Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра мережевих технологій

Магістерська робота

освітній ступінь – магістр

на тему: **«ПОБУДОВА БАГАТОРІВНЕВОГО ВЕБ-ЗАСТОСУВАННЯ З ВИСОКОЮ ДОСТУПНІСТЮ НА ПЛАТФОРМІ GOOGLE CLOUD PLATFORM»**

Виконав: студент 2-го року навчання,
Спеціальності
121 Інженерія програмного забезпечення

Василенко Олександр Сергійович

Керівник Черкасов Д.І.,
кандидат технічних наук, старший викладач

Рецензент_____
(прізвище та ініціали)

Магістерська робота захищена з оцінкою_____

Секретар ЕК_____

«____»_____ 2022 р.

Київ 2022

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мережних технологій

ЗАТВЕРДЖУЮ
зав. кафедри інформатики, к.ф-м.н., доц.
_____ Гороховський С.С.
(підпис)
“____”_____ 2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на магістерську роботу

студенту 2 р.н. магістерської програми 121 Інженерія програмного забезпечення, Василенку Олександру Сергійовичу
Тема: Побудова багаторівневого веб-застосування з високою доступністю на платформі Google Cloud Platform

Зміст ТЧ до магістерської роботи:

Зміст
Анотація
Вступ
1. Використання хмарних платформ для розробки багаторівневих веб-застосувань з високою доступністю
2. Розробка веб-застосування для бібліотеки з використанням GCP
Висновки
Список використаних джерел
Додатки

Дата видачі “___” _____ 2021 р.

Керівник
Черкасов Д.І., ст. викладач, к.т.н. _____
(підпис)

Завдання отримав
Василенко О.С. _____
(підпис)

Тема: Побудова багаторівневого веб-застосування з високою доступністю на платформі Google Cloud Platform

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на магістерську роботу.	16.10.2021	
2.	Огляд літератури за темою роботи.	31.12.2021	
3.	Здійснення аналізу сучасних методів забезпечення високої доступності веб-застосувань	31.01.2022	
4.	Огляд можливостей Google Cloud Platform та порівняння з аналогами	28.02.2022	
5.	Розробка архітектури проекту, структури сайту, схеми даних, вибір набору технологій	31.03.2022	
6.	Розробка веб-застосування для бібліотеки, інтеграція з GCP	29.05.2022	
7.	Написання пояснювальної записки	12.06.2022	
8.	Створення слайдів для доповіді та написання доповіді.	14.06.2022	
9.	Аналіз отриманих результатів з керівником, написання доповіді та попередній захист магістерської роботи	16.06.2022	
10.	Вдосконалення роботи за результатами попереднього захисту	23.06.2022	
11.	Остаточне оформлення пояснювальної роботи та слайдів	30.06.2022	
12.	Захист магістерської роботи	07.07.2022	

Студент **Василенко Олександр Сергійович**

Керівник **Черкасов Дмитро Іванович**

«_____»_____ 2021 р.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

GCP – Google Cloud Platform.

SPA – Single Page Application.

AWS – Amazon Web Services.

АНОТАЦІЯ

В рамках даної роботи проведено огляд хмарної платформи Google Cloud Platform та її аналогів у якості засобу для розміщення багаторівневого веб-застосування. Проаналізовано можливості платформи для забезпечення високої доступності та розроблено веб-застосування для бібліотеки, що використовує засоби балансування навантаження для забезпечення відмовостійкості.

Ключові слова: хмарна платформа, Google Cloud Platform, GCP, веб-застосування, балансування навантаження.

ВСТУП

Завдяки стрімкому розвитку хмарних сервісів, вони стають все більш доступними та функціональними. Хмарні обчислення та хмарні платформи широко використовуються у різних галузях підприємництва як альтернатива побудові власної інфраструктури. Такі сервіси, як Google Cloud Platform, Amazon Web Services, Microsoft Azure та інші, застосовуються дослідниками у багатьох наукових галузях, наприклад, великі дані, машинне навчання, комп'ютерний зір. Крім використання хмарних обчислень для розвитку науки та технологій, ці сервіси надають широкий вибір інструментів для комерційних та некомерційних проектів, наприклад, засоби забезпечення високої доступності та відмовостійкості. Це дозволяє розробникам сконцентрувати зусилля на розробці програмної системи замість створення та підтримки складної серверної інфраструктури. Таким чином, використання хмарних платформ може зменшити тривалість та вартість розробки. Саме тому побудова багаторівневих веб-застосувань з високою доступністю на платформі GCP є актуальною.

Таким чином, об'єктом дослідження є хмарні платформи. Ця галузь інформаційних технологій набула значного розвитку за останні роки, завдяки чому кількість хмарних сервісів збільшується, а їх функціонал - розширюється. Залежно від послуг, що надає хмара, їх поділяють на три типи: infrastructure as a service, platform as a service та software as a service. У свою чергу, IaaS дозволяє користувачам отримувати доступ до обчислювальних ресурсів з підготовленою інфраструктурою у віртуальному середовищі. Такий підхід значно спрощує процес розробки та розміщення веб-застосувань, адже відповідальність за складні інфраструктурні процеси, наприклад, виділення ресурсів для віртуальних серверів, балансування навантаження та мережеві налаштування. Зокрема, це дозволяє розробникам не займатися плануванням внутрішньої серверної архітектури для

забезпечення високої доступності розроблюваних веб-застосувань. Однак постає проблема аналізу методів, що гарантують відмовостійкость під високим навантаженням, від різних постачальників хмарних послуг. Однією із найпопулярніших хмарних платформ є GCP, тому предметом дослідження є використання її для створення багаторівневих веб-застосувань з високою доступністю.

Метою роботи є створення онлайн-системи для бібліотеки на платформі GCP та дослідження доступних методів забезпечення доступності та відмовостійкості, що надаються платформою, в умовах змінного навантаження. Система використовує авторизацію, надає можливості перегляду каталогу книжок та пошуку в ньому, надає можливість замовлення книжок з бібліотеки, а також має панель керування для адміністратора.

1 ВИКОРИСТАННЯ ХМАРНИХ ПЛАТФОРМ ДЛЯ РОЗРОБКИ БАГАТОРІВНЕВИХ ВЕБ-ЗАСТОСУВАНЬ

1.1 Багаторівневі веб застосування

Багаторівнева архітектура — це перевірена часом модель архітектури програмного забезпечення. Вона підходить для підтримки веб-застосувань корпоративного рівня, надаючи високий рівень масштабованості, безпеки, відмовостійкості, повторного використання та здатність оновлення даних, коду та обладнання без переривання працездатності. Це допомагає розробникам створювати гнучкі додатки, частини яких можна використовувати повторно. Найчастіше розглядають триярусну архітектуру, за якої багаторівневе застосування поділяється на три окремих частин: рівень представлення, рівень бізнес-логіки (прикладний рівень) та рівень керування даними. Зазвичай кожен з рівнів являє собою програму, що виконуються окремо. [1]

У будь-якій варіації багаторівневої архітектури верхнім є саме рівень презентації (представлення). Цей рівень являє собою користувацький інтерфейс, що може бути як застосуванням, встановленим на певному обчислювальному пристрої, так і веб-сторінкою. Основна мета цього рівня – надавати користувачам засіб для створення запитів до системи та зручний спосіб перегляду результату цих запитів, поданих у зручному для сприйняття форматі. [2]

Наступний рівень відповідає за бізнес-логіку застосування. На цьому етапі здійснюється перевірка запитів від користувача, здійснення певних обчислень, перетворень над даними та відправка результатів у відповідь. Бізнес-логіка діє як свого роду інтерфейс між клієнтом та рівнем доступу до даних. Дуже часто цей рівень може поділятися на більшу кількість шарів, що мають різне призначення, або коли запити мають проходити послідовну обробку. Призначення певних функцій

веб-застосування на додаткові рівні призводить до збільшення загальної потужності системи (за рахунок появи нових компонентів). [2]

На останньому рівні зазвичай знаходиться доступ до даних. Сюди інформація надходить від попередніх рівнів та зберігається у файловій системі. [2] Збереження та доступ до інформації може здійснюватися як в одній базі даних, так і розподілятися між декількома залежно від потреб та необхідності масштабування системи.

1.2 Проблема доступності веб-застосувань

У загальному випадку доступність веб-застосування розглядають як властивість, що характеризує, скільки часу відносно всього періоду роботи воно працює справо та користувачі мають до нього доступ. Якщо говорити більш конкретно – то доступність сайту в даний момент часу означає, що відкривши його, контент завантажується так, як це очікується. [11] Доступність у якості числової характеристики обчислюється за формулою (1.1).

$$A = \frac{T - t_d}{T} = \frac{t_a}{T}, \tag{1.1}$$

де A – доступність веб-застосування;

A – час вимірювання доступності.

t_d – час, який сервіс був недоступний в межах проміжку вимірювання;

t_a – час, який сервіс був доступний в межах проміжку вимірювання.

В результаті отримуємо відсоток, що характеризує те, скільки часу сайт є доступним та працює правильно. Системні адміністратори використовують цей показник, щоб вимірювати, як довго система працює без збоїв. [11]

Доступність – це вкрай важлива характеристика будь-якого веб-застосування, особливо, якщо веб-застосування виконує важливу державу, соціальну чи комерційну функцію. Якщо сервіс тим чи іншим чином пов’язаний з бізнесом та фінансами, то кожна хвилина проблем з доступністю до ресурсу призводить до збільшення збитків. Нестабільна робота веб-сервісу може спонукати клієнтів шукати аналоги серед конкурентів, тому важливо піклуватися про якісний досвід користувача, аби не втратити клієнтів. Через підвищення стабільності та швидкості інтернет підключення, а також збільшення кількості пристроїв для виходу в мережу Інтернет, у користувачів також збільшуються вимоги до онлайн сервісів, якими вони користуються, зокрема до їх доступності. Про це свідчать результати опитувань від “Digital”, за якими 39% респондентів вважають, що сайт має завантажуватися від 2 до 3 с, а 14% вважають затримку при відкриванні веб-сторінки більше 1 с неприпустимою. [12]

Окрім впливу доступності безпосередньо на досвід користувачів та продажі, важливо зазначити, що факт того, наявність збоїв в роботі веб-сервісу може негативно вплинути на його позицію у пошуковій видачі при пошуку у Google за ключовими словами. Як наслідок, це призводить до ще більшого репутаційного удару, і крім невдоволеності клієнтів, можна стикнутися з проблемою просування сервісу у пошукових системах.

Але важливо не тільки рахувати чисельне значення доступності, а й мати інтерпретацію того, яке значення доступності є достатнім. Доступність 99,9% зазвичай вважається галузевим стандартом та загальновизнаним мінімумом, якого мають досягати якісні веб-сервіси. [11] На перший погляд, ідеальним значенням доступності є 100%, адже такий показник доступності означає, що сайт працює без збоїв на протязі всього циклу роботи та будь-який користувач може відкрити його у будь-який момент часу. Однак в реальності таке значення хоч і є бажаним, але не є реалістичним, адже на доступність впливає велика кількість зовнішніх факторів,

що не залежить ані від самого веб-сервісу, ані від хостингу, на якому він знаходиться. До того ж, для звичайного користувача немає різниці, чи сервіс не відкривається через проблеми з боку клієнта, чи з боку сервера. У будь-якому випадку помилка при завантаженні буде розглядатися як проблема з доступом до сайту.

Важливо розуміти фактори, що можуть погіршувати доступність веб-сервісу, щоб відрізняти, від яких розробники можуть захиститися, на які можуть вплинути, а які не залежать від них. Серед основних чинників, що можуть впливати на доступність сайту в мережі Інтернет можна виділити такі:

a) Відключення електроенергії;

b) Помилка на сервері;

c) Спливає термін дії доменного імені;

d) Людський фактор;

e) Помилка в коді застосунку;

f) Недостатньо потужний хостинг, що не може впоратися з трафіком;

g) Сплеск трафіку (різке короткочасне збільшення кількості користувачів);

h) DoS атака (зловмисний напад на сервер або хостинг застосування з метою зашкодити роботі сервісу).

Тож постає проблема боротьби з цими проблемами та мінімізації їх наслідків. Для зменшення ризику збоїв в роботі сервісу у разі перебої з електропостачанням, зазвичай використовують джерела безперебійного живлення та резервні генератори електроенергії для серверів та мережевого обладнання. Розробник ніяк не може вплинути на захищеність сервісу від таких проблем, тому відповідальність за це лягає на хостинг, де розміщено сервіс. Людський фактор може вплинути на роботу сервісу на будь-якому з етапів налаштування сервісу – від помилкового налаштування мережевого обладнання на хостингу, до проблем в коді застосунку, що виникли через неуважність або недостатній досвід розробників.

Останні три пункти з переліку вище стосуються безпосередньо архітектури веб-застосування та ресурсів, що виділяються хостингом для коректної роботи системи під високим навантаженням. Для роботи зі змінним трафіком, якщо для веб-сервісу характерними є тимчасові сплески кількості одночасних користувачів, важливо попіклуватися про можливість налаштування змінного навантаження. Звичайно, як альтернатива, можна встановити фіксовану високу кількість використовуваних хостингом ресурсів, але такий підхід може виявитися досить дорогим і в результаті сервіс буде не таким прибутковим. До того ж, сервіс з сплесками трафіку не буде використовувати всі ресурси на протязі всього періоду функціонування, що робить такий підхід нераціональним.

Таким чином, виникає проблема балансування навантаження. Зазвичай цю функцію виконує проміжний мережевий пристрій (Load Balancer), що здійснює маршрутизацію запитів та відповідей між клієнтами та серверами. Load Balancer розподіляє запити між серверами, зважаючи на інформацію про їх потужність та кількість запитів в обробці таким чином, щоб мінімізувати ризик збоїв. Але якщо на одному з серверів все ж таки відбувся збій, і він з тих чи інших причин не відповідає, Load Balancer має тимчасово виключити його зі списку серверів, між якими він розподіляє навантаження.

Якщо ж мова йде про забезпечення тривалої роботи сервісу в умовах можливих веб-атак, важливо подбати про захист свого сайту заходами безпеки, що включають:

a) Використанні захисних екранів, що забезпечують захист від втручання в роботу сайту, завантаження шкідливого контенту та DoS-атак;

b) Регулярне сканування та видалення вірусів, оновлення бази даних шкідливого програмного забезпечення;

c) Двофакторна аутентифікація;

d) Посилений захист від DDoS-атак;

е) Моніторинг та аналіз мережевого трафіку.

Також, окрім превентивних заходів, варто подумати про зміцнення самого застосування, зробивши його більш стійким до збоїв. Як варіант, можна застосувати мікросервісну архітектуру для забезпечення горизонтального масштабування. Таким чином застосування може бути розподілено між кількома серверами. Кожен з них може або відповідати за певну частину функціоналу веб-застосування або ж дублювати функціонал іншого сервера. Таким чином навіть, якщо через різкий наплив користувачів або кібер-атаку один із серверів вийде з ладу, це призведе або до сповільнення роботи сервісу або до тимчасового збою в частині функціоналу, але застосування може продовжувати роботу, не зважаючи на технічні складнощі.

Хмарні платформи можуть вирішувати одночасно декілька проблем, пов’язаних з доступністю, тим самим дозволяючи розробникам сфокусуватися на програмуванні безпосередньо самого веб-застосування. Такі сервіси як Google Cloud Platform, Amazon Web Services та Microsoft Azure пропонують засоби для балансування навантаження, динамічного виділення ресурсів, захисту від зловмисних дій хакерів та багато інших компонентів, що спрощують процес розміщення та управління веб-сервісами з високою доступністю.

1.3 Хмарні платформи

Хмарні обчислення — це загальний термін, яким позначають модель надання послуг, пов’язаних з комп’ютерними обчисленнями, через мережу. Серед цих послуг можуть бути, наприклад, доступ до серверів, баз даних, певних програм, аналітичних застосунків та засобів штучного інтелекту. [3] Існує безліч хмарних сервісів з різноманітним призначенням, що спрощують роботу для багатьох видів діяльності, зокрема, для бізнесу. Саме тому хмарні рішення стають все більш популярними як серед звичайних користувачів, так і серед підприємців, яких

приваблює низька ціна, продуктивність, швидкість, ефективність, надійність, безпечність та зручність таких засобів. [4]

Кінцевим користувачам не потрібно піклуватися про побудову інфраструктури, забезпечення фізичного обладнання, налаштування операційних систем та програм. Таким чином компанії уникають надмірних витрат та складнощів обслуговування власної IT-інфраструктури, а замість цього платять за хмарні сервіси та налаштовують їх під власні потреби. [3] Клієнти лише користуються хмарними ресурсами, що надаються як мережеві сервіси, таким чином це дозволяє їм працювати з ними віддалено.

Хмарні послуги можуть бути як загальнодоступними, так і приватними — доступ до публічних хмар надається будь-яким бажаючим онлайн за певну плату, а приватні послуги розміщуються в закритій мережі для обмеженого кола користувачів. [4]

Публічна хмара — це модель хмарних обчислень, за якої інфраструктура сервісу призначена для вільного використання у відкритому доступі. Користувачі можуть отримати доступ до великого об’єму обчислювальної потужності через Інтернет. Однією із значущих переваг такого підходу є можливість швидкого масштабування послуги. Постачальники хмарних обчислень мають величезну кількість обчислювальної потужності, яку вони розподіляють між великою кількістю клієнтів залежно від навантаження. У таких хмар достатньо обчислювальної потужності, щоб вони могли легко впоратися з ситуацією, якщо будь-якому конкретному клієнту знадобиться більше ресурсів. Саме тому їх часто використовують для додатків, які вимагають змінної кількості необхідних ресурсів. [3]

Приватні хмарні послуги надаються внутрішнім користувачам із дата-центру компанії. За допомогою приватної хмари організація створює та підтримує власну базову хмарну інфраструктуру. Ця модель пропонує універсальність і зручність

хмари, зберігаючи при цьому управління, контроль і безпеку як у локальних центрів обробки даних. [5] Однак такий спосіб організації хмарного сервісу є досить фінансово затратним та складним, не кожна компанія може його собі дозволити. Якщо безпека даних та контроль над ними, що забезпечуються приватним мережевим екраном компанії, є пріоритетними, то створення приватної хмари може бути цілком виправданим.

