ДЕРЖАВНА СЛУЖБА УКРАЇНИ З НАДЗВИЧАЙНИХ СИТУАЦІЙ

ЛЬВІВСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ БЕЗПЕКИ ЖИТТЄДІЯЛЬНОСТІ

Кафедра інформаційних технологій та систем електронних комунікацій

Р. Р. Головатий, Н. Є. Бурак, І. О. Малець, В. М. Пилипенко, Д. А. Райта, В. Й. Довбняк

РОЗРОБКА ВЕБЗАСТОСУНКІВ НА JAVA SPRING

Збірник задач для підготовки до технічних співбесід

(рівень Junior)

Навчальний посібник

для здобувачів освіти спеціальності 122 «Комп'ютерні науки»
Львівського державного університету безпеки життєдіяльності
з дисципліни «WEB-технології та WEB-програмування»

Львів — 2026

УДК 004.738.5:004.438(075.8)

Рекомендовано до друку вченою радою Львівського державного університету безпеки життєдіяльності (протокол № ___ від «___» _____ 2026 р.)

Автори:

Головатий Роман Русланович — кандидат технічних наук;
Бурак Назарій Євгенович — кандидат технічних наук, доцент;
Малець Ігор Остапович — кандидат технічних наук, доцент;
Пилипенко Володимир Миколайович;
Райта Діана Анатоліївна;
Довбняк Віра Йосипівна.

Рецензенти:

Тригуба Анатолій Миколайович — доктор технічних наук, професор, завідувач кафедри інформаційних технологій Львівського національного університету ветеринарної медицини та біотехнологій імені С. З. Гжицького;

Ткачук Ростислав Львович — доктор технічних наук, професор, професор кафедри управління інформаційною безпекою Львівського державного університету безпеки життєдіяльності;

Розробка вебзастосунків на Java Spring : збірник задач для підготовки до технічних співбесід (рівень Junior) : навч. посіб. / Р. Р. Головатий, Н. Є. Бурак, І. О. Малець, В. М. Пилипенко, Д. А. Райта, В. Й. Довбняк. — Львів : ЛДУБЖД, 2026. — 135 с.

Посібник продовжує збірник «Основи програмування мовою Java» і присвячений розробці вебзастосунків на платформі Spring: ядро Spring, Spring Boot, Spring MVC і REST, Spring Data JPA, Spring Security, тестування, інтеграція сервісів, розгортання та використання моделей штучного інтелекту. Призначений для здобувачів освіти спеціальності 122 «Комп’ютерні науки» Львівського державного університету безпеки життєдіяльності, які вивчають дисципліну «WEB-технології та WEB-програмування» і готуються до співбесіди на позицію Java Junior Developer.

Кожне з 62 питань містить теоретичну частину, приклад задачі з покроковим розбором коду та п’ять варіантів для самостійної роботи. Усі приклади є частинами одного наскрізного проєкту — інформаційної системи чергової частини ЛДУБЖД, яка нарощується від розділу до розділу.

Розглянуто та схвалено на засіданні кафедри інформаційних технологій та систем електронних комунікацій ЛДУБЖД (протокол № ___ від «___» _____ 2026 р.)

УДК 004.738.5:004.438(075.8)

© Головатий Р. Р., Бурак Н. Є., Малець І. О., Пилипенко В. М., Райта Д. А., Довбняк В. Й. 2026
© Львівський державний університет безпеки життєдіяльності, 2026

ЗМІСТ

ВСТУП..... 4

ЧАСТИНА 1. ОСНОВИ SPRING..... 5

Розділ 1. Ядро Spring: IoC та DI..... 5

Розділ 2. Spring Boot..... 18

ЧАСТИНА 2. ВЕБРІВЕНЬ..... 31

Розділ 3. Spring MVC і REST API..... 31

Розділ 4. Серверний рендеринг..... 48

ЧАСТИНА 3. ДАНІ..... 55

Розділ 5. Spring Data JPA..... 55

Розділ 6. Кешування й асинхронність..... 72

ЧАСТИНА 4. БЕЗПЕКА І ЯКІСТЬ..... 81

Розділ 7. Spring Security..... 81

Розділ 8. Тестування..... 94

ЧАСТИНА 5. ІНТЕГРАЦІЯ ТА РОЗГОРТАННЯ..... 106

Розділ 9. Взаємодія сервісів..... 106

Розділ 10. Розгортання..... 116

Розділ 11. Spring і штучний інтелект..... 124

ВИСНОВКИ..... 131

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ..... 132

ВСТУП

Ця книжка — продовження збірника «Основи програмування мовою Java» [1]. Там ми говорили про мову: класи, колекції, винятки, потоки, JDBC, сервлети. Тут мова йтиме про те, що з усім цим робить сучасний Java-розробник щодня, тобто про вебзастосунки на платформі Spring.

