Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно інформаційних систем і програмної інженерії

(повна назва факультету)

кафедра програмної інженерії

(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр

(назва освітнього ступеня)

на тему: Розробка програмного забезпечення для спільної сценарної роботи з акцентом на структуру сюжету

Виконав(ла): студент(ка) IV курсу, групи СП-41

спеціальності 121 – Інженерія програмного забезпечення

(шифр і назва спеціальності)

_____ Бармак Р. М.

(підпис) (прізвище та ініціали)

Керівник _____ Бревус В. М.

(підпис) (прізвище та ініціали)

Нормоконтроль _____ Стоянов Ю. М.

(підпис) (прізвище та ініціали)

Завідувач кафедри _____ Петрик М. Р.

(підпис) (прізвище та ініціали)

Рецензент _____ Луцик Н. С,

(підпис) (прізвище та ініціали)

Тернопіль
2026

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет Комп'ютерно інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра Програмної інженерії
(повна назва кафедри)

Затверджую
Завідувач кафедри
Петрик М.Р.
(підпис) (прізвище та ініціали)
« » 2026 р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавр
(назва освітнього ступеня)
за спеціальністю 121 – Інженерія програмного забезпечення
(шифр і назва спеціальності)
студенту Бармак Роман Миколайович
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка програмного забезпечення для спільної сценарної роботи з акцентом на структуру сюжету

Керівник роботи Бревус Віталій Миколайович, PhD
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
Затверджені наказом ректора від «___» _____ 20___ року № _____.
2. Термін подання студентом завершеної роботи _____
3. Вихідні дані до роботи Аналіз предметної області, специфікації функціональних і нефункціональних вимог застосунку

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ. 1. Аналіз предметної області. 1.1. Постановка задачі розробки платформи для сценарної роботи. 1.2. Огляд існуючих рішень та їх обмежень. 1.3. Специфікація функціональних та нефункціональних вимог. 2. Проєктування системи. 2.1. Архітектура ПЗ. 2.2. Проєктування контрактів та OpenAPI специфікацій. 2.3. Модель бази даних. 2.4. Архітектура клієнтської частини. 3. Програмна реалізація та тестування. 3.1. Реалізація серверної частини. 3.2. Реалізація клієнтської частини. 3.3. Інтеграція бази даних за допомогою інструментів об'єктно-реляційної проєкції. 3.4. Тестування функціональності. 4. Безпека життєдіяльності,
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових слайдів)
1. Порівняльна таблиця існуючих рішень. 2. Структура репозиторія. 3. ER-діаграма моделей бази даних. 4. Архітектура серверної частини. 5. Інтерфейс Knowledge Base. 6. Екран редактору сценарії. 9. Результати автоматизованого тестування. 10. Результати тестування продуктивності інструментами веб-браузера.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Ознайомлення з завданням до кваліфікаційної роботи		
2	Аналіз технічного завдання, пошук та аналіз наукових		
	джерел з веб-розробки		
3	Огляд існуючих рішень		
4	Проєктування архітектури ПЗ, API та бази даних		
5	Налаштування середовища розробки		
6	Реалізація серверної частини		
7	Налаштування клієнтської частини		
8	Тестування функціональності, оптимізація швидкодії,		
	виправлення проблем виявлених під час тестування		
9	Підготовка проміжного звіту роботи		
10	Написання розділу 4: «Охорона праці та безпека		
	життєдіяльності»		
11	Оформлення висновків, списку використаних джерел,		
	додатків		
12	Нормоконтроль		
13	Перевірка на плагіат		
14	Попередній захист та внесення коригувань		
15	Захист кваліфікаційної роботи		

Студент _____ Бармак Р. М.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Бревус В. М.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Розробка програмного забезпечення для спільної сценарної роботи з акцентом на структуру сюжету // Кваліфікаційна робота освітнього рівня «Бакалавр» // Бармак Роман Миколайович // Тернопільський національний технічний університет імені Івана Пулюя, Факультет комп'ютерно інформаційних систем і програмної інженерії, кафедра програмної інженерії, група СП-41 // Тернопіль, 2026 // Ст. – 78, рис. – 9, табл. – 4, кресл. – 0, додат. – 3.

Ключові слова: спільна сценарна робота, структура сюжету, вебзастосунок, сюжетні сутності.

У кваліфікаційній роботі досліджено проблему підтримки спільної сценарної роботи в межах єдиного програмного середовища та запропоновано прототип програмної системи, що поєднує робочі простори, сценарії, версії тексту, сюжетні сутності, базу знань і механізми керування доступом. У першому розділі проаналізовано предметну область, розглянуто наявні аналоги та сформульовано вимоги до системи. У другому розділі описано логічну архітектуру, модель даних, технологічний стек, підхід до автентифікації та основні реалізаційні рішення. У третьому розділі наведено результати тестування, особливості розгортання та верифікації програмної системи.

Об'єкт дослідження – програмне забезпечення, процеси та інструментальні засоби його розробки, супроводження і забезпечення якості в контексті підтримки спільної сценарної роботи.

Предмет дослідження – методи, моделі та програмні засоби реалізації програмної системи для створення, редагування, організації та супроводу сценарних матеріалів із підтримкою структури сюжету.

ABSTRACT

Development of software for collaborative screenwriting with emphasis on plot structure // Qualification work for the educational level "Bachelor" // Barmak Roman Mykolaiovych // Ternopil Ivan Puluj National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Software Engineering, group SP-41 // Ternopil, 2026 // Pp. – 78, Fig. – 9, Table – 4, Drawings – 0, Appendices – 3.

Keywords: collaborative screenwriting, plot structure, web application, narrative entities.

The qualification work investigates the problem of supporting collaborative screenwriting within a single software environment and presents a software system that combines workspaces, scripts, text versions, plot-structure entities, knowledge-base objects, and access-control mechanisms. The first section analyzes the subject area, reviews existing solutions, and defines the system requirements. The second section describes the architecture, data model, technology stack, authentication approach, and key implementation decisions. The third section presents the testing results, deployment specifics, and system verification approach.

Object of research – the process of designing and developing software for collaborative screenwriting support.

Subject of research – methods, models, and software tools used to implement a software system for screenplay creation, editing, organization, and plot-structure support.

ЗМІСТ

ВСТУП

Актуальність теми зумовлена тим, що сучасна сценарна робота над фільмами, серіалами та іншими екранними творами дедалі частіше виходить за межі простого написання тексту. Сценарист або творча команда одночасно працюють із серіями, версіями сценарію, персонажами, локаціями, сценами, сюжетними арками, тематичними зв'язками та коментарями учасників. На практиці ці аспекти нерідко розподілені між кількома інструментами, що ускладнює підтримку цілісного творчого процесу, збільшує кількість контекстних перемикань і знижує керованість командної взаємодії.

Розв'язання цієї проблеми потребує створення єдиного програмного середовища, яке поєднує робочі простори, сценарне редагування, базу знань, сюжетні сутності та механізми контролю доступу. Така інтеграція є особливо важливою для веб орієнтованих систем, у яких користувач очікує отримати не окремий редактор, а цілісну платформу для спільної сценарної роботи. Отже, тема розробки програмного забезпечення для спільної сценарної роботи з акцентом на структуру сюжету є актуальною як у прикладному, так і в інженерному аспектах.

Метою кваліфікаційної роботи є вирішення прикладної задачі проєктування та реалізації прототипу програмної системи для спільної сценарної роботи з акцентом на структуру сюжету, який забезпечує ведення робочих просторів, редагування сценаріїв, підтримку версій, роботу із сюжетними сутностями, базою знань і контроль доступу користувачів.

Об'єкт дослідження – програмне забезпечення, процеси та інструментальні засоби його розробки, супроводження і забезпечення якості в контексті підтримки спільної сценарної роботи.

Предмет дослідження – архітектурні, інформаційні та програмні рішення, застосовані під час створення програмної системи для спільної сценарної роботи з підтримкою структури сюжету.

Практичне значення роботи полягає у створенні прототипу програмної системи, який забезпечує централізовану роботу зі сценарними матеріалами,

ролями доступу, базою знань і структурою сюжету в межах єдиного веб середовища.

Наукова новизна одержаних результатів полягає в обґрунтуванні та практичній реалізації підходу до побудови веб орієнтованої системи з використанням методології специфікаційно-орієнтованої розробки програмного забезпечення. Додатково новизна проявляється у використанні спільного контрактного шару для узгодження клієнтської та серверної частин системи, що підсилює простежуваність між вимогами, реалізацією та перевіркою результатів, а також реалізації підходу до побудови веб орієнтованої системи, у межах єдиного програмного середовища якої поєднано сценарне редагування, рольовий доступ, базу знань і формалізовані елементи структури сюжету

Практичне значення одержаних результатів полягає не лише у створенні працездатного прототипу, а й у можливості використання запропонованих архітектурних і програмних рішень як основи для подальшого розвитку подібних інформаційних систем творчого спрямування. Отримані результати також можуть бути використані як приклад поєднання типобезпечного технологічного стеку, спільного контрактного шару та рольового доступу в межах веб проекту.

Апробацію кваліфікаційної роботи здійснено шляхом висвітлення окремих її положень у науково-технічних публікаціях автора, присвячених специфікаційно-орієнтованій розробці програмного забезпечення із застосуванням інструментів штучного інтелекту, а також інтеграції деталізованих дозволів у процес авторизації користувачів системи.

РОЗДІЛ 1 АНАЛІЗ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

Перший розділ присвячений формалізації вихідних передумов розробки програмної системи. У ньому послідовно розглядаються проблема фрагментованої сценарної роботи, місце розроблюваної системи серед наявних аналогів, формалізовані вимоги, актори та ключові варіанти використання. Саме цей розділ створює логічний перехід від предметної області до подальшого архітектурного і програмного проєктування.

1.1 Аналіз предметної області

Предметна область цієї роботи охоплює не лише безпосереднє написання сценарного тексту, а весь супровідний процес, у якому бере участь сценарист або творча команда. На практиці сценарна робота майже ніколи не обмежується одним редактором: окремо ведуться начерки сцен, нотатки про персонажів, зв'язки між сюжетними лініями, відомості про локації, версії тексту, коментарі співредакторів і матеріали для подальшого експорту. При цьому сама структура сценарію передбачає чітку композиційну організацію – поділ на акти, сцени та сюжетні повороти. Якщо ці артефакти розподілені між різними інструментами, користувач постійно перемикається між контекстами, а узгодженість історії підтримується вручну.

Для індивідуального автора така фрагментація означає втрату цілісного бачення сюжету, дублювання даних і складність навігації між текстом та довідковими сутностями. Для команди проблема поглиблюється: додаються керування доступом, історія змін, розмежування прав, запрошення учасників і ризик неузгодженого редагування [1]. Саме тому в межах предметної області важливо розглядати сценарну систему не як простий редактор, а як робоче середовище, що має об'єднати текст, сюжетну структуру, базу знань і механізми спільної роботи.

Для уточнення місця розроблюваної системи варто зіставити її з поширеними продуктами, які вже використовуються для написання сценаріїв, спільної роботи або підготовки матеріалів до виробництва. На основі офіційного позиціонування Final Draft, Celtx, WriterDuet/WriterSolo та Highland можна виокремити такі характеристики [2-5].

1.1.1 Final Draft

Спільна робота: підтримується через Final Draft Cloud і засоби спільного доступу.

Представлення структури сюжету: підтримується частково через засоби структурного планування та інструменти розвитку сюжету.

Версійність і ревізії: підтримуються частково з орієнтацією на робочий процес ревізій і підготовки до виробництва.

База знань персонажів і локацій: слабо виражена як окрема доменна підсистема.

Експорт: підтримується.

Веб доступ: реалізовано лише частково, оскільки настільне ядро доповнюється хмарним компонентом.

1.1.2 Celtx

Спільна робота: підтримується через засоби спільного написання, коментарі, погодження та індикацію присутності учасників.

Представлення структури сюжету: підтримується через план ключових сюжетних подій, розкадрування та інструменти розвитку сюжету.

Версійність і ревізії: підтримується історія ревізій.

База знань персонажів і локацій: підтримується частково через каталоги й інструменти виробничого планування, але не як цілісна модель бази знань для сюжетних сутностей.

Експорт: підтримується.

Веб доступ: підтримується повноцінно.

1.1.3 WriterSolo

Спільна робота: підтримується обмежено, оскільки продукт орієнтований на легке автономне використання, а частина функцій недоступна без повнішого сервісного контуру.

Представлення структури сюжету: підтримується обмежено, з акцентом переважно на базовому сценарному письмі.

Версійність і ревізії: підтримуються обмежено.

База знань персонажів і локацій: не реалізована як окремий структурований блок.

Експорт: підтримується частково.

Веб доступ: підтримується.

1.1.4 Highland

Спільна робота: підтримується обмежено, оскільки продукт переважно орієнтований на індивідуального автора.

Представлення структури сюжету: підтримується частково через навігаційну панель, сцени, синопсиси, нотатки та списки персонажів і локацій.

Версійність і ревізії: підтримується режим ревізій.

База знань персонажів і локацій: підтримується частково, але без повноцінної міжсутнісної моделі знань.

Експорт: підтримується.

Веб доступ: відсутня як повноцінна веб версія, оскільки продукт орієнтований на Mac, iPad та iPhone.

Порівняння показує, що наявні рішення зазвичай сильні в одному або двох напрямах, але рідко поєднують усі потрібні властивості в одному середовищі. Final Draft є зрілим професійним інструментом сценарного форматування і підготовки матеріалів, але не акцентує предметно-орієнтовану базу знань і сюжетні сутності як центральну модель системи. Celtx, навпаки, пропонує сильніші засоби спільної роботи та story development, однак його фокус

зміщується в бік ширшого production workflow. Highland і WriterSolo зручні як інструменти для безпосереднього письма, але в меншій мірі покривають керування ролями, структуровану сюжетну модель та глибоку інтеграцію командної роботи.

Отже, прогалина полягає в недостатній представленості веб-орієнтованої системи, яка одночасно підтримує робочі простори, рольовий доступ, сценарне редагування, базу знань і формалізовані елементи структури сюжету, зокрема сцени, теми та сюжетні арки. Саме цю прогалину і покликана закрити розроблювана програмна система.

Ключові проблеми предметної області, які випливають із проведеного аналізу, можна звести до такого переліку:

– Розрив між написанням сценарного тексту та веденням довідкових сутностей історії.

– Відсутність єдиного контексту для ролей, запрошень і перевірки доступу в командній роботі.

– Недостатня формалізація сцен, тем і сюжетних арок як окремих елементів структури сюжету.

– Необхідність постійного перемикання між кількома інструментами для письма, нотаток, ревізій і експорту.

1.2 Постановка завдання та цілей

Після аналізу предметної області необхідно перейти до формалізованої постановки завдань. У межах цієї роботи програмна система розглядається не як абстрактний редактор тексту, а як інтегроване середовище для спільної сценарної роботи, у якому поєднуються керування доступом, сценарне редагування, сюжетна структура та база знань. Саме тому вимоги до системи доцільно подати у вигляді окремих груп: ціль роботи, функціональні вимоги, нефункціональні вимоги та свідомо зафіксовані межі поточної реалізації [6-7].

1.2.1 Мета роботи

Мета кваліфікаційної роботи полягає у вирішенні прикладної задачі проєктування та реалізації прототипу програмної системи для спільної сценарної роботи з акцентом на структуру сюжету, який забезпечує кероване створення, редагування та організацію сценарних матеріалів у межах спільного робочого простору. Важливо підкреслити, що мета стосується не лише реалізації окремих функцій, а побудови цілісного програмного середовища, у якому сценарний текст, сюжетні сутності, база знань та контроль доступу утворюють єдину систему.

1.2.2 Функціональні вимоги

Функціональні вимоги до системи випливають із поєднання базового сценарного контуру, механізмів спільної роботи та спеціалізованих засобів опису сюжетної структури. По-перше, система повинна підтримувати створення, редагування та організацію серій як контейнерів для сценарних матеріалів. По-друге, вона має надавати засоби роботи зі сценаріями та їхніми версіями, включаючи відкриття, збереження, повторне завантаження й експорт результатів.

Окремий блок функціональних вимог пов'язаний із підтримкою спільної роботи. Система повинна дозволяти створювати робочі простори, автоматично формувати персональний робочий простір для нового користувача, запрошувати учасників, надавати їм ролі та обмежувати доступ до ресурсів відповідно до ієрархії прав [8]. У межах поточної реалізації важливими функціями є також прийняття або відхилення запрошень, керування складом команди та перевірка доступу до матеріалів конкретного робочого простору.

Ще одну суттєву групу становлять вимоги до підтримки предметної моделі сценарію. Система повинна забезпечувати ведення бази знань персонажів, локацій, реквізиту та інших довідкових сутностей, а також окремо підтримувати елементи структури сюжету: сцени, теми, сюжетні арки й пов'язані між ними зв'язки. Таким чином, програмне забезпечення повинно працювати не лише з текстом сценарію, а й із формалізованим описом драматургічних компонентів історії [9-10].

До функціональних вимог також належать автентифікація користувача, авторизація та захист процедур, експорт результатів, а також інтеграція між сценарним редактором і пов'язаними сутностями бази знань. Сукупно це означає, що система має підтримувати такі базові сценарії: вхід у систему, роботу в робочому просторі, створення серії, редагування сценарію, використання сюжетних сутностей, запрошення учасників і виконання операцій із перевіркою прав доступу.

1.2.3 Нефункціональні вимоги

Нефункціональні вимоги спрямовані на забезпечення узгодженості, підтримуваності та відтворюваності системи. Насамперед важливою є узгодженість між клієнтською і серверною моделями даних, яка в цій роботі досягається через спільний контрактний шар. Не менш важливою вимогою є підтримка масштабованої структури коду, оскільки система охоплює кілька великих предметних блоків: спільну роботу, сценарії, базу знань, сюжетні сутності та експорт.

До нефункціональних вимог належить і можливість локального розгортання системи для розробки та тестування. Це передбачає наявність контрольованого середовища запуску, конфігурації інфраструктури та повторюваного робочого процесу для команди або окремого розробника. Окремо слід виділити вимоги до якості програмної реалізації: перевірка типів, лінтинг, автоматизовані тести та відтворювані сценарії верифікації.

