**ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ  
ІМЕНІ СЕМЕНА КУЗНЕЦЯ**

**КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ**

# **КУРСОВИЙ ПРОЄКТ: ПРОГРАМУВАННЯ**

на тему: «Розроблення програмного продукту для для обміну авторськими літературними творами з використанням бази даних»

Студента 3 курсу 6.04.122.010.21.02 групи

Спеціальності 122 "Комп’ютерні науки"

першого (бакалаврського) рівня

*Гетьмана МИКОЛИ*

Керівник: доцент кафедри ІС, к.т.н., доцент,

Олександр ПЕТРЕНКО

Національна шкала \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кількість балів: \_\_\_\_\_\_\_\_\_\_Оцінка: ECTS \_\_\_\_\_

Члени комісії

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Олег ФРОЛОВ

(підпис) (прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Віктор ЗАДАЧИН

(підпис) (прізвище та ініціали)

м. Харків – 2023 рік

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ

ІМЕНІ СЕМЕНА КУЗНЕЦЯ

Факультет економічної інформатики

Кафедра інформаційних систем

ЗАВДАННЯ

на комплексний курсовий проект: Програмування

студенту 3-го курсу групи 6.04.122.010.21.2

Гетьману Миколі Юрійовичу

1. Тема проекту: "Розроблення програмного продукту для для обміну авторськими літературними творами з використанням бази даних".
2. Термін здачі студентом закінченого проекту "20" грудня 2023 р.
3. Вхідні дані до проекту: літературні джерела, технічна документація щодо розроблення програм, ДСТУ з оформлення документації.
4. Зміст пояснювальної записки: Вступ. Специфікація проекту. Програмна документація. Висновки. Додатки.
5. Перелік графічного матеріалу: UML-діаграма класів програмної системи.

РЕФЕРАТ

Пояснювальна записка до курсового проекту: 40 с., 20 рис., 10 табл., 9 джерел.

Об'єктом даного курсового проекту є розробка веб-сайту "ScaryStories" з використанням двох мов програмування - Python та JavaScript. Метою проекту є створення інтерактивної платформи, на якій користувачі зможуть обмінюватися своїми страшними історіями та знаходити рівнодумців у віртуальній спільноті. Основні завдання включають розробку зручного інтерфейсу, функціоналу для додавання історій, а також забезпечення безпеки та конфіденційності користувачів.

У процесі розробки використовувалися дві мови програмування: Python для бекенд-розробки та обробки логіки серверної частини, а також JavaScript для фронтенд-розробки та взаємодії з користувачем у браузері.

Для забезпечення безпеки та конфіденційності були використані сучасні методи шифрування та автентифікації. Кожен користувач має можливість реєстрації, додавання своїх страшних історій. Адміністратор має доступ до інструментів управління історіями та категоріями, включаючи їх видалення у випадку порушень правил спільноти.

Веб-сайт ScaryStories розроблений з урахуванням високої взаємодії та зручності для користувачів. Крім того, він забезпечує можливість розширення функціоналу у майбутньому.

Використання двох популярних мов програмування та врахування аспектів безпеки та конфіденційності робить веб-сайт ScaryStories актуальним та конкурентоздатним у сфері віртуальних спільнот та обміну страшними історіями. Очікується, що цей проект буде продовжувати розвиватися, додавати нові функції та покращувати існуючі, щоб відповідати потребам користувачів у сучасному інтернет-середовищі.

САЙТ, БАЗА ДАННИХ, КЛІЄНТ, СЕРВЕР, ФРОНТЕНД, БЕКЕНД, АВТЕНТИФІКАЦІЯ, PYTHON, JAVA SCRIPT

ЗМІСТ

Вступ………………………………………………………………..….……5

1 Спецефікація проекту……………………….…………………..….…… 6

1.2 Постановка завдання…………………………………………..…….... 6

1.3 Вимоги до програмного забезпечення……………………..…….……8

2 Програмна документація……………………………………..…………10

2.1 Архітектура програмної системи……………………………..………10

2.2 Тестування програмної системи…………………………..….………12

2.3 Розгортання програмного продукту……………………...…………..19

2.4 Керівництво користувача………………………………..….………...20

2.4.1 Призначення програмного продукту……………….……..………..20

2.4.2 Використання програмного продукту………………….…………..21

2.4.3 Повідомлення користувачеві…………………………….………….27

Висновки……………………………………………………….…………...29

Список використаних джерел………………………………….………….30

Додатки…………………………………………………………….………..31

ВСТУП

У світі, насиченому величезною кількістю інформації та різноманітних розваг, люди постійно шукають нові способи виявлення і ділення своїх емоцій, вражень та неповторних історій. Інтернет, як сучасна платформа для спілкування, є прекрасним інструментом для об'єднання спільнот та обміну унікальним контентом.

Цей курсовий проект став результатом інноваційного підходу до створення веб-ресурсу, який відповідає потребам та інтересам широкого кола користувачів. В рамках даного проекту розроблено та реалізовано веб-сайт для обміну страшними історіями – платформу, яка дозволяє учасникам поділитися своїми найзапам'ятнішими, а порою й найжахливішими переживаннями.

Сприяючи культурній еволюції інтернет-спільнот, цей веб-сайт створений з метою надання можливості кожному користувачеві висловити свої почуття та думки, знайти рівнодумців і отримати підтримку від інших. У світі віртуальних можливостей і діалогу цей проект спрямований на об'єднання людей через емоційний досвід, що розкривається у страшних історіях, створюючи неповторну атмосферу спільноти, яка розуміє і підтримує кожного її учасника.

У цьому курсовому проекті буде представлено архітектуру та функціональні можливості веб-сайту. Основною метою є розроблення платформи, яка дозволяє кожному користувачеві поділитися своїми найзапам'ятнішими, а порою й найжахливішими переживаннями, створюючи унікальне віртуальне середовище для обміну страшними історіями та підтримки спільноти.

СПЕЦІФІКАЦІЯ ПРОЕКТУ

**1.1 Постановка завдання**

Мета цього проекту - створення веб-сайту ScaryStories, що надасть користувачам можливість обмінюватися своїми страшними історіями та взаємодіяти віртуальною спільнотою. Основні завдання включають розробку зручного інтерфейсу, функціоналу для додавання та коментування історій, а також забезпечення безпеки та конфіденційності.

Основні функції:

1. Користувач

1.1. Реєстрація користувача

* Форма реєстрації з обов'язковими полями (нікнейм, електронна пошта, пароль).
* Перевірка на унікальність електронної пошти.
* Збереження інформації про користувача в базі даних.

1.2. Логін користувача

* Форма входу з полями для електронної пошти та пароля.
* Автентифікація користувача.
* Створення та управління сесією користувача.

1. Історія

2.1. Створення історії

* Форма для створення нової історії.
* Вибір категорії для історії.
* Збереження створеної історії в базі даних.

2.2. Редагування історії

* Можливість змінювати вміст та категорію історії.
* Збереження змін в базі даних.

2.3. Видалення історії

* Можливість видаляти історії.
* Підтвердження перед видаленням.

2.4. Перегляд історії

* Список всіх створених історій користувача.
* Перегляд конкретної історії.

1. Категорія

3.1. Створення категорії

* Форма для додавання нової категорії.
* Збереження категорії в базі даних.

3.2. Редагування категорії

* Можливість змінювати назву та зображення категорії.
* Збереження змін в базі даних.

3.3. Видалення категорії

* Можливість видаляти категорії.
* Перевірка на наявність історій у цій категорії перед видаленням.

3.4. Перегляд категорії

* Список усіх категорій користувача.
* Перегляд історій, пов'язаних з конкретною категорією.

**1.2 Вимоги до програмного забезпечення**

Функціональні вимоги:

1. Створення бази даних та запис даних у певному форматі.
2. Читання даних та їх відображення.
3. Додавання нового елемента даних до бази.
4. Оновлення будь-якого елемента даних у базі.
5. Видалення будь-якого елемента даних із бази.
6. Отримання та відображення підсумкової інформації.
7. Перевірка допустимості основних даних, що вводяться користувачем.
8. Видача користувачу попереджуючих та інформаційних повідомлень.

Нефункціональні вимоги:

Мінімальні вимоги до графічного інтерфейсу користувача:

1. Елементи інтерфейсу користувача мають супроводжуватися допоміжними текстовими мітками або мати заголовки, які виконуються українською або російською мовою.
2. Головне вікно застосунку – фрейм, який має панель меню з підтримкою "акселераторів", користувальницьку піктограму системного меню, панель інструментів із підтримкою спливаючих "підказок" для кнопок.
3. Дані, що зберігаються у файловій базі даних, повинні відображатися в табличному вигляді.
4. Наявність діалогових вікон, за допомогою яких користувач має можливість додавати або редагувати дані.
5. Наявність стандартних діалогових вікон відкриття та збереження файлу, за допомогою яких користувач має можливість виконувати відповідні дії.
6. Наявність стандартних діалогових вікон для відображення попереджуючих та інформаційних повідомлень, що можуть з'являтися в ході виконання програми.
7. Наявність діалогового вікна "Про програму", за допомогою якого користувач має можливість переглянути інформацією про розроблювача програми, зокрема його (її) фотографію.
8. Усі діалогові вікна повинні бути модальними та мати фіксований розмір.

Вимоги до архітектури програми:

1. Використання не менше одної структури даних із стандартної бібліотеки колекцій
2. Використання файлових потоків введення-виведення даних.
3. Використання механізму винятків для обробки помилок.

Вимоги до вихідного коду застосунку:

1. Додержання принципу інкапсуляції щодо рівнів доступу до полів та методів класів.
2. Вихідний код кожного з класів програми повинен міститися в окремому файлі
3. Наявність документаційних коментарів (для класів – призначення класу; для методів – призначення методу, опис параметрів та значення, 10 що повертається) з обов'язковим використанням відповідних документаційних тегів.
4. Виконання угод щодо запису тексту програм певною мовою програмування.

