

The Flask Mega-Tutorial Part V: User Logins January 2 2018

(/post/the-flask-mega-tutorial-part-v-user-logins)

Posted by [Miguel Grinberg \(/author/Miguel Grinberg\)](/author/Miguel Grinberg) under [Programming \(/category/Programming\)](/category/Programming), [Security \(/category/Security\)](/category/Security), [Python \(/category/Python\)](/category/Python), [Flask \(/category/Flask\)](/category/Flask).

[Tweet](#)[Like](#)[Share](#)

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.

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

- Chapter 1: Hello, World! (/post/the-flask-mega-tutorial-part-i-hello-world)
- Chapter 2: Templates (/post/the-flask-mega-tutorial-part-ii-templates)
- Chapter 3: Web Forms (/post/the-flask-mega-tutorial-part-iii-web-forms)
- Chapter 4: Database (/post/the-flask-mega-tutorial-part-iv-database)
- Chapter 5: User Logins (/post/the-flask-mega-tutorial-part-v-user-logins) (this article)
- Chapter 6: Profile Page and Avatars (/post/the-flask-mega-tutorial-part-vi-profile-page-and-avatars)
- Chapter 7: Error Handling (/post/the-flask-mega-tutorial-part-vii-error-handling)
- Chapter 8: Followers (/post/the-flask-mega-tutorial-part-viii-followers)
- Chapter 9: Pagination (/post/the-flask-mega-tutorial-part-ix-pagination)
- Chapter 10: Email Support (/post/the-flask-mega-tutorial-part-x-email-support)
- Chapter 11: Facelift (/post/the-flask-mega-tutorial-part-xi-facelift)
- Chapter 12: Dates and Times (/post/the-flask-mega-tutorial-part-xii-dates-and-times)
- Chapter 13: I18n and L10n (/post/the-flask-mega-tutorial-part-xiii-i18n-and-l10n)
- Chapter 14: Ajax (/post/the-flask-mega-tutorial-part-xiv-ajax)
- Chapter 15: A Better Application Structure (/post/the-flask-mega-tutorial-part-xv-a-better-application-structure)
- Chapter 16: Full-Text Search (/post/the-flask-mega-tutorial-part-xvi-full-text-search)
- Chapter 17: Deployment on Linux (/post/the-flask-mega-tutorial-part-xvii-deployment-on-linux)
- Chapter 18: Deployment on Heroku (/post/the-flask-mega-tutorial-part-xviii-deployment-on-heroku)
- Chapter 19: Deployment on Docker Containers (/post/the-flask-mega-tutorial-part-xix-deployment-on-docker-containers)
- Chapter 20: Some JavaScript Magic (/post/the-flask-mega-tutorial-part-xx-some-javascript-magic)
- Chapter 21: User Notifications (/post/the-flask-mega-tutorial-part-xxi-user-notifications)

- Chapter 22: Background Jobs (</post/the-flask-mega-tutorial-part-xxii-background-jobs>)
- Chapter 23: Application Programming Interfaces (APIs) (</post/the-flask-mega-tutorial-part-xxiii-application-programming-interfaces-apis>)

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

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

In Chapter 3 (</post/the-flask-mega-tutorial-part-iii-web-forms>) you learned how to create the user login form, and in Chapter 4 (</post/the-flask-mega-tutorial-part-iv-database>) 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 (<https://github.com/miguelgrinberg/microblog/tree/v0.5>), Zip (<https://github.com/miguelgrinberg/microblog/archive/v0.5.zip>), Diff (<https://github.com/miguelgrinberg/microblog/compare/v0.4...v0.5>).

Password Hashing

In Chapter 4 (</post/the-flask-mega-tutorial-part-iv-database>) 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 (<http://werkzeug.pocoo.org/>), 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:

```
>>> from werkzeug.security import generate_password_hash
>>> hash = generate_password_hash('foobar')
>>> hash
'pbkdf2:sha256:50000$VT9fkZM8$04dfa35c6476acf7e788a1b5b3c35e217c78dc04539d295f011f01f18cd2175f'
```

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 obtain the original password. As an additional measure, if you hash the same password multiple times, you will get different results, 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_hash

# ...

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 (<https://flask-login.readthedocs.io/>). 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` for a special, anonymous user.
- `get_id()`: a method that returns a unique identifier for the user as a string (unicode, if using Python 2).

