The Flask Mega-Tutorial, Part V: User Logins

Posted by Miguel Grinberg *on* December 3, 2023 *under* Programming Security Python Flask

This is the fifth installment of the Flask Mega-Tutorial series, in which I'm going to tell you how to create a user login subsystem.

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:

In Chapter 3 you learned how to create the user login form, and in Chapter 4 you learned how to work with a database. This chapter will teach you how to combine the topics from those two chapters to create a simple user login system.

The GitHub links for this chapter are: Browse, Zip, Diff.

Password Hashing

In Chapter 4 the user model was given a `password_hash` field, that so far is unused. The purpose of this field is to hold a hash of the user password, which will be used to verify the password entered by the user during the log in process. Password hashing is a complicated topic that should be left to security experts, but there are several easy to use libraries that implement all that logic in a way that is simple to be invoked from an application.

One of the packages that implement password hashing is Werkzeug, which you may have seen referenced in the output of pip when you install Flask, since it is one of its core dependencies. Since it is a dependency, Werkzeug is already installed in your virtual environment. The following Python shell session demonstrates how to hash a password with this package:

```
>>> from werkzeug.security import generate_password_hash
>>> hash = generate_password_hash('foobar')
>>> hash
'scrypt:32768:8:1$DdbIPADqKg2nniws$4ab051ebb6767a...'
```

In this example, the password `foobar` is transformed into a long encoded string through a series of cryptographic operations that have no known reverse operation, which means that a person that obtains the hashed password will be unable to use it to recover the original password. As an additional measure, if you hash the same password multiple times, you will get different results, since all hashed passwords get a different cryptographic *salt*, so this makes it impossible to identify if two users have the same password by looking at their hashes.

The verification process is done with a second function from Werkzeug, as follows:

```
>>> from werkzeug.security import check_password_hash
>>> check_password_hash(hash, 'foobar')
True
>>> check_password_hash(hash, 'barfoo')
False
```

The verification function takes a password hash that was previously generated, and a password entered by the user at the time of log in. The function returns `True` if the password provided by the user matches the hash, or `False` otherwise.

The whole password hashing logic can be implemented as two new methods in the user model:

app/models.py: Password hashing and verification

```
from werkzeug.security import generate_password_hash, check_password_h

# ...

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

    def set_password(self, password):
        self.password_hash = generate_password_hash(password)

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

With these two methods in place, a user object is now able to do secure password verification, without the need to ever store original passwords. Here is an example usage of these new methods:

```
>>> u = User(username='susan', email='susan@example.com')
>>> u.set_password('mypassword')
>>> u.check_password('anotherpassword')
False
>>> u.check_password('mypassword')
True
```

Introduction to Flask-Login

In this chapter I'm going to introduce you to a very popular Flask extension called Flask-Login. This extension manages the user logged-in state, so that for example users can log in to the application and then navigate to different pages while the application "remembers" that the user is logged in. It also provides the "remember me" functionality that allows users to remain logged in even after closing the browser window. To be ready for this chapter, you can start by installing Flask-Login in your virtual environment:

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

As with other extensions, Flask-Login needs to be created and initialized right after the application instance in *app/__init__.py*. This is how this extension is initialized:

app/__init__.py: Flask-Login initialization

```
# ...
from flask_login import LoginManager

app = Flask(__name__)
# ...
login = LoginManager(app)

# ...
```

Preparing The User Model for Flask-Login

The Flask-Login extension works with the application's user model, and expects certain properties and methods to be implemented in it. This approach is nice, because as long as these required items are added to the model, Flask-Login does not have any other requirements, so for example, it can work with user models that are based on any database system.

The four required items are listed below:

- `is_authenticated`: a property that is `True` if the user has valid credentials or `False` otherwise.
- `is_active`: a property that is `True` if the user's account is active or `False` otherwise.
- `is_anonymous`: a property that is `False` for regular users, and `True` only for a special, anonymous user.
- `get_id()`: a method that returns a unique identifier for the user as a string.

I can implement these four easily, but since the implementations are fairly generic, Flask-Login provides a *mixin* class called `UserMixin` that includes safe implementations that are appropriate for most user model classes. Here is how the mixin class is added to the model:

app/models.py: Flask-Login user mixin class

```
# ...
from flask_login import UserMixin

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

User Loader Function

Flask-Login keeps track of the logged in user by storing its unique identifier in Flask's *user session*, a storage space assigned to each user who connects to the application. Each time the logged-in user navigates to a new page, Flask-Login retrieves the ID of the user from the session, and then loads that user into memory.