Чому саме Spring? Відповідь проста і трохи прозаїчна: це найпоширеніший фреймворк серед Java-розробників [2], і саме його найчастіше згадують у вакансіях Java Junior. Технічна співбесіда з веброзробки на Java зазвичай починається з ядра — що таке IoC-контейнер і як Spring створює об'єкти, — а вже потім переходить до транзакцій, безпеки чи тестування. Тому і посібник побудовано від ядра до периферії.

Матеріал поділено на п'ять частин. Перша присвячена ядру Spring і Spring Boot. Друга — вебрівню: контролерам, REST, валідації, обробці помилок і серверному рендерингу. Третя — даним: Spring Data JPA, транзакціям, міграціям, кешуванню. Четверта — безпеці та тестуванню. П'ята — взаємодії сервісів, розгортанню в Docker і, нарешті, використанню великих мовних моделей у застосунках Spring.

Головна відмінність від першої книжки — наскрізний проєкт. Усі приклади є частинами одного застосунку «Інформаційна система чергової частини ЛДУБЖД». У ньому реєструються виклики (пожежа, рятувальні роботи, хімічна небезпека, хибний виклик), формуються чергові зміни рятувальників, ведеться облік пожежних автомобілів і графік нарядів курсантів. Предметну область ми обирали свідомо: вона знайома курсантам нашого університету і достатньо багата, щоб на ній показати майже всі типові задачі веброзробки — від простого CRUD до безпеки та обміну повідомленнями [3]. У першому розділі застосунок складається з кількох класів у пам'яті. В останньому він уже має REST API, базу даних, авторизацію, тести, Docker-образ і помічника на основі моделі Claude.

Кожне питання має сталу структуру. Спочатку теоретична частина з посиланнями на документацію та літературу. Далі приклад задачі: умова, короткий план розв'язку і лістинг. Лістинг поділено на блоки за файлами проєкту; над кожним блоком указано ім'я файлу, ліворуч код, праворуч пояснення. Після лістингу йде коротка примітка — «Зверніть увагу», «Порада» або «З досвіду викладачів». Завершує питання таблиця з п'ятьма варіантами для самостійної роботи.

Повний код проєкту додається до посібника. Він організований як Maven-проєкт `duty-app` з пакетом `ua.edu.ldubgd.duty`; стан коду після кожного розділу позначено окремою міткою в репозиторії Git (`ch01`, `ch02` тощо), тож можна відкрити саме ту версію, яка відповідає розділу, що вивчається. Класи з префіксом `Demo` у пакеті `demo` — це навчальні точки запуску для окремих питань; у справжньому застосунку їх, звісно, не було б. Приклади написано для Java 25 та Spring Boot 4.1 [4; 5]. Для збирання знадобляться JDK 25, Maven 3.9 або новіший, а починаючи з третьої частини — Docker.

Як працювати з посібником? Наша порада — не читати його як роман. Відкрийте проєкт поруч із текстом, запустіть приклад, змініть щось і подивіться, що зламалося. Варіанти для самостійної роботи виконуйте у власній гілці репозиторію. Звіт оформлюйте так само, як для першої книжки: умова, код, результати запуску, короткі висновки, список джерел за ДСТУ 8302:2015 [6].

Окремо скажемо про асистентів штучного інтелекту. Такі інструменти, як Claude Code, сьогодні є звичайною частиною робочого середовища розробника, і забороняти їх курсантам немає сенсу [7]. Але є чесне застереження: на співбесіді асистента поруч не буде, а на роботі ви відповідатимете за код, який він написав, так само, як за власний. Тому використовуйте його, щоб розібратися в незрозумілому, а не щоб пропустити етап розуміння. Цій темі присвячено останній розділ.

Автори щиро дякують рецензентам за уважне прочитання рукопису і цінні зауваження.

ЧАСТИНА 1. ОСНОВИ SPRING

Розділ 1. Ядро Spring: IoC та DI

Питання 1.1. Inversion of Control і Dependency Injection: навіщо вони потрібні

Теоретична частина

Звичайна Java-програма сама створює всі потрібні їй об'єкти. Якщо службі реєстрації викликів потрібен засіб сповіщення, вона пише `new SmsNotifier()` — і відтепер назавжди прив'язана до SMS. Захочемо надсилати повідомлення в журнал, у месенджер чи нікуди (під час тестів) — доведеться змінювати код самої служби. Така залежність називається жорсткою (tight coupling).

