ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Система електронного замовлення та постачання ресурсів для агропідприємства»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Максим ЗЕЛІНСЬКИЙ
(підпис)

Виконав: здобувач вищої освіти групи ПД-43

Максим ЗЕЛІНСЬКИЙ

Керівник: Богдан ХУДІК
доктор філософії (PhD)

Рецензент: _____

КИЇВ - 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____Ірина ЗАМРІЙ
«_____» _____2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Зелінському Максиму Сергійовичу

1. Тема кваліфікаційної роботи: «Система електронного замовлення та постачання ресурсів для агропідприємства»
керівник кваліфікаційної роботи доктор філософії (PhD) Богдан ХУДІК,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 р. №51.

2. Строк подання кваліфікаційної роботи «29» травня 2026 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про управління логістичними процесами та ланцюгами постачання в аграрному секторі, методи автоматизації процесів електронного замовлення (B2B), технічна документація з описом фреймворків Spring Boot та Next.js, а також принципи побудови клієнт-серверної архітектури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):

1. Провести аналіз предметної області у сфері управління замовленням та постачанням аграрних ресурсів та існуючих програмних аналогів.

2. Сформулювати функціональні та нефункціональні вимоги до розроблюваної системи.

3. Здійснити обґрунтований вибір програмних засобів реалізації клієнт-

серверного веб-застосунку.

4. Спроєктувати програмну архітектуру системи, рольову модель доступу та структуру бази даних, представивши результати у форматі діаграм.

5. Реалізувати серверну та клієнтську частини системи відповідно до спроєктованої архітектури.

6. Провести комплексне тестування розробленого рішення на модульному та інтеграційному рівнях.

5. Перелік графічного матеріалу: *презентація*

1. Порівняльний аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Діаграма варіантів використання.
5. Архітектура системи.
6. Діаграма класів (Контролери, сервіси, репозиторії).
7. Діаграма ієрархії компонентів клієнтської частини.
8. Діаграма бази даних системи.
9. Екранні форми.
10. Апробація розробленого рішення.

6. Дата видачі завдання «23» лютого 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури.	23.02.-28.02.2026	
2	Аналіз та дослідження існуючих аналогів.	01.03.-07.03.2026	
3	Сформулювати функціональні та нефункціональні вимоги до розроблюваної системи.	08.03.-14.03.2026	
4	Здійснити вибір та обґрунтувати програмні засоби реалізації веб-застосунку.	15.03.-25.03.2026	
5	Виконати моделювання та проектування структури програми.	26.03.-18.04.2026	
6	Здійснити програмну реалізацію веб-застосунку та провести тестування розробленого рішення.	19.04.-28.04.2026	
7	Оформлення роботи: вступ, висновки, реферат	29.04.-05.05.2026	
8	Розробка демонстраційних матеріалів	06.05.-11.05.2026	
9	Попередній захист роботи	12.05.-22.05.2026	

Здобувач вищої освіти _____ Максим ЗЕЛІНСЬКИЙ
(підпис)

Керівник
кваліфікаційної роботи _____ Богдан ХУДІК
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 94 стор., 4 табл., 21 рис., 25 джерел.

Мета роботи – обґрунтувати та практично підтвердити шляхи автоматизації процесу управління замовленням та постачанням аграрних ресурсів за допомогою розробленого спеціалізованого клієнт-серверного програмного забезпечення з рольовою моделлю доступу та автоматизованим обліком матеріальних ресурсів.

Об'єкт дослідження – процес управління замовленням та постачанням аграрних ресурсів на агропідприємстві.

Предмет дослідження – методи та засоби побудови клієнт-серверного програмного забезпечення для автоматизації процесу управління замовленням та постачанням аграрних ресурсів.

Короткий зміст роботи: У роботі проаналізовано стан цифровізації аграрного сектору та існуючі програмні рішення, виявлено функціональні обмеження ринкових платформ для малого та середнього агробізнесу. Спроєктовано та реалізовано клієнт-серверну систему на основі Java та Spring Boot для серверної частини й Next.js для клієнтської частини, з реляційною базою даних PostgreSQL та системою автентифікації Keycloak за протоколом OAuth 2.0. Окрему увагу приділено проектуванню машини станів замовлення та рольовій моделі доступу для трьох категорій користувачів. Коректність роботи системи підтверджено комплексним тестуванням на модульному та інтеграційному рівнях.

Сферою використання додатку є автоматизація процесів закупівлі та координації постачання аграрних ресурсів на підприємствах в агропромисловому комплексі.

КЛЮЧОВІ СЛОВА: АВТОМАТИЗАЦІЯ ПРОЦЕСІВ, АГРОПІДПРИЄМСТВО, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА, JAVA, SPRING BOOT, NEXT.JS, REST API, OAUTH 2.0, KEYCLOAK, POSTGRESQL, DOCKER.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API - Application Programming Interface.

B2B - Business-to-Business.

CORS – Cross-Origin Resource Sharing.

CRUD - Create, Read, Update, Delete.

DTO - Data Transfer Object.

ERP - Enterprise Resource Planning.

HTTP / HTTPS - Hypertext Transfer Protocol / Secure.

IAM – Identity and Access Management.

IDE – Integrated Development Environment.

IoC – Inversion of Control.

JPA - Jakarta / Java Persistence API.

JPQL - Java Persistence Query Language.

JSON - JavaScript Object Notation.

JWT – JSON Web Token.

MoSCoW – Must have, Should have, Could have, Won’t have.

MVC – Model-View-Controller.

MVP – Minimum Viable Product.

ORM – Object-Relational Mapping.

PKCE – Proof Key for Code Exchange.

REST – Representational State Transfer.

SaaS – Software as a Service.

SQL – Structed Query Language.

SSR – Server-Side Rendering.

UML – Unified Modeling Language.

URI / URL – Uniform Resource Identifier / Locator.

XML – eXtensible Markup Language.

СУБД – Система управління базами даних.

ВСТУП

Актуальність теми: Сільське господарство сьогодні залишається одним з головних фундаментів української економіки, забезпечуючи в межах 10-11% загального ВВП країни, паралельно закриваючи понад 59% загальних обсягів поставок української продукції на зовнішні ринки. Водночас, згідно з аналітичними дослідженнями McKinsey Global Institute сільське господарство залишається в списку найбільш нетехнологічних галузей. Через відсутність єдиного простору для обміну інформацією між постачальниками та внутрішніми командами, узгодження замовлень часто проходить вручну. Дані доводиться переписувати з одного файлу в інший - цей процес займає третину робочого дня, а то й більше. Аналіз таких систем, як SoftFarm, BAS АГРО ERP чи SAP S/4HANA, показав те, що вони спеціалізуються лише на полях і завданнях агрономів. Процедури погодження документів із зовнішніми покупцями в них взагалі не передбачені. Тим часом класичні ERP-платформи складні для сприйняття та витягують з бюджету значні суми, через що малі та середні агрофірми уникають їхнього впровадження. Ось чому створення легкого інструменту для онлайн-укладення договорів тепер виглядає не просто доречно, а неминуче.

Об'єктом дослідження є процес управління замовленням та постачанням аграрних ресурсів на агропідприємстві.

Предметом дослідження є методи та засоби побудови клієнт-серверного програмного забезпечення для автоматизації процесу управління замовленням та постачанням аграрних ресурсів.

Метою роботи є автоматизація процесу управління замовленням та постачанням аграрних ресурсів за допомогою розробленого спеціалізованого клієнт-серверного програмного забезпечення з рольовою моделлю доступу та автоматизованим обліком матеріальних ресурсів.

Методи дослідження: Системний підхід допоміг глибше зрозуміти ринок - паралельно перевірялись головні функції конкурентів. Користувацькі сценарії виникають не з переліку бажань, а через картування реальних шляхів взаємодії; їх

важливість оцінювалась за методом MoSCoW. Дані структурувались за допомогою UML-елементів, тоді як логіку переходів замовлення описували мовою станів. Spring Boot став основою серверної частини, попри це внутрішній код спирається на класичні конструкції Java. Натомість стандартного JavaScript - фронтенд пише TypeScript разом із Next.js. Усе важливе зберігається в PostgreSQL, що стежить за формою даних і не губиться під тиском. Доступ забезпечує Keycloak, адже вміє працювати з OAuth 2.0 та видавати JWT. Кожна версія живе всередині Docker-контейнерів, тому поведінка не плутається між машинами. Перевірку роблять пошарово: одиниці коду аналізують JUnit 5 і Mockito, а цілі потоки - через Testcontainers.

Наукова новизна роботи проявляється в побудові особливої системи, здатної автоматично встановлювати початковий стан документа на основі фактичних залишків на складах. Центральне ядро керує етапами погодження таким чином, щоб будь-які непередбачені перестановки статусів було неможливим. З іншого боку, права доступу регулюються дуже точно: контроль активний прямо на рівні окремих точок входу в інтерфейс. У порівнянні з схожими рішеннями, тут архітектура спрямована на повний цикл B2B-угод. При цьому враховано специфіку аграрних компаній, тому запуск не забирає багато коштів - вся інфраструктура йде в Docker-контейнерах. Додатково надається вільний доступ до REST API для приєднання ззовні.

Практична значущість полягає у створенні програмного рішення, яке буде забезпечувати замість хаосу в листуванні - обговорення між керівництвом, спеціалістами та постачальниками, де все йде за правилами. Цифрова платформа цілком витіснила паперову нерозсудливість, де кожна дрібниця має шанс загубитись назавжди. Інформація доходить до потрібних систем через REST API, тож більше немає потреби переписувати її вручну з одного місця в інше. Навіть якщо стара система ще працює, її залишають у спокої - новий компонент працює фоново, не порушуючи звичної роботи.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Актуальність і значення цифрової автоматизації в агропромисловій сфері

В реаліях того, як світовий ринок стрімко переходить на цифрові технології, процес якого різко набрав обертів починаючи з 2010-х років, аграрний сектор демонструє певний парадокс. Залишаючись надзвичайно необхідним для підтримання стабільного постачання продуктів харчування та притоку стабільних грошових надходжень - паралельно з цим помітно великий розрив, порівняно з іншими сферами бізнесу, як за рівнем технічного та цифрового розвитку, і того, наскільки кваліфіковано використовуються сучасні програми. Як показує статистика, сільське господарство залишається одним із найповільніших секторів, які переходять до цифровізації на глобальному ринку - залишаючись на останньому місці серед основних сфер бізнесу, за сукупним індексом цифровізації McKinsey Global Institute [1]. Подібне гальмування викликане низкою внутрішніх причин - величезною територіальною віддаленістю інфраструктурних об'єктів компанії, колосальною нерівномірністю у плані технічного забезпечення між великими корпораціями та дрібними фермерами, а також звичним скептицизмом представників цього бізнесу, щодо будь-яких реформ у керуванні процесами.

Для України, агропромисловий комплекс якого припадає близько 10-11% від усього обсягу економіки країни та забезпечує від 41% (до повномасштабного вторгнення) до 59-62% (у період 2023-2024 років) сумарної частки українського експорту продукції, брак сучасних технологій у сільського господарства стає проблемою. Спостерігається величезний розрив впровадження новітніх технологій між окремими представниками сільськогосподарського сектору. Найбільші агропромислові компанії, що керують величезними площами землі, прогнозовано почали масово впроваджувати технологічні інновації у виробництві. Починаючи

від сервісів супутникового трекінгу сільськогосподарських культур та безпілотної агроавіації, до комплексних програм управління підприємством, які автоматизують логістичні потоки, грошову звітність компанії та комунікацію з постачальниками. Проте представники невеликого агросектору, які складають основу українських сільгоспвиробників, досі переважно використовують паперові записи та телефонні переговори, при оформленні заявок та організації логістики. Це призводить до взаємопов'язаних управлінських помилок і фінансових збитків.

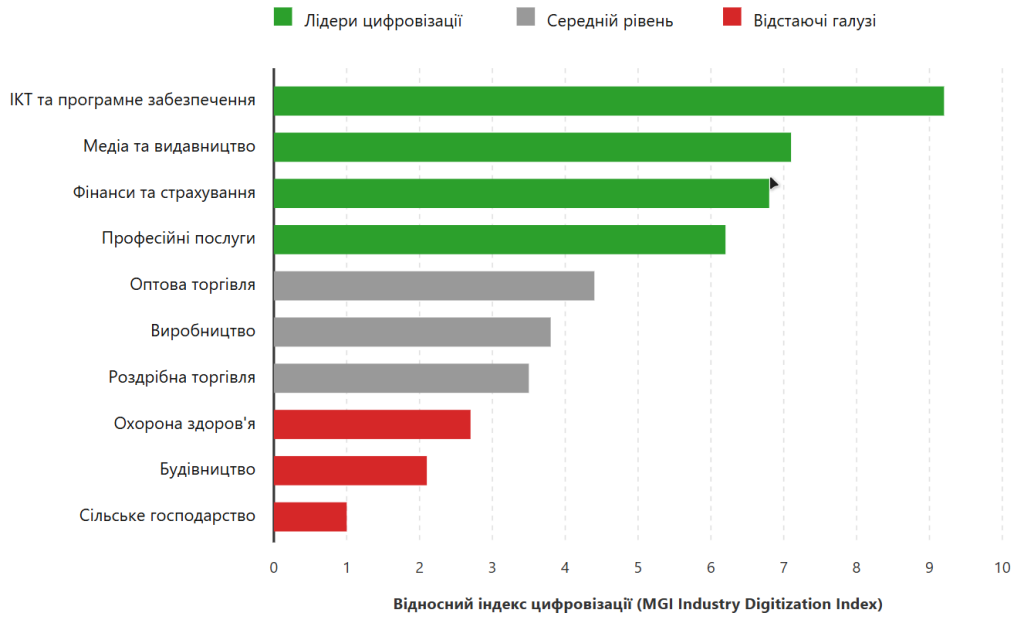


Рис. 1.1 Індекс цифровізації галузей McKinsey (Industry Digitization Index)

Підхід погодження заявок по телефону та ручного обліку в журналах, що все ще залишається звичною справою у багатьох українських виробників сільгоспродукції, тягне за собою типовий комплекс потокових труднощів, де кожен фактор працює як суттєва перешкода для нормальної роботи. Але коли ці фактори накладаються один на одного, вони створюють значно складнішу загрозу для ефективного контролю постачання всього необхідного для сільського господарства. Ключовим і найбільш вагомим чинником, що ламає всю логіку процесу, виступає повна відсутність централізованої електронної платформи закупівельних операцій. У результаті чого стає неможливим оновлювати дані в реальному часі щодо поточного стану кожного замовлення для всіх учасників

процесу. Ще один серйозний мінус полягає в тому, що постійний ризик людського фактора під час ручного переносу інформації між розрізненими базами обліку, що за наявності широкого переліку продукції (від посівного матеріалу та сортового насіння, до міндобрив, отрутохімікатів, пального та запчастин) перетворюється на закономірність, замість поодиноких випадків. Додатковим слабким місцем є відсутність чіткої хронологічної стрічки комунікацій між учасниками постачального ланцюга [2]. У результаті чого неможливо проаналізувати історію роботи з партнерами та зібрати нормальну статистику щодо закупівель, спираючись на історію попередніх періодів.

Фінансові втрати від використання ручного ведення роботи за старими схемами при забезпеченні господарства матеріалами, мають два абсолютно різних вектори, але пов'язані один з одним. Очевидні збитки виникають через неточні розрахунки обсягів необхідного постачання, що виливається або в накопичення надмірних залишків на складах, які потім втрачають вартість (що особливо небезпечно для агрохімії та міндобрив, оскільки хімічний склад псується через погані умови на складі). Або до гострої нестачі ключових найнеобхідніших матеріалів під час найважливіших етапів сезону (коли треба сіяти чи терміново косити) - що безпосередньо призводить до зниження фінального збору. Побічні витрати пов'язані з тим, що компанія марнує сили та час своїх штатних співробітників. Аналіз продуктивності функції менеджменту у секторі сільського господарства свідчить про те, що фахівці з постачання, які забезпечують господарство матеріалами "вручну", втрачають від 35% до 55% робочого часу на процеси узгодження, які є неефективними. Серед них ручне з'ясування деталей постачання у телефонному режимі, ручний перегляд паперових списків для контролю заявок, пошук даних із хаотичних журналів обліку та постійне коригування неправильно заповнених документів.

Без цифрової платформи для керування закупівлями аграрна компанія губить переваги перед тими, хто постачає матеріали. Ті, хто вже співпрацювали з клієнтами, що використовують технічні рішення, часто пропонують кращі умови там, де процеси ясні, легкі у підтримці й не потребують зайвої роботи. Там, де

немає чітких механізмів контролю замовлень, важче домовитися про стабільні договори на довгий термін. З іншого боку, саме всередині організації цей недолік дає себе знати ще чіткіше - коли справи починають йти краще, логістика стрибає у складності набагато швидше, ніж обсяги, а відсутність автоматики призводить до раптового стрибка витрат.

Виявлені обставини разом дають міцну основу для створення автоматизованої платформи замовлення й доставки продуктів аграрного призначення. Окрім того, потрібно переосмислити саме те, як закупівлі входять до цифрового середовища - шляхом переходу на повністю електронний формат реєстрації запитів. Замість телефонних згод тепер можна фіксувати кожен крок виконання прямо в системі. Комунікація, що раніше проходила сторонніми каналами, стане частиною оформленого діалогу навколо окремої заявки. Платформа із чітким поділом прав доступу та видимими етапами руху документа створить технологічну основу, за якою значно поліпшиться результативність забезпечення господарств необхідними матеріалами.

1.2 Еволюція цифрових рішень для управління ресурсами агропідприємств та аналіз аналогічних програмних рішень

Якщо подивитися, яким чином змінювалися цифрові програми управління ресурсами в агропідприємстві за останні лише тридцять років, важливо зазначити, що ця еволюція являю собою здебільшого основну логіку розвитку фірмових інформаційних технологій. Разом з цим аграрна галузь має власний темп інноваційних змін, обумовлений сезонністю та характерних рис виробництва. Фіктивно поділяють чотири покоління систем, які виникли під дією певних директивних запитів і технологічного ресурсу свого часу.

Перше покоління, яке охоплює період приблизно з початку 1990-х до середини 2000-х років, базувалося на локальних інструментах обліку, чий функціонал обмежувався вирішенням специфічних задач рахункових операцій та фіксації майна на складах без можливості зв'язати їх з поточним менеджментом підприємства. Архітектурна модель тих років будувалася як локальне сховище

даних на одній машині, без мережевої взаємодії - технічно не залишивши жодних шансів для дистанційного доступу, командної роботи кількох людей одночасно, та синхронізації робочих процесів різних служб. Для вітчизняного агробізнесу того часу, найвідомішим інструментом того часу була система 1С та його спеціалізовані агромодулі, які забезпечували виконання стандартних функцій ведення бухгалтерії. Однак від початку не була розрахована на синхронізацію з живим керуванням поточними операціями.

Друге покоління (умовно з 2005 по 2015 рр.) сформувалося в епоху появи масштабних аграрних холдингів, які вимагали контроль над багаторівневим постачальницьким ланцюжком у межах територіально віддалених підприємствах і полях. Серед інструментів такого рівня виділялися платформи SAP та Oracle, що домінували на світовому ринку. А також галузеві пакети BAS АГРО на локальному українському ринку. Вони запропонували принципово новий набір можливостей за своїм функціоналом - інтеграцію фінансових звітів та складського майна, контролю логістичних ланцюжків та кадрову службу усередині єдиного цифрового простору. Однак загальні витрати на організаційні зусилля для запуску подібних платформ був занадто високим, адже вартість ліцензій, занадто довгі терміни запуску системи в роботу та постійна потреба в залученні IT-фахівців - практично обмежував їх використання великими промисловими об'єднаннями, залишаючи підприємства середньої ланки за бортом технологічного прогресу.

Третє покоління (приблизно з 2015 по 2020 рр.) сформувалася на тлі бурхливого розвитку хмарних технологій та концепції SaaS. Через підписку, замість придбання обладнання, аграрні платформи типу Agrivi чи Cropwise Operations ставали доступними навіть малим господарствам. SoftFarm та подібні їй пропонували модулі управління без необхідності тримати внутрішню IT-інфраструктуру. Фінансовий поріг входження значно скоротився - тепер фермерам не довелося вкладатися в дата-центри. Хоч ці рішення добре охоплювали задачі агрономів: створення карт полів, аналіз стану рослин, планування удобрення. Але ось питання забезпечення господарства матеріалами часто лишили поза увагою розробників. У результаті, процеси закупівлі, контролю поставок, взаємодії з

постачальниками залишились слабко автоматизованими.

З четвертого покоління, що набирало силу з 2020-го, все будують навколо трьох важелів - орієнтація на API, мікросервіси й пріоритет мобільних пристроїв. Розробка через API спрямована таким чином, що система одразу має чіткий цифровий вихід для обміну даними, тож сторонні системи, як аналітика чи бухгалтерія, підключаються без переписування коду всередині. Хоча мікросервіси складніші у нагляді, кожен блок можна окремо масштабувати, коли потрібно - наприклад, підвищити продуктивність перед посівною. Пріоритет мобільних пристроїв тут не просто доповнення, бо багато фахівців у полі працюють лише через смартфон, адже стаціонарний ПК недоступний, а інколи й нема де зарядити.

Рис. 1.2 Еволюція поколінь інформаційних систем для агропідприємств (~1995-2024+)

Працювати із сільськогосподарською логістикою - значить переосмислювати основи побудови цифрових систем для всього сектору. Замість стабільного попиту тут стрічка пікових навантажень: саме перед посівним сезоном або збиранням врожаю фермери одночасно купують добрива, насіння й хімікати. У такі моменти платформа має витримувати потік запитів, не сповільнюючись і не допускаючи збоїв у обробці замовлень. Відстані між пунктами виробництва, базами зберігання, технічними центрами й полями часто великі - це додає складності організації

процесів. Тому доводиться заходити на платформу там, де з’єднання ледь тримається. З-за цього код доведеться переробляти так, щоб все працювало навіть при критично низькій швидкості. Особливості товарів створюють окремий клопіт - одне ллється потоками, інше важать з точністю до долі грама. Важко уявити, як вантажівки та мікродози живуть в одному інтерфейсі без хаосу. Насправді система має сама розбиратися, коли треба округлювати, а коли враховувати кожен елемент. Інакше користувач опиниться між реєстрами, що суперечать один одному. Гнучкість тут не просто бажана - вона єдиний спосіб уникнути помилок. Платформа повинна перемикатися між масштабами, немов нічого особливого не сталося.

Не кожен інструмент для керування господарством виявляється дійсно ефективним - серед них шукати доводиться ретельно. Потреба у практичному рішенні стає очевидною тоді, коли треба щось налаштувати, підключити або отримати допомогу. Вибір пав не на загальний огляд, а на конкретні платформи - чотири за ліком. Деякі створено в Україні, інші приїхали ззовні, маючи різну глибину функціоналу. Серед назв - SoftFarm, Odoo, BAS АГРО та SAP S/4HANA. Перевагу надано сценаріям закупівель типу B2B, адже саме цей напрямок тепер визначає основу тестування. Навіть незначні недоліки в цьому модулі можуть знецінити переваги. Порівнювати доводиться не за словами продавців, а за реальними результатами вимірювань.

Найближчим аналогом до запланованого рішення серед наявних стає вітчизняна розробка - SoftFarm. З його допомогою можна контролюватати господарську діяльність, враховуючи постачання та закупівлі. Однак глибший аналіз функціоналу викриває чимало недоліків. Для прикладу, від моменту подання заявки до фіксування отримання товару немає автоматичного супроводження. Крім того, у кожному окремому замовленні відсутні коментарі всередині замовлення. Комунікація менеджера з постачальником відривається від основної системи, через це дані опиняються поза загальним ланцюгом. Навіть якщо платформа базується в хмарі, без наявності API обмін інформацією з іншими інструментами стає неможливим.

Відкриту ERP-систему Odoo створили в Бельгії. Цей продукт існує у двох версіях - Community і Enterprise. Її активно використовують малі й середні компанії скрізь по світу, також і в Україні. Модульна структура робить систему особливою. Потрібні блоки додаються окремо: скажімо, Inventory допомагає слідкувати за запасами, а Purchase контролює закупівлі. Коментарі до документів додаються через спеціальний описовий інструмент. Доступ до операцій регулюється точково, залежно від ролі користувача. Хоча Odoo має широкий функціонал, для сільського господарства він працює швидше як загальний помічник - без глибокої адаптації під галузь. Замість автоматичної зміни статусів замовлень на базі фактичних запасів система вимагає ручного контролю. Резервування матеріалів під окремі потреби не має спеціалізованих інструментів, що ускладнює планування. Постачальники ззовні не отримають свого простору з контролем доступу до інформації. Один із важливих недоліків - технічна логіка запуску. Навіть коли потрібні лише кілька модулів, доводиться піднімати всю платформу разом із навколишньою інфраструктурою. Для невеликої компанії такий підхід часто створює більше проблем, аніж користі.

Систему BAS АГРО створили як повноцінний інструмент управління ресурсами, яка працює як окрема вертикальна надбудова на основній платформі BAS. Весь функціонал продукту повністю враховує специфіку українських законів, що критично важливо для щоденного заповнення податкових та бухгалтерських відомостей. З-поміж усіх вітчизняних розробок, дана система має максимально розгалужену базу функцій - управління замовленнями, складський облік, фінансове впорядкування, а також закриває завдання бухгалтерії по зарплатах та специфічного обліку сільгоспвиробництва. Але з таким універсальним набором можливостей, створює додаткові проблеми. Процес розгортання платформи вимагає тривалого налаштування через необхідність тонкого шліфування всіх алгоритмів під потреби конкретного підприємства. До того ж компанії доведеться постійно тримати на зв'язку сторонніх програмістів або профільні агенції, які будуть супроводжувати її щодня. Оскільки архітектура продукту є закритою, підключення до нього сторонніх сервісів перетворюється на проблему. Якщо

додати регулярні платежі за користування та витрати на запуск комплексу (що вимірюється сумами від кількох десятків до сотень тисяч гривень, враховуючи обсяги бізнесу) - стає зрозуміло, чому середні агропідприємства або локальні фермери, відмовляються від системи.

Якщо брати систему SAP S/4HANA, то це всеохоплююча платформа планування ресурсів, яку розробив лідер ринку бізнес-програмного забезпечення. Функціонал системи розрахований на паралельне ведення всіх процесів у великих корпораціях, куди входять інструменти для закупників, графіки відвантажень та запусків цехів, а також складного транспортування та глибокого фінансового і податкового обліку. За рівнем технічного потенціалу, платформа демонструє абсолютне лідерство у порівнянні з альтернативними програмними комплексами - архітектура дозволяє проводити складні обчислення та стеження у реальному часі, усі внутрішні компоненти за замовчуванням безперебійно пов'язані між собою. Система спокійно забезпечує взаємодію багатьох компаній чи філій в одному кабінету, а також має потужну базу відкритих шлюзів для підключення будь-якого зовнішнього сервісу. Але якщо спуститися на рівень середнього агробізнесу або локальних ферм, стає зрозуміло, що ця система для них є відвертим перебором. Адже загальні витрати на запуск (що за стандартами ринку вимірюються від сотень тисяч і легко доходить до мільйонів доларів), сам процес налаштування (що забирає від 12 до 36 місяців), жорсткі вимоги до кваліфікації робітників (яким доведеться довго вчитися користуватися програмою), і до того ж залежність від дорогих акредитованих інженерів та консультантів - автоматично викреслюють цю систему зі списку можливих варіантів для локального підприємства.

Результати порівняльного аналізу наведених рішень за ключовими функціональними та операційними критеріями продемонстровано у таблиці 1.1.

Таблиця 1.1

Порівняльний аналіз програмних рішень для управління ресурсами агропідприємств

Критерій	BAS АГРО ERP	SAP S/4HANA	SoftFarm	Odoo	Розроблювальна система
Автоматизація переходів між статусами замовлення	-	+	-	-	+
Можливість коментування замовлень	-	+	-	+	+
Розмежування прав за ролями	+	+	+	+	+
Окремий доступ для зовнішніх постачальників	+	+	-	+	+
Відкритий REST API для інтеграцій	-	-	-	+	+
Спосіб розгортання	Локально	Хмара / Сервер	SaaS	Хмара / Локально	Docker Compose

Перегляд таблиці показав недоліки діючих платформ. У місцях, де має бути співпраця селянського господарства і постачальника, процеси зміни статусів замовлень не підтягнуті до автоматики - ані SoftFarm, ані Odoo, ані BAS АГРО не прив’язують запаси на складі до стану заявки. Натомість усе разом тримати, люди копіюють інформацію через сторонні сервіси. Такий розрив призводить до плутанини: можливість обговорювати замовлення всередині системи є головно в масштабних комплексах типу SAP, які для невеликих фермерств залишаються поза межами доступності. Якщо домовленості летять у різних чатах і поштах, неточності трапляються регулярно. Однак складні платформи, як-от BAS АГРО або SAP S/4HANA, даються з трудом - переважно дрібним фермерським господарствам. Їх довге запускання разом із потужною технічною базою ускладнює використання малим бізнесом. Вони просто не завжди пасують за масштабами.

Виявлені технічні недоліки стають головним аргументом на користь

створення індивідуального рішення. Розроблювана система будується на принципово іншій логіці, порівняно з традиційними аналогами, яка фокусується на одному завданні - контролю всіх етапів кожної стадії замовлення на закупівлю (від створення запиту фахівцем до погодження постачальником), з низьким порогом входження та не вимагаючи великих інвестицій на старті.

1.3 Архітектурні підходи до побудови інформаційної системи

Вибір архітектурного напряму до формування інформаційної системи є необхідним рішенням, що чітко окреслює її гнучкість інфраструктури, швидкості роботи, бюджет на подальше обслуговування та потенціал для подальшого розвитку у відповідь на постійні зміни в критеріях бізнес-логіки проекту. У практиці створення серверного компоненту, найчастіше розглядають три архітектурні моделі, де кожен підхід має власне співвідношення між простотою, гнучкістю та витратами на адміністрування - монолітна, клієнт-серверна, а також мікросервісна архітектури.

Монолітна архітектура орієнтована на запуск повного обсягу функціональних можливостей платформи у формі єдиного збірного пакету (як у вигляді компільованого файла або веб-застосунку, який інтегрує всі інфраструктурні елементи) - інтерфейс взаємодії, бізнес-логіку та рівень персистентності. Перевагами використання монолітної структури є спрощена реалізація базового функціоналу на ранніх стадіях життєвого циклу, виключенні додаткових витрат на організацію міжмодульної взаємодії та максимально спрощеній процедурі впровадження системи. Проте для систем управління агробізнесом, монолітна структура демонструє низку фундаментальних недоліків. Наприклад, обмеження щодо гнучкого адаптування потужностей окремо взятих сервісів під час критичних сезонних сплесків активності, уповільнення подальшої підтримки коду (оскільки компоненти занадто сильно переплітаються між собою). Через це втрачається сумісна робота над реалізацією різних частин бізнес-логіки.

Мікросервісна архітектура, яка репрезентується фрагментація системи на

комплекс незалежних впроваджуваних і адаптивних сервісів [4], які відповідають за спеціалізовану функціональну область - це технологічний еталон у розрізі горизонтального розширення та експлуатаційної гнучкості. З іншого боку, впровадження мікросервісного підходу для проектів низької та помірної складності, стає занадто витратною стратегією. Критично ускладнюється інфраструктурна підтримка та експлуатація, порушується координація розподільних операцій, збільшуються накладні витрати на взаємодію між сервісами та підтримку цілосності даних у розрізі всієї системи.

Для проектованої екосистеми цифрового постачання аграрних ресурсів, затверджено в основу системи клієнт-серверний підхід, оскільки цей підхід забезпечує найкращий баланс функціональних параметрів, технологічною складність впровадження та операційними витратами на супровід системи. Серверний компонент побудовано на суровому розділені архітектурних рівнів. Один з них, рівень контролерів, який відповідає за валідацію вхідних HTTP-запитів та їх маршрутизацію. Інший, рівень сервісів - який містить ключові алгоритми бізнес-процесів. Додатково, рівень репозиторіїв, який організовує обмін даними з реляційним сховищем даних за допомогою ORM-технологій. Зазначена структура забезпечує прозоре розмежування зон відповідальності компонентів, полегшує ізольоване тестування кожного рівня та залишає можливість для гнучкого трансформування моноліту в мікросервісний шаблон (Коли виникне необхідність масштабувати лише певні функціональні вузли).

REST являє собою архітектурним напрямом в побудові API систем [3], яка задовольняє взаємодію між серверним та користувацьким компонентам завдяки протоколу HTTP. Стиль напряму формуються на ключових принципах, які спираються через фіксовану взаємозалежність (коли модель системи з використанням користувацьких та серверних компонентів утворюється без збереження стану сеансу на сервері), інтегрований інтерфейс, багаторівневу систему, можливість кешування, ініціалізацію за запитом. Застосування необхідного API у проектованій системі задовольняє незалежність серверних та користувацьких компонентів. У перспективі будь-який тип клієнта (мобільна

програма, веб-застосунок та інші), можуть співпрацювати з системою завдяки єдиному інтегрованому інтерфейсу, без змін серверної логіки.

1.4 Специфікація функціональних вимог та призначення системи

Створений інструмент - це спеціалізований веб-сервіс, побудований за перевіреною схемою клієнт–сервер. Все його призначення обертається навколо одного процесу: супроводження заявок на сільськогосподарські товари від першої ідеї до фінального затвердження постачальником. Там, де звичайні великі системи розпорошують увагу між безліччю можливостей, тут все сконцентровано на одному напрямі - забезпеченні господарств матеріалами. Цей підхід знімає зайве навантаження, тож користувач отримує лише те, чим користується кожного дня. В основі ідеї - спільний цифровий простір, де всі учасники логістики користуються одним джерелом інформації. Кожен тип доступу має власні межі, які не накладаються один на одного. Завдяки цьому координація поставок уже не плутанина. Вона перетворюється на зрозумілий і стабільний процес для компанії.

Для підтримки архітектурної цілісності системи потрібно визначити функціональні рамки (або окреслити), які процеси належать під обов'язок системи, а які залишаються поза нею. Ці данні знаходяться у віданні системи: нагляд за учасниками системи і відповідних повноважень із залученням стороннього серверу автентифікації, ведення баз даних ресурсного забезпечення спираючись на парадигму логічного вилучення, управління базою постачальників із впровадженням тимчасового відкликання активного стану, управління ланцюжками забезпечення з обов'язковою верифікацією переходів між етапами, спираючись на системні обмеження предметної області, забезпечення комунікації серед авторизованих суб'єктів системи (яке передбачає запис до хронологічного реєстру подій) виведення статистичних оцінок (спираючись на масиви агрегованої статистики руху ресурсів) для оптимізації процесів прийняття керівних рішень. Поза межами проєкту залишаються процеси управління грошовими потоками та врегулювання взаємних зобов'язань, ведення бухгалтерського обліку, завдання

агрономічного сектору (цифрове стеження за полями, контроль етапів розвитку посівів), синхронний контроль матеріальних запасів, управління внутрішньою логістикою.

Збирання потреб виконане завдяки картах користувацьких історій разом із пріоритезацією MoSCoW [20] - цей шлях допоміг знайти ключові елементи взаємодії й виділити основну функціональність. Заявки самі по собі не живуть, їх хтось активує. Першим діє менеджер - готує замову, слідкує за станом справ. Потім процес передається адміністратору, адже контроль стримує хаос серед поставок. Замість того щоб чекати, він одразу починає сортувати запити між партнерами, коригувати основну інформацію, аналізувати показники за допомогою особливої панелі. По суті - три види прав доступу, усі з окремими маршрутами, перетинів не передбачено. Діяльність постачальника активується строго після отримання завдання. Потім слідує самостійне повідомлення про доставку. У доступі до чужих замовлень немає потреби - такий порядок діє без усяких відступів. Саме так забезпечується приховування деталей кожної операції.. Взаємодія акторів з варіантами використання системи наведена на рисунку 1.3.

