Flask-Security Integration - Advanced Authentication & Authorization

Learning Objectives

By the end of this lesson, students will understand:

1. Flask-Security fundamentals and configuration
2. User authentication models with UserMixin and RoleMixin
3. Role-based access control (RBAC) implementation
4. Protected routes and decorators
5. User registration, login, and session management
6. Security best practices for web applications

Introduction: Why Flask-Security?

Building authentication from scratch is complex and error-prone. Flask-Security provides:

- **Pre-built authentication views** (login, logout, register)
- **Password hashing** with industry-standard algorithms
- **Role-based access control** (RBAC)
- **Session management** and security features
- **CSRF protection** and secure defaults
- **Email confirmation** and password recovery

What We've Already Built

In our previous lesson, we created a modular Flask application with blueprints. Now we'll add comprehensive authentication using Flask-Security, building on our existing structure:

```
fs_prep/
├── app/
│   ├── __init__.py          # ✅ Already has Flask-Security setup
│   ├── extensions.py        # ✅ Security extension configured
│   ├── models/user.py       # ✅ User & Role models with mixins
│   ├── main/views.py        # ✅ Protected routes with @login_required
```

```
│   └── auth/routes.py       # ✅ Authentication endpoints
├── config.py                # ✅ Security configuration
└── requirements.txt         # ✅ Flask-Security-Too included
```

Step 1: Understanding Flask-Security Configuration

1.1 Configuration Analysis

Let's examine how Flask-Security is configured in our existing `config.py`:

```python
# config.py
class Config:
    # Core Flask settings
    SECRET_KEY = os.getenv('SECRET_KEY')
    SQLALCHEMY_DATABASE_URI = os.getenv('DATABASE_URL')

    # Flask-Security Configuration
    SECURITY_PASSWORD_SALT = os.getenv('SECURITY_PASSWORD_SALT')
    SECURITY_REGISTERABLE = True      # Enable user registration
    SECURITY_RECOVERABLE = True       # Enable password recovery
    SECURITY_TRACKABLE = True         # Track user login activity
    SECURITY_CHANGEABLE = True        # Allow password changes
    SECURITY_CONFIRMABLE = False      # Email confirmation (disabled for dev)
    SECURITY_SEND_REGISTER_EMAIL = False  # Registration emails (disabled)
```

Key Configuration Options:

Setting	Purpose	Our Value
`SECURITY_PASSWORD_SALT`	Additional security for password hashing	From environment
`SECURITY_REGISTERABLE`	Enables `/register` endpoint	`True`
`SECURITY_RECOVERABLE`	Enables password reset functionality	`True`
`SECURITY_TRACKABLE`	Tracks login times and IP addresses	`True`
`SECURITY_CHANGEABLE`	Allows users to change passwords	`True`
`SECURITY_CONFIRMABLE`	Requires email confirmation	`False` (dev mode)

1.2 Security Best Practices in Configuration

```
# Production-ready additions to config.py
class ProductionConfig(Config):
    DEBUG = False
    SECURITY_CONFIRMABLE = True               # Require email confirmation
    SECURITY_SEND_REGISTER_EMAIL = True      # Send registration emails
    SECURITY_PASSWORD_COMPLEXITY_CHECKER = 'app.utils.password_complexity'
    SECURITY_PASSWORD_LENGTH_MIN = 8         # Minimum password length
    WTF_CSRF_ENABLED = True                  # Enable CSRF protection
    SESSION_COOKIE_SECURE = True             # HTTPS only cookies
    SESSION_COOKIE_HTTPONLY = True           # No JavaScript access to cookies
```

Step 2: Understanding the User and Role Models

2.1 Model Architecture Deep Dive

Our `app/models/user.py` implements Flask-Security's required patterns:

```
from flask_security import UserMixin, RoleMixin
from app.extensions import db
import uuid

# Many-to-many relationship table
roles_users = db.Table('roles_users',
    db.Column('user_id', db.Integer(), db.ForeignKey('user.id')),
    db.Column('role_id', db.Integer(), db.ForeignKey('role.id'))
)
```

The Association Table Pattern:

- Stores relationships between users and roles
- Allows users to have multiple roles
- SQLAlchemy manages this automatically

2.2 Role Model Analysis

```
class Role(db.Model, RoleMixin):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), unique=True, nullable=False)
    description = db.Column(db.String(255))
```

RoleMixin provides:

- Required fields for Flask-Security integration
- Standard role management methods
- Integration with permission checking

