The Flask Mega-Tutorial Part X: Email Support (/post/the-flask-mega-tutorial-part-x-email-support)

February 6 2018

Posted by Miguel Grinberg (/author/Miguel Grinberg) under Programming (/category/Programming), Python (/category/Python), Flask (/category/Flask).

This is the tenth installment of the Flask Mega-Tutorial series, in which I'm going to tell you how your application can send emails to your users, and how to build a password recovery feature on top of the email support.

For your reference, below is a list of the articles in this series.

- Chapter 1: Hello, World! (/post/the-flask-mega-tutorial-part-i-hello-world)
- Chapter 2: Templates (/post/the-flask-mega-tutorial-part-ii-templates)
- Chapter 3: Web Forms (/post/the-flask-mega-tutorial-part-iii-web-forms)
- Chapter 4: Database (/post/the-flask-mega-tutorial-part-iv-database)
- Chapter 5: User Logins (/post/the-flask-mega-tutorial-part-v-user-logins)
- Chapter 6: Profile Page and Avatars (/post/the-flask-mega-tutorial-part-vi-profile-page-and-avatars)
- Chapter 7: Error Handling (/post/the-flask-mega-tutorial-part-vii-error-handling)
- Chapter 8: Followers (/post/the-flask-mega-tutorial-part-viii-followers)
- Chapter 9: Pagination (/post/the-flask-mega-tutorial-part-ix-pagination)
- Chapter 10: Email Support (/post/the-flask-mega-tutorial-part-x-email-support) (this article)
- Chapter 11: Facelift (/post/the-flask-mega-tutorial-part-xi-facelift)
- Chapter 12: Dates and Times (/post/the-flask-mega-tutorial-part-xii-dates-and-times)
- Chapter 13: I18n and L10n (/post/the-flask-mega-tutorial-part-xiii-i18n-and-l10n)
- Chapter 14: Ajax (/post/the-flask-mega-tutorial-part-xiv-ajax)
- Chapter 15: A Better Application Structure (/post/the-flask-mega-tutorial-part-xv-a-better-application-structure)
- Chapter 16: Full-Text Search (/post/the-flask-mega-tutorial-part-xvi-full-text-search)
- Chapter 17: Deployment on Linux (/post/the-flask-mega-tutorial-part-xvii-deployment-on-linux)
- Chapter 18: Deployment on Heroku (/post/the-flask-mega-tutorial-part-xviii-deployment-on-heroku)
- Chapter 19: Deployment on Docker Containers (/post/the-flask-mega-tutorial-part-xix-deployment-on-docker-containers)

- Chapter 20: Some JavaScript Magic (/post/the-flask-mega-tutorial-part-xx-some-javascript-magic)
- Chapter 21: User Notifications (/post/the-flask-mega-tutorial-part-xxi-user-notifications)
- Chapter 22: Background Jobs (/post/the-flask-mega-tutorial-part-xxii-background-jobs)
- Chapter 23: Application Programming Interfaces (APIs) (/post/the-flask-mega-tutorial-part-xxiii-application-programming-interfaces-apis)

Note 1: If you are looking for the legacy version of this tutorial, it's here (/post/the-flask-mega-tutorial-part-i-hello-world-legacy).

Note 2: If you would like to support my work on this blog, or just don't have patience to wait for weekly articles, I am offering the complete version of this tutorial packaged as an ebook or a set of videos. For more information, visit courses.miguelgrinberg.com (https://courses.miguelgrinberg.com).

The application is doing pretty well on the database front now, so in this chapter I want to depart from that topic and add another important piece that most web applications need, which is the sending of emails.

Why does an application need to email its users? There are many reasons, but one common one is to solve authentication related problems. In this chapter I'm going to add a password reset feature for users that forget their password. When a user requests a password reset, the application will send an email with a specially crafted link. The user then needs to click that link to have access to a form in which to set a new password.

The GitHub links for this chapter are: Browse (https://github.com/miguelgrinberg/microblog/tree/v0.10), Zip (https://github.com/miguelgrinberg/microblog/archive/v0.10.zip), Diff (https://github.com/miguelgrinberg/microblog/compare/v0.9...v0.10).