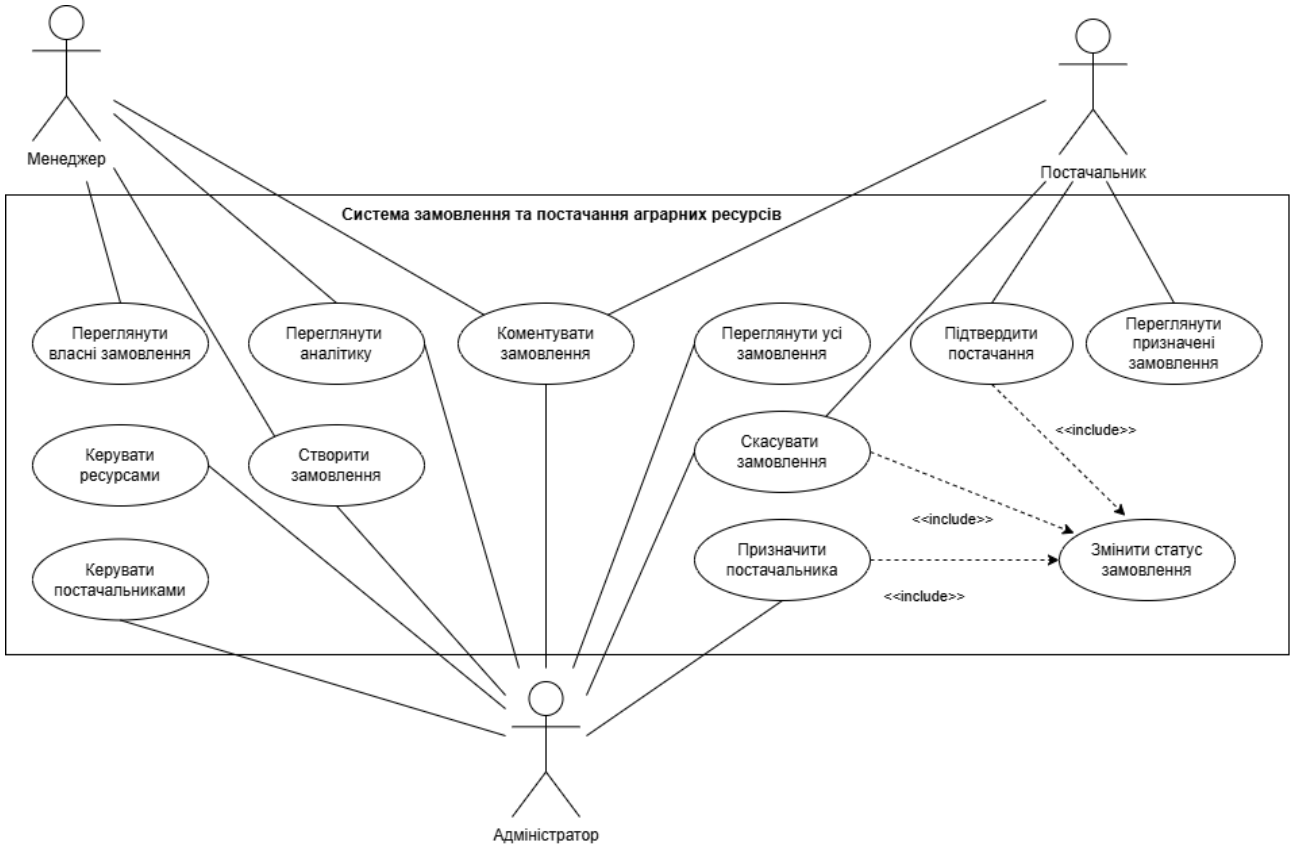


Рис. 1.3 Діаграма варіантів використання системи (Діаграма Use Case)

Базовий механізм спілкування з платформою задають функції, загальні для кожної ролі. Замість того щоб тримати паролі всередині, система отримує підтвердження особистості через сторонній сервіс перевірки. Перелік активних ресурсів і постачальників має бути видимим, бо оновлюється автоматично - користувач просто бачить те, що є наразі. Розробники обов’язково передбачають фільтри за часом і станом, коли йде мова про список заявок - це допомагає не блукати поміж старими записами. Кожна заява приймає коментар прямо на своїй сторінці, тож не доводиться шукати листи чи скролити чати. Повідомлення лишаються там, де вони найпотрібніші - поряд із даними.

З позиції менеджера платформа пропонує два основні функціонали. Створення замовлення - перший з них: обирається потрібний матеріал, кількість і час доставки, адже саме він формує запит сільськогосподарської діяльності. Аналітичний огляд - другий напрям: через нього видно прогрес кожного запиту. Певні пункти потребують коригування або нагадують про себе статусом. Такий погляд допомагає тримати процеси під контролем без зайвих повідомлень. Весь стан ланцюга поставок перед очима, коли треба приймати рішення.

Адміністратор виступає координатором усього процесу. Він тримає все на контролі, особливо щодо списку матеріалів: додає нові, міняє старі, ховає ті, що більше не потрібні. Точно так само працює з базою постачальників, але тут ще й може повернути того, кого вже відкладали в архів - бо співробітники компанії час від часу з’являються знову. Замовлення створювати теж може, хоча це не обов’язок, а лише запасний шлях, коли потрібно оформити заявку одразу згори. А ось прикріпити постачальника до конкретного замовлення - справа важлива, бо саме цим кроком запускається вся перевезення далі. Замість того щоб просто спостерігати, адміністратор має оновлювати статус замовлень - це працює за логікою станів. Інколи йому доводиться закривати чи викидати замовлення, якщо все пішло не так. Без доступу до аналітики йому важко слідкувати за логістикою; без них важко буде брати потрібне рішення про поставки взагалі.

У системі постачальник займається лише тими завданнями, що потрапили

прямо до нього. Принцип мінімуму прав тут не просто правило - це основа для захисту бізнес-зв’язків фермерського господарства з партнерами. Завершення своїх замовлень із підтвердженням доставки входить у число його можливостей, адже саме цей крок стає сигналом: матеріали вже на місці. Якщо ж реалізація поставки раптом стає під питанням, скасування замовлення також допустиме - але обов’язково з чітким поясненням причин. Розподіл прав доступу між ролями представлений у таблиці 1.2.

Таблиця 1.2

Розмежування прав доступу за ролями користувачів

Операція	Менеджер	Адміністратор	Постачальник
Автентифікація в системі	+	+	+
Перегляд ресурсів	+	+	+
Управління ресурсами (CRUD, архівування)	-	+	-
Перегляд постачальників	+	+	+
Управління постачальниками (CRUD, відновлення)	-	+	-
Перегляд замовлень та фільтрація	+	+	+
Створення замовлення	+	+	-
Призначення постачальника до замовлення	-	+	-
Зміна статусу замовлення	-	+	-
Завершення замовлення	-	+	+
Скасування замовлення	-	+	+
Додавання коментаря	+	+	+
Перегляд аналітичного дашборду	+	+	-

Кожне замовлення має п'ять статусів, переходи між якими чітко визначені логікою системи. Одразу після створення заявки, система автоматично аналізує залишки для визначення її початкового стану. Якщо матеріалів достатньо - система здійснює резервування матеріалів та переміщує замовлення до статусу очікування на виконання. За відсутності потрібної кількості, замовлення набуває статусу очікування на постачання, надсилаючи повідомлення адміністратору про необхідність дозамовлення. Внесення постачальника дає старт логістичним

операціям, що змінює статус "у процесі доставки" без потреби в ручному схваленні. Після завершення поставки товару та підписання документів, заявка успішно закривається зі статусом "виконано". Передбачено також інший можливий результат "скасованого" замовлення, що припиняє процес раніше строку, на будь-якому поточному статусі. У разі скасування, система самостійно повертає усі замовленні ресурси до складу. Обидва кінцеві статуси є незворотними, що автоматично закриває доступ до коригування даних після архівації заявки. Дозволені шляхи переміщення між станами наочно розписано по кроках на рисунку 1.4.

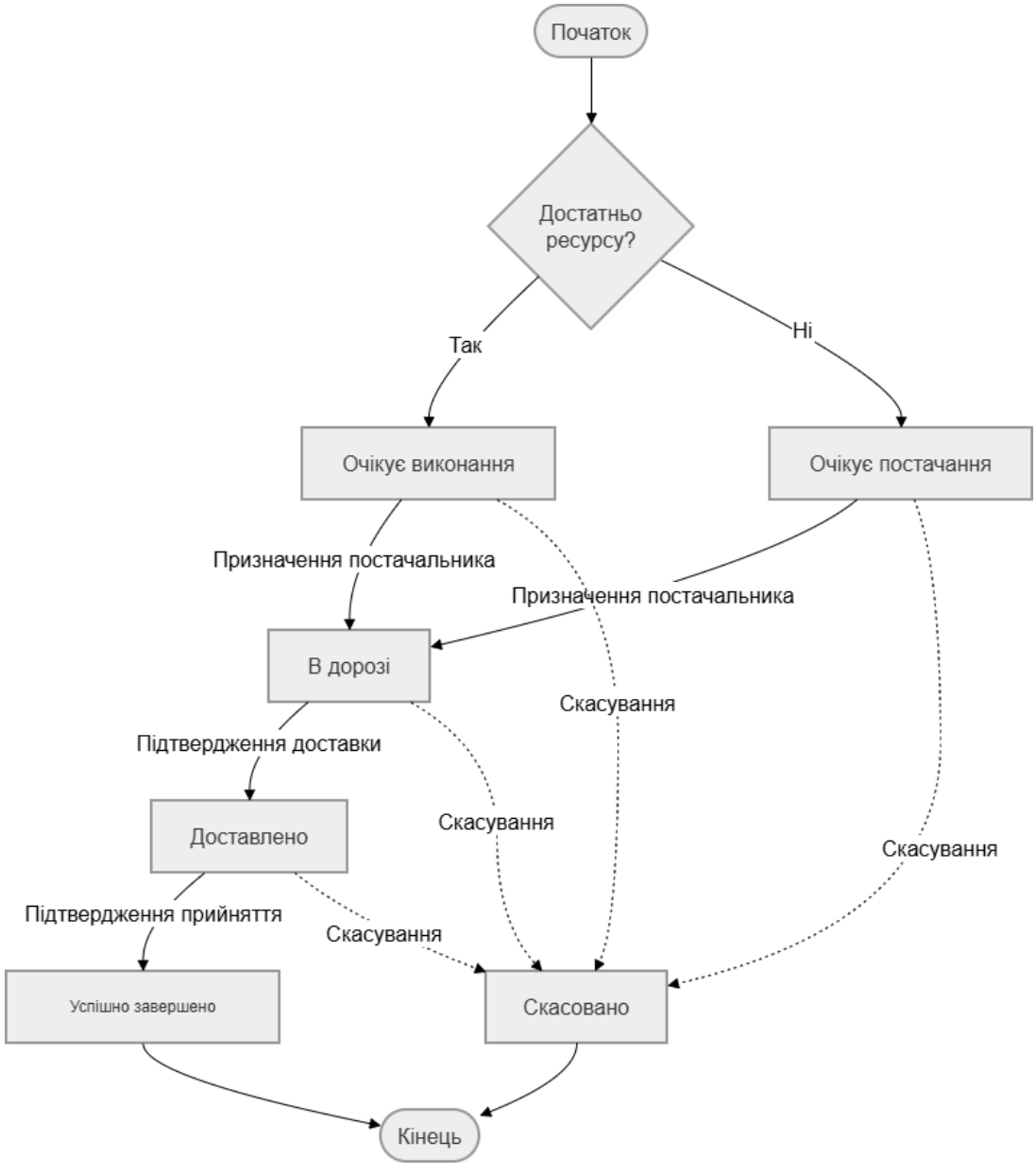


Рис. 1.4 Граф переходів машини станів замовлення

1.5 Специфікація нефункціональних вимог та призначення системи

Замість того щоб описувати окремі функції, нефункціональні вимоги встановлюють технічні рамки, які має дотримуватися вся система. Хоча поведінка при обробці запитів важлива, акцент тут - на загальні характеристики платформи. Серед них: розкладання коду, надійність механізмів захисту, способи оформлення інтерфейсів. Також враховуються регулярність оновлень і можливість тривалої експлуатації продукту. Ці критерії стають ключем до перевірки, наскільки рішення витримає справжнє навантаження, масштабування й підтримку поза межами навчального середовища.

В основі структурної побудови програмного комплексу, лежить концепція шаблону MVC, що гарантує автономність і низьку зв'язність трьох структурних модулів. Об'єкту шару JPA функціонують як компоненти моделі даних, повністю дублюючи конфігурацію таблиць у базі даних. Їх будова копіює схему бази, де навіть найменших деталей не пропущено. Щойно запит потрапляє в систему, вхідні дані на основі Spring MVC перехоплюються на оброблюються. Перевірка даних проходить через механізм Bean Validation, після чого запит переходить на рівень сервісів. Замість монолітної бази впроваджено розділення архітектурних шарів, де клієнтська сторона на Next.js, працює незалежно від серверу. Контроль даних і цілісність типів забезпечує Typescript, тоді як передача інформації відбувається через REST API. Прямий доступ до бази даних з боку клієнта повністю заблокований, відповідно до принципу закладеного проектування. Жоден компонент системи не зможе вторгатися всередину іншого, навіть якщо може. Через це забезпечується жорстке розмежування зон відповідальності.

Перевірка осіб проводиться на OAuth 2.0, разом з OpenID Connect. На рівні відокремленного сховища даних, використовується Keycloak, який зберігає профілі користувачів, хешовані паролі, а також різні привілеї для менеджерів, адміністраторів та постачальників включно. Сервіс здійснює генерацію компактних JWT-токенів, цілісність яких засвідчується складним асиметричним цифровим підписом. Приватна складова ключу залишається строго всередині

системи, тоді як публічна його частина - відкрита для отримання через JWKS-адресу. Як тільки сервер має відкриту складову ключа, забезпечується перевірка автентичності, без залучення серверів Keycloak. Біля кожного методу контролера можна зазначити дозволи - для цього використовують особливу мітку. Подібний підхід дозволяє гнучко керувати доступом саме там, де визначено логіку запиту. Перевірка повноважень вбудована у програмну логіку методу, мінімізуючи розрив між бізнес-логікою та сервісу захисту даних.

Щоб уникнути проблем між сервером та клієнтом, знадобиться документація REST API - важливо дотримуватись специфікації Open API 3.0. Кожен маршрут показує метод запиту й URL одразу, а параметри поділені: обов’язкові вказані першими, решта - через кому нижче. Структура відповіді описана за допомогою JSON Schema, до якої додаються живі приклади того, що повертає система. Розшифровку кодів відповідей наведено прямо, без зайвих слів, лише суть кожного значення. Тут немає двозначності - все працює так, як записано. Коли слідувати цьому підходу, клієнтський код генерується автономно через OpenAPI Generator. Простота у спілкуванні між платформами в агробізнесі зростає саме тут.

Запуск програми має проходити завдяки Docker Compose, всередині якого працюватимуть чотири поєднаних контейнерів - серверна частина, клієнтська частина, система автентифікації Keycloak і загальна база даних. Їхній простір обмежений окремою bridge-мережею, де кожен отримує усталене доменне ім’я. Через ці назви бекенд легко знаходить базу чи Keycloak, не потребуючи жорстких IP-адрес. Але ззовні видно лише два порти - фронтенду та сервера, всі інші служби прибрані назавжди. Збереження даних бази та конфігурацій реалмів відбувається автономно - через виділені Docker-томи на головному сервері, тому після перезапуску інформація залишається недоторканою. Перш ніж запуститись, бекенд чекає сигналу: база має бути доступна, Keycloak також мусить повідомити про свою готовність.

Система повинна підтримувати безстанну архітектуру. Будь-який запит до API має приходити окремо - сервер не тримає слідів минулих взаємодій. Замість цього, кожне звернення до захищеної частини містить у своєму заголовку Bearer-

токен, який сам по собі дає всі потрібні дані. Перевірити його достеменність та видобути права користувача - завдання ланцюжка фільтрів Spring Security перед тим, як обробка дійде до основного коду. Через такий підхід система легко розширюється: нові екземпляри сервісу працюють незалежно, бо їм не треба ділитися інформацією про активні сесії. Набір налаштувань - адреси служб, дані для входу до бази, JWT-параметри, ключі Keycloak і номери портів - подається через змінні оточення з окремого файлу, не потрапляючи прямо в код. Через це одна й та ж версія програми запускається спочатку на машині розробника, потім на внутрішньому сервері чи в хмарі - без правок у файлах додатку, треба лише підставити свій конфіг. Саме такий підхід до управління параметрами стає основою довготривалого супроводження складних систем. Схема розгортання системи (Архітектура системи) продемонстрована на Рис. 1.5.

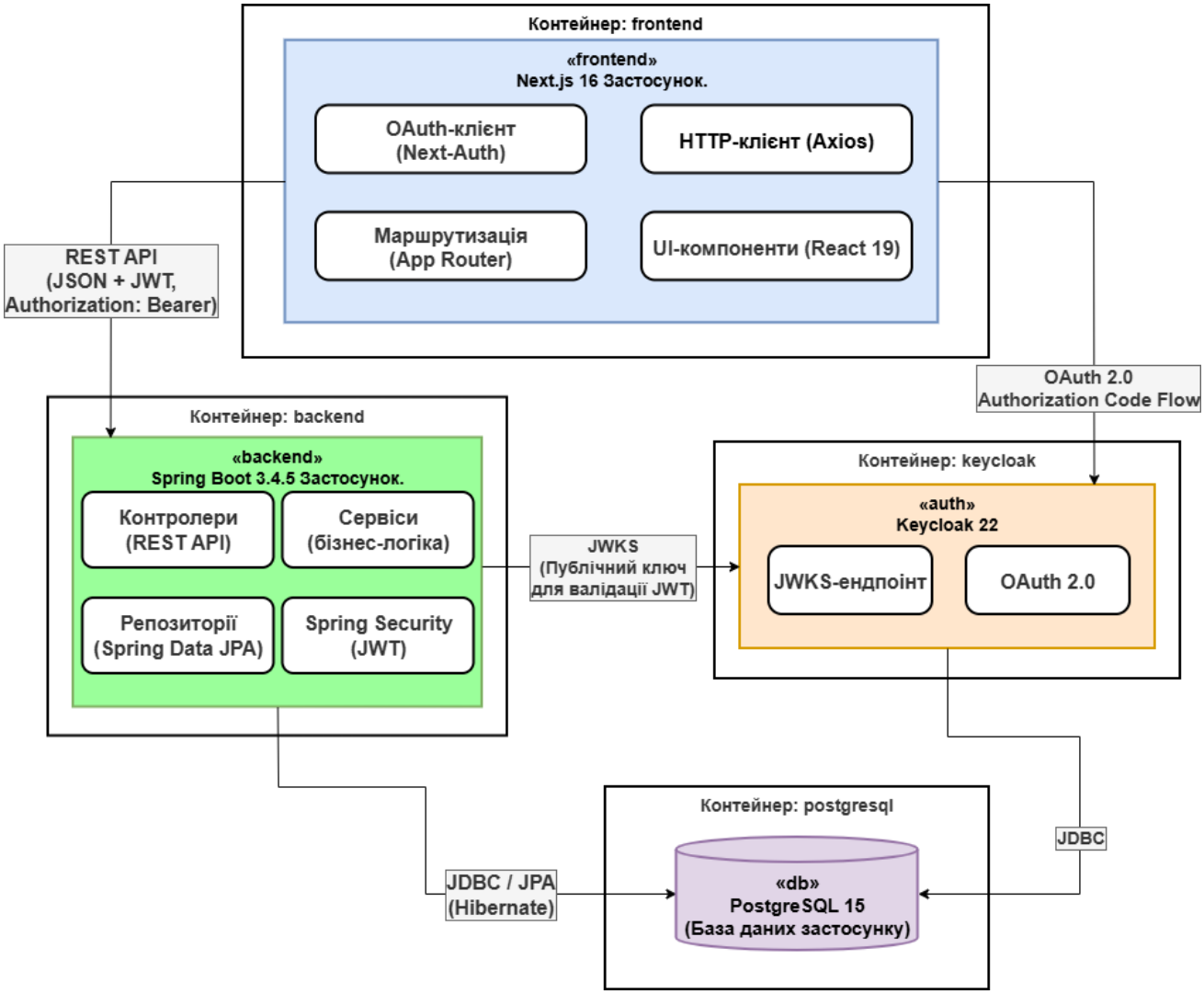


Рис. 1.5 Архітектура розгортання системи в Docker Compose

2. ЗАСОБИ РЕАЛІЗАЦІЇ ПРОГРАМНОГО ПРОДУКТУ

2.1 Вибір мов програмування та середовищ розробки

З самого початку створення програми доводиться обрати компоненти для реалізації - це становить каркас майбутнього рішення. Хоча вплив такого вибору простежується у швидкості написання коду, головне - формуються базові характеристики системи: надійність, безпечність, реактивність, потужність масштабування. Якщо проглянути неправильний стек, наслідки нагромаджуються з часом значно активніше, порівняно з темпами додавання нових функцій; далі підтримувати таке рішення коштуватиме забагато, окрім повного рефакторингу. У сферах, де технічні недоліки можуть зривати операційну сталість компанії, до процесу слід підходити особливо серйозно. Скажімо, платформа для контролю замовлень у сільському господарстві - будь-який збій із передачею стану заявок чи попадання приватної інформації до інших гравців ринку перестає бути лише питанням коду. Клієнти втрачають довіру, коли виникають такі обставини - угода може бути скасована, слідом за цим йде судова справа. Перевірка кожного елемента стека проходить окремо, хоча паралельно аналізуються альтернативні варіанти, адже важливо зрозуміти контекст ринку, внутрішні закономірності галузі та нюанси поточних операцій.

Серверну частину побудовано на основі Java разом із сімнадцятою довготривалою версією платформи [5] й інструментами Spring Framework. Перевірка типів відбувається ще на стадії кодування, задовго до запуску процесу: помилки видно одразу через внутрішню реакцію інструментарію. Зазначений механізм блокує потрапляння помилок у робоче середовище, мінімізуючи ризики збоїв під час обробки замовлень або некоректного збереження системних даних. Особливості проекту означають одне - контроль за типами має тримати всю центральну логіку. До 2029 року Java 17 отримуватиме системні поновлення - цей термін покриває більшість життєвих циклів проектів. Заморожена платформа дає

змогу не шукати альтернативи через раптові зміни середовища виконання.

Для реалізації клієнтської частини обрано TypeScript у поєднанні з фреймворком Next.js шістнадцятої версії та моделлю маршрутизації App Router. TypeScript є надмножиною JavaScript із статичною типізацією, що дозволяє описувати форму об'єктів, аргументів функцій та відповідей REST API у формі декларативних інтерфейсів. Коли бекенд повертає об'єкт замовлення з полем status, TypeScript-клієнт може оголосити, що це поле є переліченням з п'яти допустимих значень, і компілятор гарантуватиме, що у шаблоні інтерфейсу неможливо звернутися до неіснуючого стану. Головним аргументом на користь Next.js у даному проєкті є бібліотека next-auth та її підтримка Keycloak: конфігурація провайдера зводиться до оголошення кількох параметрів, що надає готовий Authorization Code Flow з PKCE, управління сесією та автоматичне оновлення токенів.

Для роботи з бекендом часто вдаються саме до IntelliJ IDEA Ultimate. Продукт від JetBrains справляється з Java та оточенням Spring ефективніше порівняно з іншими середовищами. Переміщення від впровадження залежностей прямо до цільового класу відбувається миттєво - аналізатор компонентів Spring не потребує запущеної програми. Самостійне виявлення циклічних зв'язків між окремими Bean - чергова перевага системи. Мапа Hibernate допомагає розгледіти зв'язки між сутностями - це просто. Усередині середовища працює вбудований HTTP-клієнт, тож перевірити REST API можна без зайвих кроків. Через Docker Compose керують контейнерами одразу в тому ж просторі. Час заощаджується на кожному етапі - навіть коли потрібно знайти помилку після першого запуску коду. Frontend-код створюють у Visual Studio Code, бо інструмент знайомий багатьом. Замість простого додавання - тут працює інший механізм: підтримка синтаксису, аналіз структури й автозаповнення в режимі реального часу. Сервер мови для TypeScript виявляє невідповідності типів ще до запуску, пропонуючи правильні варіанти без затримок.

2.2 Серверний стек: Spring Framework та суміжні бібліотеки

Серед інструментів для побудови програмного забезпечення на основі Java найчастіше обирають Spring Framework [6]. Замість класичного способу складання компонентів вручну - тут діє принцип зворотного управління. Об'єкти не шукають одне одного, вони оголошують свої потреби. Увесь процес керування залежностями покладений на спеціалізований механізм - контейнер IoC. Саме він виявляє потрібні елементи, створює їх та підключає там, де слід. Завдяки цьому можна запускати перевірку частин системи - до прикладу, блок опрацювання замовлень чи логістики постачальників – без потреби запуску повного середовища. Достатньо використати заглушки. Перебудова одного модулю всередині ніяк не заважає роботі решти, адже жоден рядок коду повторно не правиться.

Якщо розбирати архітектуру третьої версії Spring Boot, то вона, історично спираючись на ядро Spring Framework [6, 7], повністю зосереджена на парадигмі «працює одразу». Практика написання громіздких XML-документів чи ручного зв'язування кожного елемента остаточно відійшла в минуле. Натомість розробник оперує виключно файлом збирання Maven, куди просто підключає потрібні стартові бібліотеки (starters). Активація базового оточення відбувається під капотом і абсолютно автоматично - жодних супровідних конфігураційних скриптів писати не доведеться. Вирішальною фішкою тут є інтегрований вебсервер Tomcat. Фактично, додаток збирається в незалежний JAR-файл, що стирає звичну межу між написаним кодом та середовищем для його виконання: тепер вони існують як єдине ціле. При перенесенні цієї структури в Docker ми отримуємо максимально безболісний процес. Замість того, щоб налаштовувати розрізнені компоненти, образ будується навколо одного монолітного блоку. Як наслідок, ризик того, що система впаде через якусь дрібну розбіжність версій сервера чи оточення, зводиться до статистичної похибки.

Обробка HTTP-запитів у Spring Web MVC побудована на ідеї "Фронт-контролера" [7]. Уся вхідна мережева активність проходить крізь єдиний диспетчер. Він читає URL і метод протоколу. На основі цієї пари визначається

конкретний контролер та конкретний його метод, до якого піде запит. Прив'язку методу до URL-шляху та HTTP-методу задає декларативна анотація на самому методі. Звідки брати значення для параметрів, вирішують додаткові анотації, поставлені вже на параметрах: тіло запиту в JSON, параметри URL після знаку питання, сегменти самого шляху, заголовки. Між Java-об'єктами та JSON-структурами працює Jackson Databind. Бібліотека самостійно зіставляє назви полів класу з ключами об'єкта, причому в обидва боки симетрично.

Spring Data JPA додає рівень абстракції над специфікацією Java Persistence API [7, 8] та її еталонною реалізацією Hibernate ORM. Обсяг коду для стандартних операцій із базою даних завдяки цьому скорочується. Кожен репозиторій системи є інтерфейсом, який успадковує параметризований базовий інтерфейс репозиторію. Реалізацію розробник не пише. Spring Data JPA генерує її автоматично під час запуску застосунку, спираючись на назви методів та анотації. Параметризовані запити закладено як обов'язкову вимогу безпеки. Жодне значення від користувача не конкатенується у рядок запиту, що унеможливлює SQL-ін'єкцію навіть за зловмисно сформованих вхідних даних. Запити, які не вписуються у конвенцію похідних методів через надмірну складність умов, оформлюються через анотований JPQL або нативний SQL із явним оголошенням відповідності результату Java-об'єкту.

Spring Security вибудовує захист HTTP-рівня через ланцюжок фільтрів-перехоплювачів. Конфігурація ланцюжка декларативна. Усередині нього окремі фільтри займаються своїми перевірками без перетинів між собою. JWT-фільтр читає заголовок Authorization, дістає токен і перевіряє його підпис проти публічного ключа Keycloak. Поряд із ним працює перевірка HTTP Origin, яка порівнює заголовок із переліком доменів, занесених у CORS-конфігурацію. Окремий фільтр виокремлює ще на вході запити до публічних ендпоінтів, для яких автентифікація не потрібна, і відразу пропускає їх далі. Принципова деталь у тому, що порядок проходження ланцюжка не є жорстко лінійним до кінця. Будь-який фільтр здатний обірвати шлях запиту. Знайшов JWT-фільтр прострочений або підроблений токен, обробка зупиняється на цьому місці. Клієнт отримує HTTP 401,

контролери та сервіси про такий запит не знають нічого. Stateless-режим, у якому ланцюжок налаштовано, означає відсутність будь-якого сесійного стану на сервері. Кожен запит самодостатній, його авторизація встановлюється виключно з токена в заголовку. Додавання другого, третього, десятого екземпляра сервера не потребує жодної синхронізації між ними. CORS-конфігурація живе всередині того самого ланцюжка. Запити з доменів клієнтського застосунку проходять, усі інші браузер блокує ще до моменту відправки на сервер.

Lombok і MapStruct працюють у парі, хоч і розв'язують різні задачі. Кодогенерацією займаються обидва, але на різному рівні. Lombok прибирає рутину з оголошення класів. Анотації на класі під час компіляції розгортаються у стандартні конструктори, геттери, сеттери, метод-будівельник (Builder), реалізації equals, hashCode і toString. Що залишається в коді сутності, це власне набір полів. Шаблонний код, який нічого не додає до бізнес-логіки, просто відсутній. MapStruct натомість бере на себе перетворення між JPA-сутностями та DTO. Реалізації мапер-інтерфейсів генеруються в момент компіляції і несуть повну типову безпеку. Накладних витрат на рефлексію в runtime немає жодних. Помилка відображення, якщо вона є, проявляється під час збірки, а не вилазить серед робочого дня з першим викликом методу на проді.

2.3 Клієнтський стек: Next.js та бібліотеки React

React дев'ятнадцятої версії разом із Next.js шістнадцятої та парадигмою App Router становлять основу клієнтської частини. Сам React це бібліотека з компонентною моделлю побудови інтерфейсу. На практиці це означає, що екран складається не з єдиного цілісного шаблону, а з вкладених один в одного блоків. Картка замовлення це один такий блок. Форма створення нової заявки інший. Таблиця замовлень із фільтрами третій. Кожен блок самостійно зберігає внутрішній стан і самостійно вирішує, як саме його зміст потрапляє в DOM. Окремо варто згадати про роботу зі станами завантаження. Замість того щоб у кожному блоці вручну прописувати умовну гілку «якщо дані ще не прийшли,

показати спіннер, інакше показати вміст», React дев'ятнадцятої версії пропонує компонент Suspense та механізм паралельного рендерингу. Те, що раніше вимагало повторюваного шаблонного коду в десятках місць, тепер описується декларативно одним компонентом-обгорткою.

App Router це інший спосіб організовувати Next.js-застосунок порівняно з тим, що пропонувала попередня модель маршрутизації [9]. Центральна новація це серверні компоненти React, нова категорія компонентів, які виконуються лише на сервері Node.js і до браузера у вигляді JavaScript не доходять взагалі. Звертатися напряму до бекенду такому компоненту дозволено, читати конфігурацію сервера теж, виконувати довготривалі операції без ризику роздути браузерний bundle також. Розмір JavaScript, що дістається користувачу, від такої роботи серверного компонента ніяк не залежить. Браузеру приходить уже готовий HTML і мінімальна порція JavaScript, потрібна винятково для гідрування тих фрагментів сторінки, які мають бути інтерактивними. Клієнтські компоненти позначаються директивою на самому початку файлу і вже там, у браузері, забезпечують реакцію інтерфейсу на дії користувача: поля введення з валідацією, кнопки з обробниками подій, таблиці з фільтрацією, що оновлюється в реальному часі.

Перш ніж зупинитися на Next.js, було розглянуто кілька кандидатів, і кожен відпав зі своєї причини. Create React App довгий час був стандартним інструментом для нових React-проєктів, проте підтримка цього шаблону офіційно припинена командою React, а серверного рендерингу в ньому не передбачено архітектурно. Vite заслуговує окремої згадки: збірка швидка, гаряче перезавантаження одне з найкращих на ринку, спільнота активна. Проблема в іншому. SSR у Vite з'являється лише через сторонні бібліотеки на кшталт vite-plugin-ssr, і робота з ним вимагає ручної конфігурації, замість готового з коробки рішення. Vue з Nuxt теоретично закриває більшість тих самих задач, що і React з Next.js, але вже у власній екосистемі. Ключовий чинник, який вирішив питання на користь Next.js, лежав поза тривіальним зіставленням фреймворків. Це готова бібліотека next-auth з підтримкою Keycloak як OAuth-провайдера прямо з коробки. Конфігурація провайдера це буквально кілька параметрів у файлі: URL реалму, ідентифікатор

клієнта, секрет. У відповідь next-auth бере на себе Authorization Code Flow з PKCE, керування сесійним станом і фонове оновлення токенів, без жодного рядка ручної реалізації OAuth-логіки.

Сама інтеграція з Keycloak зроблена через next-auth, яка узагальнює роботу з будь-яким провайдером, що говорить мовою OAuth 2.0. Після того як користувач пройшов авторизацію на стороні Keycloak і повернувся назад, next-auth отримує від нього три токени за один обмін. Перший це access token, ним підписуються звернення до API. Другий refresh token, потрібен для подовження сесії без нового входу. Третій id token, у якому лежить інформація про самого користувача. Зберігаються вони в HTTP-only cookie, підписаному серверним ключем. Атрибут HTTP-only тут принциповий: жодному JavaScript на сторінці значення такої cookie не видно, через що типова крадіжка токенів через XSS виявляється безуспішною. Перевірка терміну дії access token відбувається перед кожним переходом між сторінками та перед кожним запитом до захищеного ресурсу. Якщо до закінчення лишилось мало часу, next-auth фоном обмінює refresh token на новий access token, і користувач про це навіть не дізнається. Інша ситуація, коли вже й refresh token прострочений. Тоді залишається тільки повний редирект назад на сторінку входу Keycloak.

Axios відповідає за всі HTTP-запити, які клієнтська частина надсилає на бекенд. Технічно це обгортка над браузерним Fetch API з власним інтерфейсом та підтримкою перехоплювачів запитів і відповідей. Перехоплювачі тут і є тим, заради чого Axios обрано замість прямого Fetch. На виході з застосунку працює один перехоплювач. Перед фактичною відправкою запиту він тягне з поточної сесії next-auth access token і додає його в заголовок Authorization як Bearer. Контекст виклику для цього перехоплювача прозорий, йому однаково, чи прийшов запит зі сторінки замовлень, чи з модуля аналітики, чи з форми створення постачальника. Користь від такої централізації проявляється з часом. Кожен наступний API-метод, який буде дописано в систему через місяць або через рік, отримує авторизацію автоматично, і програмісту не треба тримати в голові, чи поставив він заголовок у новій функції. На вході в застосунок працює другий перехоплювач, цей розбирає

вхідні відповіді сервера. Окремо обробляється статус 401, який сигналізує про прострочений або відкликаний access token. Замість того щоб одразу повернути помилку викликаючому коду, перехоплювач звертається до next-auth з проханням оновити токен через refresh-механізм, потім повторно відправляє оригінальний запит з оновленим заголовком. Окрім авторизаційного шару, на Axios лежать ще кілька дрібніших речей: автоматичний JSON-парсинг тіла відповідей, обробка мережевих помилок та таймаутів, скасування активних запитів через AbortController.

За візуалізацію аналітики на панелі адміністратора відповідає Recharts, бібліотека побудови інтерактивних графіків саме під React. Порівняно з низькорівневими інструментами на кшталт D3.js, у Recharts є одна важлива перевага: вона природно лягає в декларативну парадигму React. Кожен тип графіка існує як окремий React-компонент. Дані та параметри відображення передаються йому через пропси, а структура графіка описується у JSX так само, як описується звичайна розмітка. Прямого маніпулювання DOM або SVG у користувацькому коді не залишається взагалі.