Крім того, для окремих функціональних блоків доцільно фіксувати орієнтовні вимоги до продуктивності. Наприклад, перевірка прав доступу має виконуватися швидко навіть за багаторазових звернень, пошук і робота з сутностями бази знань не повинні втрачати чутливість на реалістичних наборах даних, а користувацькі сценарії завантаження основних сторінок мають залишатися прийнятними з погляду інтерактивності [11].

1.2.4 Валідація вимог

Формулювання вимог саме по собі ще не гарантує, що вони коректно відображають потреби зацікавлених сторін. Тому наступним кроком є валідація вимог, тобто отримання достатньої впевненості в тому, що зафіксовані функціональні та нефункціональні вимоги повно, точно й зрозуміло представляють актуально усвідомлені потреби користувачів, замовника та команди розробки [6-7]. У межах такої валідації доцільно відповісти на кілька базових запитань: чи охоплюють вимоги всі релевантні потреби на поточному етапі; чи немає серед них таких, що не відповідають реальним очікуванням стейкхолдерів; чи сформульовано їх однозначно та придатно до реалізації; чи є вони зрозумілими, узгодженими та достатньо повними; чи відповідає спосіб їх документування прийнятим стандартам.

Найпоширенішим методом валідації є рецензування вимог. Для розроблюваної системи це означає перегляд опису вимог, акторів і варіантів використання з кількох перспектив: користувачі й потенційні учасники сценарної співпраці оцінюють, чи повно відображено їхні робочі потреби; фахівці з програмної інженерії перевіряють ясність формулювань, відсутність пропусків і відповідність стандартам; розробники архітектури та реалізації аналізують, чи є цих вимог достатньо для побудови моделі даних, API, механізмів доступу та інтерфейсу. Ефективність такого перегляду підвищують контрольні списки, критерії якості або узгоджене "definition of done" для вимог, що допомагає системно виявляти помилки, неявні припущення, двозначності та суперечності.

Додатковим підходом є симуляція або виконання специфікації. Якщо не всі зацікавлені сторони готові детально аналізувати текстовий опис вимог, частину з них можна перевіряти через проходження демонстраційних сценаріїв, інтерпретацію UML-діаграм, use case flow та послідовне відтворення очікуваної поведінки системи. Для цієї роботи такими сценаріями є, зокрема, вхід до системи, створення робочого простору, запрошення учасника, редагування сценарію та керування сюжетними сутностями. Якщо в ході такої симуляції

специфікація послідовно відтворює очікувані політики й процеси, це підвищує впевненість у її коректності.

Ще одним методом є прототипування, яке доцільне тоді, коли вимоги складно повноцінно перевірити лише текстом або діаграмами. Прототип інтерфейсу чи окремої функціональної підсистеми наочно демонструє інтерпретацію вимог і допомагає виявити помилкові припущення, наприклад щодо динаміки сценарного редактора, навігації між робочими просторами або роботи зі структурою сюжету. Водночас цей підхід має обмеження: увага рецензентів може зміщуватися на косметичні недоліки інтерфейсу, а сама підготовка прототипу потребує додаткових витрат. Проте такі витрати є виправданими, якщо прототип дозволяє своєчасно виявити хибні або неповні вимоги та уникнути дорожчих помилок на етапах реалізації [6-7].

1.3 Пошук акторів та варіантів використання

Для подальшого проєктування системи важливо формально визначити акторів, які взаємодіють із програмним забезпеченням, а також окреслити межі самої системи. У межах цієї роботи доцільно виділити три основні типи акторів. Першим і головним актором є сценарист, який створює серії, працює зі сценаріями, керує елементами структури сюжету та використовує базу знань. Другим актором є учасник спільного робочого простору, який взаємодіє з уже створеними матеріалами відповідно до отриманої ролі. Третім актором є неавторизований користувач, чия взаємодія з системою обмежується входом, реєстрацією та переходом за запрошенням.

Основні групи дій акторів у межах системи можна узагальнити так:

1. Вхід до системи та отримання доступу до захищеного контуру.
2. Робота з робочим простором, серіями та сценаріями.
3. Керування елементами бази знань і структури сюжету.
4. Виконання захищених операцій, пов'язаних із ролями та запрошеннями.

Межа системи повинна охоплювати принаймні п'ять логічних підсистем: автентифікацію і доступ, спільну роботу в межах робочого простору, керування серіями та сценаріями, роботу зі структурою сюжету й базою знань, а також експорт результатів.

На рисунку 1.1 подано UML-діаграму варіантів використання, на якій відображено три основні типи акторів системи: неавторизованого користувача, сценариста та учасника робочого простору.

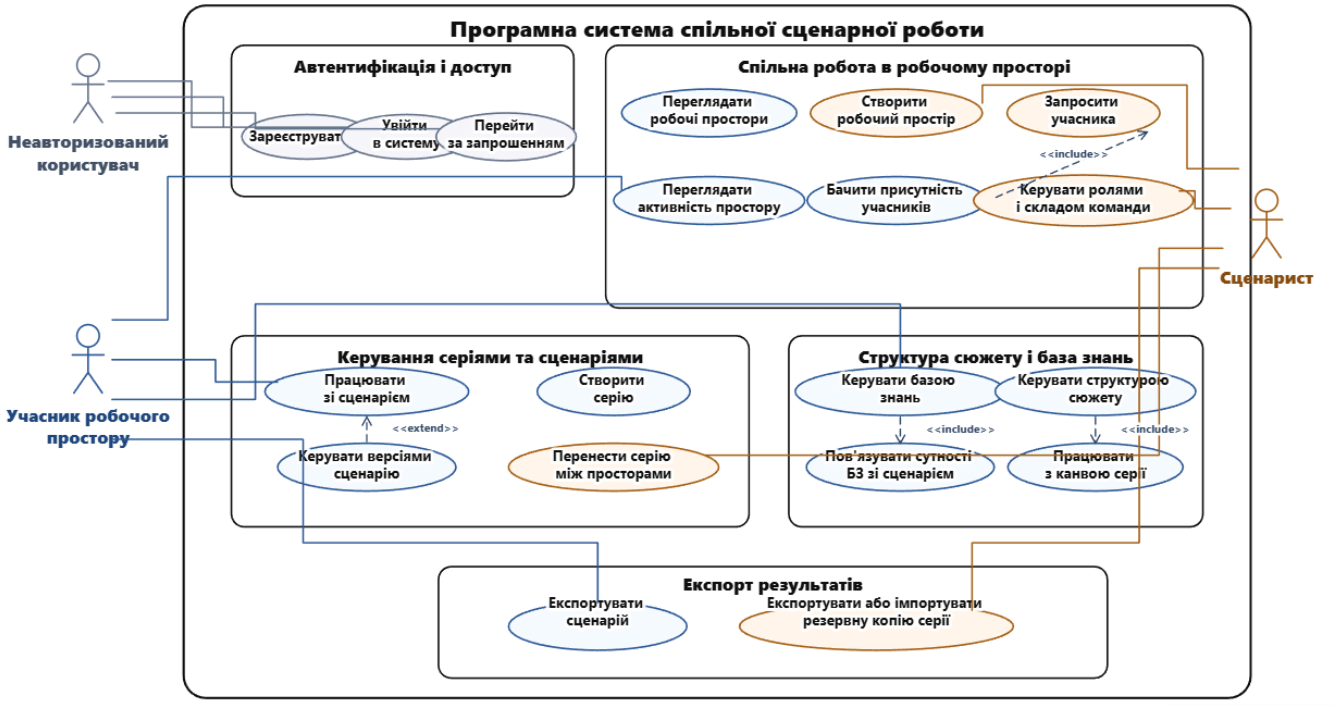

Рисунок 1.1 – UML-діаграма варіантів використання для системи спільної сценарної роботи

Подана діаграма фіксує межу системи та розподіл основних дій між неавторизованим користувачем, учасником робочого простору і сценаристом. Це створює підставу для подальшого текстового опису ключових варіантів використання, у якому кожен сценарій можна конкретизувати через передумови, основний потік і очікуваний результат.

1.4 Опис ключових варіантів використання

Ключові варіанти використання повинні відображати ті сценарії, які найповніше розкривають призначення системи та визначають її подальшу архітектуру. Для цієї роботи доцільно зосередитись на варіантах використання, пов'язаних із входом до системи, роботою в робочому просторі, керуванням серіями, редагуванням сценарію, описом сюжетної структури та перевіркою доступу.

1.4.1 Вхід користувача до системи

Актор: неавторизований користувач.

Передумови: користувач має обліковий запис або проходить процедуру первинної реєстрації.

Основний потік: користувач вводить облікові дані, система виконує автентифікацію, створює або відновлює сесію та надає доступ до персонального або спільного робочого середовища.

Альтернативний потік: у разі некоректних даних або недійсного посилання-запрошення система відхиляє вхід і повідомляє про помилку.

Постумови: користувач переходить до захищеної частини системи та отримує доступ до дозволених ресурсів.

1.4.2 Створення або відкриття робочого простору

Актор: сценарист або авторизований учасник.

Передумови: користувач автентифікований.

Основний потік: користувач відкриває список робочих просторів, створює новий робочий простір або переходить до вже існуючого. Якщо робочий простір створюється вперше, система зберігає його параметри та призначає користувача власником.

Альтернативний потік: якщо користувач не має доступу до робочого простору, система не розкриває його вміст і повертає повідомлення про недоступність ресурсу.

Постумови: користувач опиняється в контексті конкретного робочого простору, у межах якого надалі виконує більшість операцій.

1.4.3 Створення серії та робота зі сценарієм

Актор: сценарист або учасник із правами редагування.

Передумови: користувач має доступ до робочого простору із достатнім рівнем прав.

Основний потік: користувач створює серію, задає її основні метадані, після чого створює або відкриває сценарій у межах цієї серії. Далі він працює в редакторі, змінює вміст, зберігає зміни та за потреби виконує експорт.

Альтернативний потік: якщо операція виконується користувачем із недостатніми правами, система блокує модифікацію або приховує недоступний ресурс.

Постумови: серія та сценарій збережені у системі, а зміни стають доступними в подальших сценаріях роботи.

1.4.4 Створення нової версії сценарію

Актор: сценарист або учасник із правами редагування.

Передумови: у системі вже існує сценарій, прив'язаний до серії.

Основний потік: користувач відкриває сценарій, вносить зміни до тексту та ініціює збереження нового стану. Система фіксує оновлений вміст, час останнього редагування та, за потреби, створює новий версійний зріз для подальшого відстеження історії змін.

Альтернативний потік: у разі помилки збереження система повідомляє користувача і не втрачає попередній узгоджений стан даних.

Постумови: нова версія сценарного матеріалу доступна для подальшого використання та аналізу.

1.4.5 Додавання елементів структури сюжету

Актор: сценарист або учасник із правами редагування.

Передумови: існує серія, у межах якої користувач має право працювати з сюжетними сутностями.

Основний потік: користувач відкриває розділ бази знань або пов'язану вкладку, створює сцену, тему чи сюжетну арку, задає її метадані, пов'язує з персонажами, сценаріями або іншими об'єктами та зберігає результат.

Альтернативний потік: якщо користувач намагається створити некоректно пов'язану сутність або дублює критичний елемент, система виконує валідацію і вимагає виправлення даних.

Постумови: структурний елемент сюжету збережено, і він стає доступним для навігації, пошуку та подальшого аналізу.

1.4.6 Робота з базою знань

Актор: сценарист або учасник робочого простору.

Передумови: користувач має доступ до серії.

Основний потік: користувач додає або редагує персонажа, локацію, реквізит чи іншу сутність бази знань, після чого використовує її у сценарії або в структурі сюжету. Система забезпечує повторне використання цих даних у різних контекстах.

Альтернативний потік: якщо пов'язана сутність не знайдена або недоступна, система не дозволяє створити неконсистентний зв'язок.

Постумови: база знань залишається узгодженою, а її сутності можуть повторно використовуватись у різних сценаріях роботи.

1.4.7 Запрошення учасника та перевірка доступу

Актор: власник робочого простору або адміністратор.

Передумови: користувач має достатній рівень прав у межах робочого простору.

Основний потік: користувач створює запрошення, зазначає адресу електронної пошти та роль нового учасника. Після прийняття запрошення система додає користувача до робочого простору та надає доступ відповідно до ролі.

Альтернативний потік: якщо запрошення прострочене, відхилене або користувач не має прав на його створення, система блокує операцію. Аналогічно, під час спроби виконати захищену дію без належних прав система приховує недоступний ресурс або відмовляє в операції згідно з політикою доступу.

Постумови: склад учасників робочого простору і їхні права доступу залишаються узгодженими зі встановленою роллю кожного користувача.

Отже, у першому розділі визначено практичну проблему, проаналізовано предметну область, сформульовано функціональні та нефункціональні вимоги, окреслено акторів системи та описано ключові сценарії взаємодії. Це дає змогу перейти від загального бачення задачі до її формалізованого представлення в межах програмної системи.

Отримані результати мають безпосереднє значення для подальшого проєктування. Саме на основі сформульованих вимог визначаються межі системи, її основні модулі, зв'язки між сутностями та набір дій, які мають бути підтримані на рівні даних, серверної логіки й користувацького інтерфейсу. Без такого попереднього опрацювання подальша реалізація могла б виявитися непослідовною або недостатньо узгодженою з потребами користувачів.

РОЗДІЛ 2 ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОЇ СИСТЕМИ

Другий розділ присвячено технічному втіленню вимог, сформульованих у попередньому розділі. У ньому послідовно розглядаються процес розробки, архітектурна побудова системи, модель даних, UML-представлення доменних сутностей, обґрунтування технологічного стеку та реалізація ключових механізмів автентифікації, контрактної взаємодії, контролю доступу й користувацького інтерфейсу. Таким чином, розділ демонструє перехід від постановки задачі до конкретної програмної реалізації.

2.1 Вибір процесу розробки

Для розробки програмної системи було обрано процес, що поєднує специфікаційно-орієнтований підхід із використанням інструментів штучного інтелекту на допоміжних етапах формалізації та реалізації. Його суть полягає в тому, що розробка починається не з написання коду, а з формулювання специфікації, далі переходить до планування, декомпозиції задач і лише після цього до безпосередньої реалізації. Така послідовність дає змогу забезпечити простежуваність між вимогами, архітектурними рішеннями та реалізованими компонентами системи.

На відміну від спрощеного підходу, за якого індивідуальний проєкт часто розвивається через швидкі локальні правки без формального опису, специфікаційно-орієнтована схема знижує ризик втрати цілісності рішення. Для системи, що підтримує не лише редагування сценарію, а й спільну роботу, контроль доступу та елементи структури сюжету, така керованість є особливо важливою. Кожне нове функціональне розширення може бути співвіднесене з окремою вимогою, а отже легше обґрунтовується в межах кваліфікаційної роботи.

Практично процес реалізовано як послідовність етапів: specify, plan, tasks та implement, – де інструменти штучного інтелекту використовуються не для підміни інженерного рішення, а для прискорення аналізу, структуризації та підготовки

варіантів реалізації. Така організація узгоджується з підходом, у якому специфікація, план і декомпозовані завдання виступають основними інструментами керування контекстом під час розробки з підтримкою штучного інтелекту [12-13].

Порівняно з класичним Scrum або Kanban, запропонований процес краще відповідає масштабу індивідуального навчального проєкту. Він не вимагає повноцінної командної церемоніальності, але зберігає контрольованість, дисципліну опису вимог і можливість аргументовано переходити від аналізу до технічної реалізації.

Отже, вибір специфікаційно-орієнтованого процесу з підтримкою штучного інтелекту є обґрунтованим як з методологічного, так і з практичного погляду. Він забезпечує системність прийняття рішень і добре узгоджується з характером кваліфікаційної роботи, у якій важливо не лише реалізувати програмний продукт, а й послідовно пояснити логіку його побудови [6, 14-15].

Для збереження простежуваності між вимогами першого розділу та технічними рішеннями другого розділу доцільно узагальнити ключові функціональні вимоги, які безпосередньо визначають архітектуру, модель даних і прикладну реалізацію системи. Їх подано в таблиці 2.1.

Таблиця 2.1 – Основні функціональні вимоги проєктування системи

Код	**Функціональна вимога**	**Опис**
1	**2**	**3**
FR-01	Автентифікація користувача	Система забезпечує реєстрацію, вхід, відновлення сесії та ініціалізацію початкового робочого середовища користувача.
FR-02	Керування робочими просторами	Авторизований користувач може створювати робочі простори, відкривати їх і перемикатися між особистим та командними контекстами роботи.

Продовження таблиці 2.1

1	2	3
FR-03	Запрошення учасників і ролі доступу	Власник або адміністратор може створювати запрошення, призначати ролі, керувати складом команди та виконувати перевірку прав доступу.
FR-04	Керування серіями	Користувач із належними правами може створювати серії, змінювати їхні метадані та використовувати як контейнери для сценарних матеріалів.
FR-05	Робота зі сценарієм і його версіями	Система підтримує створення, відкриття, редагування, автоматичне і ручне збереження, повторне завантаження та фіксацію змін сценарного тексту.
FR-06	Ведення бази знань	Користувач може створювати й редагувати персонажів, локації, реквізит та інші довідкові сутності з повторним використанням у різних сценаріях.
FR-07	Робота зі структурою сюжету	Система підтримує сцени, теми, сюжетні арки та зв'язки між ними для формалізованого представлення історії.
FR-08	Інтеграція редактора з предметними сутностями	Редактор сценарію забезпечує перехід до пов'язаних сутностей бази знань і сюжетної структури без втрати робочого контексту.
FR-09	Експорт результатів	Користувач може експортувати сценарні матеріали у форматі PDF для подальшого використання поза системою.

Отже, таблиця 2.1 фіксує той набір функціональних орієнтирів, який безпосередньо впливає на подальші архітектурні та реалізаційні рішення. Саме тому наступний розгляд архітектури системи доцільно виконувати як розвиток цих вимог, а не як ізольований технічний опис.