Інверсія керування (Inversion of Control, IoC) пропонує перевернути цю схему: об'єкт не створює свої залежності, а отримує їх ззовні [8]. Хтось інший — інша частина програми або спеціальний контейнер — вирішує, яку саме реалізацію підставити. Найпоширенішою формою IoC є впровадження залежностей (Dependency Injection, DI): залежності передаються об'єкту через конструктор, сетер або поле [9].

Щоб це працювало, служба має залежати не від конкретного класу, а від абстракції — інтерфейсу. Це прямо випливає з принципу інверсії залежностей (літера D у SOLID): модулі верхнього рівня не повинні залежати від модулів нижнього рівня, обидва мають залежати від абстракцій [10].

Важливо розуміти, що DI — це передусім ідея, а не фреймворк. Її можна реалізувати вручну, у методі `main`, без жодної бібліотеки, і саме з цього ми почнемо. Spring лише автоматизує рутинну частину: створює об'єкти, з'єднує їх між собою та керує їхнім життєвим циклом [11]. Коли в застосунку десять класів, ручне з'єднання ще терпиме. Коли їх кілька сотень — без контейнера не обійтися.

На співбесіді питання про IoC часто ставлять так: «Що таке Spring і навіщо він потрібен?». Погана відповідь — «це фреймворк для вебзастосунків». Добра — «це IoC-контейнер, який бере на себе створення об'єктів і впровадження залежностей, а навколо нього вже побудовано модулі для вебу, даних, безпеки тощо».

Приклад задачі

Умова. Створити інтерфейс `Notifier` і дві його реалізації — `SmsNotifier` та `LogNotifier`. Службу `DispatchService`, яка реєструє виклики в `CallRegistry` і надсилає сповіщення, написати так, щоб вона не знала, яку реалізацію сповіщувача використовує. З'єднати об'єкти вручну, без Spring, і показати, що одна й та сама служба працює з обома сповіщувачами.

Анотації Spring (`@Component`, `@Service`) у коді вже є, але в цьому прикладі вони ні на що не впливають: контейнера ще немає. Ми повернемося до них у питанні 1.2.

Лістинг 1.1. Реалізація з покроковим розбором

Файл `Notifier.java` абстракція сповіщувача

```
public interface Notifier {
    void send(String message);
}
```

Інтерфейс описує лише контракт: «уміє надіслати повідомлення». Як саме — його не цікавить. Зауважте, що інтерфейс має один абстрактний метод, тобто є функціональним; це знадобиться нам у питанні 1.3.

Файл `SmsNotifier.java` перша реалізація

```
public class SmsNotifier implements Notifier {
    @Override
    public void send(String message) {
        System.out.println("[SMS] " + message);
    }
}
```

У справжній системі тут був би виклик SMS-шлюзу. Для навчання досить виводу в консоль. Друга реалізація, `LogNotifier`, відрізняється лише префіксом `[ЖУРНАЛ]`.

Файл `DispatchService.java` служба прийому викликів

```
public class DispatchService {
    private final CallRegistry registry;
    private final Notifier notifier;

    public DispatchService(CallRegistry registry,
                           Notifier notifier) {
        this.registry = registry;
        this.notifier = notifier;
    }
```

Ключове місце прикладу. Служба не містить жодного `new`: обидві залежності приходять через конструктор і зберігаються у `final`-полях. Тип поля — інтерфейс `Notifier`, а не конкретний клас.

```
    public FireCall accept(String address, CallType type) {
        FireCall call = registry.register(address, type);
        notifier.send("Виклик №" + call.id() + ": " + type
                + ", " + address);
        return call;
    }
}
```

Метод `accept` реєструє виклик і повідомляє про нього. Яким каналом піде повідомлення — вирішує той, хто створював службу.

Файл `Demo11ManualWiring.java` ручне з'єднання об'єктів

```
Clock clock = Clock.systemDefaultZone();
CallRegistry registry = new CallRegistry(clock);

Notifier sms = new SmsNotifier();
DispatchService day = new DispatchService(registry, sms);
day.accept("вул. Клепарівська, 35", CallType.FIRE);
```

Метод `main` виконує роль «контейнера»: він створює реєстр і сповіщувач та передає їх службі. Це і є впровадження залежностей — поки що вручну.

```
Notifier log = new LogNotifier();
DispatchService night = new DispatchService(registry, log);
night.accept("вул. Городоцька, 12", CallType.RESCUE);

System.out.println("Усього викликів: "
        + registry.findAll().size());
```

Друга служба використовує інший сповіщувач, але той самий реєстр. Код `DispatchService` при цьому не змінився ні на рядок.

