TP n°4 *Formulaires, Authentification*

Dans ce TP vous allez utiliser WTForms pour fabriquer des formulaires. Alors que SQLAlchemy permet de modéliser des tables avec des classes, WTForms nous permet de modéliser des formulaires avec des classes.

Exercice 1. Installation de `flask-wtf`

Pour commencer, vous devez installer le plugin `flask-wtf` :

```
$ source venv/bin/activate
(venv) $ pip install flask-wtf
```

Aucune activation n'est nécessaire, par contre il faut configurer notre app avec une clé secrète pour sécuriser les interactions et se protéger d'attaques CSRF (Cross-Site Request Forgery). Pour générer une clé secrète impossible à deviner je vous conseille d'utiliser la commande `uuidgen` :

```
(venv) $ uuidgen
bcc090e2-26b2-4c16-84ab-e766cc644320
```

vous obtiendrez bien sûr une autre valeur ! Maintenant, éditez le fichier `app.py` pour ajouter la ligne suivante :

tuto/app.py

```
app.config['SECRET_KEY'] = "bcc090e2-26b2-4c16-84ab-e766cc644320"
```

Exercice 2. Formulaire pour éditer un auteur

Dans le fichier `views.py` nous allons définir le formulaire `AuthorForm` et l'utiliser dans une vue `edit_author` :

tuto/views.py

```
from flask_wtf import FlaskForm
from wtforms import StringField, HiddenField
from wtforms.validators import DataRequired

class AuthorForm(FlaskForm):
    id   = HiddenField('id')
    name = StringField('Nom', validators=[DataRequired()])
```

```
@app.route("/edit/author/<int:id>")
def edit_author(id):
    a = get_author(id)
    f = AuthorForm(id=a.id, name=a.name)
    return render_template(
        "edit-author.html",
        author=a, form=f)
```

Vous devez ensuite écrire le template qui va avec :

tuto/templates/edit-author.html

```
{% extends "base.html" %}
{% block main %}
<h1>Edit author: {{ author.name }}</h1>
<form method="POST" action="">
    {{ form.hidden_tag() }}
    {{ form.name.label }} {{ form.name(size=50) }}
    <input type="submit" value="Enregistrer">
</form>
{% endblock %}
```

`form.hidden_tag()` insère un `div` contenant tous les champs cachés du formulaire ; ici il y en a 2 : le champ `id` et le champ pour le token CSRF qui protège l'action du formulaire contre les contrefaçons.

Pour l'instant, nous ne mettons pas d'URL dans `action` parce que nous n'avons pas écrit la fonction pour sauvegarder le résultat de l'édition.

Vérifiez que le formulaire est bien accessible. Committez !

Exercice 3. Adaptation du formulaire à bootstrap

Nous éditons un peu le template :

tuto/templates/edit-author.html

```
{% extends "base.html" %}
{% block main %}
<h1>Edit author: {{ author.name }}</h1>
<form role="form" method="POST" action="">
  {{ form.hidden_tag() }}
```

```
    <div class="form-group">
      {{ form.name.label }} {{ form.name(size=50,class_="form-control") }}
    </div>
    <input class="btn btn-default" type="submit" value="Enregistrer">
  </form>
  {% endblock %}
```

Le paramètre `class_="form-control"` indique qu'on veut ajouter la classe `form-control` au widget `input` (voir la doc de bootstrap sur les formulaires). l'API utilise `class_` avec un underscore car **`class`** est un mot clé et ne peut être utilisé comme un identifiant normal.

Dans votre navigateur, visitez le formulaire d'édition et regardez le code HTML source de la page pour constater ce qui a été généré pour le formulaire. Commitez !

