CODING NOMADS

13) Sending Emails & Email Verification Lesson

Generate Email Confirmation Tokens and Confirm Users with Flask

28 min to complete · By Brandon Gigous

Contents

With your email support established, now it's time to generate tokens for confirming your users. This is a common practice among web apps, as it prevents spam accounts from piling up. It also makes sure your users have access to the emails they provide for validation. The tokens you'll create specific for your app will be sent to users, who will then click the links that embed them, and their accounts will be confirmed.

Account Confirmation with Tokens

There's the wrong way to generate **tokens**, and then there's the right way. These unique tokens are generated by the app, for a user who has just registered. Their purpose is to make sure the user *got* that token in their email within a set amount of time. It comes in the form of a link, so when the user navigates to that webpage, the app compares that token with the current user's information. For example, the link would be something like `https://www.example.com/auth/confirm/<token_id>`.

The wrong way to generate tokens is to use the user's ID as the token. Once the app gets that link, it knows the user with that ID confirmed their account, right? Ideally, yes, but in practice you can't guarantee that. There are all sorts of bad actors and even those dang kids that never get off your lawn! People like that could exploit this bad design to confirm all sorts of arbitrary accounts, and all they have to do is punch in some lottery numbers.

The right way to generate tokens is to take humans out of the picture. Only the app should worry about tokens and who's validated their account or not. While humans are clever, computers *can* outsmart them. At least, this is true of token generation. The way this is done is to give only the app the ability to generate *valid* tokens, which enlists the help of your friend, *cryptography*.

Remember when you created forms and you needed a secret key for Flask-WTF to do cryptographic stuff? It is to protect the user session so that user information is safe. The user session cookies contain a crytpographic signature, and if any of the content of the user session is modified, the signature becomes invalid and Flask discards the session to make a new one.

A similar method can be used for confirmation tokens using the `pyjwt` package, in that tokens can be generated *for a specific user* in order to confirm their registration. If this token expires, or if another user tries using the same token, your app would see it as invalid. At the end of the day, you are using cryptography to keep your app secure!

To get it, install it through pip:

```
(env) $ pip install pyjwt
```

Generating Tokens With The `pyjwt` Package

What's more fun than messing around with cryptography in a Flask shell session? Well, maybe tire swings, but nothing *else* comes close.

Pull up a Flask shell session and follow along. What you'll do is generate a signed token. This token will look to a human like a bunch of cryptographic magic, or perhaps complete nonsense. However, it will keep a user ID "hidden" inside, which can only be unhidden if certain conditions are met. Go ahead, try it out:

```
(env) $ flask shell
>>> import jwt
>>> from flask import current_app
>>> from datetime import datetime, timedelta
>>> # create a datetime object that represents "60 minutes from righ
>>> expiration_datetime = datetime.utcnow() + timedelta(seconds=3600
>>> # "package" the data in a JSON-like format that will be tokenize
>>> # the 'exp' key lets pyjwt know when the token becomes invalid
>>> # the 'confirm_id' key is arbitrary but represents a user's id
>>> data = {"exp": expiration_datetime, "confirm_id": 14}
>>> # tokenize! pass in the secret key to sign the data
>>> token = jwt.encode(data, current_app.secret_key, algorithm="HS51
>>> # here's what the token might look like
>>> token
'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzUxMiJ9.eyJleHAiOjE2NTUzNTc1ODEsImNvbm
>>> # ... then sometime later (but before 60 minutes have passed)
>>> data = jwt.decode(token, current_app.secret_key, algorithms=["HS
>>> data
{'exp': 1655357581, 'confirm_id': 14}
```

While this package can come up with some interesting made-up words, like "zoq" or "sLeRth", that's not what you're here for. And no, it doesn't *actually* make up words on purpose, though it *is* fun to look for them. The `pyjwt` package provides you with lots of ways to generate tokens using different algorithms. The one picked here, HS512, is a pretty standard one used in the industry.

Unhiding Data

First, your app's secret key, as `current_app.secret_key` in the example above, is used as the encryption key for signing the confirmation token. These JSON Web Signatures (JWSs) are a state-of-the-art way of ensuring that data doesn't change from what the data *was* the moment it was signed. It ensures the integrity of the data. If it does change, the signature won't match up and is declared invalid data. Putting it more simply, you can think of "signing" data as analogous to signatures on a paper document, where if one signature doesn't match another, it could be a forgery. Similarly, if your app is the signer on a piece of data, its signature is its secret key. Using anything other than the app's secret key to try "unhiding" the data would be detected as a "forgery" and declared invalid.