Стилізацію інтерфейсу взяв на себе Tailwind CSS, утилітарний фреймворк, у якому власні CSS-правила замінюються набором атомарних класів. Дві властивості такого підходу варто виокремити. По-перше, стилі лежать прямо поруч з розміткою, у тому ж самому JSX-файлі. Не доводиться постійно стрибати між файлом компонента та окремою CSS-таблицею, щоб зрозуміти, як саме виглядає блок. По-друге, на етапі збірки Tailwind проходиться по всіх JSX-файлах застосунку і збирає у фінальний CSS-bundle лише ті класи, які реально вживаються в коді. Усе інше відсікається. Результат типового продакшен-CSS зазвичай тримається в межах кількох кілобайт, що для сучасних веб-застосунків з повноцінним дизайном непогана цифра.

2.4 Система управління базами даних та міграції

Вибір моделі зберігання даних задає не тільки те, як виглядає схема бази, а й

те, які операції система взагалі зможе виконувати ефективно. Для системи управління замовленнями реляційна модель напрошується сама. Зв'язки між сутностями тут жорсткі, з чіткими правилами цілісності. Замовлення без конкретного ресурсу не існує. Постачальник, за яким закріплені активні замовлення, не може бути просто видалений з довідника. Коментар поза замовленням взагалі позбавлений сенсу. Реляційна СУБД примушує ці правила виконуватись на своєму рівні, незалежно від того, чи передбачив їх прикладний код, чи десь забув. Виходить додаткова лінія захисту, яка спрацьовує навіть тоді, коли в бізнес-логіці закралась помилка.

Основною СУБД для проєкту став PostgreSQL у п'ятнадцятій версії. Це рішення зважувалось одночасно за двома критеріями, технічним та ліцензійним [10, 11, 12]. Технічна сторона тримається на тому, як PostgreSQL реалізує ACID. Замість блокувальної моделі тут застосовано MVCC, тобто багатоверсійне паралельне управління. Сенс підходу в тому, що жодна транзакція не змушена чекати завершення іншої, навіть якщо обидві працюють з однією таблицею. Читач не ставить блокування на записувача. Записувач, у свою чергу, не блокує читача. Кожна транзакція оперує власним узгодженим знімком даних, актуальним на момент її старту. Усе це не просто архітектурний нюанс. У сценарії аграрної системи така модель оживає в конкретній ситуації: адміністратор ініціює масову зміну статусів замовлень, а паралельно з ним десяток менеджерів відкривають свої списки заявок. Жоден з них не зависає в очікуванні, ніхто не отримує дані в проміжному, наполовину оновленому стані. Друга сторона, ліцензійна, для проєкту з потенційно зростаючим обсягом даних виявилась не менш вагомою. PostgreSQL поширюється під власною ліцензією типу BSD, де відсутні будь-які обмеження на комерційне використання та немає платежів, прив'язаних до обсягу даних чи кількості з'єднань. Oracle Database має зовсім іншу комерційну модель. У MySQL, залежно від редакції, теж зустрічаються ліцензійні особливості, які при горизонтальному масштабуванні починають конвертуватися в реальні витрати.

Версіями схеми бази даних у проєкті керує Flyway [13]. Це інструмент міграцій, який застосовує SQL-скрипти в строго заданому порядку і веде повну

хронологію всього, що зі схемою колись робилось. Назва файлу скрипта кодує і його порядковий номер, і коротке описання змін. У поточному проєкті використовуються три такі скрипти. Перший створює базову структуру: таблиці ресурсів, постачальників, замовлень і коментарів з усіма зовнішніми ключами, перевірочними обмеженнями та індексами, що до них належать. Другий додає до основних таблиць аудитні поля, тобто дату і час створення запису та дату і час останньої модифікації. Третій розширює таблицю замовлень парою полів для запланованої та фактичної дати постачання. Логіка роботи Flyway побудована навколо контрольних сум. Для кожного застосованого скрипта він зберігає в службовій таблиці хеш його вмісту. При наступному запуску застосунку Flyway проходиться по файлах міграцій і звіряє їх з тими записами в таблиці. Скрипти, контрольна сума яких збігається з записом, він пропускає. Нові скрипти застосовує по порядку, кожен у власній транзакції. А ось спроба змінити вже застосований скрипт навіть на один символ викликає помилку запуску з діагностичним повідомленням. Це захист від ситуації, коли в розробці одне, на тестовому стенді інше, а в продакшені взагалі третє, причому ніхто вже не пам'ятає, як так вийшло. Натомість Flyway гарантує, що схема в трьох середовищах буде ідентичною з точністю до байта.

Java-код спілкується з реляційною базою через Hibernate ORM. Цей шар відповідає за відображення Java-класів на таблиці PostgreSQL і робить це за допомогою декларативних анотацій. Принцип простий: у базі є таблиця, у коді є клас, між ними встановлена пряма відповідність. Анотації, які проставлені і над оголошенням класу, і над окремими його полями, описують усі дрібниці цієї відповідності. Назва таблиці. Імена стовпців, якщо вони не збігаються з іменами полів. Обмеження на null та на унікальність. Стратегія, за якою генерується первинний ключ. Окремо описуються зв'язки між сутностями, і тут є кілька варіантів. Один постачальник може бути пов'язаний з багатьма замовленнями. Одне замовлення тягне за собою колекцію коментарів. Кожен такий зв'язок отримує власну анотацію, де крім кардинальності прописується ще і стратегія завантаження. Стандартна поведінка Hibernate в цьому питанні консервативна.

Пов'язана колекція до бази не запитується доти, доки код фактично не звернеться до неї хоча б раз. Цей режим у документації Hibernate називають lazy loading, ліниве завантаження. Він економить ресурси в більшості випадків, але іноді шкодить продуктивності, коли пов'язані дані наперед відомо як потрібні. На цей випадок передбачений механізм явного приєднання: JPQL-запит з конструкцією JOIN FETCH одразу витягує і основну сутність, і пов'язані з нею колекції одним проходом по базі.

Пул з'єднань з базою даних у системі забезпечується HikariCP, який ідентифікується Spring Boot під час старту автоматично, через простий факт присутності його залежності у класпасі. Сама задача пулу не настільки тривіальна, як здається на перший погляд. Кожне нове TCP-з'єднання з PostgreSQL вимагає TCP-рукостискання, ініціалізації SSL-каналу та аутентифікації на стороні бази, у сумі це десятки мілісекунд на одне з'єднання. Якщо помножити цю цифру на потенційну кількість одночасних HTTP-запитів у пікові моменти роботи з системою, наприклад під час масової зміни статусів замовлень адміністратором, накладні витрати на встановлення з'єднань швидко перевищать корисний час обробки запитів. HikariCP працює інакше. Набір з'єднань піднімається наперед, тримається відкритим у фоні, і кожна транзакція застосунку отримує одне з з'єднань на час свого виконання, після чого повертає його в пул, не закриваючи фізично. Конфігураційні параметри пулу, такі як кількість з'єднань, таймаут на очікування вільного слоту, гранична тривалість утримання одного з'єднання, винесені у змінні середовища і не зашиті в код.

Поряд з HikariCP в інфраструктурі живе pgAdmin четвертої версії, графічна консоль адміністрування PostgreSQL. Розгортається вона окремим контейнером у спільній з іншими сервісами Docker bridge-мережі. Через її вебінтерфейс зручно дивитись на структуру схеми, запускати разові SQL-запити прямо з браузера, аналізувати плани виконання тих запитів, які виглядають підозріло повільними. Окремо вартує згадати, що в продакшен-конфігурації Docker Compose порт pgAdmin не публікується назовні через секцію ports, він залишається доступним тільки в межах внутрішньої мережі контейнерів, до якого ззовні дотягнутись не

вдасться без додаткового SSH-тунелю.

2.5 Система авторизації та контейнеризація

У системі автентифікацію й авторизацію передано зовнішньому серверу для керування особистостями - такий підхід часто застосовують у корпоративних проектах. Безпечне зберігання паролів, оновлення токенів і швидке закриття доступу вимагають ґрунтовних знань із криптографії. Замість цього обрано готове рішення IAM, яке бере на себе цей обов’язок через перевірену інтеграцію.

У ролі такого сервера в проєкті виступає Keycloak двадцять другої версії, відкрите рішення корпоративного класу [16]. Підіймається він власним контейнером і надає захищений вебінтерфейс для адміністрування всього, що стосується ідентичності: користувачів, їхніх ролей, груп, зареєстрованих клієнтів, налаштувань потоків автентифікації. Центральне поняття у моделі Keycloak це realm, тобто ізольований простір налаштувань. Окремий realm агропідприємства тримає в собі визначення трьох системних ролей (ADMIN, MANAGER, SUPPLIER), зареєстрованих клієнтів, параметри видачі токенів та користувачів. Інші realm'и на тому ж сервері до цих налаштувань ніякого стосунку не мають, їхні дані відокремлені повністю. Ролі визначено саме на рівні realm і вшиваються прямо в access token. Практичний наслідок такого рішення в тому, що бекенду не доводиться зайвий раз ходити до Keycloak, аби з'ясувати, хто є хто. Перевірена роль приходить разом з токеном у кожному запиті, і власна база даних застосунку звільняється від обов'язку тримати окрему таблицю відповідності користувач-роль.

Стандарти, на яких побудовано безпекову частину, конкретні: OAuth 2.0 [14] з потоком Authorization Code Flow та розширенням PKCE, плюс OpenID Connect [18] для отримання верифікованих атрибутів автентифікованого користувача. JWT-токени [17] Keycloak підписує асиметричним алгоритмом RS256. Приватний ключ ніколи не залишає сервер Keycloak. Серверна частина застосунку тримає в себе лише публічний ключ, який отримує з JWKS-ендпоінту, і використовує його винятково для перевірки автентичності підпису. Це і є ключова перевага

асиметричної схеми. Маючи публічний ключ, бекенд впевнено відрізняє справжній токен від підробленого, але водночас не має технічної можливості сам випустити новий токен від імені Keycloak.

Docker і Docker Compose разом тримають всю інфраструктуру застосунку в контейнеризованому вигляді. Логіка Docker зводиться до одного: упакувати компонент із усім, що йому потрібно для роботи, у єдиний переносний образ, який поводитиметься ідентично на ноутбуці розробника, тестовому сервері чи продакшен-вузлі. У проєкті використовується двоетапна збірка для обох Dockerfile, серверного та клієнтського. Спочатку береться важкий базовий образ з компіляторами та повним інструментарієм, у ньому проходить компіляція. Далі готовий артефакт переноситься в окремий легкий образ, де від інструментів збірки не лишається нічого, тільки runtime. Кінцевий образ виходить помітно меншим за повну збірку, а зайвих компонентів, які потенційно могли б розширити поверхню атаки, у ньому просто немає.

Docker Compose [19] декларативно описує запуск мультиконтейнерного застосунку через YAML-файл конфігурації. Система складається з чотирьох частин: кожна має свій образ, порти, налаштування оточення та порядок запуску. Оскільки всі контейнери живуть в одній внутрішній мережі типу bridge, вони легко знаходять один одного. Наприклад, бекенд спілкується з базою даних і Keycloak просто за назвою - IP-адреси не потрібні. Тільки фронтенд і бекенд виставляють порти назовні; решта приховані глибоко всередині. Дані бази чи конфігурація Keycloak не зникають після перезапуску - їх тримають окремі volume з іменем. Починаючи працювати, бекенд чекає: спершу PostgreSQL повинен сказати «готовий», потім Keycloak має підтвердити - токени можна видавати. Щоб не помилитися з налаштуваннями, всі важливі значення завантажуються через змінні оточення. Ці дані беруться з файлу .env, який стартує разом із системою. Саме там записано те, без чого система небезпечна. Але цього файлу немає в коді, який видно всім - його додає лише адміністратор, коли готує конкретне середовище.

3. МОДЕЛЮВАННЯ ТА ПРОЕКТУВАННЯ СТРУКТУРИ ПРОГРАМИ

3.1 Моделювання вимог: користувацькі історії та логіка взаємодії

User Story Mapping це метод структурування функціональних очікувань [20], який укладає вимоги у двовимірну карту. По горизонталі розгортається наскрізний бізнес-процес у хронологічному порядку, спільному для всіх учасників, це так званий backbone. По вертикалі відкладається рівень деталізації разом із пріоритетом реалізації. Для системи закупівель аграрних ресурсів такий підхід підходить майже ідеально. Різні ролі підключаються до процесу на різних фазах, і саме карта дає змогу побачити їхню взаємодію цілісно, а не фрагментами. Backbone у нашій карті складається з шести послідовних активностей, які покривають повний цикл закупівельної заявки, від моменту її реєстрації до аналізу результатів. Ці шість активностей винесені в заголовки колонок таблиці 3.1. Зібрані User Stories далі пройшли MoSCoW-пріоритизацію. Така обробка дала змогу окреслити функціональне ядро системи, відокремивши обов'язковий мінімум від того, що бажане, але не критичне для першого розгортання.

Таблиця 3.1

Карта користувацьких історій (User Story Map)

Пріоритет	Реєстрація заявки	Призначення виконавця	Здійснення постачання	Приймання та закриття	Аналіз результатів	Загальні функції
MVP (Must Have)	US01: Створення заявки на аграрний ресурс. US02: Перегляд актуального стану заявок.	US04: Призначення постачальника до заявки.	US06: Перегляд призначених заявок. US07: Позначка про доставку.	US08: Верифікація отримання та закриття заявки. US09: Скасування заявки.	-	-

Продовження таблиці 3.1

Карта користувацьких історій (User Story Map)

Пріоритет	Реєстрація заявки	Призначення виконавця	Здійснення постачання	Приймання та закриття	Аналіз результатів	Загальні функції
Should Have	US03: Коментування заявок.	US05: Управління довідниками ресурсів і постачальників.	-	-	US10: Перегляд аналітичної статистики.	-
Could Have	-	-	-	-	-	US11: Сортування табличних переліків.
Won’t Have	-	-	-	-	-	Текстовий пошук. Email-нотифікації.

Опис кожної User Story у карті стандартизований і складається з чотирьох частин: роль учасника, конкретна ціль, очікувана цінність та критерії прийняття. Критерії прийняття це перелік перевірюваних умов, за яких User Story вважається реалізованою правильно. Формулюються вони винятково з боку зовнішньої поведінки системи, тобто того, що користувач реально бачить і може спостерігати. Деталі внутрішньої реалізації в критеріях не фігурують.

Сценарії для менеджера агропідприємства:

User Story 1: Створення заявки на аграрний ресурс.

Роль: Менеджер.

Опис: Як менеджер, я хочу оформити заявку на аграрний ресурс, щоб ініціювати процес постачання без необхідності безпосереднього контакту з адміністратором.

Критерії приймання (Acceptance Criteria):

- У формі вибору доступні лише активні ресурси з довідника.
- Поле кількості приймає виключно додатні числа.
- Дата постачання не може передувати поточній даті.
- Після підтвердження форма очищується, а заявка відображається у переліку із автоматично присвоєним початковим статусом.

User Story 2: Перегляд актуального стану заявок.

Роль: Менеджер.

Опис: Як менеджер, я хочу переглядати актуальний стан своїх заявок, щоб розуміти, на якому етапі перебуває кожна закупівля.

Критерії приймання (Acceptance Criteria):

- У переліку показуються тільки ті заявки, які створив сам авторизований менеджер.
- Для кожного рядка виводяться назва ресурсу, замовлена кількість, поточний статус і, якщо вже призначений, постачальник.
- Доступна фільтрація за статусом та діапазоном дат створення замовлення.
- Кілька фільтрів комбінуються за логікою AND.

User Story 3: Коментування заявок.

Роль: Менеджер.

Опис: Як менеджер, я хочу залишати коментарі до заявки, щоб уточнювати деталі або повідомляти про зміну пріоритетів у контексті конкретної закупівлі.

Критерії приймання (Acceptance Criteria):

- Поле введення коментаря розміщене на сторінці деталей кожної заявки.
- Збережений коментар одразу з'являється в стрічці у хронологічному порядку, разом з іменем автора та часом публікації.
- Коментарі адміністратора або постачальника видно менеджеру в тій самій стрічці, і симетрично навпаки.

User Story 11: Сортування табличних переліків.

Роль: Менеджер.

Опис: Як менеджер, я хочу сортувати таблиці із замовленнями, щоб швидко знаходити найважливіші або найновіші записи при великому обсязі даних.

Критерії приймання (Acceptance Criteria):

- Сортування переліку доступне за номером замовлення.
- Підтримуються обидва напрямки сортування: за зростанням і за спаданням.
- Сама операція виконується реактивно, прямо в браузері на стороні клієнта, без повторного звернення до сервера.

Сценарії для адміністратора агропідприємства:

User Story 4: Призначення постачальника до заявки.

Роль: Адміністратор.

Опис: Як адміністратор, я хочу призначити постачальника до заявки, щоб передати відповідальність за постачання конкретному виконавцю.

Критерії приймання (Acceptance Criteria):

- Дія призначення доступна виключно для заявок, що перебувають у статусі очікування виконавця.
- У випадаючому списку для вибору відображаються лише активні постачальники.
- Після підтвердження заявка зникає з черги нерозподілених і змінює статус на етап доставки.
- Менеджер та обраний постачальник одразу бачать оновлений статус та призначення у своїх інтерфейсах.

User Story 5: Управління довідниками ресурсів і постачальників.

Роль: Адміністратор.

Опис: Як адміністратор, я хочу управляти довідником ресурсів і постачальників, щоб підтримувати актуальну номенклатуру без порушення цілісності існуючих даних.

Критерії приймання (Acceptance Criteria):

- Після деактивації ресурс зникає з форми вибору при створенні нових заявок менеджером.
- Раніше створені заявки, у яких фігурує деактивований ресурс або постачальник, продовжують відображатися як зазвичай, без втрат даних.
- Адміністратор може повторно активувати запис, після чого той знову стає доступним у формах вибору.

User Story 8: Верифікація отримання та закриття заявки.

Роль: Адміністратор.

Опис: Як адміністратор, я хочу верифікувати факт отримання ресурсу, щоб зафіксувати успішне завершення закупівлі.

Критерії приймання (Acceptance Criteria):

- Операція доступна лише для заявок, переведених постачальником у стан очікування приймання.
- Після підтвердження заявка переходить до фінального завершеного статусу.
- Система блокує будь-яке подальше редагування закритої заявки для всіх ролей (інформація стає доступною лише для читання).

User Story 9: Скасування заявки.

Роль: Адміністратор.

Опис: Як адміністратор, я хочу скасувати заявку на будь-якому активному етапі, щоб припинити процес закупівлі у разі зміни обставин.

Критерії приймання (Acceptance Criteria):

- Кнопка скасування доступна для будь-якої заявки, що ще не переведена у фінальний позитивний стан.
- Після скасування заявка набуває термінального статусу і стає недоступною для подальшої роботи.
- Якщо під час створення ресурс був зарезервований (списаний із залишків), система автоматично вивільняє та повертає цей обсяг на баланс.

User Story 10: Перегляд аналітичної статистики.

Роль: Адміністратор.

Опис: Як адміністратор, я хочу переглянути агреговану статистику по закупівлях, щоб аналізувати ефективність постачань.

Критерії приймання (Acceptance Criteria):

- Доступ до сторінки дашборду відкритий тільки користувачам з роллю адміністратора.
- На дашборді показується загальна кількість заявок та їхній розподіл за поточними статусами.
- Адміністратор бачить рейтинг постачальників, побудований за кількістю успішно завершених заявок.
- Усі показники перераховуються динамічно, залежно від обраного адміністратором часового діапазону.

Сценарії для постачальника агропідприємства:

User Story 6: Перегляд призначених заявок.

Роль: Постачальник.

Опис: Як постачальник, я хочу переглянути заявки, за які відповідаю, щоб планувати та виконувати свою роботу.

Критерії приймання (Acceptance Criteria):

- Перелік відображає виключно ті заявки, де поточний авторизований постачальник призначений відповідальним.

- Заявки, призначені іншим контрагентам, або ті, що ще не мають виконавця, не відображаються.
- У переліку доступна детальна інформація про ресурс, необхідну кількість та бажану дату постачання.

User Story 7: Підтвердження здійснення постачання.

Роль: Постачальник.

Опис: Як постачальник, я хочу позначити заявку як доставлену, щоб повідомити систему про фактичне виконання своєї частини роботи.

Критерії приймання (Acceptance Criteria):

- Кнопка підтвердження доставки активна тільки для заявок у стані активної доставки.
- Після підтвердження заявка переходить у статус очікування приймання.
- Самостійно відкотити цю дію постачальник уже не може, далі заявка переходить до зони відповідальності адміністратора.

Чітка межа для MVP з’явилася завдяки MoSCoW-пріоритизації. У групі Must Have - User Stories 1, 2, 4 та 6–9; разом вони покривають увесь процес: від створення заявки до її завершення для кожної з трьох ролей. У групу Should Have потрапили US03, US05 і US10. Хоч система й працюватиме без них, деякі процеси стануть менш чутливими. Функціональність архітектури збережеться, але позитивної динаміки не очікується. Окрему корисну функцію принесе US11, який забезпечує легкість роботи з великими масивами даних - це Could Have. Перший запуск стане можливим після повного впровадження блоку Must Have.

Між сценаріями взаємодії з системою і машиною станів, описаною в підрозділі 1.4, є прямий зв'язок: майже кожен сценарій відповідає окремому переходу між статусами. Створення заявки (US01) визначає, з якого стану замовлення стартує, спираючись на поточні залишки ресурсу. Призначення постачальника (US04) запускає перехід до стану виконання. Підтвердження доставки постачальником (US07) переводить заявку до очікування приймання.

Верифікація адміністратором (US08) закриває життєвий цикл, а скасування (US09) спрацьовує як альтернативний термінальний шлях. У підсумку User Stories і машина станів формують узгоджену пару: кожна історія прив'язана до конкретного переходу між статусами, і весь набір вкладається в єдиний бізнес-процес.

3.2 Проектування бази даних: концептуальна та логічна моделі

Реляційна схема становить той фундамент, на якому тримається вся операційна логіка системи [12]. Рішення, прийняті на цьому рівні, такі як допустимість null, каскади видалення, вибір між фізичним та логічним видаленням, після початку промислової експлуатації коригувати дуже непросто, тому обґрунтування кожного з них належить до принципових питань проектування. Предметна область розкладається на чотири сутності: аграрний ресурс (agrarian_resources), постачальник (suppliers), замовлення (orders), коментар до замовлення (order_comments). Зв'язки між ними виглядають так. Один ресурс фігурує в багатьох замовленнях. Один постачальник може відповідати за багато замовлень, але з боку замовлення цей зв'язок необов'язковий. До одного замовлення прив'язується довільна кількість коментарів.

Виокремлення коментарів у самостійну таблицю order_comments стоїть на трьох вимогах. Перша. Коментарів до однієї заявки може бути скільки завгодно, і кожен повинен зберігатись разом з метаданими: хто автор, коли опубліковано. Друга. Хронологічна сортування за полем created_at у такій структурі лягає природно, без додаткових прийомів на рівні запитів. Третя стосується аудиту. Якби коментар зберігався прямо в рядку замовлення, кожне його додавання оновлювало timestamp самої заявки, і відрізнити реальну зміну замовлення від простого коментаря в історії аудиту стало б неможливо. Зв'язок order_comments з orders налаштований як каскадний на видалення. Самостійної цінності поза замовленням коментар не має, і тримати його в базі після того, як батьківський запис уже видалено, не потрібно ані технічно, ані з точки зору бізнес-логіки. Видалення відбувається автоматично, на рівні самої бази, ніякого допоміжного коду в

застосунку для цього не передбачено.

Зовнішній ключ supplier_id у таблиці orders допускає значення NULL, і це властивість, закладена в схему свідомо. Логіка така: замовлення на самому початку життєвого циклу, у статусі PENDING_SUPPLY, ще не має жодного призначеного постачальника. NULL у відповідному полі семантично описує саме цей стан, без жодних обхідних трюків. Альтернативи виглядали гірше. Тримати в довіднику постачальників технічний «порожній» запис, до якого прив'язувались би всі ще нерозподілені замовлення, означало б одразу два мінуси: порушення нормальної форми та постійні костилі в кожному фільтраційному запиті, де такого фіктивного постачальника довелося б щоразу виключати. Інша справа з ресурсом. Тут зовнішній ключ між orders та agrarian_resources, навпаки, оголошений як NOT NULL. Замовлення без ресурсу позбавлене предмета взагалі, його існування на жодному етапі циклу не має сенсу, тож структура схеми просто не дає такій ситуації виникнути.

Для довідникових таблиць agrarian_resources та suppliers вибрано стратегію логічного видалення, soft delete. Технічно вона зведена до одного поля active типу BOOLEAN з дефолтним значенням TRUE. Деактивація запису це перемикання цього прапорця в FALSE, фізично рядок з бази нікуди не зникає. Чому не звичайне DELETE? Тому що ресурс або постачальник, на яких уже посилається хоча б одне історичне замовлення, не можна видалити чисто: або доведеться каскадно зносити всі пов'язані заявки разом з історією, або порушити цілісність зовнішніх ключів. Перший варіант неприйнятний з точки зору бізнесу, другий заборонений на рівні самої бази. Soft delete оминає обидва тупики. У формі вибору при створенні нових замовлень деактивований ресурс не з'являється, бо запит фільтрує позиції через WHERE active = TRUE. У старих заявках, де цей ресурс уже фігурує, він продовжує відображатися як належить, без жодних артефактів. Повторне ввімкнення запису теж не вимагає чогось особливого, ніяких міграцій або ручних SQL-скриптів, простий UPDATE одного поля. Для коментарів така схема не потрібна, у них немає самостійної цінності поза заявкою, тому там обрано пряме каскадне видалення.

У таблиці agrarian_resources закладено поле version типу INTEGER для

механізму оптимістичного блокування. Списання залишків при переході замовлення в PENDING_EXECUTION складається з трьох кроків: прочитати, перевірити, оновити. Без захисту тут легко виникає гонка даних. Два запити на той самий ресурс зчитують однаковий залишок, обидва визнають його достатнім, обидва зменшують. Підсумок прогнозований, залишок іде в мінус. Hibernate при збереженні рядка звіряє поточну версію в базі з тією, що була на початку транзакції. Версія не збіглась, виняток кидається, операцію можна повторити з актуальними даними. Для типового навантаження аграрної системи такі конфлікти трапляються винятково, тому оптимістична схема виявляється практичнішою за явне блокування рядків. Аудитні поля created_at та updated_at додаються другою міграцією Flyway і заповнюються автоматично через Hibernate, у тілі запиту від клієнта вони не приходять.

Для таблиці orders створено складений індекс на парі (status, requested_delivery_date). Саме така комбінація стоїть за більшістю фільтраційних запитів у системі. Адміністратор зазвичай дивиться замовлення певного статусу в межах конкретного проміжку дат постачання, окремо ці параметри використовуються рідше. Складений індекс дає PostgreSQL змогу виконувати такі запити через Index Scan замість Sequential Scan, що залишається актуальним і тоді, коли таблиця замовлень помітно зростає в обсязі. Порядок полів усередині індексу закономірний. Статус поставлений першим свідомо: всього п'ять його унікальних значень добре нарізають вибірку вже на початковому кроці пошуку. У таблиці замовлень окремо живе поле complete_date, дата завершення. Доти, доки замовлення в активному циклі, поле залишається порожнім. Воно заповнюється тільки в момент, коли заявка переходить у термінальний стан. Аналітичному модулю це дає можливість рахувати фактичний час виконання за реальними датами, а не за плановими, які могли від реальних відрізнятись. Логічна ER-діаграма з повним переліком таблиць, атрибутів, типів даних, ключів та каскадних правил винесена на рисунок 3.1.

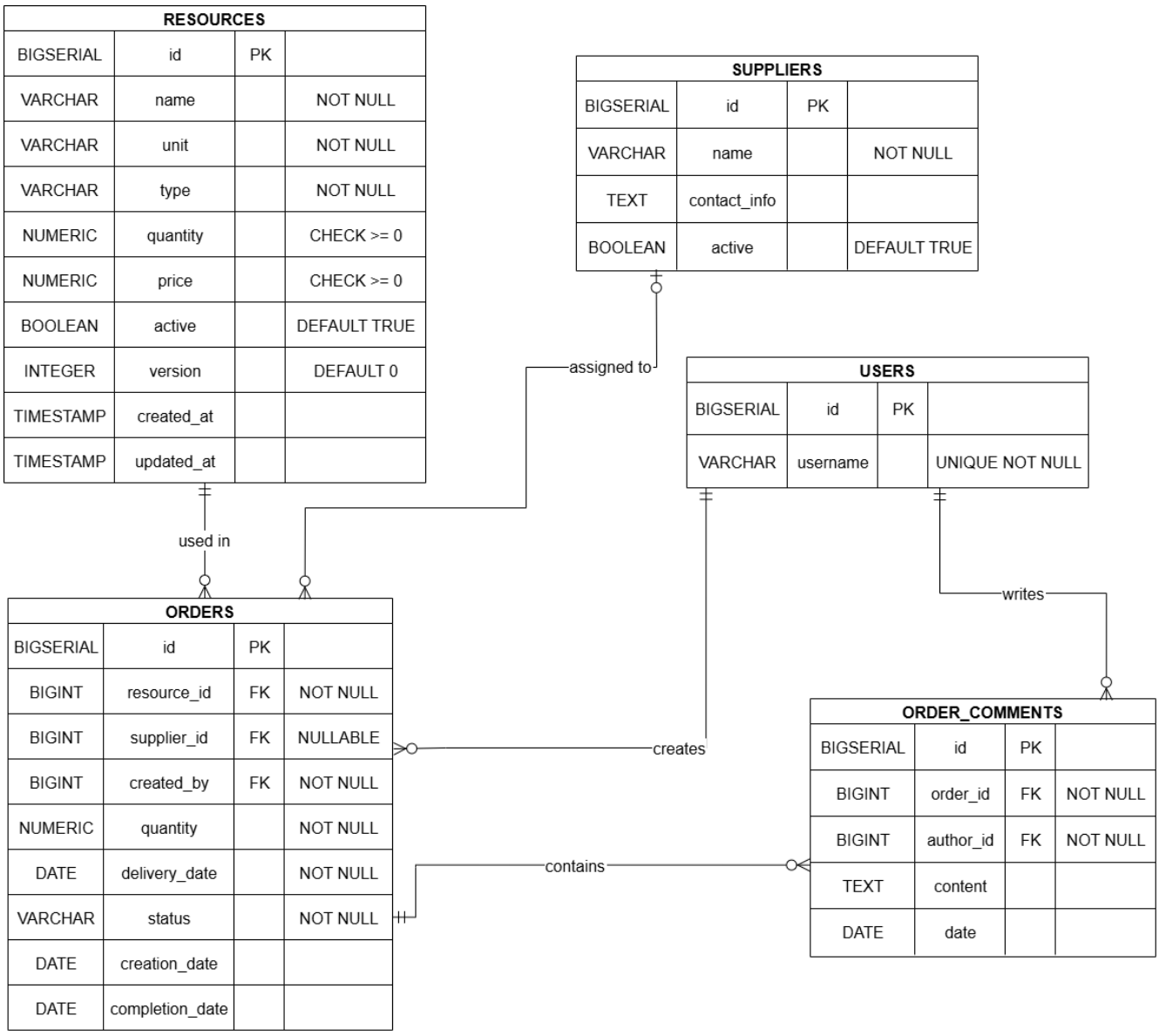


Рис. 3.1 ER-діаграма бази даних системи

Окремою позицією у схемі стоїть таблиця users, яка тримає локальне відображення користувачів системи. Авторитетним джерелом для всього, що стосується автентифікації та ролей, залишається зовнішній Keycloak, однак ідентифікатори користувачів усе ж зберігаються і на стороні бази. Без цього неможливо було б технічно прив'язати користувача до доменних об'єктів, насамперед до коментарів і замовлень, які він створює або в яких бере участь. Така архітектура дає системі цілісність даних і повноцінний аудит дій конкретного користувача, водночас не змушуючи дублювати логіку автентифікації, яка вже реалізована в IAM.

Сама схема витримана у третій нормальній формі. Кожен неключовий

атрибут залежить тільки від первинного ключа своєї таблиці, і ні від чого більше. У таблиці orders, наприклад, немає жодного дубльованого атрибута постачальника, доступ до них відкривається через зовнішній ключ, тож при зміні контактних даних постачальника аномалії оновлення тут просто не виникнуть. Перша нормальна форма дотримана за рахунок того, що кожен атрибут атомарний. Друга за рахунок того, що часткових залежностей від складених ключів у схемі немає взагалі. Усе це не випадковий побічний ефект, а результат свідомих проектних рішень, прийнятих на етапі моделювання предметної області.

3.3 Проектування серверної частини: структура класів

Серверна частина побудована за принципом шаруватої архітектури, де залежності між шарами йдуть тільки в одному напрямку. Контролер може звернутись до сервісу, сервіс до репозиторію, репозиторій до класу-сутності, але ніколи навпаки. Така односторонність дає системі дві практичні властивості. Будь-який шар переписується незалежно від інших, а під час тестування його легко ізолювати від суміжних. Розподіл коду по пакетах повністю віддзеркалює описану ієрархію. У entity лежать сутності предметної області, у dto зі своїми request і response об'єкти передачі даних, у dto.mapper інтерфейси перетворення між цими двома групами. Доступ до бази винесений у repository, бізнес-логіка в service, REST-ендпоінти у controller. Поза цією основною лінією тримаються два допоміжні пакети: config містить налаштування безпеки та фільтрації, а exception об'єднує всі виняткові ситуації разом з їх централізованим обробником.

Серед усіх предметних сутностей одне з найбільш принципових рішень припало саме на Order. Замовлення в системі грає роль агрегатного кореня, тобто навколо нього збирається все, що з ним пов'язано. У самому класі це посилання на ресурс, посилання на постачальника та колекція коментарів. Колекція коментарів цікава своєю поведінкою. Коментарі свідомо зроблені залежними сутностями, поза замовленням їх існування не має жодного сенсу. Коли видаляється замовлення, коментарі зникають разом з ним самі по собі, на рівні бази, без жодного коду, який

би вручну це обробляв. Клас Resource має у собі поле версійного лічильника, через нього в системі працює оптимістичне блокування при паралельному списанні залишків. OrderComment простіший, у ньому тільки сам текст коментаря, ідентифікатор автора та позначка часу. Supplier зберігає назву, контактну інформацію та прапорець, чи активний цей постачальник зараз. Жодна з цих сутностей не містить бізнес-логіки, всередині них лежать тільки дані та зв'язки між таблицями. Уся реальна обробка винесена нагору, на сервісний шар, де їй і місце.

Перерахування OrderStatus у системі виконує роль єдиного джерела правди щодо допустимих значень стану замовлення. У базі статус зберігається у текстовій формі, не як числовий код. Завдяки цьому дані залишаються самодокументованими, а сам тип на рівні коду відсікає будь-яке некоректне значення ще до моменту запису. Граф переходів між станами, описаний у підрозділі 1.4, реалізований у сервісному шарі як незмінна структура відображення. Для кожного поточного стану в ній прописана множина тих, у які можна перейти, і нічого більше. Якщо хтось спробує виконати недопустимий перехід, виняток ловиться глобальним обробником і повертається клієнту як структурована HTTP-відповідь з кодом 422.

DTO-шар грає роль буфера між публічним API-контрактом системи та її внутрішньою доменною моделлю. На кожну предметну сутність припадає окрема пара класів: вхідний для прийому даних від клієнта і вихідний для повернення відповіді. Запит на створення замовлення складається з мінімально необхідного набору, ідентифікатора ресурсу, кількості та бажаної дати постачання. Решти полів у цьому класі просто немає. Статус, системні дати, ідентифікатор постачальника відсутні навмисно, бо проставляє їх сама система, і дозволити клієнту впливати на них було б архітектурною помилкою. Зворотний клас, відповідь, влаштований повніше. До базових полів замовлення приклеєні вкладені об'єкти з даними про ресурс і призначеного постачальника. Постачальник тут оголошений як необов'язковий, з простої причини, на ранніх стадіях життя заявки виконавця ще може не бути. Виграш такого підходу для клієнтського застосунку конкретний, замість серії додаткових запитів на ресурс і постачальника, фронтенд отримує все

одним зверненням. Аналогічні пари існують для ресурсів і постачальників, симетрично замовленням. Аналітичні дані живуть у власному класі, у нього спаковані відразу три речі: загальна кількість замовлень, їх розподіл за статусами та рейтинг постачальників за кількістю успішно закритих заявок.