ПРОГРАМНА ДОКУМЕНТАЦІЯ

**2.1 Архітектура програмної системи**

Файлова структура проекту:

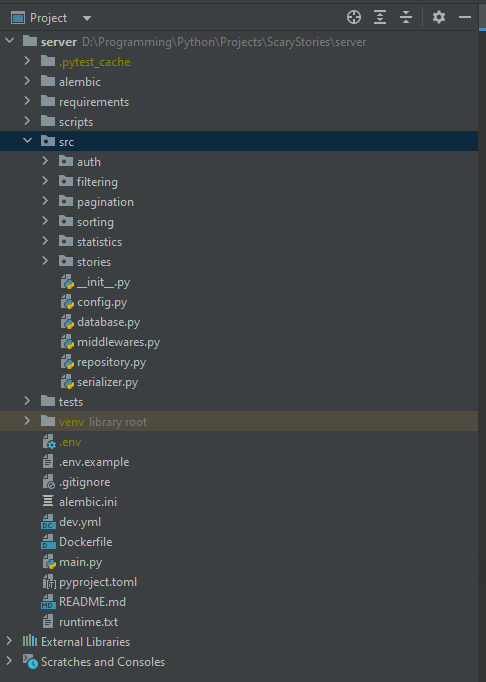


Рис. 2.1.1 Файлова структура сервера

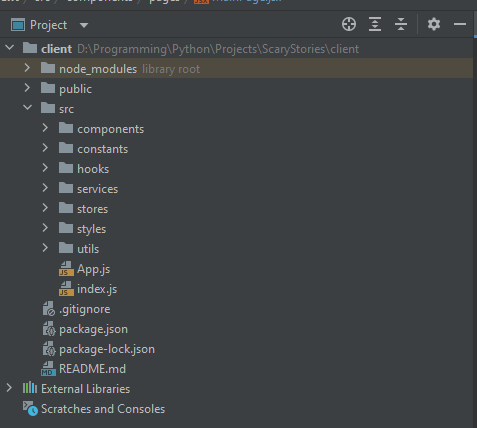


Рис. 2.1.2 Файлова структура клієнта

Структура бази даних:

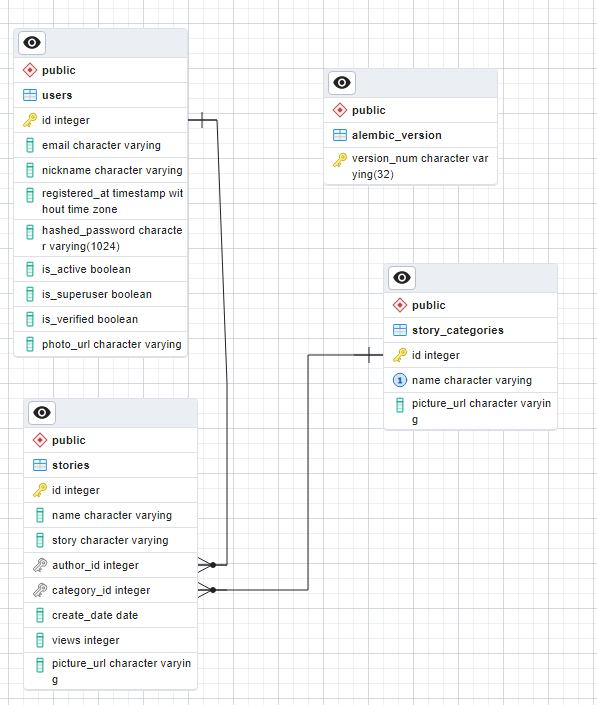


Рис. 2.1.3 ER-діаграма бази даних

Структура класів:

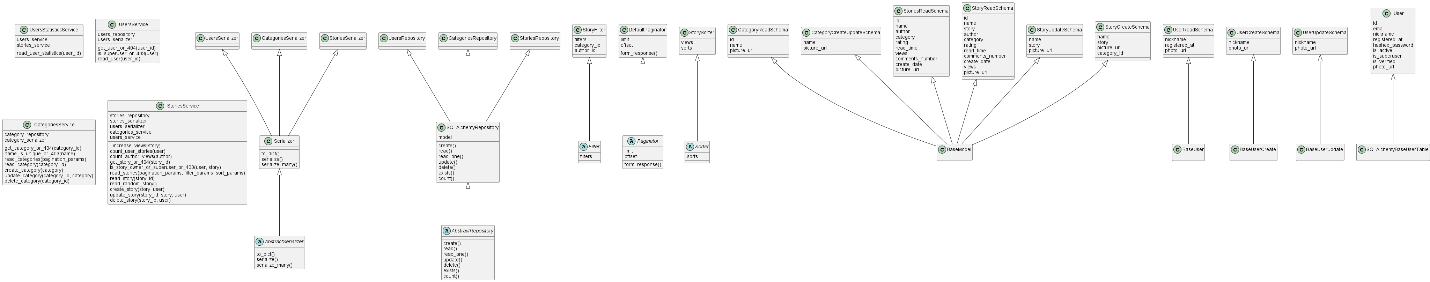
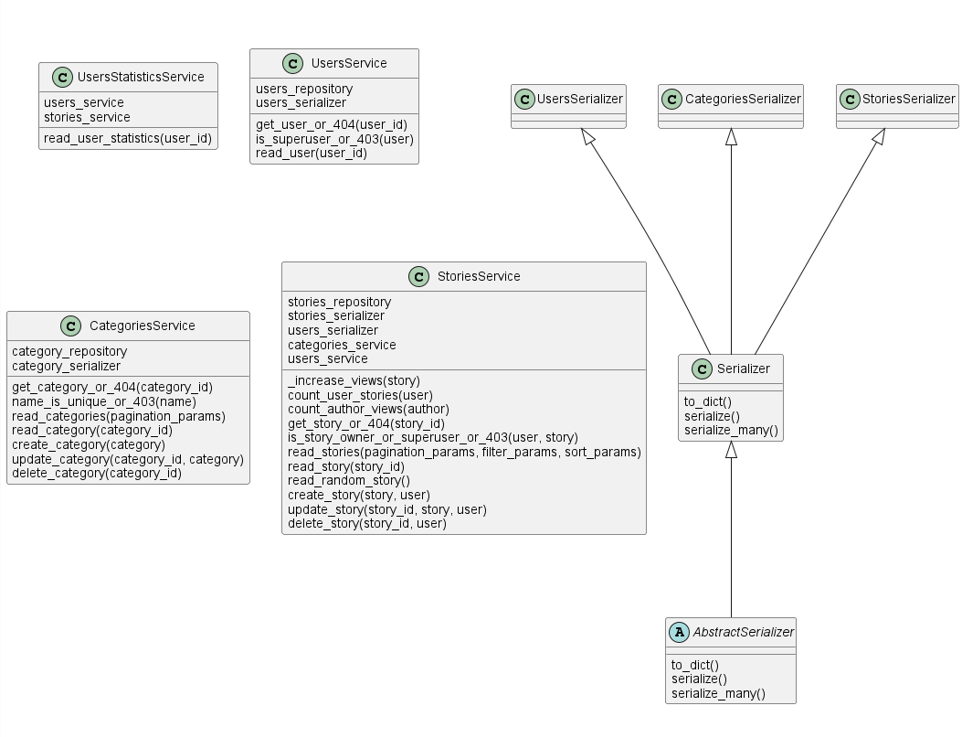
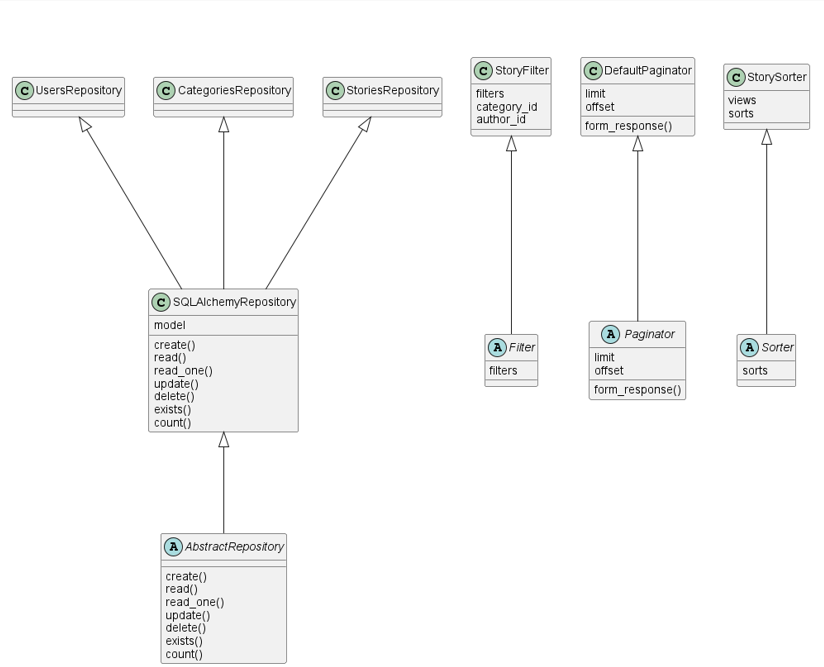
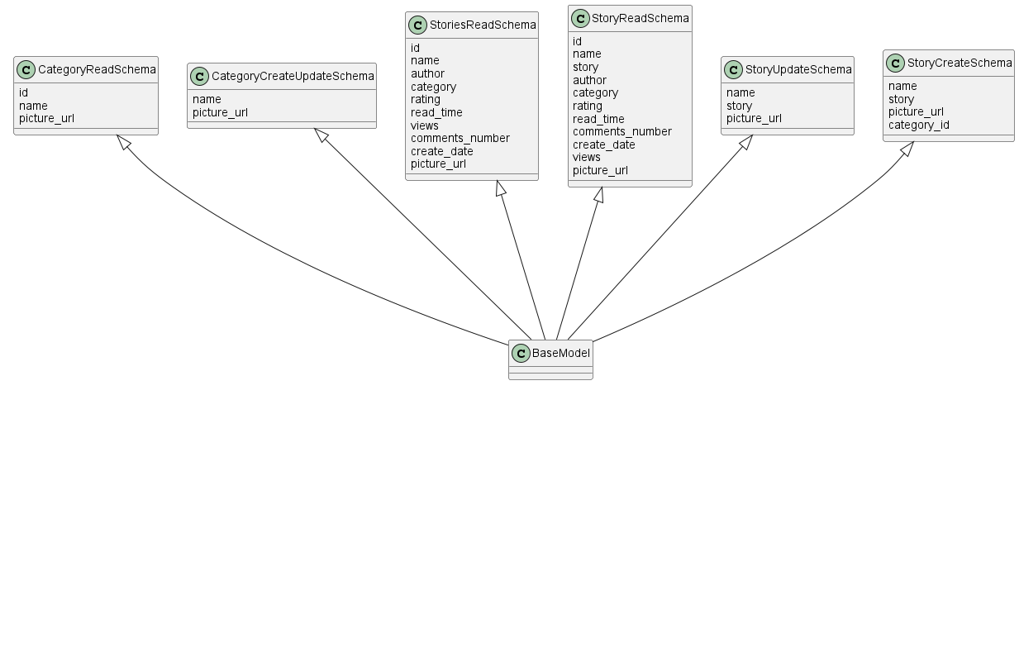
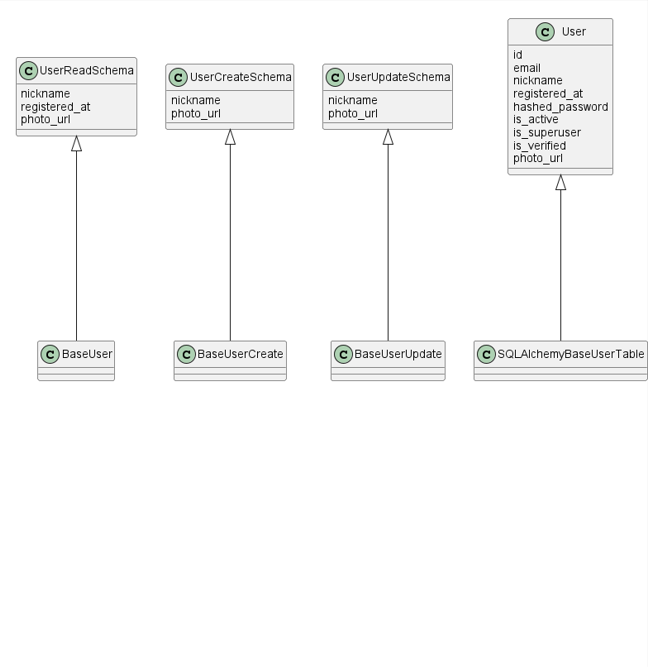