Second, the time expiration ensures that the token is validated within a certain time period. In the example above, the `exp` key is a datetime 60 minutes in the future that `pyjwt` compares with the time it does the decoding. The data becomes invalid if it expires like that half-gallon of milk in the back of Uncle Steve's refrigerator.

So, in summary: The token, or the cryptographic gobbledygook that hides the confirmation code, gets generated using the `encode()` method of the serializer. Then, to get the data back later, you provide the special token to the `decode()` method of the serializer. If the token is invalid or the time expiration has passed, the `loads()` method raises an exception.

Update The User Model

The next step is to prep your `User` model for holding a confirmation status. You'll add a new column called `confirmed` that's a boolean value. You'll be able to check this later for each user to make sure they can access certain parts of the web app. Similar to the code example above, you'll also want to implement the functionality to generate their special token based on their user ID, and then confirm that same user later with the token that was generated:

```
from flask import current_app
import jwt

class User(UserMixin, db.Model):
    # ...
    # NEW column, true if user confirmed, false otherwise
    confirmed = db.Column(db.Boolean, default=False)

    # ...

    def generate_confirmation_token(self, expiration_sec=3600):
        # For jwt.encode(), expiration is provided as a time in UTC
        # It is set through the "exp" key in the data to be tokenize
        expiration_time = datetime.utcnow() + timedelta(seconds=expi
        data = {"exp": expiration_time, "confirm_id": self.id}
        # Use SHA-512 (known as HS512) for the hash algorithm
        token = jwt.encode(data, current_app.secret_key, algorithm="
        return token

    def confirm(self, token):
        try:
            # Ensure token valid and hasn't expired
            data = jwt.decode(token, current_app.secret_key, algorit
        except jwt.ExpiredSignatureError as e:
            # token expired
            return False
        except jwt.InvalidSignatureError as e:
            # key does not match
            return False
        # The token's data must match the user's ID
        if data.get("confirm_id") != self.id:
            return False
        # All checks pass, confirm the user
        self.confirmed = True
        db.session.add(self)
        # the data isn't committed yet as you want to make sure the
        return True
```

The `generate_confirmation_token()` does just as you did a minute ago and generates a token, but the data it's signing is *this* user's ID. And as before, an optional `exp` key in the data package is provided to control the time expiration, which is one hour.

Next is the `confirm()` method, which takes a token and checks whether or not it is valid for that user. It will check

1. that the token itself is valid,
2. that the token hasn't expired, and
3. that the data hidden within matches the user's ID

If any of those are false, the method returns `False` . If all is fine and dandy, the `User` is confirmed and updated within the database. However, the database session is *not yet committed*. You need to make sure the confirmed user is currently logged in. This is a security measure, as you wouldn't want to confirm a user who can't remember their password, and therefore, can't log in.

Unit Tests

Learn by Doing

Write a unit test that generates a confirmation token for a user and that tests whether or not the user in question can be confirmed successfully.

In what ways can the confirmation succeed or fail? Ensure your test(s) capture each scenario.

Fantastic. You've gotten through generating tokens! This is an important step to ensure your app is as secure as possible. In the next lesson, your users will need to confirm their accounts to continue with using your web app. It's a standard practice in web apps, and for good reason: security for both you and your users.

But hang on, right now they don't actually *need* to confirm their accounts yet... It's time to fix that.

Confirm Your Users

You can generate tokens, but to put those tokens to use requires some additional code. In this lesson, you'll make the changes necessary to send new users an email with a token, require them to verify their email before using the rest of the site, and allow them to resend confirmation emails.

The Confirmation Link

What exactly is the registration confirmation link? All the view functions you've seen thus far have something for the user to, well, *view* through the rendering of a template. The index page, where you welcome your users, plenty of view functions for displaying and processing forms, and basic user profile pages. But what is there to *view* in a confirmation page? A page that says, "Yo, this is the confirmation page, please enjoy this picture of an ostrich while we process your confirmation?"

Keep in mind your view functions don't have to show anything at all. At the end of the day, all they are doing is handling a request. A *request* by the user to confirm their account can be processed by your app without showing any ostriches, and then *redirect* the user to another page once the processing is complete.