Exercice 4. Mise en place de l'action pour enregistrer la modif

Dans le fichier `views.py` :

tuto/views.py

```
from flask import url_for, redirect
from .app import db
from .models import Author

@app.route("/save/author/", methods=("POST",))
def save_author():
    a = None
    f = AuthorForm()
    if f.validate_on_submit():
        id = int(f.id.data)
        a = get_author(id)
        a.name = f.name.data
        db.session.commit()
        return redirect(url_for('one_author', id=a.id))
    a = get_author(int(f.id.data))
    return render_template(
        "edit-author.html",
        author=a, form=f)
```

Si nous avons mis à jour l'auteur, alors nous faisons une redirection vers sa page (dans **mon** code, cette page correspond à la fonction `one_author`). Sinon, nous renvoyons l'utilisateur vers la page du formulaire pour qu'il entre des données valides.

Vous devez aussi éditer le template pour mettre en place l'action :

tuto/templates/edit-author.html

```
...
<form role="form" method="POST"
      action="{{ url_for('save_author') }}">
...
```

Vérifiez que les différents cas possibles fonctionnent correctement. Commitez !

Exercice 5. Affichage des erreurs de validation

Nous ajoutons maintenant le support pour visualiser les erreurs de validation. Quand un champ d'un formulaire a des erreurs de validation, l'attribut `errors` contient une liste de messages d'erreur qu'on voudra afficher par exemple en dessous du widget. Il suffit pour cela de modifier un peu le template `edit-author.html` :

tuto/templates/edit-author.html

```
{% extends "base.html" %}
{% block main %}
<h1>Edit author: {{ author.name }}</h1>
<form role="form" method="POST"
       action="{{ url_for('save_author') }}">
  {{ form.hidden_tag() }}
  <div class=
       {% if form.name.errors %}
       "form-group has-error"
       {% else %}
       "form-group"
       {% endif %}
       >
       {{ form.name.label }} {{ form.name(size=50,
                                          class_="form-control") }}
  </div>
  {% if form.name.errors %}
  <ul class="list-group">
    {% for e in form.name.errors %}
    <li class="list-group-item list-group-item-danger">{{ e }}</li>
    {% endfor %}
  </ul>
  {% endif %}
```

```
  <input class="btn␣btn-default" type="submit" value="Enregistrer">
</form>
{% endblock %}
```

Vérifiez le bon fonctionement. Commitez !

Exercice 6. Ajout d'un auteur

Faites la même chose que précédemment, mais cette fois-ci, pour ajouter un nouvel auteur (donc dont l'id est `None` initialement). Pour sauvegarder ce nouvel auteur, vous devrez faire comme dans la commande `loaddb` : créer une nouvelle instance d'Author, puis l'ajouter à la transaction, puis commiter la transaction.

Exercice 7. Installation de `flask-login`

Nous allons maintenant ajouter l'authentification des utilisateurs de notre app. **Attention :** le serveur de développement sert les pages avec HTTP, c'est à dire que la connexion est **non sécurisée** et que les mots de passe vont donc circuler en clair ! *Il ne faut jamais faire cela en déploiement ! Il faut impérativement utiliser HTTPS !*

```
(venv) $ pip install flask-login
```

Exercice 8. Ajout du modèle `User`

Nous allons devoir ajouter un modèle "utilisateur" dans la base de données avec un identifiant et un mot de passe crypté avec SHA256 :

tuto/models.py

```
class User(db.Model):
    username = db.Column(db.String(50), primary_key=True)
    password = db.Column(db.String(64))
```

Quand on lit la doc pour `flask-login` (faite-le !), on apprend que la classe `User` doit offrir une certaine API qu'on peut obtenir, partiellement implémentée, par la classe`UserMixin`. On fait donc les modifications suivantes :

```
from flask_login import UserMixin

class User(db.Model, UserMixin):
    username = db.Column(db.String(50), primary_key=True)
```

```
    password = db.Column(db.String(64))

    def get_id(self):
        return self.username
```

Commitez !