I can implement these four easily, but since the implementations are fairly generic, Flask-Login provides a *mixin* class called `UserMixin` that includes generic 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 User.query.get(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
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 = User.query.filter_by(username=form.username.data).first()
        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 to obtain the user object that represents the client of the 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

`filter_by()` method of the SQLAlchemy query object. The result of `filter_by()` is a query that only includes the objects that have a matching username. Since I know there is only going to be one or zero results, I complete the query by calling `first()`, which will return the user object if it exists, or `None` if it does not. In Chapter 4 (</post/the-flask-mega-tutorial-part-iv-database>) you have seen that when you call the `all()` method in a query, 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` decorators 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 first 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:

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

```

from flask import request
from werkzeug.urls import url_parse

@app.route('/login', methods=['GET', 'POST'])
def login():
    # ...
    if form.validate_on_submit():
        user = User.query.filter_by(username=form.username.data).first()
        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 url_parse(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 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 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 relative or absolute, I parse it with Werkzeug's `url_parse()` 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 (</post/the-flask-mega-tutorial-part-ii-templates>) 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:

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></div>
    {% 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=posts)
```

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 start the application and try to access `http://localhost:5000/` or `http://localhost:5000/index`, 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.

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, SubmitField
from wtforms.validators import ValidationError, DataRequired, Email, EqualTo
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('password')])
    submit = SubmitField('Register')

    def validate_username(self, username):
        user = User.query.filter_by(username=username.data).first()
        if user is not None:
            raise ValidationError('Please use a different username.')

    def validate_email(self, email):
        user = User.query.filter_by(email=email.data).first()
        if user is not None:
            raise ValidationError('Please use a different email address.')
```

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.

I have also added two methods to this class called `validate_username()` and `validate_email()`. 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. 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 `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.

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!</a></p>
```

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

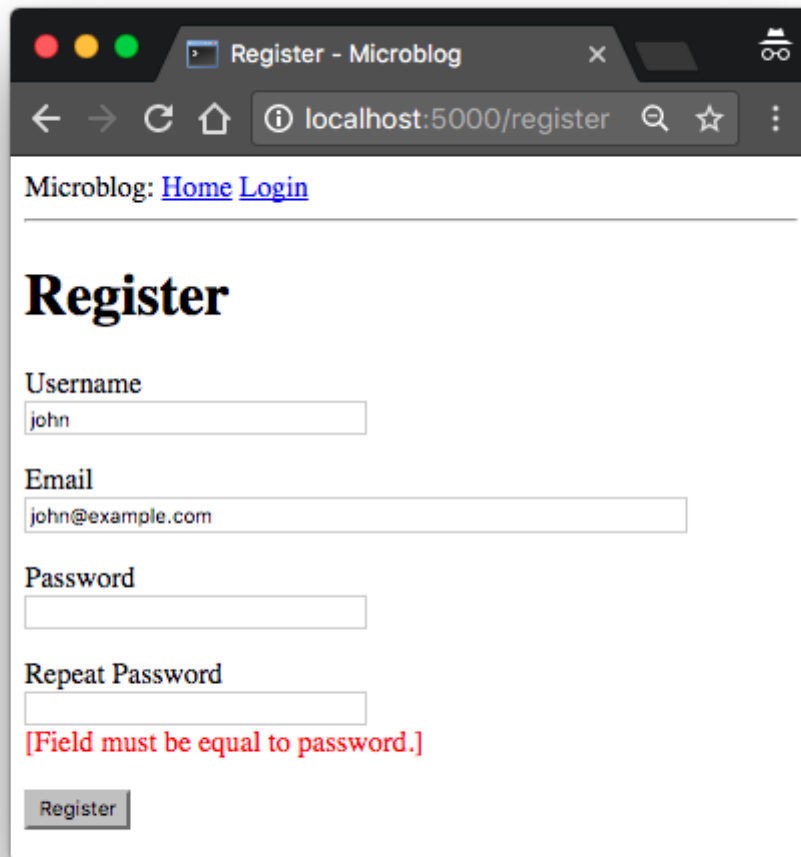
app/routes.py: User registration view function

```
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=form)
```

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.



The screenshot shows a web browser window with the title 'Register - Microblog'. The address bar shows 'localhost:5000/register'. The page content includes a navigation bar with links for 'Microblog:', 'Home', and 'Login'. Below this is a large heading 'Register'. The form contains four input fields: 'Username' (with 'john' entered), 'Email' (with 'john@example.com' entered), 'Password', and 'Repeat Password'. A red error message '[Field must be equal to password.]' is displayed below the 'Repeat Password' field. At the bottom of the form is a 'Register' button.

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.

Hello, and thank you for visiting my blog! If you enjoyed this article, please consider supporting my work on this blog on Patreon (<https://patreon.com/miguelgrinberg>)!

 **BECOME A PATRON**

(<https://patreon.com/miguelgrinberg>)

Tweet

Like



Share