So now that this fact is drilled into your brain, let's try it out. The `confirm()` view function in `app/auth/views.py` will take a token, attempt to confirm the user, then redirect to the home index page:

```
@auth.route('/confirm/<token>')
@login_required
def confirm(token):
    if current_user.confirmed:
        flash("You're already confirmed, silly!")
        return redirect(url_for('main.index'))
    if current_user.confirm(token):
        db.session.commit()
        flash('You have confirmed your account! Thank you.')
    else:
        flash("Whoops! That confirmation link either expired, or it
    return redirect(url_for('main.index'))
```

Note that you will want to see if the *current* user, `current_user` is confirmed, because we don't want this function to confirm any old user. And remember, the token cannot be decoded if the token doesn't match the user session signature.

Starting from the beginning, you wouldn't want to reconfirm a user if they are confirmed already. A redirect is performed early in that case. But if not, the token is processed and if it goes through, *the database session to add the user is then committed*. In the last lesson, the `confirm()` `User` method only updated the user within the session, so this step writes it to the database for real. An invalid or expired token will still ultimately result in a redirect.

Register. Receive Email. Confirm.

With your new token generation and email sending capabilities, the time has come to combine them! Pop open your `register()` view function once again.

You'll again make sure the user registration information is valid, then you'll `generate_confirmation_token()` for that user. Once you create a `token`, you'll `send_email()` with a template to create the confirmation link inside the email. Read those sentences again if it didn't quite make sense. :)

Just like your welcome and new user notification emails, you're mission is to create that template. It's to nicely ask your users to pretty please confirm their account. Like last time, make a text template and an HTML template.

Unlike the last time, you'll put your templates in the `templates/auth/email` directory, and call them `confirm.txt` / `html`. Technically, this email will be *part* of authenticating your users, so that's the logic behind it.

And one thing you should know: in your templates, you will need to use the `url_for()` Flask method to point to the new `confirm()` view function you made above, and pass in the `token` that way.

The `url_for()` Flask method accepts a keyword argument `_external`, which is a boolean. Set this to `True` to so that the URL is *fully qualified*, meaning it includes `https://` or `http://`, the hostname, and the port. Otherwise, your user will be pointed to a confirmation link that doesn't work!

User Authentication

The point of this series of paragraphs is to teach you how to *block* users from accessing any other page until their email is confirmed. You *want* to let them in, but rules are rules. You *could* just tack on a `login_required` decorator to each and every view function, but what if you create a new view function and forget to apply the decorator to that one? Why even worry about it?

That's kind of what the `before_app_request` decorator was made for. This is somewhat similar to an `app_errorhandler` decorator it is a *special* kind of route in that the function it is applied to doesn't need a separate `route` decorator. It also applies app-wide. Now onto what it actually does. As can be implied from the name, the function decorated with `before_app_request` is called *before* every request. That means *before* any view function is called. If the before-request handler returns a response or redirect, it will intercept any view function that was to originally going to be invoked.

You can tack the `before_app_request` callback onto a new function in the `app/auth/views.py` file to prevent unconfirmed users from accessing another page and instead present them with an "please confirm first!" page.

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


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

There are several conditions that will trigger the user to be redirected to this new `'/unconfirmed'` route.