Introduction to Flask-Mail

As far as the actual sending of emails, Flask has a popular extension called Flask-Mail (https://pythonhosted.org/Flask-Mail/) that can make the task very easy. As always, this extension is installed with pip:

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

The password reset links will have a secure token in them. To generate these tokens, I'm going to use JSON Web Tokens (https://jwt.io), which also have a popular Python package:

```
(venv) $ pip install pyjwt
```

The Flask-Mail extension is configured from the `app.config` object. Remember when in Chapter 7 (/post/the-flask-mega-tutorial-part-vii-error-handling) I added the email configuration for sending yourself an email whenever an error occurred in production? I did not tell you this then, but my choice of configuration variables was modeled after Flask-Mail's requirements, so there isn't really any additional work that is needed, the configuration variables are already in the application.

Like most Flask extensions, you need to create an instance right after the Flask application is created. In this case this is an object of class `Mail`:

app/__init__.py: Flask-Mail instance.

```
# ...
from flask_mail import Mail

app = Flask(__name__)
# ...
mail = Mail(app)
```

If you are planning to test sending of emails you have the same two options I mentioned in Chapter 7 (/post/the-flask-mega-tutorial-part-vii-error-handling). If you want to use an emulated email server, Python provides one that is very handy that you can start in a second terminal with the following command:

```
(venv) $ python -m smtpd -n -c DebuggingServer localhost:8025
```

To configure for this server you will need to set two environment variables:

```
(venv) $ export MAIL_SERVER=localhost
(venv) $ export MAIL_PORT=8025
```

If you prefer to have emails sent for real, you need to use a real email server. If you have one, then you just need to set the `MAIL_SERVER`, `MAIL_PORT`, `MAIL_USE_TLS`, `MAIL_USERNAME` and `MAIL_PASSWORD` environment variables for it. If you want a quick solution, you can use a Gmail account to send email, with the following settings:

```
(venv) $ export MAIL_SERVER=smtp.googlemail.com
(venv) $ export MAIL_PORT=587
(venv) $ export MAIL_USE_TLS=1
(venv) $ export MAIL_USERNAME=<your-gmail-username>
(venv) $ export MAIL_PASSWORD=<your-gmail-password>
```

If you are using Microsoft Windows, you need to replace `export` with `set` in each of the `export` statements above.

Remember that the security features in your Gmail account may prevent the application from sending emails through it unless you explicitly allow "less secure apps" access to your Gmail account. You can read about this here (https://support.google.com/accounts/answer/6010255?hl=en), and if you are concerned about the security of your account, you can create a secondary account that you configure just for testing emails, or you can enable less secure apps only temporarily to run your tests and then revert back to the more secure default.

Flask-Mail Usage

To learn how Flask-Mail works, I'll show you how to send an email from a Python shell. So fire up Python with `flask shell`, and then run the following commands:

```
>>> from flask_mail import Message
>>> from app import mail
>>> msg = Message('test subject', sender=app.config['ADMINS'][0],
... recipients=['your-email@example.com'])
>>> msg.body = 'text body'
>>> msg.html = '<h1>HTML body</h1>'
>>> mail.send(msg)
```

The snippet of code above will send an email to a list of email addresses that you put in the `recipients` argument. I put the sender as the first configured admin (I've added the `ADMINS` configuration variable in Chapter 7 (/post/the-flask-mega-tutorial-part-vii-error-handling)). The email will have plain text and HTML versions, so depending on how your email client is configured you may see one or the other.

So as you see, this is pretty simple. Now let's integrate emails into the application.

A Simple Email Framework

I will begin by writing a helper function that sends an email, which is basically a generic version of the shell exercise from the previous section. I will put this function in a new module called `app/email.py`:

app/email.py: Email sending wrapper function.

```
from flask_mail import Message
from app import mail

def send_email(subject, sender, recipients, text_body, html_body):
    msg = Message(subject, sender=sender, recipients=recipients)
    msg.body = text_body
    msg.html = html_body
    mail.send(msg)
```

Flask-Mail supports some features that I'm not utilizing here such as Cc and Bcc lists. Be sure to check the Flask-Mail Documentation (https://pythonhosted.org/Flask-Mail/) if you are interested in those options.