Вивід програми результат запуску

```
[SMS] Виклик №1: FIRE, вул. Клепарівська, 35
[ЖУРНАЛ] Виклик №2: RESCUE, вул. Городоцька, 12
Усього викликів: 2
```

З досвіду викладачів. *Часто постає питання: «Навіщо інтерфейс, якщо реалізація поки одна?». Чесна відповідь — інколи він справді не потрібен. Але для залежностей, які спілкуються із зовнішнім світом (SMS, пошта, інший сервіс), інтерфейс майже завжди окупається вже на етапі тестування.*

Варіанти для самостійної роботи

Завдання. За зразком прикладу описати інтерфейс залежності, дві його реалізації та службу, яка отримує залежність через конструктор. З'єднати об'єкти вручну в методі `main` і показати роботу служби з кожною реалізацією.

№	Служба	Залежність і реалізації
1	`ShiftScheduler` (формування зміни)	`RosterSource`: з файлу / з пам'яті
2	`TruckDispatcher` (виїзд автомобіля)	`RouteCalculator`: найкоротший / без платних доріг
3	`CadetDutyPlanner` (наряди курсантів)	`CalendarProvider`: робочі дні / усі дні
4	`ReportService` (звіт за добу)	`ReportFormatter`: текст / CSV
5	`HydrantChecker` (перевірка гідрантів)	`PressureSensor`: справжній / імітація

Питання 1.2. Бін і ApplicationContext. Анотації @Component, @Service, @Repository

Теоретична частина

Бін (bean) у термінології Spring — це об'єкт, який створює, налаштовує і зберігає IoC-контейнер [11]. Звичайний `new FireCall(...)` у коді — не бін, навіть якщо клас позначено анотацією. Бін з'являється лише тоді, коли контейнер сам вирішив його створити.

Сам контейнер в API Spring представлений інтерфейсом `ApplicationContext`. Він знає про всі біни, уміє повернути бін за типом чи за іменем (`getBean`), публікує події, завантажує ресурси й повідомлення. Існує й простіший інтерфейс `BeanFactory`, від якого `ApplicationContext` успадковується; на практиці ним майже ніколи не користуються напряму, але різницю між ними на співбесідах інколи запитують.

Звідки контейнер дізнається, які класи перетворити на біни? Найпоширеніший спосіб — сканування компонентів. Ми вказуємо пакет (`@ComponentScan`), і Spring знаходить у ньому класи, позначені стереотипними анотаціями [12]:

`@Component` — загальна позначка «це бін»;

`@Service` — бін з бізнес-логікою;

`@Repository` — бін доступу до даних; Spring додатково перетворює винятки конкретної бази даних на власну ієрархію `DataAccessException`;

`@Controller` і `@RestController` — біни вебрівня (розділ 3).

Технічно `@Service` нічим не відрізняється від `@Component` — це та сама анотація з іншим іменем. Різниця смислова: вона повідомляє читачеві коду, до якого шару належить клас. А от `@Repository` має реальний додатковий ефект, описаний вище. Ця деталь — улюблене уточнювальне питання інтерв'юерів.

Ім'я біна за замовчуванням утворюється з імені класу з малої літери: `DispatchService` → `dispatchService`. Його можна задати явно: `@Component("logNotifier")`.

Приклад задачі

Умова. Створити конфігураційний клас `DutyConfig` зі скануванням пакета `ua.edu.ldubgd.duty`. Підняти контекст `AnnotationConfigApplicationContext`, отримати з нього службу `DispatchService` і реєстр `CallRegistry`, зареєструвати виклик і вивести імена всіх бінів застосунку.

Зверніть увагу: тепер ми ніде не пишемо `new DispatchService(...)`. Контейнер сам знайде конструктор, побачить, що йому потрібні `CallRegistry` і `Notifier`, і передасть відповідні біни.

Лістинг 1.2. Реалізація з покроковим розбором

Файл `DutyConfig.java` конфігурація контейнера

```
@Configuration
@ComponentScan("ua.edu.ldubgd.duty")
public class DutyConfig {
```

`@Configuration` позначає клас як джерело визначень бінів, а `@ComponentScan` вмикає пошук компонентів у пакеті та всіх його підпакетах. Метод `dutyClock` цього класу розглянемо в питанні 1.6.