За перетворення між сутностями та DTO відповідають маппери, описані як декларативні інтерфейси. Така форма цікава тим, що відповідальність за коректність відображення полів переноситься з рантайму на момент компіляції. Якщо в інтерфейсі прописана відповідність неіснуючому полю, проєкт просто не збереться, і помилка спливе ще на стадії білду, а не у формі рантайм-винятку на проді. Нетривіальні перетворення теж описуються декларативно. Скажімо, коли треба витягти кілька атрибутів ресурсу з вкладеного об'єкта і виставити їх плоскими полями у відповіді клієнту, цього не доводиться писати руками. Інтерфейс маппера фіксує сам шаблон відповідності, а конкретну реалізацію генерує MapStruct сам, ще на етапі збірки.

Шар контролерів поділений за предметними областями системи. Окремий клас тримає всі ендпоінти, що стосуються замовлень, окремий для ресурсів, окремий для постачальників, окремий для аналітики. Усередині самого контролера бізнес-логіки немає взагалі. Його робота зводиться до трьох речей: прийняти запит, перевірити вхідні параметри, передати все це далі сервісу і повернути сформовану відповідь. Кожен метод контролера прикритий декларативною перевіркою ролей, через яку доступ до конкретного ендпоінту обмежується відповідно до рольової моделі системи. Ділові рішення приймаються рівнем нижче, у сервісах. Саме там лежить уся логіка системи. OrderService обробляє найскладніші сценарії, від визначення початкового стану замовлення до коректних переходів між статусами. ResourceService і SupplierService займаються своїми довідниками з підтримкою логічного видалення. AnalyticsService збирає агреговані показники для адміністративної панелі. Детальніший розбір реалізації методів контролерів та сервісів винесений у підрозділи 4.1 і 4.2 відповідно.

Динамічну фільтрацію в системі реалізовано через клас OrderSpecifications за патерном Specification. Логіка тут інша, ніж у класичному варіанті з фіксованими

методами репозиторію. Замість того щоб плодити метод під кожну комбінацію параметрів, у коді описуються невеликі атомарні предикати, для статусу свій, для діапазону дат свій, для решти критеріїв по такому ж принципу. Сервіс під час обробки запиту дивиться, які саме параметри прийшли від клієнта, і складає з потрібних предикатів кінцеву умову. Додавання нового критерію фільтрації при такій схемі обходиться дешево, треба написати один предикат, до інтерфейсу репозиторію руки взагалі не доходять. Зовсім іншу задачу закриває KeycloakRealmRoleConverter. Spring Security за замовчуванням не знає, де саме в JWT-токені Keycloak шукати ролі, бо Keycloak пакує їх у вкладене поле realm_access.roles, а не на верхній рівень claims. Конвертер дістає звідти список ролей і переробляє їх у формат GrantedAuthority, з яким уже працює стандартний механізм перевірки доступу. Поряд з ним у конфігурації стоїть глобальний обробник винятків. Його функція проста: будь-який типізований виняток бізнес-логіки чи технічна відмова перехоплюються в одному місці і перетворюються на однакову JSON-структуру відповіді з діагностичним кодом. Для клієнта це означає одне, формат помилок передбачуваний незалежно від того, у якому шарі система спіткнулась. UML-діаграми класів із зв'язками між пакетами наведена на рисунках 3.2 - 3.4.

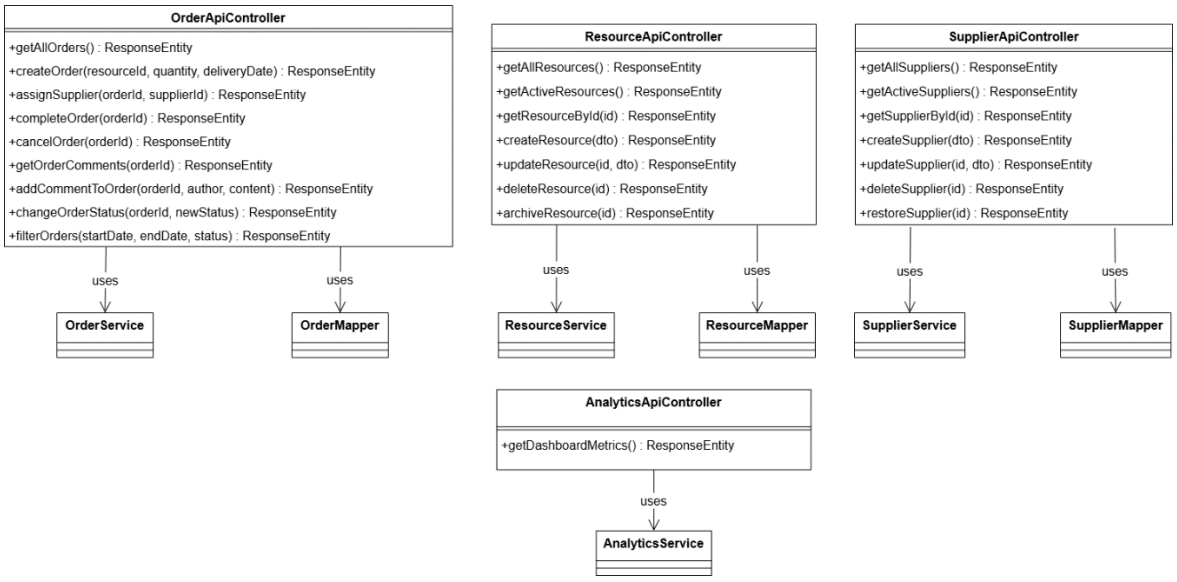


Рис. 3.2 Діаграма класів контролерів серверної частини
з основними полями та залежностями

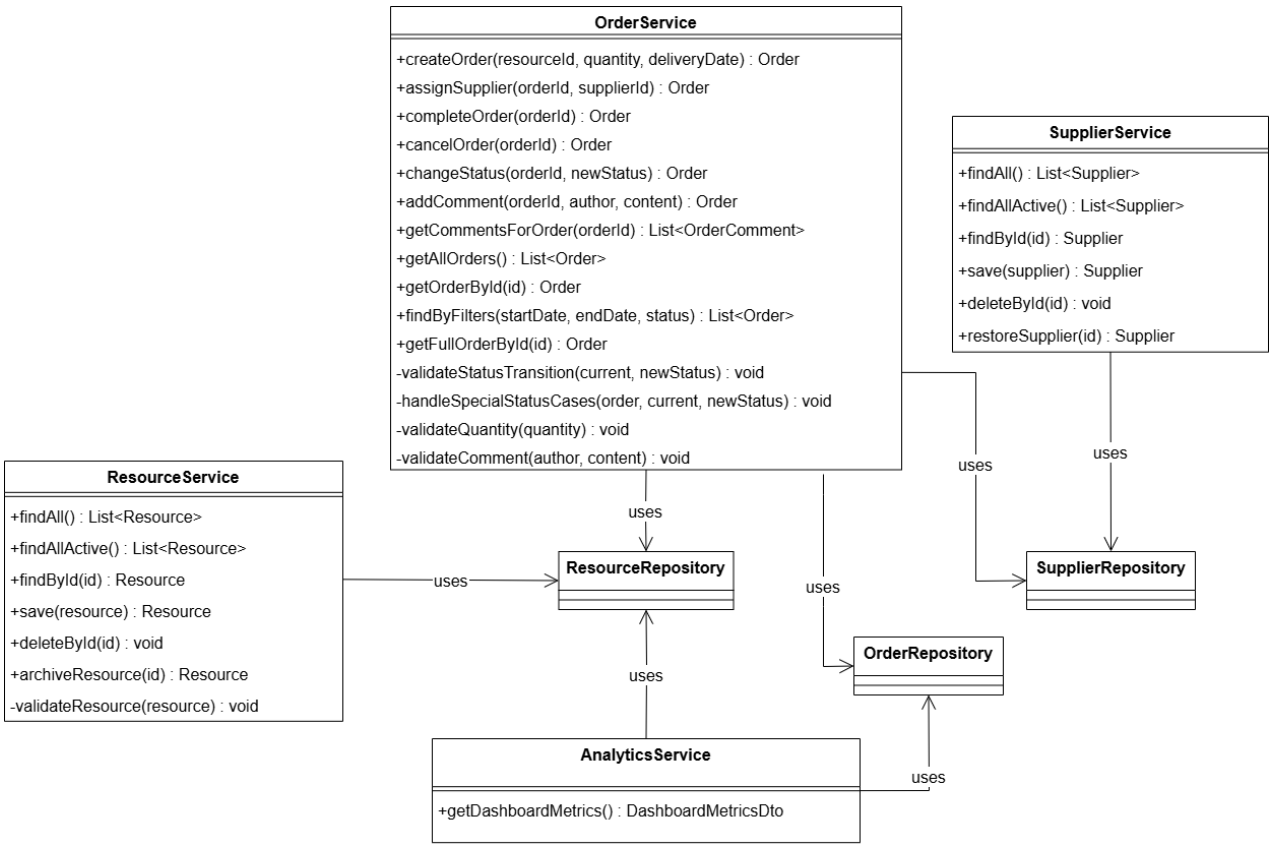


Рис. 3.3 Діаграма класів сервісів серверної частини
з основними полями та залежностями

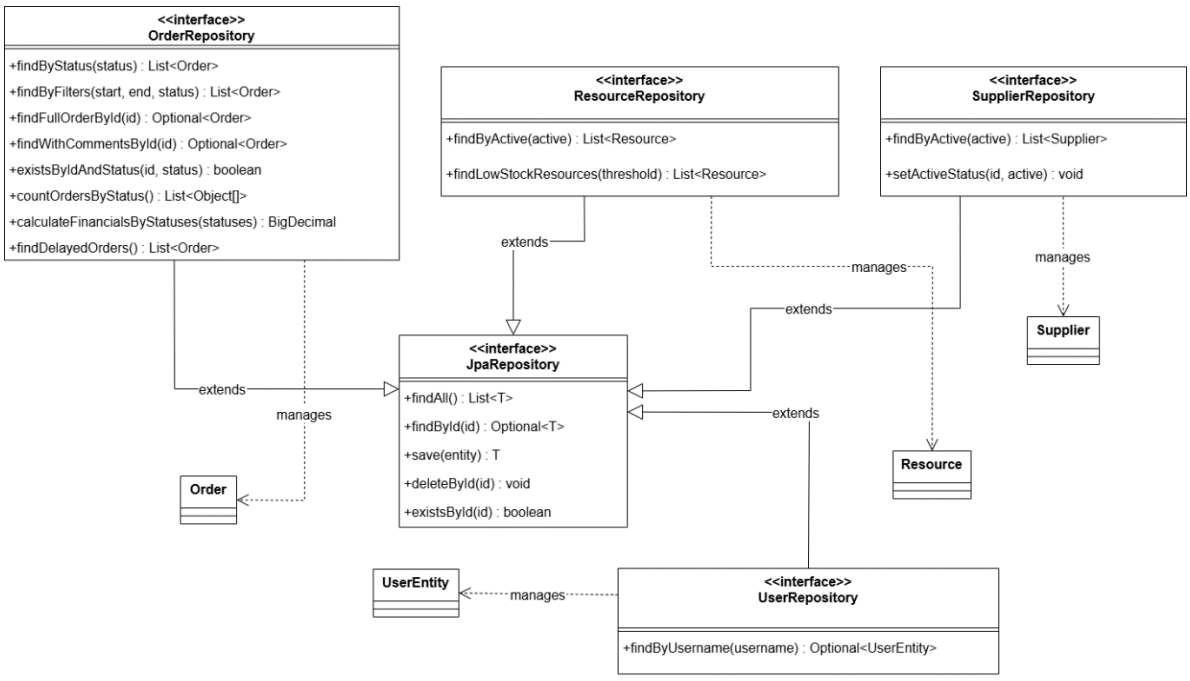


Рис. 3.4 Діаграма класів репозиторіїв серверної частини
з основними полями та залежностями

3.4 Проектування клієнтської частини: компонентна структура Next.js

Клієнтська сторона зроблена як самостійний застосунок, що звертається до серверної частини виключно через HTTP API. Прямого доступу до бази даних, до внутрішніх сервісів, до будь-яких компонентів серверу зсередини у клієнта немає за визначенням [9]. З точки зору архітектури тут одночасно вирішуються кілька доволі різних задач, чотири якщо точно, і вони майже не перетинаються між собою. Перша задача стосується того, як організована навігація між сторінками. Друга це контроль над станом автентифікації, хто зараз у сесії і чи валідна його сесія. Третя задача про те, щоб робота з API лежала окремо від компонентів, які власне малюють інтерфейс. Четверта про адаптацію видимого користувачу під ту роль, з якою він зайшов у систему. Усі чотири мають своє окреме проектне рішення, і ось чому це важливо. Якщо змішати ці відповідальності всередині одного компонента-сторінки, далі починається ефект сніжної кулі: код стає крихким, кожна нова функція тягне за собою правки в десятку місць одночасно, тести або не пишуться взагалі, або вимагають повного переписування при найменших змінах. Архітектурна деградація клієнтських застосунків у міру їх росту майже завжди починається саме з цієї точки.

Навігаційна структура працює за принципом декларативної файлової маршрутизації. Простіше кажучи, ієрархія директорій у вихідному коді один в один відповідає тому, які URL отримає користувач у браузері. Усередині цієї структури виділено чотири функціональні розділи: замовлення, ресурси, постачальники та аналітика. Кожен з розділів живе як відносно самостійний модуль зі своєю сторінкою, яка несе повну відповідальність за свій функціональний обсяг. Прямих залежностей між розділами немає. Перехід відбувається через спільну навігаційну панель зверху, а не через програмні виклики між кодом сторінок. Коренева сторінка не показує власного інтерфейсу, замість цього вона перенаправляє користувача в розділ замовлень, бо саме він є основним робочим екраном для всіх трьох ролей системи. Окремо стоїть маршрут аналітики, доступний винятково ролі ADMIN. Захищений він на двох рівнях. Клієнтська перевірка спрацьовує ще до

завантаження даних і переадресовує користувача без необхідного дозволу, що працює радше на зручність, ніж на безпеку. Реальний бар'єр сидить на сервері, де перевірка ролі прив'язана до кожного API-запиту окремо. Сподіватись тільки на клієнтську перевірку не можна в принципі, бо HTTP-запит до API цілком можна сформувати і поза межами браузера, наприклад через звичайний curl.

Управління станом у застосунку розкладене на два незалежні рівні. Глобальний рівень тримає в собі автентифікаційну інформацію, тобто хто саме зараз працює із системою, яку роль має, який токен у нього на руках. Цей стан незмінний протягом сесії і доступний з будь-якого компонента через механізм контексту, без того щоб тягти його через довжелезний ланцюжок пропсів, як це доводилось робити у старих React-застосунках. Локальний рівень працює інакше. Тут зберігаються поточні значення фільтрів на сторінці, вибраний у таблиці запис, факт відкритого модального вікна, тобто все те, що має сенс рівно в межах однієї сторінки. За її межі цей стан не виходить взагалі. Розділення цих двох рівнів зроблене свідомо. Глобальний стан залишається маленьким і не починає роздуватись з кожною новою сторінкою, а локальний дозволяє вільно змінювати себе зсередини сторінки, не зачіпаючи решту застосунку ніяк.

Усю автентифікаційну механіку винесено в окремий спеціалізований шар на основі бібліотеки next-auth. На цьому шарі відбувається весь цикл OAuth 2.0, від моменту, коли користувач натискає кнопку входу, до фонового оновлення токенів через певний час. Компоненти-сторінки з самими токенами не контактують жодним чином. Замість токенів вони отримують готовий об'єкт сесії і звертаються до нього через простий стандартизований інтерфейс, нічого не знаючи про деталі OAuth-потоку, про refresh-логіку, про формат самих токенів. Така централізація дає одну неочевидну на перший погляд перевагу. Уся прив'язка до конкретного провайдера ідентичності ховається всередині одного шару, тому якщо колись Keycloak доведеться замінити на якийсь інший IAM-сервер, переписувати компоненти-сторінки не доведеться взагалі. Зміни торкнуться лише конфігурації бібліотеки.

Підстроювання інтерфейсу під роль користувача це окрема архітектурна

задача, не звідна до двох попередніх. У системі діють три ролі, і кожна з них працює у тих самих розділах, але з різним обсягом дозволених операцій. У розділі замовлень менеджер бачить тільки свої заявки і має доступ до форми створення нової. Адміністратор у тому самому розділі отримує повний перелік замовлень з можливістю призначати постачальника та змінювати статус. Постачальнику відкривається ще вужче вікно, лише ті заявки, де він поставлений відповідальним. У розділах ресурсів і постачальників різниця ще різкіша. Адміністратору доступні форми створення, редагування та архівування записів, а менеджеру тільки перелік активних позицій без жодної можливості щось у ньому змінити. Уся ця логіка живе всередині компонентів відповідних сторінок і реалізована через умовний рендеринг елементів на підставі ролі, яку компонент дістає з автентифікаційного контексту.

Уся комунікація з серверним API в клієнтському застосунку проходить через окремий централізований модуль, який виконує функцію абстракції над HTTP-шаром. Сам по собі такий шар вирішує одну принципову задачу, не давати деталям мережевої взаємодії розповзатись по всьому коду сторінок. Усе, що пов'язано з мережею, замкнено всередині цього модуля. Він збирає тіло запиту, додає до заголовків необхідний токен авторизації, обробляє стандартні мережеві помилки і конвертує сирий JSON-відповідь сервера в зручну для роботи на стороні UI структуру. Методи всередині модуля розкладені по групах за предметними областями, і ця розкладка точно повторює групування на серверній стороні API. Сама сторінка про роботу з HTTP не думає в принципі. Вона викликає потрібний метод з потрібної групи і отримує на виході готовий типізований результат, з яким уже може рендерити свій інтерфейс.

Перевага архітектури проявляється на будь-якій модифікації. Якщо завтра треба буде поміняти базовий URL сервера, або переробити обробку HTTP 401 і HTTP 403, або підмішати в заголовки новий параметр для трекінгу, ці зміни торкнуться рівно одного файлу. Самих сторінок, працює бізнес-логіка та UI, не зазнають змін.

Типізація клієнтських структур даних потрібна для одного: тримати

узгодженість між тим, як виглядає відповідь сервера на папері, і тим, що клієнтський код реально отримує в рантаймі. TypeScript-інтерфейси на стороні клієнта повторюють форму серверних DTO, поле в поле. Інтерфейс замовлення, наприклад, містить вкладені об'єкти з даними про ресурс і про постачальника, причому постачальник позначений як необов'язкове поле. Це не помилка опису, а пряме відображення бізнес-реальності: заявка може існувати у стані, де виконавець ще не призначений.

Коли TypeScript бачить таку необов'язковість, він просто не дає компонентам поводитись з полем як з гарантовано наявним, обидві гілки, з постачальником і без нього, доводиться обробити явно. Якщо клієнтський інтерфейс розійдеться з тим, що насправді віддає API, проблема виявить себе ще на стадії компіляції, до того, як код взагалі дістанеться браузера. Слабка ланка в цій схемі одна, синхронізація серверних DTO з клієнтськими інтерфейсами лежить на людях.

Зараз її доводиться підтримувати руками, що залишає вікно для розсинхрону на момент змін у API. У наступних ітераціях проєкту цей бік планується автоматизувати через генерацію клієнтського коду напряму з OpenAPI-специфікації сервера, що повністю забере проблему ручної підтримки відповідності. Загальна схема компонентної ієрархії клієнтської частини наведена на рисунку 3.3.

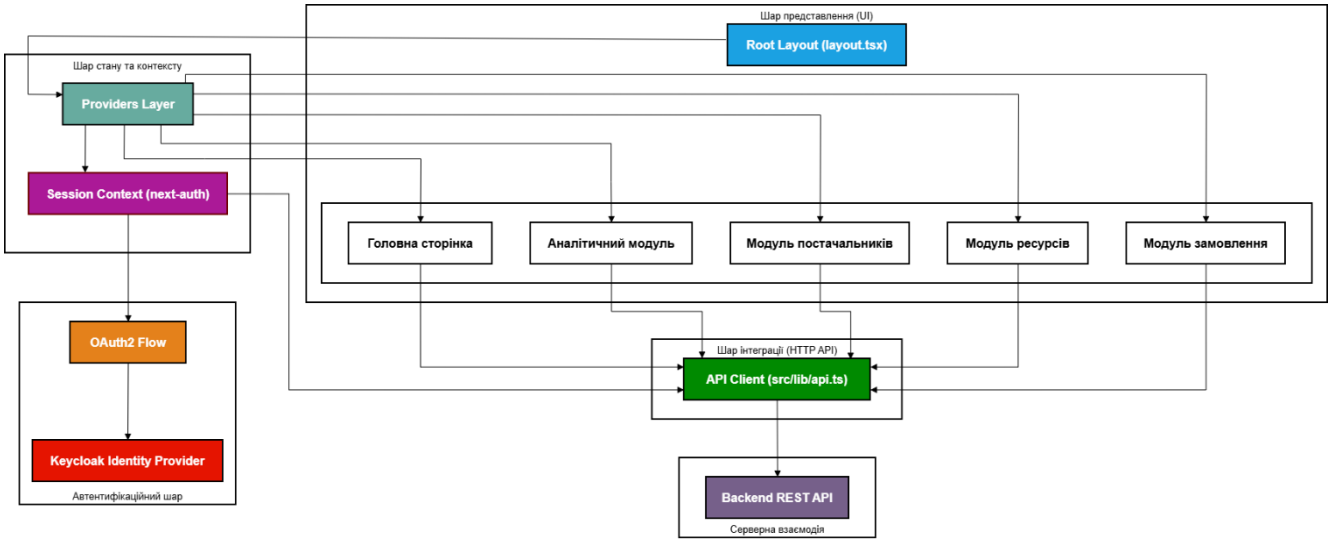


Рис. 3.5 Ієрархія компонентів клієнтської частини
та схема потоку автентифікаційного стану

3.5 Моделювання динамічної поведінки системи

Окреслена в підрозділах 3.2–3.4 система постає у вимірі статичному - з переліком блоків, із яких її зібрано, та способом їх взаємного стикування. Проте для повноцінної проектної специфікації наведеного бракує. Доти, доки не продемонстровано поведінку архітектури під час виконання реальних сценаріїв, котрі розгортаються у часі та змушують компоненти реагувати на зовнішні стимули, ця архітектура лишається безживною конструкцією. Статична схема мовчить. У яких саме точках потік зазнає розгалуження, які з рішень приймаються системою автономно - без втручання оператора, у якій послідовності керування переходить від одного компонента до іншого - відповідей на ці запитання вона не містить. В силу цього апарат UML пропонує два типи діаграм, спарених за призначенням [21]. Внутрішню логіку окремого бізнес-процесу - те, який крок випливає за яким та за якої умови активується наступний - розкриває діаграма діяльності. Натомість погляд із площини логічної у площину часову переводить діаграма послідовності, фіксуючи порядок, у якому компоненти обмінюються повідомленнями, та визначаючи, хто чиєї відповіді очікує. Спарені, ці дві проекції замикають специфікацію. Описану раніше структуру вони прив'язують до конкретної поведінки коду під час його виконання.

Для моделювання міжкомпонентної комунікації відібрано три сценарії, які разом покривають увесь стек взаємодій, від дій користувача в браузері до самої бази даних. Перший сценарій охоплює повний потік автентифікації, від моменту, коли користувач натискає кнопку входу, до моменту, коли застосунок має на руках готову сесію (Рисунок 3.6). Браузер перенаправляється на сторінку входу Keycloak, де користувач проходить перевірку, після чого повертається назад уже з одноразовим кодом авторизації. Цей код далі обмінюється на повний набір токенів, але обмін відбувається не в браузері. Запит на обмін летить прямим HTTPS-каналом між серверами, поза видимістю клієнтської сторони взагалі. Принципова деталь цієї схеми така: токен доступу жодного разу не з'являється у відкритому вигляді на стороні браузера, а тому й перехопити його через клієнтський JavaScript

просто неможливо. Другий сценарій (Рисунок 3.7) описує шлях, який проходить уже авторизований запит від моменту приходу на сервер до отриманої відповіді. Послідовність компонентів Spring Security перевіряє JWT-токен у кілька кроків: чи валідний підпис, чи не вийшов термін дії, чи токен виданий саме цьому застосунку, а не якомусь іншому. Лише після успішного проходження всього ланцюжка з токена витягуються ролі користувача, і запит нарешті доходить до самого контролера. Якщо хоч одна перевірка падає, до контролера запит не доходить взагалі. Третій сценарій (Рисунок 3.8) трасує наскрізний життєвий цикл замовлення від ініціації до закриття. Заявка проходить усі стани машини станів послідовно, починаючи зі створення менеджером і закінчуючи фінальною верифікацією від адміністратора. На кожному кроці цього шляху перевіряється рольовий доступ до конкретного ендпоінту, через що дія, призначена одній ролі, фізично не може бути виконана іншою. Усі три діаграми разом показують, як архітектурні рішення з підрозділів 1.4 та 3.3 поводяться в живому виконанні системи.

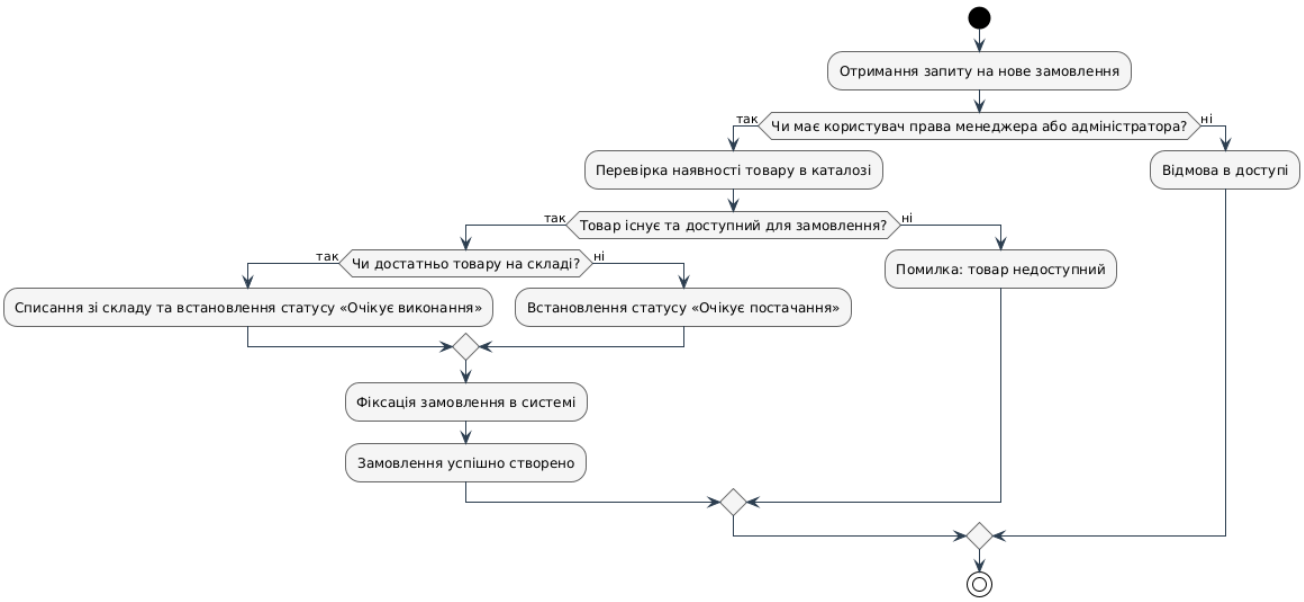


Рис. 3.6 Діаграма діяльності. Процес створення замовлення

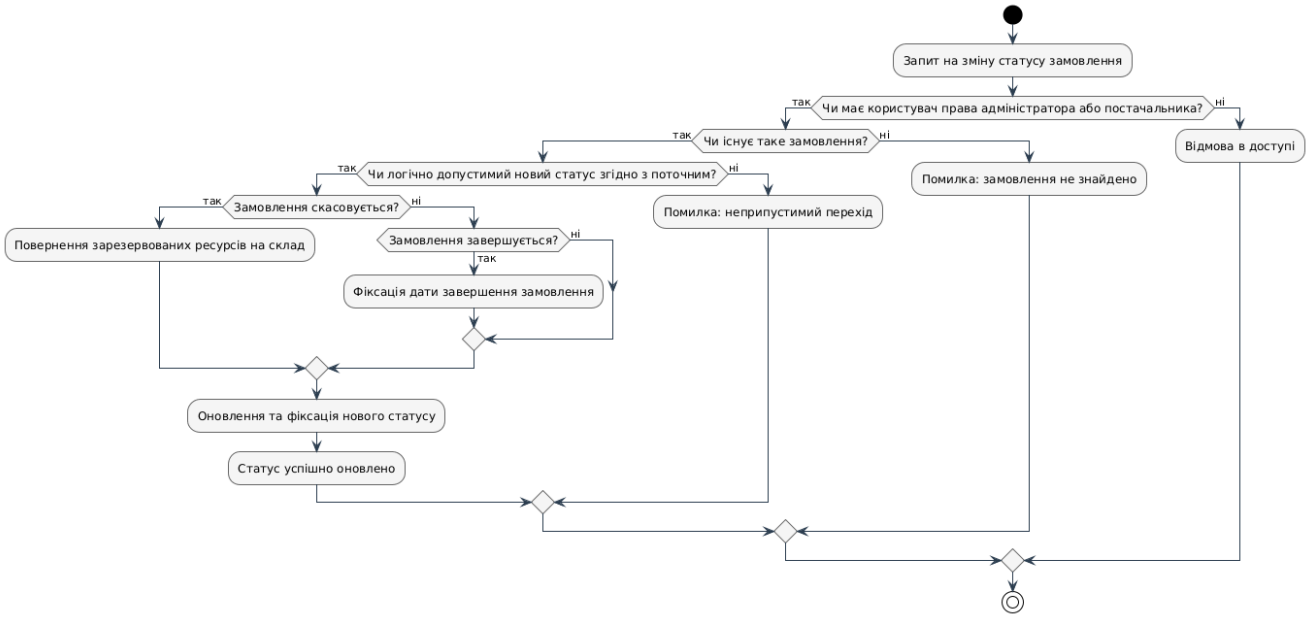


Рис. 3.7 Діаграма діяльності. Процес зміни статусу замовлення

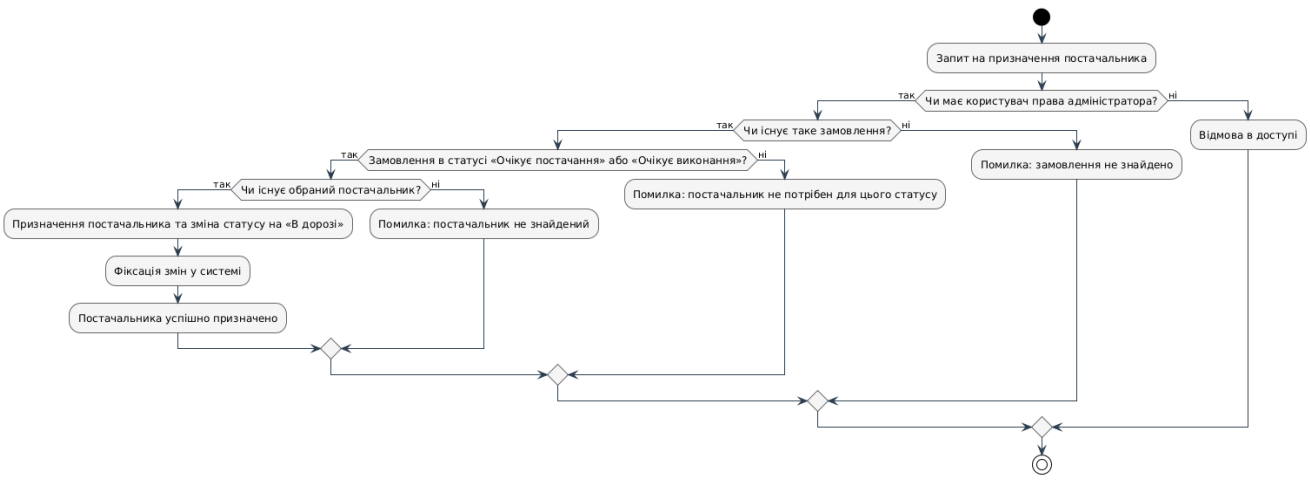


Рис. 3.8 Діаграма діяльності. Процес призначення постачальника адміністратором

Усього вибрано три сценарії, на яких будуються діаграми міжкомпонентної комунікації, і разом вони покривають увесь технологічний стек, починаючи з браузера користувача і закінчуючи самою базою даних. Перший з них (Рисунок 3.9) це повний цикл входу в систему. Усе починається з натискання кнопки на формі логіну, далі браузер перекидає користувача на сторінку Keycloak, де відбувається сама процедура входу. Після підтвердження облікових даних користувач повертається у застосунок, але повертається не сам, а з тимчасовим кодом авторизації в руках. Цей код виконує службову роль і живе всього кілька секунд. Далі застосунок обмінює його на повноцінний набір токенів, причому критично важливо, що цей обмін проходить не через браузер. Сервер застосунку звертається до Keycloak напряму, відкритим HTTPS-каналом сервер-сервер, і самі токени браузер ніколи не бачить навіть в одній зі своїх вкладок. Це і є основна цінність потоку Authorization Code: жоден чутливий артефакт не потрапляє в зону, де його міг би перехопити сторонній скрипт або браузерне розширення.

Другий сценарій (Рисунок 3.10) трохи інакший за характером. Він показує, що відбувається з одним авторизованим запитом від моменту, коли той приходить на сервер, до моменту, коли клієнт отримує відповідь. Фільтри Spring Security вишикувані в чітку послідовність, через яку запит має пройти повністю. Підпис JWT перевіряється на дійсність. Термін дії звіряється з поточним часом. Окремо перевіряється, чи саме цьому застосунку був виданий цей токен, а не якомусь сусідньому з тієї ж екосистеми. Якщо все три кроки пройдено, з тіла токена видобуваються ролі поточного користувача і запит передається контролеру. Якщо хоча б одна перевірка не пройшла, далі по ланцюгу запит не йде, відповідь повертається з відповідним кодом статусу ще до того, як про цей запит дізнається бізнес-логіка.