Requesting a Password Reset

As I mentioned above, I want users to have the option to request their password to be reset. For this purpose I'm going to add a link in the login page:

app/templates/login.html: Password reset link in login form.

```
    <p>
        Forgot Your Password?
        <a href="{{ url_for('reset_password_request') }}">Click to Reset It</a>
    </p>
```

When the user clicks the link, a new web form will appear that requests the user's email address as a way to initiate the password reset process. Here is the form class:

app/forms.py: Reset password request form.

```
class ResetPasswordRequestForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Email()])
    submit = SubmitField('Request Password Reset')
```

And here is the corresponding HTML template:

app/templates/reset_password_request.html: Reset password request template.

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

{% block content %}
    <h1>Reset Password</h1>
    <form action="" method="post">
        {{ form.hidden_tag() }}
        <p>
            {{ form.email.label }}<br>
            {{ form.email(size=64) }}<br>
            {% for error in form.email.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

I also need a view function to handle this form:

app/routes.py: Reset password request view function.

```
from app.forms import ResetPasswordRequestForm
from app.email import send_password_reset_email

@app.route('/reset_password_request', methods=['GET', 'POST'])
def reset_password_request():
    if current_user.is_authenticated:
        return redirect(url_for('index'))
    form = ResetPasswordRequestForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        if user:
            send_password_reset_email(user)
        flash('Check your email for the instructions to reset your password')
        return redirect(url_for('login'))
    return render_template('reset_password_request.html',
                           title='Reset Password', form=form)
```

This view function is fairly similar to others that process a form. I start by making sure the user is not logged in. If the user is logged in, then there is no point in using the password reset functionality, so I redirect to the index page.

When the form is submitted and valid, I look up the user by the email provided by the user in the form. If I find the user, I send a password reset email. I'm using a `send_password_reset_email()` helper function to do this. I will show you this function below.

After the email is sent, I flash a message directing the user to look for the email for further instructions, and then redirect back to the login page. You may notice that the flashed message is displayed even if the email provided by the user is unknown. This is so that clients cannot use this form to figure out if a given user is a member or not.

Password Reset Tokens

Before I implement the `send_password_reset_email()` function, I need to have a way to generate a password request link. This is going to be the link that is sent to the user via email. When the link is clicked, a page where a new password can be set is presented to the user. The tricky part of this plan is to make sure that only valid reset links can be used to reset an account's password.

The links are going to be provisioned with a *token*, and this token will be validated before allowing the password change, as proof that the user that requested the email has access to the email address on the account. A very popular token standard for this type of process is the JSON Web Token, or JWT. The nice thing about JWTs is that they are self contained. You can send a token to a user in an email, and when the user clicks the link that feeds the token back into the application, it can be verified on its own.

How do JWTs work? Nothing better than a quick Python shell session to understand them:

```
>>> import jwt
>>> token = jwt.encode({'a': 'b'}, 'my-secret', algorithm='HS256')
>>> token
b'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhIjoiYiJ9.dvOo58OBDHiuSHD4uW88nfJikhYAXc_sfUHq1mDi4G0'
>>> jwt.decode(token, 'my-secret', algorithms=['HS256'])
{'a': 'b'}
```

The `{'a': 'b'}` dictionary is an example payload that is going to be written into the token. To make the token secure, a secret key needs to be provided to be used in creating a cryptographic signature. For this example I have used the string `'my-secret'`, but with the application I'm going to use the `SECRET_KEY` from the configuration. The `algorithm` argument specifies how the token is to be generated. The `HS256` algorithm is the most widely used.

As you can see the resulting token is a long sequence of characters. But do not think that this is an encrypted token. The contents of the token, including the payload, can be decoded easily by anyone (don't believe me? Copy the above token and then enter it in the JWT debugger (https://jwt.io/#debugger-io) to see its contents). What makes the token secure is that the payload is *signed*. If somebody tried to forge or tamper with the payload in a token, then the signature would be invalidated, and to generate a new signature the secret key is needed. When a token is verified, the contents of the payload are decoded and returned back to the caller. If the token's signature was validated, then the payload can be trusted as authentic.

The payload that I'm going to use for the password reset tokens is going to have the format `{'reset_password': user_id, 'exp': token_expiration}`. The `exp` field is standard for JWTs and if present it indicates an expiration time for the token. If a token has a valid signature, but it is past its expiration timestamp, then it will also be considered invalid. For the password reset feature, I'm going to give these tokens 10 minutes of life.

