

Flask

Авторизация пользователя и начало работы с базой данных. SQLAlchemy



На этом уроке

1. Узнаем, что такое OWASP.
2. Познакомимся с библиотекой SQLAlchemy.
3. Научимся подключать SQLAlchemy к проекту через расширение Flask-SQLAlchemy.
4. Узнаем, как работает сессия и авторизация.
5. Создадим модель пользователя.

Оглавление

Теория

[SQLAlchemy](#)

[Open Web Application Security Project](#)

Практическое задание

[Установка и настройка Flask-SQLAlchemy](#)

[Создаём модель пользователя](#)

[Создаём команды для базы данных](#)

[Flask-Login](#)

[Добавляем UserMixin к модели пользователя](#)

[Подключаем блупринт авторизации и LoginManager](#)

Итоги

Практическое задание

Дополнительные материалы

Теория

SQLAlchemy

Программная библиотека на языке Python для работы с реляционными СУБД с применением технологии ORM. Служит для синхронизации объектов Python и записей реляционной базы данных. SQLAlchemy позволяет описывать структуры баз данных и способы взаимодействия с ними на языке Python без использования SQL.

Система управления базами данных, СУБД — совокупность программных и лингвистических средств общего или специального назначения, обеспечивающих управление созданием и использованием баз данных.

ORM — технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных».

SQL — декларативный язык программирования, применяемый для создания, модификации и управления данными в реляционной базе данных, управляемой соответствующей СУБД.

Возможности:

- использование ORM не является обязательным;
- устоявшаяся архитектура;
- возможность использовать SQL, написанный вручную;
- поддержка транзакций;
- создание запросов с использованием функций и выражений Python;
- модульность и расширяемость;
- дополнительная возможность раздельного определения объектного отображения и классов;
- поддержка составных индексов;
- поддержка отношений между классами, в том числе «один-ко-многим» и «многие-ко-многим»;
- поддержка ссылающихся на себя объектов;
- предварительная и последующая обработка данных (параметров запроса, результата).

Open Web Application Security Project

Open Web Application Security Project — это открытый проект обеспечения безопасности веб-приложений. Сообщество OWASP включает в себя корпорации, образовательные организации и частных лиц со всего мира.

Практическое задание

Установка и настройка Flask-SQLAlchemy

```
pip install Flask-SQLAlchemy
```

Создаём модуль models в модуле blog: `blog/models/__init__.py`

Инициализируем в модуле `blog/models/database.py` объект SQLAlchemy для работы с БД

```
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

__all__ = [
    "db",
]
```

В модуле блога в уже знакомом файле `app.py` указываем конфигурацию подключения к БД и инициализируем объект `db` для работы с базой данных:

```
from blog.models.database import db

app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///tmp/blog.db"
app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False
db.init_app(app)
```

`SQLALCHEMY_DATABASE_URI`: URI для подключения к БД. Например:

- `sqlite:///tmp/test.db`
- `mysql://username:password@server/db`

`SQLALCHEMY_TRACK_MODIFICATIONS`: Если установлено значение `True`, Flask-SQLAlchemy будет отслеживать изменения объектов и отправлять сигналы. По умолчанию используется значение `None`, которое включает отслеживание, но выдает предупреждение о том, что в будущем оно будет отключено по умолчанию. Это требует дополнительной памяти и должно быть отключено, если в этом нет необходимости. Поэтому отключаем явно.

Создаём модель пользователя

В `blog/models/user.py` объявляем модель `User` с необходимыми полями:

```
from sqlalchemy import Column, Integer, String, Boolean
from blog.models.database import db

class User(db.Model):
    id = Column(Integer, primary_key=True)
    username = Column(String(80), unique=True, nullable=False)
    is_staff = Column(Boolean, nullable=False, default=False)

    def __repr__(self):
        return f"<User #{self.id} {self.username!r}>"
```

- создаём декларативную модель, наследуемся от предоставляемой базы `db.Model`;
- добавляем колонки. Имена атрибутов будут использованы для именования колонок в БД;
- свойства колонок будут использованы для управления свойствами колонок в БД;
- добавляем функцию для отображения репрезентативного вида.