2.2 Проєктування архітектури системи

Організація програмного рішення реалізована у форматі монорепозиторію, що містить три основні складові: серверну частину, клієнтську частину та спільний пакет контрактів і типів. Використання такого підходу дає змогу централізувати спільні доменні визначення, уникнути дублювання описів API та забезпечити узгоджену взаємодію між окремими частинами системи.

Серверна частина відповідає за бізнес-логіку, доступ до бази даних, автентифікацію, авторизацію та реалізацію процедур, які використовуються клієнтом. Клієнтська частина відповідає за інтерфейс користувача, маршрутизацію, роботу з серверними даними і підтримку основних сценаріїв взаємодії. Спільний пакет виконує роль єдиного джерела істини для контрактів, схем, констант і допоміжних типів. Завдяки цьому зменшується ймовірність розходження між серверним описом API та очікуваннями клієнта, що є критично важливим для системи зі складною предметною моделлю.

Клієнт-серверна взаємодія в системі організована через oRPC-контракти. Це означає, що опис процедур, їхніх вхідних даних, вихідних структур і метаданих захисту формується в спільному пакеті, а потім використовується як сервером, так і клієнтом. Така схема особливо корисна в контексті кваліфікаційної роботи, оскільки дозволяє показати простежуваний зв'язок між вимогами, контрактами, реалізацією та тестами [16].

Вибір монорепозиторію також має практичне значення на рівні інструментів. Кореневий package.json містить спільні сценарії запуску, перевірки типів, лінтингу, тестування та локального розгортання. Під час запуску в режимі розробки спочатку збирається спільний пакет, а потім паралельно стартують сервер і клієнт. Це підтверджує, що монорепозиторій використовується не лише як спосіб групування файлів, а як робоча архітектурна рамка, яка керує життєвим циклом усієї системи.

Для поточного масштабу системи мікросервісна архітектура була б надмірною. Вона збільшила б кількість точок інтеграції, вимагала б окремого

керування міжсервісною взаємодією, авторизацією між компонентами, розподіленим логуванням і розгортанням [17]. Для даного проєкту, у якому головною метою є цілісна реалізація однієї предметної платформи, монорепозиторій із чітким поділом на сервер, клієнт і спільний шар забезпечує кращий баланс між модульністю, зрозумілістю та керованістю [18].

Архітектурну композицію системи зручно узагальнити такими принципами:

– Спільний пакет контрактів є єдиним джерелом істини для клієнта і сервера.

– Серверна частина інкапсулює бізнес-логіку, доступ до даних і механізми безпеки.

– Клієнтська частина відповідає за маршрутизацію, стан інтерфейсу та взаємодію з процедурами API.

– Монорепозиторій спрощує узгоджене збирання, локальний запуск і статичну перевірку всіх пакетів.

2.3 Побудова схем бази даних

Модель даних розроблюваної системи має одночасно підтримувати спільну роботу користувачів і представлення структури сюжету. Саме тому для зберігання даних доцільно використано MongoDB як документ-орієнтовану базу даних. Такий вибір виправданий тим, що частина сутностей має змінну або вкладену структуру, а окремі доменні об'єкти, наприклад сцени, сюжетні арки або теми, містять масиви пов'язаних елементів, які природно подаються у вигляді документів із вкладеними підструктурами [19].

На рівні прикладної моделі документну структуру інкапсульовано через технологію об'єктно-реляційного відображення Typegoose, що спрощує опис схем, індексів і зв'язків між сутностями в типізованому вигляді.

Базовим організаційним рівнем системи є робочий простір. Сутність робочого простору зберігає загальні відомості про середовище спільної роботи, а сутність учасника робочого простору фіксує належність користувача до певного

простору та його роль. Для уникнення дублювання членства використано унікальний складений індекс за полями workspaceId і userId, що гарантує існування не більше одного запису про участь конкретного користувача в конкретному робочому просторі. Така структура є важливою не лише з погляду цілісності даних, а й для реалізації подальшої моделі контролю доступу.

Наступний рівень доменної моделі утворюють сценарні сутності. Серія виступає контейнером для пов'язаних сценарних матеріалів і належить конкретному робочому простору. Сценарій та його версії забезпечують підтримку поступового редагування, а також фіксацію змін у часі. Такий поділ дозволяє розмежувати поточний робочий стан тексту і його історію, що є важливим для творчого процесу, у якому сценарний матеріал постійно еволюціонує.

Особливе значення для теми роботи мають сутності, що безпосередньо відображають структуру сюжету. Модель сцени містить не лише номер і заголовок сцени, а й атрибути часу доби, емоційного тону, конфлікту, технічних параметрів та масив композиційних бітів, які описують внутрішню композицію сцени. Це дає змогу подати сцену не як звичайний фрагмент тексту, а як структурований елемент драматургії зі збереженням композиційної організації, притаманної професійному сценарію [20]. Важливо також, що для цієї сутності реалізовано версійність та унікальність пари scriptId і sceneNumber, що спрощує роботу з послідовністю сцен у межах сценарію.

Сюжетна арка у моделі даних описується як окремий версіонований об'єкт, який містить назву, опис, статус виконання, початковий і кінцевий сценарні контексти, а також набір ключових драматургічних бітів. Додатково арка пов'язується з персонажами та темами, що дозволяє фіксувати не лише послідовність подій, а й драматургічну роль учасників сюжету. Аналогічно, сутність теми акумулює опис мотивів, пов'язаних персонажів, еволюцію теми в різних сценаріях і її появи в окремих сценах. Таким чином, модель даних підтримує не лише зберігання тексту сценарію, а й аналітичне представлення сюжетної структури.

Окремий блок моделі становить база знань, яка охоплює персонажів, локації та інші предметні сутності. Її призначення полягає в тому, щоб забезпечити цілісність довідкової інформації та повторне використання даних у сценах, сюжетних арках і сценаріях. Саме завдяки поєднанню робочих просторів, сценарних сутностей, елементів структури сюжету та бази знань модель даних розроблюваної системи узгоджується з назвою і метою кваліфікаційної роботи.

На рисунку 2.1 подано діаграму зв'язків даних, яка в узагальненому вигляді ілюструє структуру бази даних і взаємозв'язки між її основними сутностями.

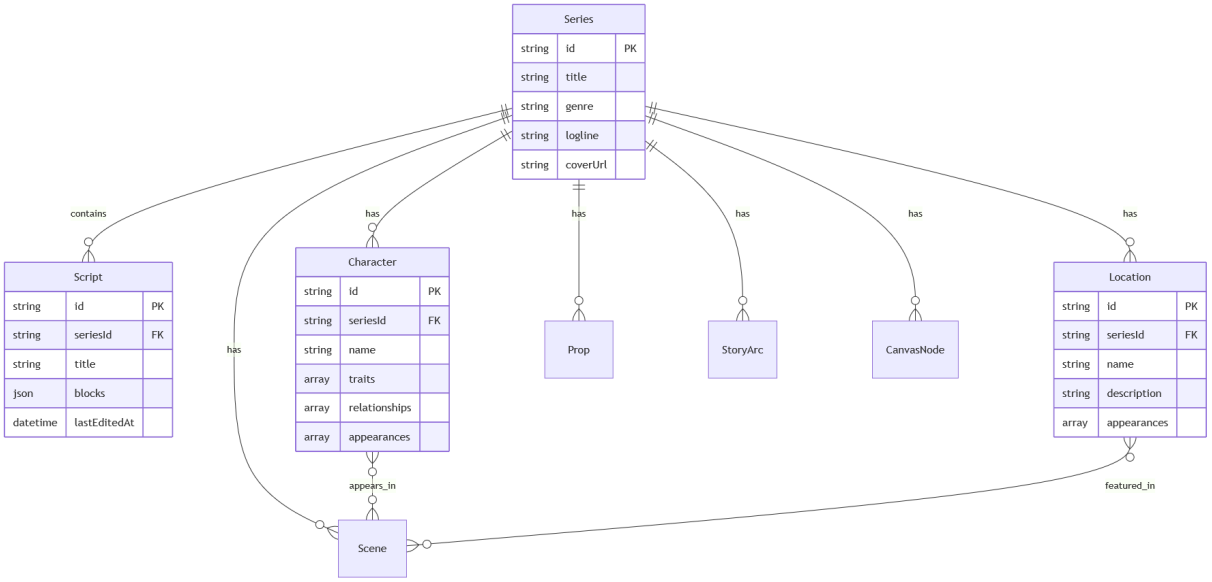

Рисунок 2.1 – Узагальнена схема даних програмної системи

Наведена схема показує, що організаційний контур спільної роботи, сценарні сутності та елементи сюжетної структури зберігаються в єдиній узгодженій моделі даних. Це важливо для подальшого переходу до UML-представлення, де ті самі об'єкти розглядаються вже з погляду логічних ролей і зв'язків.

2.4 Побудова UML-діаграм класів

UML-діаграма класів у межах цієї роботи виконує роль узагальненого представлення доменної структури системи. На відміну від опису схем бази

даних, де акцент зроблено на способі фізичного збереження інформації, UML-діаграма дозволяє зосередитися на логічних відношеннях між сутностями, їхніх ролях у предметній області та характері взаємодії між ними. Саме тому в цьому підрозділі доцільно групувати об'єкти не лише за технічною функцією, а й за їхнім смисловим призначенням.

Першу групу утворюють організаційні сутності, пов'язані зі спільною роботою. До неї належать робочий простір і його учасник. Робочий простір задає контекст колективної взаємодії, тоді як учасник робочого простору фіксує належність користувача до цього контексту та його роль у моделі доступу. Другу групу становлять сценарні сутності: серія, сценарій і версія сценарію. Вони відповідають за загальну структурну організацію сценарного матеріалу та його послідовне редагування. Третю групу утворюють сутності сюжетної структури: сцена, сюжетна арка, тема, а також інші пов'язані елементи, які відображають композиційний і драматургічний рівень опису історії.

Окремий блок UML-моделі становлять довідкові сутності бази знань: персонаж, локація, реквізит, дослідницький матеріал, універсальна допоміжна сутність та інші об'єкти. Їхнє призначення полягає в централізованому зберіганні контекстної інформації, що повторно використовується в різних частинах системи. Саме на UML-рівні зручно показати, що сцени, сюжетні арки й теми не ізольовані одна від одної, а взаємодіють із персонажами, локаціями та іншими елементами бази знань.

Узагальнено UML-діаграма має демонструвати, що розроблювана система поєднує три взаємозалежні площини: спільну роботу користувачів, сценарний текст як основний об'єкт редагування та структурне представлення сюжету. Саме ця комбінація відрізняє систему від простого редактора тексту й формує її предметну специфіку.

На рисунку 2.2 наведено UML-діаграму класів, яка узагальнює доменну структуру розроблюваної системи та зв'язки між ключовими сутностями.

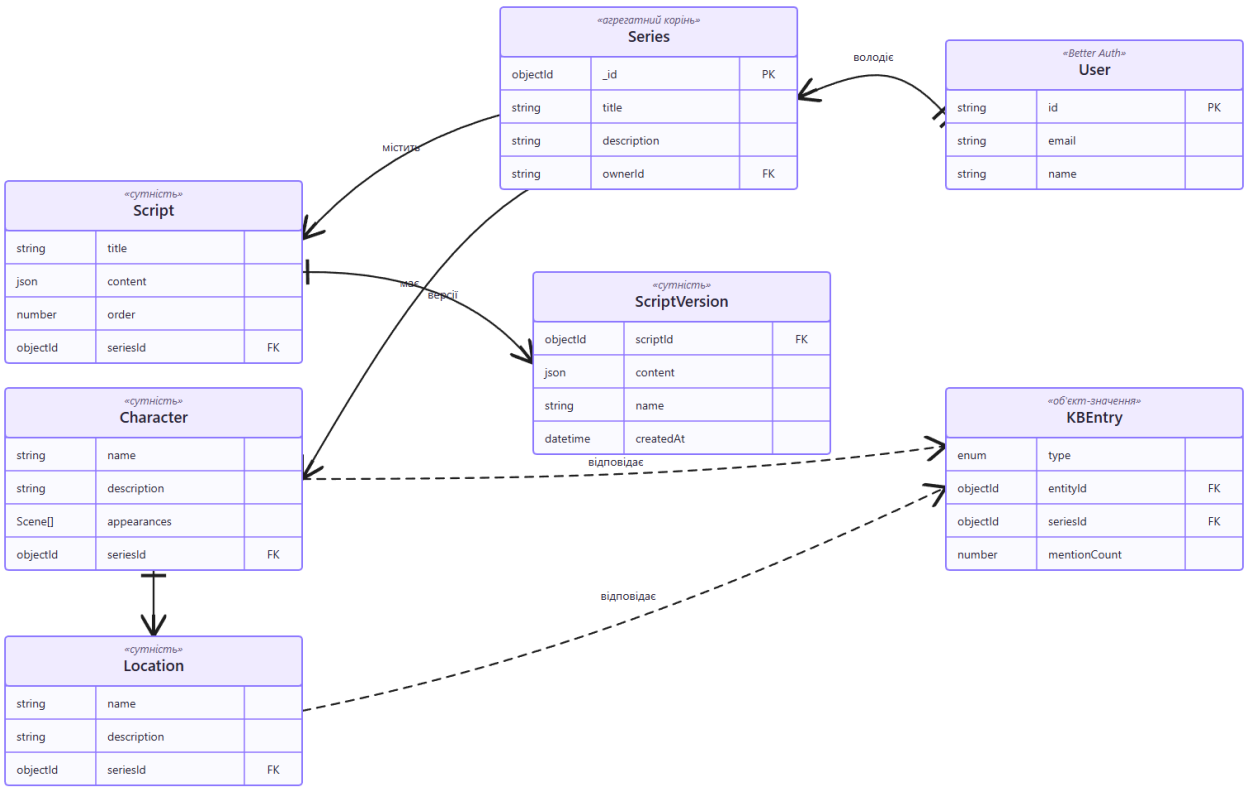

Рисунок 2.2 – UML-діаграма класів програмної системи

Таким чином, UML-діаграма узагальнює доменну композицію системи і показує, що командна взаємодія, сценарний текст та база знань утворюють єдиний об'єктний контур. Це безпосередньо впливає на вибір технологічного стеку, який повинен підтримувати таку зв'язану модель без дублювання типів і контрактів.

2.5 Вибір мови та середовища розробки

Вибір мови програмування, інструментів і середовища виконання безпосередньо впливає на підтримуваність, типобезпечність і відтворюваність системи. Тому в цьому підрозділі технологічний стек розглядається не як перелік популярних інструментів, а як набір взаємопов'язаних рішень, що відповідають архітектурі монорепозиторію, контрактному підходу та вимогам до спільної сценарної роботи.

2.5.1 Мова програмування та середовище виконання

Для реалізації системи обрано TypeScript як основну мову програмування і Node.js як середовище виконання серверної частини. Такий вибір зумовлений передусім необхідністю зберегти єдиний типізований стек між клієнтом, сервером і спільним пакетом. Завдяки цьому можна використовувати спільні типи, контракти й доменні константи без дублювання та ручної синхронізації між частинами системи.

Перевага TypeScript у контексті цієї роботи полягає не лише в статичній типізації, а й у можливості формалізувати структуру складних доменних об'єктів, пов'язаних зі сценаріями, спільною роботою та сюжетними сутностями. Це знижує кількість неузгодженостей між модулями та підвищує надійність рефакторингу [21]. Додатково в усіх основних пакетах проєкту передбачено окремі скрипти для перевірки правильності типів, що підкреслює важливість типобезпечності як частини загального процесу верифікації.

Node.js доцільно використано для серверної частини через його природну сумісність із TypeScript-екосистемою, розвинену бібліотечну базу та зручність локального розгортання. У конфігурації серверного пакета окремо зафіксовано мінімальну підтримувану версію Node.js, що додатково дисциплінує середовище виконання і спрощує відтворюваність системи.

2.5.2 Серверний стек

Серверний стек побудовано на Hono, oRPC, MongoDB і Typegoose/Mongoose. Фреймворк Hono обрано як легку серверну основу, що дозволяє будувати HTTP-шар без зайвої надбудовної складності та добре інтегрується з процедурним підходом до опису API. Для цієї роботи це важливо, оскільки сервер має не лише обробляти запити, а й залишатися достатньо прозорим для пояснення в тексті кваліфікаційної роботи.

Ключовим архітектурним рішенням є використання oRPC. На відміну від підходу, у якому серверні маршрути й клієнтські запити описуються окремо, oRPC дозволяє централізовано формувати контракт процедури та використовувати його

по обидва боки взаємодії. Це зменшує кількість помилок на стику підсистем і посилює простежуваність між предметними вимогами та реальною реалізацією API [16].

Для збереження даних використано MongoDB. Документ-орієнтована модель добре узгоджується зі структурою предметної області, де значна частина сутностей має вкладені елементи, списки зв'язків та змінний набір атрибутів. Поверх MongoDB використано Typegoose, що дозволяє описувати моделі даних у звичному типізованому стилі та поєднувати документну схему з можливостями TypeScript [19]. Для розроблюваної системи, де модель охоплює робочі простори, сценарії, сцени, арки та інші сюжетні сутності, таке поєднання є практично виправданим.

2.5.3 Клієнтський стек

Клієнтська частина системи реалізована на React 19 із використанням Vite, TanStack Router, TanStack Query, TanStack Form і Tailwind CSS. React доцільно обрано як зрілу компонентну платформу, яка добре підходить для побудови інтерфейсу з багатьма взаємопов'язаними екранами та станами. Для системи спільної сценарної роботи це важливо, оскільки інтерфейс повинен одночасно підтримувати редакторські сценарії, навігацію між сутностями та роботу з довідковими даними.

Vite використовується як засіб локальної розробки та збирання фронтенду. Його перевагою є швидкий запуск, просте налаштування та хороша інтеграція з сучасною React-екосистемою. У контексті монорепозиторію це дозволяє підтримувати фронтенд-частину в окремому пакеті без надмірного ускладнення інструментального контуру.