Окрім поділу хмар залежно від способу розміщення та поширення, хмарні обчислення поділяють за моделями надання послуг на:

a) Software as a service (SaaS). Це модель розповсюдження, за якої програмні застосунки публікуються в мережі у вигляді веб-сервісів. Користувачі можуть отримати доступ до програм і служб SaaS з будь-якого місця за допомогою комп'ютера або мобільного пристрою з доступом до Інтернету. У моделі SaaS користувачі отримують доступ до прикладного програмного забезпечення та баз даних. [5] Одними з найпоширеніших програмних продуктів, що розповсюджуються як SaaS є, наприклад, Google Drive, Microsoft Office 365 та Adobe Creative Cloud. Всі ці сервіси поєднує те, що їх клієнти використовують ці веб-сервіси як звичайні програми, користуючись всіма перевагами хмарних обчислень, однак їх контроль над цими додатками та власними даними є досить обмеженим.

b) Infrastructure as a Service (IaaS). Постачальники IaaS надають екземпляри віртуального сервера та сховище, а також інтерфейси доступу до них та налаштування, які дозволяють користувачам переносити робочі навантаження на віртуальні машини. [5] Замість того, щоб орендувати, запускати та обслуговувати власні фізичні сервери, користувачі IaaS отримують доступ до віртуальних машин, які можна гнучко конфігурувати, налаштовувати виділений обсяг пам'яті та встановлювати мережеві налаштування. Такий спосіб розробки власних веб-

застосувань є дуже привабливим для компаній, що хочуть зекономити на побудові власної інфраструктури.

c) Platform as a Service (PaaS). На відміну від IaaS, модель PaaS передбачає не тільки доступ до хмарного сховища, мережевих налаштувань та віртуальних машин, а й засоби для розробки програмного забезпечення, що дозволяють будувати власні застосунки, користуючись готовою інфраструктурою. До цих засобів також належить проміжне програмне забезпечення, управління базами даних та різноманітні API для розробки веб-сервісів. [3] Таким чином PaaS має деякі спільні риси з IaaS, адже ця модель дозволяє користуватися готовою інфраструктурою та налаштовувати її, так і з SaaS, бо така хмарна платформа надає доступ до веб-сервісів, що в свою чергу дозволяють будувати власні застосунки та розміщувати їх в Інтернеті.

d) Безсерверні обчислення (Serverless computing). Це модель хмарних обчислень, що передбачає динамічне виділення ресурсів для виконання певної задачі лише, коли здійснюється запит. Хоча фактично, в таких обчисленнях все одно є сервери, але розробникам не потрібно піклуватися про них та можуть сприймати сервер як певну абстракцію. Ресурси виділяються залежно від навантаження, а коли до безсерверної функції ніхто не звертається, ці потужності зменшуються до нуля, тим самим зменшуючи вартість обслуговування. [6]

1.4 Google Cloud Platform

1.4.1 Загальний огляд платформи

Google Cloud Platform — це набір загальнодоступних хмарних сервісів, які пропонує Google, послуговуючись власною готовою інфраструктурою. Окрім ряду інструментів керування, він надає велику кількість модульних хмарних послуг, включаючи засоби для складних обчислень, роботи з великими даними, збереження

інформації, машинного навчання, керування хмарами та різноманітні інструменти для розробників. [7] Серед сервісів, що пропонує GCP є як IaaS, PaaS так і безсерверні обчислення. Серед усіх веб-сервісів, що надає хмарна платформа від Google, найбільш значущими є:

a) Google Compute Engine, Безпечний та гнучкий обчислювальний сервіс, який дозволяє створювати та запускати віртуальні машини в інфраструктурі Google та здійснення ресурсоємних обчислень; [8]

b) Google App Engine – це PaaS що надає розробникам програмного забезпечення доступ до масштабованого хостингу Google. Розробники також можуть використовувати комплекти розробки програмного забезпечення для створення застосувань, що працюватимуть в App Engine; [7]

c) Google Kubernetes Engine (GKE), що являє собою систему для управління та оркестрування Docker контейнерів та кластерів, що запускаються в межах загальнодоступного хмарного сервісу Google. Кластери GKE працюють на основі Kubernetes, системи керування кластерами з відкритим кодом. Можна використовувати команди та ресурси Kubernetes для розгортання програм і керування ними, адміністрування, встановлення політик та моніторингу працездатності розгорнутих робочих навантажень; [9]

d) Google Cloud Storage – хмарне сховище, призначене для зберігання великих неструктурованих наборів даних. В рамках цього сервісу Google також пропонує декілька варіантів зберігання бази даних, зокрема Cloud Datastore для нереляційного сховища NoSQL, Cloud SQL для повністю реляційного сховища MySQL і власну базу даних Cloud Bigtable від Google. [7] Таким чином клієнти GCP можуть обрати сховище, що буде відповідати їх потребам та масштабуватися залежно від навантаження;

e) Ряд засобів для безсерверних обчислень:

1) Cloud Run, що дозволяє запускати застосування в контейнерах та управляти ними, абстрагуючи управління інфраструктурою, автоматично збільшуючи та зменшуючи потужність майже миттєво, залежно від трафіку;

2) Cloud Functions – сервіс, що дозволяє створювати функції, що реагують на події в хмарі. Це дозволяє розв’язувати складі задачі з оркестрування, запускаючи код у декількох середовищах одночасно;

3) Сервіси для інтеграції, що дозволяють поєднувати та оркеструвати різні сервіси, в тому числі сторонні API. Google пропонує 4 засоби інтеграції: Eventarc (активує служби за допомогою подій із служб Google або безпосередньо з коду застосунку), Scheduler (створення розкладу для автоматизації одиниць роботи), Workflows (комбінування та оркестрування API), Tasks (керування виконанням, відправкою та отриманням великої кількості розподілених завдань). [10]

f) Операційний пакет Google Cloud (колишній Stackdriver), що являє собою набір інтегрованих інструментів для моніторингу, створення журналів та звітування про служби, які запускають програми в Google Cloud та управляють ними. [7]

Наданий вище перелік є лише частиною від усіх хмарних послуг, що надає GCP. Їх загальна кількість сягає більше сотні, при цьому бібліотека цих сервісів регулярно поповнюється новими засобами та технологіями, покликані спростити процес розробки веб-застосувань та задовольнити потреби бізнесу. Загалом всі послуги, що надає GCP можна поділити на такі основні категорії:

а) Штучний інтелект та машинне навчання. Цей напрямок стрімко розвивається, через що кількість таких засобів швидко зростає. До цього розділу належать як засоби для налаштування, тренування та тестування моделей машинного навчання, так і готові «розумні» API, наприклад, для розпізнавання природньої мови;

b) Управління API. У Google є велика кількість API, що можуть стати в нагоді для розробки веб-застосувань, такі як API для роботи з картами від Google Maps;

c) Хмарні обчислення – App Engine, Compute Engine, GKE, Cloud Functions, Cloud Run та інші;

d) Аналіз великих даних;

e) Засоби управління;

f) Хмарні сховища;

g) Інтернет речей;

h) Безпека та ідентифікація

i) Робота з мережею.

1.4.2 Масштабування Google App Engine

Google App Engine — це повністю керована безсерверна платформа для розробки та розміщення веб-додатків з гнучкими налаштуваннями масштабування. Перед початком роботи з цією платформою необхідно обрати один з двох варіантів середовища: стандартне середовище (standard) та гнучке середовище (flexible). App Engine підходить для розробки з використанням мікросервісної архітектури, тому в теорії можливо одночасно використовувати обидна типи для застосування, тоді одна частина сервісів буде у стандартному середовищі, інша – у гнучкому. [13]

Google App Engine впроваджує ряд запобіжників для того, щоб у разі збою одного з застосувань, це не вплинуло на продуктивність та доступність інших застосувань, опублікованих на платформі. App Engine знаходиться за Google Front End, що пом’якшує або відражає багато атак четвертого та нижчих мережевих рівнів, такі як SYN флуд, флуд IP фрагментами, штучна нестача портів та ін. Таким чином забезпечується високий рівень доступності опублікованих веб-застосувань. [24]

У стандартному середовищі програми запускаються в пісочниці (механізм для безпечного виконання неперевіреного коду, що обмежую доступ до системних ресурсів), використовуючи середовища однієї з обраних мов програмування: Python, Java, Node.js, PHP, Ruby, Go. Зазвичай такий тип середовища використовується для більшості веб-застосувань, у яких немає постійного значного трафіку, але можуть бути сплески кількості одночасних користувачів. Як вказано в технічній документації від Google, це середовище підходить для веб-застосування, якщо:

a) Воно написано на одній з конкретних версій мов програмування, що вказані на сайті Google Cloud для стандартного середовища;

b) Розраховується для безкоштовного або дуже дешевого запуску, коли власник застосування платить лише за потреби. Наприклад, кількість екземплярів запуску програми зменшується до 0, коли немає трафіку;

c) Спостерігаються різкі сплески трафіку, що вимагають негайного масштабування. [13]

У гнучкому середовищі програми запускаються в Docker контейнерах для віртуальних машин Compute Engine. Програми. Цей тип підходить для застосувань зі значним регулярним трафіком та частими, але здебільшого помірними коливаннями кількості одночасних користувачів. Відзначається, що гнучке середовище підходить для веб-застосування, якщо воно має такі характеристики:

a) Його код написано будь-якою версією підтримуваних мов;

b) Запускається в Docker контейнері, що включає код, написаний іншими мовами програмування;

c) Використовує залежності та фреймовки з нативним кодом;

d) Використовує ресурси та сервіси з мережі Compute Engine. [13]

Підсумовуючи, можна сказати, що стандартне оточення є більш простим, дешевим та підходить для застосувань без великого постійного трафіку. У той же час, у гнучкому середовищі менше обмежень до коду та більше налаштувань.

Незалежно від обраного типу середовища, основними будівельними блоками для App Engine є екземпляри, які забезпечують усі ресурси, необхідні для успішного розміщення програми. У будь-який момент часу програма може бути запущена на одному або кількох екземплярах, а запити розподіляються між ними. Кожен екземпляр має захисний механізм, що запобігає навмисному впливу екземплярів один на одного. [15] Ці екземпляри поділяються на два основних класи: F-екземпляр (Front-end екземпляр) та B-екземпляр (Back-end екземпляр). Для кожного з них є декілька варіантів, що відрізняються обсягом виділених ресурсів, безкоштовною квотою та ціною за годину.

App Engine може автоматично створювати та закривати екземпляри, коли трафік коливається, або можна вказати цю кількість вручну. Щоб визначити, як, коли та скільки нових екземплярів створювати, треба вказати відповідні налаштування масштабування у файлі app.yaml. [15] Всього є три типи масштабування:

а) Автоматичне. Екземпляри створюються та зупиняються залежно від кількості запитів та продуктивності їх обробки (наскільки швидко відправляється відповідь). [14] Для екземплярів з автоматичних масштабуванням є ряд конфігурацій, що допоможуть налаштувати його під потреби застосування:

1) min_instances – мінімальна кількість екземплярів для сервісу;

2) max_instances – максимальна кількість екземплярів для сервісу;

3) min_pending_latency – мінімальна кількість часу, протягом якого App Engine має дозволити запиту чекати в черзі на розгляд, перш ніж запустити новий екземпляр для його обробки;

4) max_pending_latency – максимальна кількість часу, протягом якого App Engine повинен дозволяти запиту чекати в черзі на розгляд, перш ніж запускати додаткові екземпляри для обробки запитів, щоб зменшити затримку в очікуванні;

5) target_cpu_utilization – цільове використання ресурсів процесора у відсотках (За замовченням 0.6). Значення використання ЦП усереднено для всіх запущених екземплярів і використовується, щоб вирішити, коли зменшити чи збільшити кількість екземплярів;

6) max_concurrent_requests – кількість одночасних запитів в черзі для одного екземпляра.

b) Базове. Базове масштабування створює екземпляри, коли програма отримує запити. Кожен екземпляр буде закрито, коли програма стане неактивною. Базове масштабування ідеально підходить для роботи, яка є переривчастою або обумовлена діяльністю користувача. [15] Для екземплярів з базовим масштабуванням є такі конфігурації:

1) max_instances – максимальна кількість екземплярів;

2) idle_timout – час простою, після його екземпляр вимикається.

c) Ручне. Ручне масштабування визначає кількість екземплярів, які безперервно працюють незалежно від рівня навантаження. Це дозволяє виконувати такі завдання, які залежать від стану пам’яті з часом. [15] В конфігурації для ручних екземплярів є лише значення instances, що вказує незамінну кількість екземплярів.

1.4.3 Балансування навантаження

Cloud Load Balancing – це розподілений програмний сервіс, що здійснює поділ та перенаправлення трафіку між багатьма екземплярами застосунку. Балансування не є апаратним, тому користувачам не потрібно налаштовувати його

фізичну інфраструктуру. Воно підтримує більше мільйона запити на секунду з високою продуктивністю та низькою затримкою. [16] Балансування з Google Cloud має такі особливості:

a) Єдина anycast IP-адреса. Завдяки Cloud Load Balancing одна IP-адреса є інтерфейсом для всіх інших серверних екземплярів з усіх регіонів по всьому світу. Таким чином забезпечується міжрегіональне балансування навантаження, включаючи автоматичне перемикання серверів між регіонами, що перенаправляє трафік у разі, якщо цільовий сервер стає непрацездатним. Cloud Load Balancing реагує миттєво на зміни в мережі, трафіку, стану серверів та будь-яких інших пов’язаних параметрів;

b) Програмно-визначене балансування. Цей сервіс не заснований на екземплярах або окремих фізичних пристроях, тому такий підхід не має проблем з налаштуванням фізичної інфраструктури або управлінням екземплярами;

c) Автоматичне масштабування, що дозволяє збільшувати кількість виділених ресурсів при зростанні трафіку. Різкі сплески кількісті користувачів опрацьовуються, поділяючи трафік між серверами в регіонах, що можуть прийняти цей трафік;

d) Балансування 4-го та 7-го рівня. Балансування трафіку на транспортному рівні відповідно до даних та протоколів транспортного рівня, таких як TCP, UDP, ESP, ICMP та ICMPv6. Балансування трафіку на прикладному рівні виконує розподілення трафіку, спираючись на певні атрибути, такі як заголовки HTTP запитів та URI;

e) Зовнішнє та внутрішнє балансування. Зовнішнє балансування здійснюється, коли користувачі звертаються до веб-застосувань з Інтернету, а внутрішнє – коли клієнти знаходяться в Google Cloud;

f) Глобальне та регіональне балансування. Глобальне балансування означає розподілення трафіку між поєднаними серверними ресурсами зі різних

географічних регіонів, поділяючи навантаження між ними рівномірно. Регіональне балансування поділяє навантаження в межах набору серверів з того регіону, звідки приходить трафік;

g) Підтримка додаткових функцій. Балансування хмарного навантаження підтримує такі функції, як глобальне балансування навантаження IPv6, керування трафіком на основі IP-адреси, WebSockets, визначені користувачем заголовки запитів, пересилання протоколів для приватних віртуальних IP, інтеграція з Cloud CDN для доставки кешованого контенту та інтеграція з Google Cloud Armor для захисту інфраструктури від DDoS-атак. [16]

1.5 Огляд засобів забезпечення доступності інших хмарних платформ

1.5.1 Amazon Web Services

Amazon Web Services — це комплексна платформа хмарних обчислень, створена компанією Amazon, яка включає IaaS, PaaS і SaaS послуги. Серед служб та інструментів AWS є засоби використання хмарних обчислень, сховище баз даних, послуги доставки вмісту та ін. [25]

AWS було запущено в 2006 році з внутрішньої інфраструктури, створеної Amazon.com для роботи з роздрібними онлайн-операціями. AWS була однією з перших компаній, яка представила модель хмарних обчислень з оплатою по мірі використання, яка масштабується за потреби. [25]

Amazon надає доступ до своїх сервісів з десятків дата-центрів, розташованих у різних регіонах та зонах доступності по всьому світу. Регіон включає в себе набір зон доступності з близьким географічним розташуванням, що з’єднані швидкісними мережевими кабелями з мінімальною затримкою. У свою чергу, кожна зона доступності складається з ряду дата-центрів. Для роботи можна обрати одну або декілька зон доступності для забезпечення якісної роботи застосування

для цільової аудиторії. Користувач AWS може запускати віртуальні машини та дублювати інформацію в базах даних між різними зонами доступності для забезпечення надійної роботи інфраструктури, стійкої до збоїв окремих серверів або одного з дата-центрів.

Для масштабування розміщених у хмарі програм є Amazon EC2 Auto Scaling, за допомогою якого користувачі платформи можуть налаштувати автоматичний запуск та зупинку екземплярів у відповідності до потреб веб-застосувань. Окрім звичайного розподілення ресурсів в реальному часі, цей сервіс пропонує прогнозоване масштабування, що дозволяє попередити різкі зміни кількості запитів до сервісу. Прогнози формуються з використанням машинного навчання на основні історії зміни навантаження на веб-застосування та навчається на нових даних. [25]

Для захисту веб-застосувань від мережевих DDoS атак Amazon пропонує AWS Shield. Він захищає програми, що працюють у хмарі, забезпечуючи безперервний моніторинг та автоматичну нейтралізацію атак. Таким чином зменшується ризик збоїв та час затримки для користувачів застосування. Усього передбачено два типи захисту:

a) Amazon Shield Standard, що надається всім клієнтам безкоштовно. Він надає базовий захист від типових та частих DDoS така мережевого та транспортного рівнів;

b) Amazon Shield Advanced поширюється за підпискою та надає захист більш високого рівня, забезпечує засоби виявлення та нейтралізації складних широкомасштабних DDoS-атак, видимість атак у режимі, близькому до реального часу, та інтеграцію з брандмауером інтернет-додатків AWS WAF. [26]

1.5.2 Microsoft Azure

Microsoft Azure — це загальнодоступна хмарна платформа з більш ніж 200 продуктами та послугами. Azure керує та обслуговує обладнання, інфраструктуру та ресурси, доступ до яких можна отримати безкоштовно або за принципом оплати при використанні. З моменту свого створення в 2008 році Microsoft Azure виросла і стала другою за величиною з трьох найбільших загальнодоступних хмарних платформ. За даними Statista, станом на другий квартал 2021 року ринок хмарних послуг розподілений наступним чином:

a) AWS – 31 % ринку;

b) Microsoft Azure – 22% ринку;

c) GCP – 8% ринку. [27]

Для масштабування у Azure передбачено функцію AutoScale, вбудовану в Cloud Services, Mobile Services, Virtual Machine Scale Sets, та Websites, яка автоматично коригує виділені ресурси на основі трафіку користувачів. Масштабувати можна як за ресурсами процесора, так і за обсягом оперативної пам’яті. Також є засіб для планування навантаження, що дозволяє створювати календарний план, що збільшує або зменшує об’єм використовуваних ресурсів залежно від дня та часу.

Для захисту від DDoS атак у Azure є Azure DDoS Protection Standard. Ця служба здійснює аналітику загроз та автоматично знаходить і усуває різні DDoS атаки. Ця функція очищає трафік на рівні мережі до того, як він зможе вплинути на роботу застосування. На відміну від AWS, служба мережевого захисту від Microsoft не має безкоштовного варіанту.

