Lecture 17

Flask-Admin and Flask-Login

Acknowledgement:

https://flask-admin.readthedocs.io

https://flask-login.readthedocs.io

Flask-Admin

- Solves the boring problem of building an admin interface on top of an existing data model
- Lets you manage your web service’s data through a user-friendly interface, with very little effort
- Allows you to add views of tables with built in Create, Read, Update, Delete (CRUD)

```
pip install flask-admin
```

Empty Admin Page

- Navigate to http://localhost:5000/admin to see empty page

```
from flask import Flask
from flask_admin import Admin

app = Flask(__name__)
# set optional bootswatch theme
app.config['FLASK_ADMIN_SWATCH'] = 'cerulean'

admin = Admin(app, name='microblog', template_mode='bootstrap3')
# Add administrative views here

app.run()
```

Database Model Defined in SQLAlchemy

- Predefine a DB model with SQLAlchemy

```
from flask_sqlalchemy import SQLAlchemy
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///example.sqlite"
db = SQLAlchemy(app)

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String, unique=True, nullable=False)
    email = db.Column(db.String, unique=True, nullable=False)
```

Adding Model Views

- Model views allow you to add a dedicated set of admin pages for managing any model in your database
- Create instances of the ModelView class from SQLAlchemy and add them to the admin view

```
from flask_admin.contrib.sqla import ModelView
app.secret_key = 'super secret key’ # Add this to avoid an error

# Flask and Flask-SQLAlchemy initialization here
admin = Admin(app, name='microblog', template_mode='bootstrap3’)
admin.add_view(ModelView(User, db.session))
```

Add Content to Admin Page

- Save the following text as admin/index.html in your project's templates directory:

```
{% extends 'admin/master.html' %}

{% block body %}
  <p>This is an admin page. Click the navigation to explore.</p>
{% endblock %}
```

Authorization & Permissions

- Very important you don’t just let anyone access the admin page
- Create an admin user and give them access to the admin page
- You can use Flask-Login just like for other users who log in

Customizing Built-in Views

- When inheriting from ModelView, values can be specified for numerous configuration parameters
- Use these to customize the views to suit your particular models

```
class UserView(ModelView):
  can_delete = False # disable model deletion
  can_create = False # disable model creation
  can_edit = False # disable model editing
  column_exclude_list = ['password', ] # exclude the password column

admin.add_view(UserView(User, db.session))
```

Customizing Built-in Views

```
class UserView(ModelView):
  column_searchable_list = ['name', 'email'] # make columns searchable
  form_choices = { # restrict the possible values for a text-field
    'title': [
      ('MR', 'Mr’),
      ('MRS', 'Mrs’),
      ('MS', 'Ms’),
      ('DR', 'Dr’)
    ]
  }
  can_export = True # enable csv export of the model view

admin.add_view(UserView(User, db.session))
```

Many to One

- With a Many to One relationship you get a dropdown selection

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String, unique=True, nullable=False)
    email = db.Column(db.String, unique=True, nullable=False)
    def __repr__(self):
        return '<User %r>' % self.username

class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(80), nullable=False)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)
    user = db.relationship('User', backref=db.backref('posts', lazy=True))

admin = Admin(app, name='microblog', template_mode='bootstrap3')
admin.add_view(ModelView(User, db.session))
admin.add_view(ModelView(Post, db.session))
```

Wake-up!

- https://youtu.be/Lrlro3YJ15o

Flask-Login

- Provides user session management for Flask
- Low level library
  - may consider flask-security-too for higher level library (based on Flask-Login)
- Handles the common tasks:
  - Logging in
  - Logging out
  - Remembering your users' sessions over extended periods of time

Flask-Login

- What it does:
  - Stores the active user's ID in the session and lets you log them in/out easily
  - Let you restrict views to logged-in (or logged-out) users
  - Handle the normally-tricky "remember me" functionality
  - Help protect your users' sessions from being stolen by cookie thieves
- What it does not do:
  - Impose a particular database or other storage method on you
  - Restrict you to using usernames and passwords, OpenIDs, or any other method of authenticating
  - Handle permissions beyond logged in or not
  - Handle user registration or account recovery

Configuring your Application

- The login manager contains the code that lets your application and Flask-Login work together
- By default, Flask-Login uses sessions for authentication
- This means you must set the secret key on your application, otherwise Flask will give you an error message telling you to do so

```
login_manager = LoginManager()
login_manager.init_app(app)
login_manager.login_view = 'login'
app.secret_key = 'keep it secret, keep it safe' # Add this to avoid an error
```

How it Works

- You will need to provide a user_loader callback
- This callback is used to reload the user object from the user ID stored in the session
- It should take the unicode ID of a user, and return the corresponding user object

```
@login_manager.user_loader
def load_user(user_id):
  return User.get_id(user_id)
```

Your User Class

- The class that you use to represent users needs to implement these properties and methods
  - **is_authenticated** - This property should return True if the user is authenticated, i.e. they have provided valid credentials
  - **is_active** - This property should return True if this is an active user - in addition to being authenticated, they also have activated their account, not been suspended, etc
  - **is_anonymous** - This property should return True if this is an anonymous user
  - **get_id()** - This method must return a unicode that uniquely identifies this user, and can be used to load the user from the user_loader callback
- Alternatively, you may inherit from UserMixin
  - provides default implementations for all of these properties and methods
  - class User(UserMixin, db.Model)

Check Password Function

- Add a check_password function to your User Class
- Hash it if the password has been hashed

```
class User(db.Model):
    # ...
    def check_password(self, password):
        return self.password == password
```

Login Example

```
from flask_login import current_user, login_user


@app.route('/login', methods=['POST'])
def login():
    if current_user.is_authenticated:
        return redirect(url_for('index'))
    user = User.query.filter_by(username=request.json['username']).first()
    if user is None or not user.check_password(request.json['password']):
         return redirect(url_for('login'))
    login_user(user)
    return redirect(url_for('index'))
```

Protecting Views

- Views that require your users to be logged in can be decorated with the login_required decorator:

```
from flask_login import login_required


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

Logging Users Out

```
# ...
from flask_login import logout_user


# ...


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