Because Flask-Login knows nothing about databases, it needs the application's help in loading a user. For that reason, the extension expects that the application will configure a user loader function, that can be called to load a user given the ID. This function can be added in the *app/models.py* module:

app/models.py: Flask-Login user loader function

```
from app import login
# ...

@login.user_loader
def load_user(id):
    return db.session.get(User, int(id))
```

The user loader is registered with Flask-Login with the `@login.user_loader` decorator. The `id` that Flask-Login passes to the function as an argument is going to be a string, so databases that use numeric IDs need to convert the string to integer as you see above.

Logging Users In

Let's revisit the login view function, which as you recall, implemented a fake login that just issued a `flash()` message. Now that the application has access to a user database and knows how to generate and verify password hashes, this view function can be completed.

app/routes.py: Login view function logic

```
# ...
from flask_login import current_user, login_user
import sqlalchemy as sa
from app import db
from app.models import User

# ...

@app.route('/login', methods=['GET', 'POST'])
def login():
    if current_user.is_authenticated:
        return redirect(url_for('index'))
    form = LoginForm()
    if form.validate_on_submit():
        user = db.session.scalar(
            sa.select(User).where(User.username == form.username.data)
        if user is None or not user.check_password(form.password.data)
            flash('Invalid username or password')
            return redirect(url_for('login'))
        login_user(user, remember=form.remember_me.data)
        return redirect(url_for('index'))
    return render_template('login.html', title='Sign In', form=form)
```

The top two lines in the `login()` function deal with a weird situation. Imagine you have a user that is logged in, and the user navigates to the */login* URL of your application. Clearly that is a mistake, so I want to not allow that. The `current_user` variable comes from Flask-Login, and can be used at any time during the handling of a request to obtain the user object that represents the client of that request. The value of this variable can be a user object from the database (which Flask-Login reads through the user loader callback I provided above), or a special anonymous user object if the user did not log in yet. Remember those properties that Flask-Login required in the user object? One of those was `is_authenticated`, which comes in handy to check if the user is logged in or not. When the user is already logged in, I just redirect to the index page.

In place of the `flash()` call that I used earlier, now I can log the user in for real. The first step is to load the user from the database. The username came with the form submission, so I can query the database with that to find the user. For this purpose I'm using the `where()` clause, to find users with the given username. Since I know there is only going to be one or zero results, I execute the query by calling `db.session.scalar()`, which will return the user object if it exists, or `None` if it does not. In Chapter 4 you have seen that when you call the `all()` method the query executes and you get a list of all the results that match that query. The `first()` method is another commonly used way to execute a query, when you only need to have one result.

If I got a match for the username that was provided, I can next check if the password that also came with the form is valid. This is done by invoking the `check_password()` method I defined above. This will take the password hash stored with the user and determine if the password entered in the form matches the hash or not. So now I have two possible error conditions: the username can be invalid, or the password can be incorrect for the user. In either of those cases, I flash a message, and redirect back to the login prompt so that the user can try again.

If the username and password are both correct, then I call the `login_user()` function, which comes from Flask-Login. This function will register the user as logged in, so that means that any future pages the user navigates to will have the `current_user` variable set to that user.

To complete the login process, I just redirect the newly logged-in user to the index page.