TanStack Router, TanStack Query і TanStack Form формують основу прикладної логіки інтерфейсу. Router відповідає за структуровану маршрутизацію, Query забезпечує керовану роботу з серверними даними і кешем, а Form полегшує реалізацію типізованих форм. Разом ці інструменти створюють кероване середовище для побудови складного веб інтерфейсу, у якому користувач

переходить між робочими просторами, сценаріями, сюжетними сутностями й елементами бази знань. Tailwind CSS, у свою чергу, використано для швидкої та послідовної реалізації стилів на рівні компонентів.

2.5.4 Автентифікація та авторизація

Для підсистеми автентифікації та авторизації в розроблюваній системі було обрано Better Auth. Такий вибір зумовлений тим, що система має відносно локальний контур користувачів, не потребує інтеграції з зовнішніми корпоративними каталогами і водночас повинна забезпечувати надійну сесійну автентифікацію, підтримку взаємодії через cookie-файли, а також зручну інтеграцію між серверною та клієнтською частинами. Практична цінність Better Auth для цього проєкту полягає в тому, що бібліотека добре узгоджується з поточною архітектурою на базі TypeScript, Hono, MongoDB та React і не вимагає побудови окремого контуру керування ідентичностями.

Альтернативою розглядався Keycloak як повноцінна платформа керування ідентичностями та ролями доступу. Перевагою такого підходу є наявність розвиненого механізму централізованого керування користувачами, ролями, клієнтами, політиками доступу, а також підтримка сценаріїв єдиного входу, федерації ідентичностей та адміністрування через окрему консоль. Водночас для поточного масштабу розроблюваної системи такий підхід є надмірним, оскільки потребує виділеного сервера ідентифікації, додаткової конфігурації об'єктів realm і client, окремого життєвого циклу токенів і помітно збільшує операційну складність системи. Для даного проєкту, орієнтованого на одну предметну платформу з обмеженою кількістю ролей і без сценаріїв міжсистемної федерації, таке рішення не дає пропорційного виграшу відносно витрат на впровадження та супровід. Додатково інтеграція такого підходу вимагає окремо проєктувати ресурси, політики, дозволи та межі делегування авторизаційної логіки між платформою і прикладним сервісом [22].

Третьою можливою альтернативою є власна реалізація на основі JWT або самостійно написаного session-механізму. Хоча такий підхід забезпечує

максимальний контроль над поведінкою підсистеми автентифікації, він переносить на розробника відповідальність за критичні аспекти безпеки: життєвий цикл сесій, безпечне зберігання ідентифікаційних даних, обробку відновлення доступу, захист cookie або токенів, а також коректну інтеграцію з клієнтом. Для розроблюваної системи цей підхід був визнаний менш доцільним, ніж використання готового перевіреного рішення.

Таким чином, Better Auth обрано як компроміс між функціональною достатністю та складністю впровадження. Авторизація в системі реалізується не як універсальна загальнокорпоративна модель, а як предметно-орієнтований механізм контролю доступу до робочих просторів і пов'язаних із ними ресурсів. Це означає, що в системі поєднано дві площини захисту: базову автентифікацію користувача через сесію та доменний контроль доступу через ролі і перевірку належності ресурсу.

2.5.5 Інструменти розробки та інфраструктура

Інструментальний контур системи побудовано так, щоб забезпечити не лише написання коду, а й його відтворюваність, перевірюваність і підтримуваність. Для керування монорепозиторієм використано pnpm workspaces, що дозволяє централізовано запускати збірку, розробку, лінтинг і перевірку типів для всіх пакетів. Такий підхід добре поєднується з поділом на сервер, клієнт і спільний шар та спрощує синхронізований розвиток усіх частин системи.

Docker використовується для підняття необхідної інфраструктури, насамперед бази даних, у локальному середовищі розробки та під час E2E-перевірок [23]. Це зменшує залежність від ручного налаштування середовища і робить сценарії запуску більш передбачуваними. Для контролю якості коду застосовано ESLint, Prettier і перевірку типів TypeScript, а для автоматизованого тестування використано Vitest і Playwright [24]. Додатково Husky забезпечує запуск перевірок на етапі локального робочого процесу перед комітом.

У сукупності ці інструменти формують не допоміжний, а повноцінний інженерний контур системи. Саме він дозволяє розглядати розроблюване

програмне забезпечення як керований технічний продукт, а не як набір ізольованих фрагментів реалізації.

2.6 Реалізація основних класів та методів

Після обґрунтування архітектури і стеку доцільно перейти до розгляду фактичної реалізації основних механізмів системи. У цьому підрозділі увага зосереджена на тих елементах коду, які безпосередньо забезпечують цілісність клієнт-серверної взаємодії, запуск автентифікації, реалізацію моделей даних і рольовий контроль доступу.

2.6.1 Контракти та захист процедур

Одним із ключових реалізаційних рішень у системі є винесення опису процедур у спільний контрактний шар. Це означає, що маршрут, метод, структура вхідних і вихідних даних, а також окремі метадані описуються не лише на стороні сервера, а в загальнодоступному для всіх частин системи пакеті. Такий підхід полегшує синхронізацію між клієнтом і сервером та зменшує ймовірність помилок, пов'язаних із розходженням очікуваних структур даних.

Практично це реалізовано через oRPC-контракти. У спільному пакеті використовується спеціальна позначка authProcedure, яка додає до процедури метадані про необхідність автентифікації. Хоча сам цей опис є лаконічним, його роль принципова: він формує загальний механізм маркування захищених процедур, який надалі враховується під час серверної реалізації та клієнтської інтеграції. На прикладі контракту профілю користувача видно, що процедура містить одночасно маршрут, HTTP-метод, короткий опис, схему вхідних даних і типізовану структуру виходу. Отже, контрактний шар у системі виконує не декоративну, а конструктивну функцію, оскільки поєднує опис API, типізацію та вимоги до доступу.

2.6.2 Серверна реалізація автентифікації

Серверна реалізація автентифікації в розроблюваній системі побудована навколо окремого сервісу, який інкапсулює конфігурацію Better Auth і підключається до застосунку на етапі ініціалізації. Під час створення екземпляра сервісу використовується адаптер MongoDB, що дає змогу працювати з тією самою базою даних, у якій зберігаються доменні сутності системи. Окрім базових параметрів, конфігурація містить налаштування доменів, з яких надходить запит, підтримку використання емейлу замість логіна, а також параметри cookie, зокрема обмеження доступу до cookie з боку клієнтського JavaScript і передавання cookie тільки через протокол HTTPS, що важливо для коректної та безпечної роботи браузерної сесії.

Практично важливим елементом реалізації є використання обробників подій життєвого циклу користувача. Після успішного створення нового облікового запису система автоматично формує профіль користувача, створює персональний робочий простір та додає користувача до нього з роллю власника. Додатково передбачено механізм автоматичного прийняття активних запрошень, що були прив'язані до електронної адреси нового користувача. Така логіка демонструє, що підсистема автентифікації не ізольована від доменної моделі, а безпосередньо ініціалізує початковий робочий контекст користувача.

2.6.3 Реалізація моделі даних

Реалізація моделі даних у системі побудована на Typegoose-класах, які дозволяють поєднати документні схеми MongoDB із типізованим описом сутностей на рівні TypeScript. Такий підхід є доцільним для цієї роботи, оскільки забезпечує зрозумілу структуру моделей і робить їх придатними як для виконання, так і для пояснення в тексті кваліфікаційної роботи.

Як приклад базової предметної сутності доцільно розглядати модель серії. Вона містить ідентифікатор, назву, тип, жанр, логлайн, обкладинку, час останнього редагування та посилання на робочий простір. У цьому випадку особливо важливим є поле workspaceId, яке прямо показує, що навіть базова сценарна

сутність включена до контексту спільної роботи, а не існує ізольовано. Така модель є компактною і зручною для демонстрації принципів побудови доменних класів у системі.

Окремою перевагою такого способу опису моделей є те, що він дозволяє одночасно зберігати зв’язок із базою даних і підтримувати зрозумілу типізовану структуру коду. У практичній реалізації це спрощує подальше розширення предметної області, оскільки нові поля або пов’язані сутності можуть додаватися без зміни загального принципу побудови моделей.

На рисунку 2.3 подано реалізаційне представлення базового ланцюга моделей даних, який охоплює робочий простір, членство в ньому, серії, сценарії та версії сценаріїв. Саме цей фрагмент показує, як на рівні коду поєднуються організаційний контекст спільної роботи та власне сценарний матеріал.

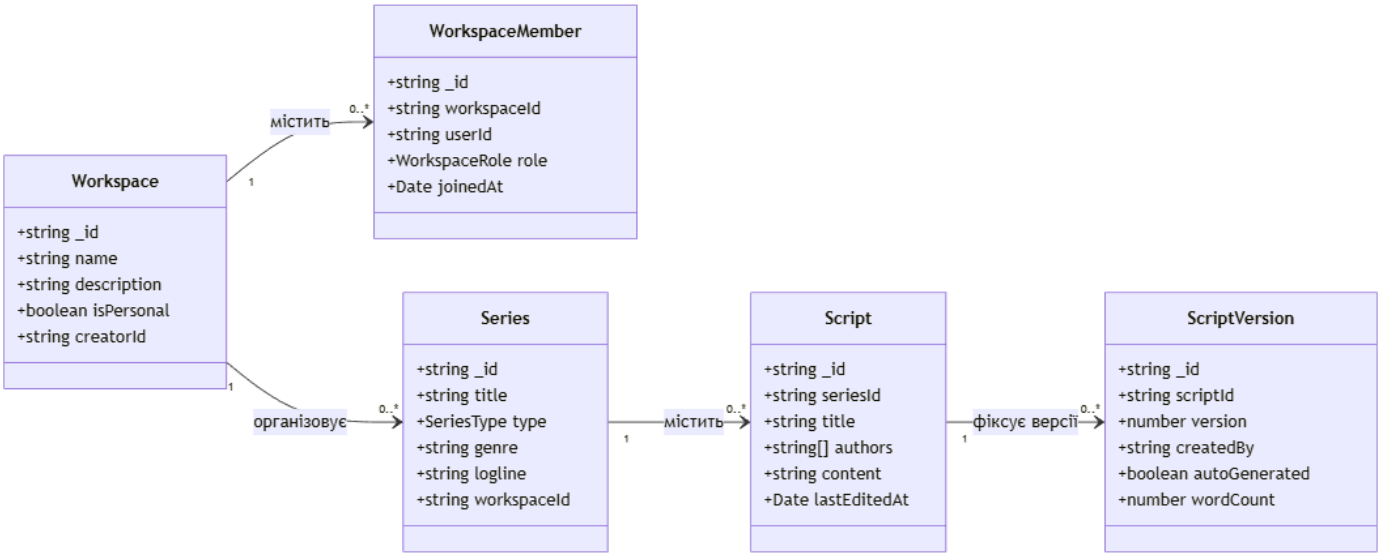

Рисунок 2.3 – Реалізація базового ланцюга моделей даних програмної системи

Для відображення специфіки теми роботи доцільно також окремо подати сцено-орієнтовані та сюжетно-аналітичні моделі. На відміну від серії, ці сутності містять складніші зв'язки з персонажами, темами, сценами або сценаріями, а також службові поля версійності. Саме вони демонструють, що програмна система працює не лише з текстом як таким, а з формалізованими елементами структури сюжету.

Базовий ланцюг моделей показує загальну логіку зберігання сценарних матеріалів, однак не розкриває повністю специфіку роботи саме зі структурою сцени. Для сценарного редактора важливо, щоб сцена не зберігалася як звичайний фрагмент тексту, а мала власні зв’язки з іншими сутностями предметної області. Це дає змогу надалі будувати інтерфейсні та аналітичні механізми не навколо суцільного документа, а навколо окремих змістових одиниць.Такий поділ є корисним і з погляду розширення системи. Наприклад, персонажі, локації або реквізит можуть повторно використовуватися в різних сценах, оновлюватися окремо та застосовуватися для фільтрації, пошуку або подальшого аналізу сценарного матеріалу. Саме тому сцено-орієнтовані моделі потребують окремого представлення.

На рисунку 2.4 подано реалізаційне представлення моделі сцени та пов'язаних довідкових сутностей. Воно показує, що сцена пов'язується зі сценарієм, а також використовує персонажів, локації та реквізит як окремі об'єкти предметної області, а не як фрагменти неструктурованого тексту.

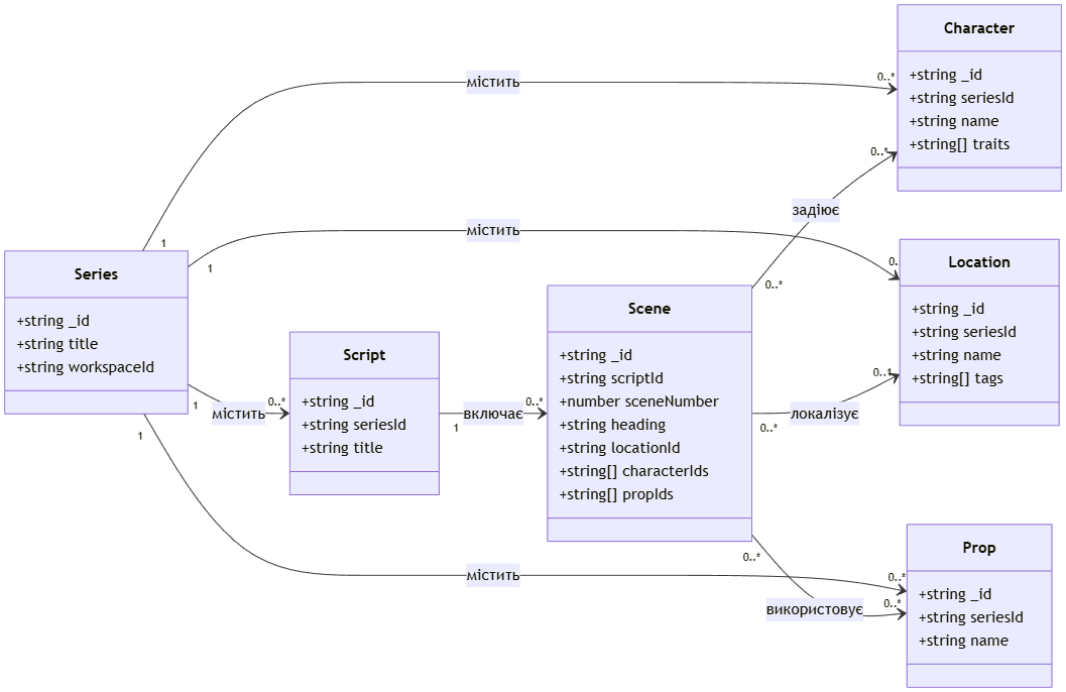

Рисунок 2.4 – Реалізація моделі сцени та пов'язаних довідкових сутностей

На рисунку 2.5 подано реалізаційне представлення моделей тем і сюжетних арок у межах програмної системи. Воно відображає, що серія слугує контейнером для тематичних і драматургічних конструкцій, а сюжетні арки можуть одночасно пов’язувати персонажів, теми та конкретні появи в сценах, формуючи зв’язки між різними рівнями сценарної структури.

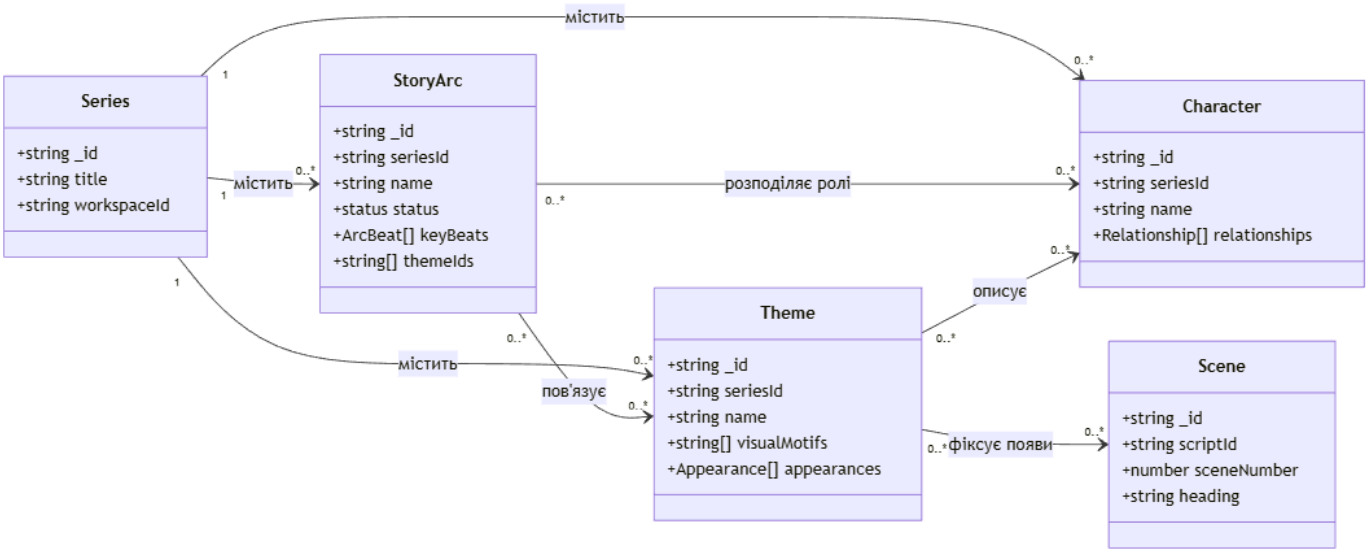

Рисунок 2.5 – Реалізація моделей тем і сюжетних арок програмної системи

Таким чином, реалізація моделі даних у розроблюваній системі поєднує три взаємопов'язані рівні: організаційний контур спільної роботи, сцено-орієнтований контур сценарного матеріалу та сюжетно-аналітичний контур тем і арок. Саме така структура дозволяє зберігати цілісність сценарного матеріалу, підтримувати його версійність і водночас працювати з елементами структури сюжету як із формалізованими об'єктами.