2 РОЗРОБКА ВЕБ-ЗАСТОСУВАННЯ ДЛЯ БІБЛІОТЕКИ З ВИКОРИСТАННЯМ GCP

2.1 Функціональні вимоги

Основна мета розробки – створення онлайн системи для роботи бібліотеки, що дозволить користувачам замовляти книжки, а працівникам бібліотеки – вести облік книжок та здійснювати обробку замовлень користувачів. Така система має надати можливість користувачам переглядати каталог бібліотеки, перевірити наявність шуканих книжок та дистанційно замовити їх. Це дозволить працівникам заздалегідь взяти відповідні примірники та підготувати їх до видачі замовнику. Все більше бібліотек використовують власні онлайн системи для замовлення книжок, що в свою чергу сприяє збільшенню кількості користувачів, адже все більше людей надає перевагу онлайн сервісам, що спрощують взаємодію з тими чи іншими установами.

Таким чином, можна вирізнити два основних типи користувачів системи: звичайний користувач та працівник бібліотеки (адміністратор). З цього випливає, що в системі обов’язково має бути авторизація та аутентифікація. Відповідно до своєї ролі, кожен користувач отримує різні дозволи для користування частинами системи:

а) Звичайний користувач може:

1) Переглядати список книжок, що є в бібліотеці. Кожен елемент списку містить інформацію про книжку, таку як: назва, автор, видавець, категорії та короткий опис. Список має бути розділений на сторінки для зручного перегляду каталогу. Також має бути спосіб пошуку потрібної книжки серед переліку за назвою та ім’ям автора. Для кожного елементу списку має бути

можливість додати його до «корзини», в якій будуть зберігатися обрані користувачем книжки перед замовленням;

2) Переглядати та управляти «корзиною». Книжки, додані до «корзини» мають відображатися в спеціальному списку, в якому можна переглянути інформацію про них, видалити зайві елементи з «корзини»;

3) Здійснювати замовлення та переглядати їх статуси. Замовлення формується за списком книжок у корзині та вказаною кількістю днів до повернення книжок в бібліотеку (мінімальна кількість днів 3, максимальна 180). Після цього замовлення отримає статус «В обробці» та потрапляє до історії замовлень, в якій користувач може переглянути попередні замовлення. Для кожного замовлення мають відображатися книжки, що входять до нього, спеціальний номер замовлення та його статус. Якщо замовлення отримало статус «Готово до видачі», користувач може вирушити до бібліотеки, щоб забрати книжки. Після цього статус змінюється на «Видано». Коли користувач повертає книжки до бібліотеки, статус замовлення змінюється на «Повернено».

b) Адміністратор може:

1) Переглядати, створювати та редагувати книжки. Адміністратор може переглядати список книжок аналогічно як і звичайний користувач, але у нього немає «корзини», щоб додавати туди книжки та створювати замовлення. Кожен з елементів списку можна редагувати, щоб змінити назву, автора, видавця, категорію, опис та чи є книжка приховано. Якщо книжку приховано, вона не буде відображатися у списку книжок для звичайного користувача, а от у адміністратора її видно з спеціальною позначкою. Так само можна створювати нові книжки, вказавши наведені параметри та додавши ISBN і кількість примірників. Після створення книжка додається до загального списку з вказаними значеннями;

2) Переглядати список замовлень від користувачів та змінювати їх статус. У кожному замовленні має відображатися його унікальний код, перелік книжок, що в нього входять та інформація про користувача, що його оформив. Замовлення, що мають статус «В обробці», можна змінити тільки на «Готово до видачі» або «Скасовано», якщо з певних причин виконати його неможливо (наприклад, примірники втрачено). Статус «Готово до видачі» можна змінити тільки на «Видано» або «Скасовано». Статус «Видано» можна змінити тільки на «Повернено». Також має бути пошук за унікальним ідентифікатором замовлення та статусом.

Для реєстрації облікового запису користувача бібліотеки необхідно надати повне ім’я, електронну пошту, адресу проживання, дату народження та пароль (від 8 до 40 символів). Для входу в систему треба ввести пароль та адресу електронної пошти, після чого користувач отримує доступ до системи з можливостями залежно від своєї ролі (адміністратор або звичайний користувач бібліотеки). Також має бути спосіб вийти з облікового запису.

2.2 Вибір набору технологій

Розробка багаторівневого веб-застосування передбачає поділ розробки на рівень представлення, рівень бізнес-логіки та рівень доступу до даних. Клієнтська частина має надавати механізми для роботи з каталогами книжок та замовлень. Для розробки якісного клієнтського застосування необхідно підібрати фреймворк, що надає можливість створювати швидкі, зручні та візуально привабливі графічні веб-інтерфейси. Серед найпопулярніших можна виокремити React, Angular та Vue. Варто порівняти їх та визначити сильні сторони та слабкості кожного з них для визначення, наскільки вони підходять для реалізації поставленої задачі.

Одним із ключових критеріїв під час вибору важливих технологій, що ляжуть в основу ваб-застосування, є спільнота. Те, наскільки ця спільнота велика та активна, визначає, наскільки легко можна знайти відповіді на ті чи інші питання, що виникають під час розробки. До того ж, якщо обраною технологією користується багато людей, серед них можуть знайтись розробники, що створюють власні доповнення до фреймворку або вдосконалювати існуючі модулі, що можуть стати в нагоді іншим. Відповідно до інформації, що надає «NPM trends» про кількість завантажень кожного з фреймворків, можна зробити висновок, що кількість завантажень через npm (менеджер пакунків для мови програмування JavaScript) у React значно перевищує показники найбільших конкурентів. На 5 червня 2022 року кількість завантажень React сягає 15754064, кількість завантажень Vue – 3345406, а у Angular – 3013403. Таким чином можна зробити висновок, що React завантажують приблизно у 5 разів частіше, ніж React чи Angular. Судячи з графіку на рисунку (2.1), кількість завантажень React має стійкий тренд до швидкого зростання.

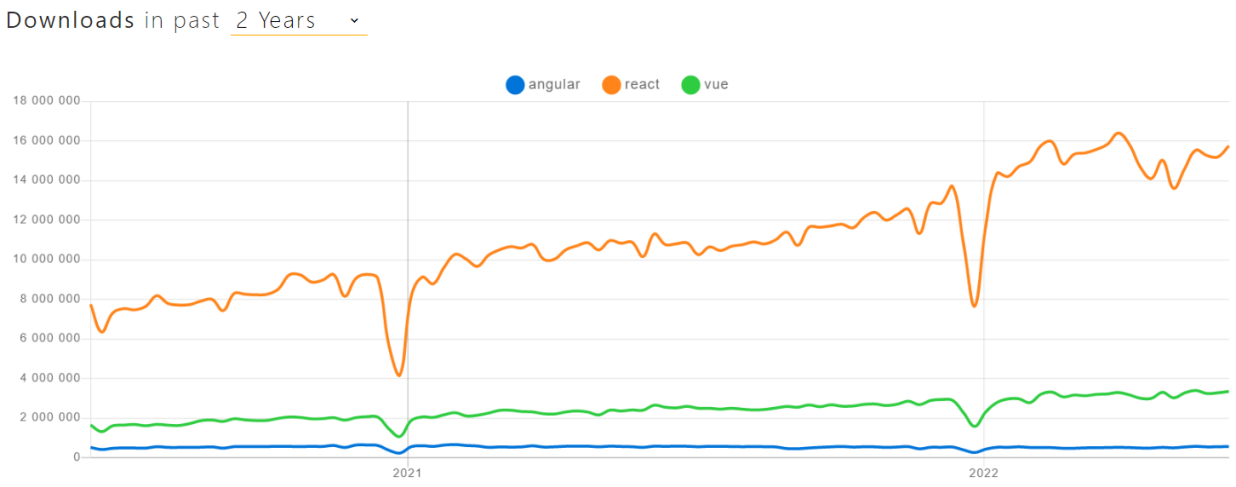

Рисунок 2.1 – Графік залежності кількості завантажень React, Angular та Vue від часу з «NPM trends» за 2 роки [17]

Однак кількість завантажень – не єдиний показник, що може свідчити про розмір спільноти та її активність. Важливо також порівняти те, наскільки часто користувачі пишуть про ці фреймворки, ставлять запитання про них на тематичних форумах та дають відповіді іншим. Саме тому досить показовим є порівняння, наскільки часто люди задають питання про той чи інший фреймворк на Stack Overflow. Судячи з графіку на рисунку (2.2), можна дійти висновку, що кількість згадувань react на форумі стрімко зростає кожного місяця та на 2022 рік значно перевищує показники найближчих конкурентів. Але варто зауважити, що певний час лідерство тримав Angular, і його популярність на Stack Overflow більша, ніж у Vue.

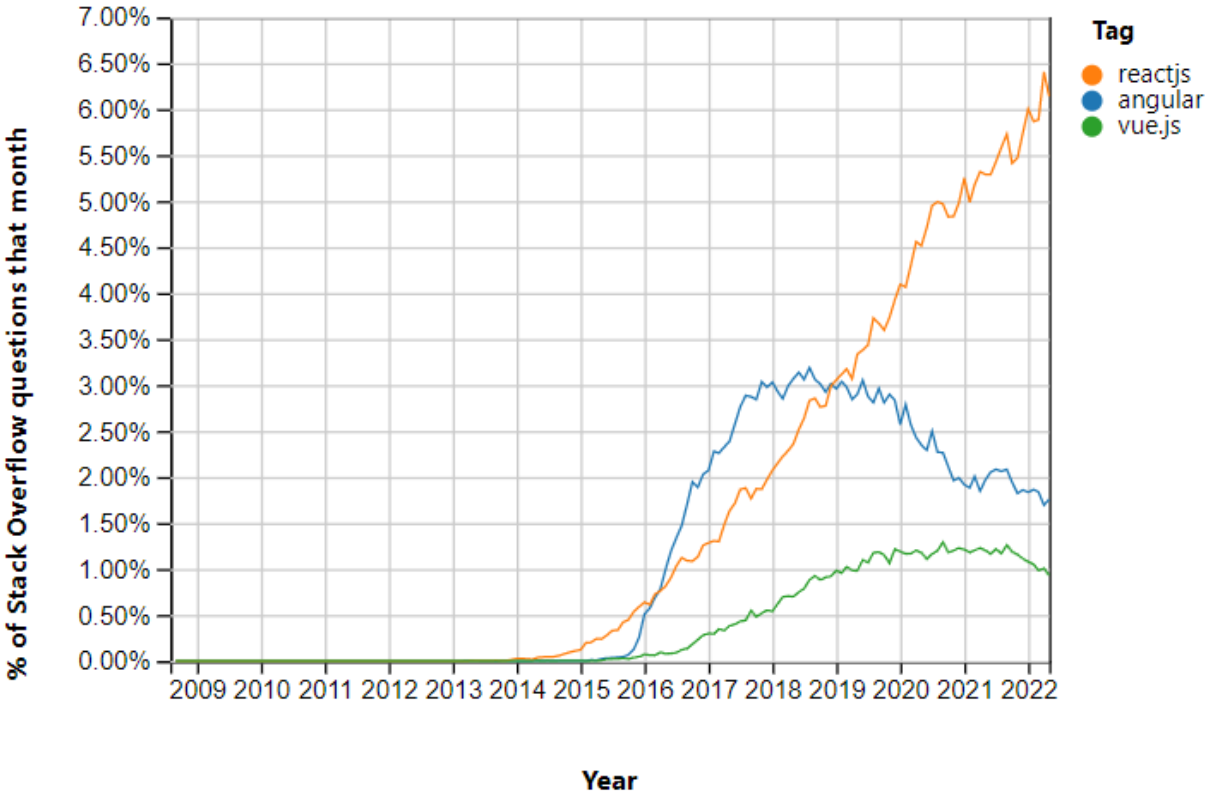

Рисунок 2.2 – Графік залежності відсотків питань про React, Angular та Vue від часу з Stack Overflow [18]

Отже можна зробити висновок, що спільнота React найактивніша та найчисленніша серед спільнот перелічених фреймворків. Це свідчить про його високу популярність та легкість в освоєнні. До того ж, якщо у фреймворку велика кількість користувачів, з цього випливає, що працюючи з ним, набагато легше знайти відповіді на різні запитання, існує більше готових перевірених практик та доповнення до нього розвиваються більш активно.

Наступною важливою властивістю шуканого фреймворку для розробки веб-застосувань є екосистема. Під цим поняттям розуміють те, наскільки широким є вибір модулів, готових компонентів та інших доповнень, що спрощують розробку та дозволяють розширювати функціонал застосування. Так як React по-суті являє собою не повноцінний фреймворк, а JavaScript бібліотеку з функціоналом фреймворку, він має найбільше сторонніх модулів. Однак варто зазначити, що Angular потребує меншої кількості залежностей, бо велика частина функціоналу вже є в основних модулях. Звернувшись до npmjs.com, можна перевірити, скільки JavaScript бібліотек мають у своїх залежностях кожен з фреймворків:

a) react – 88127;

b) vue – 56011;

c) @angular/core – 12599.

Отже можна зробити висновок, що React має найбільшу кількість сторонніх доповнень, що пояснюється величезною спільнотою, що розробляє їх. Але кількісний показник – не головне, адже найважливішим є те, чи є серед них дійсно потужні та корисні розширення, що доповнюють фреймворки якісними та перевіреними функціями. Наприклад, у React є Redux, бібліотека для керування станом, що дозволяє створювати більш надійні застосування. Вона активно розвивається, має широку підтримки та стійкі перевірені шаблони для використання. А для мобільної розробки є React Native, що дозволяє будувати мобільні застосунки, використовуючи спільний код, написаний з React.

Для керування інформацію про стан компонентів у Vue можна використовувати як Redux, так і офіційну бібліотеку, Vuex. Для розробки мобільних додатків є перспективний проект під назвою Weex. Варто зазначити, що він не настільки розвинений як React Native та має проблеми з англійською документацією, бо використовується та розробляється в Китаї.

У Angular є свій перевірений часом аналог Redux, NgRx. А для побудови мобільних застосунків можна використовувати NativeScript, що має високий рівень підтримки Angular. На відміну від Vue та React, великою перевагою Angular є наявність великої кількості готових компонентів у проекті Angular Material, розробленому Google для впровадження Material Design у Angular.

Таким чином, екосистема більш розвинена у React та Angular. React має велику підтримку спільноти, завдяки чому у нього найбільша кількість зовнішніх модулів, а Angular – розробляється Google, завдяки чому у нього є значна кількість потужних перевірених розширень та офіційних готових модулів.

Не менш важливою характеристикою для порівняння є продуктивність та швидкодія. Ефективність роботи веб-застосування дуже сильно залежить від того, як відбувається взаємодія з DOM, що представляє структуру веб-сторінки у браузері. Саме через маніпуляції з DOM здійснюється більша частина перетворень, оновлень та інших дій на сторінках. Angular використовує звичайний DOM, що призводить до того, що сторінки повністю перебудовуються навість, якщо змінюється лише один компонент. Це великий недолік даного фреймворку, особливо у випадку розробки SPA. У свою чергу, React та Vue використовують віртуальний DOM, що являє собою копію справжнього DOM, над якою здійснюються всі операції перетворення. Після внесення всіх необхідних змін віртуальний DOM порівнюється зі знімком справжнього, після чого на сторінці перебудовуються лише компоненти, шо змінилися. Такий підхід значно збільшує продуктивність та швидкодію веб-застосування. [19]

Таким чином, у якості клієнтського фреймворку для рівня представлення веб-застосування бібліотеки обрано React за його розвинену екосистему, велику спільноту та високу швидкодію. Другим не менш важливим аспектом перед початком розробки є вибір технологій для серверної частини застосування. Так як основною мовою для написання клієнтської частини є JavaScript, бажаним є використання цієї ж мови і для сервера. Використання єдиної мови для рівня представлення та рівня бізнес логіки значно спрошує роботу при розробці, так як можна застосувати подібні структури та функції, якщо в цьому виникає потреба. До того ж, при використанні спільної мови програмування для обох частин застосування, у розробника менше потреби у додатковому навчанні. Таким чином, Node.JS має велику перевагу над іншими технологіями для створення веб-застосувань.

Великою перевагою Node.JS над аналогами, такими як ASP.NET, JSP та PHP є неблокуючі ввід/вивід та асинхронна обробка запитів. Більшість серверних технологій послуговуються синхронним багатопоточним виконанням, що з одного боку дозволяє більш ефективно використовувати ресурси серверного пристрою, але з іншого боку такий підхід накладає деякі обмеження на програмний застосунок. Наприклад, кількість потоків обмежена, що потенційно зменшує максимальну пропускну здатність застосування. При великому навантаженні багатопотоковий веб-сервер споживає великий обсяг пам’яті, більшу частину часу потоки простоюють в очікуванні на завершення операцій введення/виведення перед обробкою наступних запитів. У Node.JS використовується один потік, але завдяки асинхронній роботи програми, здійснення запитів відбувається без блокування цього потоку. Після того, як запущено функцію зворотного виклику, програма одразу продовжує обслуговування наступних запитів, що забезпечує високу продуктивність та масштабованість такого застосування. Заснована на подіях і неблокуюча природа Node.js дозволяє створювати необмежену кількість модулів,

об’єднуючи їх разом, забезпечуючи вертикальну та горизонтальну масштабованість. [20]

Наступною значною перевагою Node.JS є велика та надійна екосистема. Для цього фреймворку існує велика кількість бібліотек та доповнень, що розширюють функціонал веб-сервера. Численна спільнота розроблює багато модулів з відкритим кодом, завдяки чому легко знайти інструменти для певних випадків використання. Розробники створюють нові та вдосконалюють існуючі модулі, тим самим збагачуючи арсенал доступних засобів Node.JS. Основний менеджер пакетів Node.js, npm, містить понад 800000 бібліотек і шаблонів для повторного використання, і це число продовжує зростати. [20]

Отже, Node.JS є чудовим вибором для застосувань, що потребують легкого масштабування. До того ж, цей фреймворк має перевагу над іншими, адже використовує JavaScript, що спрощує роботу розробника, що може використовувати одну мову для серверної та клієнтської частини застосування. А велика кількість доповнень та бібліотек, що вдосконалюються активною спільнотою, дозволяє впроваджувати в проект різноманітні нові технології.

Не менш важливим пунктом є вибір технологій для рівня даних. GCP пропонує Cloud Firestore у якості інструменту для збереження та роботи з даними. Це хмарна NoSQL база даних, що забезпечує автоматичне масштабування, високу продуктивність, простоту у використанні, а також надає високий рівень надійності. Ця технологія безсерверна, завдяки чому збільшення та зменшення обчислювальних потужностей відбувається швидко та тільки за потреби, без простою або переривання для технічного обслуговування. Google пропонує значні безкоштовні квоти для використання цієї технології. У безкоштовний щоденний пакети входять : 50000 зчитувань, 20000 записів, 20000 видалень та 1 гігабайт сховища. [21]

Так як Firestore засновано на NoSQL, дані зберігаються у вигляді документів з доступом в реальному часі, що в свою чергу надає можливість створювати та змінювати структури даних динамічно. Таким чином, NoSQL краще підходить для не дуже складних структур даних, яким властиві потенційні зміни, але такий підхід забезпечує значне масштабування в ширину, що дозволяє швидко працювати зі значним об’ємом інформації. У свою чергу SQL бази даних краще підходять для даних зі складною структурою, що потребують вертикального масштабування. Таким чином, NoSQL база даних Firestore підходить для веб-застосування бібліотеки, адже структура даних досить проста, однак об’єм інформації про книжки може бути значним.