Рис. 2.1.4 Повна UML-діаграмма класів сервера









Базові архітектурні концепції:

1. **Repository** – структурний паттерн проектування, який визначає стандартний спосіб організації доступу до даних в додатку.
2. **Schema** – схема подання даних.
3. **Service** – основна логіка програми.
4. **Model** – модель подання даних у БД.

**2.2 Тестування програмної системи**

Тест-вимоги:

1. Користувач
   1. Реєстрація користувача
   2. Логін користувача
2. Історія
   1. Створення історії
   2. Редагування історії
   3. Видалення історії
   4. Перегляд історії
3. Категорія
   1. Створення категорії
   2. Редагування категорії
   3. Видалення категорії
   4. Перегляд категорії

Тест-план:

**Тестовий приклад:** № 1

**Призначення:** перевірка реєстрації

**Тест-вимоги, що перевіряються:** вимога №1.1

**Передумови для тесту:** сайт має бути запущений

**Критерій проходження тесту:** користувач був створений

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № з/п | Крок сценарію | Очікуваний результат | Отриманий результат | Відмітка про проходження кроку сценарію (Так/Ні) |
| 1 | 2 | 3 | 4 | 5 |
| 1. | Натиснути кнопку Sign Up у хедері | Повинне з’явитися модальне вікно реєстрації | Вікно з’явилося | Так |
| 2. | Ввести унікальні данні нового користувача та натиснути кнопку Sign Up знизу вікна | Вікно повинне закритися, користувач зареєстрований та може перейти в профіль | Вікно закрилося, користувач зареєстрований та може перейти в профіль | Так |

**Тестовий приклад:** № 2

**Призначення:** перевірка логіну

**Тест-вимоги, що перевіряються:** вимога №1.2

**Передумови для тесту:** сайт має бути запущений, користувач - зареєстрований

**Критерій проходження тесту:** користувач ввійшов в систему

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № з/п | Крок сценарію | Очікуваний результат | Отриманий результат | Відмітка про проходження кроку сценарію (Так/Ні) |
| 1 | 2 | 3 | 4 | 5 |
| 1. | Натиснути кнопку Sign In у хедері | Повинне з’явитися модальне вікно логіну | Вікно з’явилося | Так |
| 2. | Ввести данні попередньо зареєстрованого користувача та натиснути кнопку Sign In знизу вікна | Вікно повинне закритися, користувач залогінений та може зайти в профіль | Вікно закрилося, користувач залогінений та може зайти в профіль | Так |

**Тестовий приклад:** № 3

**Призначення:** перевірка створення історії

**Тест-вимоги, що перевіряються:** вимога №2.1

**Передумови для тесту:** сайт має бути запущений, мінімум один користувач створений та авторизований, мінімум одна категорія створена

**Критерій проходження тесту:** історія була створена

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № з/п | Крок сценарію | Очікуваний результат | Отриманий результат | Відмітка про проходження кроку сценарію (Так/Ні) |
| 1 | 2 | 3 | 4 | 5 |
| 1. | Зайти в будь-яку категорію | Повинна відобразитися сторінка категорії | Сторінка відобразилася | Так |
| 2. | Натиснути кнопку додати (+) | Повинне відобразитися модальне вікно створення історії | Вікно відобразилося | Так |
| 3. | Заповнити форму даними нової історії та натиснути кнопку Add | Повинна відобразитися картка нової історії | Картка з’явилася | Так |

**Тестовий приклад:** № 4

**Призначення:** перевірка редагування історії

**Тест-вимоги, що перевіряються:** вимога №2.2

**Передумови для тесту:** сайт має бути запущений, користувач-власник або адмін – авторизований, мінімум одна історія додана

**Критерій проходження тесту:** історія була відредагована

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № з/п | Крок сценарію | Очікуваний результат | Отриманий результат | Відмітка про проходження кроку сценарію (Так/Ні) |
| 1 | 2 | 3 | 4 | 5 |
| 1. | Обрати картку історії та натиснути синій олівець | Повинне з’явитися модальне вікно для редагування історії | Вікно з’явилося | Так |
| 2. | Заповнити форму у вікні зміненими даними історії та натиснути Save у вікні | Вікно повинне пропасти, змінена інформація зберегтись | Вікно пропало, інформація збереглась | Так |

**Тестовий приклад:** № 5

**Призначення:** перевірка видалення історії

**Тест-вимоги, що перевіряються:** вимога №2.3

**Передумови для тесту:** сайт має бути запущений,

користувач-власник або адмін – авторизований, мінімум одна історія додана

**Критерій проходження тесту:** історія була видалена

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № з/п | Крок сценарію | Очікуваний результат | Отриманий результат | Відмітка про проходження кроку сценарію (Так/Ні) |
| 1 | 2 | 3 | 4 | 5 |
| 1. | Обрати картку історії та натиснути червону корзину | Повинне з’явитися модальне вікно підтвердження | Вікно з’явилося | Так |
| 2. | Натиснути кнопку Continue у модальному вікні | Картка історії повинна пропасти | Картка пропала | Так |

**Тестовий приклад:** № 6

**Призначення:** перевірка перегляду історії

**Тест-вимоги, що перевіряються:** вимога №2.4

**Передумови для тесту:** сайт має бути запущений, мінімум одна історія додана

**Критерій проходження тесту:** історія була переглянута

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № з/п | Крок сценарію | Очікуваний результат | Отриманий результат | Відмітка про проходження кроку сценарію (Так/Ні) |
| 1 | 2 | 3 | 4 | 5 |
| 1. | Перейти в головну сторінку сайту, обрати карту будь-якої історії та натиснути | Повинна відобразитися сторінка історії | Відобразилася | Так |

**Тестовий приклад:** № 7

**Призначення:** перевірка створення категорії

**Тест-вимоги, що перевіряються:** вимога №3.1

**Передумови для тесту:** сайт має бути запущений, адмін – авторизований

**Критерій проходження тесту:** категорія була створена

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № з/п | Крок сценарію | Очікуваний результат | Отриманий результат | Відмітка про проходження кроку сценарію (Так/Ні) |
| 1 | 2 | 3 | 4 | 5 |
| 1. | Перейти на сторінку категорій | Повинна відобразитися сторінка категорій | Відобразилася | Так |
| 2. | Натиснути кнопку додати (+) | Повинне відобразитися модальне вікно створення категорії | Вікно відобразилося | Так |
|  | Заповнити форму даними нової категорії та натиснути кнопку Add | Повинна відобразитися картка нової категорії | Картка з’явилася | Так |

**Тестовий приклад:** № 8

**Призначення:** перевірка редагування категорії

**Тест-вимоги, що перевіряються:** вимога №3.2

**Передумови для тесту:** сайт має бути запущений, адмін – авторизований , мінімум одна категорія додана

**Критерій проходження тесту:** категорія була відредагована

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № з/п | Крок сценарію | Очікуваний результат | Отриманий результат | Відмітка про проходження кроку сценарію (Так/Ні) |
| 1 | 2 | 3 | 4 | 5 |
| 1. | Обрати картку категорії та натиснути синій олівець | Повинне з’явитися модальне вікно для редагування категорії | Вікно з’явилося | Так |
| 2. | Заповнити форму у вікні зміненими даними категорії та натиснути Save у вікні | Вікно повинне пропасти, змінена інформація зберегтись | Вікно пропало, інформація збереглась | Так |

**Тестовий приклад:** № 9

**Призначення:** перевірка видалення категорії

**Тест-вимоги, що перевіряються:** вимога №3.3

**Передумови для тесту:** сайт має бути запущений, адмін – авторизований , мінімум одна категорія додана

**Критерій проходження тесту:** категорія була видалена

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № з/п | Крок сценарію | Очікуваний результат | Отриманий результат | Відмітка про проходження кроку сценарію (Так/Ні) |
| 1 | 2 | 3 | 4 | 5 |
| 1. | Обрати картку категорії та натиснути червону корзину | Повинне з’явитися модальне вікно підтвердження | Вікно з’явилося | Так |
| 2. | Натиснути кнопку Continue у модальному вікні | Картка категорії повинна пропасти | Картка пропала | Так |

