

Flask

Part 3: Form Validation. Comments. Admin panel.

Lab Objectives

Objectives of this lab are to complete developing essential functionality for our **Blogging Site**, by:

- adding validation to user authentication functionality;
- implementing functionality to allow users to post comments on blog articles;
- incorporating a basic administration panel to provide site management tools.

NB: As for all labs, you should do all the work with your virtual environment **activated**.

General Comments

As before, this exercise is not assessed. If you do not manage to finish all the tasks in the lab, please attempt to finish them in your own time.

Use the suggested resources and links, provided throughout this document and on the last page, to help you understand the code. Snapshots of the state of the project are available on Learning Central (see 'Week 11 - Afternoon Session - Flask 3' folder). You can download this to help you if you are stuck, to check your progress (and, perhaps, at times to save your typing *from scratch*). However, please make sure you understand each line of the code!

It is okay to discuss the solutions with your peers (these exercises are not assessed!), however, make sure you understand everything by yourself.

IMPORTANT!! The labs will give you some basic understanding of how to develop a website in Flask. To get more comprehensive understanding of how this 'stuff' works, it is strongly advised that you read the recommended book, suggested documentation, 'quickstarts', textbook, as well as complete a few tutorials. However, the suggested resources are just suggestions and the list is non-exhaustive! There are lots of other tutorials and resources on the Web.

Abbreviations used in this document

- **db** – database
- **NB** – Nota bene

FORM VALIDATION ¹

In our previous lab we practised developing Flask forms. However, we assumed that the users would only provide input that is valid. If they don't, the system will behave unexpectedly and possibly crash. For example, if a user tries to register with a username that already exists, we would get a SQLAlchemy's '*database integrity error*'. To avoid issues like this, we need to validate the users' input and provide them with hints and help, e.g. we want to reinforce the rule that the username and email is unique and inform the user if this username and email are already taken.

In this section, we will work on the following files: `forms.py` to define functions for user input validation, appropriate templates (e.g. `register.html` to tell the server how to render the content), and `routes.py` to bind specific URLs to our functions.

1. Let's check `username` already exists. In `forms.py`, we would specify this rule as:

```
...
def validate_username(self, username):
    user = User.query.filter_by(username=username.data).first()
    if user:
        raise ValidationError('Username already exist. Please choose a
        ↪ different one.')
```

NB: The above code requires an import of `ValidationError` from `wtforms.validators` and an import of `User` from `blogs.models`.

2. Still in `forms.py`, create a similar validation for the users' **emails**, i.e.

```
...
def validate_email(self, email):
```

3. We could also specify certain rules for passwords. Suppose we want to reinforce the following rule: a password must only contain alphanumeric characters (i.e. any characters, e.g. `abcde1` will be valid, but not `abc1`).²

We can use a regular expression (regex) for this. Update `password` in `forms.py`, as follows:

```
...
password = PasswordField('Password', validators=[DataRequired(),
        Regexp('^{6,8}$', message='Your password should be
        ↪ between 6 and 8 characters long.')]])
...
```

NB: Don't forget to import `Regexp` from `wtforms.validators`.

If you want to learn more about regex:

- See [Learning Materials/ Optional Materials/\[Python\] Regular Expressions](#), and in particular "Lecture II" under [RegEx Lecture Material](#).

¹ In this lab, we will be using Flask to validate the forms. Alternatively, you might want to look into using JavaScript to achieve this.

² This, of course, is a simple requirement for the password. If we want to make the rule more complicated, we would need to use a more complicated regex. For instance, if we want to make sure that a user's password must be between 6 and 8 characters long AND contains at least one numeric digit, the regex for this would be: `^(?=.*\d).{6,8}$`.

- https://www.w3schools.com/python/python_regex.asp has a number of examples which you can try out.

4. Next, implement appropriate validation for the *log in* functionality.

Routing

5. After we specified the logic, we need to add checking the form is valid when it is submitted, so in `routes.py` instead of using

```
if request.method == 'POST':
```

we need to use:

```
if form.validate_on_submit():
```

in the appropriate `@app.route(...)` decorators (`register()` and `login()`).

Error Messages

Any good system should provide its user with feedback. We have already encountered that we can specify a message to the user during form validation (Task 3). We can also use a messaging system provided by Flask, called '*flashing system*'.

NB: This functionality requires an import of `flash` from `flask` in `routes.py`.