Импортируем его в `blog/models/__init__.py` по двум причинам:

- проще импортировать все модели из модуля `models`;
- при инициализации модуля все модели будут созданы, метадата алхимии узнает обо всех моделях, и миграции будут корректно отслеживать изменения.

```
from blog.models.user import User

__all__ = [
    "User",
]
```

Создаём команды для базы данных

Команды две: `init_db`, нужная для инициализации базы при первом запуске, и `create_users`, которая поможет нам создавать пользователей.

```
@app.cli.command("init-db")
def init_db():
    """
    Run in your terminal:
    flask init-db
    """
    db.create_all()
    print("done!")

@app.cli.command("create-users")
def create_users():
    """
    Run in your terminal:
    flask create-users
    > done! created users: <User #1 'admin'> <User #2 'james'>
    """
    from blog.models import User
    admin = User(username="admin", is_staff=True)
    james = User(username="james")

    db.session.add(admin)
    db.session.add(james)
    db.session.commit()

    print("done! created users:", admin, james)
```

Переходим в папку с проектом (где лежит файл `wsgi.py`) и выполняем команды для создания таблиц в базе данных и добавления туда пользователей:

- `flask init-db`
- `flask create-users`

Теперь надо научить список пользователей обращаться к базе данных, чтобы при открытии страницы показывать актуальный список пользователей.

В `blog/templates/users/details.html` обновляем `title` страницы

```
{% block title %}
  User #{{ user.id }}
{% endblock %}
```

И обновляем body

```
<h1>
    {{ user.username }}
</h1>
{% if user.is_staff %}
    <span class="badge bg-secondary">staff</span>
{% endif %}
```

Далее меняем `blog/templates/users/list.html`. Тут мы итерируемся по переданным пользователям (ниже отредактируем view пользователей и добавим туда передачу пользователей из БД), добавляем опциональный бейджик, что пользователь является админом:

```
{% for user in users %}
<li>
    <a href="{{ url_for('users_app.details', user_id=user.id) }}">
    {{ user.username -}}
    </a>
    {% if user.is_staff %}
        <span class="badge bg-secondary">staff</span>
    {% endif %}
```

И самое главное — редактируем вьюшки в `blog/views/users.py`: отправляем запрос в базу данных, чтобы вытащить всех пользователей для страницы списка. И ищем пользователя по переданному ID для отрисовки страницы пользователя.

```
from blog.models import User

users_app = Blueprint("users_app", __name__)

@users_app.route("/", endpoint="list")
def users_list():
    users = User.query.all()
    return render_template("users/list.html", users=users)

@users_app.route("/<int:user_id>/", endpoint="details")
def user_details(user_id: int):
    user = User.query.filter_by(id=user_id).one_or_none()
    if user is None:
        raise NotFound(f"User #{user_id} doesn't exist!")

    return render_template("users/details.html", user=user)
```

Flask-Login

Устанавливаем Flask-Login==0.5.0

Сделаем страницы авторизации (вход, выход и т.д.) в виде блупринта.

Создаём файл `blog/views/auth.py` и инициализируем новый блупринт `auth_app`.

Для начала нам нужно инициализировать `LoginManager` и прописать необходимые для его работы свойства:

- `login_manager` — инициализируем объект для обработки авторизации;
- `login_manager.login_view = "auth_app.login"` — указываем view для авторизации;
- при помощи декоратора `login_manager.user_loader` вытаскиваем искомого пользователя (по ID);
- при помощи декоратора `login_manager.unauthorized_handler` указываем обработку неавторизированной попытки доступа к защищённым view. Выполняем редирект на страницу авторизации;
- в `__all__` указываем объекты, которые будем импортировать.

```
from flask import Blueprint, render_template, request, redirect, url_for
from flask_login import LoginManager, login_user, logout_user, login_required

from blog.models import User

auth_app = Blueprint("auth_app", __name__)

login_manager = LoginManager()
login_manager.login_view = "auth_app.login"

@login_manager.user_loader
def load_user(user_id):
    return User.query.filter_by(id=user_id).one_or_none()

@login_manager.unauthorized_handler
def unauthorized():
    return redirect(url_for("auth_app.login"))

__all__ = [
    "login_manager",
    "auth_app",
]
```


Теперь создаём шаблон для логина: `blog/templates/auth/login.html`. Добавляем обычную форму

```
{% extends 'base.html' %}

{% block title %}
    Login as
{% endblock %}

{% block body %}
    <h1>Login as user:</h1>
    {% if error %}
        <div class="alert alert-danger" role="alert">
            {{ error }}
        </div>
    {% endif %}
    <form method="post">
        <label for="input-username">Login as (username):</label>
        <input
            id="input-username"
            type="text"
            name="username"
            placeholder="guest"
            required
        >
        <button type="submit">Login</button>
    </form>
{% endblock %}
```