**Тестовий приклад:** № 10

**Призначення:** перевірка перегляду категорії

**Тест-вимоги, що перевіряються:** вимога №3.4

**Передумови для тесту:** сайт має бути запущений, адмін – авторизований, мінімум одна категорія додана

**Критерій проходження тесту:** категорія була переглянута

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № з/п | Крок сценарію | Очікуваний результат | Отриманий результат | Відмітка про проходження кроку сценарію (Так/Ні) |
| 1 | 2 | 3 | 4 | 5 |
| 1. | Перейти в сторінку категорій | Повинні відобразитися картки доданих категорій | Відобразилися | Так |

Звіт про тестування:

1. Користувач – успішно пройдено
   1. Реєстрація – успішно пройдено
   2. Логін – успішно пройдено
2. Історія – успішно пройдено
   1. Створення – успішно пройдено
   2. Редагування – успішно пройдено
   3. Видалення – успішно пройдено
   4. Перегляд – успішно пройдено
3. Категорія – успішно пройдено
   1. Створення – успішно пройдено
   2. Редагування – успішно пройдено
   3. Видалення – успішно пройдено
   4. Перегляд – успішно пройдено

Всі тести – успішно пройдено

**2.3 Розгортання програмного продукту**

Вимоги (браузера Chrome) до апаратних засобів**:**

* Розмір файлу встановлення: близько 50-100 МБ
* Оперативна пам'ять: 2 ГБ або більше
* Процесор: Intel Pentium 4 або вище
* Вільне місце на жорсткому диску: близько 200 МБ

Вимоги (браузера Chrome) до програмних засобів:

* Операційна система: Windows 7, Windows 8/8.1, Windows 10 або пізніша версія

Розгортання користувачем не потребує (тому що веб-сайт).

**2.4 Керівництво користувача**

**2.4.1 Призначення програмного продукту**

Призначенням цього веб-сайту є обмін страшними історіями, який об'єднує спільноту користувачів через поділ своїх найзапам'ятніших переживань. Веб-сайт дозволяє користувачам ділитися різноманітним контентом, таким як вражаючі події, невідомі явища або фантастичні сценарії.

Основна мета проекту - створення емоційного зв'язку між учасниками спільноти. Користувачі можуть висловлювати свої почуття та думки через створення та читання страшних історій.

Проект спрямований на сприяння культурної еволюції в інтернет-середовищі, де користувачі взаємодіють, реагують на історії один одного та знаходять рівнодумців.

Веб-сайт створює віртуальне середовище, де користувачі можуть відчути атмосферу страху та захоплення через читання і створення страшних історій.

2.4.2 Використання програмного продукту



Рис. 2.4.2.1 Хедер веб-сайту

**Хедер** – структурний елемент, який присутній на кожній сторінці сайту.

Хедер надає такі можливості як:

1. Логін
2. Реєстрація
3. Навігація сайтом

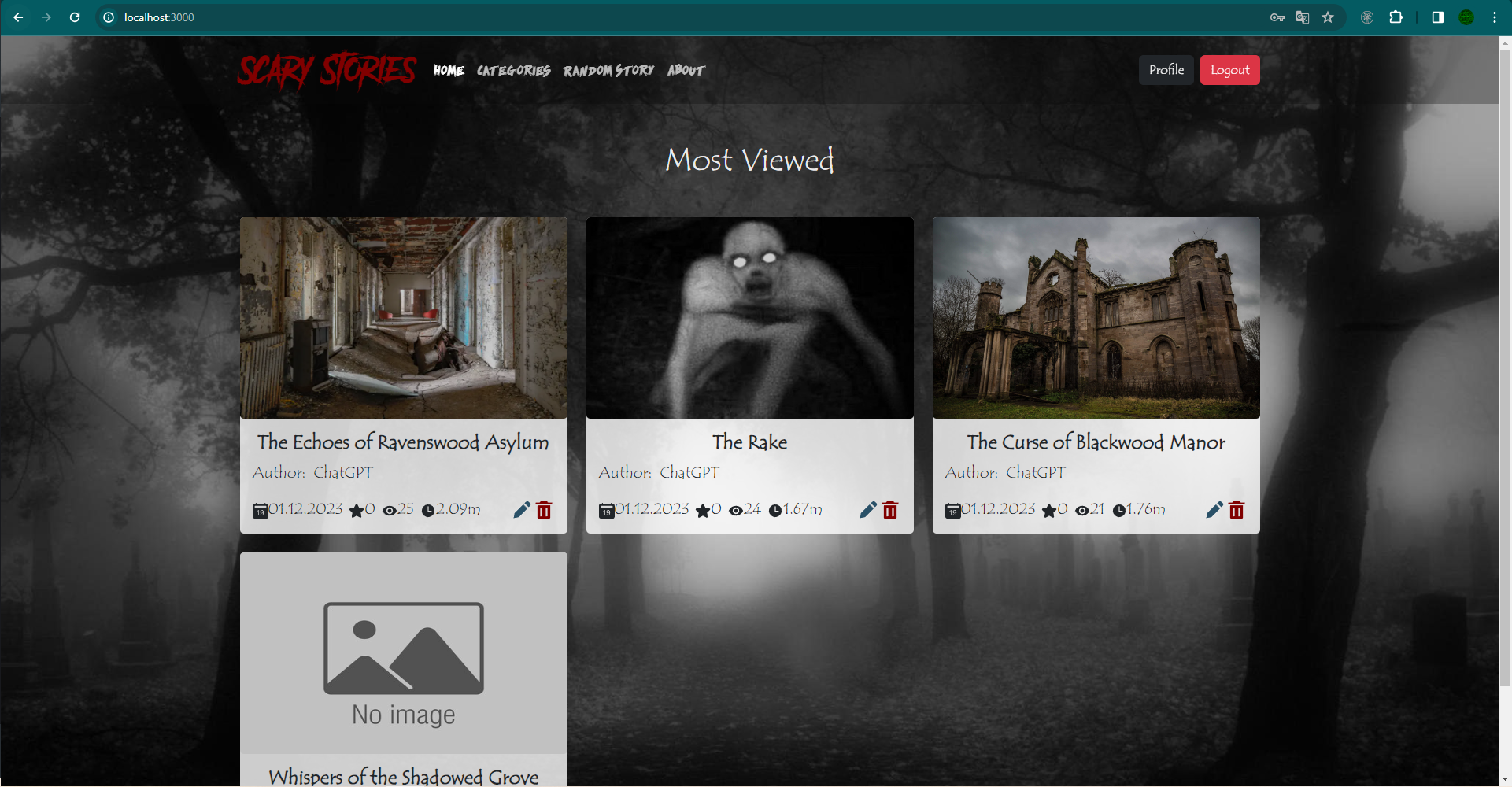


Рис. 2.4.2.2 Головна сторінка сайту

Головна сторінка надає такі можливості як:

1. Навігація між найбільш популярними історіями
2. Редагування історії (для власника або адміністратора)
3. Видалення історії (для власника або адміністратора)

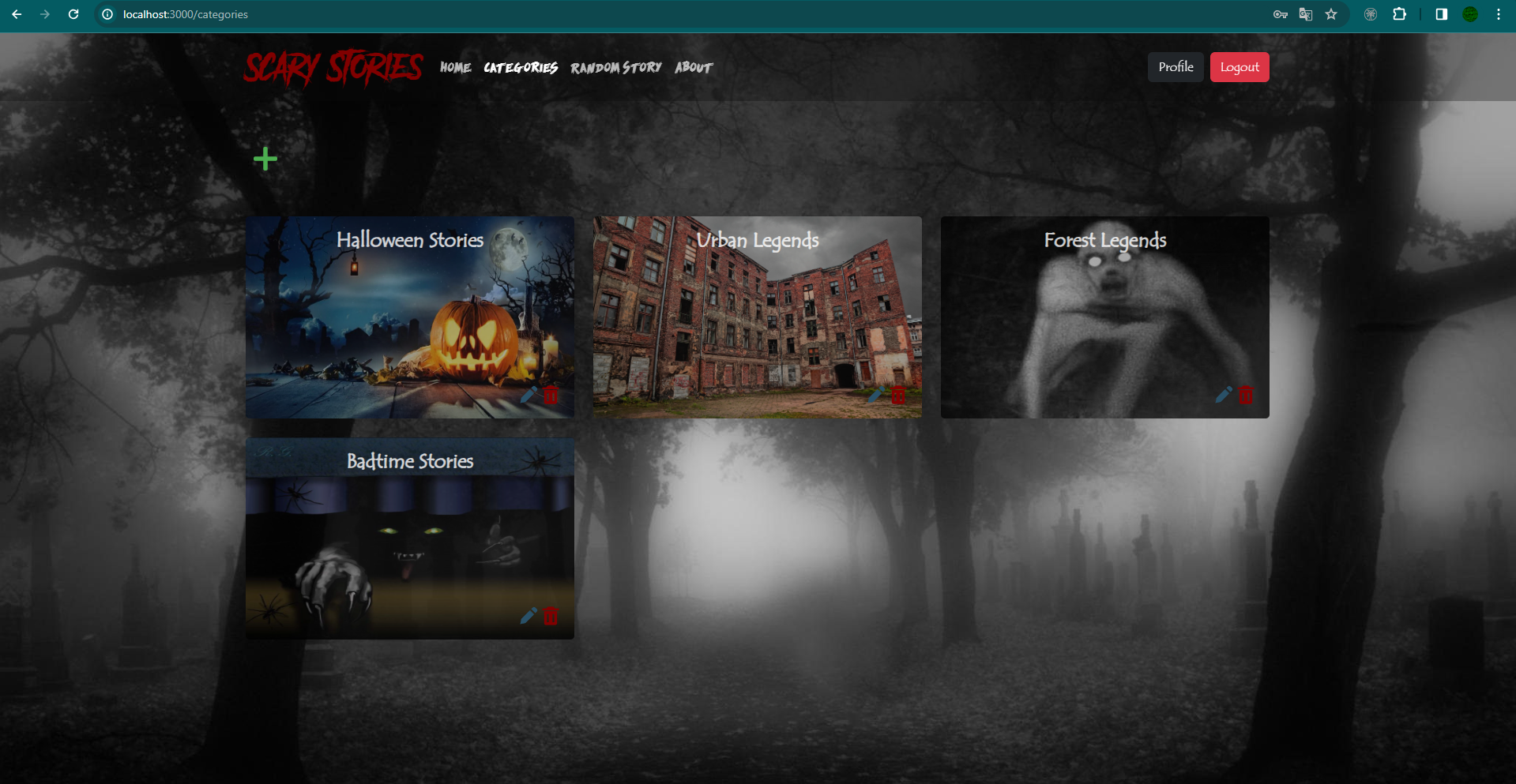


Рис. 2.4.2.3 Сторінка категорій

Сторінка категорій надає такі можливості як:

