



The Flask Mega-Tutorial, Part X: Email Support

Posted by [Miguel Grinberg](#) on [December 3, 2023](#) under [Programming](#)
[Python](#) [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.

You are reading the 2024 edition of the Flask Mega-Tutorial. The complete course is also available to order in e-book and paperback formats from [Amazon](#). Thank you for your support!

If you are looking for the 2018 edition of this course, you can find it [here](#).

For your reference, here is the complete list of articles in this series:

- [Chapter 1: Hello, World!](#)
- [Chapter 2: Templates](#)
- [Chapter 3: Web Forms](#)
- [Chapter 4: Database](#)
- [Chapter 5: User Logins](#)
- [Chapter 6: Profile Page and Avatars](#)
- [Chapter 7: Error Handling](#)
- [Chapter 8: Followers](#)
- [Chapter 9: Pagination](#)
- [Chapter 10: Email Support](#) (this article)
- [Chapter 11: Facelift](#)
- [Chapter 12: Dates and Times](#)
- [Chapter 13: I18n and L10n](#)
- [Chapter 14: Ajax](#)
- [Chapter 15: A Better Application Structure](#)
- [Chapter 16: Full-Text Search](#)
- [Chapter 17: Deployment on Linux](#)
- [Chapter 18: Deployment on Heroku](#)
- [Chapter 19: Deployment on Docker Containers](#)
- [Chapter 20: Some JavaScript Magic](#)
- [Chapter 21: User Notifications](#)
- [Chapter 22: Background Jobs](#)
- [Chapter 23: Application Programming Interfaces \(APIs\)](#)

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 will then need 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](#), [Zip](#), [Diff](#).

Introduction to Flask-Mail

As far as the actual sending of emails, Flask has a popular extension called [Flask-Mail](#). 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](#), 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](#) 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 options I mentioned in [Chapter 7](#). If you want to use an emulated email server, then you can start the same SMTP debugging server used earlier in a second terminal with the following command:

```
(venv) $aiosmtpd -n -c aiosmtpd.handlers.Debugging -l localhost:8025
```

To configure the application to use 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.

Unfortunately the security features in your Gmail account may prevent the application from sending emails through it. Some accounts allow it when you explicitly allow "less secure apps" access to your Gmail account, but this isn't always available. You can read about this [here](#).

If you'd like to use a real email server, but don't want to complicate yourself with the Gmail configuration, [SendGrid](#) is a good option that gives you 100 emails per day using a free account.

Flask-Mail Usage

To learn how Flask-Mail works, I'll show you how to send an email from a Python shell session. 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 email 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](#)). 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.

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](#) 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 Res
</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 = db.session.scalar(
            sa.select(User).where(User.email == form.email.data))
        if user:
            send_password_reset_email(user)
            flash('Check your email for the instructions to reset your pas
            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. The `send_password_reset_email()` helper function performs this task. I will show you this function next.

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
'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhIjoib'J9.dv0o580BDHiuSHD4uW8
>>> 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 Flask configuration. The `algorithm` argument specifies how the token signature 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](#) 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 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')

    @staticmethod
    def verify_reset_password_token(token):
```

```
try:
    id = jwt.decode(token, app.config['SECRET_KEY'],
                    algorithms=['HS256'])['reset_password']
except:
    return
return db.session.get(User, id)
```

The `get_reset_password_token()` function returns a JWT token as a string, which is generated directly by the `jwt.encode()` function.

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 is 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

The `send_password_reset_email()` function relies on the `send_email()` function I wrote above to generate the password reset emails.

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.

To distinguish email templates from regular HTML templates, let's create a *email* subdirectory inside *templates*:

```
(venv) $ mkdir app/templates/email
```

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.