Logging Users Out

I know I will also need to offer users the option to log out of the application. This can be done with Flask-Login's `logout_user()` function. Here is the logout view function:

app/routes.py: Logout view function

```
# ...
from flask_login import logout_user

# ...
```

```
@app.route('/logout')
def logout():
    logout_user()
    return redirect(url_for('index'))
```

To expose this link to users, I can make the Login link in the navigation bar automatically switch to a Logout link after the user logs in. This can be done with a conditional in the *base.html* template:

app/templates/base.html: Conditional login and logout links

```
        <div>
            Microblog:
            <a href="{{ url_for('index') }}">Home</a>
            {% if current_user.is_anonymous %}
            <a href="{{ url_for('login') }}">Login</a>
            {% else %}
            <a href="{{ url_for('logout') }}">Logout</a>
            {% endif %}
        </div>
```

The `is_anonymous` property is one of the attributes that Flask-Login adds to user objects through the `UserMixin` class. The `current_user.is_anonymous` expression is going to be `True` only when the user is not logged in.

Requiring Users To Login

Flask-Login provides a very useful feature that forces users to log in before they can view certain pages of the application. If a user who is not logged in tries to view a protected page, Flask-Login will automatically redirect the user to the login form, and only redirect back to the page the user wanted to view after the login process is complete.

For this feature to be implemented, Flask-Login needs to know what is the view function that handles logins. This can be added in *app/__init__.py*:

```
# ...
login = LoginManager(app)
login.login_view = 'login'
```

The `'login'` value above is the function (or endpoint) name for the login view. In other words, the name you would use in a `url_for()` call to get the URL.

The way Flask-Login protects a view function against anonymous users is with a decorator called `@login_required`. When you add this decorator to a view function below the `@app.route` decorator from Flask, the function becomes protected and will not allow access to users that are not authenticated. Here is how the decorator can be applied to the index view function of the application:

app/routes.py: @login_required decorator

```
from flask_login import login_required

@app.route('/')
@app.route('/index')
@login_required
def index():
    # ...
```

What remains is to implement the redirect back from the successful login to the page the user wanted to access. When a user that is not logged in accesses a view function protected with the `@login_required` decorator, the decorator is going to redirect to the login page, but it is going to include some extra information in this redirect so that the application can then return to the original page. If the user navigates to */index*, for example, the `@login_required` decorator will intercept the request and respond with a redirect to */login*, but it will add a query string argument to this URL, making the complete redirect URL */login?next=/index*. The `next` query string argument is set to the original URL, so the application can use that to redirect back after login.

Here is a snippet of code that shows how to read and process the `next` query string argument. The changes are in the four lines below the `login_user()` call.

app/routes.py: Redirect to \"next\" page

```
from flask import request
from urllib.parse import urlsplit

@app.route('/login', methods=['GET', 'POST'])
def login():
    # ...
    if form.validate_on_submit():
        user = db.session.scalar(
            sa.select(User).where(User.username == form.username.data)
        if user is None or not user.check_password(form.password.data)
            flash('Invalid username or password')
            return redirect(url_for('login'))
        login_user(user, remember=form.remember_me.data)
        next_page = request.args.get('next')
        if not next_page or urlsplit(next_page).netloc != '':
            next_page = url_for('index')
        return redirect(next_page)
    # ...
```

Right after the user is logged in by calling Flask-Login's `login_user()` function, the value of the `next` query string argument is obtained. Flask provides a `request` variable that contains all the information that the client sent with the request. In particular, the `request.args` attribute exposes the contents of the query string in a friendly dictionary format. There are actually three possible cases that need to be considered to determine where to redirect to after a successful login:

- If the login URL does not have a `next` argument, then the user is redirected to the index page.
- If the login URL includes a `next` argument that is set to a relative path (or in other words, a URL without the domain portion), then the user is redirected to that URL.
- If the login URL includes a `next` argument that is set to a full URL that includes a domain name, then this URL is ignored, and the user is redirected to the index page.

The first and second cases are self-explanatory. The third case is in place to make the application more secure. An attacker could insert a URL to a malicious site in the `next` argument, so the application only redirects when the URL is relative, which ensures that the redirect stays within the same site as the application. To determine if the URL is absolute or relative, I parse it with Python's `urlsplit()` function and then check if the `netloc` component is set or not.