Файл `CallRegistry.java` бін доступу до даних

```
@Repository
public class CallRegistry {
    private final List<FireCall> calls = new ArrayList<>();
    private final Clock clock;
    private long nextId = 1;

    public CallRegistry(Clock clock) {
        this.clock = clock;
    }
```

Поки що виклики зберігаються у списку в пам’яті; у розділі 5 цей клас поступиться місцем репозиторію Spring Data JPA. Анотація `@Repository` вже зараз правильно описує його роль.

```
public FireCall register(String address, CallType type) {
    FireCall call = new FireCall(nextId++, address, type,
            LocalDateTime.now(clock));
    calls.add(call);
    return call;
}
```

`FireCall` — це запис (record), тобто незмінний носій даних [13]. Він не є біном: виклики створює сам застосунок, і їх може бути скільки завгодно.

Файл `Demo12Context.java` запуск контейнера

```
try (var ctx =
    new AnnotationConfigApplicationContext(DutyConfig.class)) {
    var service = ctx.getBean(DispatchService.class);
    service.accept("вул. Клепарівська, 35", CallType.FIRE);

    CallRegistry registry = ctx.getBean(CallRegistry.class);
    System.out.println("У реєстрі: "
            + registry.findAll().size());
```

Контекст реалізує `AutoCloseable`, тому його зручно відкривати в `try-with-resources`. `getBean` за типом повертає вже готовий, повністю з’єднаний об’єкт.

```
Arrays.stream(ctx.getBeanDefinitionNames())
      .filter(n -> !n.startsWith("org.springframework"))
      .sorted()
      .forEach(n -> System.out.println("  бін: " + n));
```

Відфільтровуємо службові біни самого Spring і виводимо лише наші. Серед них є й `dutyConfig`: конфігураційний клас теж є біном.

Вивід програми результат запуску

```
[SMS] Виклик №1: FIRE, вул. Клепарівська, 35
У реєстрі: 1
  бін: alarmService
  бін: callForm
  бін: callRegistry
  бін: dispatchService
  бін: dutyClock
  бін: dutyConfig
  бін: dutyOfficer
  бін: logNotifier
  бін: smsNotifier
  бін: vehicleDirectory
```

Зверніть увагу. *Під час запуску ви також побачите рядки, які друкує бін `VehicleDirectory` («Конструктор…», «@PostConstruct…»). Ми навмисно не прибирали їх з виводу: вони показують, що контейнер створює всі singleton-біни одразу під час старту, навіть ті, які ми не запитували через `getBean`. Детальніше — у питанні 1.5.*

Варіанти для самостійної роботи

Завдання. Позначити класи свого варіанта відповідними стереотипними анотаціями, підняти контекст через `AnnotationConfigApplicationContext` і вивести імена всіх бінів. Окремо пояснити, чому обрано саме таку анотацію для кожного класу.

№	Класи	Що вивести додатково
1	`RescuerRegistry`, `ShiftService`, `ShiftReport`	кількість бінів у контексті
2	`TruckRepository`, `FleetService`, `FuelCalculator`	чи є в контексті бін з іменем `fleetService`
3	`CadetRepository`, `DutyService`, `DutyPrinter`	тип біна `dutyPrinter`
4	`HydrantRepository`, `InspectionService`, `MapRenderer`	імена бінів, що закінчуються на `Service`
5	`EquipmentRepository`, `WarehouseService`, `BarcodeReader`	бін, отриманий за іменем через `getBean(String)`

Питання 1.3. Способи впровадження залежностей; чому краще через конструктор

Теоретична частина

Spring підтримує три способи передати бінові його залежності [12]. Через конструктор: залежності — параметри конструктора. Через сетер: контейнер викликає метод `setXxx`, позначений `@Autowired`. Через поле: анотація `@Autowired` ставиться безпосередньо на поле, навіть приватне, і Spring записує туди значення через рефлексію.

Якщо в класі єдиний конструктор, анотація `@Autowired` на ньому не потрібна — Spring використає його автоматично. Саме так написані всі наші служби.

Офіційна документація Spring і більшість практиків рекомендують впровадження через конструктор для обов'язкових залежностей [12; 14]. Аргументів кілька. Об'єкт неможливо створити в неповному стані: без потрібних параметрів конструктор просто не викличеш. Поля можна зробити `final`, і клас стає незмінним щодо своїх залежностей. Клас легко створити в модульному тесті звичайним `new`, без Spring і без рефлексії. І нарешті, довгий список параметрів конструктора добре видно — це чесний сигнал, що клас робить забагато.

