**Лабораторна робота №6**

**Авторизація, серверна частина**

**Мета:** ознайомитись з принципами написання серверної частина за допомогою Python реалізувати авторизацію та використання криптографічних методів хешування, використання salt.

## 6.1 Теоретичні відомості

У криптографії , сіль (salt) – це випадкові дані, що подаються як додаткові вхідні дані до односторонньої функції , яка хешує дані , пароль або парольну фразу. Соління допомагає захиститися від атак, які використовують попередньо обчислені таблиці (наприклад, райдужні таблиці ), значно збільшуючи розмір таблиці, необхідної для успішної атаки. Воно також допомагає захистити паролі, які зустрічаються кілька разів у базі даних, оскільки для кожного екземпляра пароля використовується нова сіль. Крім того, соління не створює жодного навантаження на користувачів.

Зазвичай для кожного пароля випадковим чином генерується унікальна сіль. Сіль і пароль (або його версія після розтягування ключа ) об'єднуються та передаються до криптографічної хеш-функції , а вихідне хеш-значення потім зберігається разом із сіллю в базі даних. Сіль не потрібно шифрувати, оскільки знання солі не допоможе зловмиснику.

Соління широко використовується в кібербезпеці, від облікових даних систем Unix до безпеки в Інтернеті .

Без солі ідентичні паролі будуть зіставлятися з ідентичними хеш-значеннями, що може полегшити хакеру вгадування паролів за їх хеш-значенням.

| Ім'я користувача | Рядок для хешування | Хешоване значення = SHA256 |
|---|---|---|
| user1 | password123 | EF92B778BAFE771E89245B89ECBC08A44A4E166C06659911881F383D4473E94F |
| user2 | password123 | EF92B778BAFE771E89245B89ECBC08A44A4E166C06659911881F383D4473E94F |

Замість цього генерується сіль, яка додається до кожного пароля, що призводить до того, що результуючий хеш виводить різні значення для одного й того ж початкового пароля.

| Ім'я користувача | Значення солі | Рядок для хешування | Хешоване значення = SHA256 (пароль + значення Salt) |
|---|---|---|---|
| user1 | D;%yL9TS:5PalS/d | password123D;%yL9TS:5PalS/d | 9C9B913EB1B6254F4737CE947EFD16F16E916F9D6EE5C1102A2002E48D4C88BD |
| user2 | )<,-<U(jLezy4j>* | password123)<,-<U(jLezy4j>* | 6058B4EB46BD6487298B59440EC8E70EAE482239FF2B4E7CA69950DFBD5532F2 |

Сіль і хеш потім зберігаються в базі даних. Щоб пізніше перевірити правильність пароля, який вводить користувач, можна виконати той самий процес (додати сіль цього користувача до пароля та обчислити результуючий хеш): якщо результат не відповідає збереженому хешу, можливо, було введено неправильний пароль.

На практиці, сіль зазвичай генерується за допомогою криптографічно захищеного генератора псевдовипадкових чисел (CSPRNG). CSPRNG призначені для створення непередбачуваних випадкових чисел, які можуть бути буквено-цифровими. Хоча загалом це не рекомендується через низьку безпеку, деякі системи використовують позначки часу або прості лічильники як джерело солі. Іноді сіль може бути згенерована шляхом поєднання випадкового значення з додатковою інформацією, такою як позначка часу або дані, специфічні для користувача, щоб забезпечити унікальність у різних системах або періодах часу.

Використання однієї й тієї ж солі для всіх паролів небезпечне, оскільки попередньо обчислена таблиця, яка враховує лише сіль, зробить її марною.

Генерація попередньо обчислених таблиць для баз даних з унікальними солями для кожного пароля не є життєздатною через обчислювальні витрати на це. Але якщо для всіх записів використовується спільна сіль, створення такої таблиці (яка враховує сіль) стає життєздатною та, можливо, успішною атакою.

Оскільки повторне використання солі може призвести до того, що користувачі з однаковим паролем матимуть однаковий хеш, злом одного хешу може призвести до компрометації й інших паролів.