Так як у Firestore не передбачено можливості для повнотекстового пошуку, необхідно застосувати сторонні API для здійснення відповідних операцій фільтрації. Для реалізації повнотекстового пошуку використано сервіс Algolia, що дозволяє індексувати документи, здійснювати текстовий пошук, фільтрацію та пагінацію результатів. Серед аналогічних сервісів великою перевагами Algolia є простота у використанні та безкоштовний варіант використання.

2.3 Структура хмарного веб-застосування

Для розроби веб-застосування бібліотеки на рівні представлення використано React, для серверного рівня бізнес-логіки – Node.JS, а для роботи з даними – Firestore та Algolia. Такий набір технологій дозволяє створювати швидкі та легко масштабовані веб-застосування. Для розміщення системи у хмарі, використано Google App Engine, що дозволяє створювати множину сервісів в межах одного застосування, налаштовувати масштабування для кожного з них та встановлювати засоби взаємодії між ними. У кожного сервісу можуть бути різні версії. Після кожного завантаження оновлення коду сервісу в хмару створюється нова версія, що

стає основною. Саме до останньої версії спрямовується весь трафік за замовченням, але є налаштування, що дозволяють перенаправляти його до однієї з попередніх версій або поділити його між декількома різними версіями. Кожна версія запускається у певній кількості екземплярів, що визначається автоматично або вручну відповідно до правил, вказаних у конфігурації даної версії сервісу.

Всього для веб-застосування бібліотеки у Google App Engine визначено два сервіси: сервіс представлення («Client») та сервіс бізнес-логіки («api»). Обидна сервіси використовують стандартне середовище, тому що для веб-застосування бібліотеки є характерним низький рівень середнього трафіку зі значними сплесками кількості користувачів на початку та наприкінці навчальних семестрів у закладах вищої та середньої освіти. До того ж такі сервіси можуть за потреби масштабуватися до 0, що значно зменшує вартість використання хмарного середовища.

Так як застосування розміщується за одним спільним доменом, запити поділяються між ними залежно від посилання. Якщо запит починається з «/api», він перенаправляється до сервісу «api», інакше він розглядатиметься сервісом «client», як видно на рисунку (2.3).

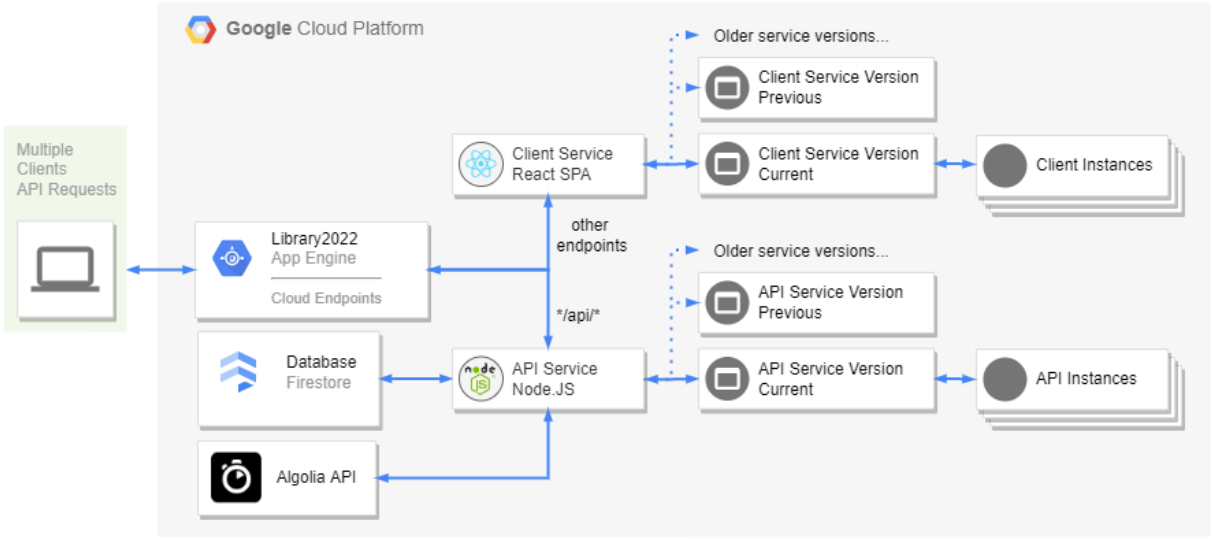

Рисунок 2.3 – Структура веб-застосування для бібліотеки в GCP

Налаштування сервісів «api» та «client» визначено у спеціальних конфігураційних файлах: api.yaml та client.yaml відповідно. Вони містять інформацію про код, змінні оточення та налаштування автоматичного масштабування, при цьому «client» встановлено як сервіс за замовченням, бо саме інтерфейс користувача має відкриватися при переході на сайт бібліотеки. Усі налаштування масштабування записано у параметрі automatic_scaling для кожного з сервісів:

a) Для сервісу «api» він має наступний вигляд:

1) target_cpu_utilization: 0.7;

2) max_instances: 1000;

3) max_concurrent_requests: 50;

b) Для сервісу «client» він має наступний вигляд:

1) target_cpu_utilization: 0.65;

2) max_instances: 100;

3) max_concurrent_requests: 25.

Для сервісу «api» максимальна кількість екземплярів та максимальна кількість запитів в черзі на обробку більші, ніж у «client». Це пояснюється тим, що на сервер бізнес логіки буде більше навантаження, бо користувач завантажує односторінковий веб-інтерфейс лише один раз, але створює багато запитів для роботи з книжками та замовленнями.

2.4 Сервіс бізнес-логіки

Сервіс бізнес-логіки («api») реалізовано за допомогою Rest API на Node.JS з підключенням до хмарної бази даних Firestore. Для спрощення роботи з багатьма компонентами та типами даних, було застосовано TypeScript, що додає до звичайного JavaScript явну типізацію. Усі серверні конфігурації та кінцеві точки

встановлено за допомогою фреймворку Express, що надає достатній набір функцій для мобільної та веб-розобки. Express дозволяє визначати таблицю маршрутизації запитів для виконання різних дій, залежно від HTTP методу та URL. Також за допомогою цього фреймворку можна легко налаштувати проміжні обробники HTTP запитів. Основна задача цієї частини системи – обробка запитів від сервісу представлення та взаємодія з хмарним сховищем для збереження, читання та оновлення інформації в ньому. Це серверне застосування приймає такі запити:

a) «/api/auth/signup» типу POST. За цим маршрутом здійснюється реєстрація нового користувача типу «USER». Для створення нового облікового запису необхідно надати: email, password, address (адресу проживання), dateOfBirth (дата народження) та fullName (повне ім’я). Пароль шифрується за допомогою криптографічної функції bcrypt, що убезпечує від атак з райдужними таблицями за допомогою солі (salt). Адреса електронної пошти має бути унікальною, тому у відповідь на запит з email, що вже є в базі даних, буде надіслано повідомлення про помилку;

b) «/api/auth/signin» типу POST. Цей маршрут відповідає з аутентифікацію користувача. Для входу в систему необхідно надати пароль та адресу електронної пошти. Якщо вхід у систему відбувся успішно, інформація про користувача відправляється клієнту разом з сесійним cookie. У цьому cookie зберігається JWT токен, в якому зашифровано роль користувача («USER» або «ADMIN») та його електронну пошту. Це cookie використовується в інших запитах для авторизації. Якщо користувача з вказаним email немає в базі або вказано невірний пароль, у відповідь надсилається відповідне повідомлення про помилку;

c) «/api/auth/signout» типу GET. Цей маршрут використовується для виходу з облікового запису. Сесійне cookie видаляється, у відповідь надсилається повідомлення про успішну операцію;

d) «/api/book/:isbn» типу GET. За цим маршрутом авторизований користувач може отримати доступ до інформації про книжку з вказаним ISBN. У разі, якщо така книга є в базі даних, у відповідь надходить: isbn, title, author, publisher, description, categories (перелік категорій), isHidden (чи є книга прихованою для звичайних користувачів в загальному переліку) та items (перелік екземплярів книги). Якщо книги з вказаним ISBN не існує, користувач отримає повідомлення про помилку. Доступ мають тільки «USER» та «ADMIN»;

e) «/api/book/» типу GET. Це запит для здійснення пошуку книжоку з пагінацією за вказаними параметрами. Серед параметрів є search (пошуковий запит за назвою та (або) ім'ям автора), startAt (порядковий номер початку пагінації), limit (кількість книжок на сторінці для пагінації), categories (перелік категорій книжок, серед яких здійснюється пошук). Для пошуку в колекції використовується Algolia. Доступ мають тільки «USER» та «ADMIN»;

f) «/api/book/:isbn» типу POST. Запит для оновлення інформації про книгу з вказаним ISBN. Підлягають оновленню наступні властивості: title, author, publisher, description, isHidden. Оновлюються тільки ті параметри, що вказано у запиті. Якщо книги з даним ISBN не існує, користувач отримає повідомлення про помилку. Доступ має тільки «ADMIN»;

g) «/api/book/» типу POST. За цим запитом здійснюється створення нової книги в базі. Для створення необхідно надати таку інформацію про книгу: isbn, title, author, publisher, description, categories, isHidden, itemCount (кількість екземплярів у бібліотеці). Якщо книга з вказаним ISBN вже існує, буде надіслано помилку. Доступ має тільки «ADMIN»;

h) «/api/order» типу GET. Цей запит відповідає за пошук замовлень для перегляду з пагінацією. Серед вхідних параметрів є startAt, limit, status (шуканий статус замовлень), search (пошуковий запит за ідентифікатором), userEmail (email

користувача, запити якого треба знайти). Властивості startAt, search та userEmail є необов’язковими. Доступ мають тільки «USER» та «ADMIN»;

і) «/api/order» типу POST. За цим запитом здійснюється створення нового замовлення. Цей запит приймає такі параметри: userFullName (повне ім’я користувача), userEmail (електронна пошта користувача), items (перелік екземплярів книжок), dateToReturn (очікувана дата повернення книги в бібліотеку). Кожен елемент переліку екземплярів має id (ідентифікатор цієї конкретної книги), isbn, author, title. Після створення замовлення йому надається статус «PENDING» та додається dateOrdered з поточним часом та датою. Доступ до оформлення замовлень має тільки «USER»;

j) «/api/order/:id» типу POST. Це запит для оновлення статусу замовлення з id у самому запиті та status у його тілі. Якщо статус змінюється на «RECEIVED», у нього автоматично змінюється значення dateReceived на поточні дату і час, що свідчить про отримання книг замовником. Якщо статус змінюється на «RETURNED» або «CANCELED», додається dateReturned з поточною датою і часом, що свідчить про завершення замовлення. Має доступ тільки «USER» та «ADMIN»;

Доступ до кінцевих точок для книг та замовлень надається тільки авторизованим користувачам залежно від ролі, тому перед передачі запитів для обробки до контролерів, вони проходять через проміжні обробники. Всього для сервісу бізнес-логіки передбачено два проміжних обробники:

а) authenticateJWT, що здійснює валідацію JWT токена, збереженого в сесійному cookie, та дешифрує роль та адресу електронної пошти, що зберігаються в ньому. Якщо операцію виконано успішно, запит передається наступним обробникам, інакше, якщо токен відсутній або некоректний, обробка запиту зупиняється, наступні обробники не виконуються, а у відповідь відправляється повідомлення про помилку;

b) roleCheck, що перевіряє, чи співпадає роль користувача, отримана з токена, з однією з ролей, необхідних для продовження обробки запиту. Якщо умова виконується, запит передається наступним обробникам, інакше обробка зупиняється та надсилається помилка.

Обробник roleCheck обов'язково має стояти тільки після authenticateJWT, щоб перевірити роль після валідації та дешифровки токену. За допомогою roleCheck обмежується доступ до певних кінцевих точок залежно від ролі поточного користувача. Таким чином за допомогою цих проміжних обробників забезпечується авторизація та аутентифікація, після чого запити передаються відповідними контролерам, що в свою чергу користуються внутрішніми сервісами для роботи з базою даних Firestore, як це зображено на рисунку (2.4).

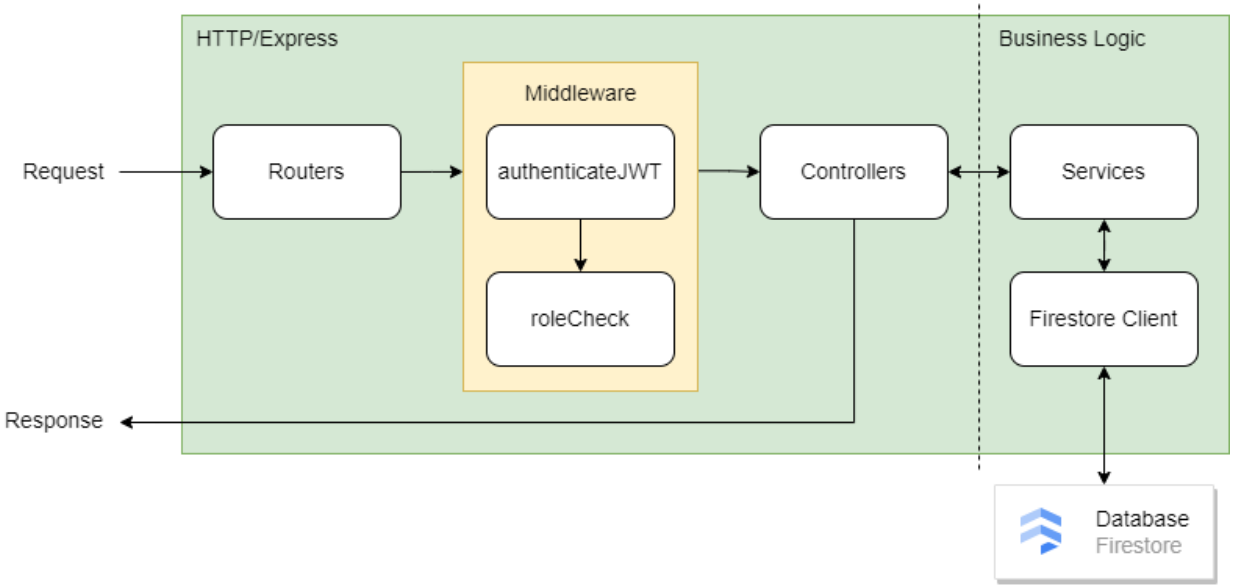

Рисунок 2.4 – Архітектура веб-сервісу бізнес-логіки

Така архітектура Rest API надає можливість зручно створювати нові кінцеві точки, додаючи відповідні контролери та сервіси, а проміжні обробники дозволяють убезпечувати їх від неавторизованого доступу.

2.5 Сервіс представлення даних

2.5.1 Архітектура клієнтського сервісу

Для клієнтської частини системи було розроблено односторінкове (SPA) React застосування. Односторінковий інтерфейс вміщується на одній сторінці веб-оглядача з метою забезпечити користувачам досвід близький до користування звичайно комп'ютерною програмою. Вся необхідна HTML розмітка, JavaScript код та CSS стилі завантажуються один раз із сторінкою, після чого динамічно змінюються та довантажують за потреби. Таке веб-застосування не здійснює перенаправлення до інших сторінок сайту та не викликає оновлення вкладки веб-оглядача.

Так як клієнтська частина системи виконана у формі ондосторінкового React застосування, це значно зменшує навантаження на сервіс «client», адже під час навігації по сайту користувач не надсилає запит для отримання кожної сторінки веб-застосування, а тільки під час завантаження та ручного оновлення сторінки. Тому в плані кількості запитів навантаження на сервіс «client» буде менше, ніж на «api», але в плані об'єму трафіку ситуація протилежна, бо клієнт має отримати одразу всю розмітку та код.

Як і сервіс бізнес логіки, сервіс представлення даних використовує TypeScript для явної типізації, що спрощує використання компонентів React та взаємодію з сервером. Для реалізації візуального інтерфейсу використано фреймворк Material UI. Це бібліотека компонентів React з відкритим вихідним кодом, яка реалізує Material Design від Google. Вона включає в себе повну колекцію готових компонентів з широкими можливостями для налаштування та кастомізації: контейнери, кнопки, текстові поля та ін. [22] Також цей фреймворк включає іконки Material Icons від Google, перетворені у React компоненти, що спрощує їх впровадження. Використання компонентів з готовими стилями, анімаціями та

функціями дозволяє швидко будувати складні та візуально привабливі сучасні користувацькі інтерфейси, що слідують правилам та вимогам Material Design.

В основі архітектури клієнтського застосування лежить модель Redux для керування станами. Вона описує централізоване сховище станів, які потрібно використовувати в різних частинах програми, встановлюючи правила, які гарантують, що стан можна оновлювати лише в передбачуваний спосіб. Шаблони та інструменти, що надає Redux, полегшують розуміння того, коли, де, чому і як оновлюється глобальний стан у програмі, а також які дії виконуватиме програма, коли відбудуться ці зміни. [23] Redux має такі основні концепти:

a) Дії (actions). Описують подію, що сталася в програмі. Обов'язково має значення type, що описує тип події, але може містити й додаткову інфомрацію у payload;

b) Редуктори (reducers). Виконують роль слухачів подій, що оброблюють події на основі отриманого типу дії. Вони не виконують жодної асинхронної логіки, а тільки оновлюють значення поточного стану у незмінній копії (не змінює стан напряму);

c) Сховище (store). Це об'єкт, у якому зберігаються поточні стани;

d) Відправка (dispatch). Єдиний спосіб оновити стан – викликати функцію dispatch, передавши відповідний об'єкт дії;

e) Селектори (selectors). Дозволяють отримати частину інформації зі сховища.

У поєднанні з TypeScript такий підхід дозволяє гарантувати не лише наявність стану, а й типи даних, що зберігаються. Ця модель реалізується за допомогою бібліотек Redux Toolkit та React-Redux. Redux Toolkit містить пакети та функції, що реалізують найкращі практики Redux та спрощують реалізацію програм з використанням цієї моделі. Серед них є засоби для роботи з thunk, що дозволяють реалізовувати асинхронну логіку, встановлювати проміжні обробники, для

здійснення доступу до мережевих API. Таким чином архітектура роботи з даними у сервісі рівня представлення має вигляд як на рисунку (2.5).

