PART II

Example: A Social Blogging Application

CHAPTER 8
User Authentication

Most applications need to keep track of who its users are. When users connect with the 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.

The most commonly used method of authentication requires users to provide a piece of identification (either their email or username) and a secret password. In this chapter, the complete authentication system for Flasky is created.

Authentication Extensions for Flask

There are many excellent Python authentication packages, but none of them do everything. The user authentication solution presented in this chapter uses several packages and provides the glue that makes them work well together. This is the list of packages that will be used:

- 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

Password Security

The safety of user information stored in databases is often overlooked during the design of web applications. If an attacker is able to break into your server and access your user database, then you risk the security of your users, and the risk is bigger than you think. It is a known fact that most users use the same password on multiple sites, so even if you don't store any sensitive information, access to the passwords stored in your database can give the attacker access to accounts your users have on other sites.

The key to storing user passwords securely in a database relies not on storing the password itself but a *hash* of it. A password hashing function takes a password as input and applies one or more cryptographic transformations to it. The result is a new sequence of characters that has no resemblance to the original password. Password hashes can be verified in place of the real passwords because hashing functions are repeatable: given the same inputs, the result is always the same.

Password hashing is a complex task that is hard to get right. It is recommended that you don't implement your own solution but instead rely on reputable libraries that have been reviewed by the community. If you are interested in learning what's involved in generating secure password hashes, the article Salted Password Hashing - Doing it Right is a worthwhile read.

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:sha1, 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 retrieved from the database and the password entered by the user. A return value of `True` indicates that the 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 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 function is implemented through a write-only property called `password`. When this property is set, the setter method will call Werkzeug's `generate_password_hash()` function and write the result to the `password_hash` field. Attempting to read the `password` property will return an error, as clearly the original password cannot be recovered once hashed.

The `verify_password` method takes a password and passes it to Werkzeug's `check_password_hash()` function for verification against the hashed version stored in the `User` model. If this method returns `True`, then the password is correct.

If you have cloned the application's Git repository on GitHub, you can run `git checkout 8a` to check out this version of the application.

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

```
(venv) $ python manage.py shell
>>> u = User()
>>> u.password = 'cat'
>>> u.password_hash
'pbkdf2:sha1:1000$duxMk0OF$4735b293e397d6eeaf650aaf490fd9091f928bed'
>>> u.verify_password('cat')
True
>>> u.verify_password('dog')
False
>>> u2 = User()
>>> u2.password = 'cat'
>>> u2.password_hash
'pbkdf2:sha1:1000$UjvnGeTP$875e28eb0874f44101d6b332442218f66975ee89'
```

Note how users `u` and `u2` have completely different password hashes, even though they both use the same password. To ensure that this functionality continues to work in the

future, the above tests can be written as unit tests that can be repeated easily. In Example 8-2 a new module inside the *tests* package is shown with three new tests that exercise the recent changes to the `User` model.

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)
```

Creating an Authentication Blueprint

Blueprints were introduced in Chapter 7 as a way to define routes in the global scope after the creation of the application was moved into a factory function. The routes related to the user authentication system can be added to a `auth` blueprint. Using different blueprints for different sets of application functionality is a great way to keep the code neatly organized.

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: 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: Blueprint routes and view functions

```
from flask import render_template
from . import auth

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

Note that the template file given to `render_template()` is stored inside the *auth* folder. This folder must be created inside *app/templates*, as Flask expects the templates to be relative to the application's template folder. By storing the blueprint templates in their own folder, there is no risk of naming collisions with the `main` blueprint or any other blueprints that will be added in the future.

Blueprints can also be configured to have their own independent folder for templates. When multiple template folders have been configured, the `render_template()` function searches the templates folder configured for the application first and then searches the template folders defined by blueprints.

The `auth` blueprint needs to be attached to the application in the `create_app()` factory function as shown in Example 8-5.