Exercice 9. Création de nouvelles tables

Nous venons d'ajouter un modèle, mais la table correspondante n'existe pas encore dans la base de données. C'est la fonction `db.create_all()` qui permet de créer les tables manquantes. Nous allons donc ajouter une commande `syncdb` pour invoquer cette fonction :

tuto/commands.py

```
@app.cli.command()
def syncdb():
    '''Creates all missing tables.'''
    db.create_all()
```

Maintenant, on peut l'utiliser :

```
(venv) $ flask syncdb
```

On peut alors vérifier que la table `user` a bien été créée :

```
(venv) $ sqlite3 myapp.db
SQLite version 3.8.7.2 2014-11-18 20:57:56
Enter ".help" for usage hints.
sqlite> .tables
author  book    user
sqlite>
```

Commitez !

Exercice 10. Ajout d'utilisateurs

Pour ajouter des utilisateurs dans notre base de données, nous allons réaliser une commande `newuser` :

tuto/commands.py

```
@app.cli.command()
@click.argument('username')
@click.argument('password')
def newuser(username, password):
    '''Adds a new user.'''
    from .models import User
    from hashlib import sha256
    m = sha256()
    m.update(password.encode())
    u = User(username=username, password=m.hexdigest())
    db.session.add(u)
    db.session.commit()
```

Ajoutons maintenant un utilisateur `denys` avec mot de passe `foo` :

```
(venv) $ flask newuser denys foo
```

Commitez !

Exercice 11. Commande pour changer le mot de passe d'un utilisateur.

Ajoutez une commande `passwd` pour changer le mot de passe d'un utilisateur :

```
(venv) $ flask passwd denys baz
```

Commitez !

Exercice 12. Activation du plugin

tuto/app.py

```
from flask_login import LoginManager
login_manager = LoginManager(app)
```

Puis nous devons fournir un "callback" pour charger un utilisateur étant donné son identifiant. Ce callback sera utilisé par flask pour l'instance de `User` d'un utilisateur authentifié, étant donné son cookie de session :

tuto/models.py

```
from .app import login_manager

@login_manager.user_loader
def load_user(username):
    return User.query.get(username)
```

Commitez !

Exercice 13. Mise en place d'un formulaire d'authentification

Dans `views.py`, vous devez créer un formulaire pour permettre à un utilisateur de s'authentifier. On lui adjoindra une méthode `get_authenticated_user` qui vérifie que l'utilisateur existe, et que son mot de passe est correct et renvoie l'instance de `User` représentant cet utilisateur. Si l'utilisateur n'existe pas ou que son mot de passe est faux, il renvoie `None`.

tuto/views.py

```
from wtforms import PasswordField
from .models import User
from hashlib import sha256

class LoginForm(FlaskForm):
    username = StringField('Username')
    password = PasswordField('Password')

    def get_authenticated_user(self):
        user = User.query.get(self.username.data)
        if user is None:
            return None
        m = sha256()
        m.update(self.password.data.encode())
        passwd = m.hexdigest()
        return user if passwd == user.password else None
```

Maintenant que nous avons le formulaire, il faut créer une vue pour l'utiliser :

tuto/views.py

```
from flask_login import login_user, current_user
from flask import request
```

```
@app.route("/login/", methods=("GET","POST",))
def login():
    f = LoginForm()
    if f.validate_on_submit():
        user = f.get_authenticated_user()
        if user:
            login_user(user)
            return redirect(url_for("home"))
    return render_template(
        "login.html",
        form=f)
```

Enfin il nous faut le template correspondant (que j'ai emprunté à un example de la doc de bootstrap et légèrement adapté) :

tuto/templates/login.html

```
{% extends "base.html" %}
{% block main %}
<form class="form-horizontal" role="form" method="POST"
      action="{{ url_for('login') }}">
  {{ form.hidden_tag() }}
  <div class="form-group">
    <label for="username" class="col-sm-2 control-label"
           >{{ form.username.label }}</label>
    <div class="col-sm-10">
      {{ form.username(size=50, class_="form-control",
                       placeholder="Username") }}
    </div>
  </div>
  <div class="form-group">
    <label for="password" class="col-sm-2 control-label"
           >{{ form.password.label }}</label>
    <div class="col-sm-10">
      {{ form.password(size=50, class_="form-control",
                       placeholder="Password") }}
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-10">
      <button type="submit" class="btn btn-default">Sign in</button>
    </div>
  </div>
```

```
</form>
{% endblock %}
```