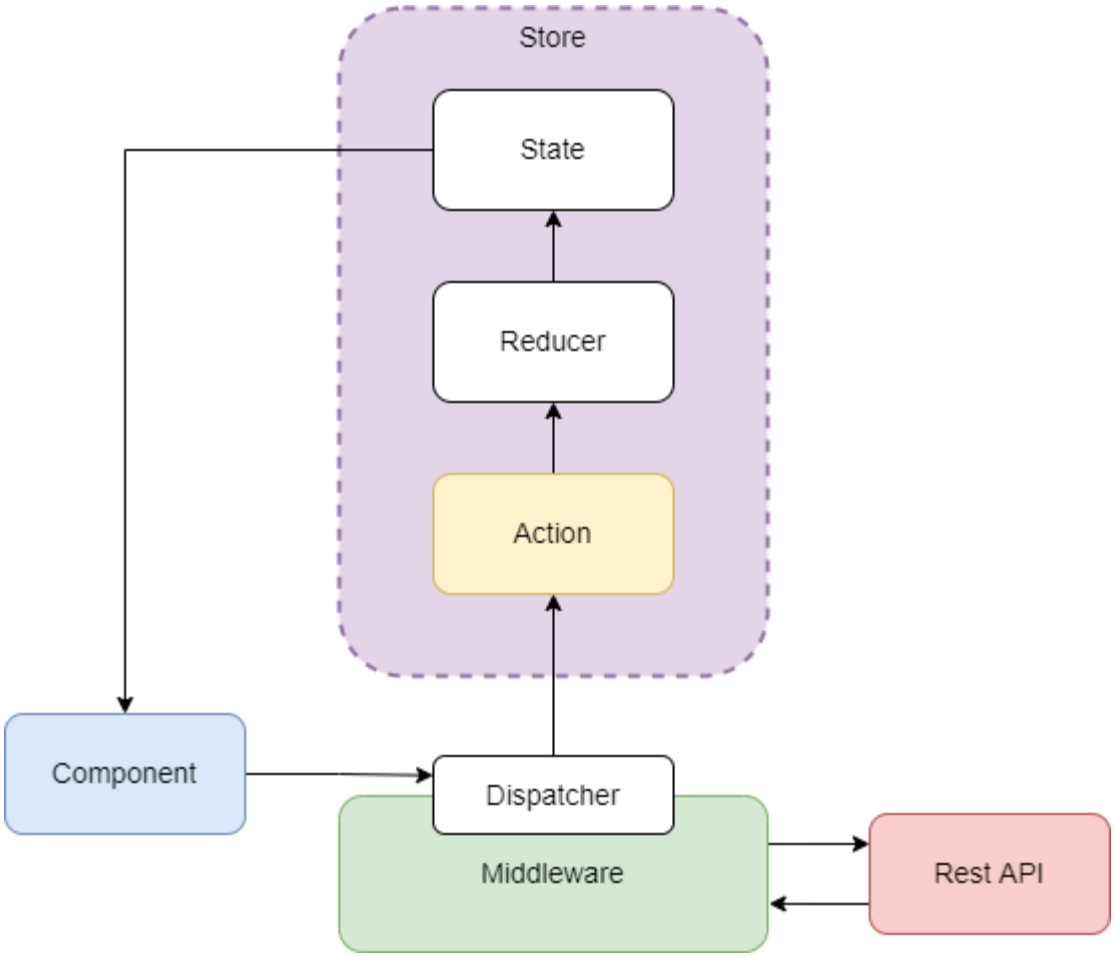

Рисунок 2.5 – Архітектура Redux в клієнтському сервісі

2.5.2 Структура користувацького інтерфейсу

Користувацький інтерфейс веб-застосування бібліотеки містить наступні сторінки:

a) Реєстрація (за маршрутом «/signup»);

b) Вхід (за маршрутом «/signin»);

c) Перелік книжок (за маршрутом «/book»);

d) Перелік замовлень (за маршрутом «/order»);

e) Кошик (за маршрутом «/cart»);

f) Створення книжки (за маршрутом «/book/create»);

g) Редагування книжки (за маршрутом «book/edit/:isbn» з ISBN відповідної книги).

Коли користувач заходить на сайт, у першу чергу він має увійти в систему для того, щоб користуватися онлайн-ресурсом бібліотеки (рисунок 2.6). Для входу в обліковий запис відвідувач має ввести адресу електронної пошти та пароль. У разі, якщо дані некоректні (пароль неправильний або користувача з таким email немає в базі), то користувач отримає повідомлення про помилку, а якщо все вірно – відкриється сторінка з каталогом книжок.

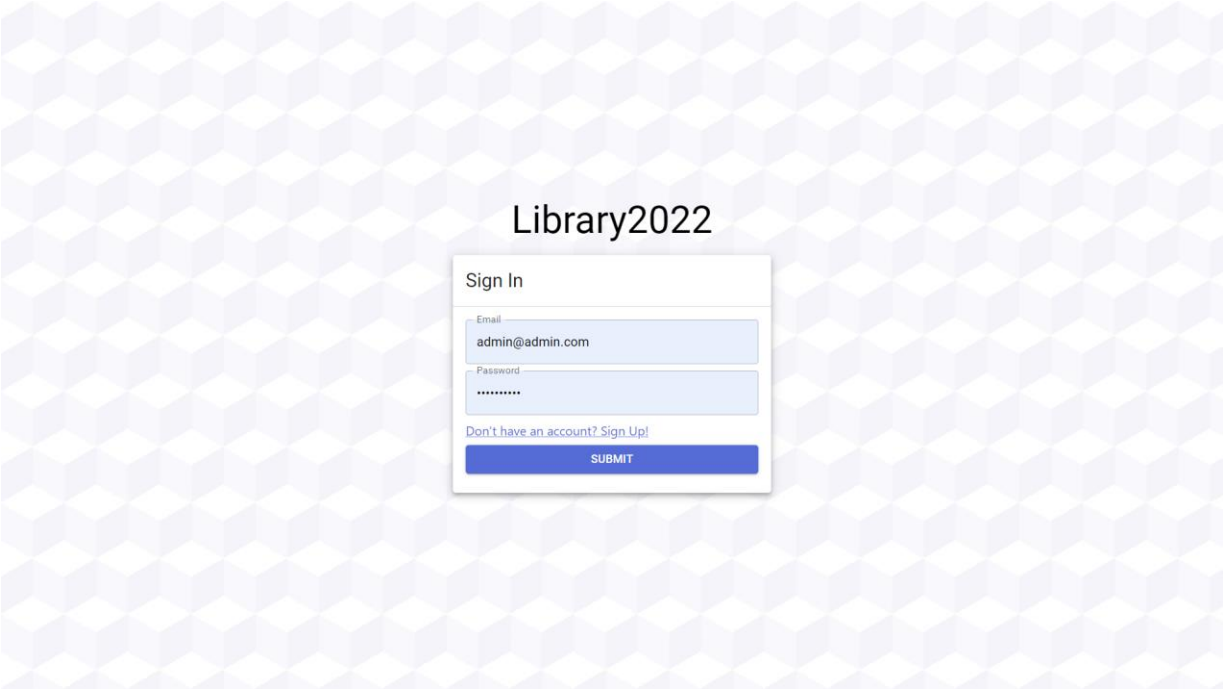

Рисунок 2.6 – Сторінка входу

Якщо у відвідувача немає облікового запису, він може створити новий, перейшовши за посиланням на сторінку реєстрації (рисунок 2.7). На сторінці

реєстрації користувач має надати своє повне ім’я (мінімум 2 символи), коректну адресу електронної пошти, дату народження (що не може бути в майбутньому), адресу проживання (мінімум 4 символи), пароль (від 8 до 40 символів) та підтвердження пароль (що має співпадати з введеним паролем). Для перевірки коректності значень у всіх формах використовується бібліотека Yup. Якщо користувач спробує надіслати некоректні дані, він отримає помилку, а якщо все введено вірно, і обліковий запис створено, він потрапить на сторінку входу.

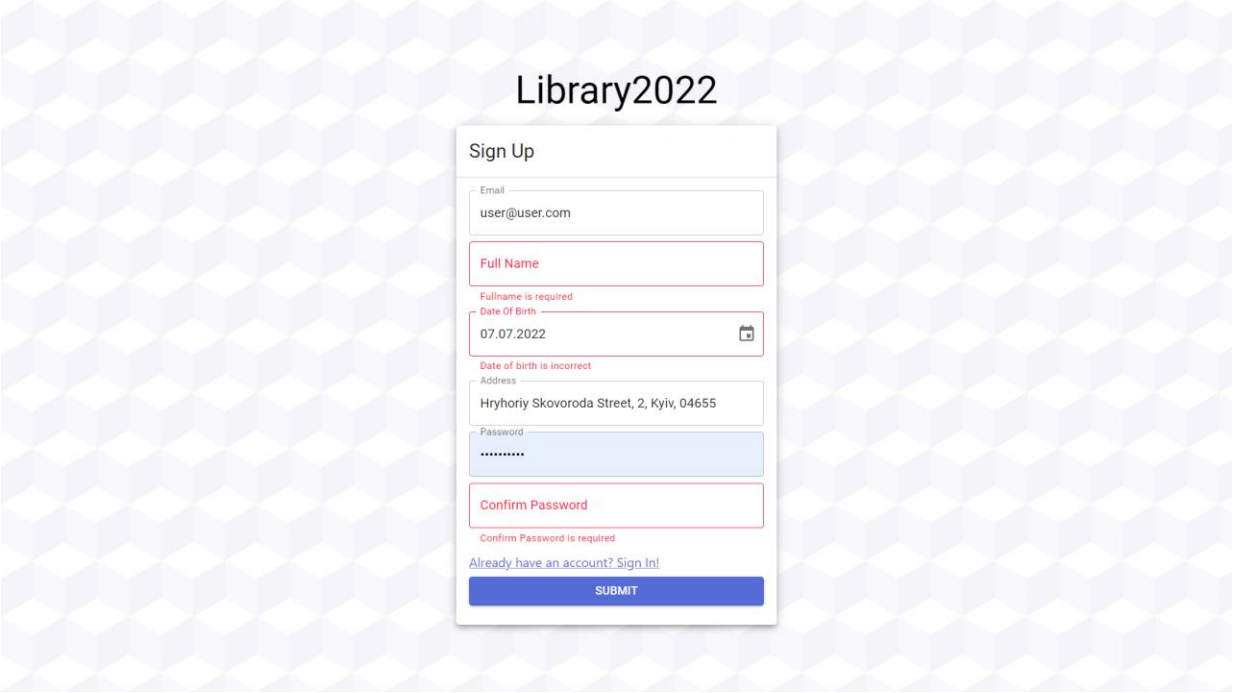

Рисунок 2.7 – Сторінка реєстрації

Увійшовши у систему, користувач потрапить до каталогу книжок, а згори над основним наповненням сторінки з’явиться панель навігації, де можна переключатися між основними сторінками застосування (список замовлень, каталог книжок, створення книжки для адміністратора та кошик для звичайного користувача). Кнопка кошику присутня тільки для користувача типу «USER», а біля неї знаходиться число, що позначає кількість книжок у кошику. У правому куту

панелі навігації знаходиться кнопка виходу з системи, натиснувши на яку користувач вийде з облікового запису та потрапить на сторінку входу.

Сторінка каталогу книжок містить список книжок у бібліотеці та панель пошуку (рисунок 2.8). Пошук здійснюється за назвою або ім’ям автора у першому пошуковому полі та за категорією у другому полі. Категорії можна обрати з випадаючого списку або ввести вручну. Якщо введено декілька категорій, буде показано тільки ті книги, що мають всі категорії з переліку. Список книжок використовує пагінацію, відображаючи по 10 книг на одній сторінці. Переключатися між сторінками каталогу можна за допомогою спеціальних панелей, що знаходяться над та під списком.

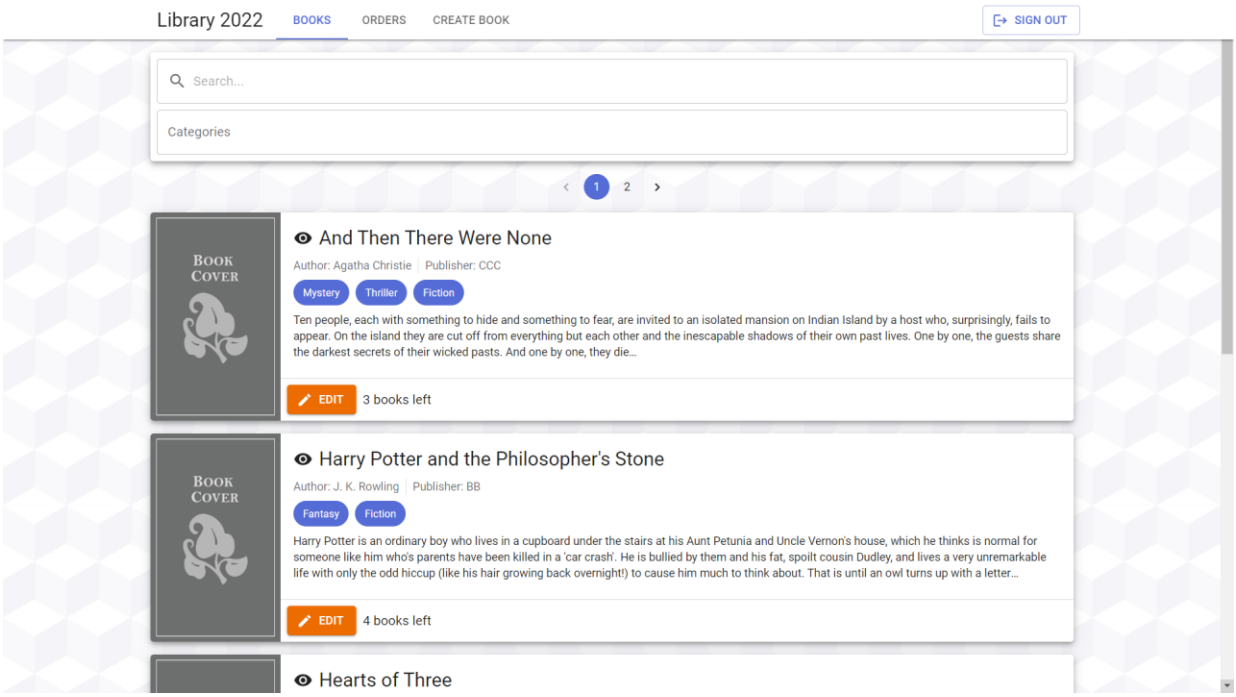

Рисунок 2.8 – Сторінка каталогу книжок

Кожна книга у переліку представлена у вигляді картки, де вказано назву, автора, видавництво, список категорій, короткий опис та кількість доступних екземплярів. Для звичайного користувача у нижній частині картки є кнопка, щоб

додати книгу до кошика («ADD»). Якщо у бібліотеці недостатньо доступних екземплярів або книжка вже є в кошику, ця кнопка буде неактивною. Всього до кошику можна додати не більше 3 одиниць. Якщо каталог переглядає адміністратор, біля назви книги буде іконка, що позначає, чи вона прихована для інших користувачів, а замість кнопки, що додає до кошику, буде кнопка редагування («EDIT»), натиснувши на яку адміністратор потрапить на сторінку редагування книги.

На сторінці редагування книги представлено заповнену форму з інформацією про обрану книгу (рисунок 2.9). У цій формі можна змінити назву книги, ім’я автора, назву видавництва, опис, категорії та чи є книга прихованою. Категорії можна обрати з переліку або ввести власноруч, їх можна бути декілька або жодної. Кількість примірників та ISBN залишаються фіксованими. Для форм редагування та створення книги мають виконуватися такі умови:

a) ISBN має складатися з 13-ти цифр;

b) Назва не може бути порожньою;

c) Ім’я автора мінімум з 2 літер;

d) Назва видавництва мінімум з 2 літер;

e) Опис не може бути порожнім.

Якщо всі умови виконуються, після відправки форми буде показано повідомлення про успіх, інакше біля хибних полів з’явиться опис помилки. Сторінка створення книги працює аналогічним чином, але в формі нової книги можна вказати ISBN та кількість примірників (рисунок 2.10). Після оновлення або створення нової книги, ці зміни відтворяться у книжковому каталозі.

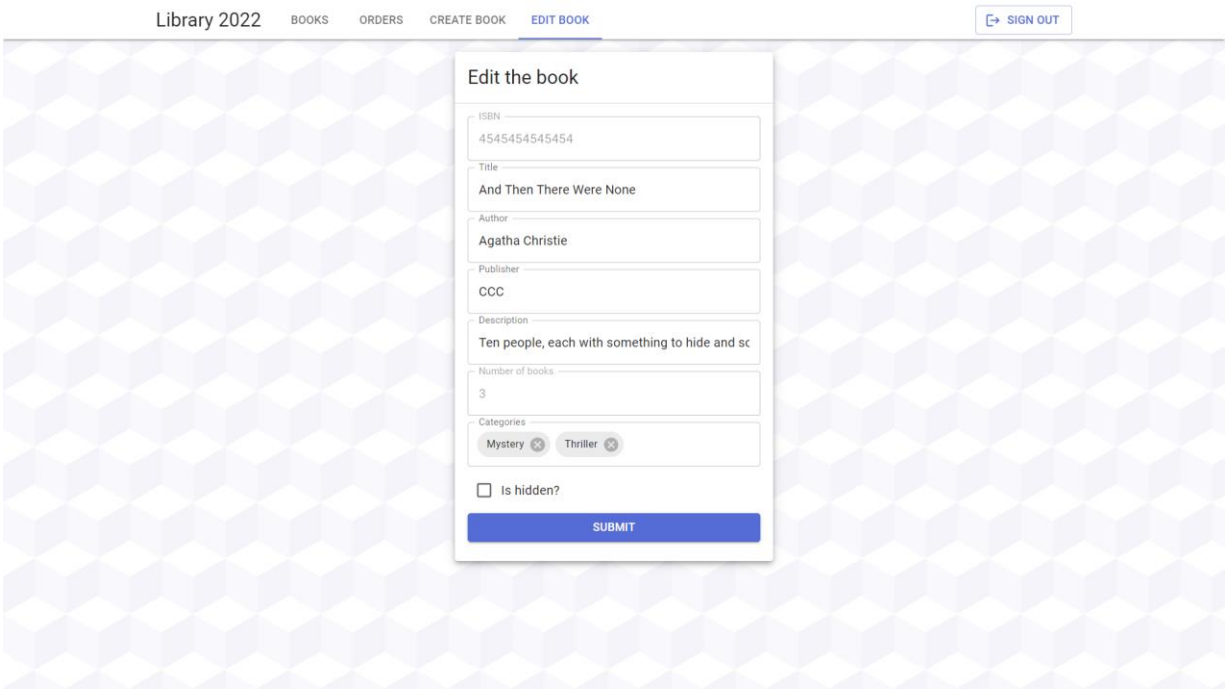

Рисунок 2.9 – Сторінка редагування книги

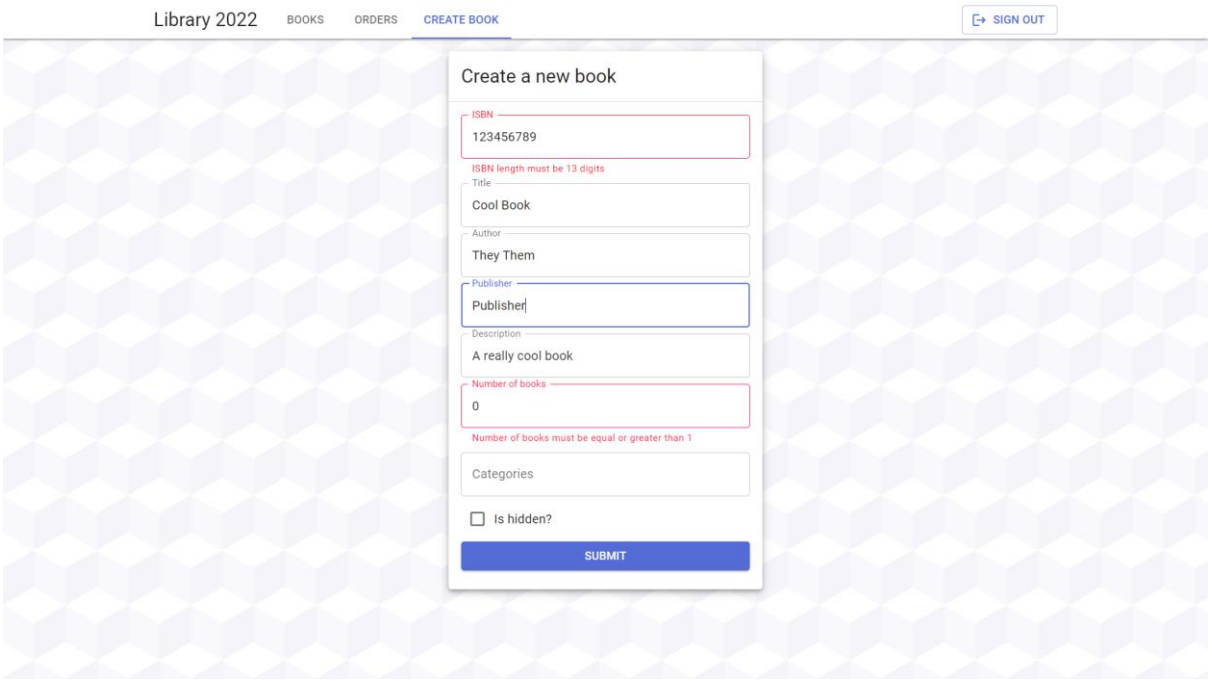

Рисунок 2.10 – Сторінка створення книги

Коли користувач типу «USER» обрав бажані книжки, він може переглянути обрані примірники на сторінці кошику (рисунок 2.11). У цьому розділі можна