6. The following is an example of a flash message, which we can add to `routes.py`:

```
...
flash('Invalid email address or password.')
...
```

7. To enable the '*flashing system*', we need to add code to the templates to instruct the server to display the messages:

(a) *site-wide*, by adding the following to `layout.html`:

```
<div>
{% with messages = get_flashed_messages() %}
  {% if messages %}
    <ul class=flashes>
      {% for message in messages %}
        <li>{{ message }}</li>
      {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
</div>
```

(b) and on a *specific page*, e.g. in `register.html`:

```
{% for error in form.username.errors %}
  <span style="color: red;">[{{ error }}]</span>
{% endfor %}
```

NB: More information on *Message Flashing* in Flask can be found at: <https://flask.palletsprojects.com/en/1.1.x/patterns/flashing/>

POSTING COMMENTS

In this section of the lab, we will implement a basic ‘commenting’ functionality, which will be stored in our db in a new table. This functionality will allow users to post comments under blog articles and respond to other user’s comments.

NB: As usual, the complete code is provided in the snapshot of the project (on Learning Central). If you are stuck, please consult these files.

Adding a Comment

8. **Adding a comment** to a blog post, as with any new feature, requires us to update each part of our MVC architecture. We must update our Models (`models.py`), our Controllers (`forms.py` and `routes.py`), and our View (`post.html`).

- (a) We must create a new table to store comments for posts, intuitively this can be done by creating a new class in `models.py`:

```
...
class Comment(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    date = db.Column(db.DateTime, nullable=False,
        ↪ default=datetime.utcnow)
    content = db.Column(db.Text, nullable=False)
    parent_id = db.Column(db.Integer, db.ForeignKey('comment.id'),
        ↪ nullable=True)
    post_id = db.Column(db.Integer, db.ForeignKey('post.id'),
        ↪ nullable=False)
    author_id = db.Column(db.Integer, db.ForeignKey('user.id'),
        ↪ nullable=False)
    parent = db.relationship('Comment', backref='comment_parent',
        remote_side=id, lazy=True)

    def __repr__(self):
        return f"Post('{self.date}', '{self.content}')
```

- (b) Similar to the `User` and `Post` classes, we must update our `User` class in `models.py` to link the relationship between `User` and `Comment`:

```
...
comment=db.relationship('Comment',backref='user',lazy=True)
...
```

Don’t forget to update your database.

- (c) The logic for a valid form entry for a comment is described in `forms.py` by the class `CommentForm(FlaskForm)`:

```
...
class CommentForm(FlaskForm):
    comment = StringField('Comment', validators=[InputRequired()])
    submit = SubmitField('Post comment')
```

NB: This form needs `InputRequired` to be imported from `wtforms.validators`.

- (d) To view comments left on a post the `post(post_id)` function in `routes.py` must be *updated*:

```
...
@app.route("/post/<int:post_id>")
def post(post_id):
    post = Post.query.get_or_404(post_id)
    comments = Comment.query.filter(Comment.post_id == post.id)
    form = CommentForm()

    return render_template('post.html', post=post, comments=comments, form=form)
...
```

- (e) Similarly, to submit new comments we must add a new function to `routes.py`:

```
...
@app.route('/post/<int:post_id>/comment', methods=['GET', 'POST'])
@login_required
def post_comment(post_id):
    post = Post.query.get_or_404(post_id)
    form = CommentForm()
    if form.validate_on_submit():
        db.session.add(Comment(content=form.comment.data,
                                ↪ post_id=post.id, author_id=current_user.id))
        db.session.commit()
        flash("Your comment has been added to the post", "success")
        return redirect(f'/post/{post.id}')

    comments = Comment.query.filter(Comment.post_id == post.id)
    return render_template('post.html', post=post, comments=comments,
        ↪ form=form)
```

NB: The final two lines of `post_comment()` provide the user with feedback upon an error, assuming that the form submitted is not valid.

NB: Make sure to update the necessary imports at the top of `routes.py`:

```
...
from blog.models import Comment
from blog.forms import CommentForm
from flask_login import login_required, current_user
...
```

9. To display comments associated with a post and to submit new comments we add the following to `post.html`:

```
...
{% for comment in comments %}
    <p>{{ comment.content }}&nbsp;<small>Author:
        {{ comment.user.username }}</small></p>
{% endfor %}

<form method="POST" action="{{ url_for('post_comment', post_id=post.id)
    ↪ }}">
    {{ form.hidden_tag() }}
```

```

<div class="">
    {{ form.comment.label }} {{ form.comment }}
    {% for error in form.comment.errors %}
        <span style="color: red;">[{{ error }}]</span>
    {% endfor %}
</div>
<div class="">
    {{ form.submit() }}
</div>
</form>
...