2.3 User Model Analysis

Our User model includes several categories of fields:

```
class User(db.Model, UserMixin):
    # Basic identity fields
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(255), unique=True, nullable=False)
    password = db.Column(db.String(255), nullable=False)

    # Profile information
    first_name = db.Column(db.String(100))
    last_name = db.Column(db.String(100))

    # Account status
    active = db.Column(db.Boolean, default=True)
    confirmed_at = db.Column(db.DateTime)
    created_at = db.Column(db.DateTime, default=db.func.current_timestamp())

    # Flask-Security required fields
    fs_uniquifier = db.Column(db.String(64), unique=True, nullable=False,
                              default=lambda: str(uuid.uuid4()))

    # Tracking fields (when SECURITY_TRACKABLE=True)
    current_login_at = db.Column(db.DateTime)
    last_login_at = db.Column(db.DateTime)
    current_login_ip = db.Column(db.String(45))
    last_login_ip = db.Column(db.String(45))
    login_count = db.Column(db.Integer, default=0)
```

Field Categories Explained:

1. **Identity Fields**: Core user identification
2. **Profile Fields**: User information for display
3. **Status Fields**: Account management
4. **Security Fields**: Flask-Security requirements
5. **Tracking Fields**: Security auditing

2.4 The fs_uniquifier Field

This field is crucial for Flask-Security:

```
fs_uniquifier = db.Column(db.String(64), unique=True, nullable=False,
                          default=lambda: str(uuid.uuid4()))
```

Purpose:

- Provides a unique identifier separate from user ID
- Used for token generation and validation
- Enhances security by allowing token invalidation
- Required by Flask-Security 4.0+

Step 3: Application Factory and Security Setup

3.1 Extension Initialization

Our `app/extensions.py` centralizes extension objects:

```
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate
from flask_security import Security
from flask_mail import Mail

# Extension instances
db = SQLAlchemy()
migrate = Migrate()
security = Security()
mail = Mail()
```

Benefits of this pattern:

- Avoids circular imports
- Clean separation of concerns
- Easy testing and configuration

3.2 Flask-Security Integration in App Factory

In `app/__init__.py`, Flask-Security is initialized with a UserDatastore:

```
def create_app():
    app = Flask(__name__)
    app.config.from_object(Config)

    # Initialize extensions
    db.init_app(app)
    migrate.init_app(app, db)
    mail.init_app(app)

    # Import models AFTER db initialization
    from app.models.user import User, Role

    # Setup Flask-Security with UserDatastore
    from flask_security import SQLAlchemyUserDatastore
    user_datastore = SQLAlchemyUserDatastore(db, User, Role)
    security.init_app(app, user_datastore)
```

Critical Order of Operations:

1. Initialize Flask app and config
2. Initialize database extension
3. Import models (after db is available)
4. Create UserDatastore
5. Initialize Security extension

3.3 Understanding UserDatastore

The `SQLAlchemyUserDatastore` is Flask-Security's interface to your database:

```
user_datastore = SQLAlchemyUserDatastore(db, User, Role)
```

UserDatastore provides methods for:

- Creating and finding users
- Managing roles and permissions
- Activating/deactivating users
- Password management

Common UserDatastore methods:

```
# In your application code
from flask_security import user_datastore
```

```
# Create a user
user_datastore.create_user(
    email='user@example.com',
    password=hash_password('password'),
    first_name='John'
)

# Find a user
user = user_datastore.find_user(email='user@example.com')

# Add role to user
role = user_datastore.find_role('Citizen')
user_datastore.add_role_to_user(user, role)

# Commit changes
user_datastore.commit()
```

Step 4: Authentication Routes and Protection

4.1 Built-in Flask-Security Routes

Flask-Security automatically provides these endpoints:

Endpoint	Purpose	Template
`/login`	User login form	`security/login_user.html`
`/register`	User registration	`security/register_user.html`
`/logout`	User logout	Redirect
`/reset`	Password reset request	`security/forgot_password.html`
`/reset/<token>`	Password reset form	`security/reset_password.html`
`/change-password`	Change password	`security/change_password.html`
`/confirm/<token>`	Email confirmation	`security/send_confirmation.html`

4.2 Protecting Routes with Decorators

Our existing routes use Flask-Security decorators:

```python
# app/main/views.py
from flask_security import login_required, current_user

@main_bp.route('/dashboard', methods=['GET'])
@login_required  # Requires user to be logged in
def dashboard():
    """Protected dashboard route."""
    return {
        "message": f"Welcome to your dashboard, {current_user.email}!",
        "user_id": current_user.id,
        "user_roles": [role.name for role in current_user.roles],
        "status": "success"
    }, 200
```