Якщо сіль занадто коротка, зловмисник може попередньо обчислити таблицю всіх можливих солей, доданих до кожного ймовірного пароля. Використання довгої солі гарантує, що така таблиця буде надмірно великою 16 байт (128 біт) або більше зазвичай достатньо для забезпечення достатньо великого простору можливих значень, мінімізуючи ризик колізій (тобто двох різних паролів з однаковою сіллю).

```
CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            login TEXT UNIQUE NOT NULL,
            password TEXT NOT NULL,
            salt TEXT NOT NULL
        );
```

Оператор CREATE TABLE IF NOT EXISTS users ( id INTEGER PRIMARY KEY AUTOINCREMENT, username TEXT UNIQUE NOT NULL, password_hash TEXT NOT NULL, salt TEXT NOT NULL ); описує структуру таблиці користувачів у базі даних SQLite і є частиною мовою SQL для визначення структури даних. Ключове слово CREATE TABLE створює нову таблицю, а фраза IF NOT EXISTS додає важливу властивість ідемпотентності: команда не викличе помилку, якщо таблиця з таким ім'ям уже існує, а просто нічого не зробить. Назва users — це ім'я таблиці, в якій зберігатимуться записи про користувачів. Поле id INTEGER PRIMARY KEY AUTOINCREMENT означає, що кожен запис отримує унікальний цілочисельний ідентифікатор, який автоматично збільшується при кожній вставці нового рядка. PRIMARY KEY гарантує унікальність цього поля і робить його основним ключем, за яким можна однозначно ідентифікувати рядок. Поле username TEXT UNIQUE NOT NULL зберігає логін користувача як текстовий рядок, при цьому NOT NULL забороняє зберігати порожнє значення, а UNIQUE гарантує, що двоє різних користувачів не можуть мати один і той самий username, тобто логін стає бізнес-унікальним ідентифікатором. Поле password TEXT NOT NULL призначене для зберігання не самого пароля, а його криптографічного хешу, що є базовою вимогою безпеки: хеш — це одностороннє перетворення, з якого практично неможливо відновити початковий пароль, але можна перевірити, чи введений пароль співпадає з тим, що був збережений. Поле salt TEXT NOT NULL зберігає "сіль" — випадковий рядок, який додається до пароля перед хешуванням. Сіль робить хеші унікальними навіть для однакових паролів у різних користувачів і захищає від попередньо підготовлених словників хешів (rainbow tables). Загалом ця структура таблиці реалізує мінімальну, але правильну з точки зору практики безпеки модель користувача: автоінкрементний ідентифікатор, унікальний логін, захищений пароль через хеш + сіль.

```
from pydantic import BaseModel

class LoginRequest (BaseModel):
  login: str
  password: str
```

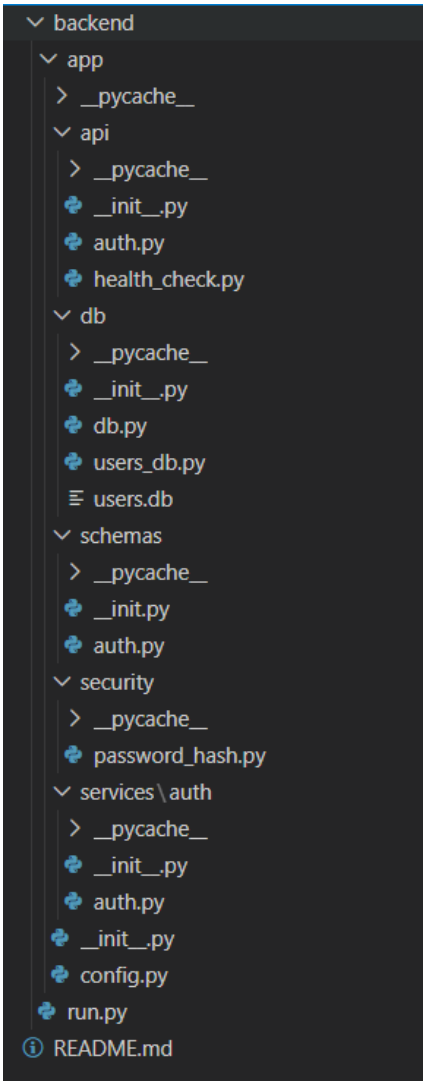
Pydantic-моделі — це Python-класи, що успадковують BaseModel з бібліотеки Pydantic і використовуються для формального опису структури даних, їхньої типізації та валідації.