1. Навігація між категоріями
2. Редагування категорії (для адміністратора)
3. Видалення категорії (для адміністратора)
4. Додавання категорії (для адміністратора)

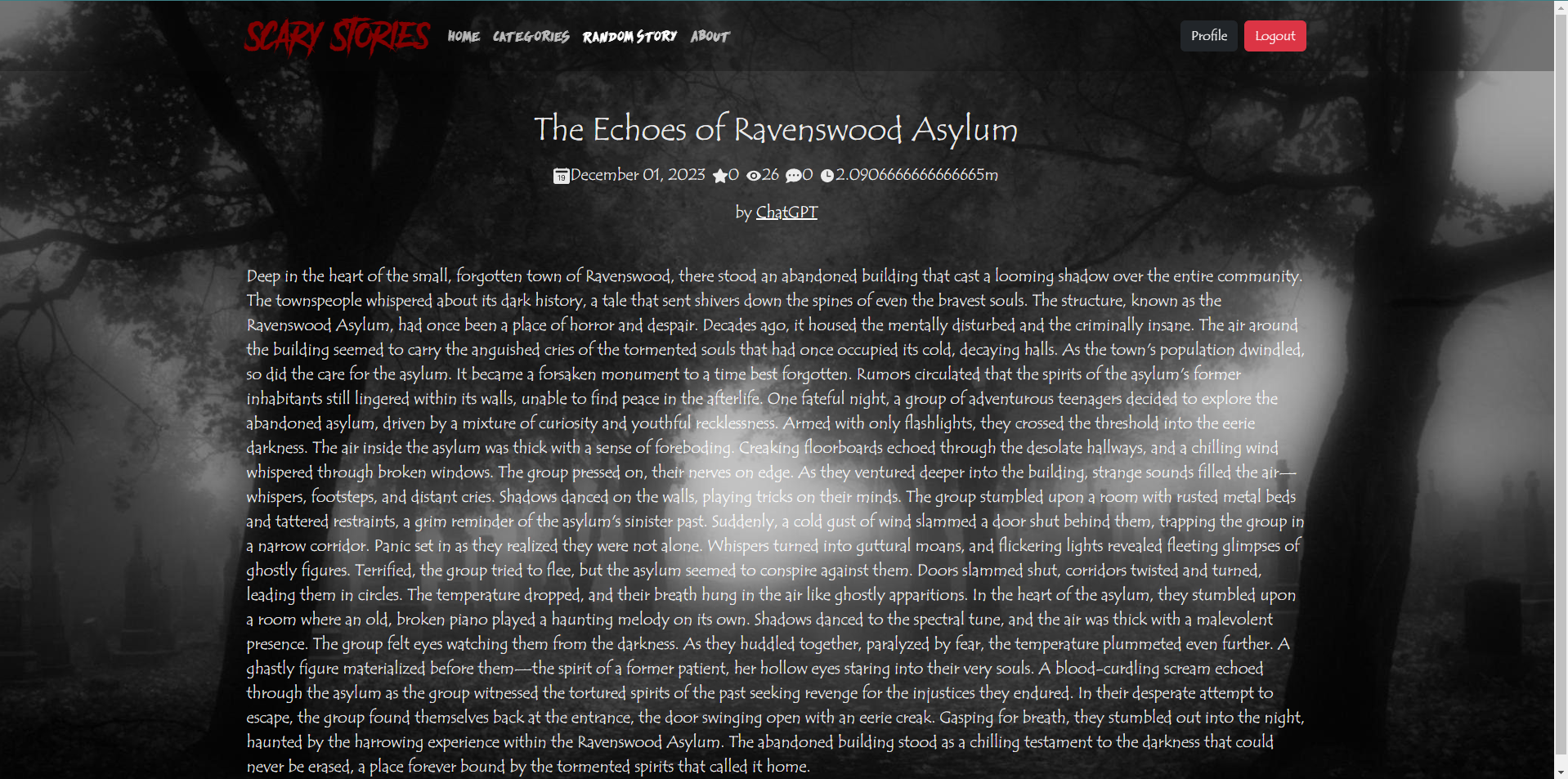


Рис. 2.4.2.4 Сторінка історії

Сторінка історії надає такі можливості як:

1. Читання історії
2. Навігація до історій автора

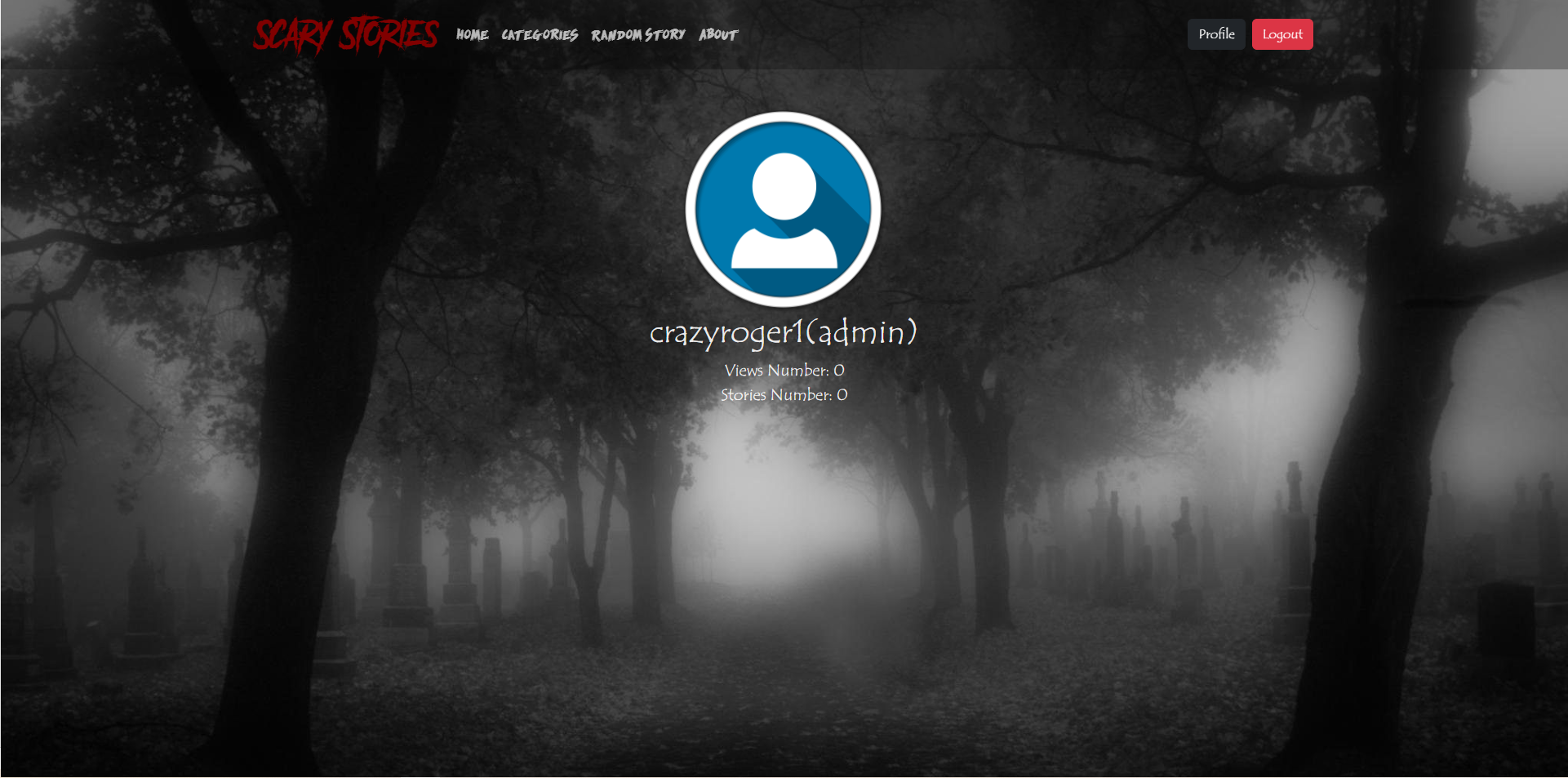


Рис. 2.4.2.5 Сторінка профілю

Сторінка профілю надає такі можливості як:

