Authentication & Security – II

Flask-Login, access control, secure routes

Authentication & Security — II

1) Flask-Login — purpose

A lightweight Flask extension that handles user session management:

- login,
- logout,
- current_user,
- @login_required

- It avoids reimplementing session bookkeeping;
- integrates with Flask easily and gives common protections (session protection, user loader hooks).

2) UserMixin, login_user, logout_user, current_user

- UserMixin implements required properties (is_authenticated, is_active, get_id).
- login_user() creates a session;
- logout_user() clears it;
- current_user exposes the logged-in user object

It provides clean API to check auth state and run logic based on authenticated identity.

3) @login_required and protecting routes

- Decorator redirects unauthenticated users to the login page (configurable)
- It ensures only authenticated users can access sensitive pages.

4) Role-based access control (RBAC) & permissions

- Store a role/permission set on user (e.g., role='admin') and enforce checks at view level or through decorators.
- It helps different users as they need different access (admin panel vs general user);
- RBAC enforces principle of least privilege.

5) Fresh logins and sensitive actions

- fresh_login_required forces users to re-authenticate for critical actions (password change, money transfer).
- It Mitigates risks from long-lived sessions that may have been hijacked.

6) Session protection & cookie security

1. Use SESSION_COOKIE_SECURE, SESSION_COOKIE_HTTPONLY, SESSION_COOKIE_SAMESITE.
2. Enable login_manager.session_protection = "strong". Regenerate session id after login.
3. It Reduces session theft (XSS, eavesdropping) and session fixation attacks.

7) Logout & session revocation

- logout_user() plus optionally server-side session store (Redis) to be able to forcibly revoke server-side sessions.

- Invalidate tokens/sessions promptly after logout or admin revocation.

8) Protecting views in blueprints & APIs

- Apply decorators at blueprint level or use token-based auth (e.g., JWT) for APIs rather than Flask-Login sessions.
- APIs are stateless; cookies/session-based auth is not ideal for machine-to-machine calls.

9) Secure config & secrets management

- Keep SECRET_KEY, DB URI, salts out of code (use env vars or secrets manager). (Use HTTPS in production.)
- Prevents attackers from forging sessions or reading credentials.

10) Logging & audit trails

- Log login attempts, failed logins, and admin actions.
- Detects attacks and supports incident response.

Step-by-step Flask program (single-file demo)

- **Features:**
  register/login/logout,
  protected user page,
  admin-only page,
  role_required decorator,
  secure cookie settings,
  CSRF via Flask-WTF
- **Requirements:**
  >> pip install Flask Flask-Login Flask-WTF Flask-SQLAlchemy

