check for permission to write blog posts is done with the decorator, but to allow a user to edit a blog post the function must also ensure that the user is the author of the post or else is an administrator. This check is added explicitly to the view function. If this check had to be added in many view functions, building a decorator for it would be a good way to avoid code repetition.

Pagination of Large Resource Collections

The GET requests that return a collection of resources can be extremely expensive and difficult to manage for very large collections. Like web applications, web services can choose to paginate collections.

Example 14-21. app/api/posts.py: Post pagination

```
@api.route('/posts/')
def get_posts():
    page = request.args.get('page', 1, type=int)
    pagination = Post.query.paginate(
        page, per_page=current_app.config['FLASKY_POSTS_PER_PAGE'], error_out=False)
    posts = pagination.items
    prev = None
    if pagination.has_prev:
        prev = url_for('api.get_posts', page=page-1)
    next = None
    if pagination.has_next:
        next = url_for('api.get_posts', page=page+1)
    return jsonify({
        'posts': [post.to_json() for post in posts],
        'prev_url': prev,
        'next_url': next,
        'count': pagination.total
    })
```

The `posts` field in the JSON response contains the data items as before, but now it is just a page and not the complete set. The `prev_url` and `next_url` items contain the resource URLs for the previous and following pages, or `None` when a page in that direction is not available. The `count` value is the total number of items in the collection.