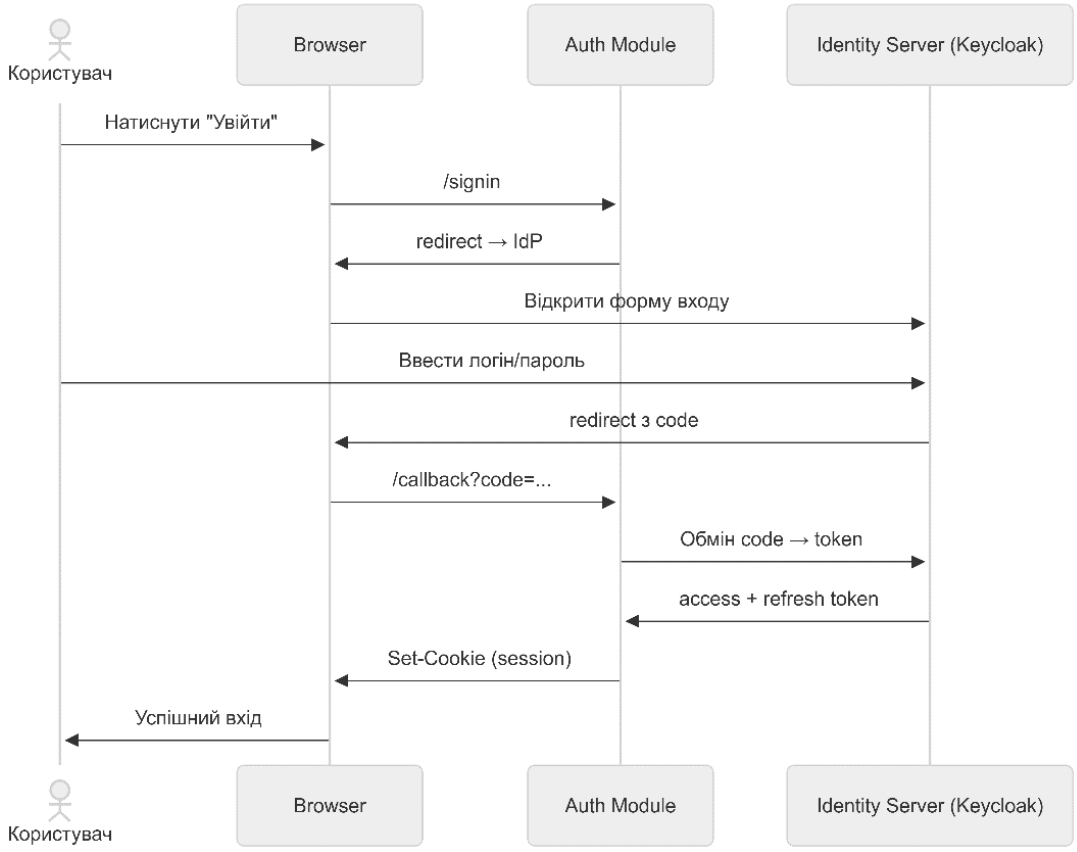


Рис. 3.9 Діаграма послідовності. Автентифікація користувача через сервер ідентифікації

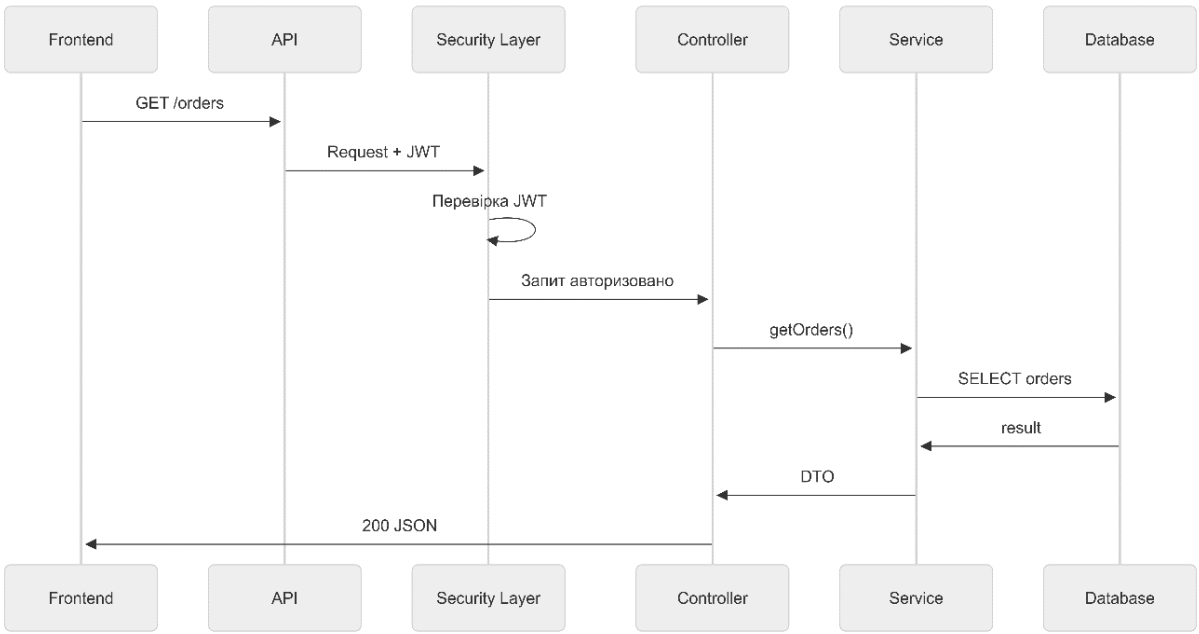


Рис. 3.10 Діаграма послідовності. Обробка авторизованого запиту через ланцюжок компонентів безпеки

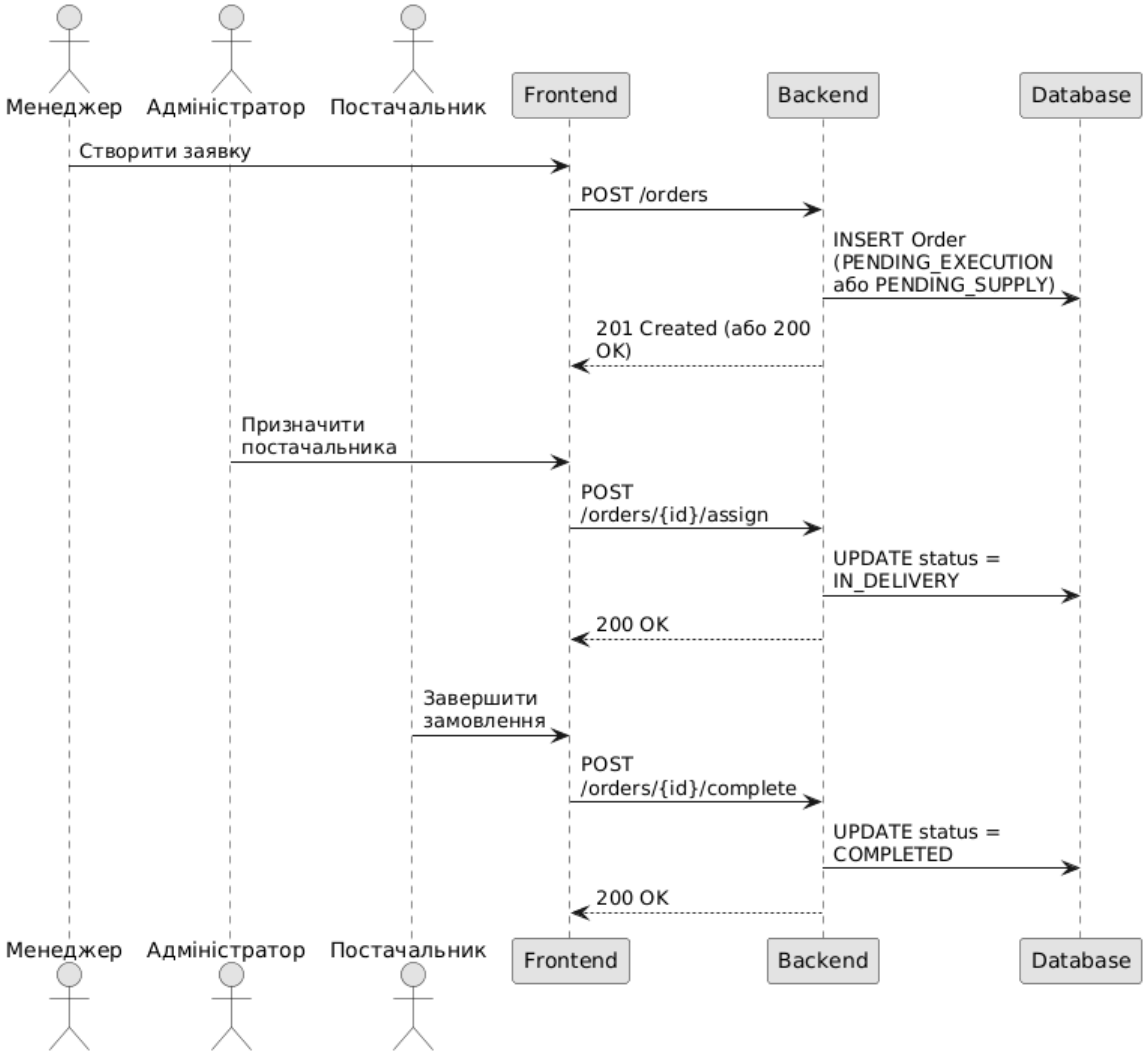


Рис. 3.11 Діаграма послідовності. Наскрізний цикл замовлення від ініціації до завершення

Третій сценарій (Рисунок 3.11) дивиться на систему ширше і трасує життя замовлення наскрізно, від моменту створення до закриття. Заявка проходить усі стани машини станів по порядку, від першої дії менеджера до фінальної верифікації адміністратором. Кожен перехід між станами це окремий ендпоінт API, і на кожному з них незалежно перевіряється рольовий доступ. Через це сама природа архітектури не дозволяє одній ролі виконати дію іншої, навіть якщо хтось спробує сформувати запит вручну в обхід інтерфейсу. Разом три діаграми показують одне: рішення, описані у підрозділах 1.4 та 3.3 на рівні структури, в реальному виконанні поводяться саме так, як було задумано на етапі проектування.

4. ПРОГРАМНА РЕАЛІЗАЦІЯ

4.1 Реалізація серверного REST API

Структура REST API, описана в підрозділі 3.3, реалізована у вигляді контролерів. Це класи, що приймають вхідні HTTP-запити безпосередньо й виступають точкою входу до серверної логіки [3]. Бізнес-логіки в самих контролерах немає взагалі. Їхні обов'язки скромні: прийняти запит, перевірити вхідні параметри на валідність, передати все далі у відповідний метод сервісу і повернути клієнту сформовану HTTP-відповідь. Спокуса вкласти умовну логіку предметної області прямо в контролер на практиці завжди обертається проблемою. Бізнес-правила починають дублюватись між кількома контролерами, а тестувати їх окремо від HTTP-шару стає неможливо. Через це весь такий код тримається на сервісному рівні, контролер залишається тонкою прошарковою оболонкою без власних рішень.

Структурно API-рівень побудований через розмежування контекстів. На кожну предметну сутність системи припадає рівно один клас контролера: окремий для замовлень, окремий для ресурсів, окремий для постачальників, окремий для аналітики. Перетинів між ними немає. Така схема себе виправдовує сильніше за альтернативні варіанти, як-от поділ контролерів за HTTP-методами чи за ролями користувачів. Якщо завтра доведеться розширити функціонал замовлень, інші контролери залишаться недоторканими, бо просто не пов'язані з цією предметною областю. Кожен контролер несе анотацію @RestController з зафіксованим на рівні класу URI-префіксом: /api/orders, /api/resources, /api/suppliers, /api/analytics. Залежності потрапляють у контролер виключно через конструктор. Така схема Spring-ін'єкції в принципі не дає створити об'єкт у неповному стані, бо контейнер просто не зможе його зібрати без усього потрібного.

OrderApiController вийшов найнасиченішим контролером у системі, і причина проста: предметна область замовлень містить найширший набір сценаріїв.

Крім звичайних операцій читання, тут реалізовано цілий перелік дій: створення нової заявки, призначення постачальника, зміна поточного статусу замовлення, явне закриття і скасування заявки з усіма побічними ефектами, які за ними йдуть, додавання коментаря до конкретної заявки і, нарешті, фільтрована вибірка за довільною комбінацією параметрів.

Ендпоінт фільтрації варто розглянути окремо. На вхід він приймає набір необов'язкових параметрів запиту, серед них початкова дата, кінцева дата та стан замовлення. Жоден з них не обов'язковий сам по собі, але їхні комбінації між собою визначають конкретний сценарій вибірки. Формування умов винесене з контролера в сервісний шар, де на основі реально переданих параметрів динамічно збирається предикат. Така архітектура виявилась досить гнучкою на розширення. Якщо в майбутньому з'явиться потреба фільтрувати, скажімо, ще й за конкретним постачальником або типом ресурсу, доведеться додати тільки новий предикат у клас OrderSpecifications. Сам контролер при цьому залишиться недоторканим.

Рольове розмежування доступу до ендпоінтів зроблене через анотацію @PreAuthorize, яка ставиться прямо на методах контролера. Для операцій читання вистачає базової перевірки на факт автентифікації, тобто доступ відкривається будь-якому залогіненому користувачу незалежно від його ролі. Мутаційні дії над довідниками, як-от створення чи редагування ресурсів і постачальників, обмежені виключно роллю ADMIN. Зі зміною стану замовлення ситуація тонша. Завершення та скасування заявки дозволені і ADMIN, і SUPPLIER, бо обидві ролі беруть участь у фінальних етапах життєвого циклу. А ось призначення постачальника до конкретного замовлення може зробити лише адміністратор, бо саме ця дія є точкою початку логістичної фази. Розміщення перевірок безпеки на рівні методів, а не глобально на рівні URL-конфігурації, було усвідомленим рішенням. Один і той самий URI цілком може потребувати різних прав залежно від HTTP-методу, скажімо, GET доступний усім авторизованим, а DELETE тільки адміну, і анотація на методі фіксує таке обмеження прямо там, де воно семантично доречне. Є одна неочевидна деталь, на яку легко натрапити випадково. Якщо в конфігурації не активувати @EnableMethodSecurity, всі ці @PreAuthorize-анотації залишаються

синтаксично валідними, проєкт компілюється без помилок, але фреймворк просто ігнорує їх під час виконання. І ніякого попередження при цьому не видає, що буває причиною прихованих діркок у безпеці.

ResourceApiController - зосереджено повний функціонал керування довідником ресурсів. По суті, це основа всіх закупівель. Тут можна робити все з записами - створювати, змінювати, деактивувати: створення, модифікація або переклад у неактивні проходить тут. Змінювати можуть тільки адміни, бо дані дуже важливі. Це нормально інтегрується з нашою системою прав доступу. Якщо цього не буде, клієнт не зможе нормально оформити замовлення - щось обов'язково схибнеться. Переглянути довідник може хтось із користувачів одразу після авторизації, але редагування можуть тільки адміни. Інакше буде повний бардак у системі.

Чому окремий ендпоінт для активних ресурсів існує як самостійний метод, а не як параметр загальної вибірки. Тут міркування суто прагматичне. Менеджер, що створює нову заявку, повинен бачити у формі вибору лише ті номенклатурні позиції, з якими справді можна оформити закупівлю прямо зараз. Архівовані чи деактивовані позиції у такому списку лише заважатимуть. Архівування своєю чергою винесене як окремий ендпоінт, а не оформлене як часткове оновлення поля, бо за своєю природою це не просто перемикання прапорця, а самостійна бізнес-дія. У неї свої наслідки для пов'язаних замовлень, які раніше посилались на цей ресурс, і для аналітики, яка враховує історичні дані.

SupplierApiController дзеркалить структуру ResourceApiController практично один в один, надаючи повний CRUD над довідником постачальників разом з підтримкою деактивації та зворотного відновлення записів. Така симетрія між двома контролерами зроблена свідомо. Розробник, який щойно розібрався з API ресурсів, фактично вже знає й API постачальників, бо інтерфейси будуються за тим самим шаблоном.

Окремо виділяється AnalyticsApiController - доступ до нього тільки на читання, працює виключно для ролі ADMIN. Єдиний маршрут у контролері: GET /api/analytics/dashboard, який повертає дані через DashboardMetricsDto. Всі метрики

зібрані в одному об’єкті не тому, що так стильно, а просто щоб уникнути безлічі окремих запитів. Клієнту не треба ходити по черзі за кожним показником - все приходить разом, одним завданням. Менше мерехтіння на екрані, менше навантаження, менше очікування. Коли навантаження падає - код стає легшим для читання, а всі дані потрапляють у браузер разом. DashboardMetricsDto об’єднує три групи цифр просто всередині себе. Загальна ціна активних замовлень лежить поряд із сумою тих, що вже закриті - все це фінансова частина. Стани заявок відображаються окремо: скільки на кожному етапі, хоч уже й прострочено постачання. Товари, запаси яких опустились нижче норми, видно в складському розділі.

У системі за скрізне опрацювання помилок відповідає клас GlobalExceptionHandler - його позначено анотацією @ControllerAdvice. Трапилось виключення десь глибоко - воно проникає назовні, потрапляє до цього обробника і там переробляється. Результат завжди один: структурована відповідь у форматі JSON, без винятків. Цей формат заданий класом ErrorResponse, де є рівно чотири поля. Статусний код HTTP знаходиться там разом із назвою проблеми словами. Далі йде пояснення конкретного провалу. Ще одне поле - мітка часу, оформлено за стандартом ISO 8601. Інакше не буває: кожна помилка, велика чи маленька, слухається через одну схему. Єдиний вигляд для всіх - це правило, а не можливість. Хоч там іде про помилку перевірки даних, хоч про відсутність запису - форма відповіді завжди однакова. Так само, коли логіка операції не пройшла. Клієнт очікує певного формату й без проблем його аналізує. У глибині системи виділяють чотири типи ситуацій. Якщо не знайдена сутність, тоді помикла 404. Неправильний аргумент у методі - помилка 400. Якщо ж дані не пройшли валідацію через анотації - теж 400, але з повним переліком полів, що провалились. Коли виникає ResponseStatusException, код автоматично підбирається на основі логіки операції - його вирішує сам шар служби. Інші проблеми, крім перших чотирьох типів, просто оселилися в базовому Exception із статусом 500. Щодо перевірки даних: система сканує всі поля, де щось не те, витягує окремі коментарі про помилки. Потім усе це з’єднується в одну стрічку, яка приходить назад. Тепер користувач одразу бачить

усі промахи, а не гадає, що далі поламалося.

Валідація вхідних даних на рівні контролерів зроблена через анотації Bean Validation, серед яких @NotNull, @NotBlank, @Positive, @FutureOrPresent. Активується вся ця перевірка анотацією @Valid або @Validated, поставленою прямо на параметрі методу, і спрацьовує ще до того, як об'єкт встигає дійти до сервісу. Якщо хоч одне з обмежень порушене, Spring сам викидає відповідний виняток ще на вході, а сервісному шару дістаються тільки гарантовано валідні дані. Це класична реалізація принципу раннього виявлення помилок, негодящі дані відсікаються ще до того, як знадобиться лізти в базу за чимось. Конкретно для класу запиту на створення замовлення прописані такі обмеження: ідентифікатор ресурсу й кількість обов'язкові, кількість має бути строго більшою за нуль, бажана дата постачання не може опинитись у минулому щодо поточного дня.

4.2 Реалізація бізнес-логіки

Сервісний шар це той рівень системи, на якому проектні рішення розділу 3 переходять зі схем у живий код. Кожен сервісний клас має чітко окреслену зону відповідальності, що не перетинається з зонами сусідніх класів. Завдяки такому поділу кожен сервіс тестується незалежно, а супровід окремих модулів не тягне за собою правок у всій системі. Усі сервісні класи позначені анотацією @Service і отримують свої залежності виключно через конструктор. Анотація @Transactional ставиться вибірково, лише на тих методах, які виконують кілька пов'язаних операцій з базою і повинні або всі завершитись успішно разом, або всі разом відкотитися. Методи з однією-єдиною читальною операцією залишаються без явного транзакційного контролю, бо зайвий @Transactional на них тільки додав би накладних витрат без жодної користі.

Серед усіх сервісів саме OrderService став головним - і наймасштабнішим випробуванням. Алгоритм для нових замовлень працює так: спочатку встановлюється їхній початковий стан, це робить метод createOrder. Якщо кількість менша або дорівнює нулю - процес зупиняється миттєво, система просто викидає

помилку. Потім береться потрібний ресурс прямо з репозиторію, далі порівнюється те, що хочуть, із тим, що взагалі є на складі. Коли ресурсів вистачає, система автоматично дає замовленню мітку PENDING_EXECUTION - і одразу ж вираховує потрібну кількість із доступного запасу. Ці дві події трапляються разом, всередині єдиної транзакції, тож неможливо, наприклад, зарезервувати товар без оновлення статусу чи навпаки.

Якщо ж його не вистачає, замовлення переходить у стан PENDING_SUPPLY, а наявні залишки залишаються непорушеними. Тут має значення один нюанс: клієнт ніколи не вказує бажаний статус у запиті. Початковий стан визначає лише сам сервіс, спираючись на фактичні дані, а не на те, що хочеться замовнику. Лише коли статус уже на місці, береться день за дату створення - і цю сутність записують через репозиторій. Те, що повертається, потрапляє до мапера, він формує кінцеву відповідь: туди входить ресурс разом із даними про постачальника.

Метод отримання замовлень поточного користувача повертає тільки ті записи, які прив'язані до ідентифікатора автентифікованого користувача, а сам цей ідентифікатор сервіс дістає прямо з контексту безпеки Spring. Окремий метод отримання конкретного замовлення додає до цього перевірку права доступу до самого запису. Менеджер бачить виключно ті заявки, які він особисто створив. Постачальник тільки ті, де він проставлений як відповідальний за виконання. Адміністратор має доступ до будь-якого замовлення в системі без обмежень.

Метод assignSupplier відповідає за призначення постачальника на конкретне замовлення. Перш ніж щось зробити, реалізація проходить двома перевірками. Заявка повинна бути у стані PENDING_SUPPLY, бо саме з цього стану й тільки з нього призначення взагалі допустиме. Сам постачальник теж повинен бути активним на момент призначення, деактивовані з вибірки виключаються. Якщо обидві перевірки пройдені, метод встановлює посилання на постачальника та через виклик validateStatusTransition переводить заявку в стан IN_DELIVERY, причому обидві ці зміни лягають в одну транзакцію. Сама логіка дозволених переходів між станами винесена в окремий приватний метод validateStatusTransition. Усередині нього лежить незмінна структура відображення допустимих переходів, описана у

підрозділі 1.4. Великий плюс такої декларативної структури проявляється на тлі альтернативи з ланцюжком if-else. Граф переходів читається прямо зі структури даних, без необхідності аналізувати умовну логіку, а додати ще один дозволений перехід можна виправленням буквально одного запису, не торкаючись жодного оператора умови. Якщо ж хтось намагається виконати недопустимий перехід, метод піднімає виключення з кодом HTTP 400, у повідомленні якого вказані обидва стани: поточний і той, у який намагались перейти.

Коли справа доходить до особливих статусів - наприклад, скасування чи завершення - все це обробляється в handleSpecialStatusCases. Замовлення, яке потрапляє у стан CANCELLED, одразу ж звільняє раніше заблоковані запаси; поновлення залишку завжди йде разом зі зміною статусу. Операції огорнуті одною транзакцією: або все проходить, або нічого - ніяких половинчастих результатів. Якщо ж замовлення переходить у COMPLETED, система записує поточну дату як момент закриття. Потім цю дату використовуватимемо для аналітики - щоб бачити, скільки насправді тривало виконання. Виносимо всю цю логіку в окремий метод, щоб вирішити одну проблему: якщо обробку побічних ефектів залишити у різних гілках коду, один із шляхів точно про неї "забуде". Але тут так не вийде. Кожен перехід тепер проходить через те саме місце. Метод addComment спирається на агрегатний патерн. Коментар опиняється всередині замовлення, прямо в його колекції. Потім JPA самостійно подбає про збереження - каскадом, без додаткових наказів.

ResourceService і SupplierService працюють майже однаково - обидва керують своїми довідниками. Перед тим як додати новий елемент, система перевіряє - чи не існує вже такого самого за найменуванням. Оновлення торкається лише тих полів, що прийшли у запиті, решта, наприклад ID чи час створення, залишається недоторканою. Архівація просто міняє стан прапорця «активний», повертаючи його назад за потреби. Коли запис працює - стає вимкненим. У разі коли призупинений - знову запускається. Цей подвійний напрям дозволяє обійтись без двох окремих точок входу, бо вся робота виконується через єдиний підхід. На відміну від ResourceService, SupplierService працює трохи інакше за своїми

реакціями. Видалення постачальника через прапорець активності не порушує зв’язки з іншими даними - у замовленнях посилання лишається. Хоча сам постачальник тепер позначений як неактивний, його старі записи недоторкані. Так сталося тому, що система обрала м’яке видалення, про що йшлося раніше, коли аналізували альтернативи. Для роботи з поточними даними зробили окремий запит, який автоматично фільтрує неактивні записи. Цей механізм працює там, де важливо показувати лише доступне для використання. Інакше - у внутрішньому інтерфейсі - задіюється інший підхід: просто повертається все, без приховання минулого стану. Адміністраторам потрібна повна картина, тож вони бачать кожен запис, навіть той, котрий вже не беруть до уваги звичайні користувачі.

AnalyticsService виконує виключно читальну роль у системі, жодних мутацій даних на цьому сервісі немає взагалі. Усередині він виконує серію спеціалізованих запитів до бази. Замовлення підраховуються за кожним станом через групування за відповідним полем. Фінансові суми рахуються через агрегатні функції як добуток кількості замовленої позиції на ціну ресурсу. Окремо робиться вибірка замовлень, у яких дата постачання вже минула, а заявка ще не закрита. Ще одна вибірка дістає з довідника ресурси, чий залишок упав нижче порогового значення. Усі ці окремі результати збираються в один зведений об'єкт DashboardMetricsDto, який далі йде клієнту єдиним пакетом. Винесення всієї аналітичної логіки в окремий сервіс мало принципову причину. Аналітика за своєю природою це чисте читання даних, без потреби в транзакційному контролі мутацій. Тримати її разом із мутаційними сервісами означало б змішувати дві принципово різні моделі роботи з базою, що ускладнило б і конфігурацію транзакцій, і подальший супровід.

OrderSpecifications відповідає за динамічне складання умов вибірки через Criteria API. На кожен критерій фільтрації заведено окремий статичний метод, який формує елементарну умову порівняння конкретного поля сутності із значенням, переданим з ззовні. Якщо параметр у запиті відсутній, відповідна умова в фінальний SQL просто не потрапляє. Зведення окремих умов докупи відбувається вже в сервісному шарі, де вони з'єднуються через логічні операції в потрібному порядку. Така схема нормально працює з будь-якою комбінацією переданих і

відсутніх параметрів. Якщо в майбутньому з'явиться новий критерій фільтрації, додавати треба буде тільки новий метод у самому класі специфікацій. Інтерфейс репозиторію залишається недоторканим.

Перетворення між JPA-сутностями і об'єктами передачі даних зроблене через маппер-інтерфейси з бібліотеки MapStruct. Один важливий нюанс реалізації стосується маппінгу від запиту до сутності. Поля, які встановлюються самою системою, серед них ідентифікатор, стан, аудитні дати, зовнішні зв'язки, явно виключені з процесу маппінгу. Це принципово, бо клієнт передає у запиті лише ідентифікатор ресурсу та кількість, а решту атрибутів проставляє вже сервіс на власну відповідальність. Якби ці поля не були виключені, маппер перетер би системні значення тим, що прийшло від клієнта, з усіма наслідками. Зворотний маппінг сутності в об'єкт відповіді використовує вкладені правила, які дозволяють включити вибрані поля пов'язаних сутностей прямо в плоскій структурі відповіді.

Контроль над версіями схеми бази даних у проєкті забезпечується завдяки Flyway [13], який використовує три міграційні скрипти, впорядковані за чергою. Замість загального опису - перший із них містить сукупність DDL-команд: створення таблиць, налаштування первинних і зовнішніх ключів, формування складеного індексу на полях стану та дати поставки. На другому етапі до центральних таблиць приєднуються колонки для аудиту, водночас реалізується лічильник версій задля оптимістичної блокувальної стратегії. Потреба в третьому кроці виникає через розбіжності - тип дати поставки потребує корекції, адже Hibernate трактує його інакше, ніж СУБД. Щойно міграція запускається, Flyway одразу генерує її хеш і фіксує значення в спеціальній внутрішній таблиці. Коли хто-небудь модифікує вже задіяний скрипт, під час чергового запуску Flyway виявляє невідповідність контрольних сум - процес просто не почнеться, замість цього з’явиться повідомлення про проблему. Такий підхід перешкоджає тому, щоб у різних середовищах опинилися різні стані баз даних, якби хтось ненароком правив минулий файл міграції.

4.3 Реалізація системи безпеки та авторизації

Підсистема безпеки в проєкті побудована на взаємодії трьох незалежних компонентів. Перший це конфігурація Keycloak як зовнішнього сервера ідентичностей. Другий це SecurityConfig, який налаштовує ланцюжок фільтрів Spring Security всередині самого застосунку. Третій це KeycloakRealmRoleConverter, що приводить формат тверджень JWT-токена від Keycloak до того, що очікує побачити стандартний механізм фреймворку. У кожного з цих компонентів своя відособлена зона відповідальності, і кожен можна замінити чи переписати, не торкаючись інших.

Для налаштування сервера ідентичностей я використовую файл agriculture-realm [16] в JSON форматі. Усередині цього файлу описано три речі. Клієнт agriculture-client з конфіденційним типом доступу та явно вказаним URI перенаправлення після успішного входу. Перелік трьох ролей реалму: ADMIN, MANAGER, SUPPLIER. Параметри термінів дії токенів, як для access-токенів, так і для refresh-токенів. Файл зберігається як артефакт проєкту і монтується в контейнер Keycloak через Docker volume. Завдяки такому підходу можна легко відтворити всі налаштування в різних середовищах. Коли хтось буде розгортати систему на новій машині, все налаштується автоматично - точно так само, як у мене, без жодного клацання в адміністративній консолі Keycloak. Ролі визначено саме на рівні реалму, а не на рівні клієнта, і це свідоме рішення. Ролі реалму потрапляють у токен через стандартну структуру тверджень і не треба писати додаткові перетворювачі на клієнті. Параметри підключення до Keycloak передаються в застосунок виключно через змінні середовища, у вихідному коді жодного хардкоду адрес чи секретів немає.

Ланцюжок фільтрів у SecurityConfig формується завдяки функціональному DSL - HttpSecurity [15]. Вирішальне значення тут має саме стратегія роботи з сесіями. Стан автентифікації між запитами сервером не зберігається ніколи. Кожен окремий запит проходить перевірку лише за допомогою токена, який клієнт передав разом із ним. Саме така поведінка, позбавлена стану, створює реальну

можливість для масштабування системи вгоризонт. Один із екземплярів бекенду спроможний опрацювати будь-що без додаткових домовленостей про сесійні дані. Ще одна вигода stateless-підходу - природна ізоляція від CSRF-атак. Без активної сесії на сервері такі вторгнення просто не працюють. Вимкнення механізму захисту тут не помилка, а логічний результат дизайну системи. Налаштування CORS визначено окремо. Доступ отримують лише домени клієнтського середовища. Правила чітко фільтрують зайві запити. Разом із реальним застосуванням методів на боці сервера дозволяються лише потрібні HTTP-дії, у тому числі OPTIONS - це важливо для перевірок від браузера перед основним запитом. Заголовки ж містять Authorization, адже без нього не обійтися при роботі з токенами, а також Content-Type, оскільки формат даних має бути чітко визначений як JSON.

Натомість конфігурація сервера OAuth2 спирається на ендпоінт публічних ключів Keycloak [14, 17, 18], де беруться дані для перевірки JWT. Важливим моментом стає використання саме RS256 - не симетричного, а асиметричного методу шифрування. У системі лишається тільки публічний ключ; більше нічого не потрібно, щоб переконатися в походженні токена. Той самий токен із такими ж даними просто не вдасться сфабрикувати: без доступу до приватного ключа підпис буде недійсним. Його ніхто не передає далі - він весь час залишається всередині Keycloak. Цей ендпоінт має одну корисну особливість - можливість автозаміни ключів. Коли в Keycloak створюють нову пару ключів, бекенд під час чергової проваленої перевірки самостійно завантажує актуальний ключ із JWKS-набору, навіть якщо конфігурація застосунку лишається без змін. У Spring Security отриманий публічний ключ тимчасово зберігається локально, а поновлення відбувається лише після невдалих спроб валідації. Через такий механізм взаємодія з Keycloak скорочується до мінімуму під час типового обслуговування запитів, що значно менше навантажує мережу. Кожен вхідний JWT-токен обов’язково проходить чотири незалежні контрольні процедури. Починається все з перевірки підпису - його звірять із відкритим ключем. Час життя токена має не спливати: система зрівнює його з актуальних UTC-позначкою на сервері. Важливо, щоб адреса емітента збігалася з посиланням на потрібний реалм; інакше - сторонній

Keycloak, і це проблема. Ще треба знайти ID аудиторії, який підтверджує: токен створений спеціально для нашої системи. Хоч один збій серед цих кроків - одразу відмова через код 401; далі, у логіку операцій, запит уже не проникає.

KeycloakRealmRoleConverter розв'язує суто структурну несумісність між форматом ролей у JWT-claims від Keycloak [15, 22] і тим, як Spring Security очікує побачити ці ролі. Стандартний конвертер з коробки шукає ролі у плоских твердженнях токена, тобто прямо на верхньому рівні. Keycloak же кладе ролі реалму у вкладену структуру realm_access.roles, на один рівень глибше. Через цю невідповідність без власного конвертера будь-яка перевірка через @PreAuthorize просто не спрацьовує, як би правильно вона не була написана. Конвертер дістає вкладену структуру тверджень, витягує з неї список ролей і перетворює кожну роль на об'єкт повноваження з префіксом ROLE_ у верхньому регістрі. Префікс цей не довільний, така конвенція жорстко вшита в Spring Security для роботи методу hasRole(). Загальну схему ланцюжка фільтрів і шлях, який проходить токен крізь компоненти безпеки, можна побачити на рисунку 4.10.

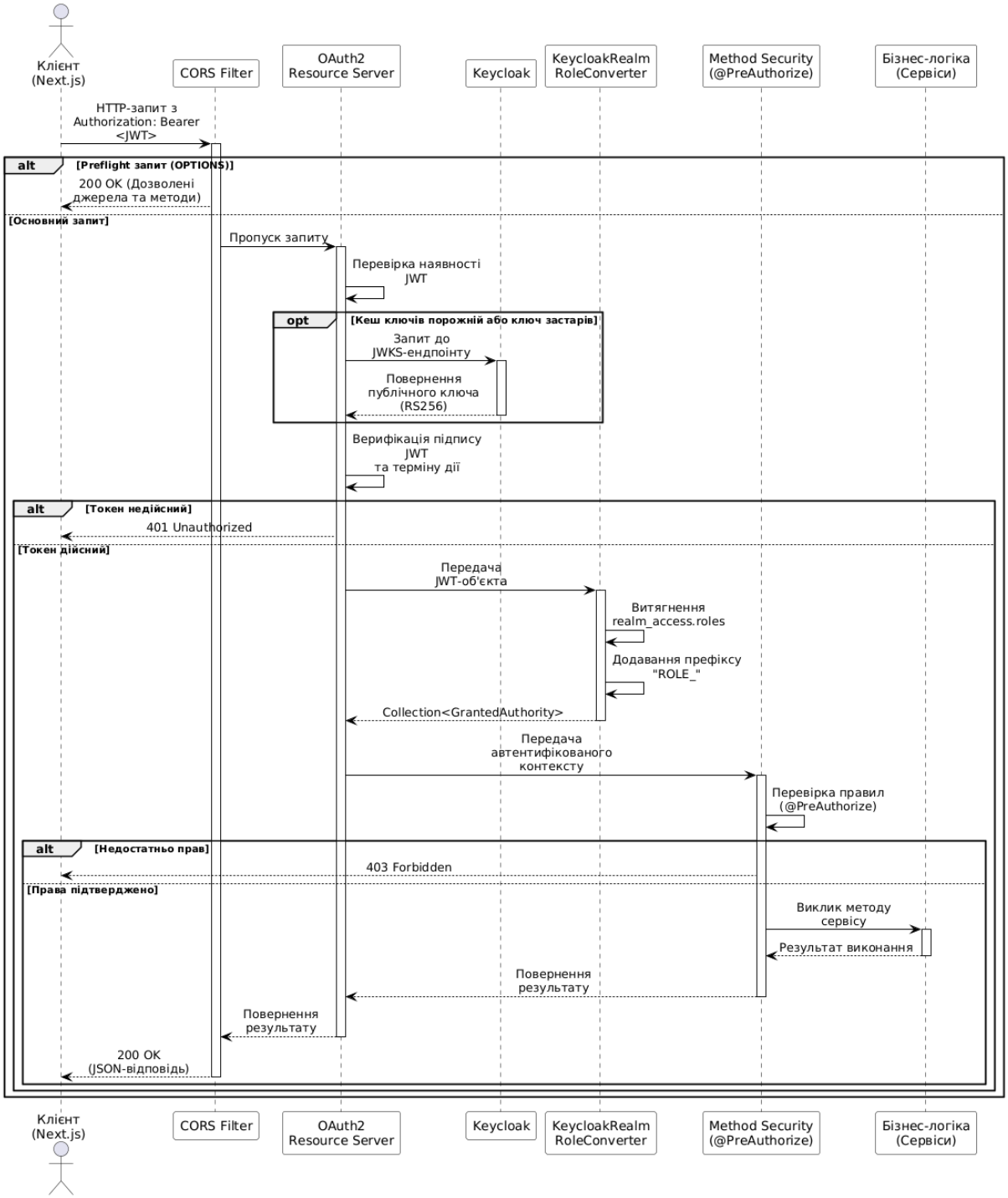


Рис. 4.1 Діаграма послідовності. Схема ланцюжка фільтрів Spring Security та шлях JWT-токена до бізнес-логіки

Рішення застосовувати анотації прямо на рівні методів, щоб контролювати доступ до операцій [22], замість того, щоб покладатися на глобальну фільтрацію через URL, стало важливим кроком. Це спирається не на теорію - досвід підказав такий напрямок. Коли мова йде про один і той самий URI, права кардинально різняться в залежності від того, що робить користувач: клієнт може читати дані

через GET, маючи базову роль, однак вилучення цих самих даних командою DELETE вимагає найвищого рівня доступу. Коли налаштовуєш по URL, то не бачиш, що саме хоче зробити користувач. Зате анотація, прив’язана до конкретного методу, бачить усю картину. Так працює точність. Але тут є нюанс - для ендпоінтів постачальника важливо переконатися, що замовлення належить саме йому. Замість стандартної схеми, спершу отримують ідентифікатор користувача через контекст безпеки. Перш ніж виконувати головну логіку, система порівнює цей ідентифікатор з тим, який вказано в замовленні як постачальник. Такий підхід дозволяє швидко відсіяти несанкціоновані запити.