Create app.py with the code below and run it.

```
# app.py
from flask import Flask, render_template_string, redirect, url_for, flash, request, abort
from flask_sqlalchemy import SQLAlchemy
from flask_login import LoginManager, UserMixin, login_user, logout_user, login_required, \
    current_user, fresh_login_required
from werkzeug.security import generate_password_hash, check_password_hash
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, SubmitField
from wtforms.validators import DataRequired, Length, EqualTo, Regexp

# ----- Config -----
app = Flask(__name__)
app.config['SECRET_KEY'] = 'replace-with-secure-random-secret'  # put this in env in production
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///auth2_demo.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

# Session cookie security
app.config['SESSION_COOKIE_HTTPONLY'] = True
app.config['SESSION_COOKIE_SECURE'] = False   # set True in production with HTTPS
app.config['SESSION_COOKIE_SAMESITE'] = 'Lax' # or 'Strict'

db = SQLAlchemy(app)
login_manager = LoginManager(app)
login_manager.login_view = 'login'
login_manager.session_protection = "strong"  # basic protection against session theft

# ----- Models -----
class User(UserMixin, db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    password_hash = db.Column(db.String(256), nullable=False)
    role = db.Column(db.String(20), default='user')  # 'user' or 'admin'

    def set_password(self, password: str):
```

```
        self.password_hash = generate_password_hash(password)

    def check_password(self, password: str) -> bool:
        return check_password_hash(self.password_hash, password)

    def is_admin(self) -> bool:
        return (self.role or '').lower() == 'admin'

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))

# ----- Forms -----
class RegisterForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired(), Length(min=3, max=80),
                Regexp(r'^[A-Za-z0-9_.-]+$', message="Letters, numbers and _ . - only")])
    password = PasswordField('Password', validators=[DataRequired(), Length(min=8)])
    confirm = PasswordField('Confirm', validators=[DataRequired(), EqualTo('password')])
    submit = SubmitField('Register')

class LoginForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    password = PasswordField('Password', validators=[DataRequired()])
    submit = SubmitField('Login')

# ----- Utility: role_required decorator -----
from functools import wraps
def role_required(role_name):
    def decorator(fn):
        @wraps(fn)
        @login_required
        def wrapper(*args, **kwargs):
            if not current_user.is_authenticated:
                return login_manager.unauthorized()
            if (current_user.role or '').lower() != role_name.lower():
                # Option: return 403 or redirect with flash
                abort(403)
            return fn(*args, **kwargs)
        return wrapper
    return decorator

# ----- Routes -----
@app.route('/')
def index():
    if current_user.is_authenticated:
        return render_template_string("""
            <h2>Welcome {{current_user.username}} ({{current_user.role}})</h2>
            <p><a href="{{ url_for('protected') }}">Your dashboard</a></p>
```

```
            <p><a href="{{ url_for('admin') }}">Admin area</a> (admin only)</p>
            <p><a href="{{ url_for('logout') }}">Logout</a></p>
        """)
    return "<h2>Welcome Guest</h2><a href='/login'>Login</a> | <a href='/register'>Register</a>"

@app.route('/register', methods=['GET','POST'])
def register():
    form = RegisterForm()
    if form.validate_on_submit():
        if User.query.filter_by(username=form.username.data).first():
            flash("Username already taken", "danger")
            return redirect(url_for('register'))
        u = User(username=form.username.data)
        u.set_password(form.password.data)
        # For demo: make the first registered user an admin
        if User.query.count() == 0:
            u.role = 'admin'
        db.session.add(u)
        db.session.commit()
        flash("Registered. Please login.", "success")
        return redirect(url_for('login'))
    return render_template_string(REG_TEMPLATE, form=form)

@app.route('/login', methods=['GET','POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        u = User.query.filter_by(username=form.username.data).first()
        if u and u.check_password(form.password.data):
            login_user(u, remember=False, fresh=True)
            # It's good practice to regenerate session id after login in production.
            flash("Logged in.", "success")
            next_page = request.args.get('next')
            return redirect(next_page or url_for('protected'))
        flash("Invalid credentials", "danger")
    return render_template_string(LOGIN_TEMPLATE, form=form)

@app.route('/logout')
@login_required
def logout():
    logout_user()
    flash("You have logged out.", "info")
    return redirect(url_for('index'))

@app.route('/protected')
@login_required
def protected():
    return render_template_string("""
```

```
    <h3>Protected dashboard</h3>
    <p>Hello {{current_user.username}} (role: {{current_user.role}})</p>
    <p><a href="{{ url_for('index') }}">Home</a></p>
  """)

@app.route('/admin')
@role_required('admin')
def admin():
  users = User.query.all()
  return render_template_string("""
    <h3>Admin Area</h3>
    <p>Only admins can see this.</p>
    <ul>{% for u in users %}<li>{{u.username}} — {{u.role}}</li>{% endfor %}</ul>
    <p><a href="{{ url_for('index') }}">Home</a></p>
  """, users=users)

# Optional: sensitive action that requires fresh login
@app.route('/sensitive-action')
@fresh_login_required
def sensitive_action():
  return "<p>Sensitive action performed (fresh login required).</p>"

# ----- Simple inline templates -----
REG_TEMPLATE = """
<!doctype html><title>Register</title>
<h2>Register</h2>
<form method="post">
 {{ form.hidden_tag() }}
 <p>{{ form.username.label }}<br>{{ form.username() }}</p>
 <p>{{ form.password.label }}<br>{{ form.password() }}</p>
 <p>{{ form.confirm.label }}<br>{{ form.confirm() }}</p>
 <p>{{ form.submit() }}</p>
</form>
<a href="{{ url_for('login') }}">Login</a>
"""

LOGIN_TEMPLATE = """
<!doctype html><title>Login</title>
<h2>Login</h2>
<form method="post">
 {{ form.hidden_tag() }}
 <p>{{ form.username.label }}<br>{{ form.username() }}</p>
 <p>{{ form.password.label }}<br>{{ form.password() }}</p>
 <p>{{ form.submit() }}</p>
</form>
<a href="{{ url_for('register') }}">Register</a>
"""
```

```
# ----- Error handlers -----
@app.errorhandler(403)
def forbidden(e):
    return "<h3>403 Forbidden</h3><p>You don't have permission to access this.</p>", 403

# ----- Run app -----
if __name__ == '__main__':
    with app.app_context():
        db.create_all()
    app.run(debug=True)
```

Note: Program enforces security

1. **Registration**: password saved as a hash (generate_password_hash) — plain text never stored.
2. **Login**: login_user(u) creates a secure session. login_manager.session_protection = "strong" adds additional session checks.
3. **Protected views**: @login_required blocks unauthenticated access. role_required('admin') checks both login and the user's role. Unauthorized access returns 403.
4. **Fresh login requirement**: @fresh_login_required forces re-authentication for very sensitive endpoints.
5. **Cookie flags and config**: SESSION_COOKIE_HTTPONLY and SESSION_COOKIE_SAMESITE reduce cookie-theft and CSRF risk. (Make SESSION_COOKIE_SECURE=True in production with HTTPS.)
6. **First user admin** (demo only): The program makes first registered user an admin — useful for testing. In real apps manage roles in admin UI or migrations.
7. **Logout**: logout_user() destroys the session.

Note:

- Use **HTTPS** and set SESSION_COOKIE_SECURE=True.
- Store SECRET_KEY and DB credentials in environment variables or secrets manager.
- Use **Argon2**/bcrypt via passlib for stronger hashing.
- Apply **rate-limiting** on login endpoint (e.g., Flask-Limiter).
- Consider **server-side sessions** (Redis) for ability to revoke sessions centrally.
- Use **CSRF protection** on state-changing actions (Flask-WTF does this if forms are used).
- Implement **audit logging**, account lockout on repeated failures, and **MFA** for higher security accounts.
- Keep role logic centralized (policy/permission system) rather than ad-hoc decorators as app grows.