Example 8-5. app/__init__.py: Blueprint attachment

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

    return app
```

The `url_prefix` argument in the blueprint registration is optional. When used, all the routes defined in the blueprint will be registered with the given prefix, in this case */auth*. For example, the */login* route will be registered as */auth/login*, and the fully qualified URL under the development web server then becomes *http://localhost:5000/auth/login*.

If you have cloned the application's Git repository on GitHub, you can run `git checkout 8b` to check out this version of the application.

User Authentication with Flask-Login

When users log in to the application, their authenticated state has to be recorded so that it is remembered as they navigate through different pages. Flask-Login is a small but extremely useful extension that specializes in managing this particular aspect of a user authentication system, without being tied to a specific authentication mechanism.

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

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

Preparing the User Model for Logins

To be able to work with the application's `User` model, the Flask-Login extension requires a few methods to be implemented by it. The required methods are shown in Table 8-1.

Table 8-1. Flask-Login user methods

Method	Description
`is_authenticated()`	Must return `True` if the user has login credentials or `False` otherwise.
`is_active()`	Must return `True` if the user is allowed to log in or `False` otherwise. A `False` return value can be used for disabled accounts.
`is_anonymous()`	Must always return `False` for regular users.
`get_id()`	Must return a unique identifier for the user, encoded as a Unicode string.

These four methods can be implemented directly as methods in the model class, but as an easier alternative Flask-Login provides a `UserMixin` class that has default implementations that are appropriate for most cases. The updated `User` model is shown in Example 8-6.

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

```
from flask.ext.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'))
```

Note that an `email` field was also added. In this application, users will log in with their email, as they are less likely to forget their email addresses than their usernames.

Flask-Login is initialized in the application factory function, as shown in Example 8-7.

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

```
from flask.ext.login import LoginManager

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

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

The `session_protection` attribute of the `LoginManager` object can be set to `None`, `'basic'`, or `'strong'` to provide different levels of security against user session tampering. With the `'strong'` setting, Flask-Login will keep track of the client's IP address and browser agent and will log the user out if it detects a change. The `login_view` attribute sets the endpoint for the login page. Recall that because the login route is inside a blueprint, it needs to be prefixed with the blueprint name.

Finally, Flask-Login requires the application to set up a callback function that loads a user, given the identifier. This function is shown in Example 8-8.

Example 8-8. app/models.py: User loader callback function

```
from . import login_manager

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

The user loader callback function receives a user identifier as a Unicode string. The return value of the function must be the user object if available or `None` otherwise.

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.ext.login import login_required

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

If this route is accessed by a user who is not authenticated, Flask-Login will intercept the request and send the user to the login page instead.

Adding a Login Form

The login form that will be presented to users has a text field for the email address, a password field, a "remember me" checkbox, and a submit button. The Flask-WTF form class is shown in Example 8-9.

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

```
from flask.ext.wtf import Form
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import Required, Email

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

The email field takes advantage of the `Length()` and `Email()` validators provided by WTForms. The `PasswordField` class represents an `<input>` element with `type="password"`. The `BooleanField` class represents a checkbox.

The template associated with the login page is stored in *auth/login.html*. This template just needs to render the form using Flask-Bootstrap's `wtf.quick_form()` macro. Figure 8-1 shows the login form rendered by the web browser.

The navigation bar in the *base.html* template uses a Jinja2 conditional to display "Sign In" or "Sign 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: Sign In and Sign Out navigation bar links

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

The `current_user` variable used in the conditional is defined by Flask-Login and is automatically available to view functions and templates. This variable contains the user currently logged in, or a proxy anonymous user object if the user is not logged in. Anonymous user objects respond to the `is_authenticated()` method with `False`, so this is a convenient way to know whether the current user is logged in.