Коли ти оголошуєш модель на кшталт class LoginRequest(BaseModel): username: str; password: str, ти задаєш контракт: об’єкт типу LoginRequest повинен містити поля username і password саме як рядки. При побудові такого об’єкта з “сирих” даних (наприклад, з JSON із HTTP-запиту) Pydantic автоматично перевіряє наявність необхідних полів, їхній тип, пробує виконати конверсію (наприклад, з числа в рядок чи навпаки), і якщо дані не відповідають моделі, викидає структуровану помилку валідації. Це дозволяє чітко відокремити рівень транспорту (JSON, request body) від внутрішнього рівня логіки, відбувається робота з уже гарантовано валідними об’єктами. Теоретично Pydantic реалізує схему типізованих даних, схожу за ідеєю на DTO (Data Transfer Object) в інших мовах, але завдяки інтеграції з type hints у Python, вона також покращує автодоповнення, статичний аналіз і робить API самодокументованим: за моделями можна зрозуміти, які саме поля очікує ендпоінт і що він повертає. У контексті авторизації та реєстрації Pydantic-моделі використовуються для опису вхідних структур на кшталт LoginRequest, що забезпечує єдину точку правди для валідації даних користувача перед тим, як ці дані потрапляють у бізнес-логіку або в базу даних. Важливо, що Pydantic не лише перевіряє типи, а й повертає детальний опис помилок, який зручно трансформувати у JSON-відповідь з поясненням, що саме не так у надісланих даних.

```
from flask import Blueprint
bp = Blueprint("health_check", __name__)

@bp.get("/health_check")
def health_check():
  return "Ok"
```

Данна конструкція визначає HTTP-ендпоінт у Flask (через Blueprint), який виступає як health-check сервісу. Декоратор @bp.get("/health") оголошує, що функція health_check має викликатися при отриманні GET-запиту на шлях /health у рамках цього blueprint’а. Теоретично це реалізація патерну REST-ендпоінта: функція є “view function” або “контролером”, який приймає HTTP-запит (через Flask), обробляє його і повертає HTTP-відповідь. У Flask можна повертати як спеціальний об’єкт Response, так і просто словник; у випадку повернення словника Flask автоматично серіалізує його в JSON і встановлює відповідний заголовок Content-Type: application/json, а також статус-код 200 за замовчуванням. Health-check ендпоінт з точки зору теорії є сервісним ендпоінтом для моніторингу: зовнішні системи (балансувальники, оркестратори контейнерів, системи моніторингу) періодично звертаються до нього, щоб перевірити, чи додаток “живий” і чи здатен відповідати на запити. Повертаючи просту фіксовану структуру {"status": "ok", "message": "alive"}, сервіс явно сигналізує про свою працездатність; у більш розширеному варіанті тут можуть додаватися перевірки підключення до бази, черг, інших залежностей. З точки зору архітектури REST, це ресурс, який не представляє бізнес-дані, а є технічним/службовим ресурсом, що дозволяє інтеграційній інфраструктурі приймати рішення: залишати інстанс у пулі, виводити його з балансу, перезапускати контейнер тощо. Таким чином, цей рядок коду формалізує просту, але дуже важливу практику — наявність чіткого, машиночитаного індикатора здоров’я веб-сервісу.




Під час опису ієрархії папок для бекенду на Python у статті варто підкреслити, що структура проєкту має відображати логіку роботи застосунку і допомагати швидко орієнтуватися у коді. На верхньому рівні зазвичай розташовують мінімальний набір файлів: точку входу в застосунок (`run.py` або `main.py`), де створюється екземпляр веб-фреймворку та підключаються маршрути, а також `README.md` з інструкціями зі встановлення та запуску. Основний код доцільно згрупувати в пакет `app` з файлом `__init__.py`, який виступає “ядром” бекенду. Усередині цього пакета код поділяють за ролями: рівень API, рівень бізнес-логіки, робота з базою даних, схеми даних, безпека та конфігурація.

У підпапці `api` зберігають модулі, що описують HTTP-маршрути: окремі файли для авторизації (`auth.py`), роботи з користувачами, перевірки “живості” сервера (`health_check.py`) тощо. Важливо, щоб у цих модулях була мінімальна логіка: вони лише приймають дані від клієнта, викликають відповідні сервіси й повертають відповіді, спираючись на Pydantic-схеми. Власне бізнес-логіку зручно винести в папку `services`: тут розміщують функції й класи, які реалізують сценарії на кшталт “створити користувача”, “виконати логін”, “оновити токен”. Такі сервіси працюють із шаром доступу до даних і компонентами безпеки, але не знають нічого про HTTP-запити, завдяки чому їх легко тестувати окремо.