Pour que l'utilisateur puisse constater qu'il est bien loggé, on va ajouter son nom, à droite dans la barre de navigation du template de base, ainsi qu'une icone pour le logout. Donc dans le , on va rajouter à la fin :

tuto/templates/base.html

```
<ul class="nav␣navbar-nav␣navbar-right">
  {% if current_user.is_authenticated() %}
  <li><a>{{ current_user.username }}</a></li>
  <li><a href="{{␣url_for('logout')␣}}">
      <span class="glyphicon␣glyphicon-log-out" title="Logout"
          ></span></a></li>
  {% endif %}
</ul>
```

et pour que le logout fonctionne, il faut rajouter une vue de logout :

tuto/views.py

```
from flask_login import logout_user

@app.route("/logout/")
def logout():
    logout_user()
    return redirect(url_for('home'))
```

Vérifiez que login et logout fonctionnent. Commitez !

Exercice 14. Protection des vues nécessitant une anthentification

Dans le template de page d'un auteur, nous ne mettrons le lien permettant d'accéder à la page d'édition que si l'utilisateur est authentifié.

tuto/templates/author.html

```
...
{% if current_user.is_authenticated %}
<a href="{{␣url_for('edit_author',␣id=author.id)␣}}">Edit</a>
{% endif %}
...
```

Attention ! Ceci n'est pas suffisant car un utilisateur pourrait taper l'url de la page d'édition d'un auteur dans la barre d'adresse de son navigateur web. Il faut donc protéger la vue d'édition. On peut le faire très facilement avec un décorateur :

tuto/views.py

```
from flask_login import login_required

@app.route("/edit/author/<int:id>")
@login_required
def edit_author(id):
    ...
```

Le premier décorateur `@login_required` transforme la fonction de vue `edit_required` en une nouvelle fonction qui n'authorise l'accès que si l'utilisateur est authentifié. Le second décorateur `@app.route(...)` associe cette nouvelle fonction à la route.

Exercice 15. Mise en place de la redirection automatique

Pour rendre les choses plus pratiques, on voudrait que lorsque, lorsqu'une vue est ainsi protégée et que l'utilisateur n'est pas encore authentifié, il soit automatiquement redirigé vers la vue de login. Ceci peut être réalisé très simplement par la configuration suivante :

tuto/app.y

```
login_manager.login_view = "login"
```

On informe ainsi le `login_manager` du nom de la fonction réalisant la vue de login. Dans notre cas, Cette fonction s'appelle `login`.

Une fois que l'utilisateur s'est authentifié, il faudrait qu'il soit redirigé vers la page à laquelle il tentait initialement d'accéder. Lors de la redirection initiale vers la page de login, l'url d'où on vient est précisé par le paramètre `next` de la requète.

Il faudrait se souvenir de ce paramètre dans notre formulaire de login pour pouvoir, après un login réussi, rediriger l'utilisateur vers la page à laquelle il tentait initialement d'accéder. On va donc ajouter un champ dans notre formulaire :

tuto/views.py

```
class LoginForm(FlaskForm):
    username = StringField('Username')
    password = PasswordField('Password')
    next = HiddenField()
    ...
```

et on modifie notre vue login de la manière suivante :

tuto/views.py

```
@app.route("/login/", methods=("GET","POST",))
def login():
    f = LoginForm()
    if not f.is_submitted():
        f.next.data = request.args.get("next")
    elif f.validate_on_submit():
        user = f.get_authenticated_user()
        if user:
            login_user(user)
            next = f.next.data or url_for("home")
            return redirect(next)
    return render_template(
        "login.html",
        form=f)
```

vérifiez que les redirections fonctionnent.