2.6.4 Реалізація контролю доступу

Контроль доступу в розроблюваній системі реалізовано як окремий доменний механізм поверх базової автентифікації. Сам факт наявності сесії лише підтверджує особу користувача, але не надає права на виконання будь-якої операції. Для цього в системі використовується рольова модель робочих просторів, у межах якої користувач може мати різні рівні доступу до конкретного

робочого простору [8]. Такий підхід є принципово важливим, оскільки одна й та сама особа може працювати з кількома просторами, а права доступу залежать від контексту взаємодії.

Сервіс перевірки дозволів реалізує ієрархію ролей і надає операції для перевірки членства, мінімального рівня доступу та можливості керування іншими користувачами. Для підвищення продуктивності інформація про членство кешується, а для недоступних ресурсів використовується помилка NOT_FOUND, який не розкриває факт існування робочого простору сторонньому користувачеві. Окремо реалізовано правила керування ролями: власник може змінювати ролі учасників нижчого рівня, адміністратор має обмежені повноваження, а користувач із нижчими правами не може змінювати учасників рівного або вищого статусу.

Коректність цієї моделі підтверджується інтеграційними тестами, у яких перевіряються як дозволені, так і заборонені сценарії. Зокрема, тестовий набір демонструє, що власник може змінювати роль адміністратора, адміністратор не може змінювати роль власника, а редактор не має права змінювати ролі інших учасників. Отже, в системі реалізовано не абстрактну декларацію RBAC, а конкретний предметно-орієнтований механізм контролю доступу, який підлягає автоматизованій перевірці.

2.7 Розробка інтерфейсу користувача

Інтерфейс користувача побудовано як набір взаємопов'язаних маршрутів і компонентів, які відображають основні сценарії роботи з системою. Структурну основу вебзастосунку формує TanStack Router із файловою організацією маршрутів. Такий підхід дозволяє прозоро пов'язати конкретний екран із відповідним файлом маршруту та зменшує складність навігації в кодовій базі. В інтерфейсі чітко простежується поділ між загальнодоступними сторінками та сторінками, доступними лише авторизованому користувачеві.

Одним із базових інтерфейсних вузлів є сторінка робочих просторів. Вона дозволяє користувачеві переглядати особистий і командні робочі простори,

створювати новий робочий простір і переходити до подальшої роботи над матеріалами. Саме цей екран добре демонструє, що система від початку орієнтована на спільну взаємодію, а не лише на індивідуальне редагування сценарного тексту.

Центральним робочим екраном є редактор сценарію. Відповідний маршрут забезпечує попереднє завантаження даних сценарію, підтримку автоматичного збереження, ручного збереження, відкриття параметрів сценарію та експорт у PDF. Додатково редактор інтегровано з механізмом відображення присутності інших учасників, що підтверджує орієнтацію системи на сценарії спільної роботи. Важливо й те, що з редактора можна переходити до пов'язаних об'єктів бази знань, зокрема персонажів або інших згаданих сутностей. Отже, інтерфейс редактора поєднує безпосереднє редагування тексту з навігацією по пов'язаному предметному контексту.

Окремий важливий блок інтерфейсу становить база знань. Цей екран організовано як багатовкладковий простір, у межах якого користувач може працювати з персонажами, локаціями, реквізитом, сценами, хронологією, сюжетними арками, темами та іншими сутностями. Така побудова інтерфейсу безпосередньо підтримує заявлений акцент на структурі сюжету, оскільки дозволяє представляти історію через сукупність взаємопов'язаних об'єктів, а не лише як послідовність текстових блоків.

Додатково в системі передбачено спеціалізовані екрани, зокрема представлення серії у вигляді канви, налаштування робочого простору та маршрути, пов'язані із запрошеннями учасників. На рівні реалізації це підкріплюється компонентним підходом: окремі компоненти інтерфейсу, діалоги, картки, панелі редагування та навігаційні елементи формують повторно використовуваний інтерфейсний шар. Отже, розробка інтерфейсу користувача в системі спрямована не на косметичне оформлення, а на підтримку ключових сценаріїв спільної сценарної роботи, керування знаннями та структурою сюжету.

Для інтерфейсного шару особливо важливими є такі властивості:

1. Швидкий перехід між робочими просторами, серіями і конкретними сценарними об'єктами.

2. Поєднання редактора тексту з доступом до пов'язаних сутностей бази знань.

3. Відображення командного контексту без перевантаження основного сценарного процесу.

4. Наявність спеціалізованих екранів для бази знань, канви та налаштувань робочого простору.

Ці властивості визначають практичну цінність інтерфейсу для сценарної роботи: користувач може редагувати текст і переглядати сюжетні сутності.

На рисунках 2.6 і 2.7 наведено приклади ключових користувацьких екранів, які демонструють поєднання сценарного редактора та багатовкладкового інтерфейсу бази знань.

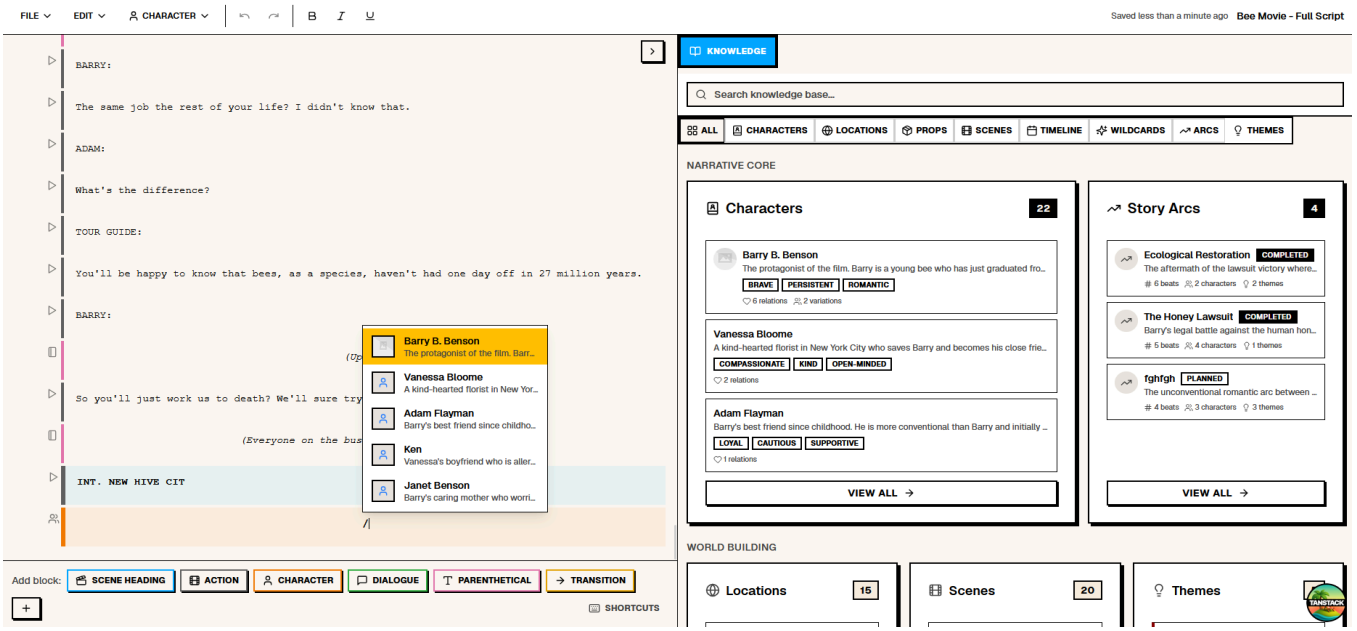

Рисунок 2.6 – Екран редактора сценарію

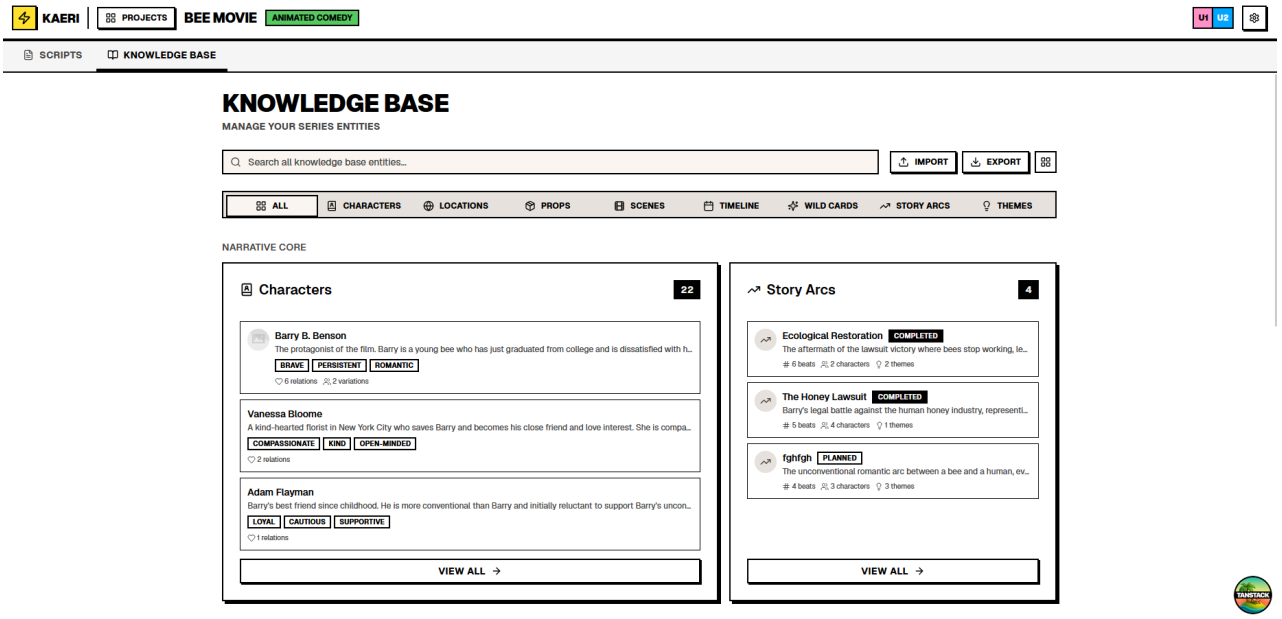

Рисунок 2.7 – Екран бази знань із сюжетними сутностями

Наведені екрани дають змогу наочно побачити, як основні сценарії роботи реалізуються в інтерфейсі системи. Екран редактора сценарію зосереджує користувача на безпосередньому опрацюванні тексту, але водночас не ізолює його від пов'язаного контексту, оскільки потрібні дії, параметри та переходи до інших сутностей залишаються доступними в межах того самого робочого середовища. Екран бази знань, своєю чергою, показує, що персонажі, сцени, теми та сюжетні арки подаються не як розрізнені довідкові записи, а як структурований набір взаємопов'язаних об'єктів. Таким чином, візуальний шар системи підтримує цілісний сценарний процес і поєднує письмо, навігацію та керування сюжетною інформацією в одному інтерфейсі.

Водночас ці приклади показують і загальний принцип побудови користувацького середовища: кожен екран виконує окрему практичну роль, але не існує ізольовано від інших частин системи. Робота з текстом, перегляд довідкових сутностей, перехід між сценарними об'єктами та командна взаємодія об'єднані спільною навігаційною логікою. Завдяки цьому інтерфейс сприймається не як набір окремих сторінок, а як послідовно організований робочий простір для спільної сценарної діяльності.

Отже, у другому розділі розглянуто процес розробки, архітектуру, модель даних, технологічний стек і ключові рішення, покладені в основу розроблюваної системи. Наведені матеріали показують, яким чином вимоги попереднього розділу були перетворені на конкретну програмну структуру, що підтримує спільну сценарну роботу, контроль доступу та формалізоване подання елементів сюжету.

Розглянуті рішення утворюють єдину технічну основу системи. Монорепозиторій, спільний контрактний шар, модель даних і побудова інтерфейсу узгоджені між собою та підпорядковані спільній меті: забезпечити цілісність предметної моделі, зручність розвитку системи й узгодженість між клієнтською та серверною частинами.

Таким чином, другий розділ завершує перехід від постановки задачі до її практичної реалізації. На цій основі далі можна перейти до перевірки працездатності системи, опису її розгортання та верифікації отриманих результатів. Це, своєю чергою, дозволяє оцінити не лише коректність окремих реалізаційних рішень, а й цілісність системи як програмного продукту. Водночас саме такий підхід дає змогу простежити зв'язок між початковими вимогами, архітектурними рішеннями та фактично реалізованим функціоналом системи.

РОЗДІЛ 3 ТЕСТУВАННЯ, ВПРОВАДЖЕННЯ ТА ПІДТРИМКА

Третій розділ присвячений підтвердженню працездатності та відтворюваності розроблюваної системи. У ньому розглянуто багаторівневе тестування, локальне розгортання, системні вимоги до середовища розробки та засоби верифікації якості, які використовуються в робочому процесі. Саме цей розділ має показати, що результат роботи є не лише концептуально обґрунтованим, а й технічно перевіреним.

3.1 Тестування програмної системи

Тестування програмної системи в межах цієї роботи організоване за кількома рівнями і не зводиться до ручної перевірки інтерфейсу. Такий підхід є необхідним, оскільки система охоплює як серверну бізнес-логіку, так і клієнтські сценарії взаємодії, а також механізми автентифікації та контролю доступу. Відповідно, перевірка якості має поєднувати тести серверних сервісів, інтеграційні перевірки контрактів і наскрізні тести користувацьких сценаріїв [24-25, 6].

3.1.1 Види та план тестування

План тестування доцільно описувати як тестову піраміду з трьох рівнів. На нижньому рівні розташовані модульні та сервісні тести, які перевіряють окремі частини серверної логіки: автентифікацію, роботу із серіями, сюжетними сутностями та правилами співпраці. На цьому рівні зручно перевіряти локальні інваріанти, валідацію даних і реакцію системи на помилки.

Наступний рівень становлять інтеграційні тести серверної логіки. Вони перевіряють взаємодію модулів у прикладних сценаріях: створення робочого простору, роботу із запрошеннями, керування ролями, створення серій і збереження сценарних даних. Прикладами є тести підсистеми співпраці, зокрема створення й прийняття запрошень та перевірка рольових обмежень.

Верхній рівень піраміди становлять наскрізні тести (E2E), які відтворюють користувацькі сценарії через інтерфейс вебзастосунку. Вони потрібні для перевірки того, що поведінка системи як цілісного продукту відповідає очікуванням користувача: захищені маршрути перенаправляють до форми входу, автентифікований користувач потрапляє на дашборд, а ключові сторінки завантажуються в коректному контексті.

Отже, тестовий план охоплює серверну коректність, користувацьку поведінку та перевірку контролю доступу. Рольова модель підтверджується конкретними тестами обмеження доступу, запрошення учасників і захищених операцій, а не залишається лише архітектурним наміром.

Структуру тестування в системі можна коротко подати таким списком:

– Сервісні та модульні тести для локальних інваріантів бізнес-логіки.

– Інтеграційні тести для сценаріїв співпраці, ролей і доступу до ресурсів.

– Наскрізні тести (E2E) для перевірки користувацьких сценаріїв через браузерний інтерфейс.

– Статичні перевірки типів і правил коду як доповнення до виконуваних тестів.

На рисунку 3.1 наведено приклад результатів тестового прогону, який ілюструє факт успішного виконання серверного тестового набору.

```
server:test:
server:test:  390 pass
server:test:  0 fail
server:test:  1117 expect() calls
server:test: Ran 390 tests across 20 files. [74.51s]

 Tasks:    3 successful, 3 total
Cached:    1 cached, 3 total
  Time:    1m14.988s
```

Рисунок 3.1 – Приклад результатів запуску автоматизованих тестів

Наведений результат підтверджує, що тестовий план у системі реалізовано як виконуваний автоматизований набір перевірок, а не лише як формальний опис намірів. Це дає змогу перейти до конкретизації тестових сценаріїв, які безпосередньо відображають функціональні вимоги та обмеження доступу.

Водночас для повноти оцінювання недостатньо лише констатувати наявність автоматизованих перевірок. Важливо також показати, які саме прикладні ситуації покриваються тестами та чому обрані сценарії є показовими для розроблюваної системи. Саме тому наступним кроком є виокремлення тестових сценаріїв, що відтворюють типові дії користувачів і перевіряють коректність обмежень доступу в реальних умовах використання.

3.1.2 Розробка тестових сценаріїв

Для практичної верифікації системи доцільно виділити кілька репрезентативних тестових сценаріїв, які охоплюють як базові функції, так і критичні обмеження доступу.

Таблиця 3.1 – Репрезентативні тестові сценарії перевірки функціональних вимог

Сценарій	**Передумови**	**Ключові кроки**	**Очікуваний результат**	**Підтвердження**
1	**2**	**3**	**3**	**4**
Успішна реєстрація або вхід користувача	Користувач не має активної сесії	Відкрити форму автентифікації, ввести коректні облікові дані, завершити вхід	Система створює або відновлює сесію та надає доступ до захищеного контуру	серверні тести підсистеми автентифікації, наскрізні сценарії входу та реєстрації

Продовження таблиці 3.1

1	2	3	4	5
Створення серії	Користувач автентифікований і має доступ до робочого простору	Відкрити відповідний робочий простір, задати метадані серії, підтвердити створення	Нова серія зберігається і з'являється в інтерфейсі	серверні тести серій, наскрізні сценарії дашборду
Редагування сценарію	Існує серія та сценарій із правами редагування	Відкрити редактор, змінити вміст, виконати збереження	Оновлений текст сценарію зберігається та доступний під час повторного відкриття	тести підсистеми сценаріїв і редактора, інтеграційні перевірки збереження
Операція з обмеженням доступу для ролі нижчого рівня	У робочому просторі є користувач із недостатніми правами	Спробувати виконати захищену операцію, наприклад створення запрошення або зміну ресурсу	Система відхиляє дію або приховує ресурс згідно з політикою доступу	тести підсистеми співпраці для ролей та запрошень
Вхід у систему та перехід до дашборду	Захищений маршрут викликається без активної сесії або з валідною сесією	Відкрити захищену сторінку, за потреби авторизуватись, перейти до дашборду	Неавторизований користувач перенаправляється на форму входу, авторизований потрапляє до дашборду	наскрізні тести сторінок входу та дашборду

Наведені сценарії є репрезентативними, оскільки охоплюють повний шлях від входу до системи до виконання захищених прикладних операцій. У фінальній версії розділу їх можна доповнити фактичними результатами прогону та ідентифікаторами конкретних тест-кейсів.