Впровадження через поле виглядає найкоротшим, тому його так люблять у навчальних прикладах з інтернету. Але воно ховає залежності, не дозволяє `final` і робить клас непридатним до використання поза контейнером. Сьогодні IntelliJ IDEA навіть підсвічує його попередженням «Field injection is not recommended».

Сетер має сенс для необов'язкових залежностей, які можна замінити після створення об'єкта. На практиці це трапляється рідко.

Ще одна перевага конструктора, про яку варто згадати на співбесіді: циклічні залежності (A потребує B, а B — A) з ним виявляються одразу під час старту, і Spring Boot за замовчуванням забороняє їх. Це добре — цикл майже завжди свідчить про помилку в проєктуванні.

Приклад задачі

Умова. Показати головну практичну перевагу впровадження через конструктор: створити `DispatchService` у звичайному коді без Spring, підставивши фіксований

годинник і «сповіщувач», який просто збирає повідомлення у список. Перевірити текст надісланого повідомлення і час реєстрації виклику.

Такий прийом — підміна справжньої залежності простою заглушкою — лежить в основі модульного тестування, якому присвячено розділ 8.

Лістинг 1.3. Реалізація з покроковим розбором

Файл `Demo13ConstructorInjection.java` служба без контейнера

```
Clock fixed = Clock.fixed(
        Instant.parse("2026-09-01T06:00:00Z"),
        ZoneId.of("Europe/Kyiv"));
CallRegistry registry = new CallRegistry(fixed);
```

`Clock.fixed` завжди повертає один і той самий момент часу. Саме тому `CallRegistry` отримує годинник через конструктор, а не викликає `LocalDateTime.now()` напряму: тепер час можна контролювати.

```
List<String> sent = new ArrayList<>();
var service = new DispatchService(registry, sent::add);
```

`Notifier` — функціональний інтерфейс, тож замість окремого класу підставляємо посилання на метод `sent::add`. Кожне «надіслане» повідомлення просто потрапить у список.

```
service.accept("вул. Шевченка, 7", CallType.HAZMAT);

System.out.println("Надіслано: " + sent);
System.out.println("Час: "
        + registry.findAll().get(0).receivedAt());
```

Тепер можна точно перевірити, що й коли сталося. 06:00 за UTC — це 09:00 за київським часом (1 вересня діє літній час, UTC+3).

Вивід програми результат запуску

```
Надіслано: [Виклик №1: HAZMAT, вул. Шевченка, 7]
Час: 2026-09-01T09:00
```

Порада. *Спробуйте переписати `DispatchService` із впровадженням через поле (`@Autowired private Notifier notifier;`) і повторити цей приклад. Ви швидко переконаєтеся, що без Spring або рефлексії підставити заглушку вже не вийде.*

Варіанти для самостійної роботи

Завдання. Написати службу свого варіанта з впровадженням залежностей через конструктор і `final`-поля. У методі `main` створити її без Spring, підставивши фіксований `Clock` і лямбда-заглушку, та перевірити результат.

№	Служба	Залежності	Що перевірити
1	`ShiftService`	`Clock`, `ShiftNotifier`	текст повідомлення про початок зміни о 08:00
2	`FuelLog`	`Clock`, `FuelSensor`	що запис має дату, задану годинником
3	`CadetLeaveService`	`Clock`, `ApprovalSender`	що звільнення у вихідні не погоджується
4	`InspectionPlanner`	`Clock`, `MailSender`	дату наступної перевірки гідранта (+90 днів)
5	`AlarmTimer`	`Clock`, `Siren`	що сирена не вмикається вночі з 23:00 до 06:00

Питання 1.4. Області видимості бінів: singleton і prototype

Теоретична частина

Скільки екземплярів біна створить контейнер? Відповідь залежить від області видимості (scope) [15]. За замовчуванням усі біни мають область `singleton`: на весь контекст створюється рівно один об'єкт, і кожен, хто його запитує, отримує посилання на той самий екземпляр.

Не плутайте це з патерном «Одинак» (Singleton) з книжки «банди чотирьох» [16]. Класичний одинак гарантує один екземпляр на весь процес JVM за допомогою приватного конструктора і статичного поля. Singleton у Spring означає «один на контейнер»: можна підняти два контексти й отримати два об'єкти, а сам клас лишається звичайним.