переглянути список обраних книжок та прибрати деякі з них за бажанням. Якщо прибрати всі книжки, користувач автоматично перейде на сторінку каталогу. Над переліком книг у кошику є форма для оформлення замовлення. У формі треба вказати дату очікуваного повернення (від 3 до 180 днів від поточної дати). Якщо дату введено вірно, то натиснувши на кнопку «ORDER», буде створено нове замовлення, а користувача перенаправить на сторінку з переліком замовлень. Якщо дату вказано невірно, біля поля з’явиться відповідна помилка. Також можна натиснути на кнопку «CANCEL», щоб відмінити замовлення, тоді кошик очиститься, і користувач повернеться до каталогу книжок.

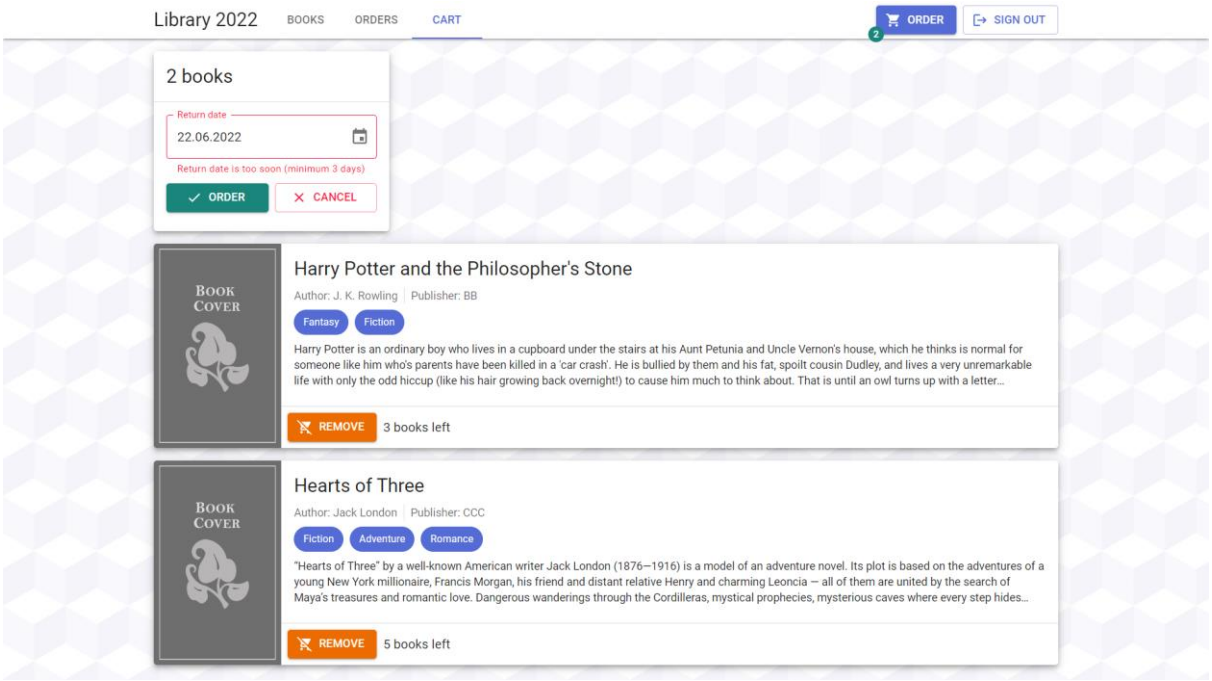

Рисунок 2.11 – Сторінка кошику

На сторінці замовлень знаходиться список замовлень з пагінацією та панель пошуку для здійснення пошуку за ідентифікатором та статусом. Для пошуку за статусом можна обрати тільки один з переліку. Звичайний користувач може переглядати лише власні замовлення, а адміністратор бачить повний список

замовлень від всіх користувачів. Кожне замовлення представлено у вигляді картки, що містить таку інформацію: унікальний ідентифікатор, статус, дата оформлення, дата очікуваного повернення, дата отримання (якщо користувач отримав замовлення), дата повернення (якщо користувач повернув книги, або замовлення було скасовано), повне ім’я користувача, його електронна пошта, кількість книжок у замовлення та перелік примірників. Для кожного примірника зазначено назву книги, ім’я автора, ISBN та ідентифікаційний номер примірника.

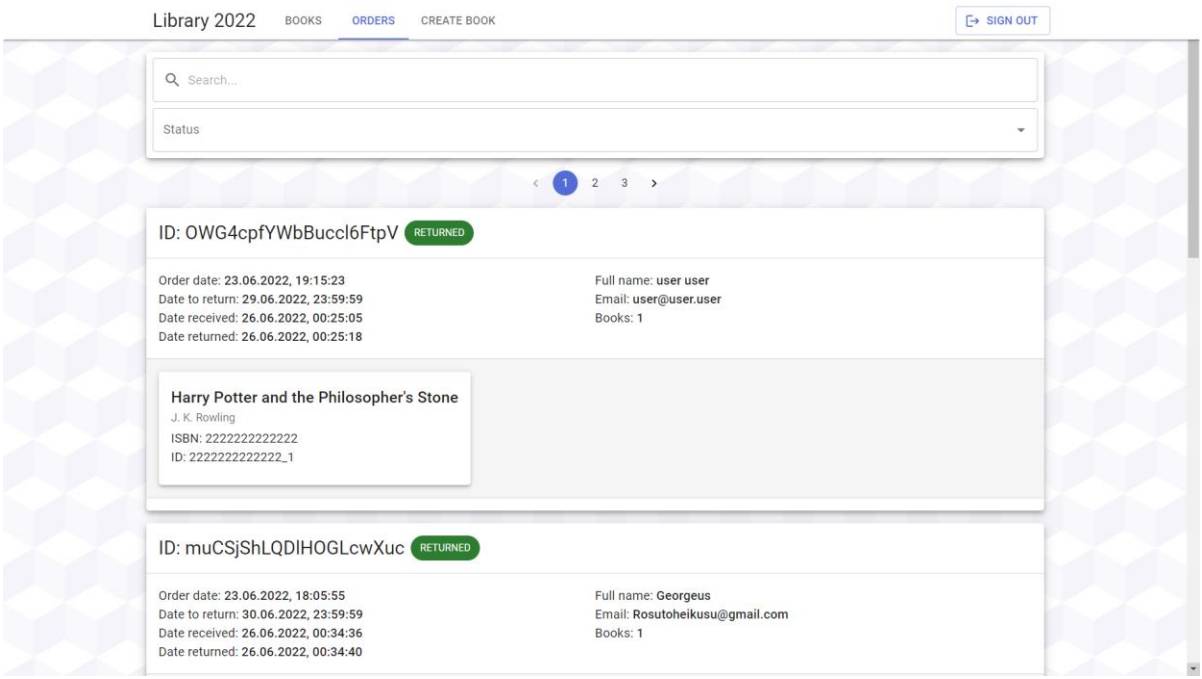

Рисунок 2.12 – Сторінка перегляду замовлень

Статуси замовлення працюють наступним чином:

a) Якщо замовлення має статус «PENDING», звичайний користувач може його відмінити (кнопка «CANCEL»), а адміністратор може або відмінити або підтвердити готовність (кнопка «READY»);

b) Якщо замовлення має статус «READY», звичайний користувач все ще може відмінити його, а адміністратор може відмінити або підтвердити, що користувач забрав свої книжки з бібліотеки, натиснувши «RECEIVED»;

c) Якщо замовлення має статус «RECEIVED», адміністратор може підтвердити, що книги повернули до бібліотеки, натиснувши «RETURNED»;

d) Над замовленнями, що мають статус «RETURNED» або «CANCELED» не можна здійснювати будь-які дії.

Коли замовлення має статуси «PENDING», «READY», «RECEIVED», примірники, що знаходяться в ньому, є недоступними, тому кількість доступних примірників зменшується для відповідних книжок у каталозі. Якщо замовлено всі примірники книги, її більше не можна додати до замовлення. Коли статус замовлення змінюється на «RETURNED» або «CANCELED», примірники з нього знову стають доступними.

2.5.3 Структура даних у Firestore

Структура бази даних для веб-застосування бібліотеки складається з трьох колекцій документів: books (інформації про книги), orders (інформація про замовлення) та users (інформація про користувачів). Так як Firestore – це NoSQL база даних, для представлення даних у вигляді документів є прийнятним дублювання інформації у різних сутностях, що спрощує читання інформації, але ускладнює процес оновлення. Документи з колекції book мають таку структуру:

a) author – ім’я автора;

b) categories – список категорій книги;

c) description – опис книги;

d) isHidden – чи книга прихована від звичайних користувачів;

e) isbn – ISBN книги;

f) items – інформація про примірники книги:

1) id – унікальний ідентифікатор екземпляру книги;

2) ordered – чи цей примірники книги замовили.

g) publisher – видавництво;

h) title – назва книги.

Інформація про книги також дублюється у Algolia, що створює індекс книг для пошуку за назвою та автором, а також фільтрації за категоріями. Документи з колекції orders мають наступну структуру:

a) dateOrdered – дата і час створення;

b) dateReceived – дата і час отримання;

c) dateReturned – дата і час повернення;

d) dateToReturn – дата і час очікуваного повернення;

e) id – ідентифікатор;

f) isbns – перелік ISBN обраних книг;

g) items – інформація про примірники з замовлення:

1) author – ім’я автора;

2) id – ідентифікатор примірника;

3) isbn – ISBN книги;

4) title – назва книги;

h) status – статус замовлення;

i) userEmail – електронна пошта користувача, що замовив;

j) userFullName – ім’я користувача, що замовив.

Документи з колекції users мають таку структуру:

a) address – адреса проживання;

b) dateOfBirth – дата народження;

c) email – електронна пошта;

d) fullName – ім’я;

e) password – пароль (у зашифрованому вигляді);

f) role – роль користувача.

2.6 Тестування доступності веб-застосування

У якості платформи для тестування доступності розробленого веб-застосування використано веб-сервіс Loadster. Цей онлайн ресурс надає можливість як для тестування як інтерфейсів веб-сайтів, так і для перевірки роботи API під навантаженням. Також передбачено можливість розробки сценаріїв використання застосування для тестування цього сценарію за допомогою певної кількості ботів, що виконують його одночасно.

Метою тестування є перевірка веб-застосування на швидкодію та доступність з певною кількістю одночасних користувачів. Для тестування веб-сайту було розроблено сценарій, що використовує більшу частину функціоналу сайту для адміністратора. Запис сценаріїв відбувається за допомогою спеціального розширення під назвою «Loadster Recorder» для веб-оглядача Google Chrome, що записує дії користувача на сторінці сайту для подальшого відтворення. Записаний сценарій передбачає наступну послідовність дій:

a) Відкрити сторінки сайту;

b) Увійти у систему за допомогою email та паролю тестового облікового запису адміністратора;

c) Ввести в пошукову стрічку для назви книги «and»;

d) Перейти на сторінку редагування першої книги зі списку;

e) Натиснути кнопку «SUBMIT» для збереження книги без змін;

f) Перейти на сторінку створення книги;

g) Натиснути кнопку «SUBMIT», щоб з’явилося повідомлення про помилку.

h) Перейти на сторінку списку замовлень;

i) Обрати для пошуку статус «PENDING» з випадаючого списку;

j) Вийти з облікового запису, натиснувши кнопку «SIGN OUT».

З використанням описаного сценарію було проведено тестування, що застосовує 25 ботів для одночасного доступу до веб-сайту. Загальна тривалість тесту складає 5 хв 23 с. За результатами тесту середній час відповіді на запит складає 0.49 с, 50% запитів займає менше 0.11 с, а 90% запитів займає менше 0.43 с (рисунок 2.14). Таким чином, швидкодія застосування є прийнятною, адже очікуваний показник є менше 1 с. Найбільший час очікування спостерігається на сторінці з книжками (0.49 с) пояснюється це тим, що Важливим фактором, що забезпечує швидку роботу інтерфейсу, є застосування архітектури SPA, що забезпечує завантаження сторінок без оновлення вкладки веб-оглядача та без необхідності завантажувати розмітку та стилі для кожної сторінки.

Під час тестування GCP автоматично запускає максимум чотири екземпляри для сервісу «api», однак активними залишаються максимум два, а здебільшого - один. Екземпляри, що простоюють, виступають у ролі запасних на випадок збільшення навантаження. Отже GCP забезпечує динамічне масштабування та виділення резервних ресурсів під час різкого збільшення навантаження, що забезпечує високий рівень доступності застосувань, розміщених за допомогою Google App Engine.

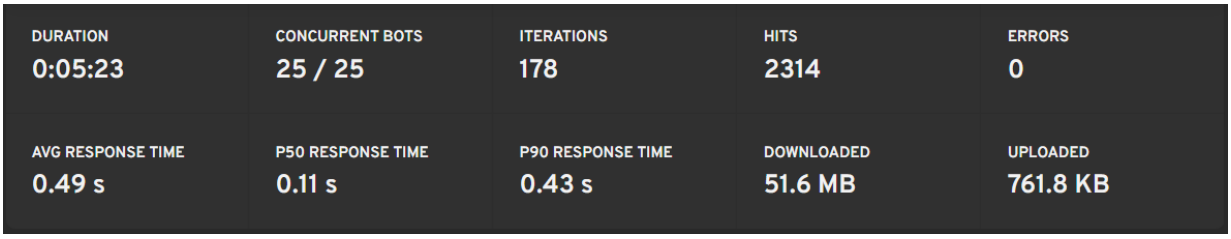

Рисунок 2.14 – Результати тестування веб-застосування бібліотеки за допомогою Loadster

ВИСНОВКИ

В результаті проведеної роботи було розроблено багаторівневе веб-застосування для бібліотеки з використанням GCP. Серед ресурсів, що надає хмарна платформа від Google було використано Google App Engine для розміщення веб-застосування та Firestore для роботи з хмарною NoSQL базою даних. App Engine надає можливість створювати масштабовані веб-застосування з високою доступністю, що забезпечується Google Front End, що убезпечує та зменшує вплив від великої кількості мережевих атак.

Розроблене веб-застосування надає простий та адаптивний інтерфейс для роботи з книжками та замовленнями. Основа системи складається з сервісу інтерфейсу користувача та сервісу бізнес-логіки, для яких налаштовано автоматичне масштабування, що забезпечує високий рівень доступності. Завдяки тому, що користувацький інтерфейс виконано за архітектурою SPA, це значно зменшує кількість запитів до серверу для відображення сторінок веб-ресурсу, що зменшує навантаження на сервіс.

Розроблене рішення має деякі перспективи для покращення, пов’язані у першу чергу з розширенням функціоналу самого застосування. Наприклад, можна додати процедуру підтвердження поштової адреси користувача під час реєстрації та сторінку профілю користувача для перегляду та редагування інформації облікового запису. Також недоліком є відсутність можливості змінювати кількість примірників кожної книги. До того ж, хоч використання SPA робить процес користування системою більш швидким, такий підхід до розробки інтерфейсу спричиняє проблеми для ранжування у пошукових системах, які необхідно буде вирішувати під час SEO оптимізації.

Таким чином розроблене багаторівневе веб-застосування бібліотеки реалізує поставлені функціональні вимоги та забезпечує високий рівень доступності за допомогою GCP.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Martin M. N Tier(Multi-Tier), 3-Tier, 2-Tier Architecture with EXAMPLE [Електронний ресурс] / Matthew Martin // Guru99. – Режим доступу: https://www.guru99.com/n-tier-architecture-system-concepts-tips.html (дата звернення: 31.05.2022). – Назва з екрана.

2. What is Three-Tier Architecture [Електронний ресурс] // IBM - Deutschland | IBM. – Режим доступу: https://www.ibm.com/cloud/learn/three-tier-architecture (дата звернення: 31.05.2022). – Назва з екрана.

3. Ranger S. What is cloud computing? Everything you need to know about the cloud explained [Електронний ресурс] / Steve Ranger // ZDNet. – Режим доступу: https://www.zdnet.com/article/what-is-cloud-computing-everything-you-need-to-know-about-the-cloud/ (дата звернення: 01.06.2022). – Назва з екрана.

4. Frankenfield J. How Cloud Computing Works [Електронний ресурс] / Jake Frankenfield // Investopedia. – Режим доступу: https://www.investopedia.com/terms/c/cloud-computing.asp (дата звернення: 01.06.2022). – Назва з екрана.