```
<!doctype html>
<html>
  <body>
    <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</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>
  </body>
</html>
```

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 that only include the path portion of the URL. This is normally sufficient for links that are generated in web pages, because the web browser completes the URL by taking the missing parts from the URL in the address bar. 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 log in.

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 debugging email server you may not have noticed this, but sending an email for real slows the application down considerably. All the interactions that need to happen when sending an email make the task slow, usually taking 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 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 yet, 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 Flask, 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.

Continue on to the [next chapter](#).

Become a Patron!

Hello, and thank you for visiting my blog! If you enjoyed this article, please consider supporting my work on this blog on [Patreon](#)!

 **BECOME A PATRON**

Share this post:

[Hacker News](#)[Reddit](#)[Twitter](#)[LinkedIn](#)[Facebook](#)[E-Mail](#)

31 comments



#1 **ArrayOfLilly** said a year ago

I think Google permits to send emails beside normal security settings, if you turned on the 2FA and generate a unique app password for this purpose. Unless you mean exactly that risk, but what was a long time ago that literally had to be set up this is gone.

I found: "As of 2022 May 22 Google removed the ability to allow less secure devices."

The app password feature is now found in the same "Signing in to Google" area, under the 2FA setting.



#2 **Neelarghya Kundu** said 10 months ago

Why do I get "The syntax of command is incorrect", when using set MAIL_USERNAME=<your-gmail-username> command ?



#3 **Miguel Grinberg** said 10 months ago

@Neelarghya: My guess is that the shell that you are using needs a different syntax to set an environment variable.



#4 **Punch** said 9 months ago

I'm trying to figure out how to boot off all active sessions when the user resets its password. I guess you could alter the user loader you provide to Flask-Login to use a changeable secondary ID, to invalidate the local remember me token upon password reset, but what if one of the sessions doesn't enable remember me? Flask-Login calls the user loader only to validate the

remember me cookie, doesn't it?

Any pointers are appreciated. Great blog by the way!



#5 Miguel Grinberg said 9 months ago

@Punch: what I would do: 1. Store an additional field in the user table with the timestamp of the last password change. 2. Store the timestamp of the log in in the user session. 3. In the user loader function, make sure that login timestamp \geq password change timestamp before returning the user.



#6 Punch said 9 months ago

Thanks! I guess I misunderstood the flask-login user loader decorator/callback, it does make calls to validate the user on the session cookie, too. Using a timestamp is working like a charm.

I was thinking about supporting multiple sessions in that scenario (sessions are only invalidated after pass reset) but I guess instead of the log in timestamp, I'd have to use an obfuscated/hashed pass-last-modified timestamp on the session cookie, so I could keep more than one session alive. Timestamp disclosure is bad in this scenario, and I'd probably have to use another secret key to hash that info (can't reuse werkzeug's generate_password_hash then).

Or maybe it would be time to use a real IdP through OAuth in that case. Hehe.



#7 Miguel Grinberg said 9 months ago

@Punch: What I proposed should allow multiple sessions to be active, as long as they were all created after the password change. Maybe I'm not fully understanding what you want to do, but I don't see why multiple sessions cannot coexist.



#8 Punch said 9 months ago

Actually I forgot I had timestamps be updated after every log in action, too. Classic mental lapse. It's working great with multiple sessions now :)



#9 Eldar said 9 months ago

At the end, I was able to send email only watching your video (#Flask 2 about contexts) and writing the code in application context;