Область `prototype` діє протилежно: кожен виклик `getBean` або кожне впровадження створює новий об'єкт. Знищенням прототипів контейнер не займається — методи з `@PreDestroy` для них не викликаються.

Для вебзастосунків є ще області `request`, `session` і `application`; їх використовують значно рідше.

Оскільки singleton-бін один на всіх, він спільний для всіх потоків. Звідси практичне правило: служби мають бути без змінного стану (stateless) або потокобезпечними [17]. Наш `CallRegistry` зі звичайним `ArrayList` всередині для навчального прикладу підходить, але в багатопотоковому вебзастосунку його довелося б синхронізувати — у розділі 5 ми просто передамо зберігання базі даних.

І нарешті, класична пастка, яку обов'язково запитують на співбесіді. Якщо prototype-бін впровадити в singleton через конструктор, він буде створений один раз — разом із singleton — і далі житиме вічно. Тобто «прототип» фактично стане одинаком. Щоразу отримувати новий екземпляр допомагає `ObjectProvider<T>` (або `@Lookup`-метод).

Приклад задачі

Умова. Створити prototype-бін `CallForm` — чернетку картки виклику, яку черговий заповнює для кожного нового звернення. Показати, що `CallRegistry` один на контекст, а `CallForm` щоразу новий. Створити службу `DutyOfficer`, яка видає нову чернетку через `ObjectProvider`, і переконатися, що дві чернетки не впливають одна на одну.

Лістинг 1.4. Реалізація з покроковим розбором

Файл `CallForm.java` бін з областю prototype

```
@Component
@Scope("prototype")
public class CallForm {
    private String address;
    private CallType type = CallType.FIRE;
```

На відміну від служб, чернетка має змінний стан, тож ділити один екземпляр між усіма черговими не можна. Тому область — `prototype`.

```
public void fill(String address, CallType type) {
    this.address = address;
    this.type = type;
}
```

Метод `summary` (у лістингу не показано) повертає рядок на кшталт «FIRE — вул. Зелена, 101» або «адресу не вказано».

Файл `DutyOfficer.java` singleton, що видає прототипи

```
@Service
public class DutyOfficer {
    private final ObjectProvider<CallForm> forms;

    public DutyOfficer(ObjectProvider<CallForm> forms) {
        this.forms = forms;
    }

    public CallForm newForm() {
        return forms.getObject();
    }
}
```

`DutyOfficer` — singleton. Якби ми впровадили в нього `CallForm` напряму, усі чергові отримували б одну й ту саму чернетку. `ObjectProvider` звертається до контейнера при кожному виклику `getObject()`.

Файл `Demo14Scopes.java` порівняння екземплярів

```
CallRegistry r1 = ctx.getBean(CallRegistry.class);
CallRegistry r2 = ctx.getBean(CallRegistry.class);
System.out.println("singleton: r1 == r2 -> " + (r1 == r2));

CallForm f1 = ctx.getBean(CallForm.class);
CallForm f2 = ctx.getBean(CallForm.class);
System.out.println("prototype: f1 == f2 -> " + (f1 == f2));
```

Порівнюємо посилання оператором ==, а не `equals`: нас цікавить саме тотожність об’єктів (див. питання про == і `equals` у першій книжці [1]).

```
DutyOfficer officer = ctx.getBean(DutyOfficer.class);
CallForm a = officer.newForm();
CallForm b = officer.newForm();
a.fill("вул. Зелена, 101", CallType.FIRE);
System.out.println("a: " + a.summary());
System.out.println("b: " + b.summary());
```

Заповнюємо лише першу чернетку. Друга лишається порожньою — отже, це справді різні об’єкти.

Вивід програми результат запуску

```
singleton: r1 == r2 -> true
prototype: f1 == f2 -> false
a: FIRE — вул. Зелена, 101
b: FIRE — адресу не вказано
```

Зверніть увагу. *Замініть у `DutyOfficer` поле `ObjectProvider<CallForm>` на `CallForm` і повертайте його з `newForm()`. Після заповнення чернетки `a` чернетка `b` теж «отримає» адресу — це та сама пастка, про яку йшлося в теоретичній частині.*

Варіанти для самостійної роботи

Завдання. Створити prototype-бін і singleton-службу, яка видає нові екземпляри через `ObjectProvider`. Продемонструвати різницю областей видимості порівнянням посилань.