1. Перегляд статистики

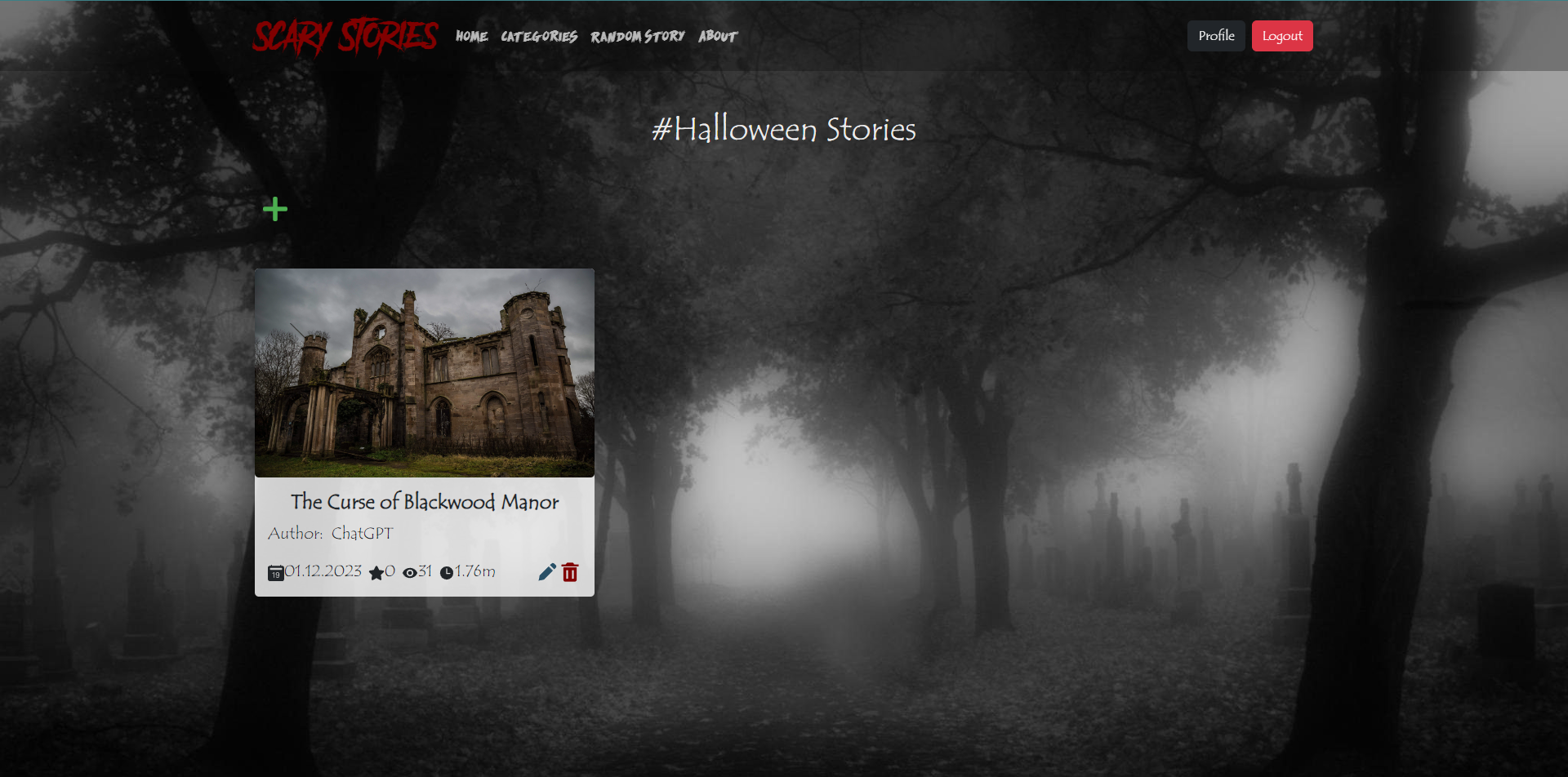


Рис. 2.4.2.6 Сторінка обраної категорії

Сторінка категорії надає такі можливості як:

1. Додавання історії (для зареєстрованих користувачів)
2. Видалення історії (для власника або адміністратора)
3. Редагування історії (для власника або адміністратора)