Available Protection Decorators:

```python
from flask_security import (
    login_required,      # User must be logged in
    roles_required,      # User must have specific role(s)
    roles_accepted,      # User must have ANY of the listed roles
    permissions_required, # User must have specific permission
    auth_required        # Token-based authentication
)

# Example usage:
@roles_required('County Admin')
def admin_only_view():
    pass

@roles_accepted('Department Officer', 'County Admin')
def officer_or_admin_view():
    pass

@auth_required('token')  # For API endpoints
def api_endpoint():
    pass
```

4.3 The current_user Object

Flask-Security provides `current_user` globally:

```python
from flask_security import current_user

# Check if user is authenticated
if current_user.is_authenticated:
```

```
    print(f"User {current_user.email} is logged in")

# Access user properties
user_id = current_user.id
user_email = current_user.email
user_roles = [role.name for role in current_user.roles]

# Check for specific roles
if current_user.has_role('County Admin'):
    print("User is an admin")

# Check if user is active
if current_user.is_active:
    print("User account is active")
```

current_user Properties:

- `is_authenticated`: True if user is logged in
- `is_active`: True if user account is active
- `is_anonymous`: True if user is not logged in
- `id`: User's database ID
- `email`: User's email address
- `roles`: List of user's roles

Step 5: Role-Based Access Control (RBAC)

5.1 Understanding RBAC in Flask-Security

Role-Based Access Control allows you to:

- Assign roles to users
- Control access based on roles
- Create hierarchical permission systems

Our predefined roles (from the County Services project):

```
# Three roles for our system
roles = [
    {"name": "Citizen", "description": "Regular citizen user"},
    {"name": "Department Officer", "description": "Officer in specific departments"},
```

```
    {"name": "County Admin", "description": "County administrator with full access"}
]
```

5.2 Implementing Role Checks in Routes

```
# app/auth/routes.py example
from flask_security import login_required, roles_required, current_user

@auth_bp.route('/users', methods=['GET'])
@login_required
@roles_required('County Admin')  # Only County Admins can access
def list_users():
    """List all users - admin only."""
    users = User.query.all()
    # ... rest of implementation
```

5.3 Conditional Access Based on User Properties

Sometimes you need more complex access control:

```
@main_bp.route('/dashboard', methods=['GET'])
@login_required
def dashboard():
    """Dashboard with role-based content."""
    user_roles = [role.name for role in current_user.roles]

    if 'County Admin' in user_roles:
        # Admin sees everything
        data = get_admin_dashboard_data()
    elif 'Department Officer' in user_roles:
        # Officer sees department-specific data
        data = get_officer_dashboard_data(current_user.department_id)
    else:
        # Citizens see personal data only
        data = get_citizen_dashboard_data(current_user.id)

    return {"data": data, "user_roles": user_roles}
```

5.4 Template-Based Role Checking

When you add templates, you can check roles there too:

```
<!-- Example template usage -->
{% if current_user.has_role('County Admin') %}
    <a href="/admin-panel">Admin Panel</a>
{% endif %}

{% if current_user.has_role('Department Officer') or current_user.has_role('County Admin'
    <a href="/manage-permits">Manage Permits</a>
{% endif %}
```

Step 6: Database Initialization and User Management

6.1 Understanding init_db.py

Our initialization script creates the database schema and default data:

```
#!/usr/bin/env python3
"""Initialize database with basic roles and a test user."""

from app import create_app
from app.extensions import db
from app.models.user import User, Role
from flask_security.utils import hash_password
import uuid

def init_database():
    """Initialize database with basic roles and test user."""
    app = create_app()

    with app.app_context():
        # Create tables
        db.create_all()

        # Create roles if they don't exist
        roles_data = [
            {"name": "Citizen", "description": "Regular citizen user"},
            {"name": "Department Officer", "description": "Officer in departments"},
            {"name": "County Admin", "description": "County administrator"}
        ]

        for role_data in roles_data:
            role = Role.query.filter_by(name=role_data["name"]).first()
            if not role:
                role = Role(**role_data)
```

```
                db.session.add(role)

        # Create test admin user
        admin_role = Role.query.filter_by(name="County Admin").first()
        admin_user = User.query.filter_by(email="admin@test.com").first()

        if not admin_user:
            admin_user = User(
                email="admin@test.com",
                password=hash_password("password123"),
                first_name="Admin",
                last_name="User",
                fs_uniquifier=str(uuid.uuid4()),
                active=True
            )
            admin_user.roles.append(admin_role)
            db.session.add(admin_user)

        db.session.commit()
        print("Database initialized successfully!")
```