Showing The Logged-In User in Templates

Do you recall that way back in Chapter 2 I created a fake user to help me design the home page of the application before the user subsystem was in place? Well, the application has real users now, so I can now remove the fake user and start working with real users. Instead of the fake user I can use Flask-Login's `current_user` in the template *index.html* template:

app/templates/index.html: Pass current user to template

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

{% block content %}
    <h1>Hi, {{ current_user.username }}!</h1>
    {% for post in posts %}
    <div><p>{{ post.author.username }} says: <b>{{ post.body }}</b></p
    {% endfor %}
{% endblock %}
```

And I can remove the `user` template argument in the view function:

app/routes.py: Do not pass user to template anymore

```
@app.route('/')
@app.route('/index')
@login_required
def index():
    # ...
    return render_template("index.html", title='Home Page', posts=post
```

This is a good time to test how the login and logout functionality works. Since there is still no user registration, the only way to add a user to the database is to do it via the Python shell, so run `flask shell` and enter the following commands to register a user:

```
>>> u = User(username='susan', email='susan@example.com')
>>> u.set_password('cat')
>>> db.session.add(u)
>>> db.session.commit()
```

If you now start the application and go to the application's */* or */index* URLs, you will be immediately redirected to the login page, and after you log in using the credentials of the user that you added to your database, you will be returned to the original page, in which you will see a personalized greeting and the mock blog posts. If you then click the logout link in the top navigation bar, you will be sent back to the index page as an anonymous user, and immediately redirected to the login page again by Flask-Login.

User Registration

The last piece of functionality that I'm going to build in this chapter is a registration form, so that users can register themselves through a web form. Let's begin by creating the web form class in *app/forms.py*:

app/forms.py: User registration form

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitFi
from wtforms.validators import ValidationError, DataRequired, Email, E
import sqlalchemy as sa
from app import db
from app.models import User

# ...

class RegistrationForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    email = StringField('Email', validators=[DataRequired(), Email()])
    password = PasswordField('Password', validators=[DataRequired()])
    password2 = PasswordField(
        'Repeat Password', validators=[DataRequired(), EqualTo('passwo
    submit = SubmitField('Register')

    def validate_username(self, username):
        user = db.session.scalar(sa.select(User).where(
            User.username == username.data))
        if user is not None:
            raise ValidationError('Please use a different username.')

    def validate_email(self, email):
        user = db.session.scalar(sa.select(User).where(
            User.email == email.data))
        if user is not None:
            raise ValidationError('Please use a different email addres
```

There are a couple of interesting things in this new form related to validation. First, for the `email` field I've added a second validator after `DataRequired`, called `Email`. This is another stock validator that comes with WTForms that will ensure that what the user types in this field matches the structure of an email address.

The `Email()` validator from WTForms requires an external dependency to be installed:

```
(venv) $ pip install email-validator
```

Since this is a registration form, it is customary to ask the user to type the password two times to reduce the risk of a typo. For that reason I have `password` and `password2` fields. The second password field uses yet another stock validator called `EqualTo`, which will make sure that its value is identical to the one for the first password field.

When you add any methods that match the pattern `validate_<field_name>`, WTForms takes those as custom validators and invokes them in addition to the stock validators. I have added two of those methods to this class for the `username` and `email` fields. In this case I want to make sure that the username and email address entered by the user are not already in the database, so these two methods issue database queries expecting there will be no results. In the event a result exists, a validation error is triggered by raising an exception of type `ValidationError`. The message included as the argument in the exception will be the message that will be displayed next to the field for the user to see.

Note how the two validation queries are issued. These queries will never find more than one result, so instead of running them with `db.session.scalars()` I'm using `db.session.scalar()` in singular, which returns `None` if there are no results, or else the first result.