Ззовні сервер автентифікації передає дані - всередині програма має свою базу. Під час першого входження людини система перевіряє, чи є у ній уже цей ідентифікатор, взятий безпосередньо з токену. У разі його відсутності запис про користувача генерується автоматично, ґрунтуючись на імені та email-адресі з тіла токена. Кожен компонент має свою чітку роль. У Keycloak зберігається вся інформація про особу - це основний реєстр. Локальна таблиця в базі додатку має обмежене завдання: пов’язувати дані з автором. Зв’язок створюється під час першого входження, більше таке не повторюється. Потім система працює без зайвих запитів. Навантаження через цей процес практично відсутнє. Перевірка стану відбувається один раз - подальші кроки не потрібні.

4.4 Реалізація клієнтської частини

Клієнтську частину ми зробили як окремий Next.js-застосунок, який працює з сервером тільки через HTTP API. На рівні реалізації тут вирішуються дві принципово різні групи задач. Перше - це підключення до Keycloak, щоб керувати сесіями користувачів. Друге - створення інтерфейсу для кожного модуля окремо. Обидві групи задач реалізовані з урахуванням специфіки App Router. За замовчуванням всі сторінки тут - це серверні компоненти, і це базова поведінка. Якщо ж компонент має містити клієнтську інтерактивність, його доводиться явно позначати директивою 'use client' на самому початку файлу.

Через бібліотеку next-auth реалізовано підключення до Keycloak - усе базується на самостійно налаштованому OAuth 2.0-провайдері всередині файлу маршруту для входу. Автоматичне виявлення точок взаємодії не використовувалось, замість нього чітко прописані шляхи до сторінок: авторизації, оновлення токена й профілю користувача. Особливість такої схеми ховається всередині роботи з контейнерами - справа в тому, що адреси неспівпадаючі. Коли користувач намагається увійти, його браузер пересилає запит на зовнішній порт Keycloak, адже саме через нього система доступна ззовні. Заміна токенів - справа за серверним кодом Next.js, який працює через внутрішню адресу в Docker. Це навмисне так влаштовано, бо браузер просто не побачить служби всередині Docker-мережі, а от Node.js там себе почуває стабільно. Ролі людини беруться прямо з даних у JWT-токені, без зайвих кроків. Техніка проста: виділяється потрібна частина токена, аналізується структура прав у межах реалму, далі ці дані потрапляють у сесію завдяки системі колбеків у next-auth. Щоб все це не плуталося пізніше, тип сесії доповнюють у окремому файлі TypeScript - і перевірка правильності полів проходить ще перед запуском, а не під час роботи. Згадаємо ще один момент - SessionProvider. Він працює як окремий клієнтський шматочок коду. Через це головна структура сторінки лишається на сервері. Завдяки цьому автоматично отримуємо всі переваги серверного рендерингу там, де необхідно.

HTTP-клієнт на стороні клієнта реалізовано за допомогою Axios. Замість того щоб фіксувати URL серверного API, система отримує його значення зі змінної середовища - це дозволяє використовувати один і той самий код у різних режимах: під час локальної розробки, у тестовому стенді або на боці продуктиву. Перед відправкою будь-якого запиту спеціальний перехоплювач автоматично перевіряє стан автентифікації через next-auth; коли знайдено дійсний токен, до запиту додається заголовок Authorization у форматі Bearer. Кожна сторінка при цьому просто викликає методи клієнта - про управління авторизацією нічого не знає, оскільки вся логіка ізолюється всередині перехоплювача. Коли сервер надсилає код, який показує - токен більше не дійсний, срабатывает перехоплювач вхідних відповідей. Тепер користувач одразу потрапляє на екран авторизації, минуючи

зависання у невизначеному режимі. Усередині модуль поділено на тематичні блоки: аналітика, постачальники, ресурси, операції з замовленнями. Саме ця організація повторює логіку серверного API, через що в клієнтському коді простіше шукати потрібне.

Сторінка замовлень за кількістю реалізованих сценаріїв вийшла найскладнішим компонентом клієнтської частини. Внутрішній стан компонента тримає чотири речі одночасно: масив замовлень, прапорець завантаження, поточні значення активних фільтрів і ідентифікатор відкритого модального вікна, якщо воно є. У таблиці для кожного запису виводиться кілька стовпців: назва замовленого ресурсу, кількість, бюджет заявки, термін постачання з кольоровим маркером, ім'я постачальника та поточний стан замовлення. Сам бюджет рахується тут же на клієнті, як добуток кількості на ціну одиниці, яка приходить у вкладеному об'єкті відповіді сервера. Відображається ця сума у форматі гривні через стандартний механізм локалізованого форматування числа. Термін постачання отримує кольорове кодування залежно від того, скільки днів залишилось до бажаної дати. Понад чотирнадцять днів дає зелений колір. Від семи до чотирнадцяти фіолетовий. Від трьох до семи помаранчевий. Менше трьох червоний. Така візуалізація дає змогу одразу побачити критичні позиції в списку, без потреби читати дати й рахувати дні вручну.

Для фільтрації замовлень на сторінці передбачений окремий розкривний блок, у якому користувач задає три параметри: початкову дату періоду, кінцеву дату і стан замовлення. Поведінка тут залежить від того, заповнено хоч щось чи ні. Якщо вибрано хоча б один з параметрів, запит уходить на ендпоінт серверної фільтрації. У випадку, коли всі поля порожні, клієнт просто звертається за повним переліком без жодних умов. Сортування за номером замовлення натомість виконується не на сервері, а вже на клієнтській стороні, після того як дані прийшли в браузер. За цим вибором стоїть конкретна інженерна логіка. Фільтрація за датою і станом це робота, яка повинна йти на стороні бази, бо потенційно йдеться про великі обсяги записів, через які прокручувати в браузері було б повільно. Сортування за числовим ідентифікатором це натомість тривіальна операція над уже

завантаженим масивом, посилати її назад на сервер не має сенсу. Для роботи з конкретним замовленням передбачено чотири різні модальні вікна. Створення нової заявки виглядає як форма з трьох полів: вибір активного ресурсу зі списку, числове поле для кількості і поле дати з обмеженням, що минулі дати в ньому вибрати неможливо. Призначення постачальника відкривається лише для адміністратора і містить випадаючий список тільки активних партнерів. Зміна стану замовлення відображає виключно ті переходи, які допустимі з поточного статусу заявки, без можливості вибрати недозволений. Бічна панель коментарів виводить хронологічну стрічку повідомлень, а в самому низу панелі стоїть поле для введення нового коментаря.

На двох окремих сторінках - одна про ресурси, інша про постачальників - працюють за тією ж схемою. Хоча контент різниться, загальна форма збігається до найменшої деталі. Скрізь бачимо таблички зі списками, де кожен елемент доступний для огляду чи корекції через спливаюче вікно. Також передбачено окрему операцію, щоб вмикати чи вимикати запис. Якщо стан неактивний, рядок лишається на своєму місці, тільки виглядає інакше - так, що очі одразу помічають цю відмінність. Нічого не приховується назавжди, просто виділяється трохи інакшим способом. Замість клавіші деактивації у неактивних записів з’являється кнопка відновлення - нею можна повернути елемент назад у дію. Після кожної зміни, будь то новий запис, правки чи зміна статусу, відразу йде запит до API за свіжими даними. Тим самим забезпечується повна синхронізація між тим, що показано на екрані, і справжнім станом даних. Цікаво працює й спливаюче вікно для додавання та редагування. Формально це один компонент у React, але працює він по-різному - все залежить від параметрів, отриманих під час відкривання. Заповнена форма з'являється тоди, коли система завантажує початкові дані з певного запису - у такому разі вона працює як інтерфейс для редагування. Без наявних даних поля лишаються чистими, а екран перетворюється на простір для нового введення.

Сторінка аналітичного дашборду закрита для всіх ролей крім ADMIN. На вході перевіряється сесія, і якщо в ній немає ролі адміністратора, користувача

одразу переадресовує на сторінку замовлень, навіть не намагаючись щось рендерити. Доступ до самої сторінки відкривається тільки для ролі ADMIN. Сам дашборд при завантаженні робить один-єдиний запит до аналітичного ендпоінту, а потім показує отримані показники через візуалізації з бібліотеки Recharts. Розподіл замовлень за станами зображається через кругову діаграму, де видно частку кожного статусу в загальній масі заявок. Рейтинг постачальників за кількістю виконаних замовлень виводиться через горизонтальну стовпчасту діаграму, форма якої добре підходить для порівняння рангованих значень. Блоки прострочених замовлень та ресурсів з критично низьким залишком виведені у вигляді структурованих таблиць з пагінацією, бо для переліків такого типу таблична форма читається легше за будь-який графік. Загалом вибір типу візуалізації під кожен блок зроблений усвідомлено, виходячи з природи самих даних. Для часток від цілого підходить кругова, для порівняння рангованих значень горизонтальна стовпчаста, для переліків з кількома атрибутами на запис таблиця.

5. ТЕСТУВАННЯ

5.1 Стратегія та методологія тестування

Тестування починають не наприкінці, а тримають у процесі - постійно оновлюють дані про стан системи через весь цикл створення. Базу становить звичайна піраміда тестів: чіткий поділ за рівнями складності й обсягом потужностей для прогону. Знизу - багато окремих коротких перевірок службових частин, швидко дають результат. Десь посередині - інтеграція блоків, сполучення функцій між собою. Угорі ж - випробування взаємодії з мережею, де одночасно додаються правила доступу й ролі, щоб не гаяти час на окремі сесії аналізу захисту.

Тестування на модульному рівні спрямоване головним чином на окремих сервісних компонентів, які працюють без залучення суміжних підсистем. Замість реального оточення, середовище наповнюють віртуальними макетами компонентів, що дозволяють керувати поведінку. Час на виконання такої серії тестів відбувається майже миттєво - від сили пару десятків мілісекунд. Така легкість зумовлена тим, що система не витрачає ресурси на підйом усієї програмної платформи, чи встановлювати зв'язок із базами даних. Без цих накладок автотести виконуються у безперервному режимі, безпосередньо у процесі програмування.

Перевірка інтеграційних процесів охоплює дві ключові підходи. Замість повної інфраструктури - тут працює MockMvc, щоб моделювати HTTP-запити прямо в серцевині Spring MVC. Цей спосіб торкається лише мінімального набору компонентів: маршрутизація, аутентифікація, формування відповідей. Інакше буває, коли потрібна справжня взаємодія з БД - у цьому разі на допомогу кличуть Testcontainers [25]. Контейнер із базою піднімається в Docker безпосередньо на час прогону тестів. Так забезпечується реалістичне оточення для перевірки доступу до даних. Очевидно, у такій самій послідовності перевіряють контролер авторизації через позначку @WithMockUser. Помилка 403 з’являється тоді, коли хтось без прав пробує щось зробити. Це нормальна реакція сервера на спробу вторгнення.

Під час вибору інструментарію керувалися сумісністю з навколишнім середовищем на основі Spring Boot, а також активною участю спільноти користувачів. Основна робота з перевірок лягла на JUnit 5 (Jupiter) [23, 24], адже він тримає крок із тим, що зараз працює найкраще. Назви кожного окремого тесту стали зрозумілішими через @DisplayName - журнал перевірок таким чином не нагадує шифровку. Для конструювання тестових аналогів при тестуванні окремих модулів інтегрують інструментарій Mockito, завдяки чому можна перевіряти цільовий метод у чистому вигляді. Запускати всю програму для одного контролера немає потреби: @WebMvcTest із Spring Boot Test сам складе потрібний фрагмент навколо. У парі з MockMvc працює Spring Security Test, який моделює процес входження. Для прикладу, @WithMockUser створює уявного юзера, отже перевірку маршрутів можна проводити за різними рівнями доступу, не запускаючи реальний Keycloak-сервер.

Тестовий набір оцінюється через три ключові аспекти. Основний - це бізнес-логіка: кожне правило на рівні сервісу має окремий тест і на успішний, і на помилковий випадок. Інший - крайні ситуації: передбачені нульові об’єми, менше нуля, прогалина у тексті, ID, яких немає взагалі. Ще один - права користувачів: будь-яка зміна проходить перевірку чи дозволена роль може її зробити, тоді як заборонена - не зможе її виконати. Усього реалізовано 58 сценаріїв. Розподілені вони порівну між вісьмома групами. Чотири класи тести для логіки: там 31 приклад. Решта чотири - для контролерів, де стоїть 27 інтеграційних перевірок.

5.2 Модульне тестування (Unit Tests)

Перевірка окремих частин коду стосується переважно логіки сервісів - місця реалізації логіки завдання, прийняття рішень і обробки наслідки дій. Замість запуску цілого оточення Spring чи підключення до сховища даних, використовують об'єкти-заглушки, що моделюють реальну взаємодію компонентів. Ці заглушки налаштовуються для кожного окремого випадку, забезпечуючи контроль над результатами. Завдяки цьому увага залишається на його внутрішній логіці, а не на

зовнішніх залежностях. Кожна група тестів використовує можливості Mockito, що дозволяє уникнути повної активації середовища Spring. У таких класах явно визначаються макети залежностей, далі створюється екземпляр сервісу, куди передаються ці замінники. Перед початком окремого тесту викликається процедура, що формує дані до належного формату, робивши їх готовими до використання протягом усіх перевірок.

Найважче у системі працювати з OrderService - через це потрібно задіяти дванадцять тестів. Причина в тому, що всередині діють різні правила для прийняття рішень, виникають додаткові наслідки, а також йде спілкування одразу з трьома сховищами. Коли перевіряють створення заявки, беруть до уваги чотири ситуації, які разом описують усі можливі початкові умови. У першому випадку: товару на складі вистачає - система має записати замовлення як очікуване, плюс знизити доступний обсяг саме на ту кількість, яку вимагали. Перевірка стану містить у собі аналіз виникнення побічних ефектів - так, метод збереження активу із новим значенням фіксує, чи коригування обсягу відбувається всередині однієї транзакції. Тестування замовленні під час дефіциту товару проходить окреме тестування. Тут статус змінюється на "замовлення очікує", проте сам запас залишається незмінним. Відсутність операції запису підтверджує, що система не реагує зайвими діями. Для граничних умов передбачено власні тести: нуль, від'ємні значення - відсікаються автоматично, те саме стосується неможливого ресурсу. Якщо хтось відправляє дані відразу на сервер, минаючи інтерфейс додатка, саме ці тести зупинять некоректне введення.

Перевірка способу призначення постачальника будується навколо аналізу умов виконання дії. У разі коректного стану замовлення система має перемкнути його статус на "у процесі доставки" одночасно прив’язуючи до потрібного постачальника. Якщо замовлення вже закрито, така спроба блокується і повертає чіткий код невдачі. Переважна частина тестів стосується зміни статусів: кожен правильний перехід запускає оновлення, тоді як заборонена зміна залишається без змін. Окрім основного потоку, детально вивчаються наслідки скасування - щоб усе відбувалось без зайвих помилок. Ситуація з замовленнями, які були у черзі на

виконання, передбачає повернення всіх невикористаних ресурсів прямо на склад. Якщо ж скасування відбувається на етапі "очікується поставка", обсяг ресурсів не змінюється. Перед остаточним закриттям кожне замовлення проходить перевірку, де аналізується правильність проставленої дати завершення. У той самий час тестують також можливість залишити коментар, який має опинитися в картці агрегата з указаним користувачем та чітко оформленим текстом.

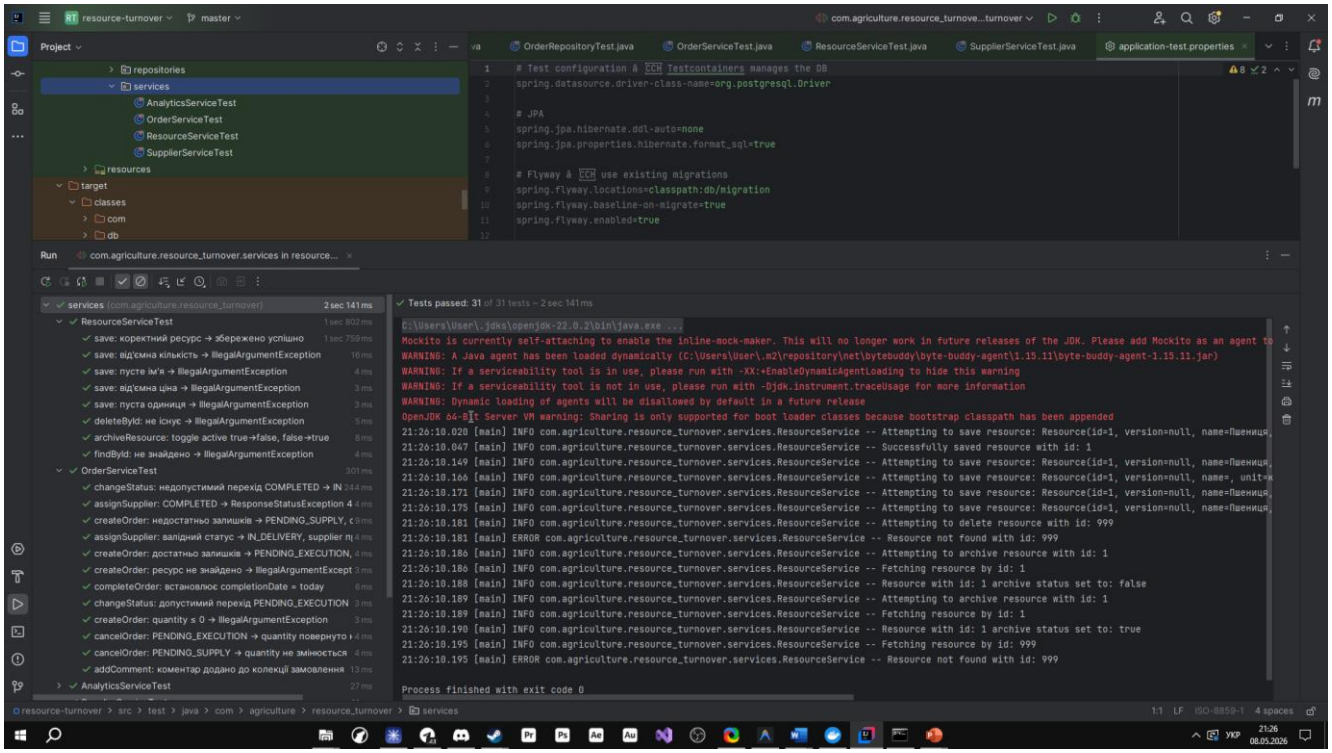


Рис. 5.1 Модульне тестування сервісів. Частина 1

Перевірка ResourceService спрямована на контролювання правильності вхідних даних - ключового елемента бізнес-логіки для збереження цілісності довідника. Вісім шляхів перевірки виконуються окремо, хоча мета всіх - поле має потрапити в межі. Наприклад, кількість матеріалу не може бути нижче нуля, бо система це блокує одразу. Основний приклад вдалого запиту стає точкою відліку для решти ситуацій. Окрім числових значень, важливо, щоб одиниця виміру була чітко прописана, інакше все руйнується. Натомість значення проходять через чітко визначені перевірки. Кожна дія має бути точною - це тримає всю систему на плаву. Один зламаний компонент і помилка повзе далі, майже невидимо. На порозі

система зупиняє спад за допомогою спеціальних обмежень. Усі внутрішні етапи працюють синхронно, попри те що анотації JPA теж втручаються. Навіть коли хтось намагається обійти маршрут, система лишається замкненою. На запит щодо старих даних система вказує: працює чи ні. Коли процес уже йде, результатом стане лише позначка про стан. Значення повертається тоді, якщо процес починають спочатку. Усередині одного сценарію така перевірка робить саму дію зрозумілішою.

Тестування сервісу SupplierService побудоване за аналогічним принципом, однак тут розроблено всього шість тестових сценаріїв, оскільки процеси взаємодії з постачальниками не мають складних розгалужень. Важливим аспектом тестування є логічне видалення записів, де перевірка гарантує, що запис не ділиться на фізичному рівні - а лише переводиться у стан прихованого або архівного елемента. Паралельно з цим, тест на відновлення даних фокусується на тому, чи повертаються дані у загальний контур відображення. Таким чином комбінація тестів повністю закривають сценарії управління видимістю даних, а саме архівацію та повернення в експлуатацію.

Найбільш комплексною частиною є AnalyticsService, у якому усього один публічний метод збирає масиви даних із багатьох точок, а далі розкладає отримані сутності за допомогою нелінійних алгоритмів групування. Перевірка даного модуля базується на п'яти тестових сценаріях, кожен з яких відтворює унікальну комбінацію вхідних даних та контролює правильність побудови кінцевого аналітичного звіту. Найпростіша перевірка не підтверджує бездоганність роботи всього ланцюжка, але він чітко верифікує архітектуру згенерованого об'єкта та наявність у ній обов'язкових атрибутів. Окремий сценарій відтворює поведінку системи за відсутності записів у БД, спрямований на валідацію поведінки за нульової вибірки. В результаті чого всі лічильники прибутку повертаються до початкового нульового стану, а вихідні таблиці пустують, що допомагає підтвердити захист від виникнення помилок нульового вказівника. Окремо розглядається ситуація з відсутніми даними в обчислювальних полях - логіка сервісу передбачає автоматичну заміну пустих значень на нуль. Замість банального

порівняння очікуваних результатів із фактичними, аналізується чутливість зведеної таблиці до обліку критично малих обсягів ресурсів. Окремим пунктом у цьому ж модулі виступає аудит прострочених днів, адже кожне отримане значення математично виводиться з дат оригінальних записів.

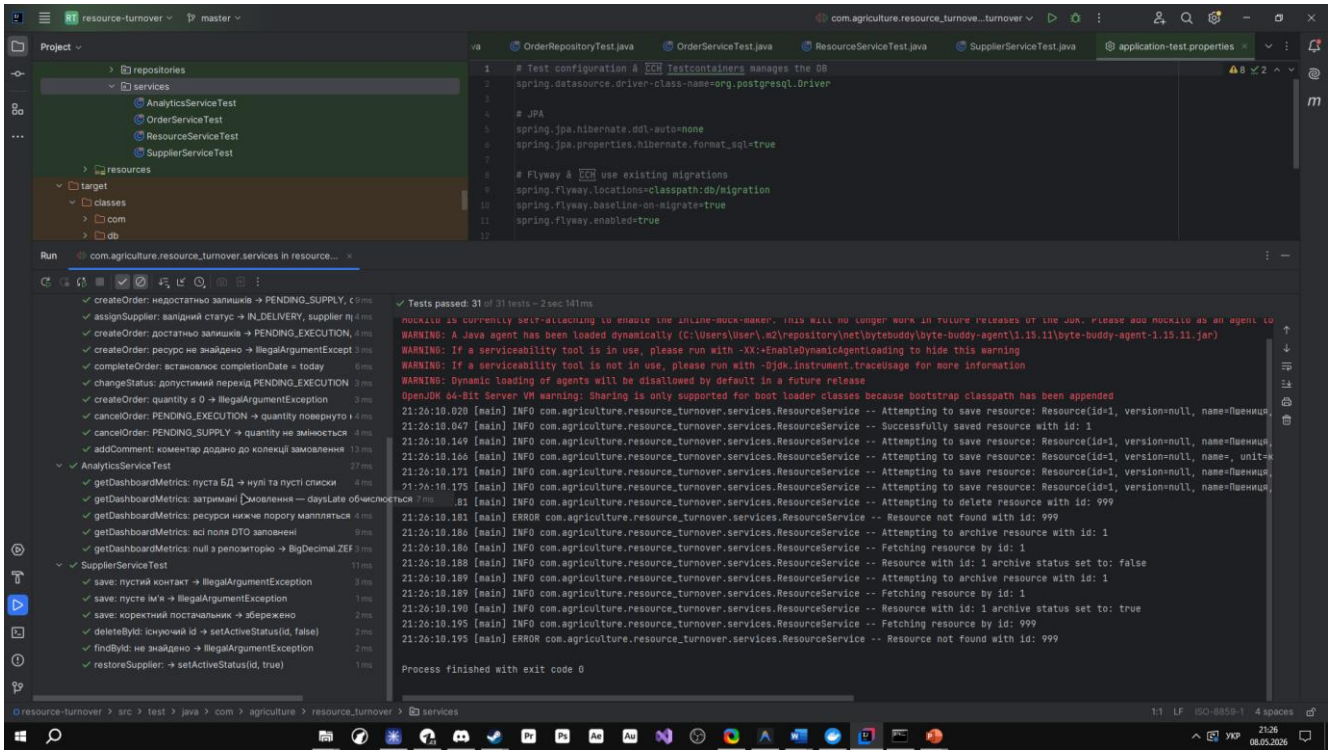


Рис. 5.2 Модульне тестування сервісів. Частина 2

5.3 Інтеграційне тестування

Головна мета інтеграційних тестів полягає у валідації стикування та спільних процесів між окремими сервісами, особисто в умовах живого розгортання програми. Замість того щоб відділяти елементи одна від одної, перевіряють весь ланцюжок HTTP-запиту всередині додатка. У цьому процесі беруть участь не тільки Spring-маршрути, однак також верифікація за допомогою токенів та опрацювання надісланих даних. Лише на цьому етапі видно: чи код відповіді правильний, чи результат виклику збігається з прогнозованим. Тут часто виникають ускладнення через хибні адреси або несправні конвертери. Латентні помилки в коді часом оминають базові внутрішні фільтри валідації. У подібних

ситуаціях саме такий вид аналізу показує найвищу ефективність. Коли технічні межі починають функціонувати некоректно - метод одразу виділяє точку збою.

Саме так працює система з @WebMvcTest - вона просить фреймворк запустити міні-середовище, призначене тільки для тестування вебшару. Замість справжніх сервісів підставляють заглушки, тому результат не стрибає через сторонні фактори. Наприклад, коли нема доступу до БД або сервер недоступний, перевірка все одно проходить чітко. Без зайвого шуму навколо Keycloak чи подібних систем, налаштування лишаються простими й прозорими. Особливий клас TestMethodSecurityConfig береться за анотації типу @PreAuthorize саме в тестах, активуючи їх локально. Перевірити правила доступу можна без підключення до сервера. Через тестування взаємодії з базою, запускається окремий контейнер із PostgreSQL - усе це організує Testcontainers. Коли контейнер працює, Flyway негайно застосовує міграції. Так повторюється точна копія реальної структури бази.

Серед усіх точок входу найбільше завдань покладено на OrderApiController - саме тому довелося створити десять різних інтеграційних тестів. Кожна кінцева точка аналізується послідовно, причому обов’язково перевіряються випадки успішної автентифікації та ситуації, коли доступ заблоковано через ролеві обмеження. Перевірити отримання списку замовлень допомагає такий підхід: клієнт проходить реєстрацію, надсилає запит - система повинна видати код 200 разом з масивом даних. Такий механізм одночасно гарантує працездатність маршрутизації, а ще переконує, що Jackson коректно опрацьовує серіалізацію. Поряд з цим дотримуються базові норми безпеки для доступу до маршруту. Перевірка процесу створення замовлення відбувається через пару тестів, спрямованих на контроль прав користувачів - один аналізує успішний запит від менеджера, інший моделює заборону для постачальника. Такі ж обмеження діють на кінцевій точці вибору постачальника: адміністратор працює без перешкод, у разі менеджера система поверне помилку. У той же час набір тестів доповнили сценаріями, що охоплюють весь цикл замовлення. Вони включають зміну станів, фіксацію закриття чи скасування, взаємодію з фільтрами пошуку, а ще додавання

приміток.

Для покриття логіки ResourceApiController і SupplierApiController розроблено дві окремі серії перевірок, по сім сценаріїв у кожній. Конфігурація перевірок є абсолютно ідентичною, оскільки самі класи мають однакову бізнес-структуру. Сценарії охоплюють весь життєвий цикл сутності на основі прав доступу із жорстким контролем прав користувачів - будь-який обліковий запис із діючим токеном, може вільно вичитувати дані. Тоді як операції запису, редагування та умовного стирання інформації вимагають виключно прав адміністратора. Сценарій створення об'єкта ідеально ілюструє роботу наскрізного ланцюжка додатка: вхідні JSON-дані перетворюються на об'єкт запиту, проходить етап маппінгу в сутність, фіксується в базі через сервіс, а потім дзеркально конвертується у фінальний результат зі статус-кодом HTTP 201. Водночас сценарій відновлення профілю постачальника, контролює правильність роботи двостороннього логічного стирання.

Пул перевірок для AnalyticsApiController складається всього з трьох контрольних сценаріїв, що зумовлено простотою самого класу, який експонує лише одну точку входу. Перевірка з правами адміністратора охоплює максимальний сценарій виконання: у відповідь сервер повертає успішний код 200, разом із повним набором фінансових та операційних метрик. Окремо перевіряється авторизація користувача з токеном менеджера: система успішно пускає його в аналітичний контур, згідно з бізнес правилами. Тоді як користувачі з правилами постачальників, закономірно отримують помилку авторизації.

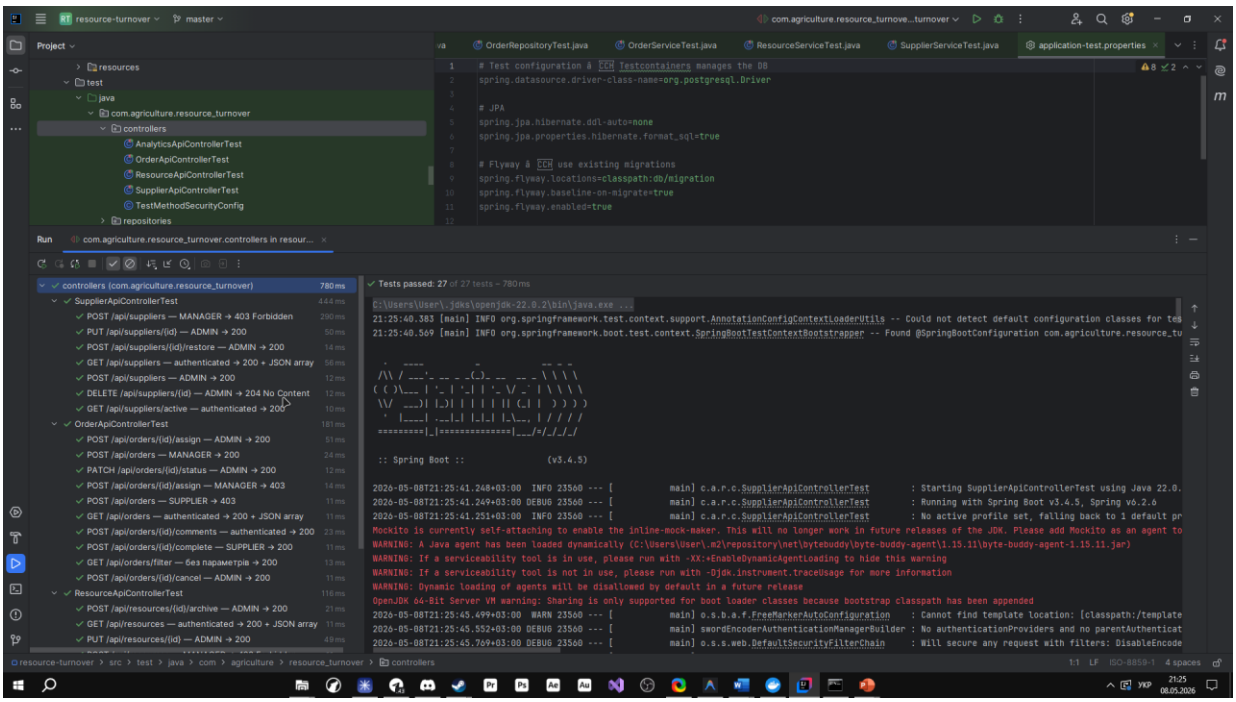


Рис. 5.3 Інтеграційне тестування контролерів. Частина 1

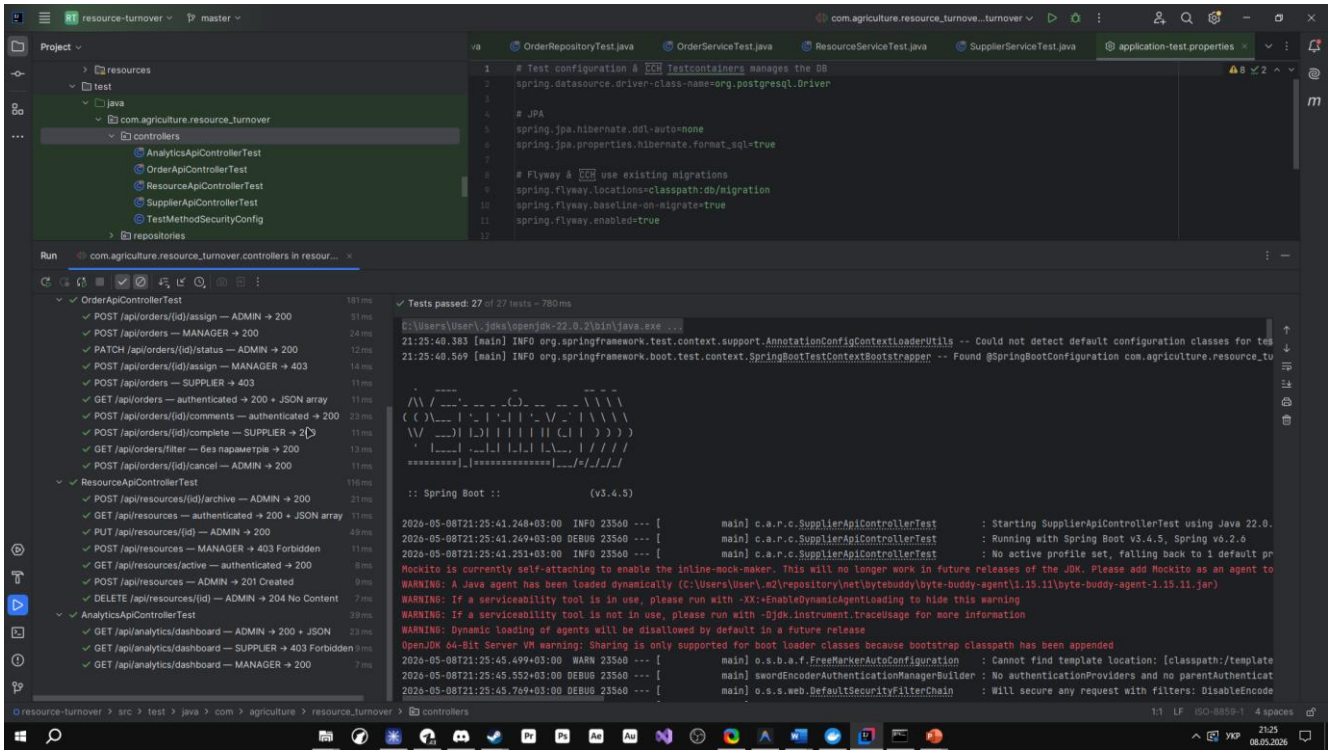


Рис. 5.4 Інтеграційне тестування контролерів. Частина 2

5.4 Тестування безпеки та рольового доступу

Перевірка систем захисту увійшла до числа ключових завдань на етапі перевірки платформи. Це пов’язано з обробкою даних, які мають різний рівень конфіденційності. Потреба у точному поділі доступу виникає серед груп користувачів. У політиці безпеки програми виділяються три основні ролі. Ними є адміністратор, менеджер - а також постачальник.