When the user clicks on the emailed link, the token is going to be sent back to the application as part of the URL, and the first thing the view function that handles this URL will do is to verify it. If the signature is valid, then the user can be identified by the ID stored in the payload. Once the user's identity is known, the application can ask for a new password and set it on the user's account.

Since these tokens belong to users, I'm going to write the token generation and verification functions as methods in the `User` model:

app/models.py: Reset password token methods.

```
from time import time
import jwt
from app import app

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

    def get_reset_password_token(self, expires_in=600):
        return jwt.encode(
            {'reset_password': self.id, 'exp': time() + expires_in},
            app.config['SECRET_KEY'], algorithm='HS256').decode('utf-8')

    @staticmethod
    def verify_reset_password_token(token):
        try:
            id = jwt.decode(token, app.config['SECRET_KEY'],
                            algorithms=['HS256'])['reset_password']
        except:
            return
        return User.query.get(id)
```

The `get_reset_password_token()` function generates a JWT token as a string. Note that the `decode('utf-8')` is necessary because the `jwt.encode()` function returns the token as a byte sequence, but in the application it is more convenient to have the token as a string.

The `verify_reset_password_token()` is a static method, which means that it can be invoked directly from the class. A static method is similar to a class method, with the only difference that static methods do not receive the class as a first argument. This method takes a token and attempts to decode it by invoking PyJWT's `jwt.decode()` function. If the token cannot be validated or is expired, an exception will be raised, and in that case I

catch it to prevent the error, and then return `None` to the caller. If the token is valid, then the value of the `reset_password` key from the token's payload is the ID of the user, so I can load the user and return it.

Sending a Password Reset Email

Now that I have the tokens, I can generate the password reset emails. The `send_password_reset_email()` function relies on the `send_email()` function I wrote above.

app/email.py: Send password reset email function.

```
from flask import render_template
from app import app

# ...

def send_password_reset_email(user):
    token = user.get_reset_password_token()
    send_email('[Microblog] Reset Your Password',
               sender=app.config['ADMINS'][0],
               recipients=[user.email],
               text_body=render_template('email/reset_password.txt',
                                         user=user, token=token),
               html_body=render_template('email/reset_password.html',
                                         user=user, token=token))
```

The interesting part in this function is that the text and HTML content for the emails is generated from templates using the familiar `render_template()` function. The templates receive the user and the token as arguments, so that a personalized email message can be generated. Here is the text template for the reset password email:

app/templates/email/reset_password.txt: Text for password reset email.

```
Dear {{ user.username }},

To reset your password click on the following link:

{{ url_for('reset_password', token=token, _external=True) }}

If you have not requested a password reset simply ignore this message.

Sincerely,

The Microblog Team
```

And here is the nicer HTML version of the same email:

app/templates/email/reset_password.html: HTML for password reset email.

```
<p>Dear {{ user.username }},</p>
<p>
    To reset your password
    <a href="{{ url_for('reset_password', token=token, _external=True) }}">
        click here
    </a>.
</p>
<p>Alternatively, you can paste the following link in your browser's address bar:</p>
<p>{{ url_for('reset_password', token=token, _external=True) }}</p>
<p>If you have not requested a password reset simply ignore this message.</p>
<p>Sincerely,</p>
<p>The Microblog Team</p>
```

The `reset_password` route that is referenced in the `url_for()` call in these two email templates does not exist yet, this will be added in the next section. The `_external=True` argument that I included in the `url_for()` calls in both templates is also new. The URLs that are generated by `url_for()` by default are relative URLs, so for example, the `url_for('user', username='susan')` call would return */user/susan*. This is normally sufficient for links that are generated in web pages, because the web browser takes the remaining parts of the URL from the current page. When sending a URL by email however, that context does not exist, so fully qualified URLs need to be used. When `_external=True` is passed as an argument, complete URLs are generated, so the previous example would return *http://localhost:5000/user/susan*, or the appropriate URL when the application is deployed on a domain name.