Key Functions:

- `hash_password()`: Securely hashes passwords
- `str(uuid.uuid4())`: Generates unique identifiers
- Role and user creation with proper relationships

6.2 Password Security

Flask-Security handles password hashing automatically:

```
from flask_security.utils import hash_password, verify_password

# Hash a password
hashed = hash_password('plaintext_password')

# Verify a password
is_valid = verify_password('plaintext_password', hashed)
```

Security features:

- Uses bcrypt by default
- Includes salt for additional security
- Configurable hash complexity

- Automatic rehashing of weak passwords

Step 7: API Authentication and JSON Responses

7.1 API Endpoints with Authentication

Our existing API routes demonstrate authentication patterns:

```
# app/auth/routes.py
@auth_bp.route('/current-user', methods=['GET'])
@login_required
def get_current_user():
    """Get current authenticated user information."""
    return {
        "user": {
            "id": current_user.id,
            "email": current_user.email,
            "first_name": current_user.first_name,
            "last_name": current_user.last_name,
            "full_name": current_user.full_name,
            "active": current_user.active,
            "roles": [role.name for role in current_user.roles],
            "created_at": current_user.created_at.isoformat() if current_user.created_at
        },
        "status": "success"
    }, 200
```

7.2 Token-Based Authentication

For API clients, you can use token authentication:

```
from flask_security import auth_required

@api_bp.route('/protected-endpoint', methods=['GET'])
@auth_required('token')  # Requires authentication token
def protected_api():
    return {"message": "This requires a token", "user": current_user.email}
```

Token generation for users:

```
from flask_security.utils import get_token_status

# Generate authentication token for user
token = user.get_auth_token()

# Verify token
user, status = get_token_status(token)
```

7.3 CSRF Protection for Forms

Flask-Security includes CSRF protection:

```
# In config.py
WTF_CSRF_ENABLED = True
SECURITY_CSRF_PROTECT_MECHANISMS = ['form', 'basic']
```

In forms:

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, SubmitField
from wtforms.validators import DataRequired, Email

class LoginForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Email()])
    password = PasswordField('Password', validators=[DataRequired()])
    submit = SubmitField('Sign In')
    # CSRF token is automatically included
```

Step 8: Testing Your Authentication System

8.1 Manual Testing Workflow

1. **Start the application:**

   ```
   python run.py
   ```

2. **Test registration:**

```
curl -X POST http://127.0.0.1:5000/register \
     -H "Content-Type: application/json" \
     -d '{"email":"test@example.com", "password":"password123"}'
```

3. **Test login:**

```
curl -X POST http://127.0.0.1:5000/login \
     -H "Content-Type: application/json" \
     -d '{"email":"test@example.com", "password":"password123"}' \
     -c cookies.txt
```

4. **Test protected route:**

```
curl -X GET http://127.0.0.1:5000/dashboard \
     -b cookies.txt
```

8.2 Testing with Python Requests

```
import requests

session = requests.Session()

# Login
login_data = {"email": "admin@test.com", "password": "password123"}
response = session.post("http://127.0.0.1:5000/login", json=login_data)

# Access protected route
dashboard_response = session.get("http://127.0.0.1:5000/dashboard")
print(dashboard_response.json())

# Check current user
user_response = session.get("http://127.0.0.1:5000/auth/current-user")
print(user_response.json())
```

8.3 Database Verification

Check your database to see created users:

```
# In Flask shell: flask shell
from app.models.user import User, Role

# List all users
```

```
users = User.query.all()
for user in users:
    print(f"User: {user.email}, Roles: {[r.name for r in user.roles]}")

# Check specific user
admin = User.query.filter_by(email='admin@test.com').first()
print(f"Admin active: {admin.active}")
print(f"Admin roles: {[r.name for r in admin.roles]}")
```

Step 9: Advanced Security Features

9.1 Session Management

Flask-Security provides robust session management:

```
# In config.py
SESSION_COOKIE_SECURE = True      # HTTPS only (production)
SESSION_COOKIE_HTTPONLY = True    # No JavaScript access
SESSION_COOKIE_SAMESITE = 'Lax'   # CSRF protection
PERMANENT_SESSION_LIFETIME = 3600  # Session timeout (seconds)
```

9.2 Login Tracking

With `SECURITY_TRACKABLE = True`, Flask-Security tracks:

```
# User login information is automatically stored
print(f"User login count: {current_user.login_count}")
print(f"Last login: {current_user.last_login_at}")
print(f"Last IP: {current_user.last_login_ip}")
```

9.3 Account Activation/Deactivation

```
from flask_security import user_datastore

# Deactivate user
user = User.query.filter_by(email='user@example.com').first()
user_datastore.deactivate_user(user)
user_datastore.commit()

# Reactivate user
```

```
user_datastore.activate_user(user)
user_datastore.commit()
```

9.4 Password Policies

Implement password complexity requirements:

```
# Custom password validator
def password_complexity(password):
    """Custom password complexity checker."""
    if len(password) < 8:
        return "Password must be at least 8 characters long"
    if not any(c.isupper() for c in password):
        return "Password must contain at least one uppercase letter"
    if not any(c.islower() for c in password):
        return "Password must contain at least one lowercase letter"
    if not any(c.isdigit() for c in password):
        return "Password must contain at least one digit"
    return None

# In config.py
SECURITY_PASSWORD_COMPLEXITY_CHECKER = 'app.utils.password_complexity'
```

Step 10: Error Handling and User Experience

10.1 Custom Error Pages

Create templates for authentication errors:

```
# app/__init__.py - Add error handlers
@app.errorhandler(401)
def unauthorized(error):
    return {"error": "Authentication required", "status": 401}, 401

@app.errorhandler(403)
def forbidden(error):
    return {"error": "Access forbidden", "status": 403}, 403
```

10.2 Flash Messages

Flask-Security provides flash messages for user feedback:

```python
from flask import flash

# Custom messages in your routes
@main_bp.route('/custom-action')
@login_required
def custom_action():
    # Perform action
    flash('Action completed successfully!', 'success')
    return redirect(url_for('main.dashboard'))
```

10.3 Redirects After Login

Configure where users go after login:

```python
# In config.py
SECURITY_POST_LOGIN_REDIRECT_ENDPOINT = 'main.dashboard'
SECURITY_POST_LOGOUT_REDIRECT_ENDPOINT = 'main.home'
```

Step 11: Production Considerations

11.1 Environment Variables

Create a production `.env` file:

```
# Production .env
SECRET_KEY=your-very-long-random-secret-key-here
DATABASE_URL=postgresql://user:password@localhost/county_services
SECURITY_PASSWORD_SALT=your-password-salt-here

# Email configuration
MAIL_SERVER=smtp.yourdomain.com
MAIL_PORT=587
MAIL_USE_TLS=true
MAIL_USERNAME=noreply@yourdomain.com
MAIL_PASSWORD=your-email-password

# Security settings
SECURITY_CONFIRMABLE=true
SECURITY_SEND_REGISTER_EMAIL=true
```

11.2 Database Migration

When deploying, use Flask-Migrate:

```
# Create migration
flask db migrate -m "Add user tracking fields"

# Apply migration
flask db upgrade
```

11.3 Security Headers

Add security headers for production:

```
# app/__init__.py
from flask_talisman import Talisman

def create_app():
    app = Flask(__name__)

    # Security headers
    Talisman(app, force_https=True)

    @app.after_request
    def after_request(response):
        response.headers['X-Content-Type-Options'] = 'nosniff'
        response.headers['X-Frame-Options'] = 'DENY'
        response.headers['X-XSS-Protection'] = '1; mode=block'
        return response
```

Step 12: Extending the Authentication System

12.1 Adding Custom User Fields

Extend the User model for your application:

```
# app/models/user.py additions
class User(db.Model, UserMixin):
    # ... existing fields ...

    # County Services specific fields
    county_id = db.Column(db.Integer, db.ForeignKey('county.id'))
    department_id = db.Column(db.Integer, db.ForeignKey('department.id'))
    phone_number = db.Column(db.String(20))
```

```
    address = db.Column(db.Text)

    # Relationships
    county = db.relationship('County', backref='users')
    department = db.relationship('Department', backref='officers')
```

12.2 Custom Registration Form

Create custom forms for additional fields:

```
# app/forms.py
from flask_security.forms import RegisterForm
from wtforms import StringField, SelectField
from wtforms.validators import Optional

class ExtendedRegisterForm(RegisterForm):
    first_name = StringField('First Name')
    last_name = StringField('Last Name')
    phone_number = StringField('Phone Number', validators=[Optional()])
    county = SelectField('County', coerce=int)