3.2 Розгортання та системні вимоги

Розділ розгортання має показати, що програмна система відтворюється у контрольованому середовищі без неформальних ручних налаштувань. У поточному проєкті локальний запуск організований через кореневий сценарій монорепозиторію pnpm dev, який спочатку піднімає контейнеризовані сервіси з файла docker-compose.dev.yml, потім збирає спільний пакет контрактів і запускає серверну та клієнтську частини в режимі розробки [23]. Така організація запуску є важливою з погляду відтворюваності, оскільки зменшує кількість ручних кроків і синхронізує запуск залежних компонентів.

Файл docker-compose.dev.yml використовується для локального середовища розробки та піднімає щонайменше два інфраструктурні сервіси: MongoDB і Valkey. Для бази даних використовується порт 6060, а для кешу та швидких службових операцій доступний порт 6379. Окремий файл docker-compose.e2e.yml призначений для ізольованого середовища наскрізного тестування; у ньому застосовуються інші порти, зокрема 6061 для MongoDB і 6380 для Valkey, що дозволяє запускати автоматизовані перевірки окремо від основного середовища розробки.

Конфігурація прикладного рівня задається через .env файли для серверної та клієнтської частин. Для сервера визначаються, зокрема, адреса CORS-джерела, секрет і URL автентифікації, рядок підключення до MongoDB, параметри Valkey, а також хост і порт запуску API. Для вебчастини задається адреса серверного API, а конфігурація Vite додатково підтримує параметри хоста і порту локального запуску. За поточними значеннями за замовчуванням сервер запускається на порту 5050, а клієнтська частина використовує порт 3030.

Мінімальні системні вимоги до середовища розробника випливають із фактичного стеку проєкту. Потрібна актуальна версія Node.js не нижче 20, менеджер пакетів pnpm, Docker із підтримкою Compose, а також вільні локальні порти для сервера, вебклієнта, MongoDB і Valkey. У поєднанні ці умови

забезпечують можливість повторюваного розгортання системи, локального тестування та подальшого запуску наскрізних перевірок.

Для компактного подання цих передумов їх було узагальнено в таблиці 3.2.

Таблиця 3.2 – Мінімальні вимоги до локального середовища розгортання

Компонент	**Мінімальна вимога**	**Призначення**
Node.js	Версія 20 або новіша	Запуск серверної і клієнтської частин, збірка пакетів і виконання сценаріїв автоматизації.
pnpm	Встановлений менеджер пакетів, сумісний із конфігурацією проєкту	Керування залежностями монорепозиторію та запуск кореневих сценаріїв.
Docker Compose	Доступний і запущений локально	Підняття MongoDB і Valkey для режимів розробки та ізольованого тестування.
Конфігураційні файли	Налаштовані .env файли для серверної і клієнтської частин на основі .env.example	Передавання параметрів API, бази даних, автентифікації та вебклієнта.
Вільні локальні порти	3030, 5050, 6060, 6379, а для ізольованих перевірок також 6061 і 6380	Запуск вебзастосунку, API та контейнеризованих інфраструктурних сервісів без конфліктів.

Подана таблиця фіксує мінімальний інфраструктурний контур, необхідний для відтворення системи без прихованих залежностей від конкретного робочого місця. Завдяки цьому розгортання можна розглядати як контрольований процес, що безпосередньо підтримує подальшу верифікацію програмної системи.

3.3 Верифікація програмної системи

Верифікація програмної системи в межах цієї роботи повинна доводити не лише факт наявності окремих тестів, а й відповідність реалізованих модулів

сформульованим вимогам. Для цього доцільно використати таблицю відповідності, у якій групи вимог зіставляються з основними модулями реалізації та способами їх перевірки. Такий підхід дозволяє формально показати, що вимоги до автентифікації, спільної роботи, сценарного редагування, сюжетної структури, бази знань, експорту та контролю доступу мають не лише словесний опис, а й програмне втілення та підтвердження засобами тестування й статичних перевірок.

Практична цінність такого табличного подання полягає в тому, що воно поєднує три рівні аргументації: вимогу, модуль і тест. Це дає змогу чітко простежити, як функціональна вимога переходить у конкретний програмний механізм і як саме підтверджується її перевірка. Крім того, такий спосіб подання робить зв’язки між елементами системи більш наочними та придатними для подальшого аналізу. Наприклад, вимоги до входу в систему і створення базового користувацького контексту пов'язуються з підсистемою автентифікації та перевіряються серверними й наскрізними тестами. Вимоги до спільної роботи в межах робочого простору підтверджуються сервісами співпраці та тестами сценаріїв запрошення і керування ролями. Вимоги до елементів структури сюжету зіставляються з моделями сцен, сюжетних арок і тем, а їх перевірка підтримується відповідними тестами серверної логіки й наскрізними користувацькими сценаріями. Отже, верифікація в системі має наскрізний характер, а не зводиться до окремих локальних перевірок.

Сформоване такого подання відповідності наведено в таблиці 3.3.

Таблиця 3.3 – Узагальнення відповідності вимог, реалізації та перевірки

Група вимог	**Основні модулі реалізації**
1	**2**
Автентифікація та формування початкового користувацького контексту (FR-01, FR-02)	клієнтський сервіс автентифікації, контрактні позначки захищених процедур

Продовження таблиці 3.3

1	2
Спільна робота в межах робочого простору та рольова модель (FR-03)	сервіси робочих просторів, учасників, запрошень і дозволів
Серії, сценарії та редагування тексту (FR-04, FR-05)	моделі серій, сценаріїв і версій сценарію, відповідні контракти та серверні сервіси
Сюжетна структура і база знань (FR-06, FR-07)	моделі сцен, арок, тем, персонажів, локацій та інші сутності бази знань
Експорт, відтворюваність і контроль якості (FR-08, FR-9)	модулі експорту, Docker-конфігурації, кореневі сценарії pnpm, Playwright- і Vitest-конфігурації

Такий спосіб подання результатів верифікації є корисним і з аналітичного погляду. Він дозволяє побачити, що вимоги до предметної області підтверджуються не ізольованими тестами, а узгодженим набором перевірок, який охоплює моделі даних, серверні сервіси, клієнтські переходи та експлуатаційні сценарії запуску. Отже, таблиця 3.3 виконує не лише довідкову функцію, а й демонструє повноту покриття основних функціональних напрямів розроблюваної системи.

Окрему роль у верифікації відіграють інструменти статичного контролю якості. Команда pnpm lint використовується для перевірки стилю, узгодженості коду та виявлення базових потенційних помилок на ранньому етапі. Команда pnpm check-types забезпечує статичну перевірку типів і контрактів, що особливо важливо для архітектури зі спільним контрактним шаром між клієнтом і сервером [16, 21]. Використання таких засобів дає змогу виявляти частину проблем ще до запуску функціональних тестів, що зменшує ймовірність накопичення помилок на пізніших етапах розробки. У контексті цієї роботи статичні перевірки доповнюють автоматизоване тестування, посилюючи загальну надійність процесу верифікації та підтримуючи узгодженість між окремими частинами програмної системи.

Важливо підкреслити, що наведені засоби верифікації працюють не ізольовано, а як взаємопов’язаний механізм забезпечення якості. Автоматизовані тести перевіряють поведінку системи в прикладних сценаріях, статичний аналіз дозволяє виявляти помилки ще до виконання коду, а контрольоване середовище розгортання забезпечує відтворюваність результатів перевірки. У сукупності це дає змогу оцінювати програмну систему не лише з погляду наявності окремих функцій, а й з погляду стабільності її реалізації, узгодженості між модулями та придатності до подальшого розвитку.

Отже, у третьому розділі показано, що розроблювана система має не лише реалізований функціонал, а й відтворюваний контур запуску, багаторівневу систему тестування та формалізовані засоби верифікації. Поєднання автоматизованих тестів, контрольованого середовища розгортання, статичних перевірок і таблиці відповідності вимог, реалізації та перевірки дає підстави вважати систему технічно перевіреною, відтворюваною та придатною до подальшого супроводу. Таким чином, результати цього розділу підтверджують, що розроблене програмне забезпечення відповідає сформульованим вимогам не лише на концептуальному рівні, а й на рівні практично перевірених механізмів реалізації.

РОЗДІЛ 4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

Під час розробки програмної системи для спільної сценарної роботи працівники виконують переважно інтелектуальну та операторську діяльність із використанням персональних комп'ютерів, периферійного обладнання та мережевих засобів зв'язку. Для такого виду праці питання безпеки життєдіяльності охоплюють як готовність до дій у разі нещасного випадку, так і правильну організацію самого трудового процесу [26-27]. У підручниках з охорони праці та безпеки життєдіяльності наголошується, що безпечне робоче середовище має включати профілактику виробничих ризиків, підготовленість працівників до дій у небезпечних ситуаціях і раціональну організацію праці [26-27]. Саме тому у даному розділі розглянуто дві взаємопов'язані теми: долікарську допомогу при пораненнях і розробку раціональної діяльності трудового колективу в умовах комп'ютеризованого робочого середовища.

4.1 Долікарська допомога при пораненнях

Під час роботи над програмним продуктом імовірність тяжких виробничих поранень є меншою, ніж у промисловому виробництві, однак повністю виключати травмування не можна. У приміщеннях, де розміщено комп'ютерну техніку та допоміжне обладнання, можливі порізи склом, гострими краями металевих конструкцій, ножами для розпакування техніки, а також ушкодження м'яких тканин у разі падіння предметів, руйнування меблів або аварійної евакуації. Тому працівники повинні володіти хоча б базовими навичками надання домедичної допомоги та знати, як діяти до прибуття бригади екстреної медичної допомоги [27].

У навчальній літературі з безпеки життєдіяльності підкреслюється, що допомога при пораненні повинна починатися з оцінки безпеки місця події. Особа, яка надає допомогу, спочатку має переконатися, що їй самій і постраждалому не загрожують додаткові небезпечні фактори, а потім організувати виклик екстреної

медичної допомоги. Після цього оцінюють характер рани, наявність кровотечі, больовий синдром, порушення рухів ушкодженої кінцівки та можливі ознаки перелому [27]. Якщо поранення супроводжується відкритим переломом або значною кровотечею, пріоритетом є припинення крововтрати, адже саме вона найчастіше становить безпосередню загрозу життю.

Базовий алгоритм дій при пораненнях у трудовому колективі доцільно узагальнити так:

- Забезпечити безпеку місця події для рятувальника, постраждалого й оточуючих.
- Викликати екстрену медичну допомогу, коротко повідомивши адресу, характер травми та стан постраждалого.
- Оглянути рану та визначити, чи є масивна зовнішня кровотеча.
- За наявності кровотечі застосувати прямий тиск на рану або інші засоби її зупинки відповідно до порядків МОЗ; у випадках травматичної ампутації чи критичної крововтрати діяти за алгоритмом контролю масивної кровотечі.
- Після зменшення або зупинки кровотечі накласти на рану стерильну або чисту пов’язку.
- Якщо є підозра на перелом, за можливості знерухомити ушкоджену кінцівку підручними або стандартними засобами, не змінюючи її положення силоміць [27].
- Заспокоїти постраждалого, вкрити його для профілактики переохолодження та забезпечити постійний нагляд до прибуття медиків.

Не менш важливим є перелік заборонених дій. При пораненнях не можна торкатися рани брудними руками, промивати глибоку рану випадковими рідинами, самостійно витягати уламки скла чи металу, намагатися вправити уламки кісток, а також залишати постраждалого без спостереження. Якщо стан людини погіршується, необхідно якомога швидше забезпечити повторну медичну оцінку й контроль стану постраждалого [27]. Такі вимоги є особливо актуальними для колективів, у яких більшість працівників не мають медичної освіти й повинні діяти чітко в межах простого, заздалегідь відпрацьованого алгоритму.

Для програмної команди практичне забезпечення готовності до таких випадків має включати організаційні заходи. На робочому місці повинна бути доступна укомплектована аптечка, працівники мають знати її місце розташування, а також порядок оповіщення відповідальних осіб. Доцільно проводити періодичні інструктажі з домедичної допомоги та включати короткий алгоритм дій до локальних інструкцій з охорони праці [26-27]. Отже, навіть у проєкті, пов’язаному з розробкою програмного забезпечення, підготовленість до дій при пораненнях є необхідною складовою культури безпеки трудового колективу.

Важливою складовою підготовки працівників є також розуміння основних ознак, за якими можна оцінити тяжкість поранення. До таких ознак належать інтенсивна кровотеча, швидке промокання пов’язки кров’ю, різкий біль, деформація кінцівки, порушення рухливості, неприродне положення руки чи ноги, блідість шкіри, холодний піт, запаморочення та ознаки шоку. Якщо постраждалий скаржиться на слабкість, спрагу, нудоту або втрачає свідомість, це може свідчити про суттєву крововтрату, навіть якщо сама рана здається не дуже великою. У такому випадку не можна обмежуватися лише накладанням пов’язки, а потрібно якнайшвидше організувати медичну допомогу та постійний контроль стану постраждалого.

У межах офісного або змішаного формату роботи окрему увагу слід приділяти сценаріям, які часто сприймаються як побутові й тому недооцінюються. Наприклад, поріз під час розпакування техніки чи монтажу периферії може супроводжуватися ушкодженням сухожиль або значною кровотечею, хоча зовні така травма здається незначною. Травмування під час падіння може поєднуватися не лише з раною, а й із забоєм, вивихом або переломом. Саме тому надання допомоги при пораненнях не повинно зводитися до механічного наклеювання пластиру: працівник має оцінити загальний стан людини, характер ушкодження та вирішити, чи безпечно залишити постраждалого на місці, чи потрібна термінова медична евакуація.

З позиції організації охорони праці корисно передбачити розподіл ролей у разі нещасного випадку. Один працівник надає домедичну допомогу, другий

викликає екстрену медичну допомогу та зустрічає бригаду, третій повідомляє відповідальну особу або керівника підрозділу, а за потреби організовується звільнення проходу для медиків. Такий порядок скорочує втрату часу й зменшує хаотичність дій у стресовій ситуації. Для невеликої команди розробки це може бути оформлено як коротка інструкція або пам’ятка біля аптечки чи у внутрішній базі знань.

Після надання первинної допомоги й передачі постраждалого медичним працівникам питання безпеки не вичерпується. Необхідно зафіксувати обставини події, проаналізувати причину травмування, перевірити стан обладнання та за потреби переглянути локальні правила безпеки. Якщо травма сталася через несправний стілець, пошкоджений кабель, невдале розміщення техніки або відсутність належного інструменту, усунення цих причин є не менш важливим, ніж сама допомога постраждалому. Таким чином, домедична допомога при пораненнях у трудовому колективі повинна розглядатися не ізольовано, а як частина цілісної системи профілактики нещасних випадків, навчання персоналу та постійного вдосконалення умов праці.

4.2 Розробка раціональної діяльності та створення сприятливих умов трудового

Розробка програмної системи є прикладом діяльності, у якій основне навантаження пов’язане не з важкою фізичною працею, а з тривалим зосередженням уваги, роботою з екранними пристроями, необхідністю безперервної комунікації та високою інтенсивністю розумових операцій. У таких умовах головними ризиками стають зорове стомлення, статичне навантаження на шию, спину і кисті рук, нервово-емоційне напруження, дефіцит рухової активності, а також погіршення психологічного клімату в команді [26-27].

Раціональна діяльність трудового колективу в межах цього проєкту повинна будуватися на поєднанні організаційної дисципліни та ергономічних вимог до робочого місця [26-27]. Робоче місце має забезпечувати достатній простір для

зміни положення тіла, належне освітлення, нормативний мікроклімат, зручне розташування екрана, клавіатури й документів, а також використання стійкого крісла з регулюванням висоти сидіння і нахилу спинки. Для працівників, які займаються програмуванням, тестуванням, написанням технічної документації та проєктуванням інтерфейсів, такі вимоги не є формальністю, а безпосередньо впливають на працездатність, концентрацію уваги та якість виконання завдань.

Практична розробка раціонального режиму праці для колективу має включати:

- Чіткий розподіл ролей і відповідальності між учасниками команди, щоб уникати дублювання функцій і хаотичного навантаження.
- Планування робочого дня із чергуванням інтенсивної розумової праці та регламентованих перерв, необхідних під час роботи з екранними пристроями [26-27].
- Обмеження тривалих безперервних сесій за комп’ютером і заохочення до короткої зміни пози, розминки та відпочинку для очей.
- Організацію робочих місць із належним освітленням, мікрокліматом і мінімізацією відблисків на екранах [26-27].
- Узгоджений порядок командних нарад, повідомлень і дедлайнів, який знижує інформаційне перевантаження.
- Фіксацію вимог охорони праці у внутрішніх інструкціях, правилах трудового розпорядку та локальних актах підприємства.

Окремим складником сприятливих умов праці є психосоціальне середовище. Для ІТ-колективів воно має не менше значення, ніж технічне оснащення робочого місця, оскільки саме через комунікаційні перевантаження, конфлікти ролей, нечіткі пріоритети та постійний стрес часто виникає професійне вигорання. У підручниках з охорони праці та безпеки життєдіяльності наголошується, що безпечні умови праці охоплюють не лише фізичні виробничі фактори, а й психофізіологічні та організаційні чинники, які впливають на стан працівника, його працездатність і стійкість до перевантаження [26-27]. Для розробників програмного забезпечення це означає необхідність прозорого планування задач,

коректного надання зворотного зв’язку, недопущення мобінгу, а також своєчасного реагування керівника на ознаки перевтоми чи тривалої конфліктності в команді.

Важливим інструментом закріплення таких підходів є внутрішні положення підприємства або команди, у яких доцільно визначити порядок інструктажів з охорони праці, правила користування обладнанням, режим перерв, вимоги до організації дистанційної чи офісної роботи, процедури реагування на стресові ситуації та доступність домедичної допомоги [26].