To display this form on a web page, I need to have an HTML template, which I'm going to store in file *app/templates/register.html*. This template is constructed similarly to the one for the login form:

app/templates/register.html: Registration template

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

{% block content %}
    <h1>Register</h1>
    <form action="" method="post">
        {{ form.hidden_tag() }}
        <p>
            {{ form.username.label }}<br>
            {{ form.username(size=32) }}<br>
            {% for error in form.username.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <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.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 login form template needs a link that sends new users to the registration form, right below the form:

app/templates/login.html: Link to registration page

```
    <p>New User? <a href="{{ url_for('register') }}">Click to Register
```

And finally, I need to write the view function that is going to handle user registrations in *app/routes.py*:

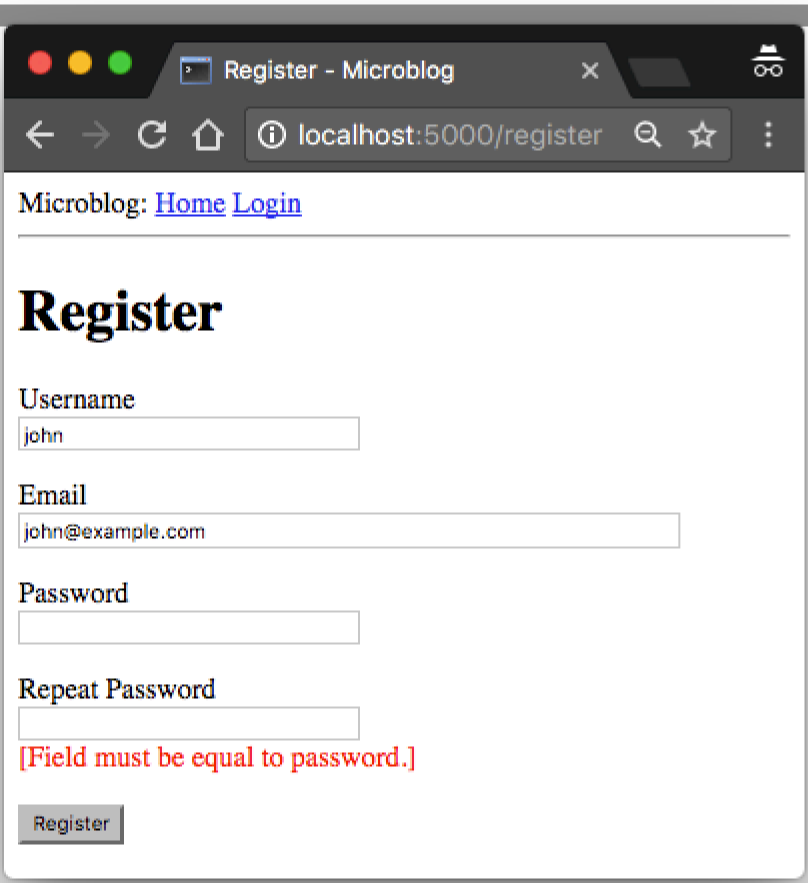
app/routes.py: User registration view function

```
from app import db
from app.forms import RegistrationForm

# ...

@app.route('/register', methods=['GET', 'POST'])
def register():
    if current_user.is_authenticated:
        return redirect(url_for('index'))
    form = RegistrationForm()
    if form.validate_on_submit():
        user = User(username=form.username.data, email=form.email.data
        user.set_password(form.password.data)
        db.session.add(user)
        db.session.commit()
        flash('Congratulations, you are now a registered user!')
        return redirect(url_for('login'))
    return render_template('register.html', title='Register', form=for
```

And this view function should also be mostly self-explanatory. I first make sure the user that invokes this route is not logged in. The form is handled in the same way as the one for logging in. The logic that is done inside the `if validate_on_submit()` conditional creates a new user with the username, email and password provided, writes it to the database, and then redirects to the login prompt so that the user can log in.



Microblog: Home Login

Register

Username
john

Email
john@example.com

Password

Repeat Password

[Field must be equal to password.]

Register

With these changes, users should be able to create accounts on this application, and log in and out. Make sure you try all the validation features I've added in the registration form to better understand how they work. I am going to revisit the user authentication subsystem in a future chapter to add additional functionality such as to allow the user to reset the password if forgotten. But for now, this is enough to continue building other areas of the application.

Continue on to the next chapter.

Buy me a coffee?

Thank you for visiting my blog! If you enjoyed this article, please consider supporting my work and keeping me caffeinated with a small one-time donation through Buy me a coffee. Thanks!