# In config.py
SECURITY_REGISTER_FORM = 'app.forms.ExtendedRegisterForm'
```

12.3 Role Assignment Logic

Implement automatic role assignment:

```
# app/utils.py
def assign_default_role(user):
    """Assign default role to new users."""
    from flask_security import user_datastore
    from app.models.user import Role

    citizen_role = Role.query.filter_by(name='Citizen').first()
    if citizen_role:
        user_datastore.add_role_to_user(user, citizen_role)
        user_datastore.commit()

# Use in registration process
@app.after_request
def after_register(sender, user, **extra):
    assign_default_role(user)
```

Summary: What We've Accomplished

✅ Core Authentication Features

1. **User Registration & Login** - Secure user accounts with email verification
2. **Password Security** - Bcrypt hashing with salt
3. **Role-Based Access Control** - Three-tier role system (Citizen, Officer, Admin)
4. **Session Management** - Secure cookie-based sessions
5. **Protected Routes** - Decorator-based route protection
6. **API Authentication** - Token-based API access
7. **User Tracking** - Login history and security auditing

✅ Security Best Practices

1. **CSRF Protection** - Automatic form protection
2. **Password Policies** - Configurable complexity requirements
3. **Session Security** - HttpOnly, Secure, SameSite cookies
4. **Input Validation** - WTForms integration
5. **Error Handling** - Proper error responses
6. **Database Security** - ORM protection against SQL injection

✅ Development Features

1. **Modular Structure** - Blueprint-based organization
2. **Configuration Management** - Environment-based config
3. **Database Migrations** - Version-controlled schema changes
4. **Testing Ready** - Easy unit and integration testing
5. **Production Ready** - Scalable architecture

Next Steps: Building on This Foundation

Immediate Enhancements (Lesson 3)

1. **Templates** - HTML templates for web interface
2. **Forms** - WTForms integration for user input
3. **County/Department Models** - Organizational structure

4. **Profile Management** - User profile editing

Advanced Features (Future Lessons)

1. **Email Integration** - Registration confirmation and notifications
2. **API Documentation** - Swagger/OpenAPI integration
3. **Audit Logging** - Comprehensive activity tracking
4. **Two-Factor Authentication** - Enhanced security
5. **Social Login** - OAuth integration
6. **Permission System** - Fine-grained access control

County Services Specific (Final Project)

1. **Permit Application Workflow** - Multi-step forms
2. **Department Assignment** - Officer-to-department mapping
3. **County Scoping** - Multi-tenant architecture
4. **Document Management** - File upload and storage
5. **Reporting System** - Analytics and dashboards

Practice Exercises

Exercise 1: Role Testing

1. Create a new user via the API
2. Assign different roles to the user
3. Test access to role-protected endpoints
4. Verify role-based responses

Exercise 2: Custom Decorators

Create a custom decorator that combines role checking with county verification:

```
def county_role_required(role_name):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            # Check if user has role
            if not current_user.has_role(role_name):
                abort(403)
```

```
            # Check if user belongs to a county
            if not current_user.county_id:
                abort(403, "User must be assigned to a county")

            return f(*args, **kwargs)
        return decorated_function
    return decorator
```

Exercise 3: Password Reset Flow

1. Implement password reset email sending
2. Create reset token handling
3. Test the complete reset workflow

Exercise 4: User Management API

Create API endpoints for:

- Listing users with pagination
- Updating user profiles
- Deactivating/reactivating users
- Assigning roles programmatically

Troubleshooting Common Issues

1. "Table doesn't exist" errors

```
# Solution: Initialize the database
python init_db.py
```

2. "fs_uniquifier not found" errors

```
# Solution: Update existing users
for user in User.query.all():
    if not user.fs_uniquifier:
        user.fs_uniquifier = str(uuid.uuid4())
db.session.commit()
```

3. CSRF token errors

```
# Solution: Ensure CSRF is properly configured
WTF_CSRF_ENABLED = True
WTF_CSRF_SECRET_KEY = 'your-csrf-secret'
```

4. Session not persisting

```
# Solution: Check secret key configuration
SECRET_KEY = 'your-secret-key-must-be-consistent'
```

Additional Resources

- Flask-Security Documentation
- Flask-SQLAlchemy Relationships
- OWASP Authentication Guidelines
- Flask Security Best Practices
- Bcrypt Password Hashing

Remember: Authentication is just the foundation. In real applications, you'll also need authorization logic, audit trails, and comprehensive testing. The modular structure we've built makes adding these features straightforward and maintainable!