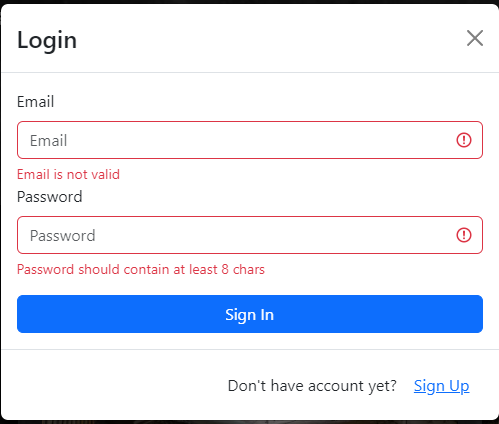


Рис. 2.4.2.7 Модальне вікно логіну користувача

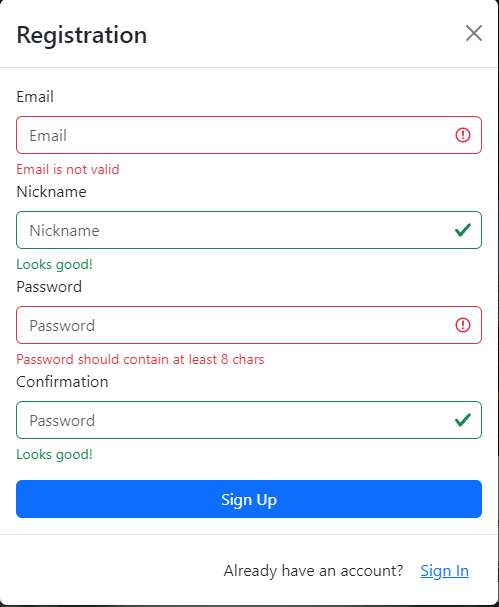


Рис. 2.4.2.8 Модальне вікно реєстрації користувача

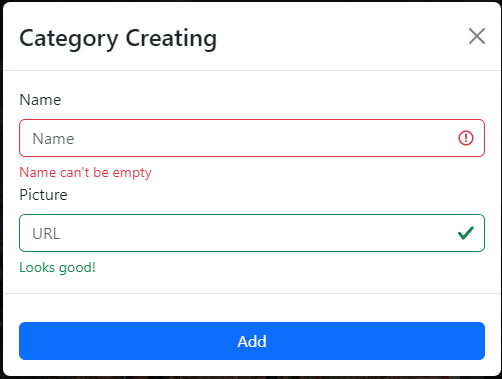


Рис. 2.4.2.9 Модальне вікно створення категорії

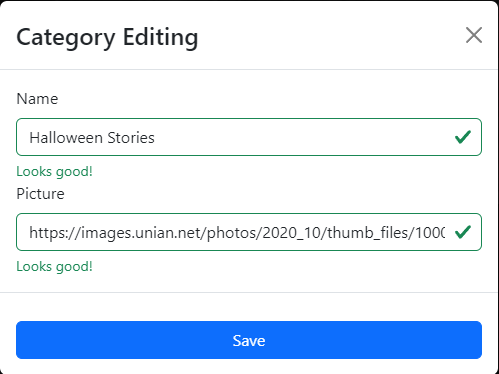


Рис. 2.4.2.10 Модальне вікно редагування категорії

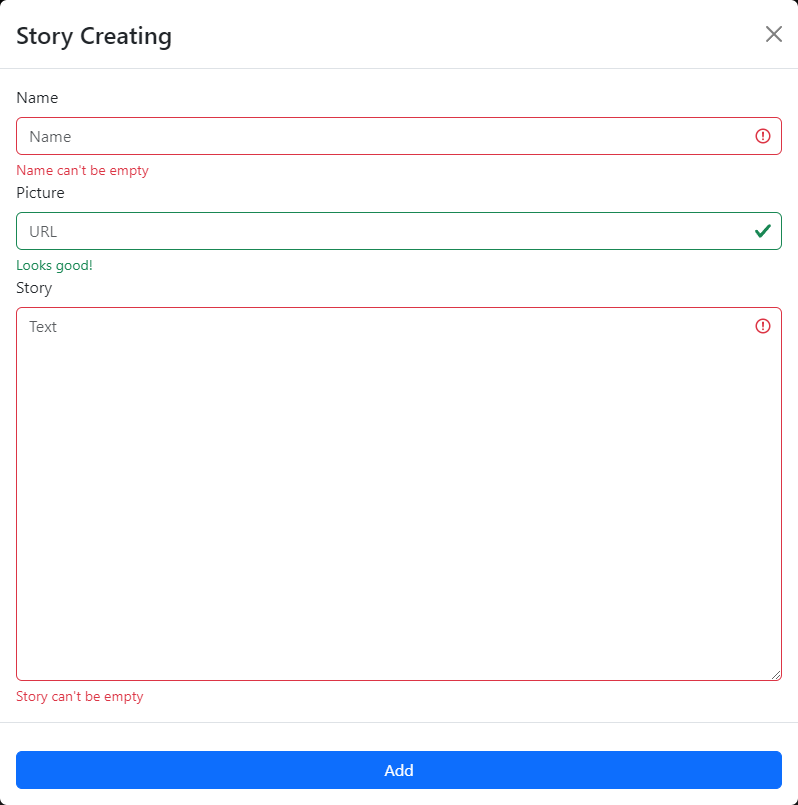


Рис. 2.4.2.11 Модальне вікно створення історії

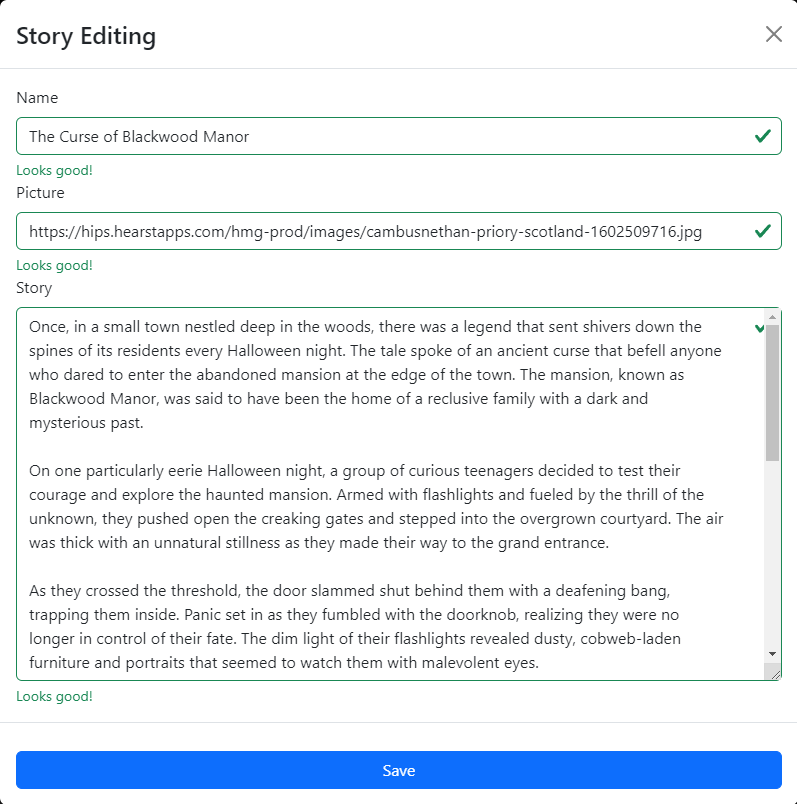


Рис. 2.4.2.12 Модальне вікно редагування історії

2.4.3 Повідомлення користувачеві

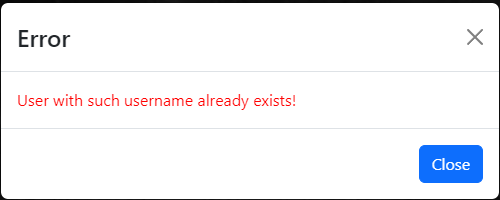


Рис. 2.4.3.1 Модальне вікно похибки, коли користувач з таким іменем вже існує

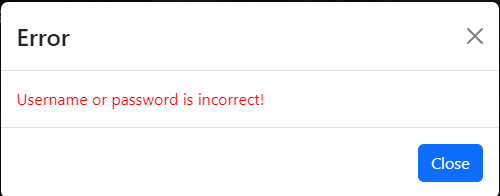


Рис. 2.4.3.2 Модальне вікно похибки, коли пароль не правильний

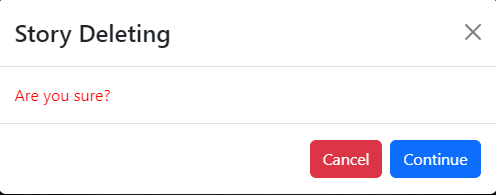


Рис. 2.4.3.3 Модальне вікно підтвердження видалення історії

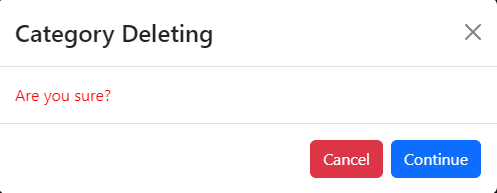


Рис. 2.4.3.4 Модальне вікно підтвердження видалення категорії

ВИСНОВКИ

Результатом виконання курсового проекту є сайт “Scary Stories”, який дозволяє обмінюватись авторськими літературними творами з використанням бази даних, мов програмування Python та JS.

Основна мета проекту - створення віртуального середовища для обміну авторськими страшними історіями - була досягнута. Всі вимоги – виконані.

Розроблена архітектура та функціональні можливості веб-сайту не лише відповідають сучасним технологічним вимогам, але й враховують потреби різноманітних користувачів. Платформа вдало поєднує можливість висловлювання особистих почуттів і думок з можливістю знаходження рівнодумців та отримання підтримки від інших учасників.

Отже, курсовий проект зі створення платформи для обміну авторськими творами з використанням баз даних є успішно виконаним і відповідає вимогам сучасних веб-сайтів.

СПИСОК ВИКОРИСТАННИХ ДЖЕРЕЛ

1. Документація React [Електроний ресурс]: <https://react.dev/>
2. Документація React-Bootstrap [Електроний ресурс]: <https://react-bootstrap.netlify.app/>
3. Документація FastAPI [Електроний ресурс]: <https://fastapi.tiangolo.com/>
4. Документація FastAPI-Users [Електроний ресурс]: <https://fastapi-users.github.io/fastapi-users/12.1/>
5. Документація SQLAlchemy [Електроний ресурс]: <https://www.sqlalchemy.org/>
6. В
7. В
8. В

ДОДАТКИ

src/repository.py

from abc import ABC, abstractmethod  
from typing import Iterable  
  
from sqlalchemy import insert, select, delete, update, func  
  
from src.database import async\_session\_maker, Base  
  
  
class AbstractRepository(ABC):  
 @abstractmethod  
 async def create(self, returning\_fields: tuple[str] | None = ('id',), \*\*data): ...  
  
 @abstractmethod  
 async def read(self, \*filters, limit: int = None, offset: int = None): ...  
  
 @abstractmethod  
 async def read\_one(self, \*filters): ...  
  
 @abstractmethod  
 async def update(self, \*filters, \*\*data) -> None: ...  
  
 @abstractmethod  
 async def delete(self, \*filters) -> None: ...  
  
 @abstractmethod  
 async def exists(self, \*filters) -> bool: ...  
  
 @abstractmethod  
 async def count(self, \*filters) -> int: ...  
  
  
class SQLAlchemyRepository(AbstractRepository):  
 model: type[Base] = None  
  
 async def create(self, returning\_fields: tuple[str] = ('id',), \*\*data):  
 async with async\_session\_maker() as session:  
 stmt = insert(self.model).values(\*\*data)  
  
 if returning\_fields:  
 stmt = stmt.returning(  
 \*(getattr(self.model, field) for field in returning\_fields)  
 )  
 result = await session.execute(stmt)  
 await session.commit()  
  
 if returning\_fields:  
 return result.scalar\_one()  
 # return result.scalars().all()  
  
 async def \_read(self, \*filters, limit: int = None, offset: int = None, sorts: list = None):  
 async with async\_session\_maker() as session:  
 query = select(self.model)  
  
 if limit:  
 query = query.limit(limit)  
  
 if offset:  
 query = query.offset(offset)  
  
 if filters:  
 query = query.filter(\*filters)  
  
 if isinstance(sorts, Iterable):  
 query = query.order\_by(\*sorts)  
  
 result = await session.execute(query)  
 return result  
  
 async def read(self, \*filters, limit: int = None, offset: int = None, sorts: list = None) -> list:  
 result = await self.\_read(\*filters, limit=limit, offset=offset, sorts=sorts)  
 return result.scalars().all()  
  
 async def read\_one(self, \*filters):  
 result = await self.\_read(\*filters)  
 return result.scalars().first()  
  
 async def update(self, \*filters, \*\*data):  
 async with async\_session\_maker() as session:  
 stmt = update(self.model).filter(\*filters).values(\*\*data)  
 await session.execute(stmt)  
 await session.commit()  
  
 async def delete(self, \*filters):  
 async with async\_session\_maker() as session:  
 stmt = delete(self.model).filter(\*filters)  
 await session.execute(stmt)  
 await session.commit()  
  
 async def exists(self, \*filters) -> bool:  
 result = await self.\_read(\*filters)  
 return result.first() is not None  
  
 async def count(self, \*filters) -> int:  
 async with async\_session\_maker() as session:  
 query = select(func.count()).select\_from(self.model)  
  
 if filters:  
 query = query.filter(\*filters)  
  
 result = await session.execute(query)  
 return result.scalar()

src/database.py

from typing import AsyncGenerator  
  
from sqlalchemy import MetaData  
from sqlalchemy.ext.asyncio import create\_async\_engine, async\_sessionmaker, AsyncSession  
from sqlalchemy.orm import declarative\_base  
  
from src.config import settings  
  
engine = create\_async\_engine(  
 f'postgresql+asyncpg://{settings.db\_user}:{settings.db\_pass}'  
 f'@{settings.db\_host}:{settings.db\_port}/{settings.db\_name}'  
)  
  
async\_session\_maker = async\_sessionmaker(engine, expire\_on\_commit=False)  
  
metadata = MetaData()  
Base = declarative\_base(metadata=metadata)  
  
  
async def get\_async\_session() -> AsyncGenerator[AsyncSession, None]:  
 async with async\_session\_maker() as session:  
 yield session

src/auth/models.py

from datetime import datetime  
  
import sqlalchemy as db  
from sqlalchemy.ext.asyncio import AsyncSession  
from fastapi import Depends  
from fastapi\_users.db import SQLAlchemyBaseUserTable  
from fastapi\_users\_db\_sqlalchemy import SQLAlchemyUserDatabase  
  
from src.database import Base, get\_async\_session  
  
  
class User(SQLAlchemyBaseUserTable[int], Base):  
 \_\_tablename\_\_ = 'users'  
  
 id = db.Column(db.Integer, primary\_key=True)  
 email = db.Column(db.String, nullable=False)  
 nickname = db.Column(db.String, nullable=False)  
 registered\_at = db.Column(db.TIMESTAMP, default=datetime.utcnow)  
 hashed\_password: str = db.Column(db.String(length=1024), nullable=False)  
 is\_active: bool = db.Column(db.Boolean, default=True, nullable=False)  
 is\_superuser: bool = db.Column(db.Boolean, default=False, nullable=False)  
 is\_verified: bool = db.Column(db.Boolean, default=False, nullable=False)  
 photo\_url = db.Column(db.String, nullable=True)  
  
  
async def get\_user\_db(session: AsyncSession = Depends(get\_async\_session)):  
 yield SQLAlchemyUserDatabase(session, User)

src/auth/repositories.py

from src.repository import SQLAlchemyRepository  
  
from .models import User  
  
  
class UsersRepository(SQLAlchemyRepository):  
 model = User

src/auth/services.py

from fastapi import HTTPException  
  
from src.repository import AbstractRepository  
from src.serializer import AbstractSerializer  
from .models import User  
from .schemas import UserReadSchema  
from .enums import ErrorMessages  
  
  
class UsersService:  
 def \_\_init\_\_(self, users\_repository: AbstractRepository, users\_serializer: AbstractSerializer):  
 self.users\_repository = users\_repository  
 self.users\_serializer = users\_serializer  
  
 async def get\_user\_or\_404(self, user\_id: int):  
 user = await self.users\_repository.read\_one(User.id == user\_id)  
 if not user:  
 raise HTTPException(status\_code=404, detail=ErrorMessages.NOT\_FOUND)  
 return user  
  
 async def is\_superuser\_or\_403(self, user: User):  
 if not user.is\_superuser:  
 raise HTTPException(status\_code=403, detail=ErrorMessages.IS\_NOT\_SUPERUSER)  
  
 async def read\_user(self, user\_id: int):  
 return self.users\_serializer.serialize(  
 await self.get\_user\_or\_404(user\_id=user\_id),  
 UserReadSchema  
 )

src/statistics/services.py

from src.auth.services import UsersService  
from src.stories.services import StoriesService  
  
from .schemas import UserStatisticsReadSchema  
  
  
class UsersStatisticsService:  
 def \_\_init\_\_(self, users\_service: UsersService, stories\_service: StoriesService):  
 self.users\_service = users\_service  
 self.stories\_service = stories\_service  
  
 async def read\_user\_statistics(self, user\_id: int):  
 user = await self.users\_service.get\_user\_or\_404(user\_id=user\_id)  
 return UserStatisticsReadSchema(  
 stories\_number=await self.stories\_service.count\_user\_stories(user=user),  
 views\_number=await self.stories\_service.count\_author\_views(author=user)  
 )