```
from flask_mail import Message
from app import mail
with app.app_context():
    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)
```



#10 **varad** said 9 months ago

Hey I have to export all this environment variables every time I restart my vs code, is this is a bug or it's just have to do this way?

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



#11 **Miguel Grinberg** said 9 months ago

@varad: your shell provides options run code when a new shell is opened, including setting environment variables, so you can do this according to your shell, for example by setting your variables in a `.bashrc` or `.zshrc` file. Later in this tutorial you will also learn how to work with a `.env` file to store environment variables associated with your project.



#12 **Grant** said 9 months ago

Little odd, but when I tried to create the identical JWT token on a Python shell using identical commands, I got 'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhbjoiYiJ9.dvOo580BDHiuSHD4uW88nfJikhYAXc_sfUHq1mDi4G0' while this document produces 'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhbjoiYiJ9.dvOo580BDHiuSHD4uW88nfJik_sfUHq1mDi4G0'. Also, when I go to the linked JWT debugger, I get the former, and not the latter. IOW I cannot produce the same token you did as an example.



#13 Miguel Grinberg said 9 months ago

@Grant: The example JWT I show here is just that, an example. It is only useful so that you have an idea of what the JWT looks like, I do not claim that it is accurate and that you should get the exact same one. If you get a different one it does not matter, as long as it can be decoded!



#14 TheBlack said 9 months ago

Hello, in `app.models.User.verify_reset_password_token()` function (<https://github.com/miguelgrinberg/microblog/blob/a3cc2beff1ec8d49b99dab7fa0f4dd03b1614e4c/app/models.py#L105>) is it required to use "except:" (broad exception clause) instead of "except jwt.PyJWTError:"? Or we expect there any other exceptions than jwt module may raise? Thanks in advance.



#15 Miguel Grinberg said 9 months ago

@TheBlack: It isn't required to use a broad exception clause, it is just a choice. If you prefer to use a scoped exception clause that is fine, but you have to be prepared for your application to crash if there is any condition that you may not be aware of inside the PyJWT library (including bugs, present or in the future) that may raise a different exception. My preference is that in this case I want to be 100% sure that if PyJWT cannot process the token I handle the error and prevent unexpected crashes, but I recognize this is a personal preference.



#16 giuseppe said 7 months ago

Hi Miguel,
I'm still here asking for your help. I copied the password reset

code. The second terminal receive correctly the mail sent by the first one. In the received mail, as it should be, there are all the phrases introduced in the /templates/email/reset_password.txt file. There is also the link to copy and paste in the address bar of the browser.

If I copy this link into the browser no window for changing the password will open and you will remain on the Home page.

Is the signature invalidated? I enclose a small part of the output received. I hope it is not so much. Thank you very much.

p>\r\n

Alternatively, you can paste the following link in your browser's address bar:

\r\n

http://localhost:5000/reset_password/eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJyZXNldF9wYXNzd29yZCI6MSwiZXhwljoxNzEwNzU5NjI5LjE0NTEwNn0.01d8OIB73txGWzXlwapG5xKHtASXu mhVGt8EIQEW5s

\r\n

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

\r\n

Sincerely,

\r\n

The Microblog Team

\r\n </body>\r\n</html>\r\n-----1130241371---

\r\n\r\n-----1902386745---\r\n.\r\n'

reply: b'250 OK\r\n'

reply: retcode (250); Msg: b'OK'

data: (250, b'OK')

send: 'quit\r\n'

reply: b'221 Bye\r\n'

reply: retcode (221); Msg: b'Bye'

127.0.0.1 -- [18/Mar/2024 11:50:29] "GET /login HTTP/1.1" 200 -

127.0.0.1 -- [18/Mar/2024 11:52:41] "GET

/reset_password/eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJyZXNldF9wYXNzd29yZCI6MSwiZXhwljoxNzEwNzU5NjI5LjE0NTEwNn0.01d8OIB73txGWzXlwapG5xKHtASXu mhVGt8EIQEW5s< HTTP/1.1" 302 -

127.0.0.1 -- [18/Mar/2024 11:52:41] "GET /index HTTP/1.1" 302

-

127.0.0.1 -- [18/Mar/2024 11:52:41] "GET /login?next=/index HTTP/1.1" 200 -



#17 Miguel Grinberg said 7 months ago

@giuseppe: you should add some print statements in the code that validates the token to determine what's failing. The example email that you shared above looks fine, and the token seems fine too (but of course I'm unable to verify the signature, as that requires your secret key). Now you need to find out where in the code that runs when you enter the reset URL the token is rejected.



#18 Kun Minh said 7 months ago

Thank you for your detailed instructions, I have a question that I would like you to explain.

The core of the password change function is:

- 1 - Sending an email to the correct email address of the user who needs to change their password
- 2 - The user's information (User.id) is encrypted in the token, to make the link to reset password form

I am wondering why it is necessary to encrypt the User information?

Because if the email is sent correctly to the User who wants to change their password, then all they need is a link to access the password change form for that specific user. The link can be in the form of `/resetpassword/<user.id>`, which is sufficient without the need for encryption.

So what is the purpose of requiring a token? Is it to prevent attacks or is there any other reason?

Thank you.



#19 Miguel Grinberg said 7 months ago

@Kun: If the reset link has the user id without encoding, then the user that receives the link can edit the id and change the password of any user in the system. To prevent this, the user id is encoded in a token.



#20 righthippie said 6 months ago

Is any way to make web token works only one time? If token not expire I can click by link again and change password again. And

again till token will expire.



#21 Miguel Grinberg said 6 months ago

@righthippie: Yes. You can create a new database model of used tokens and each time a token is used add it as a new entry in this table. Then every time you receive a token you first search this table to make sure the token isn't already there. You can reduce the size of the table by running a background job that removes tokens that are already expired.



#22 pelorustechologies said 5 months ago

Fantastic tutorial! This guide on adding email support to Flask apps is detailed and user-friendly. Perfect for developers enhancing their projects.



#23 Rustam said 5 months ago

Hey Miguel,
email is sent via separate thread, so what happens if sending email fails?
how do we inform user about it, within the same request context?
we can't right? this is a disadvantage, our api endpoint becomes non atomic



#24 Miguel Grinberg said 5 months ago

@Rustam: When you send an email you cannot know if the email was delivered, it's just how they work. Email servers have complicated logic to retry asynchronously, so an immediate failure does not mean anything. Sending emails from a thread does not change that, you just have to send it and hope for the best.



#25 James Bunt said 3 months ago

Thanks a lot for this very complete tutorial that for me as a 'beginner' (started self-learning 3 years ago) in programming seems to be a good path to finally take the serious direction towards learning foundations about the use of databases and the creation of a web-project which I might be able to base on what I can learn here!

I am actually stuck in the email-part at the moment.

I managed the `send_email()` with the `aiosmtpd-mailserver`, but following along, I continuously get this error:

`ImportError: cannot import name 'mail' from partially initialized module 'app' (most likely due to a circular import) (app/init.py)`

I already moved `'forms.py'` and `'routes.py'` in a `'main'` directory and wrote the separate `'email.py'` for password reset in `'auth'`, as I saw you did on github, but still get the error.

Is this about a certain order of imports?

I like to list all imports on top of the document as it seems more organized to me, but should I do differently here?

I actually import like that:

`'/app/models.py':`