Робота з базою даних зосереджується в папці `db`. У загальному модулі `db.py` зазвичай описують створення підключення, фабрику сесій або функцію `get_db`, а в окремих файлах на кшталт `users_db.py` — конкретні репозиторії з CRUD-операціями. Це дозволяє ізолювати SQL-запити та змінювати спосіб зберігання даних (SQLite, PostgreSQL, інша СУБД) без переписування всього бекенду. Сам файл бази, якщо використовується SQLite (`users.db`), у

статті варто згадати як технічний артефакт, який краще винести в окрему директорію й не тримати під контролем версій. Для опису даних на рівні API варто виділити папку `schemas`, де зберігаються Pydantic-моделі для запитів та відповідей. Вони виконують роль контрактів між клієнтом і сервером, а також між різними шарами застосунку, і не прив'язані напряму до структури таблиць у базі.

Окремий аспект — безпека. У папці `security` доцільно сконцентрувати все, що стосується хешування паролів, перевірки їх коректності, генерації й валідації JWT-токенів та налаштувань аутентифікації. Такий поділ спрощує зміну алгоритму хешування або формату токенів: достатньо змінити код в одному місці, не проходячи увесь проєкт. Конфігураційні параметри застосунку варто описати в окремому модулі `config.py`: тут зчитуються змінні середовища (секретний ключ, параметри бази даних, режим розробки чи продакшену), а назовні експортується клас або об'єкт конфігурації, який використовують інші модулі. Важливо наголосити, що секрети не повинні бути "зашиті" в репозиторій у відкритому вигляді — для цього застосовують `.env`-файли та системні змінні. У статті також можна коротко згадати про службові папки `__pycache__`, які породжуються самим Python і не є частиною архітектури, а також про доцільність наявності окремої папки `tests` для автоматичних тестів. Загальний висновок, який варто сформулювати: продумана ієрархія папок у бекенді на Python розділяє HTTP-маршрути, бізнес-логіку, доступ до даних, схеми й безпеку, робить код передбачуваним для інших розробників і полегшує підтримку та розвиток проєкту.

### 6.2 Підготовка до роботи

1. Встановити VSCode, Python.3.10, GitBash та Postman(або будь-який інший додаток для тестування API).

2. На сайті GitHub створити репозиторій та додати викладача до репозиторію. Settings -> Collaborators -> Add people.

3. Створити в репозиторії папку backend та провести базові налаштування серверу використовуючи flask та створивши роут health-check, (Pull Request створювати поки не потрібно) перевірити коректність роботи та запушити зміни.

### 6.3 Порядок виконання лабораторної роботи

1. Реалізувати авторизацію на сайті. Користувачі повинні вводити власний логін та пароль. Створити роути /api/auth/login та /api/auth/registration;

2. Для збереження даних користувачів використовувати базу даних SQLite;

3. Забезпечити коректну роботу роутів та опрацювання помилок;

4. Використовувати Pydantic для валідації віхідних даних;

5. Для хешування паролю використовувати сіль та sha256 з бібліотеки hashlib;

6. За допомогою Postman протестувати API.

7. Після завершення запушити зміни на GitHub та створити Pull Request та додати викладача у Reviewers і дочекатись схвалення.

### 6.4 Зміст звіту

1. Найменування і мета роботи;
2. Код по кожному пункту порядку виконання роботи;
3. Результати роботи по кожному пункту виконання роботи;

4. Висновки.

### 6.5 Контрольні запитання

1. Що таке “сіль” (salt) у контексті зберігання паролів і яку задачу безпеки вона вирішує.

2. Який статус-код HTTP за замовчуванням повертає Flask, якщо у view-функції повернути словник, і чому це підходить для health-check?

3. Що таке Pydantic-модель і навіщо вона потрібна в контексті веб-API?

4. Чому для поля  login доцільно використовувати обмеження UNIQUE NOT NULL?

5. Чому health-check ендпоінт вважається технічним/службовим, а не бізнесовим ресурсом API?