Під час тестування контролери з каскадним керуванням правами враховують ролі завдяки маркуванню @WithMockUser - воно симулює користувацькі профілі. Замість реального входження, всередині безпечного оточення генерується умовний токен із чітко визначеними правами доступу, щоб простежити поведінку системи при запитах від різних користувачів. Не потрібно запускати окремий екземпляр Keycloak або підтягувати облікові дані ззовні. Якщо знадобиться оцінка логіки для ролей, переважає підхід попарного аналізу. У кожному тесті бізнес-функції - дві умови: одна передбачає валідний токен, інша моделює запит без прав доступу. Такий підхід перевіряє чіткість роботи захисних механізмів: авторизовані отримують шлях до даних, всі інші блокуються на вході.

Перевірка виявила неполадку - центральний обробник некоректно трактував заборону доступу. Замість очікуваного коду 403, коли Spring Security активує @PreAuthorize, сервер викидав код 500. Це скидалося на серйозну збійну ситуацію всередині системи. Тепер такий випадок перехоплює спеціальний механізм. Його завдання - формувати правильний JSON разом із статусом 403. У ході аналізу виявилося, що без цих перевірок користувач ризикував марно провести багато часу, досліджуючи неіснуючі помилки. Правила надання доступу, підтверджені експериментально, наведені далі - у таблиці під номером 5.1.

Таблиця 5.1

Матриця рольового доступу, верифікована тестами

Операція	**ADMIN**	**MANAGER**	**SUPPLIER**
GET /api/orders	+ (200)	+ (200)	+ (200)
POST /api/orders	+ (200)	+ (200)	- (403)
POST /api/orders/{id}/assign	+ (200)	- (403)	-
POST /api/orders/{id}/complete	+ (200)	-	+ (200)
POST /api/orders/{id}/cancel	+ (200)	-	+ (200)
PATCH /api/orders/{id}/status	+ (200)	-	-
GET /api/resources	+ (200)	+ (200)	+ (200)
POST /api/resources	+ (201)	- (403)	- (403)
PUT /api/resources/{id}	+ (200)	- (403)	- (403)
DELETE /api/resources/{id}	+ (204)	- (403)	- (403)
GET /api/suppliers	+ (200)	+ (200)	+ (200)
POST /api/suppliers	+ (200)	- (403)	- (403)
DELETE /api/suppliers/{id}	+ (204)	- (403)	- (403)
GET /api/analytics/dashboard	+ (200)	+ (200)	- (403)

Перевірку людей у системі часто забезпечують через JWT-маркери, які видає Keycloak. Джерелом ключів для валідації виступає спеціальне посилання онлайн. Проблеми з авторизацією є поодинокими, проте мають чіткі форми прояву. Відсутність заголовка Authorization - одразу статус 401. Якщо минув строк дії маркера - результат аналогічний, все ще 401. Коли токен є, проте не вистачає доступу для ролі - відповідь сервера стає 403. SQL-атаки майже не проходять, адже Spring Data JPA працює з параметризованими запитами. Щодо CORS - список доменів зафіксовано всередині SecurityConfig, як це було описано раніше. Реакція системи на небезпечні запити перевіряється не напряму, а крізь результати автентифікаційних тестів.

5.5 Зведені результати тестуванні

Зведені результати виконання тестового набору систематизовані за ознакою модуля та типу тестів. У процесі розробки та виконання тестового набору було виявлено та усунено три категорії дефектів, що підтверджує практичну цінність обраної стратегії тестування. Перший дефект стосувався некоректного перехоплення виняткової ситуації відмови доступу загальним обробником, що призводило до повернення HTTP 500 замість HTTP 403. Другий дефект - технічний: мутаційні запити у MockMvc завершувалися відмовою через відсутність токена захисту від міжсайтових підроблених запитів. Третій дефект - відсутність активації підтримки декларативних перевірок ролей у тестовому контексті @WebMvcTest. Усі дефекти виявлені саме у процесі виконання тестів та усунені до завершення розробки відповідного модуля.

На основі накопичених результатів тестування, можна об'єктивно судити про готовність продукту та його повної синхронізації з бізнес-вимогами проєкту. Абсолютно кожен пункт функціональних критеріїв приймання вже відпрацьовано кодом. Самостійний прорахунок алгоритмом стартового статусу купівельного запиту, спираючись на актуальні ліміти доступної сировини, надійно зафіксовано в межах перевірок для OrderService. Життєвий цикл статусів об'єкта із жорстким блокуванням кінцевих точок життєвого циклу, успішно протестовано через цільові автотести.

Відновлення балансів сировини на складах у момент відкликання або розірвання замовлення валідується через взаємопов'язані парні сценарії на основі варіативності його початкових статусів. Повний життєвий цикл роботи із системними довідковими даними з урахуванням м'якого стирання даних через прапорці неактивності, надійно опрацьовано автоматичними сценаріями від класів обробки даних ResourceService та SupplierService. Автоматичний підрахунок та групування бізнес-метрик, забезпечуючи безпечний обхід помилок нульового вказівника при нульовій вибірці, надійно зафіксована у перевірочних сценаріях для сервісного шару AnalyticsService.

Через матрицю з таблиці 5.1 пройшли перевірку нефункціональні вимоги, паралельно досліджуючи поведінку окремих контролерів. У важких умовах система тримається саме тому, що контроль помилок ведеться на рівні сервісу, а центральний механізм опрацювання винятків було суттєво модернізовано. Протестовані сценарії тепер гратимуть роль бар’єру для зникнення можливостей - будь-яка шкідлива правка коду одразу позначиться на результаті запуску набору.

ВИСНОВКИ

У межах виконання дипломної роботи здійснено повний цикл аналізу предметної області, проектування та програмної реалізації системи електронного замовлення і постачання аграрних ресурсів для агропідприємства. Поставлені завдання виконано в повному обсязі, а отримані результати підтверджують досягнення мети дослідження.

У ході виконання роботи досягнуто таких результатів:

1. Проведено аналіз стану цифровізації аграрного сектору та існуючих програмних рішень. Хоча сфера має велике значення для державної економіки, справжнє оцифрування ще не вирішило численні проблеми. Замість загальноприйнятих систем фермери все частіше обирають інструменти типу SoftFarm чи Odoo, проте цілісна модель для логістики й документообігу відсутня. Малим і середнім угіддям доводиться працювати в умовах постійного тиску на витрати, попри те, що потреба в ефективних механізмах стрибкоподібно зростає. Без дорогих аналогів - таких, як SAP S/4HANA - організації тепер навчаються знаходити свої шляхи.

2. Сформульовано потреби системи - завдяки цьому виникли головні проектні матеріали. Замість переліку функцій - детальний опис робочих процесів для кожної ролі: у трьох видів користувачів свої права доступу. Етапи замовлення змінюються послідовно, не пропускаючи жодного кроку; заплановано п’ять станів і конкретні умови переходу. Дані структуровано за допомогою ER-діаграми, тим часом архітектура сервера ілюстрована через UML-схему класів. Імітацію дій з людиною записано як окремі сценарії, долучивши до них графічне представлення логічних потоків.

3. Вибрано програмні інструменти для створення клієнт-серверного веб-застосунку - спираючись на потреби проекту. На сервері працює Java разом із Spring Boot, що забезпечує стабільне функціонування всіх API. Інтерфейс з

побудований на базі Next.js - тому реакція на дії користувача відбувається практично одразу. Для зберігання даних обрано PostgreSQL, адже він добре себе показує при роботі зі складними запитами. Автентифікацію та авторизацію виділено на окремий сервер - там уже підтримує Keycloak. Docker Compose допомагає упакувати все середовище так, щоб воно однаково працювало скрізь: локально, на тестах або в продакшені.

4. Замість переліку елементів - показано будову системи разом із основними моделями. Деталі про дані наведено на ER-діаграмі, тим часом серверна частина подана через схеми класів у стилі UML, розділених за рівнями архітектури. Особливо виділено діаграму використань: на ній точно позначено, як кожна з трьох ролей працює з можливостями системи. Процеси взаємодії людей і платформи записано кроками, до того ж логіка цих процесів має також візуальне оформлення.

5. На основі спроєктованої архітектури реалізовано серверну та клієнтську частини системи. У просторі сервера діють чотири контролери, причому кожен спеціалізується на конкретній задачі. Логіку опрацювання даних зосереджено у сервісному шарі замість того, щоб розподіляти її хаотично. Замовлення фільтруються нестандартно - тут допомагає Criteria API. Користувач бачить лише те, що потрібне саме йому, решту система приховує. Аналітичні дані з’являються у панелі, яку побудовано спеціально для адміністратора. Сформовано та впроваджено інфраструктуру захисту даних і виведено в автоматичний режим етап розгортання додатка на серверах. Завдяки Keycloak та OAuth 2.0, перевірка особистостей стала можливою; JWT-підпис виконується на основі асиметричного ключа. У системі не передбачено зберігання стану - кожна взаємодія розпочинається наново. Користувачі працюють у межах окремих сесій, які не перетинаються одне з одним. Доступ забезпечується через конкретні дозволи, призначені лише для певних завдань. Кожна операція проходить у строгому локальному контексті. Керування контейнерами - завдання Docker Compose, усередині якого працюють чотири окремі служби, кожна самодостатня.

6. Була проведена повна перевірка системи за планом піраміди тестування. Зроблено п'ятдесят вісім тестів, які охоплюють модульне тестування

бізнес-логіки сервісного шару та інтеграційне тестування REST-контролерів з перевіркою ролейних обмежень. Всі тести пройшли успішно, у часі перевірки виявили і виправили три помилки, що підтверджує цінність обраної стратегії верифікації.

Таким чином, поставлені в межах кваліфікаційної роботи завдання виконано в повному обсязі - від аналізу предметної області та обґрунтування технологічного стеку до програмної реалізації та верифікації якості через комплексне тестування.

Результатом роботи став спеціалізований веб-застосунок електронного замовлення та постачання аграрних ресурсів, що закриває виявлений під час аналізу функціональний розрив на ринку та може бути використаний для цифровізації закупівельних процесів на агропідприємствах малого та середнього бізнесу.

Робота пройшла апробацію. За її результатами опубліковано такі тези доповідей:

1. Зелінський М.С., Худік Б.О. Розробка інформаційної системи управління замовленням для інтернет-магазину. Матеріали II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025, С. 462-463.

2. Зелінський М.С., Худік Б.О. Проектування та реалізація клієнт-серверної системи електронного замовлення та постачання аграрних ресурсів. Матеріал Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в ІКТ», 23 квітня 2026 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2026, С. 458-460.

3. Зелінський М.С. Захист даних в електронній системі агрозамовлень. Матеріали Всеукраїнської науково-практичної конференції «Цифрова трансформація кібербезпеки», 29 квітня 2026 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2026, С. 39-40.

ПЕРЕЛІК ПОСИЛАНЬ

1. McKinsey Global Institute. Digital America: A Tale of the Haves and Have-Mores. 2015. URL: https://www.mckinsey.com/~/media/mckinsey/industries/technology%20media%20and%20telecommunications/high%20tech/our%20insights/digital%20america%20a%20tale%20of%20the%20haves%20and%20have%20mores/digital%20america%20full%20report%20december%202015.pdf (дата звернення: 23.02.2026).

2. Tzachor A., Richards C. E., Jeen S. Transforming agrifood production systems and supply chains with digital twins. npj | Science of Food. 2022. Vol. 6, no. 1. P. 1–5. URL: https://doi.org/10.1038/s41538-022-00162-2

3. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : дис. … д-ра філос. наук. University of California, Irvine, 2000. URL: https://roy.gbiv.com/pubs/dissertation/fielding_dissertation.pdf

4. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol : O'Reilly Media, 2021. 616 p.

5. Evans B. J., Clark J., Verburg M. The Well-Grounded Java Developer. 2nd ed. Shelter Island : Manning Publications, 2022. 560 p.

6. Walls C. Spring in Action. 6th ed. Shelter Island : Manning Publications, 2022. 520 p.

7. Larsson M. Microservices with Spring Boot 3 and Spring Cloud. 3rd ed. Birmingham : Packt Publishing, 2023. 706 p.

8. Garcia M. M., Telang T. Learn Microservices with Spring Boot 3: A Practical Approach Using Event-Driven Architecture, Cloud-Native Patterns, and Containerization. New York : Apress, 2023. 436 p.

9. Next.js Documentation. Vercel, Inc. 2024. URL: https://nextjs.org/docs (дата звернення: 23.02.2026).

10. Obe R., Hsu L. PostgreSQL: Up and Running: A Practical Introduction to the Advanced Open Source Database. 3rd ed. Sebastopol : O'Reilly Media, 2017. 374 p.

11. PostgreSQL 15 Documentation. The PostgreSQL Global Development Group. 2023. URL: https://www.postgresql.org/docs/15/ (дата звернення: 02.03.2026).

12. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol : O'Reilly Media, 2017. 611 p.
13. Flyway Database Migrations Documentation. Redgate Software. 2024. URL: https://documentation.red-gate.com/flyway (дата звернення: 02.03.2026).
14. Richer J., Sanso L. OAuth 2 in Action. Shelter Island : Manning Publications, 2017. 360 p.
15. Spilca L. Spring Security in Action. Shelter Island : Manning Publications, 2020. 480 p.
16. Keycloak 22 Server Administration Guide. Red Hat, Inc. 2023. URL: https://www.keycloak.org/docs/22.0/server_admin/ (дата звернення: 28.02.2026).
17. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT) : RFC 7519. Internet Engineering Task Force. 2015. URL: https://datatracker.ietf.org/doc/html/rfc7519 (дата звернення: 25.02.2026).
18. Sakimura N., Bradley J., Jones M., de Medeiros B., Mortimore C. OpenID Connect Core 1.0 incorporating errata set 2. OpenID Foundation. 2023. URL: https://openid.net/specs/openid-connect-core-1_0.html (дата звернення: 28.02.2026).
19. Docker Compose Documentation. Docker, Inc. 2024. URL: https://docs.docker.com/compose/ (дата звернення: 01.03.2026).
20. Patton J. User Story Mapping: Discover the Whole Story, Build the Right Product. Sebastopol : O'Reilly Media, 2014. 322 p.
21. Fowler M. UML Distilled: A Brief Guide to the Standard Object Modeling Language. 3rd ed. Boston : Addison-Wesley Professional, 2003. 208 p.
22. Sandhu R. S., Coyne E. J., Feinstein H. L., Youman C. E. Role-Based Access Control Models. IEEE Computer. 1996. Vol. 29, no. 2. P. 38–47.
23. Tudose C. JUnit in Action. 3rd ed. Shelter Island : Manning Publications, 2020. 576 p.
24. JUnit 5 User Guide. JUnit Team. 2023. URL: https://junit.org/junit5/docs/current/user-guide/ (дата звернення: 25.02.2026).
25. Testcontainers for Java Documentation. AtomicJar, Inc. 2024. URL: https://java.testcontainers.org/ (дата звернення: 01.03.2026).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Система електронного замовлення та постачання ресурсів для агропідприємства

Виконав студент 4 курсу
групи ПД-43
Зелінський Максим Сергійович
Керівник роботи
доцент кафедри ІПЗ, доктор філософії (PhD) Худік Богдан Олександрович

Київ – 2026

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: автоматизація процесу управління замовленням та постачанням аграрних ресурсів за допомогою розробленого спеціалізованого клієнт-серверного програмного забезпечення з рольовою моделлю доступу та автоматизованим обліком матеріальних ресурсів.

Об’єкт дослідження: процес управління замовленням та постачанням аграрних ресурсів на агропідприємстві.

Предмет дослідження: методи та засоби побудови клієнт-серверного програмного забезпечення для автоматизації процесу управління замовленням та постачанням аграрних ресурсів.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Провести аналіз предметної області у сфері управління замовленням та постачанням аграрних ресурсів та існуючих програмних аналогів.
2. Сформулювати функціональні та нефункціональні вимоги до розроблюваної системи.
3. Здійснити обґрунтований вибір програмних засобів реалізації клієнт-серверного веб-застосунку.
4. Спроєктувати програмну архітектуру системи, рольову модель доступу та структуру бази даних, представивши результати у форматі діаграм.
5. Реалізувати серверну та клієнтську частини системи відповідно до спроєктованої архітектури.
6. Провести комплексне тестування розробленого рішення на модульному та інтеграційному рівнях.

ПОРІВНЯЛЬНИЙ АНАЛІЗ АНАЛОГІВ

Критерій	**BAS АГРО ERP**	**SAP S/4HANA**	**SoftFarm**	**Odoo**	**Розроблювана система**
Автоматизація переходів між статусами замовлення	-	+	-	-	+
Можливість коментування замовлень	-	+	-	+	+
Розмежування прав за ролями	+	+	+	+	+
Окремий доступ для зовнішніх постачальників	+	+	-	+	+
Відкритий API для інтеграцій	-	-	-	+	+
Спосіб розгортання	Локально	Хмара / Сервер	SaaS	Хмара / Локально	Docker Compose

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

Загальні функціональні вимоги.

1. Система повинна забезпечувати автентифікацію користувачів через зовнішній сервер ідентифікації.
2. Має бути реалізована можливість перегляду переліку ресурсів та постачальників.
3. Має бути реалізована можливість перегляду переліку замовлень з фільтрацією за статусом та датою.
4. Повинна бути реалізована можливість додавання коментарів до замовлення.
5. Менеджер повинен мати можливість створювати нове замовлення з вибором ресурсу, його кількості та терміну постачання.

Функціональні вимоги для ролі «Менеджер»:

1. Менеджер повинен мати можливість створювати нове замовлення з вибором ресурсу, його кількості та терміну постачання.
2. Менеджер повинен мати можливість переглядати аналітичний дашборд системи.

Функціональні вимоги для ролі «Адміністратор»:

1. Адміністратор повинен мати можливість керувати довідником ресурсів - створювати, редагувати та архівувати позиції номенклатури.
2. Адміністратор повинен мати можливість керувати довідником постачальників - створювати, редагувати, архівувати та відновлювати записи.
3. Адміністратор повинен мати можливість створювати нові замовлення.
4. Адміністратор повинен мати можливість призначати постачальника до конкретного замовлення.
5. Адміністратор повинен мати можливість змінювати статус замовлення відповідно до правил машини станів.
6. Адміністратор повинен мати можливість завершувати та скасовувати замовлення.
7. Адміністратор повинен мати можливість переглядати аналітичний дашборд системи.

Функціональні вимоги для ролі «Постачальник»:

1. Постачальник повинен мати можливість завершувати призначені йому замовлення з підтвердженням виконання.
2. Постачальник повинен мати можливість скасовувати призначені йому замовлення.

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Нефункціональні вимоги:

1. Серверна частина системи повинна бути побудована за шаблоном MVC з «шаруватою» організацією на рівні контролерів, сервісів та репозиторіїв.
2. Серверна частина повинна підтримувати «безстанну» (stateless) архітектуру для забезпечення горизонтального масштабування.
3. Клієнтська частина повинна працювати незалежно від серверної через REST API, з принциповою забороною прямого доступу до бази даних з клієнтської сторони.
4. Система повинна виконувати автентифікацію користувачів через протокол OAuth 2.0 з використанням сервера ідентифікації Keycloak.
5. Перевірка автентичності користувачів повинна виконуватися через JWT-токени з асиметричним цифровим підписом.
6. Контроль доступу до ендпоінтів API повинен реалізовуватися декларативно на рівні методів контролерів.
7. Вхідні дані API повинні проходити валідацію через механізм Bean Validation.
8. REST API системи повинно бути документоване за специфікацією OpenAPI 3.0.
9. Розгортання системи повинно виконуватися через Docker Compose з п'ятьма контейнерами – серверна частина, клієнт, система автентифікації, база даних та адміністративний інструмент бази.
10. Параметри підключення до бази даних, сервера автентифікації та інших зовнішніх сервісів повинні зберігатися у змінних середовища, що забезпечує конфігурованість системи без зміни вихідного коду

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

1. Backend (Java Stack):
- **Мова:** Java 17 (LTS).
- **Фреймворк:** Spring Boot 3.4.5 (Core, Web MVC).
- **Доступ до даних:** Spring Data JPA, Hibernate 6.6.
- **Валідація та мапінг:** Bean Validation, Lombok, MapStruct.

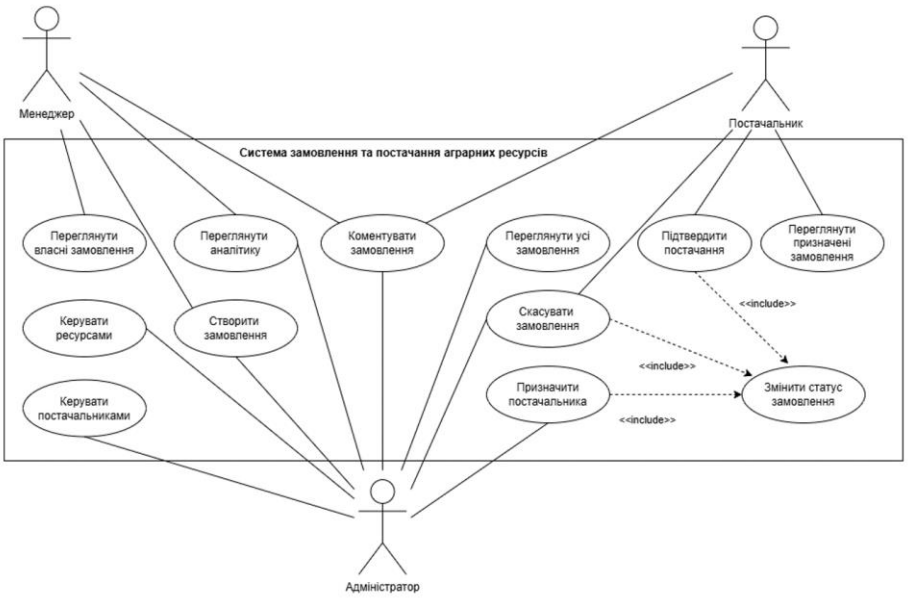
2. Frontend (Modern Web):
- **Мова:** TypeScript (^5).
- **Ядро UI:** React 19
- **Фреймворк:** Next.js 16 (SSR/SSG).
- **Стилізація:** Tailwind CSS 4.
- **Комунікація:** Axios, Next-Auth (Keycloak integration).

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

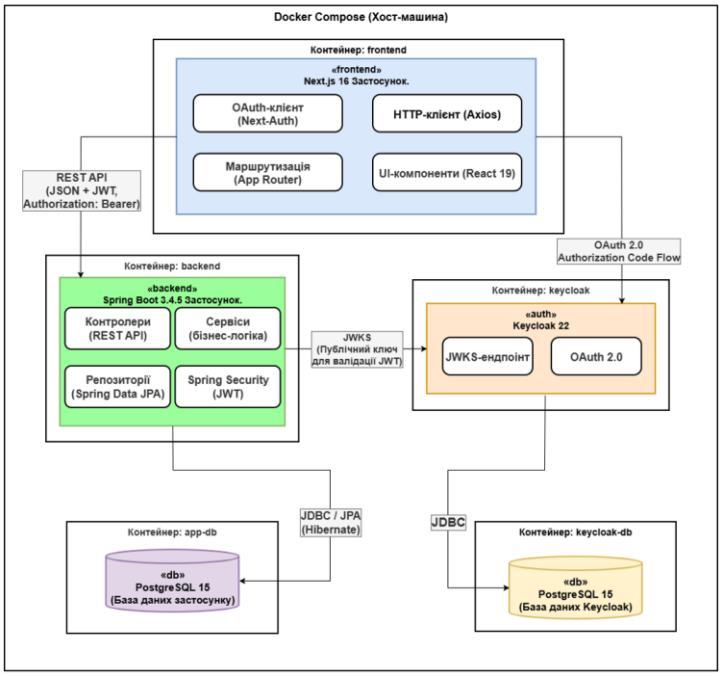
3. База даних та безпека:
- **СУБД:** PostgreSQL 15 (Alpine).
- **Контроль доступу:** Keycloak 22.0 (IAM-сервер).
- **Протоколи:** OAuth 2.0 / OpenID Connect, JWT.
- **Міграції:** Flyway.

4. Інфраструктура та тестування:
- **Контейнеризація:** Docker Desktop & Docker Compose.
- **Збірка проєкту:** Apache Maven.
- **Тестування:** JUnit 5, Testcontainers (PostgreSQL), Spring Security Test.
- **IDE:** IntelliJ IDEA, VS Code, pgAdmin 4.

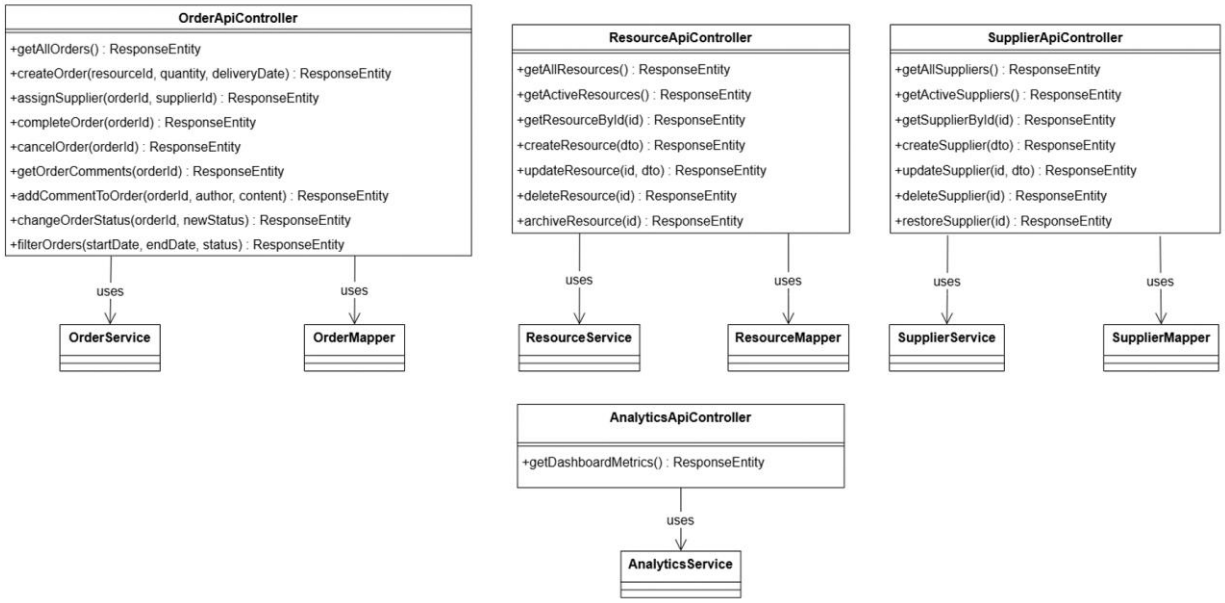
ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ



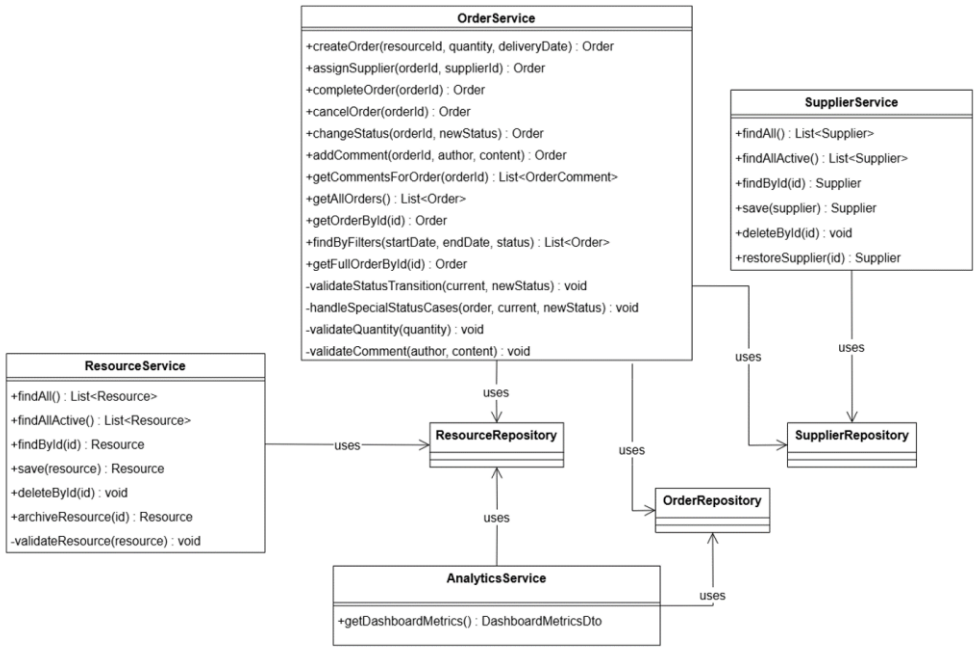
АРХІТЕКТУРА СИСТЕМИ



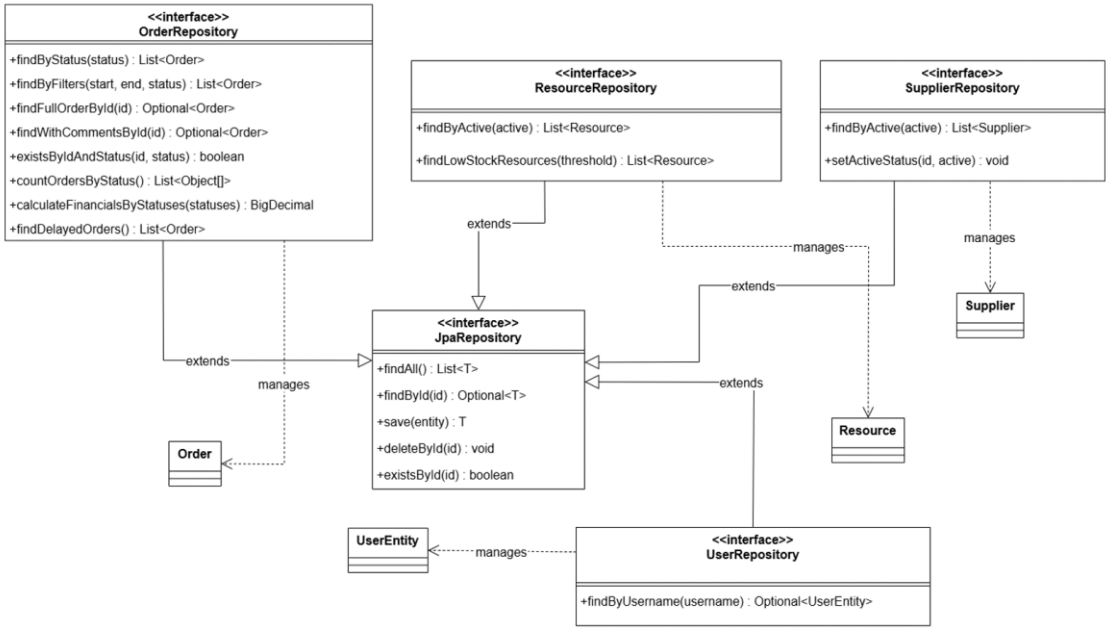
ДІАГРАМА КЛАСІВ – КОНТРОЛЕРИ



ДІАГРАМА КЛАСІВ – СЕРВІСИ

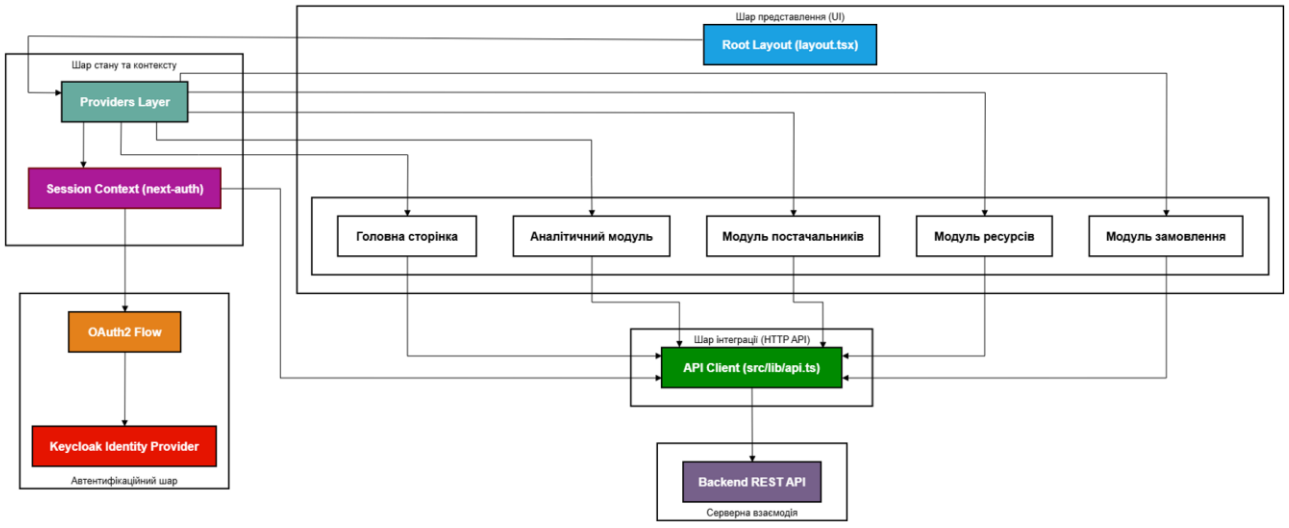


ДІАГРАМА КЛАСІВ – РЕПОЗИТОРІЇ



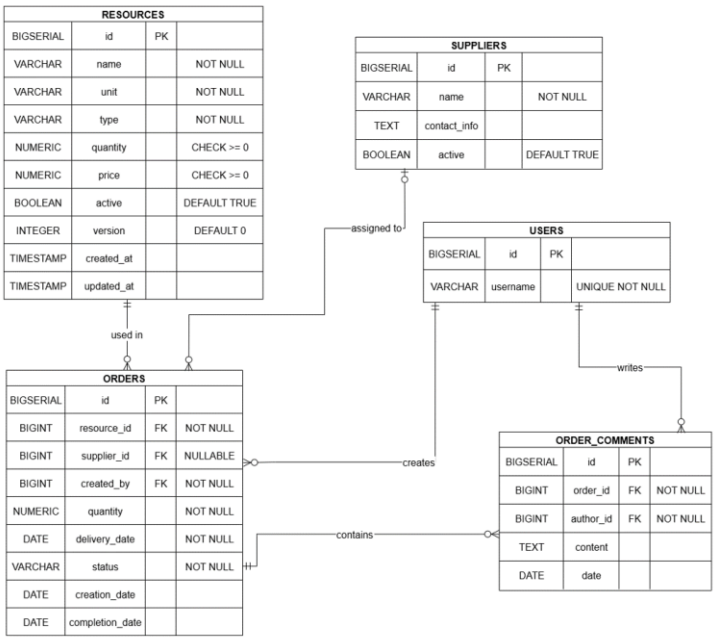
13

ІЄРАРХІЯ КОМПОНЕНТІВ КЛІЄНТСЬКОЇ ЧАСТИНИ



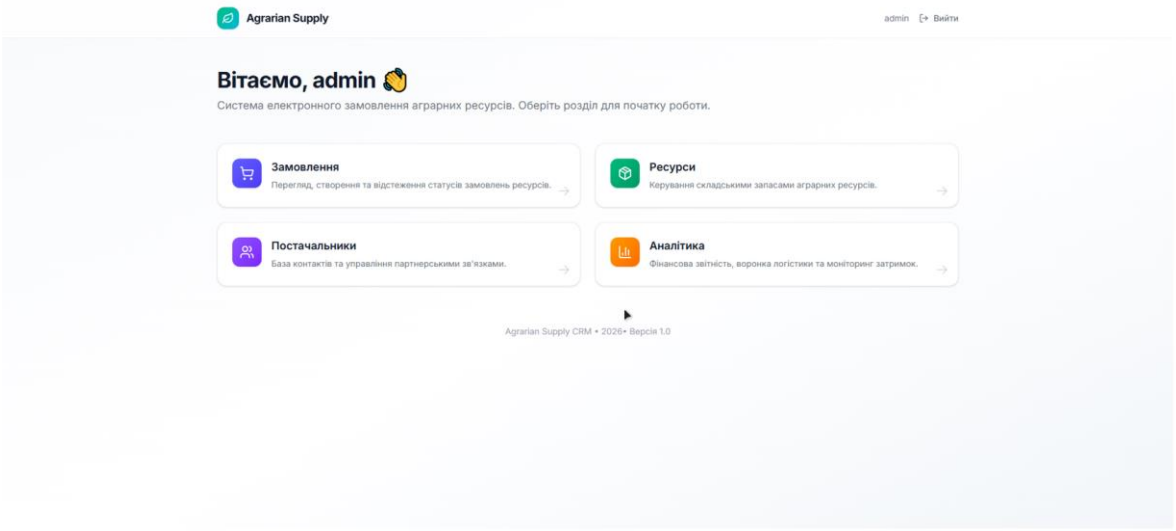
14

ER-ДІАГРМА БАЗИ ДАНИХ СИСТЕМИ



15

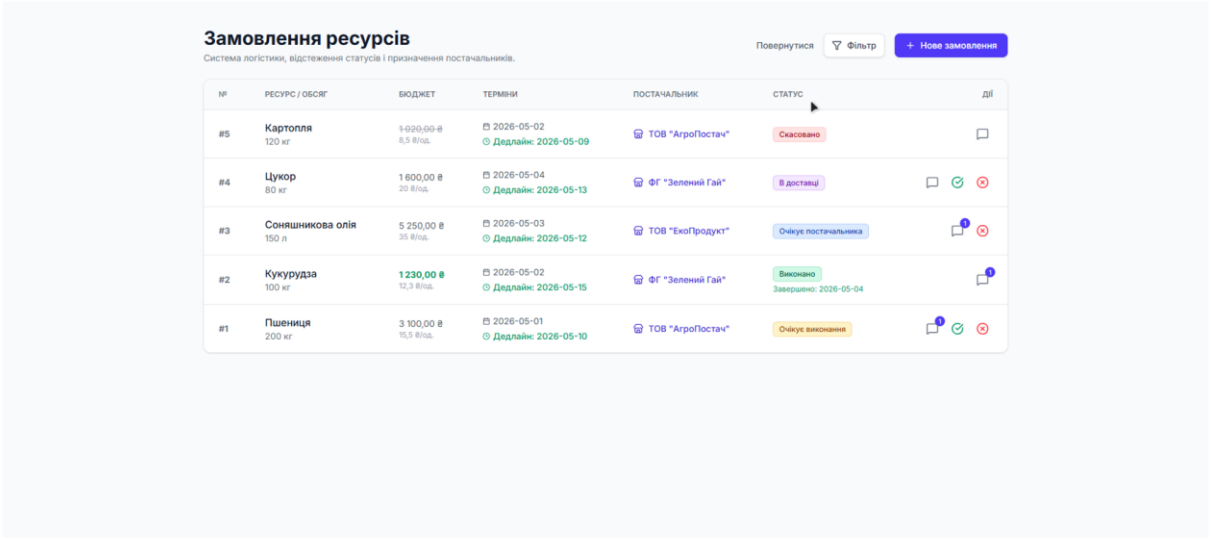
ЕКРАННІ ФОРМИ



Головна сторінка веб-застосунку (після авторизації)