Для працівників, які більшу частину часу проводять за комп’ютером, суттєве значення має правильна ергономіка робочої пози. Екран повинен бути розташований так, щоб уникати надмірного нахилу голови вперед або постійного повороту шиї, клавіатура й миша мають знаходитися в зоні зручного досягання, а висота стола і крісла повинна дозволяти зберігати природне положення рук і спини. За відсутності таких умов навіть відносно безпечна офісна праця з часом призводить до хронічного стомлення, болю в спині, перенапруження м’язів плечового пояса та зниження продуктивності. Тому раціональна організація праці в ІТ-команді має враховувати не лише строки виконання задач, а й фізіологічні можливості людини.

Окремої уваги потребує організація праці у форматі дистанційної або гібридної роботи, яка є поширеною для команд розробки програмного забезпечення. Формально працівник може перебувати поза офісом, однак це не скасовує вимог до безпечного робочого місця та раціонального режиму праці. На практиці роботодавець або керівник команди має визначити базові вимоги до домашнього робочого місця: достатнє освітлення, наявність зручного столу та крісла, відсутність небезпечного розміщення кабелів, можливість робити перерви без постійного перебування перед екраном. Для працівника це означає, що трудова дисципліна і самоконтроль стають частиною особистої безпеки, а не лише питанням продуктивності.

Раціональна діяльність колективу також тісно пов’язана з правильним управлінням навантаженням. У проєктах з розробки вебсистем часто виникають пікові періоди, пов’язані з дедлайнами, виправленням критичних дефектів,

інтеграцією нових модулів або підготовкою демонстраційних матеріалів. Якщо такі періоди не супроводжуються переглядом пріоритетів і перерозподілом задач, окремі учасники команди швидко переходять у режим постійного перевантаження. Це призводить до збільшення кількості помилок, конфліктів у комунікації, погіршення якості рішень і зниження загальної стійкості колективу. Отже, турбота про сприятливі умови праці включає не лише контроль фізичних параметрів середовища, а й виважене керування темпом роботи.

Для підтримання здорового трудового клімату доцільно використовувати прості, але системні організаційні заходи: реалістичне планування спринтів або етапів робіт, узгодження пріоритетів до початку виконання задач, обмеження кількості паралельних термінових доручень, фіксацію зон відповідальності та регулярне обговорення ризиків. Корисною є також практика коротких робочих зустрічей, на яких обговорюються не лише технічні результати, а й перевантаження, блокери та потреба в допомозі. Такий підхід зменшує ймовірність того, що накопичений стрес або нерівномірний розподіл роботи залишаться непоміченими до моменту конфлікту чи професійного виснаження.

Отже, створення сприятливих умов трудового колективу у межах програмного проєкту полягає не лише в забезпеченні комп’ютерною технікою, а й у побудові цілісного безпечного середовища праці. Поєднання ергономічного робочого місця, раціонального режиму праці, зрозумілого розподілу відповідальності, нормативного мікроклімату та здорового психосоціального клімату дає змогу зменшити негативний вплив професійних факторів, зберегти працездатність працівників і підвищити результативність команди.

Не менш важливо, що належна організація умов праці впливає і на довгострокову стійкість команди. У середовищі, де працівники мають змогу працювати в безпечному фізичному просторі, отримують зрозумілі правила взаємодії, можуть дотримуватися режиму праці та відпочинку і не перебувають у стані постійного перевантаження, зменшується ризик професійного вигорання, накопичення помилок і зниження мотивації. Для програмного проєкту це має безпосереднє значення, оскільки якість кінцевого результату значною мірою

залежить від стабільності уваги, послідовності мислення та злагодженості командної роботи.

З практичного погляду результати цього розділу доцільно розглядати як рекомендаційну основу для організації робочого середовища під час виконання подібних ІТ-проєктів. Дотримання вимог домедичної готовності, ергономіки, психофізіологічної безпеки та раціонального планування праці дає змогу не лише виконувати нормативні вимоги, а й підтримувати продуктивний режим роботи впродовж усього життєвого циклу програмної системи. Таким чином, четвертий розділ логічно доповнює технічну частину кваліфікаційної роботи, показуючи, що створення якісного програмного продукту нерозривно пов'язане з безпечними й організовано впорядкованими умовами праці.

ВИСНОВКИ

У кваліфікаційній роботі досягнуто поставленої мети та виконано сформульовані у вступі завдання проєктування і реалізації програмної системи для спільної сценарної роботи з акцентом на структуру сюжету. У ході виконання роботи проаналізовано предметну область, визначено функціональні та нефункціональні вимоги, побудовано логічну архітектуру системи, модель даних і реалізовано основні програмні компоненти розробленої програмної системи.

У результаті створено прототип програмної системи, який забезпечує керування серіями, сценаріями, версіями тексту, базою знань та механізмами контролю доступу. Обґрунтовано використання монорепозиторію як форми організації кодової бази, а також стеку TypeScript, React, Hono, MongoDB та Better Auth. Окремо визначено, що повноцінне IAM/RBAC-рішення класу Keycloak є надмірним для поточного scope, тоді як обраний підхід забезпечує достатній баланс між функціональністю та складністю реалізації.

Важливим результатом роботи є також підтвердження того, що підтримка структури сюжету може бути інтегрована в єдине програмне середовище разом із механізмами спільної роботи, веденням бази знань і рольовим доступом. Це дозволяє розглядати розроблене рішення не як окремий редактор сценарного тексту, а як основу для комплексної вебплатформи, орієнтованої на повніший творчий і організаційний цикл роботи над сценарними матеріалами.

Основні результати виконаної роботи можна узагальнити так:

1. Сформульовано вимоги до програмної системи для спільної сценарної роботи та визначено її ключові сценарії використання.
2. Спроєктовано архітектуру монорепозиторного веб застосунку зі спільним контрактним шаром між клієнтом і сервером.
3. Побудовано модель даних, яка охоплює робочі простори, серії, сценарії, базу знань і формалізовані елементи структури сюжету.
4. Реалізовано механізми автентифікації, авторизації, рольового доступу та клієнт-серверної взаємодії на основі типобезпечних контрактів.

5. Підтверджено працездатність системи засобами автоматизованого тестування, статичної перевірки типів, лінтингу та контрольованого локального розгортання.

Працездатність системи підтверджено тестуванням на різних рівнях, а якість реалізації додатково верифіковано засобами статичного аналізу, перевірки типів і контрольованим процесом розгортання. Практичне значення роботи полягає у створенні основи для подальшого розвитку веб платформи для сценаристів та використанні отриманих рішень у подібних інформаційних системах творчого спрямування. Закладені архітектурні рішення дають змогу розширювати функціональність системи без суттєвого перегляду її базових компонентів.

Отже, поставлені у кваліфікаційній роботі завдання виконано, а мету роботи досягнуто. Отримані результати демонструють цілісний інженерний підхід до створення прикладної веб системи для сценарної роботи. Подальший розвиток роботи може бути пов'язаний із розширенням колаборативних можливостей, удосконаленням інструментів аналітики сюжету та поглибленням інтеграції між редактором сценарію і базою знань.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] D. Sun and C. Sun, "Context-Based Operational Transformation in Distributed Collaborative Editing Systems," IEEE Transactions on Parallel and Distributed Systems, vol. 20, no. 10, pp. 1454-1470, Oct. 2009, doi: 10.1109/TPDS.2008.240.

[2] Final Draft, Inc., "Final Draft," [Online]. Available: https://www.finaldraft.com/. Accessed: May 3, 2026.

[3] Celtx, Inc., "Celtx," [Online]. Available: https://www.celtx.com/. Accessed: May 3, 2026.

[4] WriterSolo, "WriterSolo," [Online]. Available: https://www.writersolo.com/. Accessed: May 3, 2026.

[5] The Quote-Unquote Company, "Highland Pro," [Online]. Available: https://quoteunquoteapps.com/highland/. Accessed: May 3, 2026.

[6] H. Washizaki, Ed., Guide to the Software Engineering Body of Knowledge (SWEBOK Guide), Version 4.0, IEEE Computer Society, 2024. [Online]. Available: https://www.swebok.org. Accessed: May 18, 2026.

[7] ISO/IEC/IEEE 29148:2018, "Systems and software engineering — Life cycle processes — Requirements engineering," ISO, Geneva, Switzerland, 2018.

[8] R. Sandhu, D. Ferraiolo, and R. Kuhn, "The NIST Model for Role-Based Access Control: Towards a Unified Standard," in Proceedings of the Fifth ACM Workshop on Role-Based Access Control, Berlin, Germany, 2000, pp. 47-63, doi: 10.1145/344287.344301.

[9] M. Alavi and D. E. Leidner, "Review: Knowledge Management and Knowledge Management Systems: Conceptual Foundations and Research Issues," MIS Quarterly, vol. 25, no. 1, pp. 107-136, Mar. 2001, doi: 10.2307/3250961.

[10] M. Al-Emran, V. Mezhuyev, A. Kamaludin, and K. Shaalan, "The impact of knowledge management processes on information systems: A systematic review," International Journal of Information Management, vol. 43, pp. 173-187, Dec. 2018, doi: 10.1016/j.ijinfomgt.2018.08.001.

[11] ISO/IEC 25010:2023, "Systems and software engineering — Systems and

software Quality Requirements and Evaluation (SQuaRE) — Product quality model," ISO, Geneva, Switzerland, 2023.

[12] Р. Бармак and І. Дегодюк, “Методологія специфікаційно-орієнтованої розробки програмного забезпечення за допомогою інструментів штучного інтелекту,” in Матеріали IX Міжнародної студентської науково-технічної конференції “Природничі та гуманітарні науки. Актуальні питання”, Тернопіль, Україна, 24–25 квіт. 2026 р., pp. 154–155. [Online]. Available: http://elartu.tntu.edu.ua/handle/lib/52378.

[13] B. P. Deepak, "Spec-Driven Development: From Code to Contract in the Age of AI Coding Assistants," arXiv:2602.00180, 2026. [Online]. Available: https://arxiv.org/abs/2602.00180. Accessed: Mar. 30, 2026.

[14] ISO/IEC/IEEE 12207:2017, "Systems and software engineering — Software life cycle processes," ISO, Geneva, Switzerland, 2017.

[15] Д. М. Михалик, Г. Б. Цуприк, and В. М. Бревус, Методичні вказівки до виконання кваліфікаційної роботи бакалавра для здобувачів спеціальності 121 – Інженерія програмного забезпечення, всіх форм навчання. Тернопіль, Україна: Тернопільський національний технічний університет імені Івана Пулюя, 2024. [Online]. Available: http://elartu.tntu.edu.ua/handle/lib/50317. Accessed: May 3, 2026.

[16] oRPC contributors, "oRPC: Typesafe APIs Made Simple," [Online]. Available: https://orpc.dev/. Accessed: May 3, 2026.

[17] S. Newman, "Building Microservices: Designing Fine-Grained Systems," 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2021.

[18] M. Richards and N. Ford, "Fundamentals of Software Architecture: An Engineering Approach," 1st ed. Sebastopol, CA, USA: O'Reilly Media, 2020.

[19] R. Cattell, "Scalable SQL and NoSQL Data Stores," ACM SIGMOD Record, vol. 39, no. 4, pp. 12-27, May 2011, doi: 10.1145/1978915.1978919.

[20] S. Field, "Screenplay: The Foundations of Screenwriting," rev. ed. New York, NY, USA: Delta Trade Paperbacks, 2005.

[21] Z. Gao, C. Bird, and E. T. Barr, "To Type or Not to Type: Quantifying Detectable Bugs in JavaScript," in 2017 IEEE/ACM 39th International Conference on

Software Engineering (ICSE), Buenos Aires, Argentina, 2017, pp. 758-769, doi: 10.1109/ICSE.2017.75.

[22] І. Дегодюк and Р. Бармак, “Інтеграція деталізованих дозволів Keycloak у процес авторизації користувачів системи,” in Матеріали IX Міжнародної студентської науково-технічної конференції “Природничі та гуманітарні науки. Актуальні питання”, Тернопіль, Україна, 24–25 квіт. 2026 р., pp. 178–179. [Online]. Available: http://elartu.tntu.edu.ua/handle/lib/52393.

[23] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," Linux Journal, vol. 2014, no. 239, Art. 2, Mar. 2014.

[24] G. J. Myers, C. Sandler, and T. Badgett, "The Art of Software Testing," 3rd ed. Hoboken, NJ, USA: John Wiley & Sons, 2011.

[25] J. Humble and D. Farley, "Continuous Delivery: Reliable Software Releases through Build, Test, and Deploy Automation," Upper Saddle River, NJ, USA: Addison-Wesley, 2010.

[26] К. Н. Ткачук and Н. О. Халімовський, Eds., Основи охорони праці. Київ, Україна: Основа, 2006.

[27] Є. П. Желібо and В. В. Зацарний, Безпека життєдіяльності: підручник. Київ, Україна: Каравела, 2023.

ДОДАТКИ

Додаток А – Тези конференції по темі “Методологія специфікаційно-орієнтованої розробки програмного забезпечення за допомогою інструментів штучного інтелекту”

IX Міжнародна студентська науково - технічна конференція
"ПРИРОДНИЧІ ТА ГУМАНІТАРНІ НАУКИ. АКТУАЛЬНІ ПИТАННЯ"

УДК 004.41:004.8
Бармак Р. – ст. гр. СП-41, Дегодюк І. – ст. гр. СП-31
Тернопільський національний технічний університет імені Івана Пулюя

МЕТОДОЛОГІЯ СПЕЦИФІКАЦІЙНО-ОРІЄНТОВАНОЇ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ЗА ДОПОМОГОЮ ІНСТРУМЕНТІВ ШТУЧНОГО ІНТЕЛЕКТУ

Науковий керівник: PhD Бревус В. М.

Barmak R., Dehodiuk I.
Ternopil Ivan Puluj National Technical University

METHODOLOGY OF SPECIFICATION-ORIENTED SOFTWARE DEVELOPMENT USING ARTIFICIAL INTELLIGENCE TOOLS

Supervisor: PhD Brevus V. M.

Ключові слова: специфікаційно-орієнтована розробка, програмування з використанням штучного інтелекту, автоматизація розробки програмного забезпечення.
Keywords: Spec-Driven Development, AI-assisted Programming, Software Development Automation

Вступ. Стрімкий розвиток великих мовних моделей відкриває нові можливості автоматизації процесу розробки програмного забезпечення. Проте, згідно з даними галузевих звітів, безпосередня генерація коду інструментами штучного інтелекту може призводити до проблем: порушення архітектурних принципів, відхилення від встановлених конвенцій кодової бази та генерація нерелевантного або некоректного коду[1].

Мета роботи – дослідження і формалізація методології специфікаційно-орієнтованої розробки, що забезпечує структуроване управління контекстом для ефективної розробки програмного забезпечення з використанням інструментів штучного інтелекту.

Основна частина. Методологія специфікаційно-орієнтованої розробки базується на принципі «пріоритет специфікації». Ця концепція реалізує необхідний в епоху ШІ-асистентів перехід від парадигми написання коду до розробки на основі контрактів[2]. Методологія включає три ключові компоненти:

1. Конституція проєкту – нормативний документ, що визначає архітектурні принципи, конвенції кодування та обов'язкові перевірки якості. Як свідчать результати досліджень, такий підхід дозволяє впроваджувати принцип «безпеки за проектуванням» безпосередньо в процеси автоматизованої генерації коду[3]. Конституція інтегрується у контекст роботи асистента штучного інтелекту за допомогою спеціальних файлів інструкцій, які автоматично завантажуються самими інструментами розробки.
2. Ієрархія специфікацій – структурована система документів:
   - План – технічний контекст, архітектурні рішення, фази виконання;
   - Специфікація – користувацькі сценарії з виставленими пріоритетами для фокусу, функціональні вимоги, крайні випадки;
   - Завдання – декомпозовані атомарні завдання з чіткими критеріями прийняття.

3. Зворотний зв'язок та ітерації – механізми перевірки згенерованого коду через автоматичні перевірки типів у коді та якості написаного коду.

Ключовою перевагою методології є інтеграція контексту безпосередньо в репозиторій проєкту, що дозволяє асистенту штучного інтелекту автоматично враховувати архітектурні обмеження та конвенції при генерації коду.

На відміну від класичної методології Agile, де документація часто мінімізується на користь робочого коду, специфікаційно-орієнтована розробка встановлює специфікацію як основний артефакт. Такий підхід узгоджується з принципами формальних методів розробки, водночас залишаючись практичним і гнучким до змін вимог. Порівняно з традиційним підходом розробки через тестування, специфікаційно-орієнтований підхід розширює концепцію контракту за межі модульних тестів, охоплюючи архітектурні рішення, користувацький досвід та бізнес-логіку.

Перехід до специфікаційно-орієнтованої розробки вимагає трансформації процесів команди розробки. Критичним фактором успіху є створення культури документування перед написанням коду, що традиційно може зустрічати опір від розробників, звиклих до безпосередньої імплементації. Важливим елементом є інтеграція специфікацій у систему контролю версій поряд з кодом, що забезпечує синхронізацію документації та реалізації. Додатковою перевагою такого підходу є можливість проводити перевірку коду на рівні специфікацій до початку розробки, що дозволяє виявляти архітектурні проблеми та непорозуміння на ранніх етапах.

Результати. Застосування методології специфікаційно-орієнтованої розробки продемонструвало:

- зменшення кількості виправлень коду під час під час перевірки коду, згенерованого асистентом штучного інтелекту;
- збереження і дотримання архітектурної цілісності проєкту під час генерації коду штучним інтелектом;
- можливість паралельної роботи над різними функціональними модулями без конфліктів;
- підвищення передбачуваності результатів генерації коду штучним інтелектом;

Висновки. Методологія специфікайно-орієнтованої розробки ефективно вирішує проблему контекстного управління в розробці, забезпечуючи структурований підхід до формулювання завдань та перевірки результатів.

[1] D. DeBellis, K. Storer, N. Harvey, M. Beane, R. Edwards, E. Fraser et al, "DORA 2025 State of AI-Assisted Software Development Report," DORA, Google, 2025. [Online]. Available: https://dora.dev/dora-report-2025 [Accessed: Mar. 30, 2026].