```
from datetime import datetime, timezone
from typing import Optional
import sqlalchemy as sa
import sqlalchemy.orm as so
from time import time
import jwt
from app import db, login, app
from flask_login import UserMixin
from werkzeug.security import generate_password_hash,
check_password_hash
```

`'/app/main/routes.py':`

```
from datetime import datetime, timezone
from flask import render_template, request, flash, redirect,
url_for
from flask_login import current_user, login_user, logout_user,
login_required
from urllib.parse import urlsplit
import sqlalchemy as sa
from app.forms import LoginForm, RegistrationForm,
EditProfileForm, EmptyForm, PostForm,
ResetPasswordRequestForm, ResetPasswordForm
from app.models import User, Post
from app import app, db
from app.email import send_password_reset_email
```

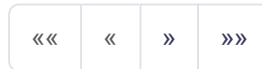
`'/app/email.py':`

```
from flask_mail import Message
from flask import render_template
```

```
<h1>from app import mail</h1>
```

I will continue trying ... suppose I did a wrong import somewhere.

In worse case I will just delete and start again ... learning by doing ;)





If you would you like to support my work on my [Flask Mega-Tutorial series](#) on this blog and as a reward have access to the complete tutorial nicely structured as a book and/or a set of videos, you can now order it from my [Courses](#) site or from [Amazon](#).

[Click here to get the Book!](#)

[Click here to get the Video Course!](#)

About Miguel

Welcome to my blog!








I'm a software engineer and technical writer, currently living in Drogheda, Ireland.






























You can also find me on [Twitter](#), [Mastodon](#), [Github](#), [LinkedIn](#), [YouTube](#), [Facebook](#) and [Patreon](#).

Thank you for visiting!

Categories

-  [AI](#) 2
-  [AWS](#) 1
-  [Arduino](#) 7
-  [Authentication](#) 10
-  [Blog](#) 1
-  [C++](#) 5
-  [CSS](#) 1
-  [Cloud](#) 10
-  [Database](#) 22
-  [Docker](#) 5

	Filmmaking	6
	Flask	126
	Games	1
	Heroku	1
	IoT	8
	JavaScript	35
	MicroPython	9
	Microdot	1
	Microservices	2
	Movie Reviews	5
	OpenStack	1
	Personal	3
	Photography	7
	Product Reviews	2
	Programming	187
	Project Management	1
	Python	169
	REST	7
	Rackspace	1
	Raspberry Pi	8
	React	18
	Robotics	6
	Security	12
	Video	22
	WebSocket	2
	Webcast	3
	Windows	1

© 2012-2024 by Miguel Grinberg. All rights reserved. [Questions?](#)