5. Chai W. What is Cloud Computing? Everything You Need to Know [Електронний ресурс] / Wesley Chai, Stephen J. Bigelow // SearchCloudComputing. – Режим доступу: https://www.techtarget.com/searchcloudcomputing/definition/cloud-computing (дата звернення: 01.06.2022). – Назва з екрана.

6. Fanchi C. What Is Serverless Computing? | Backendless Mobile Backend [Електронний ресурс] / Christopher Fanchi // Backendless. – Режим доступу: https://backendless.com/what-is-serverless-computing/ (дата звернення: 04.06.2022). – Назва з екрана.

7. Bigelow S. J. What is Google Cloud? [Електронний ресурс] / Stephen J. Bigelow // SearchCloudComputing. – Режим доступу:

https://www.techtarget.com/searchcloudcomputing/definition/Google-Cloud-Platform (дата звернення: 03.06.2022). – Назва з екрана.

8. Products and Services | Google Cloud [Електронний ресурс] // Google Cloud. – Режим доступу: https://cloud.google.com/products/ (дата звернення: 03.06.2022). – Назва з екрана.

9. GKE overview | Kubernetes Engine Documentation | Google Cloud [Електронний ресурс] // Google Cloud. – Режим доступу: https://cloud.google.com/kubernetes-engine/docs/concepts/kubernetes-engine-overview (дата звернення: 04.06.2022). – Назва з екрана.

10. Serverless computing | Google Cloud [Електронний ресурс] // Google Cloud. – Режим доступу: https://cloud.google.com/serverless (дата звернення: 04.06.2022). – Назва з екрана.

11. Binns R. What is Uptime, and How Can You Maximize Your Website's? [Електронний ресурс] / Rob Binns // Website Builder Expert. – Режим доступу: https://www.websitebuilderexpert.com/web-hosting/what-is-uptime/ (дата звернення: 04.06.2022). – Назва з екрана.

12. Leng A. 1 in 2 visitors abandon a website that takes more than 6 seconds to load - Digital.com [Електронний ресурс] / Allen Leng // Digital.com. – Режим доступу: https://digital.com/1-in-2-visitors-abandon-a-website-that-takes-more-than-6-seconds-to-load/ (дата звернення: 04.06.2022). – Назва з екрана.

13. Choose an App Engine environment | App Engine Documentation | Google Cloud [Електронний ресурс] // Google Cloud. – Режим доступу: https://cloud.google.com/appengine/docs/the-appengine-environments (дата звернення: 05.06.2022). – Назва з екрана.

14. Under the Hood with GCP’s App Engine [Електронний ресурс] // The Applied Architect. – Режим доступу: http://www.theappliedarchitect.com/under-the-hood-with-gcps-app-engine/ (дата звернення: 05.06.2022). – Назва з екрана.

15. How Instances are Managed | App Engine standard environment for Node.js docs | Google Cloud [Електронний ресурс] // Google Cloud. – Режим доступу: https://cloud.google.com/appengine/docs/standard/nodejs/how-instances-are-managed (дата звернення: 05.06.2022). – Назва з екрана.

16. Cloud Load Balancing overview | Google Cloud [Електронний ресурс] // Google Cloud. – Режим доступу: https://cloud.google.com/load-balancing/docs/load-balancing-overview (дата звернення: 05.06.2022). – Назва з екрана.

17. @angular/core vs react vs vue | npm trends. npm trends: Compare NPM package downloads. URL: https://www.npmtrends.com/@angular/core-vs-react-vs-vue (дата звернення: 14.06.2022).

18. Stack Overflow. Stack Overflow Insights - Developer Hiring, Marketing, and User Research. URL: https://insights.stackoverflow.com/trends?tags=angular,reactjs,vue.js (дата звернення: 14.06.2022).

19. Dziuba A. Angular vs. React vs. Vue.js - choosing a JavaScript framework for your project. Relevant Software. URL: https://relevant.software/blog/angular-vs-react-vs-vue-js-choosing-a-javascript-framework-for-your-project/ (дата звернення: 17.06.2022).

20. Danielkievich A. Exploring Node.js Pros and Cons - Forbytes. Forbytes. URL: https://forbytes.com/blog/nodejs-pros-and-cons/ (дата звернення: 18.06.2022).

21. Firestore: NoSQL document database | Google Cloud. Google Cloud. URL: https://cloud.google.com/firestore#all-features (дата звернення: 18.06.2022).

22. Material UI - Overview - Material UI. MUI: The React component library you always wanted. URL: https://mui.com/material-ui/getting-started/overview/ (дата звернення: 25.06.2022).

23. Redux Essentials, Part 1: Redux Overview and Concepts | Redux. Redux - A predictable state container for JavaScript apps. | Redux. URL: https://redux.js.org/tutorials/essentials/part-1-overview-concepts (дата звернення: 25.06.2022).

24. Best Practices for DDoS Protection and Mitigation on Google Cloud Platform. Cloud Computing Services | Google Cloud. URL: https://cloud.google.com/files/GCPDDoSprotection-04122016.pdf (дата звернення: 26.06.2022).

25. Gillis A. What is AWS (Amazon Web Services) and How Does it Work?. SearchAWS. URL: https://www.techtarget.com/searchaws/definition/Amazon-Web-Services (дата звернення: 27.06.2022).

26. Managed DDoS protection - AWS Shield - Amazon Web Services. Amazon Web Services, Inc. URL: https://aws.amazon.com/shield/?nc1=h_ls&whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=desc (дата звернення: 27.06.2022).

27. Klint L. What is Azure? Microsoft's cloud platform explained. A Cloud Guru. URL: https://acloudguru.com/blog/engineering/what-is-microsoft-azure (дата звернення: 27.06.2022).

ДОДАТОК А. ЧАСТИНА ПРОГРАМНОГО КОДУ СЕРВІСУ БІЗНЕС-ЛОГІКИ

src/services/book.ts

```
import {
  CollectionReference,
  DocumentData,
  Query,
  QuerySnapshot,
} from '@google-cloud/firestore'
import { Book, BookInfo, BookItem, NewBook } from 'types'

import AlgoliaClient from '../algolia/algoliaClient'
import FirestoreClient from '../firestore/firestoreClient'
import OrderServices from './order'

const bookCol = FirestoreClient.collection('books')
const ordersCol = FirestoreClient.collection('orders')
const bookIndex = AlgoliaClient.initIndex('books_index')

export interface GetBooksProps {
  showHidden: boolean
  startAt: number
  limit: number
}

export interface SearchBooksProps {
```

```
  showHidden: boolean
  startAt: number
  limit: number
  categories: string[]
  search: string
}

const dataToBook = (data: DocumentData, isbn: string): BookInfo => {
  return {
    isbn,
    title: data.title,
    author: data.author,
    publisher: data.publisher,
    description: data.description,
    categories: data.categories,
    isHidden: data.isHidden,
    items: data.items,
  }
}

const snapshotToBooks = (snapshot: QuerySnapshot): BookInfo[] => {
  return snapshot.docs.map((doc) => dataToBook(doc.data(), doc.id))
}

const search = async ({
  showHidden,
  startAt,
```

```
  limit,
  categories,
  search,
}: SearchBooksProps): Promise<{ books: BookInfo[]; totalCount: number }> => {
  const categoriesFilterStr = categories.length
    ? `categories:${categories.join(' AND categories:')}`
    : ''
  const isHiddenFilterStr = showHidden ? '' : 'isHidden:false'
  const filterStr = [categoriesFilterStr, isHiddenFilterStr]
    .filter((x) => x)
    .join(' AND ')
  console.log(filterStr)
  const result = await bookIndex.search<BookInfo>(search, {
    filters: filterStr || undefined,
    offset: startAt,
    length: limit,
  })
  return { books: result.hits as BookInfo[], totalCount: result.nbHits }
}

const getAll = async ({ showHidden, startAt, limit }: GetBooksProps) => {
  let query: CollectionReference | Query = bookCol
  if (!showHidden) {
    query = bookCol.where('isHidden', '==', false)
  }
  query = query.orderBy('title')
  const totalCount = (await query.get()).size
```

```
  const first = (await query.limit(startAt + 1).get()).docs.pop()
  if (first) {
    query = query.startAt(first.data().title)
  }
  if (limit) {
    query = query.limit(limit)
  }
  const snapshot = await query.get()
  return { books: snapshotToBooks(snapshot), totalCount }
}

const getByISBN = async (isbn: string) => {
  const snapshot = await bookCol.doc(isbn).get()
  const data = snapshot.data()
  if (data) {
    return dataToBook(data, isbn)
  } else {
    throw new Error(`No book with ISBN: ${isbn} found`)
  }
}

const update = async (
  isbn: string,
  newBook: Partial<Book>,
): Promise<BookInfo> => {
  const bookRef = bookCol.doc(isbn)
  const existingBook = await bookRef.get()
```

```
const bookData = existingBook.data()
if (!bookData) {
  throw new Error(`Book with ISBN: ${isbn} doesn't exist`)
}
const batch = FirestoreClient.batch()
const oldBook = dataToBook(bookData, isbn)
const authorIsNew = newBook.author && newBook.author !== oldBook.author
const titleIsNew = newBook.title && newBook.title !== oldBook.title
if (authorIsNew || titleIsNew) {
  const snap = await ordersCol.where('isbns', 'array-contains', isbn).get()
  const orders = OrderServices.snapshotToOrders(snap)
  orders.forEach((order) => {
    batch.update(ordersCol.doc(order.id), {
      items: order.items.map((item) => ({
        ...item,
        author: authorIsNew ? newBook.author : item.author,
        title: titleIsNew ? newBook.title : item.title,
      })),
    })
  })
}
batch.update(bookRef, newBook)
const res: BookInfo = {
  ...dataToBook(bookData, isbn),
  ...newBook,
}
await batch.commit()
```

```
  await bookIndex.saveObject({ ...res, objectID: isbn })
  return res
}

const create = async (newBook: NewBook) => {
  const book: BookInfo = {
    isbn: newBook.isbn,
    title: newBook.title,
    author: newBook.author,
    publisher: newBook.publisher,
    description: newBook.description,
    categories: newBook.categories,
    isHidden: newBook.isHidden,
    items: [],
  }
  const bookRef = bookCol.doc(book.isbn)
  const existingBook = await bookRef.get()
  if (existingBook.data()) {
    throw new Error(`Book with ISBN: ${book.isbn} exists`)
  }
  for (let i = 0; i < newBook.itemCount; i += 1) {
    const bookItem: BookItem = {
      id: `${newBook.isbn}_${i}`,
      isbn: newBook.isbn,
      ordered: false,
    }
    book.items.push(bookItem)
```

```
  }
  bookRef.set(book, { merge: true })
  await bookIndex.saveObject({ ...book, objectID: book.isbn })
  return book
}

const BookServices = {
  getAll,
  getByISBN,
  create,
  update,
  dataToBook,
  snapshotToBooks,
  search,
}

export default BookServices
```

src/services/order.ts

```
import {
  CollectionReference,
  DocumentData,
  Query,
  QuerySnapshot,
  WriteBatch,
} from '@google-cloud/firestore'
```

```
import { NewOrder, Order, OrderStatus } from 'types'

import FirestoreClient from '../firestore/firestoreClient'
import BookServices from './book'

const ordersCol = FirestoreClient.collection('orders')
const booksCol = FirestoreClient.collection('books')

export interface GetOrderProps {
  startAt: number
  limit: number
  status?: string
  search?: string
  userEmail?: string
}

const dataToOrder = (data: DocumentData): Order => {
  return {
    id: data.id,
    userFullName: data.userFullName,
    userEmail: data.userEmail,
    items: data.items,
    status: data.status,
    dateToReturn: data.dateToReturn,
    dateOrdered: data.dateOrdered,
    dateReceived: data.dateReceived,
    dateReturned: data.dateReturned,
```

```
    isbns: data.isbns,
  }
}

const snapshotToOrders = (snapshot: QuerySnapshot): Order[] => {
  return snapshot.docs.map((doc) => dataToOrder(doc.data()))
}

interface UpdateItemsInBooksProps {
  batch: WriteBatch
  newOrdered: boolean
  itemIds: string[]
  isbns: string[]
}

const updateItemsInBooks = async ({
  batch,
  newOrdered,
  itemIds,
  isbns,
}: UpdateItemsInBooksProps) => {
  const snap = await booksCol.where('isbn', 'in', isbns).get()
  const books = BookServices.snapshotToBooks(snap)
  books.forEach((book) => {
    batch.update(booksCol.doc(book.isbn), {
      items: book.items.map((item) => ({
        ...item,
```

```
        ordered: itemIds.indexOf(item.id) >= 0 ? newOrdered : item.ordered,
      })),
    })
  })
}

const create = async (newOrder: NewOrder): Promise<Order> => {
  const batch = FirestoreClient.batch()
  const orderRef = ordersCol.doc()
  const isbns = newOrder.items.map((item) => item.isbn)
  const itemIds = newOrder.items.map((item) => item.id)
  const order: Order = {
    ...newOrder,
    id: orderRef.id,
    isbns,
    dateOrdered: new Date().toISOString(),
    status: 'PENDING',
  }
  await updateItemsInBooks({
    batch,
    newOrdered: true,
    isbns,
    itemIds,
  })
  batch.set(orderRef, order)
  await batch.commit()
  return order
```

```
}

const isOrdered = (status: OrderStatus) => {
  return status === 'PENDING' || status === 'READY' || status === 'RECEIVED'
}

const updateOrderStatus = async (orderId: string, newStatus: OrderStatus) => {
  const orderRef = ordersCol.doc(orderId)
  const orderData = (await orderRef.get()).data()
  if (!orderData) {
    throw new Error(`Order with id: ${orderId} doesn't exist`)
  }
  const batch = FirestoreClient.batch()
  const order = dataToOrder(orderData)
  const updateData: Partial<Order> = { status: newStatus }
  const itemIds = order.items.map((item) => item.id)
  const oldOrdered = isOrdered(order.status)
  const newOrdered = isOrdered(newStatus)
  const newOrder: Order = { ...order, status: newStatus }
  if (oldOrdered !== newOrdered) {
    await updateItemsInBooks({
      batch,
      newOrdered,
      isbns: order.isbns,
      itemIds,
    })
  }
```

```
  if (newStatus === 'RECEIVED') {
    updateData.dateReceived = new Date().toISOString()
  }
  if (newStatus === 'CANCELED' || newStatus === 'RETURNED') {
    updateData.dateReturned = new Date().toISOString()
  }
  batch.update(orderRef, updateData)
  await batch.commit()
  return newOrder
}

const getAll = async ({
  userEmail,
  status,
  limit,
  startAt,
  search,
}: GetOrderProps) => {
  let query: CollectionReference | Query = ordersCol
  if (userEmail) {
    query = query.where('userEmail', '==', userEmail)
  }
  if (status) {
    query = query.where('status', '==', status)
  }
  if (search && search.length) {
    query = query
```

```
      .where('id', '>=', search)
      .where('id', '<=', search + '~')
      .orderBy('id')
  }
  query = query.orderBy('dateOrdered', 'desc')
  const orderedData = await query.get()
  const totalCount = orderedData.size
  const first = (await query.limit(startAt + 1).get()).docs.pop()
  if (first) {
    query = query.startAt(first.data().dateOrdered)
  }
  if (limit) {
    query = query.limit(limit)
  }
  const snapshot = await query.get()
  return { orders: snapshotToOrders(snapshot), totalCount }
}

const OrderServices = {
  create,
  getAll,
  dataToOrder,
  snapshotToOrders,
  updateOrderStatus,
}

export default OrderServices
```

src/services/user.ts

```
import { DocumentData, QuerySnapshot } from '@google-cloud/firestore'
import { NewUser, User } from 'types'

import FirestoreClient from '../firestore/firestoreClient'

const userCol = FirestoreClient.collection('users')

const dataToUser = (data: DocumentData, email: string): User => {
  return {
    email,
    password: data.password,
    role: data.role,
    address: data.address,
    fullName: data.fullName,
    dateOfBirth: data.dateOfBirth,
  }
}

const qureyToUsers = (snapshot: QuerySnapshot): User[] => {
  return snapshot.docs.map((doc) => dataToUser(doc.data(), doc.id))
}

const getAll = async () => {
  const res = await userCol.orderBy('email').get()
  return qureyToUsers(res)
```

```
}

const getByEmail = async (email: string) => {
  const res = await userCol.doc(email).get()
  const data = res.data()
  return data ? dataToUser(data, email) : undefined
}

const create = async (newUser: NewUser): Promise<User> => {
  const user: User = {
    ...newUser,
    role: 'USER',
  }
  await userCol.doc(newUser.email).set(user, { merge: true })
  return user
}

const UserServices = {
  getAll,
  getByEmail,
  create,
}

export default UserServices
```

src/controllers/auth.ts

```
import bcrypt from 'bcryptjs'
import { Request, Response } from 'express'
import jwt from 'jsonwebtoken'

import UserServices from '../services/user'

const signup = async (req: Request, res: Response) => {
  try {
    const { email, password, address, fullName, dateOfBirth } = req.body
    const user = await UserServices.getByEmail(email)
    if (user) {
      return res.status(400).send('User with this email already exists.')
    }
    await UserServices.create({
      password: bcrypt.hashSync(password, 8),
      email,
      address,
      fullName,
      dateOfBirth,
    })
    return res.status(200).send('User registered successfully!')
  } catch (error: any) {
    return res.status(500).send(error.message)
  }
}

const signin = async (req: Request, res: Response) => {
```

```
  try {
    const user = await UserServices.getByEmail(req.body.email)
    if (!user) {
      return res.status(404).send('User Not found.')
    }
    const passwordIsValid = bcrypt.compareSync(req.body.password, user.password)
    if (!passwordIsValid) {
      return res.status(401).send('Invalid Password!')
    }
    const token = jwt.sign(
      { email: user.email, role: user.role },
      process.env.JWT_SECRET || 'secret',
      {
        expiresIn: 86400, // 24 hours
      },
    )
    if (!req.session) {
      req.session = { token }
    } else {
      req.session.token = token
    }
    return res.status(200).send(user)
  } catch (error: any) {
    return res.status(500).send(error.message)
  }
}
```

```
const signout = async (req: Request, res: Response) => {
  if (req.session) {
    req.session.token = null
  }
  res.status(200).json('User signed out successfully')
}

const AuthController = {
  signup,
  signin,
  signout,
}

export default AuthController
```