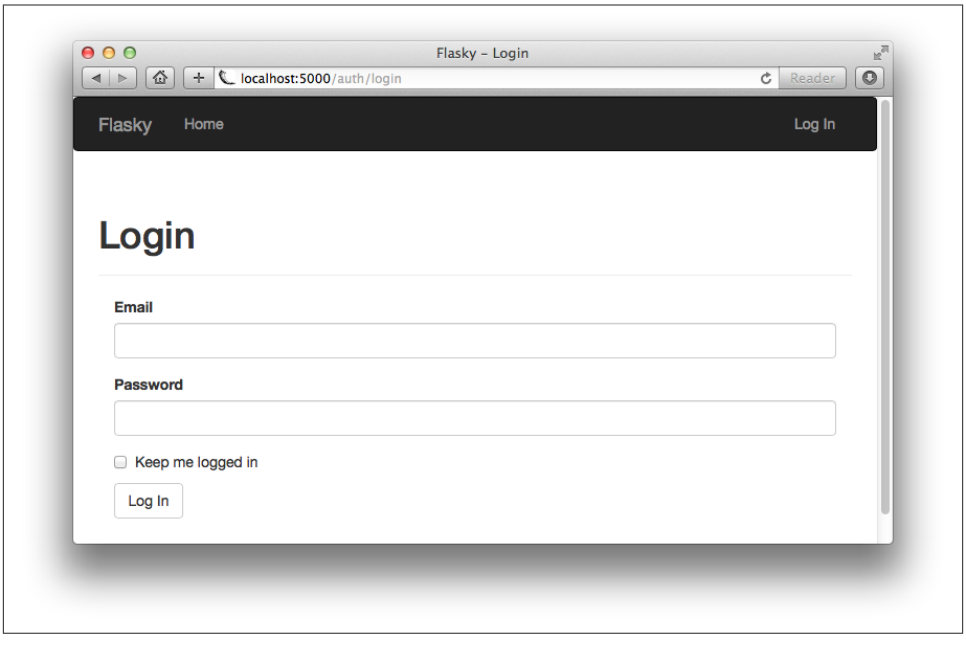

Figure 8-1. Login form

Signing Users In

The implementation of the `login()` view function is shown in Example 8-11.

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

```
from flask import render_template, redirect, request, url_for, flash
from flask.ext.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)
            return redirect(request.args.get('next') or url_for('main.index'))
        flash('Invalid username or password.')
    return render_template('auth/login.html', form=form)
```

The view function creates a `LoginForm` object and uses it like the simple form in Chapter 4. When the request is of type `GET`, the view function just renders the template, which

in turn displays the form. When the form is submitted in a `POST` request Flask-WTF's `validate_on_submit()` function validates the form variables, and then attempts to log the user in.

To log a user in, the function begins by loading the user from the database using the `email` provided with the form. If a user with the given email address exists, then its `verify_password()` method is called with the password that also came with the form. If the password is valid, Flask-Login's `login_user()` function is invoked to record the user as logged in for the user session. The `login_user()` function takes the user to log in and an optional "remember me" Boolean, which was also submitted with the form. A value of `False` for this argument causes the user session to expire when the browser window is closed, so the user will have to log in again next time. A value of `True` causes a long-term cookie to be set in the user's browser and with that the user session can be restored.

In accordance with the Post/Redirect/Get pattern discussed in Chapter 4, the `POST` request that submitted the login credentials ends with a redirect, but there are two possible URL destinations. If the login form was presented to the user to prevent unauthorized access to a protected URL, then Flask-Login saved the original URL in the `next` query string argument, which can be accessed from the `request.args` dictionary. If the `next` query string argument is not available, a redirect to the home page is issued instead. If the email or the password provided by the user are invalid, a flash message is set and the form is rendered again for the user to retry.

On a production server, the login route must be made available over secure HTTP so that the form data transmitted to the server is encrypted. Without secure HTTP, the login credentials can be intercepted during transit, defeating any efforts put into securing passwords in the server.

The login template needs to be updated to render the form. These changes are shown in Example 8-12.

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

```
{% 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

The implementation of the logout route is shown in Example 8-13.

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

```
from flask.ext.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'))
```

To log a user out, Flask-Login's `logout_user()` function is called to remove and reset the user session. The logout is completed with a flash message that confirms the action and a redirect to the home page.

If you have cloned the application's Git repository on GitHub, you can run `git checkout 8c` to check out this version of the application. This update contains a database migration, so remember to run `python manage.py db upgrade` after you check out the code. To ensure that you have all the dependencies installed, also run `pip install -r requirements.txt`.

Testing Logins

To verify that the login functionality is working, the home page can be updated to greet the logged-in user by name. The template section that generates the greeting is shown in Example 8-14.

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

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

In this template once again `current_user.is_authenticated()` is used to determine whether the user is logged in.

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

```
(venv) $ python manage.py 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.