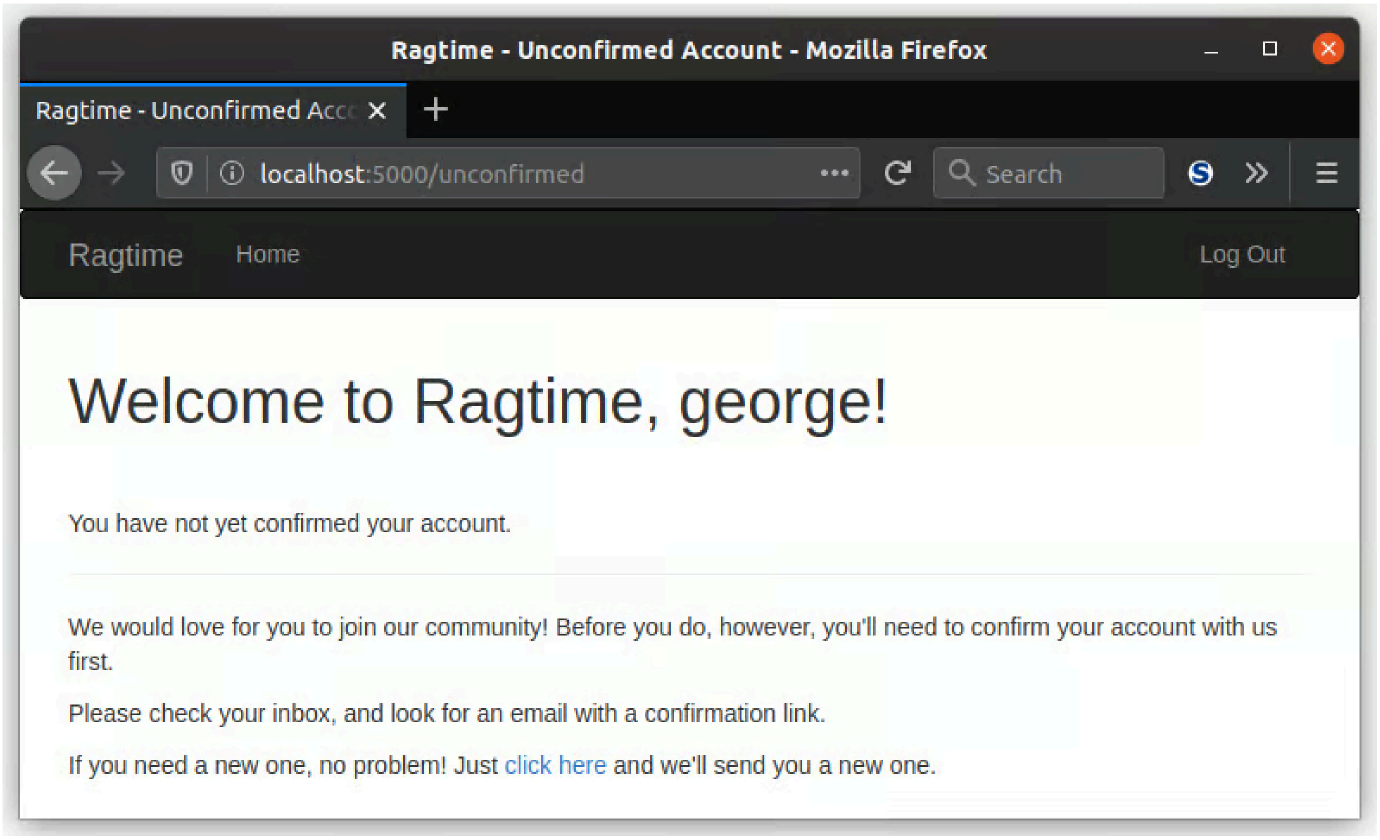
1. The user must be logged in, as in `current_user.is_authenticated` is `True`
2. The user has not confirmed their email
3. The blueprint that handles the original request is not the `auth` blueprint (you'll still want to allow them to confirm their account!)
4. The request is not for a static file (in `app/static`)

Then the `unconfirmed()` view function handles showing the user the "please verify!" template, unless they happen to not be logged in or they already confirmed their account.

Get Mentorship

"Your Account Is Unconfirmed" Template

As you may have guessed, you'll need to create yet another template that asks the user they can't use the site until they verify their account. This one is easy, though. Just let them know they still need to verify their account. But don't forget to let them have the app resend a new verification code! Call this template `unconfirmed.html` as in the above view function.


Ragtime - Unconfirmed Account - Mozilla Firefox
Ragtime - Unconfirmed Acc
localhost:5000/unconfirmed
Search
Ragtime
Home
Log Out

Welcome to Ragtime, george!

You have not yet confirmed your account.

We would love for you to join our community! Before you do, however, you'll need to confirm your account with us first.

Please check your inbox, and look for an email with a confirmation link.

If you need a new one, no problem! Just click here and we'll send you a new one.

Coolio, now you should be all set to welcome and then ask your users to confirm their new accounts! But while testing this, haven't you noticed your app get a teensy bit slower? Find out why in the next lesson.

Summary: Email Confirmation Tokens

In this lesson you've:

- Understood the concept of confirmation tokens and why they're a better choice compared to using user IDs.

- Installed the `pyjwt` package to handle cryptographic token generation and confirmation.
- Practiced creating tokens using Flask's shell and `pyjwt`, embedding user-specific information such as user IDs, and managing token expiration times.
- Enhanced the `User` model to include a 'confirmed' field and methods for generating confirmation tokens and confirming the user based on a valid token.
- Developed unit tests to validate the functionality of the token generation and user confirmation process.
- Implemented confirmation link handling in Flask by creating a route that uses the generated token to validate and confirm the user's email.
- Set up user authentication flow, restricting access to the application until the user's email has been confirmed using Flask's `before_app_request` to intercept requests.
- Crafted user interface templates for the confirmation process.

Now, with these functionalities in place, your Flask application has a robust user authentication system that validates user emails through a secure confirmation process.

Previous Next → Video: Tokens and Confirmation Emails