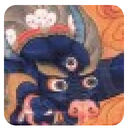
Buy me a coffee

Share this post

Hacker News Reddit Twitter LinkedIn

Facebook E-Mail

63 comments



#1 **Ben** said a year ago

This is really good stuff - but, again, it would be awesome to see how best to implement an integrated RBAC (or ABAC, or any *BAC) especially permissions. My specific interest is in the design of an MSA such that each separate service architecture offers a set of roles – which are then coordinated by the user service. Eg, I may be a forum 'administrator' for a selection of discussions, but merely have view access for all media banks. The ability to switch between instance permissions (via eg, categories, discussions, or other arbitrary groups) against a generic set of permissions offers great flexibility, but I struggle to see how best to implement that. It may be that I am conflating authentication with authorisation - but the latter depends strongly on the former, and when we leave behind a monolithic architecture, I'm curious about how best to manage cross-architecture access privileges.

#2 Miguel Grinberg said a year ago

@Ben: Once you learn the basic authentication patterns such as the one I present in this tutorial, you should be able to expand it to build your own. Or maybe find a Flask extension that does what you need.

#3 **Andy** said a year ago

@Miguel Grinberg How come we don't have to do
with app.app_context():
...
when interacting with the table? I've seen other tutorials and documentation use this way of creating a db. Does Flask-migrate do that for us?

#4 Miguel Grinberg said a year ago

@Andy: Flask-Migrate activates a context when it needs one.

#5 **Richard** said a year ago

Why did you change from using werkzeug.urls.url_parse(next_page).netloc in the 2018 tutorial to using urllib.parse.urlsplit(next_page).netloc in this tutorial?

#6 Miguel Grinberg said a year ago

@Richard: there is no url_parse function in Werkzeug anymore. They removed it.

#7 **Tom** said a year ago

I notice you do validation of the login form (does the user exist?) in routes.py but you do validation of the registration form (does the user exist?) in forms.py. Curious as to your reasoning behind this design decision.

#8 Miguel Grinberg said a year ago

@Tom: A login form does not normally do field by field validation. You accept the username and password, and if the authentication fails you just tell this to the user, without describing why the login failed. For other forms you typically want to have individual field validation.

#9 **Eldar** said a year ago

Great Miguel! Thanks for your tutorial, I learn a lot from it

#10 **Tom** said a year ago

I've been so impressed with this tutorial that I bought the Amazon Kindle version to support you. I've also purchased the SQLAlchemy 2.0 one; which is amazing. The way you explain each block of code in full, allows me to not just understand how to do something, but go deeper and understand the why. Thanks so much for the amazing quality of your work!

#11 **Ross** said a year ago

I have a question about the register() function at the end of this chapter. Why do we create a new form (form = RegistrationForm()) and then immediately call form.validate_on_submit()? Won't calling validate_on_submit() try to validate a newly initialized form?

I know that is not what is happening, but how does validate_on_submit() validate the form that was sent to this view function via a POST request.

#12 Miguel Grinberg said a year ago

@Ross: this is covered in the forms chapter. A form object initialized itself automatically with the contents of the flask request. If the request includes form data, this data is copied to the form object on creation.

#13 **Varshni** said a year ago

Hi, I find this tutorial really awesome. Your articles are detailed and easy to understand. Really appreciate your efforts. I just can't understand what purpose does the user loader function serves in the app.

#14 Miguel Grinberg said a year ago

@Varshni: each time a client sends a request, the user loader callback is used to bring the load the user object that the client is authenticated against from the database.

#15 **DH** said a year ago

Thank you for updating your tutorial - perfect timing. Can't wait to buy the new edition!!!!!!!!

Two quick questions:

Does the 2024 edition address the flask-login and other issues you raised in the "We Have To Talk About Flask" post??

I built an app based on the original FMT and it works great - except the same user login ID can be used by multiple people concurrently on different browsers. Did I miss something or is there a fix for this?

Thanks again!!

#16 **Yuming** said a year ago

Hi Miguel,
Although users' passwords are hashed before being stored in the database, we developers can still see plain text passwords on the server before they are hashed (use print on deployment sites, use debugger or wireshark on localhost, etc). I wonder if the password mechanism we learn from this chapter is secure enough for serious web sites?