```

10. When the user clicks on the link, a comment is added to the blog post and the user is redirected to `post.html`, effectively refreshing the page.

NB: In the above code, we use `{% statement %}` and `{{ expression }}` in the Jinja template engine to execute instructions similar to in Python (see Jinja Template documentation for more information on built in filters, expressions, etc.: <https://jinja.palletsprojects.com/en/2.11.x/templates/>).

ADMIN PANEL

An administration panel will provide a GUI that allows novice administrators to manage the website through a simple interface.

11. To implement basic database manipulation functionality, we can add the following code to our `__init__.py` file:

```

...
from flask_admin import Admin
from flask_admin.contrib.sqla import ModelView
from blog.models import User, Post, Comment
admin = Admin(app, name='Admin panel', template_mode='bootstrap3')
admin.add_view(ModelView(User, db.session))
admin.add_view(ModelView(Post, db.session))
admin.add_view(ModelView(Comment, db.session))

```

If we visit `http://127.0.0.1:5000/admin`, we can view and edit our db's tables from the horizontal navigation bar.

12. We can improve the security of this by making it so that only users with administrative rights can access these pages.
 - (a) First, we need to update our `User()` class in `models.py` by adding the following field to identify whether a user is indeed an administrator:

```

...
is_admin = db.Column(db.Boolean, nullable=False, default=False)
...

```

- (b) We must update our `user` table in our db to reflect this change, i.e. we must tell the system that a user is an `admin` user. This can be done by changing `is_admin` attribute to `1` (from `0`), using a db GUI (e.g. phpMyAdmin or MySQL Workbench). Alternatively, you can use the following MySQL commands:

```
UPDATE user
SET is_admin = 1
WHERE username = 'johnsmith';
```

- (c) We need to extend the functionality of the built-in `ModelView` by modifying the function `is_accessible()`. To do this we create a new file called `views.py` containing the following code:

```
from flask_admin.contrib.sqla import ModelView
import flask_login as login
from blog.models import User

class AdminView(ModelView):
    def is_accessible(self):
        if login.current_user.is_authenticated:
            if login.current_user.get_id():
                user = User.query.get(login.current_user.get_id())
                return user.is_admin
        return False
```

This code checks that the user is logged in and then verifies that the `is_admin` field reads a `True`.

- (d) We must now modify our code in `__init__.py` file to use this new class:

```
...
from flask_admin import Admin
from blog.views import AdminView
from blog.models import User, Post, Comment
admin = Admin(app, name='Admin panel', template_mode='bootstrap3')
admin.add_view(AdminView(User, db.session))
admin.add_view(AdminView(Post, db.session))
admin.add_view(AdminView(Comment, db.session))
```







- (e) Finally, we can improve the ‘*feedback*’ of the site by incorporating some logic into the admin home screen. Create a file in the following directory of your project `templates/admin/index.html`:

```
{% extends 'admin/master.html' %}
{% block body %}
{% if current_user.is_admin %}
    Welcome to the admin panel {{ current_user.username }}!
{% else %}
    Hello, please login as an admin before accessing this panel &nbsp;|
    <a href="{{ url_for('login') }}">Login&nbsp;</a>
{% endif %}
{% endblock %}
```

CONCLUDING REMARKS

This completes the series of labs on Flask, and you should now have a basic blogging site. You could use the lab work for your coursework, but you will need to carry on and implement additional functionality, required for the coursework - see the coursework brief on Learning Central. You can also style your website to create a ‘*look and feel*’ you want your website to have.

Due to the time and scope constraints, the work we produced in these labs has **several limitations**, e.g. with regard to enhanced security and functionality. This constitutes **further work**, e.g. in your coursework and other independent work, other modules and beyond.

Useful resources	
Flask Website:	https://flask.palletsprojects.com/en/1.1.x/ 
Flask Tutorial:	https://flask.palletsprojects.com/en/1.1.x/tutorial/ 
Flask-WTF home page:	https://flask-wtf.readthedocs.io/en/stable/ 
Flask-WTF Quickstart:	https://flask-wtf.readthedocs.io/en/stable/quickstart.html 
WTForms Form Validation	https://flask.palletsprojects.com/en/1.1.x/patterns/wtforms/ 
Python RegEx	https://www.w3schools.com/python/python_regex.asp 
Book on Flask:	M. Grinberg’s (2018). "Flask web development : developing web applications with Python", O’Reilly.
Plenty of video tutorials on YouTube (<i>as usual, in no particular order!</i>), e.g.:	[1] , [2] (these links are to the first lessons in a series; other episodes cover the flask forms).
A number of cheat sheets could be found on the web.	