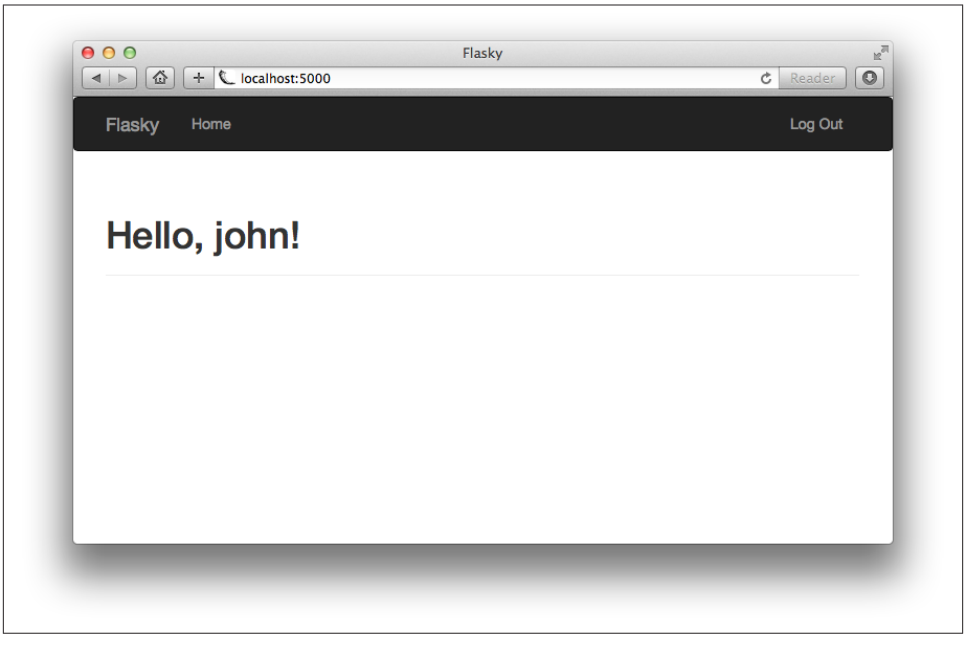

Figure 8-2. Home page after successful login

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. This form is shown in Example 8-15.

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

```
from flask.ext.wtf import Form
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import Required, Length, Email, Regexp, EqualTo
```

```
from wtforms import ValidationError
from ..models import User

class RegistrationForm(Form):
    email = StringField('Email', validators=[Required(), Length(1, 64),
                                             Email()])
    username = StringField('Username', validators=[
        Required(), 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=[
        Required(), EqualTo('password2', message='Passwords must match.')])
    password2 = PasswordField('Confirm password', validators=[Required()])
    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.')
```

This form uses the `Regexp` validator from WTForms to ensure that the `username` field contains letters, numbers, underscores, and dots only. The two arguments to the validator that follow the regular expression are the regular expression flags and the error message to display on failure.

The password is entered twice as a safety measure, but this step makes it necessary to validate that the two password fields have the same content, which is done with another validator from WTForms called `EqualTo`. This validator is attached to one of the password fields with the name of the other field given as an argument.

This form also has two custom validators implemented as methods. When a form defines a method with the prefix `validate_` followed by the name of a field, the method is invoked in addition to any regularly defined validators. In this case, the custom validators for `email` and `username` ensure that the values given are not duplicates. The custom validators indicate a validation error by throwing a `ValidationError` exception with the text of the error message as argument.

The template that presents this format 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.