When we publish web sites like Microblog, there are always trade offs: If we provide some features only for authenticated users, then some potential users may not want to register, on the other hand, if we provide some features without requiring user registration, then bad guys may abuse our sites. I wonder if you have any suggestion? Is it better if we allow users to signup/signin through Google/Facebook etc?

In comment #2 of above, you mentioned that "Once you learn the basic authentication patterns such as the one I present in this tutorial, you should be able to expand it to build your own. Or maybe find a Flask extension that does what you need.". Can you elaborate a little bit more? How secure is secure enough? Any good Flask extension or tutorial to recommend?

Thank you in advance!

#17 Miguel Grinberg said a year ago

@Yuming: what is the concern that you have with the solution used in this tutorial? The solution is secure. With wireshark you cannot see any passwords, because you are supposed to use TLS on your production site. Debugger or prints are not a concern in production, so they do not affect the security of the production site.

What I sad in #2 refers to adding roles and permissions on top of the authentication method used in this tutorial. It does not refer to making the solution more secure.

#18 Miguel Grinberg said a year ago

@DH:

> *Does the 2024 edition address the flask-login and other issues you raised in the "We Have To Talk About Flask" post??*

The issues are problems in the way the Flask project is run, they are not bugs that need to be fixed in this tutorial. As of now, everything works fine. I cannot guarantee that this will always be the case, as Flask likes to make backwards incompatible changes that may force me to make changes in the future to keep things working.

> *I built an app based on the original FMT and it works great - except the same user login ID can be used by multiple people concurrently on different browsers. Did I miss something or is there a fix for this?*

Most applications allow you to log in from multiple times, this isn't a bug. You can add logic to prevent multiple instances if you want, but this tutorial does not take that non-standard approach.

#19 **Alex** said a year ago

Hello, how can I change the error text?

#20 Miguel Grinberg said a year ago

@Alex: head over to the WTForms documentation to learn how to configure your validators: https://wtforms.readthedocs.io/en/3.1.x/validators/.

#21 **Yuming** said a year ago

Hi Miguel,
My concern expressed in message #16 above is not with the solution taught by this chapter, but is a more general concern with this type of password management.

Even on production sites, Linode.com for example, TLS is terminated by a Nginx proxy server, so our apps can still see clear text passwords after the network data is decrypted by the Nginx. If I add print(form.username.data) in your login() function above and re-start Microblog with gunicorn through a remote ssh session, then I can see every logging-in user's clear text password.

You can see that my concern is not about exposing our users' passwords to hackers, but about the fact that we, as developers or web masters, have the possibility to peek our users' passwords. I wonder how many people are knowledgeable enough to understand this fact and avoid using their important passwords for less important websites that require user login? Will we lose potential viewers if we expose some nice app features only to authenticated users?

Using public-private keys can avoid passing clear text passwords, but that is too technical for ordinary people. Maybe authentication using Google etc can make people more comfortable? After all, Google is more trustworthy than the developers behind millions of small websites?

Thanks!

#22 Miguel Grinberg said a year ago

@Yuming:

> *Even on production sites, Linode.com for example, TLS is terminated by a Nginx proxy server*

It doesn't really matter where TLS is terminated. The server will always have access to the unencrypted request with all of its data.

> *If I add print(form.username.data) in your login() function above and re-start Microblog with gunicorn through a remote ssh session, then I can see every logging-in user's clear text password.*

What would be purpose of printing or logging passwords? This only makes sense if you are running a malicious site that wants to steal passwords.

> *You can see that my concern is not about exposing our users' passwords to hackers, but about the fact that we, as developers or web masters, have the possibility to peek our users' passwords.*

Well yes, but users wouldn't create an account on a website they don't trust, right?

> *I wonder how many people are knowledgeable enough to understand this fact and avoid using their important passwords for less important websites that require user login?*

I think from the fact that you have to type your password and submit it in a form it should be clear to most people that the password will be, at least for a short amount of time, accessible to the server. How would the server do the authentication if it can't see the password?

Also, the accepted practice is that you shouldn't ever reuse your passwords, so the idea that you have "important" passwords that you use on several sites is really bad. Each password should be used on one site.