16

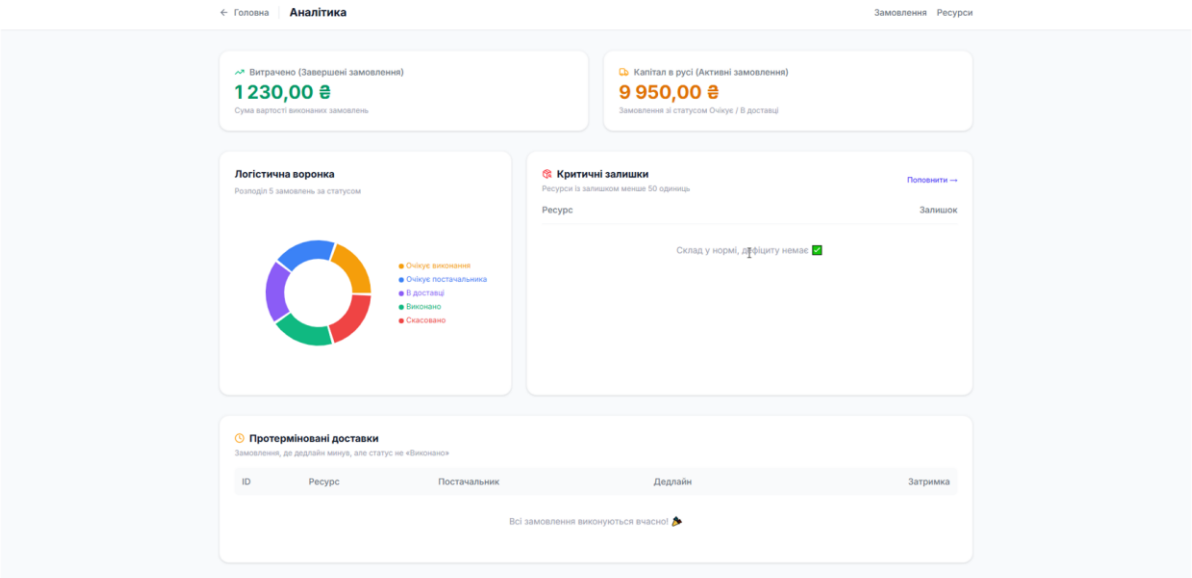
ЕКРАННІ ФОРМИ



Сторінка замовлення ресурсів

17

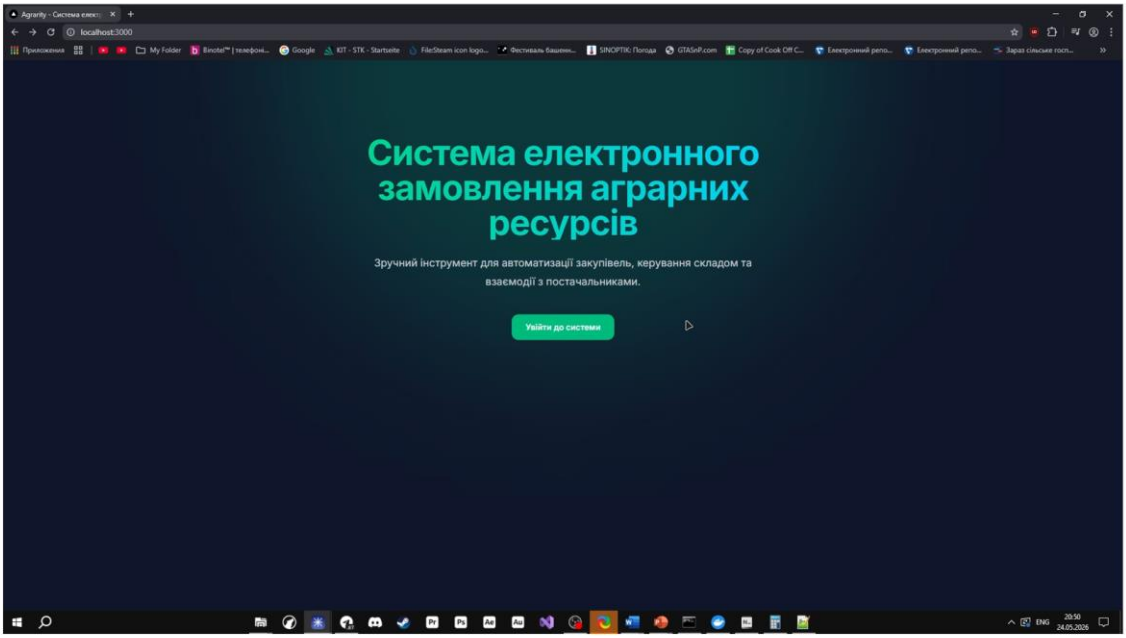
ЕКРАННІ ФОРМИ



Сторінка аналітики

18

ДЕМОНСТРАЦІЯ РОБОТИ ЗАСТОСУНКУ



19

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

Тези доповідей:

1. Зелінський М.С., Худік Б.О. Розробка інформаційної системи управління замовленням для інтернет-магазину. Матеріали II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025, С. 462-463.

2. Зелінський М.С., Худік Б.О. Проектування та реалізація клієнт-серверної системи електронного замовлення та постачання аграрних ресурсів. Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в ІКТ», 23 квітня 2026 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2026, С. 458-460.

3. Зелінський М.С. Захист даних в електронній системі агрозамовлень. Матеріали Всеукраїнської науково-практичної конференції «Цифрова трансформація кібербезпеки», 29 квітня 2026 року, Київ, ДУІКТ. Збірник тез. К.: ДУІКТ, 2026, С. 39-40.

20

ВИСНОВКИ

1. Проаналізовано предметну область та існуючі ринкові аналоги у сфері управління замовленням та постачанням аграрних ресурсів; виявлено функціональні обмеження ринкових платформ для агропромислового бізнесу малого та середнього сегменту.
2. На основі проаналізованих існуючих ринкових рішень сформовано функціональні та нефункціональні вимоги до системи, що визначають функціональні межі застосунку та принципи його взаємодії з користувачами.
3. Обґрунтовано вибір програмних засобів реалізації: Java та Spring Boot для серверної частини, Next.js для клієнтського інтерфейсу, СУБД PostgreSQL та Keycloak для безпеки.
4. Реалізовано серверну та клієнтську частини системи відповідно до спроєктованої архітектури, з документуванням результатів у форматі діаграм класів за архітектурними шарами та моделі бази даних, а також з налаштованим розгортанням через Docker Compose.
5. Проведено комплексне тестування розробленого рішення на модульному та інтеграційному рівнях (58 тестових випадків), що підтвердило коректність бізнес-логіки та автоматизованих сценаріїв.
6. Результатом роботи є спеціалізований веб-застосунок, що автоматизує процес замовлення і постачання аграрних ресурсів через машину станів замовлення, рольове розмежування доступу та централізоване управління матеріальними ресурсами.

ДОДАТОК Б. ЛІСТИНГ ПРОГРАМНИХ МОДУЛІВ

OrderService.java

```
package com.agriculture.resource_turnover.services;

import
com.agriculture.resource_turnover.config.OrderSpecifications;
import com.agriculture.resource_turnover.models.*;
import
com.agriculture.resource_turnover.repositories.OrderRepositor
y;
import
com.agriculture.resource_turnover.repositories.ResourceReposi
tory;
import
com.agriculture.resource_turnover.repositories.SupplierReposit
ory;
import jakarta.persistence.EntityNotFoundException;
import jakarta.transaction.Transactional;
import lombok.RequiredArgsConstructor;
import org.springframework.data.jpa.domain.Specification;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Service;
import
org.springframework.web.server.ResponseStatusException;

import java.math.BigDecimal;
import java.time.LocalDate;
import java.util.List;
import java.util.Map;

@Service
@RequiredArgsConstructor
public class OrderService {
   private final OrderRepository orderRepository;
   private final ResourceRepository resourceRepository;
   private final SupplierRepository supplierRepository;

   @Transactional
   public Order createOrder(Long resourceId, BigDecimal
quantity, LocalDate deliveryDate) {
      validateQuantity(quantity);
      Resource resource = getResourceById(resourceId);

      Order order = new Order();
      order.setResource(resource);
      order.setQuantity(quantity);
      order.setDeliveryDate(deliveryDate);

      if (resource.getQuantity().compareTo(quantity) >= 0) {
         order.setStatus(OrderStatus.PENDING_EXECUTION);
         resource.setQuantity(resource.getQuantity().subtract(qu
antity));
         resourceRepository.save(resource);
      } else {
         order.setStatus(OrderStatus.PENDING_SUPPLY);
      }

      return orderRepository.save(order);
   }

   @Transactional
   public Order assignSupplier(Long orderId, Long supplierId)
{
      Order order = getOrderById(orderId);

      if (order.getStatus() != OrderStatus.PENDING_SUPPLY
&& order.getStatus() !=
OrderStatus.PENDING_EXECUTION) {
         throw new
ResponseStatusException(HttpStatus.BAD_REQUEST,
               "Only PENDING_SUPPLY or
PENDING_EXECUTION orders can be assigned");
      }

      Supplier supplier = getSupplierById(supplierId);
      order.setSupplier(supplier);
      order.setStatus(OrderStatus.IN_DELIVERY);
      return orderRepository.save(order);
   }

   @Transactional
   public Order completeOrder(Long orderId) {
      return changeStatus(orderId, OrderStatus.COMPLETED);
   }

   @Transactional
   public Order cancelOrder(Long orderId) {
      return changeStatus(orderId, OrderStatus.CANCELLED);
   }

   @Transactional
   public Order changeStatus(Long orderId, OrderStatus
newStatus) {
      Order order = getOrderById(orderId);
      OrderStatus currentStatus = order.getStatus();

      validateStatusTransition(currentStatus, newStatus);

      handleSpecialStatusCases(order, currentStatus,
newStatus);

      order.setStatus(newStatus);
      return orderRepository.save(order);
   }

   private void validateStatusTransition(OrderStatus current,
OrderStatus newStatus) {
      Map<OrderStatus, List<OrderStatus>>
allowedTransitions = Map.of(
            OrderStatus.PENDING_EXECUTION,
List.of(OrderStatus.IN_DELIVERY,
OrderStatus.CANCELLED, OrderStatus.COMPLETED,
OrderStatus.PENDING_EXECUTION,
OrderStatus.PENDING_SUPPLY),
            OrderStatus.PENDING_SUPPLY,
List.of(OrderStatus.IN_DELIVERY,
OrderStatus.CANCELLED, OrderStatus.COMPLETED,
OrderStatus.PENDING_EXECUTION,
OrderStatus.PENDING_SUPPLY),
            OrderStatus.IN_DELIVERY,
List.of(OrderStatus.IN_DELIVERY,
OrderStatus.CANCELLED, OrderStatus.COMPLETED,
OrderStatus.PENDING_EXECUTION,
OrderStatus.PENDING_SUPPLY),
            OrderStatus.COMPLETED, List.of(),
            OrderStatus.CANCELLED, List.of()
      );

      if (!allowedTransitions.get(current).contains(newStatus)) {
         throw new
ResponseStatusException(HttpStatus.BAD_REQUEST,
               String.format("Недопустимый переход статуса с
%s на %s", current, newStatus));
      }
   }
```

```
    private void handleSpecialStatusCases(Order order,
OrderStatus current, OrderStatus newStatus) {
        if ((current == OrderStatus.PENDING_EXECUTION ||
current == OrderStatus.IN_DELIVERY) && newStatus ==
OrderStatus.CANCELLED) {
            Resource resource = order.getResource();
            resource.setQuantity(resource.getQuantity().add(order.g
etQuantity()));
            resourceRepository.save(resource);
        }

        if (newStatus == OrderStatus.COMPLETED) {
            order.setCompletionDate(LocalDate.now());
        }
    }

    @Transactional
    public Order addComment(Long orderId, String author,
String content) {
        validateComment(author, content);
        Order order = getOrderById(orderId);

        OrderComment comment = new OrderComment();
        comment.setAuthor(author);
        comment.setContent(content);
        comment.setOrder(order);
        order.getComments().add(comment);

        return orderRepository.save(order);
    }

    public List<OrderComment> getCommentsForOrder(Long
orderId) {
        Order order = getOrderById(orderId);
        return order.getComments();
    }

    public List<Order> getAllOrders() {
        return orderRepository.findAll();
    }

    public Order getOrderById(Long id) {
        return orderRepository.findById(id)
                .orElseThrow(() -> new
IllegalArgumentException("Order not found with id: " + id));
    }

    public List<Order> findByFilters(LocalDate startDate,
LocalDate endDate, OrderStatus status) {

        Specification<Order> spec =
Specification.where(OrderSpecifications.creationDateAfterOrE
qual(startDate))
                .and(OrderSpecifications.creationDateBeforeOrEqual
(endDate))
                .and(OrderSpecifications.hasStatus(status));

        return orderRepository.findAll(spec);
    }

    public Order getFullOrderById(Long id) {
        return orderRepository.findFullOrderById(id)
                .orElseThrow(() -> new
EntityNotFoundException("Order not found with id: " + id));
    }

    private void validateQuantity(BigDecimal quantity) {
        if (quantity == null ||
quantity.compareTo(BigDecimal.ZERO) <= 0) {
            throw new IllegalArgumentException("Quantity must
be positive");
        }
    }

    private Resource getResourceById(Long resourceId) {
        return resourceRepository.findById(resourceId)
                .orElseThrow(() -> new
IllegalArgumentException("Resource not found with id: " +
resourceId));
    }

    private Supplier getSupplierById(Long supplierId) {
        return supplierRepository.findById(supplierId)
                .orElseThrow(() -> new
IllegalArgumentException("Supplier not found with id: " +
supplierId));
    }

    private void validateComment(String author, String content)
{
        if (author == null || author.isBlank()) {
            throw new IllegalArgumentException("Author cannot
be empty");
        }
        if (content == null || content.isBlank()) {
            throw new IllegalArgumentException("Content cannot
be empty");
        }
    }
}
```

ResourceSerivce.java

```
package com.agriculture.resource_turnover.services;

import com.agriculture.resource_turnover.models.Resource;
import
com.agriculture.resource_turnover.repositories.ResourceReposi
tory;
import jakarta.transaction.Transactional;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Service;

import java.math.BigDecimal;
import java.util.List;
import java.util.Optional;

@Slf4j
@Service
@RequiredArgsConstructor
public class ResourceService {

    private final ResourceRepository resourceRepository;

    public List<Resource> findAll() {
        log.info("Fetching all resources");
        return resourceRepository.findAll();
    }

    public List<Resource> findAllActive() {
        log.info("Fetching all active resources");
        return resourceRepository.findByActive(true);
    }

    public Resource findById(Long id) {
        log.info("Fetching resource by id: {}", id);
        return resourceRepository.findById(id)
```

```
            .orElseThrow(() -> {
                String errorMsg = "Resource not found with id: " +
id;
                log.error(errorMsg);
                return new IllegalArgumentException(errorMsg);
            });
    }

    @Transactional
    public Resource save(Resource resource) {
        log.info("Attempting to save resource: {}", resource);

        validateResource(resource);

        Resource savedResource =
resourceRepository.save(resource);
        log.info("Successfully saved resource with id: {}",
savedResource.getId());

        return savedResource;
    }

    @Transactional
    public void deleteById(Long id) {
        log.info("Attempting to delete resource with id: {}", id);

        if (!resourceRepository.existsById(id)) {
            String errorMsg = "Resource not found with id: " + id;
            log.error(errorMsg);
            throw new IllegalArgumentException(errorMsg);
        }

        resourceRepository.deleteById(id);
        log.info("Successfully deleted resource with id: {}", id);
    }

    @Transactional
    public Resource archiveResource(Long id) {
        log.info("Attempting to archive resource with id: {}", id);

        Resource resource = findById(id);
        resource.setActive(!resource.isActive());

        Resource updatedResource =
resourceRepository.save(resource);
        log.info("Resource with id: {} archive status set to: {}",
                id, updatedResource.isActive());

        return updatedResource;
    }

    private void validateResource(Resource resource) {
        if (resource.getName() == null ||
resource.getName().trim().isEmpty()) {
            throw new IllegalArgumentException("Resource name
cannot be empty");
        }
        if (resource.getUnit() == null ||
resource.getUnit().trim().isEmpty()) {
            throw new IllegalArgumentException("Unit cannot be
empty");
        }
        if (resource.getType() == null ||
resource.getType().trim().isEmpty()) {
            throw new IllegalArgumentException("Type cannot be
empty");
        }
        if (resource.getQuantity() == null ||
resource.getQuantity().compareTo(BigDecimal.ZERO) < 0) {
            throw new IllegalArgumentException("Quantity must
be positive or zero");
        }
        if (resource.getPrice() == null ||
resource.getPrice().compareTo(BigDecimal.ZERO) < 0) {
            throw new IllegalArgumentException("Price must be
positive or zero");
        }
    }
}
```

SupplierService.java

```
package com.agriculture.resource_turnover.services;

import com.agriculture.resource_turnover.models.Supplier;
import
com.agriculture.resource_turnover.repositories.SupplierReposit
ory;
import jakarta.transaction.Transactional;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
@RequiredArgsConstructor
public class SupplierService {
    private final SupplierRepository supplierRepository;

    public List<Supplier> findAll() {
        return supplierRepository.findAll();
    }

    public List<Supplier> findAllActive() {
        return supplierRepository.findByActive(true);
    }

    public Supplier findById(Long id) {
        return supplierRepository.findById(id)
                .orElseThrow(() -> new
IllegalArgumentException("Supplier not found with id: " +
id));
    }

    }

    @Transactional
    public Supplier save(Supplier supplier) {
        if (supplier.getName() == null ||
supplier.getName().trim().isEmpty()) {
            throw new IllegalArgumentException("Supplier name
cannot be empty");
        }
        if (supplier.getContactInfo() == null ||
supplier.getContactInfo().trim().isEmpty()) {
            throw new IllegalArgumentException("Contact info
cannot be empty");
        }
        return supplierRepository.save(supplier);
    }

    @Transactional
    public void deleteById(Long id) {
        if (supplierRepository.existsById(id)) {
            supplierRepository.setActiveStatus(id, false);
        }
    }

    @Transactional
    public Supplier restoreSupplier(Long id) {
        supplierRepository.setActiveStatus(id, true);
        return supplierRepository.findById(id).orElseThrow();
    }
}
```

AnalyticService.java

```
package com.agriculture.resource_turnover.services;

import
com.agriculture.resource_turnover.dto.DashboardMetricsDto;
import com.agriculture.resource_turnover.models.Order;
import com.agriculture.resource_turnover.models.OrderStatus;
import com.agriculture.resource_turnover.models.Resource;
import
com.agriculture.resource_turnover.repositories.OrderRepositor
y;
import
com.agriculture.resource_turnover.repositories.ResourceReposi
tory;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import
org.springframework.transaction.annotation.Transactional;

import java.math.BigDecimal;
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

@Service
@RequiredArgsConstructor
@Transactional(readOnly = true)
public class AnalyticsService {
   private final OrderRepository orderRepository;
   private final ResourceRepository resourceRepository;

   public DashboardMetricsDto getDashboardMetrics() {
      DashboardMetricsDto metrics = new
DashboardMetricsDto();

      List<Object[]> statusCounts =
orderRepository.countOrdersByStatus();
      Map<String, Long> statusMap = new HashMap<>();
      for (OrderStatus status : OrderStatus.values()) {
         statusMap.put(status.name(), 0L);
      }
      for (Object[] row : statusCounts) {
         OrderStatus status = (OrderStatus) row[0];
         Long count = (Long) row[1];
         statusMap.put(status.name(), count);
      }
      metrics.setOrdersByStatus(statusMap);

      BigDecimal spendings =
orderRepository.calculateFinancialsByStatuses(List.of(OrderSt
atus.COMPLETED));
      metrics.setFinancialSpendings(spendings != null ?
spendings : BigDecimal.ZERO);

      BigDecimal holdings =
orderRepository.calculateFinancialsByStatuses(
            List.of(OrderStatus.PENDING_EXECUTION,
OrderStatus.PENDING_SUPPLY,
OrderStatus.IN_DELIVERY));
      metrics.setActiveHoldings(holdings != null ? holdings :
BigDecimal.ZERO);

      List<Resource> lowStock =
resourceRepository.findLowStockResources(new
BigDecimal("50.0"));
      metrics.setLowStockResources(lowStock.stream().map(r -
>
         new
DashboardMetricsDto.LowStockResourceDto(r.getId(),
r.getName(), r.getUnit(), r.getQuantity())
      ).collect(Collectors.toList()));

      List<Order> delayedOpt =
orderRepository.findDelayedOrders();
      LocalDate today = LocalDate.now();
      List<DashboardMetricsDto.DelayedOrderDto>
delayedDtos = delayedOpt.stream().map(o -> {
         long daysLate =
ChronoUnit.DAYS.between(o.getDeliveryDate(), today);
         return new DashboardMetricsDto.DelayedOrderDto(
            o.getId(),
            o.getResource().getName(),
            o.getSupplier() != null ? o.getSupplier().getName() :
"Без постачальника",
            o.getDeliveryDate().toString(),
            o.getStatus().name(),
            daysLate > 0 ? daysLate : 1
         );
      }).collect(Collectors.toList());
      metrics.setDelayedOrders(delayedDtos);

      return metrics;
   }
}
```

AnalyticsApiController.java

```
package com.agriculture.resource_turnover.controllers.api;

import
com.agriculture.resource_turnover.dto.DashboardMetricsDto;
import
com.agriculture.resource_turnover.services.AnalyticsService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import
org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.GetMapping;
import
org.springframework.web.bind.annotation.RequestMapping;
import
org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api/analytics")
@RequiredArgsConstructor
public class AnalyticsApiController {

   private final AnalyticsService analyticsService;

   @PreAuthorize("hasRole('ADMIN') or
hasRole('MANAGER')")
   @GetMapping("/dashboard")
   public ResponseEntity<DashboardMetricsDto>
getDashboardMetrics() {
      return
ResponseEntity.ok(analyticsService.getDashboardMetrics());
   }
}
```

OrderApiController.java

```
package com.agriculture.resource_turnover.controllers.api;

import
com.agriculture.resource_turnover.dto.OrderCommentDto;
```

```
import
com.agriculture.resource_turnover.dto.OrderResponseDto;
import
com.agriculture.resource_turnover.dto.mapper.OrderMapper;
import com.agriculture.resource_turnover.models.Order;
import com.agriculture.resource_turnover.models.OrderStatus;
import
com.agriculture.resource_turnover.services.OrderService;
import lombok.RequiredArgsConstructor;
import
org.springframework.format.annotation.DateTimeFormat;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import
org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.*;
import
org.springframework.web.server.ResponseStatusException;

import java.math.BigDecimal;
import java.time.LocalDate;
import java.util.List;
import java.util.stream.Collectors;

@RestController
@RequestMapping("/api/orders")
@RequiredArgsConstructor
public class OrderApiController {
    private final OrderService orderService;
    private final OrderMapper orderMapper;

    @PreAuthorize("isAuthenticated()")
    @GetMapping
    public ResponseEntity<List<OrderResponseDto>>
getAllOrders() {
        return
ResponseEntity.ok(orderService.getAllOrders().stream()
                .map(orderMapper::toResponseDto)
                .collect(Collectors.toList()));
    }

    @PreAuthorize("hasRole('ADMIN') or
hasRole('MANAGER')")
    @PostMapping
    public ResponseEntity<OrderResponseDto> createOrder(
            @RequestParam Long resourceId,
            @RequestParam BigDecimal quantity,
            @RequestParam @DateTimeFormat(iso =
DateTimeFormat.ISO.DATE) LocalDate deliveryDate) {
        Order order = orderService.createOrder(resourceId,
quantity, deliveryDate);
        return
ResponseEntity.ok(orderMapper.toResponseDto(order));
    }

    @PreAuthorize("hasRole('ADMIN')")
    @PostMapping("/{orderId}/assign")
    public ResponseEntity<OrderResponseDto> assignSupplier(
            @PathVariable Long orderId,
            @RequestParam Long supplierId) {
        return
ResponseEntity.ok(orderMapper.toResponseDto(orderService.a
ssignSupplier(orderId, supplierId)));
    }

    @PreAuthorize("hasRole('ADMIN') or
hasRole('SUPPLIER')")
    @PostMapping("/{orderId}/complete")
    public ResponseEntity<OrderResponseDto>
completeOrder(@PathVariable Long orderId) {
        return
ResponseEntity.ok(orderMapper.toResponseDto(orderService.c
ompleteOrder(orderId)));
    }

    @PreAuthorize("hasRole('ADMIN') or
hasRole('SUPPLIER')")
    @PostMapping("/{orderId}/cancel")
    public ResponseEntity<OrderResponseDto>
cancelOrder(@PathVariable Long orderId) {
        return
ResponseEntity.ok(orderMapper.toResponseDto(orderService.c
ancelOrder(orderId)));
    }

    @PreAuthorize("isAuthenticated()")
    @GetMapping("/{orderId}/comments")
    public ResponseEntity<List<OrderCommentDto>>
getOrderComments(@PathVariable Long orderId) {
        return
ResponseEntity.ok(orderService.getCommentsForOrder(orderI
d).stream()
                .map(orderMapper::toCommentDto)
                .collect(Collectors.toList()));
    }

    @PreAuthorize("isAuthenticated()")
    @PostMapping("/{orderId}/comments")
    public ResponseEntity<OrderResponseDto>
addCommentToOrder(
            @PathVariable Long orderId,
            @RequestParam String author,
            @RequestParam String content) {
        return
ResponseEntity.ok(orderMapper.toResponseDto(orderService.a
ddComment(orderId, author, content)));
    }

    @PreAuthorize("hasRole('ADMIN')")
    @PatchMapping("/{orderId}/status")
    public ResponseEntity<OrderResponseDto>
changeOrderStatus(
            @PathVariable Long orderId,
            @RequestParam OrderStatus newStatus) {
        return
ResponseEntity.ok(orderMapper.toResponseDto(orderService.c
hangeStatus(orderId, newStatus)));
    }

    @PreAuthorize("isAuthenticated()")
    @GetMapping("/filter")
    public ResponseEntity<List<OrderResponseDto>>
filterOrders(
            @RequestParam(required = false)
@DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
LocalDate startDate,
            @RequestParam(required = false)
@DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
LocalDate endDate,
            @RequestParam(required = false) OrderStatus status) {

        try {
            return
ResponseEntity.ok(orderService.findByFilters(startDate,
endDate, status).stream()
                    .map(orderMapper::toResponseDto)
                    .collect(Collectors.toList()));
        } catch (IllegalArgumentException e) {
```

```
            throw new
ResponseStatusException(HttpStatus.BAD_REQUEST,
e.getMessage());
        }
    }
}
```

ResourceApiController.java

```
package com.agriculture.resource_turnover.controllers.api;

import
com.agriculture.resource_turnover.dto.ResourceRequestDto;
import
com.agriculture.resource_turnover.dto.ResourceResponseDto;
import
com.agriculture.resource_turnover.dto.mapper.ResourceMappe
r;
import com.agriculture.resource_turnover.models.Resource;
import
com.agriculture.resource_turnover.services.ResourceService;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import
org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.*;
import jakarta.validation.Valid;

import java.util.List;
import java.util.stream.Collectors;

@Slf4j
@RestController
@RequiredArgsConstructor
@RequestMapping("/api/resources")
public class ResourceApiController {

    private final ResourceService resourceService;
    private final ResourceMapper resourceMapper;

    @PreAuthorize("isAuthenticated()")
    @GetMapping
    public ResponseEntity<List<ResourceResponseDto>>
getAllResources() {
        return
ResponseEntity.ok(resourceService.findAll().stream()
                .map(resourceMapper::toResponseDto)
                .collect(Collectors.toList()));
    }

    @PreAuthorize("isAuthenticated()")
    @GetMapping("/active")
    public ResponseEntity<List<ResourceResponseDto>>
getActiveResources() {
        return
ResponseEntity.ok(resourceService.findAllActive().stream()
                .map(resourceMapper::toResponseDto)
                .collect(Collectors.toList()));
    }

    @PreAuthorize("isAuthenticated()")
    @GetMapping("/{id}")
    public ResponseEntity<ResourceResponseDto>
getResourceById(@PathVariable Long id) {
        return
ResponseEntity.ok(resourceMapper.toResponseDto(resourceSe
rvice.findById(id)));
    }

    @PreAuthorize("hasRole('ADMIN')")
    @PostMapping
    public ResponseEntity<ResourceResponseDto>
createResource(
            @Valid @RequestBody ResourceRequestDto dto) {
        Resource created =
resourceService.save(resourceMapper.toEntity(dto));
        return
ResponseEntity.status(HttpStatus.CREATED).body(resourceM
apper.toResponseDto(created));
    }

    @PreAuthorize("hasRole('ADMIN')")
    @PutMapping("/{id}")
    public ResponseEntity<ResourceResponseDto>
updateResource(
            @PathVariable Long id,
            @Valid @RequestBody ResourceRequestDto dto) {
        Resource resource = resourceMapper.toEntity(dto);
        resource.setId(id);

        return
ResponseEntity.ok(resourceMapper.toResponseDto(resourceSe
rvice.save(resource)));
    }

    @PreAuthorize("hasRole('ADMIN')")
    @DeleteMapping("/{id}")
    public ResponseEntity<Void>
deleteResource(@PathVariable Long id) {
        resourceService.deleteById(id);
        return ResponseEntity.noContent().build();
    }

    @PreAuthorize("hasRole('ADMIN')")
    @PostMapping("/{id}/archive")
    public ResponseEntity<ResourceResponseDto>
archiveResource(@PathVariable Long id) {
        return
ResponseEntity.ok(resourceMapper.toResponseDto(resourceSe
rvice.archiveResource(id)));
    }
}
```

SupplierApiController.java

```
package com.agriculture.resource_turnover.controllers.api;

import
com.agriculture.resource_turnover.dto.SupplierRequestDto;
import
com.agriculture.resource_turnover.dto.SupplierResponseDto;
import
com.agriculture.resource_turnover.dto.mapper.SupplierMapper
;
import com.agriculture.resource_turnover.models.Supplier;
import
com.agriculture.resource_turnover.services.SupplierService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import
org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.stream.Collectors;

@RestController
@RequestMapping("/api/suppliers")
```

```
@RequiredArgsConstructor
public class SupplierApiController {

    private final SupplierService supplierService;
    private final SupplierMapper supplierMapper;

    @PreAuthorize("isAuthenticated()")
    @GetMapping
    public ResponseEntity<List<SupplierResponseDto>>
getAllSuppliers() {
        return
ResponseEntity.ok(supplierService.findAll().stream()
                .map(supplierMapper::toResponseDto)
                .collect(Collectors.toList()));
    }

    @PreAuthorize("isAuthenticated()")
    @GetMapping("/active")
    public ResponseEntity<List<SupplierResponseDto>>
getActiveSuppliers() {
        return
ResponseEntity.ok(supplierService.findAllActive().stream()
                .map(supplierMapper::toResponseDto)
                .collect(Collectors.toList()));
    }

    @PreAuthorize("isAuthenticated()")
    @GetMapping("/{id}")
    public ResponseEntity<SupplierResponseDto>
getSupplierById(@PathVariable Long id) {
        return
ResponseEntity.ok(supplierMapper.toResponseDto(supplierSer
vice.findById(id)));
    }

    @PreAuthorize("hasAuthority('ROLE_ADMIN')")
    @PostMapping
    public ResponseEntity<SupplierResponseDto>
createSupplier(@RequestBody SupplierRequestDto dto) {
        Supplier supplier =
supplierService.save(supplierMapper.toEntity(dto));
        return
ResponseEntity.ok(supplierMapper.toResponseDto(supplier));
    }

    @PreAuthorize("hasRole('ADMIN')")
    @PutMapping("/{id}")
    public ResponseEntity<SupplierResponseDto>
updateSupplier(@PathVariable Long id, @RequestBody
SupplierRequestDto dto) {
        Supplier supplier = supplierMapper.toEntity(dto);
        supplier.setId(id);
        return
ResponseEntity.ok(supplierMapper.toResponseDto(supplierSer
vice.save(supplier)));
    }

    @PreAuthorize("hasRole('ADMIN')")
    @DeleteMapping("/{id}")
    public ResponseEntity<Void>
deleteSupplier(@PathVariable Long id) {
        supplierService.deleteById(id);
        return ResponseEntity.noContent().build();
    }

    @PreAuthorize("hasRole('ADMIN')")
    @PostMapping("/{id}/restore")
    public ResponseEntity<SupplierResponseDto>
restoreSupplier(@PathVariable Long id) {
        return
ResponseEntity.ok(supplierMapper.toResponseDto(supplierSer
vice.restoreSupplier(id)));
    }
```