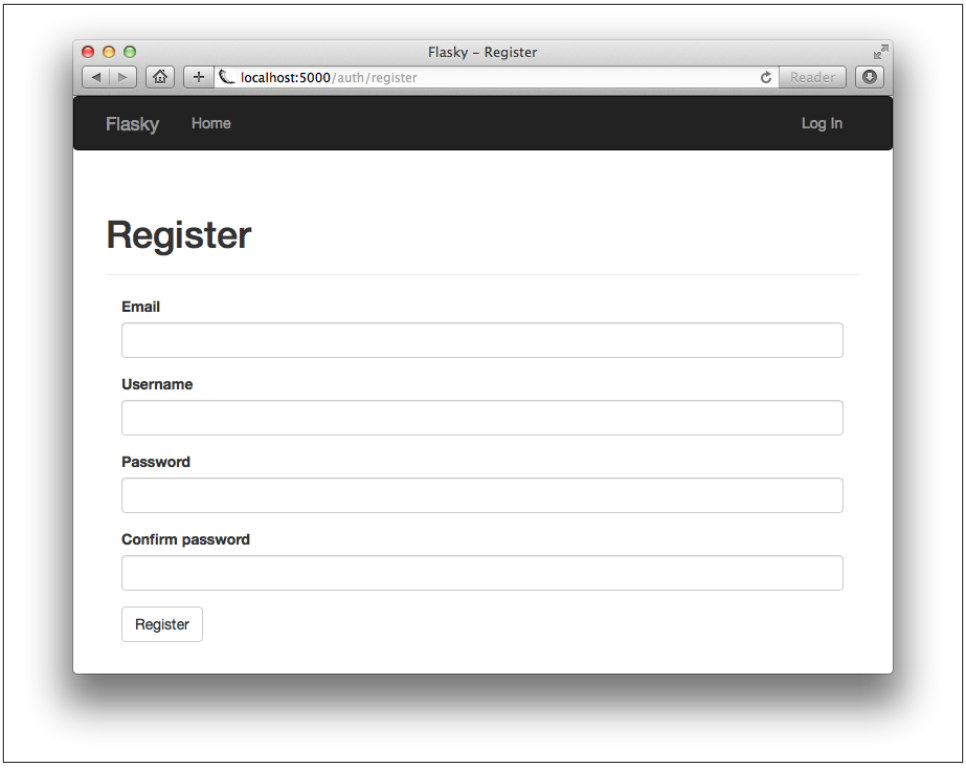

Figure 8-3. New user registration form

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?
    <a href="{{ url_for('auth.register') }}">
        Click here to register
    </a>
</p>
```

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 user-provided information. 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)
        flash('You can now login.')
        return redirect(url_for('auth.login'))
    return render_template('auth/register.html', form=form)
```

If you have cloned the application's Git repository on GitHub, you can run `git checkout 8d` to check out this version of the application.

Account Confirmation

For certain types of applications, it is important to ensure that the user information provided during registration is valid. A common requirement is to ensure that the user can be reached through the provided email address.

To validate the email address, applications send a confirmation email to users immediately after they register. The new account is initially marked as unconfirmed until the instructions in the email are followed, which proves that the user can be reached. The account confirmation procedure usually involves clicking a specially crafted URL link that includes a confirmation token.

Generating Confirmation Tokens with itsdangerous

The simplest account confirmation link would be a URL with the format *http://www.example.com/auth/confirm/<id>* included in the confirmation email, where *id* is the numeric `id` assigned to the user in the database. When the user clicks the link, the view function that handles this route receives the user `id` to confirm as an argument and can easily update the confirmed status of the user.

But this is obviously not a secure implementation, as any user who figures out the format of the confirmation links will be able to confirm arbitrary accounts just by sending random numbers in the URL. The idea is to replace the *id* in the URL with a token that contains the same information securely encrypted.

If you recall the discussion on user sessions in Chapter 4, Flask uses cryptographically signed cookies to protect the content of user sessions against tampering. These secure

cookies are signed by a package called *itsdangerous*. The same idea can be applied to confirmation tokens.

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

```
(venv) $ python manage.py shell
>>> from manage import app
>>> from itsdangerous import TimedJSONWebSignatureSerializer as Serializer
>>> s = Serializer(app.config['SECRET_KEY'], expires_in = 3600)
>>> token = s.dumps({ 'confirm': 23 })
>>> token
'eyJhbGciOiJIUzI1NiIsImV4cCI6MTM4MTcxODU1OCwiaWF0IjoxMzgxNzE0OTU4fQ.ey ...'
>>> data = s.loads(token)
>>> data
{u'confirm': 23}
```

Itsdangerous provides several types of token generators. Among them, the class `TimedJSONWebSignatureSerializer` generates JSON Web Signatures (JWS) with a time expiration. The constructor of this class takes an encryption key as argument, which in a Flask application can be the configured `SECRET_KEY`.