> *Using public-private keys can avoid passing clear text passwords, but that is too technical for ordinary people.*

TLS protects the communication so that nothing is sent in clear text. If you want to avoid sending a password at all, then maybe look into the Digest authentication method. It's rarely used these days, though.

> *Maybe authentication using Google etc can make people more comfortable? After all, Google is more trustworthy than the developers behind millions of small websites?*

You think? You keep going back to the suggestion that the server runs a malicious website. This isn't a problem if you don't reuse your passwords, as is standard practice. If we remove the malicious website situation, what's left is hacks that expose secrets. And I'd say Google is more likely to get hacked than some unknown website with a small userbase.

There is nothing wrong with using a third-party authentication provider, I do it on many of my projects and it is a well accepted practice. But I don't believe it is more secure. Maybe it adds the convenience to the user to not have to create a separate password, but if you have all your logins tied to Google and one day Google happens to be hacked, then you've lost everything. The most secure solution in my opinion is to use unique passwords on all sites, so that when one site gets hacked only your account on that site is compromised.

#23 **Yuming** said a year ago

"Well yes, but users wouldn't create an account on a website they don't trust, right?", that is exactly what my concern is. I think it is a chicken and egg problem. I believe quite some people do avoid registering to unfamiliar websites and providing real email address, then the good features of our websites available only for authenticated viewers won't have a chance to attract more viewers. That was why I asked if supporting sign-up through Google etc can help attract more viewers.

When I said "Using public-private keys can avoid passing clear text passwords" in comment #21 above, I did not mean TLS/SSL, I was talking about using a key pair for password-less login, as covered in Chapter 17 of this tutorial.

Thanks for your replies, I really appreciate your help.

#24 **Miguel Grinberg** said a year ago

@Yuming: I don't know if Google signups attract more users. I don't believe it does, but I don't really know. People will create an account on your website if they get value out of it. Each person has their preference with regards to login methods, so if you want to cover all the bases you should offer both username/password and Google/Facebook style logins. Then people can use what they like. In terms of security, username/password is more secure in my opinion, as I explained above. I always use username/password when signing up to a new website. I have very few accounts connected to a 3rd party provider.

#25 **sam** said a year ago

how do delete the user i have registered together with his post and profile or otherwise just removing the user from the database

«« « » »»

Leave a Comment

Name

Email

Comment

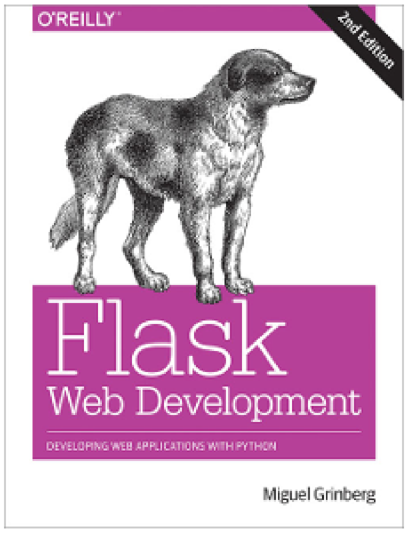
Captcha

I'm not a robot

reCAPTCHA
Privacy - Terms

Submit

Flask Web Development, 2nd Edition



If you want to learn modern web development techniques with Python and Flask, you may find the second edition of my O'Reilly book useful.

Click here to get this Book!

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 Github, LinkedIn, Bluesky, Mastodon, Twitter, YouTube, and Patreon.

Thank you for visiting!

Categories

- AI *2*
- Arduino *7*
- Authentication *10*
- Blog *1*
- C++ *5*
- CSS *1*
- Cloud *11*
- Database *23*

Docker	5
Filmmaking	6
Flask	128
Games	1
IoT	8
JavaScript	35
MicroPython	9
Microdot	1
Microservices	2
Movie Reviews	5
Personal	3
Photography	7
Product Reviews	2
Programming	191
Project Management	1
Python	173
REST	7
Raspberry Pi	8
React	18
Reviews	1
Robotics	6
Security	12
Video	22
WebSocket	2
Webcast	3
Windows	1