Добавляем view для входа. Пока что вход делаем просто по юзернейму — рассмотрим авторизацию по паролю немного позже. Здесь проверяем наличие пользователя и выполняем авторизацию.

```
@auth_app.route("/login/", methods=["GET", "POST"], endpoint="login")
def login():
    if request.method == "GET":
        return render_template("auth/login.html")

    username = request.form.get("username")
    if not username:
        return render_template("auth/login.html", error="username not passed")

    user = User.query.filter_by(username=username).one_or_none()
    if user is None:
        return render_template("auth/login.html", error=f"no user {username!r} found")

    login_user(user)
    return redirect(url_for("index"))
```

Также добавляем view для выхода. И тут же для примера добавляем view, которое требует авторизации, чтобы его можно было посмотреть:

```
@auth_app.route("/logout/", endpoint="logout")
@login_required
def logout():
    logout_user()
    return redirect(url_for("index"))

@auth_app.route("/secret/")
@login_required
def secret_view():
    return "Super secret data"
```

Редактируем `blog/templates/base.html`, добавляя информацию об авторизации в навигацию:

```
<div class="navbar-nav ms-auto">
  {% if current_user.is_authenticated %}
    <a class="nav-link" href="{{ url_for('logout') }}">Logout</a>
  {% else %}
    <a href="{{ url_for('login') }}"
      class="nav-link {% if request.endpoint == 'login' -%}active{%- endif %}">
      Login
    </a>
  {% endif %}
</div>
```

И в индекс `blog/templates/index.html`:

```
<h1>
  Hello
  {% if current_user.is_authenticated %}
    {{ current_user.username }}
  {% else %}
    anon
  {% endif %}
</h1>
```

Добавляем UserMixin к модели пользователя

Что это такое и зачем нужно? Для входа в систему требуется модель пользователя со следующими свойствами:

- имеет метод `is_authenticated()`, который возвращает `True`, если пользователь предоставил действительные учетные данные;
- имеет метод `is_active()`, который возвращает `True`, если учетная запись пользователя активна;
- имеет метод `is_anonymous()`, который возвращает `True`, если текущий пользователь является анонимным пользователем;
- имеет метод `get_id()`, который, учитывая пользовательский экземпляр, возвращает уникальный ID для этого объекта.

Класс `UserMixin` обеспечивает реализацию этих свойств. Именно по этой причине вы можете вызвать, например, `is_authenticated`, чтобы проверить, являются ли предоставленные учетные данные правильными, вместо того чтобы определять метод для этого самостоятельно.

Вновь меняем `blog/models/user.py`:

```
from flask_login import UserMixin

# И подмешиваем миксин в класс:
class User(db.Model, UserMixin):
    ...
```

Подключаем блупринт авторизации и LoginManager

Снова редактируем `blog/app.py`:

```
from blog.views.auth import login_manager, auth_app

# для работы авторизации нам обязательно нужен SECRET_KEY в конфигурации, добавляем
app.config["SECRET_KEY"] = "abcdefg123456"

app.register_blueprint(auth_app, url_prefix="/auth")

login_manager.init_app(app)
```

Чтобы проинициализировать базу данных, необходимо выполнить `flask init-db`. Затем создаём пользователей командой `flask create-users`. (если этого ещё не сделали).

Теперь мы можем выполнять авторизацию от имени любого существующего пользователя, просматривать защищённый view, выходить.

Итоги

На занятии мы рассмотрели азы работы с SQLAlchemy, работу с сессиями, моделями, а также познакомились с расширением Flask-SQLAlchemy. Научились пользоваться библиотекой Flask-login, которая помогает легко настроить авторизацию и доступ, а также научились хранить данные в сессии пользователя.

Практическое задание

1. Подключить Flask-SQLAlchemy в свой Flask проект.
2. Создать базовую модель пользователя, добавить к ней UserMixin.
3. Добавить страницу авторизации.
4. Создать один view, который недоступен анонимным пользователям.

Дополнительные материалы

1. [Custom Commands](#).
2. [User class Flask-Login required props](#).
3. [flask_login.UserMixin](#).
4. [SECRET_KEY](#).
5. [Remember Me](#).
6. [Alternative Tokens](#).