The `dumps()` method generates a cryptographic signature for the data given as an argument and then serializes the data plus the signature as a convenient token string. The `expires_in` argument sets an expiration time for the token expressed in seconds.

To decode the token, the serializer object provides a `loads()` method that takes the token as its only argument. The function verifies the signature and the expiration time and, if found valid, it returns the original data. When the `loads()` method is given an invalid token or a valid token that is expired, an exception is thrown.

Token generation and verification using this functionality can be added to the `User` model. The changes are shown in Example 8-18.

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})

    def confirm(self, token):
        s = Serializer(current_app.config['SECRET_KEY'])
        try:
```

```
            data = s.loads(token)
        except:
            return False
        if data.get('confirm') != self.id:
            return False
        self.confirmed = True
        db.session.add(self)
        return True
```

The `generate_confirmation_token()` method generates a token with a default validity time of one hour. The `confirm()` method verifies the token and, if valid, sets the new `confirmed` attribute to `True`.

In addition to verifying the token, the `confirm()` function checks that the `id` from the token matches the logged-in user, which is stored in `current_user`. This ensures that even if a malicious user figures out how to generate signed tokens, he or she cannot confirm somebody else's account.

Because a new column was added to the model to track the confirmed state of each account, a new database migration needs to be generated and applied.

The two new methods added to the `User` model are easily tested in unit tests. You can find the unit tests in the GitHub repository for the application.

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. This change is shown in Example 8-19.

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)
```

Note that a `db.session.commit()` call had to be added, even though the application configured automatic database commits at the end of the request. The problem is that new users get assigned an `id` when they are committed to the database. Because the `id` is needed for the confirmation token, the commit cannot be delayed.

The email templates used by the authentication blueprint will be added in the folder *templates/auth/email* to keep them separate from the HTML templates. As discussed in Chapter 6, for each email two templates are needed for the plain- and rich-text versions of the body. As an example, Example 8-20 shows the plain-text version of the confirmation email template, and you can find the equivalent HTML version in the GitHub repository.

Example 8-20. app/auth/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.
```

By default, `url_for()` generates relative URLs, so, for example, `url_for('auth.confirm', token='abc')` returns the string `'/auth/confirm/abc'`. This, of course, is not a valid URL that can be sent in an email. Relative URLs work fine when they are used within the context of a web page because the browser converts them to absolute by adding the hostname and port number from the current page, but when sending a URL over email there is no such context. The `_external=True` argument is added to the `url_for()` call to request a fully qualified URL that includes the scheme (*http://* or *https://*), hostname, and port.

The view function that confirms accounts is shown in Example 8-21.

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

```
from flask.ext.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):
        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.

The function first checks if the logged-in user is already confirmed, and in that case it redirects to the home page, as obviously there is nothing to do. This can prevent unnecessary work if a user clicks the confirmation token multiple times by mistake.

Because the actual token confirmation is done entirely in the `User` model, all the view function needs to do is call the `confirm()` method and then flash a message according to the result. When the confirmation succeeds, the `User` model's `confirmed` attribute is changed and added to the session, which will be committed when the request ends.

Each application can decide what unconfirmed users are allowed to do before they confirm their account. One possibility is to allow unconfirmed users to log in, but only show them a page that asks them to confirm their accounts before they can gain access.

This step can be done using Flask's `before_request` hook, which was briefly described in Chapter 2. From a blueprint, the `before_request` hook applies only to requests that belong to the blueprint. To install a hook for all application requests from a blueprint, the `before_app_request` decorator must be used instead. Example 8-22 shows how this handler is implemented.