Resetting a User Password

When the user clicks on the email link, a second route associated with this feature is triggered. Here is the password request view function:

app/routes.py: Password reset view function.

```
from app.forms import ResetPasswordForm

@app.route('/reset_password/<token>', methods=['GET', 'POST'])
def reset_password(token):
    if current_user.is_authenticated:
        return redirect(url_for('index'))
    user = User.verify_reset_password_token(token)
    if not user:
        return redirect(url_for('index'))
    form = ResetPasswordForm()
    if form.validate_on_submit():
        user.set_password(form.password.data)
        db.session.commit()
        flash('Your password has been reset.')
        return redirect(url_for('login'))
    return render_template('reset_password.html', form=form)
```

In this view function I first make sure the user is not logged in, and then I determine who the user is by invoking the token verification method in the `User` class. This method returns the user if the token is valid, or `None` if not. If the token is invalid I redirect to the home page.

If the token is valid, then I present the user with a second form, in which the new password is requested. This form is processed in a way similar to previous forms, and as a result of a valid form submission, I invoke the `set_password()` method of `User` to change the password, and then redirect to the login page, where the user can now login.

Here is the `ResetPasswordForm` class:

app/forms.py: Password reset form.

```
class ResetPasswordForm(FlaskForm):
    password = PasswordField('Password', validators=[DataRequired()])
    password2 = PasswordField(
        'Repeat Password', validators=[DataRequired(), EqualTo('password')])
    submit = SubmitField('Request Password Reset')
```

And here is the corresponding HTML template:

app/templates/reset_password.html: Password reset form template.

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

{% block content %}
    <h1>Reset Your Password</h1>
    <form action="" method="post">
        {{ form.hidden_tag() }}
        <p>
            {{ form.password.label }}<br>
            {{ form.password(size=32) }}<br>
            {% for error in form.password.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>
            {{ form.password2.label }}<br>
            {{ form.password2(size=32) }}<br>
            {% for error in form.password2.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

The password reset feature is now complete, so make sure you try it.

Asynchronous Emails

If you are using the simulated email server that Python provides you may not have noticed this, but sending an email slows the application down considerably. All the interactions that need to happen when sending an email make the task slow, it usually takes a few seconds to get an email out, and maybe more if the email server of the addressee is slow, or if there are multiple addressees.

What I really want is for the `send_email()` function to be *asynchronous*. What does that mean? It means that when this function is called, the task of sending the email is scheduled to happen in the background, freeing the `send_email()` to return immediately so that the application can continue running concurrently with the email being sent.

Python has support for running asynchronous tasks, actually in more than one way. The `threading` and `multiprocessing` modules can both do this. Starting a background thread for email being sent is much less resource intensive than starting a brand new process, so I'm going to go with that approach:

app/email.py: Send emails asynchronously.

```
from threading import Thread
# ...

def send_async_email(app, msg):
    with app.app_context():
        mail.send(msg)


def send_email(subject, sender, recipients, text_body, html_body):
    msg = Message(subject, sender=sender, recipients=recipients)
    msg.body = text_body
    msg.html = html_body
    Thread(target=send_async_email, args=(app, msg)).start()
```

The `send_async_email` function now runs in a background thread, invoked via the `Thread()` class in the last line of `send_email()`. With this change, the sending of the email will run in the thread, and when the process completes the thread will end and clean itself up. If you have configured a real email server, you will definitely notice a speed improvement when you press the submit button on the password reset request form.

You probably expected that only the `msg` argument would be sent to the thread, but as you can see in the code, I'm also sending the application instance. When working with threads there is an important design aspect of Flask that needs to be kept in mind. Flask uses *contexts* to avoid having to pass arguments across functions. I'm not going to go into a lot of detail on this, but know that there are two types of contexts, the *application*

context and the *request context*. In most cases, these contexts are automatically managed by the framework, but when the application starts custom threads, contexts for those threads may need to be manually created.

There are many extensions that require an application context to be in place to work, because that allows them to find the Flask application instance without it being passed as an argument. The reason many extensions need to know the application instance is because they have their configuration stored in the `app.config` object. This is exactly the situation with Flask-Mail. The `mail.send()` method needs to access the configuration values for the email server, and that can only be done by knowing what the application is. The application context that is created with the `with app.app_context()` call makes the application instance accessible via the `current_app` variable from Flask.