[2] B. P. Deepak, "Spec-Driven Development: From Code to Contract in the Age of AI Coding Assistants," 2026. [Online]. Available: https://arxiv.org/abs/2602.00180 (accessed Mar. 30, 2026).

[3] R. M. Srinivas, "Constitutional Spec-Driven Development: Enforcing Security by Construction in AI-Assisted Code Generation," 2026. [Online]. Available: https://arxiv.org/abs/2602.02584 (accessed Mar. 30, 2026).

Додаток Б – Тези конференції по темі “Інтеграція деталізованих дозволів Keycloak у процес авторизації користувачів системи”

IX Міжнародна студентська науково - технічна конференція
"ПРИРОДНИЧІ ТА ГУМАНІТАРНІ НАУКИ. АКТУАЛЬНІ ПИТАННЯ"

УДК 004.41
Дегодюк І. – ст. гр. СП-31, Бармак Р. – ст. гр. СП-41
Тернопільський національний технічний університет імені Івана Пулюя

ІНТЕГРАЦІЯ ДЕТАЛІЗОВАНИХ ДОЗВОЛІВ KEYCLOAK У ПРОЦЕС АВТОРИЗАЦІЇ КОРИСТУВАЧІВ СИСТЕМИ

Науковий керівник: PhD Бревус В. М.

Dehodiuk I., Barmak R.
Ternopil Ivan Puluj National Technical University

INTEGRATION OF KEYCLOAK FINE-GRAINED PERMISSIONS INTO USER AUTHENTICATION FLOW

Supervisor: PhD Brevus V. M.
Ключові слова: Аутентифікація Користувачів, Деталізовані Дозволи, Keycloak.
Keywords: User Authentication, Fine-Grained Permissions, Keycloak.

Вступ. Процес аутентифікації користувачів у системі відповідає за розподіл дозволів для конкретного користувача. На поточний момент, у цього процесу немає стандарту, що забезпечуватиме гнучкість та безпеку системи. Це призводить до зловживання користувачів власними правами, навіть, у додатках світового значення.

Мета роботи — дослідження підходів для інтеграції деталізованих дозволів Keycloak у мікросервісну систему. Ознайомлення із доступними програмними інтерфейсами та пошук оптимального рівня делегації логіки процесу аутентифікації.

Основна частина. Починаючи з версії 26.2.0, Keycloak дозволяє налаштовувати деталізовані дозволи для клієнтських застосунків. Дозвіл складається із декількох незалежних компонентів [1]:

1. Ресурс: об’єкт системи, доступ до якого повинен бути обмеженим. Keycloak не обмежує формат ресусу, тому він може бути представлений будь-яким чином. Опційно, ресурс може оголошувати сфери застосування, для більш детального контролю дозволу.

2. Політика: визначає умову необхідну для отримання доступу. Умова може перевіряти певну властивість користувача, наявність атрибутів. Додатково, умови можуть бути тимчасовими.

3. Дозвіл: сутність об’єднує одну, або більше, політику із ресурсом та його сферами застосування. Keycloak реалізовує програмний інтерфейс із трьома режимами роботи, який дозволяє регулювати рівень делегації логіки [2]. Режим визначається за допомогою параметру response_mode, що приймає 3 значення: 1. Decision: повна делегація авторизації, Keycloak самостійно приймає рішення щодо доступу користувача. 2. Permissions: Keycloak поверне список дозволених дій для поточного користувача. Програма повинна обробити їх та прийняти рішення самостійно. 3. Параметр відсутній: Keycloak поверне RPT токен, що містить дозволи та додаткові дані про користувача. Підхід забезпечує найнижчий рівень делегації. Токен може обмінюватись між різними мікросервісами.

Повноцінна система вимагає великої кількості дозволів та чіткої стратегії для їхньої організації [3]. Деталізовані дозволи Keycloak можуть бути налаштовані за допомогою Terraform провайдеру, що спрощує процес управління та забезпечує версійність.

На рисунку 1 представлена високорівнева структура системи після інтеграції деталізованих дозволів Keycloak за допомогою описаних підходів. Залежно від рівня делегації, роль виконавця політик (PEP) може виконувати, або API Gateway, або сервіс що опрацьовує запит.

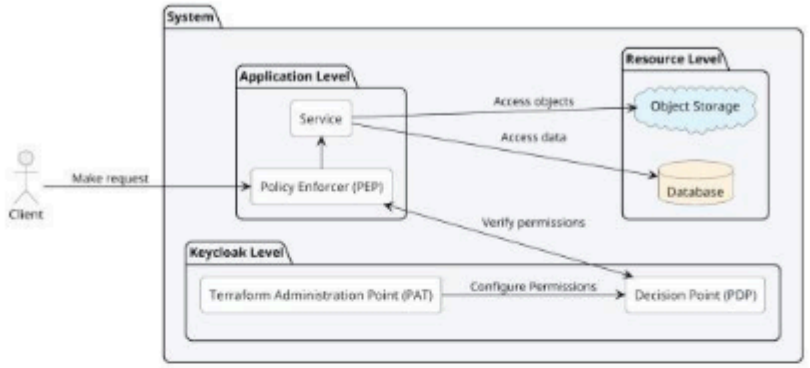

Рисунок 1 — Структурна діаграма системи

Результати. Досліджені підходи для інтеграції деталізованих дозволів Keycloak із інтеграцією Terraform провайдеру дозволяють:

- Централізувати та стандартизувати процес авторизації користувачів.
- Налаштовувати рівні доступу користувачів без змін у програмному коді.
- Відслідковувати історію змін конфігурації.
- Відновити створену конфігурацію у випадку критичних збоїв системи.

Висновки. Досліджені підходи інтеграції деталізованих дозволів Keycloak забезпечують низку переваг та значно спрощують процес авторизації користувачів системи. Можливість регулювання рівня делегації робить процес гнучким та уможливлює його застосування у різноманітних системах. Використання Terraform забезпечує версійність та відновлюваність конфігурації, підвищуючи надійність усього потоку авторизації.

Список джерел.

[1] K. M and N. Kumaresh, "Implementation of Dynamic Role Based Access Control in Multi-Tenant Cloud". Accessed: Mar. 23, 2026. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/11234495

[2] Gamayanto, Indra, Michael Christ Kurniawan, and Gabriello Klavin Sanyoto. "Security Evaluation of Keycloak-Based Role-Based Access Control in Microservice Architectures Using the OWASP ASVS Framework". Accessed: Mar. 23, 2026. [Online]. Available: https://jurnal.polibatam.ac.id/index.php/JAIC/article/view/11604

[3] V. Voicu, D. Petreuş, E. Cebuc and S. B. Marcu, "University Identity and Access Management Infrastructure with Keycloak: Lessons Learned". Accessed: Mar. 23, 2026. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10722517/metrics#metrics

Додаток В – Лістинг коду сервісу серій

```
import { inject, injectable } from 'tsyringe';
import type z from 'zod';

import { errorCodes, WORKSPACE_ROLES } from '@kaeri/shared';
import { createSeriesInputSchema, updateSeriesPatchSchema } from
'@kaeri/shared/contract/series.contract';

import { CharacterModel } from '@~/db/models/character.model';
import { LocationModel } from '@~/db/models/location.model';
import { PropModel } from '@~/db/models/prop.model';
import { SceneModel } from '@~/db/models/scene.model';
import { ScriptModel } from '@~/db/models/script.model';
import { SeriesModel } from '@~/db/models/series.model';
import { StoryArcModel } from '@~/db/models/story-arc.model';
import { WorkspaceMemberModel } from
'@~/db/models/workspace-member.model';
import { WorkspaceModel } from '@~/db/models/workspace.model';
import { TOKENS } from '@~/di/tokens';
import type { PermissionService } from
'@~/features/collaboration/permission.service';
import { buildCacheKey, CACHE_TTL } from
'@~/features/valkey/valkey.constants';
import { ORPCBadRequestError, ORPCNotFoundError,
ORPCUnprocessableContentError } from '@~/lib/orpc-error-wrapper';

import type { TypedEventBus } from '../events/event-bus';
import type { iWithLogger, LoggerFactory } from
'../logger/logger.types';
import type { ValkeyService } from '../valkey/valkey.service';
import { SERIES_EVENTS } from './series.events';

type CreateSeriesInput = z.infer<typeof createSeriesInputSchema>;
type UpdateSeriesPatch = z.infer<typeof updateSeriesPatchSchema>;

@injectable()
export class SeriesService implements iWithLogger {
  public readonly logger: iWithLogger['logger'];

  constructor(
    @inject(TOKENS.LoggerFactory) loggerFactory: LoggerFactory,
    @inject(TOKENS.ValkeyService) private readonly valkey:
ValkeyService,
    @inject(TOKENS.EventBus) private readonly eventBus:
TypedEventBus,
    @inject(TOKENS.PermissionService) private readonly
permissionService: PermissionService,
  ) {
    this.logger = loggerFactory.create('series-service');
  }
```

```
  public async create(data: CreateSeriesInput, userId: string) {
    if (!data.title?.trim()) {
      throw ORPCBadRequestError(errorCodes.SERIES_TITLE_REQUIRED);
    }

    // Get or create user's personal workspace
    let workspace = await WorkspaceModel.findOne({ creatorId:
userId, isPersonal: true });
    if (!workspace) {
      workspace = await WorkspaceModel.create({
        name: 'Personal Workspace',
        isPersonal: true,
        creatorId: userId,
      });
      // Create owner membership
      await WorkspaceMemberModel.create({
        workspaceId: workspace._id,
        userId,
        role: WORKSPACE_ROLES.OWNER,
        joinedAt: new Date(),
      });
    }

    const series = await SeriesModel.create({
      title: data.title,
      type: data.type ?? 'standalone',
      genre: data.genre,
      logline: data.logline,
      coverUrl: data.coverUrl,
      workspaceId: workspace._id,
      lastEditedAt: new Date(),
    });

    this.logger.info('Series created', { seriesId: series._id,
title: series.title });

    // Emit event for cache invalidation
    this.eventBus.emit(SERIES_EVENTS.CREATED, { seriesId: series._id
});

    return series;
  }

  public async update(seriesId: string, patch: UpdateSeriesPatch) {
    const series = await SeriesModel.findById(seriesId);
    if (!series) {
      throw ORPCNotFoundError(errorCodes.SERIES_NOT_FOUND);
    }

    if (patch.title !== undefined) series.title = patch.title;
    if (patch.type !== undefined) series.type = patch.type;
    if (patch.genre !== undefined) series.genre = patch.genre;
```

```
    if (patch.logline !== undefined) series.logline = patch.logline;
    if (patch.coverUrl !== undefined) series.coverUrl =
patch.coverUrl;

    series.lastEditedAt = new Date();
    await series.save();

    // Emit event for cache invalidation
    this.eventBus.emit(SERIES_EVENTS.UPDATED, { seriesId: series._id
});
    this.logger.info('Series updated', { seriesId: series._id });

    return series;
  }

  public async delete(seriesId: string) {
    const series = await SeriesModel.findById(seriesId);
    if (!series) {
      throw ORPCNotFoundError(errorCodes.SERIES_NOT_FOUND);
    }

    // Check for dependent scripts
    const scriptCount = await ScriptModel.countDocuments({ seriesId
});
    if (scriptCount > 0) {
      throw
ORPCUnprocessableContentError(errorCodes.SERIES_DELETE_HAS_DEPENDENC
IES);
    }

    // TODO: Check for KB entities, canvas nodes/edges, etc.

    await SeriesModel.deleteOne({ _id: seriesId });

    // Emit event for cache invalidation
    this.eventBus.emit(SERIES_EVENTS.DELETED, { seriesId });
    this.logger.info('Series deleted', { seriesId });

    return { success: true };
  }

  /**
   * List series accessible to the user.
   * Only returns series from workspaces where the user is a member.
   */
  public async list(userId: string, limit = 20, offset = 0) {
    // Get all workspace IDs where the user is a member
    const memberships = await WorkspaceMemberModel.find({ userId
}).select('workspaceId').lean();
    const workspaceIds = memberships.map((m) => m.workspaceId);

    // If user has no workspace memberships, return empty list
    if (workspaceIds.length === 0) {
```

```
      return { items: [], total: 0 };
    }

    const [items, total] = await Promise.all([
      SeriesModel.find({ workspaceId: { $in: workspaceIds } })
        .sort({ lastEditedAt: -1 })
        .limit(limit)
        .skip(offset)
        .lean(),
      SeriesModel.countDocuments({ workspaceId: { $in: workspaceIds
} }),
    ]);

    return { items, total };
  }

  public async get(seriesId: string) {
    return this.valkey.cached(buildCacheKey.series(seriesId),
CACHE_TTL.ENTITY_MEDIUM, async () => {
      this.logger.debug('Fetching series from database', { seriesId
});
      const series = await SeriesModel.findById(seriesId);
      if (!series) throw
ORPCNotFoundError(errorCodes.SERIES_NOT_FOUND);

      return series;
    });
  }

  public async exportSummary(seriesId: string) {
    const series = await this.get(seriesId);
    const scripts = await ScriptModel.find({ seriesId })
      .select('_id title authors genre logline coverUrl
lastEditedAt')
      .sort({ lastEditedAt: -1 })
      .lean();

    return { series, scripts };
  }

  public async getAnalytics(seriesId: string) {
    const series = await SeriesModel.findById(seriesId);
    if (!series) {
      throw ORPCNotFoundError(errorCodes.SERIES_NOT_FOUND);
    }

    // Fetch all counts in parallel
    const [
      scriptCount,
      characterCount,
      locationCount,
      propCount,
      sceneCount,
```

```
      characters,
      locations,
      storyArcs,
      scripts,
    ] = await Promise.all([
      ScriptModel.countDocuments({ seriesId }),
      CharacterModel.countDocuments({ seriesId }),
      LocationModel.countDocuments({ seriesId }),
      PropModel.countDocuments({ seriesId }),
      SceneModel.countDocuments({ seriesId }),
      CharacterModel.find({ seriesId }).select('_id name
appearances').lean(),
      LocationModel.find({ seriesId }).select('_id name').lean(),
      StoryArcModel.find({ seriesId }).select('_id name status
keyBeats').lean(),
      ScriptModel.find({ seriesId }).select('_id content
updatedAt').lean(),
    ]);

    // Calculate total word count from scripts
    let totalWordCount = 0;
    const writingActivity: Array<{ date: string; wordCount: number;
scriptId?: string }> = [];

    for (const script of scripts) {
      const content = script.content ?? '';
      const words = content.split(/\s+/).filter(Boolean).length;
      totalWordCount += words;

      // Track writing activity by date
      if (script.updatedAt) {
        const date = new
Date(script.updatedAt).toISOString().split('T')[0];
        writingActivity.push({
          date,
          wordCount: words,
          scriptId: script._id,
        });
      }
    }

    // Get scenes grouped by location
    const scenesWithLocation = await SceneModel.find({ seriesId,
locationId: { $exists: true, $ne: null } })
      .select('locationId')
      .lean();

    const locationSceneCount = new Map<string, number>();
    for (const scene of scenesWithLocation) {
      if (scene.locationId) {
        locationSceneCount.set(scene.locationId,
(locationSceneCount.get(scene.locationId) ?? 0) + 1);
      }
```

```
    }

    const locationDistribution = locations
      .map((loc) => ({
        locationId: loc._id,
        locationName: loc.name,
        sceneCount: locationSceneCount.get(loc._id) ?? 0,
      }))
      .sort((a, b) => b.sceneCount - a.sceneCount);

    // Calculate character appearances
    const characterAppearances = characters
      .map((char) => ({
        characterId: char._id,
        characterName: char.name,
        appearanceCount: char.appearances?.length ?? 0,
      }))
      .sort((a, b) => b.appearanceCount - a.appearanceCount);

    // Story arc progress
    const storyArcProgress = storyArcs.map((arc) => ({
      arcId: arc._id,
      arcName: arc.name,
      status: arc.status,
      beatCount: arc.keyBeats?.length ?? 0,
    }));

    return {
      overview: {
        scriptCount,
        characterCount,
        locationCount,
        propCount,
        sceneCount,
        totalWordCount,
      },
      locationDistribution,
      characterAppearances,
      storyArcProgress,
      writingActivity,
    };
  }

  /**
   * Transfer a series to another workspace.
   *
   * Requirements:
   * - User must have ADMIN or higher in the source workspace
   * - User must have ADMIN or higher in the destination workspace
   * - All content (scripts, KB entities, canvas) is preserved
atomically
   *
   * @param seriesId - The series to transfer
```

```
   * @param targetWorkspaceId - The destination workspace
   * @param userId - The user performing the transfer
   */
  public async transfer(seriesId: string, targetWorkspaceId: string,
userId: string) {
    // Fetch the series first
    const series = await SeriesModel.findById(seriesId);
    if (!series) {
      throw ORPCNotFoundError(errorCodes.SERIES_NOT_FOUND);
    }

    const fromWorkspaceId = series.workspaceId;

    // Cannot transfer to the same workspace
    if (fromWorkspaceId === targetWorkspaceId) {
      throw
ORPCBadRequestError(errorCodes.SERIES_ALREADY_IN_WORKSPACE);
    }

    // Verify ADMIN permissions in source workspace
    await this.permissionService.requirePermission(fromWorkspaceId,
userId, WORKSPACE_ROLES.ADMIN);

    // Verify ADMIN permissions in destination workspace
    await
this.permissionService.requirePermission(targetWorkspaceId, userId,
WORKSPACE_ROLES.ADMIN);

    // Atomic update - just change the workspaceId
    // All linked content (scripts, KB entities, canvas) remains
tied to the series via seriesId
    await SeriesModel.updateOne({ _id: seriesId }, { $set: {
workspaceId: targetWorkspaceId } });

    // Emit event for cache invalidation
    this.eventBus.emit(SERIES_EVENTS.TRANSFERRED, {
      seriesId,
      fromWorkspaceId,
      toWorkspaceId: targetWorkspaceId,
      transferredById: userId,
    });

    this.logger.info('Series transferred', {
      seriesId,
      fromWorkspaceId,
      toWorkspaceId: targetWorkspaceId,
      transferredById: userId,
    });

    return { success: true };
  }
}
```