Example 8-22. app/auth/views.py: Filter unconfirmed accounts in before_app_request handler

```
@auth.before_app_request
def before_request():
    if current_user.is_authenticated() \
            and not current_user.confirmed \
            and request.endpoint[:5] != 'auth.':
        return redirect(url_for('auth.unconfirmed'))

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

The `before_app_request` handler will intercept a request when three conditions are true:

1. A user is logged in (`current_user.is_authenticated()` must return `True`).
2. The account for the user is not confirmed.
3. The requested endpoint (accessible as `request.endpoint`) is outside of the authentication blueprint. Access to the authentication routes needs to be granted, as those are the routes that will enable the user to confirm the account or perform other account management functions.

If the three conditions are met, then a redirect is issued to a new */auth/unconfirmed* route that shows a page with information about account confirmation.

When a `before_request` or `before_app_request` callback returns a response or a redirect, Flask sends that to the client without invoking the view function associated with the request. This effectively allows these callbacks to intercept a request when necessary.

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 account 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.

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

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

This route repeats what was done in the registration route using `current_user`, the user who is logged in, as the target user. This route is also protected with `login_required` to ensure that when it is accessed, the user that is making the request is known.

If you have cloned the application's Git repository on GitHub, you can run `git checkout 8e` to check out this version of the application. This update contains a database migration, so remember to run `python manage.py db upgrade` after you check out the code.

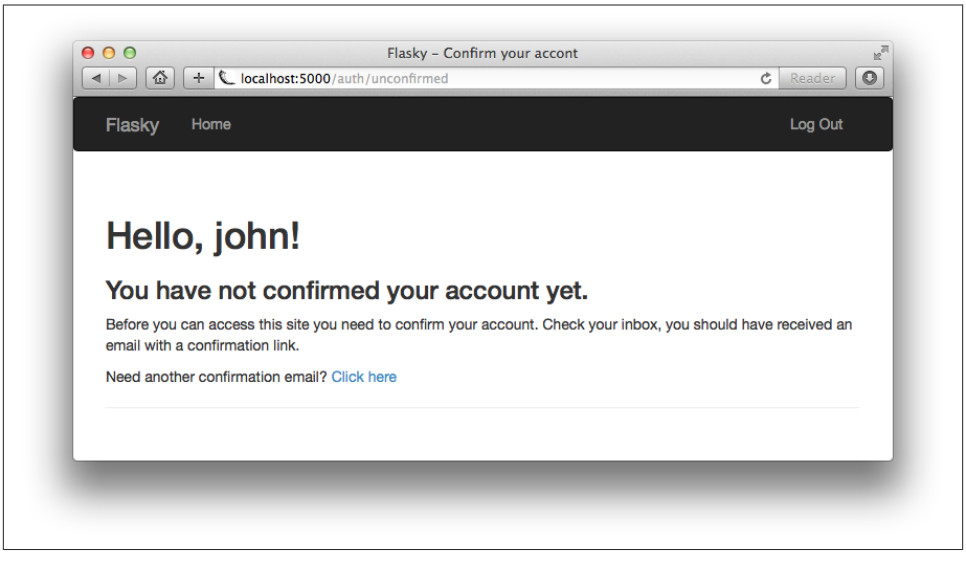

Figure 8-4. Unconfirmed account page

Account Management

Users who have accounts with the application may need to make changes to their accounts from time to time. The following tasks can be added to the authentication blueprint using the techniques presented in this chapter:

Password updates

Security conscious users may want to change their passwords periodically. This is an easy feature to implement, because as long as the user is logged in, it is safe to present a form that asks for the old password and a new password to replace it. (This feature is implemented as commit `8f` in the GitHub repository.)

Password resets

To avoid locking users out of the application when they forget their passwords, a password reset option can be offered. To implement password resets in a secure way, it is necessary to use tokens similar to those used to confirm accounts. When a user requests a password reset, an email with a reset token is sent to the registered email address. The user then clicks the link in the email and, after the token is verified, a form is presented where a new password can be entered. (This feature is implemented as commit `8g` in the GitHub repository.)

Email address changes

Users can be given the option to change the registered email address, but before the new address is accepted it must be verified with a confirmation email. To use this

feature, the user enters the new email address in a form. To confirm the email address, a token is emailed to that address. When the server receives the token back, it can update the user object. While the server waits to receive the token, it can store the new email address in a new database field reserved for pending email addresses, or it can store the address in the token along with the `id`. (This feature is implemented as commit `8h` in the GitHub repository.)

In the next chapter, the user subsystem of Flasky will be extended through the use of user roles.