src/controllers/book.ts

```
import { Request, Response } from 'express'
import { Book } from 'types'

import BookServices from '../services/book'

const getAll = async (req: Request, res: Response) => {
  const { role } = res.locals.user
  const search = req.query.search as string
  const startAt = Number(req.query.startAt as string)
  const limit = Number(req.query.limit as string)
```

```
  const categories = req.query.categories
    ? (req.query.categories as string).split(';')
    : []
  const showHidden = role === 'ADMIN'
  try {
    const result = await (search || categories.length
      ? BookServices.search({
          showHidden,
          startAt,
          limit,
          search,
          categories,
        })
      : BookServices.getAll({
          showHidden,
          startAt,
          limit,
        }))
    return res.status(200).send(result)
  } catch (error: any) {
    return res.status(500).send(error.message)
  }
}

const getByISBN = async (req: Request, res: Response) => {
  const isbn = req.params.isbn
  try {
```

```
    const book = await BookServices.getByISBN(isbn)
    return res.status(200).send(book)
  } catch (error: any) {
    return res.status(500).send(error.message)
  }
}

const create = async (req: Request, res: Response) => {
  const {
    isbn,
    title,
    author,
    publisher,
    description,
    categories,
    isHidden,
    itemCount,
  } = req.body
  try {
    const book = await BookServices.create({
      isbn,
      title,
      author,
      publisher,
      description,
      categories,
      isHidden,
```

```
      itemCount,
    })
    return res.status(200).send(book)
  } catch (error: any) {
    return res.status(500).send(error.message)
  }
}

const update = async (req: Request, res: Response) => {
  const isbn = req.params.isbn
  const newBook = req.body as Partial<Book>
  Object.keys(newBook).forEach((key) =>
    newBook[key as keyof Book] === undefined
      ? delete newBook[key as keyof Book]
      : {},
  )
  try {
    const book = await BookServices.update(isbn, newBook)
    return res.status(200).send(book)
  } catch (error: any) {
    return res.status(500).send(error.message)
  }
}

const BookController = {
  getAll,
  getByISBN,
```

```
  create,
  update,
}

export default BookController
```

src/controllers/order.ts

```
import { Request, Response } from 'express'
import { OrderStatus } from 'types'

import OrderServices from '../services/order'

const getAll = async (req: Request, res: Response) => {
  try {
    const search = req.query.search as string | undefined
    const status = req.query.status as string | undefined
    const userEmail = req.query.userEmail as string | undefined
    const startAt = Number(req.query.startAt as string)
    const limit = Number(req.query.limit as string)
    const result = await OrderServices.getAll({
      startAt,
      limit,
      search,
      status,
      userEmail,
    })
```

```
    return res.status(200).send(result)
  } catch (error: any) {
    return res.status(500).send(error.message)
  }
}

const create = async (req: Request, res: Response) => {
  const { userFullName, userEmail, items, dateToReturn } = req.body
  try {
    const order = await OrderServices.create({
      userFullName,
      userEmail,
      items,
      dateToReturn,
    })
    return res.status(200).send(order)
  } catch (error: any) {
    return res.status(500).send(error.message)
  }
}

const updateOrderStatus = async (req: Request, res: Response) => {
  try {
    const orderId = req.params.id
    const newStatus = req.body.status as OrderStatus
    const order = await OrderServices.updateOrderStatus(orderId, newStatus)
    return res.status(200).send(order)
```

```
  } catch (error: any) {
    return res.status(500).send(error.message)
  }
}

const OrderController = {
  getAll,
  create,
  updateOrderStatus,
}

export default OrderController
```

ДОДАТОК Б. ЧАСТИНА ПРОГРАМНОГО КОДУ КЛІЄНТСЬКОГО СЕРВІСУ

src/services/auth.ts

```
import { SignInRequest, SignUpRequest, User } from '../types'
import apiClient from './apiClient'

export const signUp = async (signUpReq: SignUpRequest) => {
  return apiClient.post<string>('auth/signup', signUpReq)
}
export const signIn = async (signInReq: SignInRequest) => {
  return apiClient.post<User>('auth/signin', signInReq)
}

export const signOut = () => {
  return apiClient.get<string>('auth/signout')
}

export const getCurrentUser = () => {
  const userStr = localStorage.getItem('user')
  if (userStr) return JSON.parse(userStr) as User
  return undefined
}
```

src/services/book.ts

```
import {
```

```
  Book,
  CreateBookRequest,
  GetBooksRequest,
  GetBooksResponse,
  UpdateBookRequest,
} from '../types'
import apiClient from './apiClient'

export const getAll = (getBooksReq: GetBooksRequest) => {
  return apiClient.get<GetBooksResponse>('book', {
    params: { ...getBooksReq, categories: getBooksReq.categories.join(';') },
  })
}
export const getByISBN = (isbn: string) => {
  return apiClient.get<Book>(`book/${isbn}`)
}

export const update = (newBook: UpdateBookRequest) => {
  return apiClient.post<Book>(`book/${newBook.isbn}`, newBook)
}

export const create = (newBook: CreateBookRequest) => {
  return apiClient.post<Book>('book', newBook)
}
```

src/services/order.ts

```
import {
  Book,
  CreateBookRequest,
  GetBooksRequest,
  GetBooksResponse,
  UpdateBookRequest,
} from '../types'
import apiClient from './apiClient'

export const getAll = (getBooksReq: GetBooksRequest) => {
  return apiClient.get<GetBooksResponse>('book', {
    params: { ...getBooksReq, categories: getBooksReq.categories.join(';') },
  })
}
export const getByISBN = (isbn: string) => {
  return apiClient.get<Book>(`book/${isbn}`)
}

export const update = (newBook: UpdateBookRequest) => {
  return apiClient.post<Book>(`book/${newBook.isbn}`, newBook)
}

export const create = (newBook: CreateBookRequest) => {
  return apiClient.post<Book>('book', newBook)
}
```

src/containers/BookCreationPage/index.tsx

```
import {
  Autocomplete,
  Button,
  Card,
  CardContent,
  CardHeader,
  Checkbox,
  Container,
  FormControlLabel,
  Stack,
  TextField,
} from '@mui/material'
import { unwrapResult } from '@reduxjs/toolkit'
import { useFormik } from 'formik'
import React from 'react'
import * as Yup from 'yup'

import { useAppDispatch } from '../../app/hooks'
import { createBook } from '../../features/book/bookSlice'
import { defaultBookCategories } from '../../utils/constants'
import { toastWrapper } from '../../utils/toastWrapper'

const validationSchema = () => {
  return Yup.object().shape({
```

```
    isbn: Yup.string().required('ISBN is required').length(13, 'ISBN length must be 13 digits'),
    title: Yup.string().required('Title is required'),
    author: Yup.string()
      .required('Author name is required')
      .min(2, 'Author name must be at least 2 characters'),
    publisher: Yup.string()
      .required('Publisher name is required')
      .min(2, 'Publisher name must be at least 2 characters'),
    description: Yup.string().required('Description is required'),
    categories: Yup.array().of(Yup.string()),
    isHidden: Yup.boolean(),
    itemCount: Yup.number()
      .required('Number of books in library is required')
      .min(1, 'Number of books must be equal or greater than 1'),
  })
}

const BookCreationPage: React.FC = () => {
  const dispatch = useAppDispatch()
  const formik = useFormik<{
    isbn: string
    title: string
    author: string
    publisher: string
    description: string
    categories: string[]
```

```
    isHidden: boolean
    itemCount: number
  }>({
    initialValues: {
      isbn: '',
      title: '',
      author: '',
      publisher: '',
      description: '',
      categories: [],
      isHidden: false,
      itemCount: 0,
    },
    validationSchema,
    onSubmit: async (values) => {
      await toastWrapper({
        action: async () => {
          const result = await dispatch(createBook(values))
          unwrapResult(result)
          formik.resetForm()
        },
      })
    },
  })
  return (
    <Container sx={{ marginTop: 8, marginBottom: 2, display: 'flex', justifyContent: 'center' }}>
```

```
<Card sx={{ width: { xs: 1, sm: 400 } }} elevation={5}>
 <CardHeader title='Create a new book' sx={{ borderBottom: 1, borderColor: 'divider' }} />
 <CardContent>
  <form onSubmit={formik.handleSubmit}>
   <Stack spacing={1}>
    <TextField
     fullWidth
     id='isbn'
     name='isbn'
     label='ISBN'
     type='text'
     value={formik.values.isbn}
     onChange={formik.handleChange}
     error={formik.touched.isbn && Boolean(formik.errors.isbn)}
     helperText={formik.touched.isbn && formik.errors.isbn}
    />
    <TextField
     fullWidth
     id='title'
     name='title'
     label='Title'
     type='text'
     value={formik.values.title}
     onChange={formik.handleChange}
     error={formik.touched.title && Boolean(formik.errors.title)}
     helperText={formik.touched.title && formik.errors.title}
```

```
/>
<TextField
  fullWidth
  id='author'
  name='author'
  label='Author'
  type='text'
  value={formik.values.author}
  onChange={formik.handleChange}
  error={formik.touched.author && Boolean(formik.errors.author)}
  helperText={formik.touched.author && formik.errors.author}
/>
<TextField
  fullWidth
  id='publisher'
  name='publisher'
  label='Publisher'
  type='text'
  value={formik.values.publisher}
  onChange={formik.handleChange}
  error={formik.touched.publisher && Boolean(formik.errors.publisher)}
  helperText={formik.touched.publisher && formik.errors.publisher}
/>
<TextField
  fullWidth
  multiline
  id='description'
```

```
  name='description'
  label='Description'
  type='text'
  value={formik.values.description}
  onChange={formik.handleChange}
  error={formik.touched.description && Boolean(formik.errors.description)}
  helperText={formik.touched.description && formik.errors.description}
/>
<TextField
  fullWidth
  id='itemCount'
  name='itemCount'
  label='Number of books'
  type='number'
  value={formik.values.itemCount}
  onChange={formik.handleChange}
  error={formik.touched.itemCount && Boolean(formik.errors.itemCount)}
  helperText={formik.touched.itemCount && formik.errors.itemCount}
/>
<Autocomplete
  clearOnBlur
  multiple
  freeSolo
  filterSelectedOptions
  onChange={(_event, value) => {
    formik.setFieldValue('categories', value)
  }}
```

```
  renderInput={(params) => (
    <TextField
      {...params}
      fullWidth
      id='categories'
      name='categories'
      label='Categories'
      value={formik.values.categories}
      error={formik.touched.categories && Boolean(formik.errors.categories)}
      helperText={formik.touched.categories && formik.errors.categories}
    />
  )}
  options={defaultBookCategories}
/>
<FormControlLabel
  control={
    <Checkbox
      id='isHidden'
      name='isHidden'
      value={formik.values.isHidden}
      onChange={formik.handleChange}
    />
  }
  label='Is hidden?'
  labelPlacement='end'
/>
<Button color='primary' variant='contained' fullWidth type='submit'>
```

```
            Submit
          </Button>
        </Stack>
      </form>
    </CardContent>
  </Card>
 </Container>
 )
}

export default BookCreationPage
```

src/containers/BookEditPage/index.tsx

```
import {
  Autocomplete,
  Button,
  Card,
  CardContent,
  CardHeader,
  Checkbox,
  Container,
  FormControlLabel,
  Stack,
  TextField,
} from '@mui/material'
import { unwrapResult } from '@reduxjs/toolkit'
```

```
import { useFormik } from 'formik'
import React, { useEffect } from 'react'
import { useParams } from 'react-router-dom'
import * as Yup from 'yup'

import { useAppDispatch, useAppSelector } from '../../app/hooks'
import { getBookByISBN, updateBook } from '../../features/book/bookSlice'
import { defaultBookCategories } from '../../utils/constants'
import { toastWrapper } from '../../utils/toastWrapper'

const validationSchema = () => {
  return Yup.object().shape({
    title: Yup.string().required('Title is required'),
    author: Yup.string()
      .required('Author name is required')
      .min(2, 'Author name must be at least 2 characters'),
    publisher: Yup.string()
      .required('Publisher name is required')
      .min(2, 'Publisher name must be at least 2 characters'),
    description: Yup.string().required('Description is required'),
    categories: Yup.array().of(Yup.string()),
    isHidden: Yup.boolean(),
  })
}

const BookEditPage: React.FC = () => {
  const dispatch = useAppDispatch()
```

```
const { bookByISBN, isLoading } = useAppSelector((state) => state.book)
const params = useParams<{ isbn: string }>()
const isbn = params.isbn
const currentBook = isbn ? bookByISBN[isbn] : undefined

const formik = useFormik<{
  isbn: string
  title: string
  author: string
  publisher: string
  description: string
  itemCount: number
  categories: string[]
  isHidden: boolean
}>({
  initialValues: {
    isbn: isbn || '',
    title: currentBook?.title || '',
    author: currentBook?.author || '',
    publisher: currentBook?.publisher || '',
    description: currentBook?.description || '',
    itemCount: currentBook?.items.length || 0,
    categories: currentBook?.categories || ['Fantasy'],
    isHidden: !!currentBook?.isHidden,
  },
  validationSchema,
  onSubmit: async (values) => {
```

```
      await toastWrapper({
        action: async () => {
          const result = await dispatch(updateBook(values))
          unwrapResult(result)
        },
      })
    },
  })

  useEffect(() => {
    if (isbn) {
      dispatch(getBookByISBN(isbn))
    }
  }, [isbn])

  useEffect(() => {
    if (currentBook) {
      formik.setValues({ ...currentBook, itemCount: currentBook.items.length })
    }
  }, [currentBook])

  return (
    <Container sx={{ marginTop: 8, marginBottom: 2, display: 'flex', justifyContent: 'center' }}>
      <Card sx={{ width: { xs: 1, sm: 400 } }} elevation={5}>
        <CardHeader title='Edit the book' sx={{ borderBottom: 1, borderColor: 'divider' }} />
```

```
<CardContent>
  <form onSubmit={formik.handleSubmit}>
    <Stack spacing={1}>
      <TextField
        fullWidth
        id='isbn'
        name='isbn'
        label='ISBN'
        type='text'
        value={isbn}
        disabled
      />
      <TextField
        fullWidth
        id='title'
        name='title'
        label='Title'
        type='text'
        value={formik.values.title}
        onChange={formik.handleChange}
        error={formik.touched.title && Boolean(formik.errors.title)}
        helperText={formik.touched.title && formik.errors.title}
        disabled={isLoading}
      />
      <TextField
        fullWidth
        id='author'
```

```
  name='author'
  label='Author'
  type='text'
  value={formik.values.author}
  onChange={formik.handleChange}
  error={formik.touched.author && Boolean(formik.errors.author)}
  helperText={formik.touched.author && formik.errors.author}
  disabled={isLoading}
/>
<TextField
  fullWidth
  id='publisher'
  name='publisher'
  label='Publisher'
  type='text'
  value={formik.values.publisher}
  onChange={formik.handleChange}
  error={formik.touched.publisher && Boolean(formik.errors.publisher)}
  helperText={formik.touched.publisher && formik.errors.publisher}
  disabled={isLoading}
/>
<TextField
  fullWidth
  id='description'
  name='description'
  label='Description'
  type='text'
```

```
  value={formik.values.description}
  onChange={formik.handleChange}
  error={formik.touched.description && Boolean(formik.errors.description)}
  helperText={formik.touched.description && formik.errors.description}
  disabled={isLoading}
/>
<TextField
  fullWidth
  id='itemCount'
  name='itemCount'
  label='Number of books'
  type='number'
  value={formik.values.itemCount}
  disabled
/>
<Autocomplete
  clearOnBlur
  multiple
  freeSolo
  filterSelectedOptions
  value={formik.values.categories}
  onChange={(_event, value) => {
    formik.setFieldValue('categories', value)
  }}
  renderInput={(params) => (
    <TextField
      {...params}
```

```
      fullWidth
      id='categories'
      name='categories'
      label='Categories'
      value={formik.values.categories}
      error={formik.touched.categories && Boolean(formik.errors.categories)}
      helperText={formik.touched.categories && formik.errors.categories}
    />
  )}
  options={defaultBookCategories}
  disabled={isLoading}
/>
<FormControlLabel
  control={
    <Checkbox
      id='isHidden'
      name='isHidden'
      value={formik.values.isHidden}
      onChange={formik.handleChange}
    />
  }
  label='Is hidden?'
  labelPlacement='end'
  disabled={isLoading}
/>
<Button
  color='primary'
```

```
            variant='contained'
            fullWidth
            type='submit'
            disabled={isLoading}
          >
            Submit
          </Button>
        </Stack>
      </form>
    </CardContent>
  </Card>
 </Container>
 )
}

export default BookEditPage
```

src/containers/BookListPage/index.tsx

```
import { Autocomplete, Box, Card, Container, Pagination, Stack, TextField } from '@mui/material'
import React, { useEffect, useState } from 'react'

import { useAppDispatch, useAppSelector } from '../../app/hooks'
import BookCard from '../../components/BookCard'
import LoadingWrapper from '../../components/LoadingWrapper'
import SearchBar from '../../components/SearchBar'
```

```
import { getAllBooks } from '../../features/book/bookSlice'
import { defaultBookCategories } from '../../utils/constants'

const BookListPage: React.FC = () => {
  const itemsPerPage = 5
  const dispatch = useAppDispatch()
  const { isLoading, books, totalCount } = useAppSelector((state) => state.book)
  const [currentPage, setCurrentPage] = useState<number>(1)
  const [search, setSearch] = useState<string>('')
  const [categories, setCategories] = useState<string[]>([])
  const numberOfPages = Math.ceil(totalCount / itemsPerPage)

  const handleChangePage = (_event: React.ChangeEvent<unknown>, value: number) =>
{
    setCurrentPage(value)
  }

  const handleChangeSearch = (value: string) => {
    setCurrentPage(1)
    setSearch(value)
  }

  const handleChangeCategories = (value: string[]) => {
    setCurrentPage(1)
    setCategories(value)
  }
```

```
useEffect(() => {
  dispatch(
    getAllBooks({
      startAt: itemsPerPage * (currentPage - 1),
      limit: itemsPerPage,
      categories,
      search,
    }),
  )
}, [currentPage, categories, search])

return (
  <Container sx={{ marginTop: 8, marginBottom: 2 }}>
    <Card elevation={5} sx={{ padding: 1 }}>
      <Stack spacing={1}>
        <SearchBar onSearch={handleChangeSearch} />
        <Autocomplete
          clearOnBlur
          multiple
          freeSolo
          filterSelectedOptions
          onChange={(_event, value) => {
            handleChangeCategories(value)
          }}
          renderInput={(params) => (
            <TextField
              {...params}
```

```
            fullWidth
            id='categories'
            name='categories'
            label='Categories'
            value={categories}
          />
        )}
        options={defaultBookCategories}
      />
    </Stack>
  </Card>
  <Box sx={{ display: 'flex', justifyContent: 'center', marginBottom: 2, marginTop: 2
}}>
    <Pagination
      count={numberOfPages}
      color='primary'
      page={currentPage}
      onChange={handleChangePage}
    />
  </Box>
  <LoadingWrapper isLoading={isLoading}>
    <Stack spacing={2}>
      {books.map((book) => (
        <BookCard key={book.isbn} book={book} />
      ))}
      <Box sx={{ display: 'flex', justifyContent: 'center' }}>
        <Pagination
```

```
            count={numberOfPages}
            color='primary'
            page={currentPage}
            onChange={handleChangePage}
          />
        </Box>
      </Stack>
    </LoadingWrapper>
  </Container>
 )
}

export default BookListPage
```