src/stories/repositories/stories.py

from sqlalchemy import func, select  
  
from src.repository import SQLAlchemyRepository  
from src.stories.models import Story  
from src.database import async\_session\_maker  
  
  
class StoriesRepository(SQLAlchemyRepository):  
 model = Story  
  
 async def get\_random\_story(self) -> Story:  
 async with async\_session\_maker() as session:  
 statement = select(self.model).order\_by(func.random()).limit(1)  
 result = await session.execute(statement)  
 random\_story = result.scalar\_one\_or\_none()  
 return random\_story  
  
 async def count\_author\_views(self, author\_id: int) -> int:  
 async with async\_session\_maker() as session:  
 query = select(func.sum(Story.views)).select\_from(self.model)  
 query = query.filter(Story.author\_id == author\_id)  
 result = await session.execute(query)  
 return result.scalar() or 0

src/stories/repositories/categories.py

from src.repository import SQLAlchemyRepository  
from src.stories.models import Category  
  
  
class CategoriesRepository(SQLAlchemyRepository):  
 model = Category

src/stories/services/categories.py

import sqlalchemy.exc  
from fastapi import HTTPException  
  
from src.repository import AbstractRepository  
from src.serializer import AbstractSerializer  
from src.stories.schemas import CategoryReadSchema, CategoryCreateUpdateSchema  
from src.stories.models import Category  
from .enums import ErrorMessages  
from src.pagination.paginators import Paginator  
  
  
class CategoriesService:  
 def \_\_init\_\_(self, category\_repository: AbstractRepository, category\_serializer: AbstractSerializer):  
 self.category\_repository = category\_repository  
 self.category\_serializer = category\_serializer  
  
 async def get\_category\_or\_404(self, category\_id: int) -> Category:  
 category = await self.category\_repository.read\_one(Category.id == category\_id)  
 if not category:  
 raise HTTPException(status\_code=404, detail=ErrorMessages.CATEGORY\_NOT\_FOUND)  
 return category  
  
 async def name\_is\_unique\_or\_403(self, name: str):  
 category = await self.category\_repository.read\_one(Category.name == name)  
 if category:  
 raise HTTPException(status\_code=403, detail=ErrorMessages.CATEGORY\_WITH\_NAME\_ALREADY\_EXISTS)  
  
 async def read\_categories(self, pagination\_params: Paginator):  
 return self.category\_serializer.serialize\_many(  
 await self.category\_repository.read(limit=pagination\_params.limit, offset=pagination\_params.offset),  
 CategoryReadSchema  
 )  
  
 async def read\_category(self, category\_id: int):  
 return self.category\_serializer.serialize(  
 await self.get\_category\_or\_404(category\_id=category\_id),  
 CategoryReadSchema  
 )  
  
 async def create\_category(self, category: CategoryCreateUpdateSchema):  
 await self.name\_is\_unique\_or\_403(category.name)  
 category\_id = await self.category\_repository.create(\*\*category.model\_dump())  
 return CategoryReadSchema(name=category.name, id=category\_id)  
  
 async def update\_category(self, category\_id: int, category: CategoryCreateUpdateSchema) -> None:  
 await self.get\_category\_or\_404(category\_id=category\_id)  
  
 category\_from\_db = await self.category\_repository.read\_one(Category.name == category.name)  
 if category\_from\_db and category\_from\_db.id != category\_id:  
 raise HTTPException(status\_code=403, detail=ErrorMessages.CATEGORY\_WITH\_NAME\_ALREADY\_EXISTS)  
  
 await self.category\_repository.update(Category.id == category\_id, \*\*category.model\_dump())  
  
 async def delete\_category(self, category\_id: int) -> None:  
 await self.get\_category\_or\_404(category\_id=category\_id)  
 try:  
 await self.category\_repository.delete(Category.id == category\_id)  
 except sqlalchemy.exc.IntegrityError:  
 raise HTTPException(403, ErrorMessages.CATEGORY\_IN\_USE)

src/stories/services/stories.py

import datetime  
  
from fastapi import HTTPException  
  
from src.filtering import Filter  
from src.sorting import Sorter  
from src.pagination import Paginator  
from src.serializer import AbstractSerializer  
from src.stories.schemas import StoryCreateSchema, StoryReadSchema, StoriesReadSchema, StoryUpdateSchema  
from src.auth.schemas import UserReadSchema  
from src.auth.models import User  
from src.auth.services import UsersService  
from src.stories.utils.count import get\_read\_time\_min  
from src.stories.models import Story  
from src.stories.repositories import StoriesRepository  
from .categories import CategoriesService  
from .enums import ErrorMessages  
  
  
class StoriesService:  
 def \_\_init\_\_(  
 self,  
 stories\_repository: StoriesRepository,  
 stories\_serializer: AbstractSerializer,  
 users\_serializer: AbstractSerializer,  
 users\_service: UsersService,  
 categories\_service: CategoriesService  
 ):  
 self.stories\_repository = stories\_repository  
 self.stories\_serializer = stories\_serializer  
 self.users\_serializer = users\_serializer  
 self.categories\_service = categories\_service  
 self.users\_service = users\_service  
  
 async def \_increase\_views(self, story: Story):  
 await self.stories\_repository.update(Story.id == story.id, views=story.views + 1)  
  
 async def count\_user\_stories(self, user: User) -> int:  
 return await self.stories\_repository.count(Story.author\_id == user.id)  
  
 async def count\_author\_views(self, author: User) -> int:  
 return await self.stories\_repository.count\_author\_views(author.id)  
  
 async def get\_story\_or\_404(self, story\_id: int) -> Story:  
 story = await self.stories\_repository.read\_one(Story.id == story\_id)  
 if not story:  
 raise HTTPException(status\_code=404, detail=ErrorMessages.STORY\_NOT\_FOUND)  
 return story  
  
 async def is\_story\_owner\_or\_superuser\_or\_403(self, user: User, story: Story):  
 if user.is\_superuser:  
 return  
  
 if user.id == story.author\_id:  
 return  
  
 raise HTTPException(status\_code=403, detail=ErrorMessages.NO\_PERMISSION)  
  
 async def read\_stories(self, pagination\_params: Paginator, filter\_params: Filter, sort\_params: Sorter = None):  
  
 results = []  
 for story in await self.stories\_repository.read(  
 \*filter\_params.filters,  
 limit=pagination\_params.limit,  
 offset=pagination\_params.offset,  
 sorts=sort\_params.sorts  
 ):  
 results.append(StoriesReadSchema(  
 \*\*story.\_\_dict\_\_,  
 rating=0,  
 read\_time=get\_read\_time\_min(story.story),  
 comments\_number=0,  
 author=await self.users\_service.read\_user(story.author\_id),  
 category=await self.categories\_service.read\_category(story.category\_id)  
 ))  
  
 return results  
  
 async def read\_story(self, story\_id: int):  
 story = await self.get\_story\_or\_404(story\_id=story\_id)  
 await self.\_increase\_views(story)  
 return StoryReadSchema(  
 \*\*story.\_\_dict\_\_,  
 rating=0,  
 read\_time=get\_read\_time\_min(story.story),  
 comments\_number=0,  
 author=await self.users\_service.read\_user(story.author\_id),  
 category=await self.categories\_service.read\_category(story.category\_id)  
 )  
  
 async def read\_random\_story(self):  
 story = await self.stories\_repository.get\_random\_story()  
 if not story:  
 raise HTTPException(404, detail=ErrorMessages.STORY\_NOT\_FOUND)  
  
 await self.\_increase\_views(story)  
 return StoryReadSchema(  
 \*\*story.\_\_dict\_\_,  
 rating=0,  
 read\_time=get\_read\_time\_min(story.story),  
 comments\_number=0,  
 author=await self.users\_service.read\_user(story.author\_id),  
 category=await self.categories\_service.read\_category(story.category\_id)  
 )  
  
 async def create\_story(self, story: StoryCreateSchema, user: User):  
 data = story.model\_dump()  
 story\_id = await self.stories\_repository.create(  
 \*\*data,  
 author\_id=user.id,  
 )  
  
 return StoryReadSchema(  
 \*\*data,  
 id=story\_id,  
 rating=0,  
 read\_time=get\_read\_time\_min(story.story),  
 views=0,  
 create\_date=datetime.datetime.utcnow(),  
 comments\_number=0,  
 author=self.users\_serializer.serialize(user, UserReadSchema),  
 category=await self.categories\_service.read\_category(story.category\_id)  
 )  
  
 async def update\_story(self, story\_id: int, story: StoryUpdateSchema, user: User):  
 story\_from\_db = await self.get\_story\_or\_404(story\_id=story\_id)  
 await self.is\_story\_owner\_or\_superuser\_or\_403(user=user, story=story\_from\_db)  
 await self.stories\_repository.update(Story.id == story\_id, \*\*story.model\_dump())  
  
 async def delete\_story(self, story\_id: int, user: User):  
 story = await self.get\_story\_or\_404(story\_id=story\_id)  
 await self.is\_story\_owner\_or\_superuser\_or\_403(user=user, story=story)  
 await self.stories\_repository.delete(Story.id == story\_id)

src/stories/models.py

import datetime  
  
import sqlalchemy as db  
from sqlalchemy.orm import relationship  
  
from src.database import Base  
from src.auth.models import User  
  
  
class Category(Base):  
 \_\_tablename\_\_ = 'story\_categories'  
  
 id = db.Column(db.Integer, primary\_key=True)  
 name = db.Column(db.String, nullable=False, unique=True)  
 picture\_url = db.Column(db.String, nullable=True)  
  
 stories = relationship('Story', back\_populates='category')  
  
  
class Story(Base):  
 \_\_tablename\_\_ = 'stories'  
  
 id = db.Column(db.Integer, primary\_key=True)  
 name = db.Column(db.String, nullable=False)  
 story = db.Column(db.String, nullable=False)  
 author\_id = db.Column(db.Integer, db.ForeignKey(User.id, ondelete='SET NULL'), nullable=True)  
 category\_id = db.Column(db.Integer, db.ForeignKey('story\_categories.id', onupdate='CASCADE'))  
 create\_date = db.Column(db.Date, default=datetime.datetime.utcnow)  
 views = db.Column(db.Integer, default=0)  
 picture\_url = db.Column(db.String, nullable=True)  
  
 category = relationship('Category', back\_populates='stories')  
 author = relationship('User')