№	Prototype-бін	Singleton-служба
1	`ShiftDraft` (чернетка зміни)	`ShiftPlanner`
2	`InspectionAct` (акт перевірки)	`Inspector`
3	`LeaveRequest` (рапорт на звільнення)	`CadetOffice`
4	`FuelReceipt` (чек на пальне)	`FleetManager`
5	`TrainingSession` (заняття на полігоні)	`TrainingScheduler`

Питання 1.5. Життєвий цикл біна, @PostConstruct і @PreDestroy

Теоретична частина

Від створення до знищення бін проходить кілька етапів [18]. Спрощено їх можна перелічити так:

1) контейнер читає визначення біна (клас, область видимості, залежності);
2) створює екземпляр — викликає конструктор, передаючи в нього залежності;
3) впроваджує залежності через сетери та поля, якщо такі є;
4) викликає методи ініціалізації — насамперед позначені `@PostConstruct`;
5) бін готовий і використовується застосунком;
6) під час закриття контексту викликає методи знищення — `@PreDestroy`.

Між етапами 3 і 4, а також після 4 працюють `BeanPostProcessor` — спеціальні біни, що можуть змінити або обгорнути будь-який інший бін. Саме так Spring додає транзакції, кешування та безпеку: замість вашого об’єкта в контексті опиняється проксі. До цієї теми ми повернемося в розділах 5 і 6.

Анотації `@PostConstruct` і `@PreDestroy` належать не до Spring, а до специфікації Jakarta Annotations, тому їх імпортують з пакета `jakarta.annotation` [19]. Якщо ви бачите в старому коді `javax.annotation.PostConstruct` — це версія до Spring Boot 3, і з поточними версіями вона вже не працює.

Навіщо окремий метод ініціалізації, якщо є конструктор? По-перше, у момент виклику `@PostConstruct` усі залежності вже впроваджено, зокрема через поля й сетери. По-друге, конструктор прийнято залишати простим: важка робота (читання довідника, підключення до зовнішнього сервісу) у ньому ускладнює тестування та налагодження. Альтернативи — інтерфейси `InitializingBean`/`DisposableBean` або атрибути `initMethod`/`destroyMethod` в анотації `@Bean`.

Пам’ятайте: для prototype-бінів `@PreDestroy` не викликається. Контейнер видав об’єкт і забув про нього.

Приклад задачі

Умова. Створити бін `VehicleDirectory` — довідник пожежних автомобілів частини. Під час ініціалізації довідник «завантажує» перелік автомобілів, а під час закриття контексту повідомляє про завершення роботи. Вивести номери етапів життєвого циклу і перелік автомобілів, готових до виїзду.

Лістинг 1.5. Реалізація з покроковим розбором

Файл `VehicleDirectory.java` довідник автомобілів

```
@Component
public class VehicleDirectory {
    private final List<FireTruck> trucks = new ArrayList<>();

    public VehicleDirectory() {
        System.out.println("1. Конструктор: список порожній ("
                + trucks.size() + ")");
    }
```

Конструктор лише створює порожній список. Друкуємо його розмір, щоб побачити: на цьому етапі даних ще немає.

```
@PostConstruct
void load() {
    trucks.add(new FireTruck("ВС 0101 ДС", "АЦ-40", true));
    trucks.add(new FireTruck("ВС 0102 ДС", "АД-30", false));
    System.out.println("2. @PostConstruct: завантажено "
            + trucks.size() + " автомобілі");
}
```

Метод ініціалізації. Реальний застосунок читав би дані з бази чи файлу; ми додаємо дві машини — автоцистерну і автодрабину, яка зараз на ремонті. Метод може бути не `public`.

```
    public List<FireTruck> available() {
        return trucks.stream()
                .filter(FireTruck::available)
                .toList();
    }

    @PreDestroy
    void shutdown() {
        System.out.println("3. @PreDestroy: закрито");
    }
}
```

`available()` повертає лише справні машини. `@PreDestroy` — місце для звільнення ресурсів: закриття з'єднань, збереження кешу на диск тощо.

Файл `Demo15Lifecycle.java` відкриття і закриття контексту

```
var ctx =
    new AnnotationConfigApplicationContext(DutyConfig.class);

var directory = ctx.getBean(VehicleDirectory.class);
System.out.println("   Готові до виїзду: "
        + directory.available());

ctx.close();
System.out.println("4. Контекст закрито");
```

Тут ми свідомо не використовуємо `try-with-resources`, щоб явно показати виклик `close()`. Без нього методи `@PreDestroy` не виконаються.

Вивід програми результат запуску

```
1. Конструктор: список порожній (0)
2. @PostConstruct: завантажено 2 автомобілі
   Готові до виїзду: [FireTruck[plate=ВС 0101 ДС, model=АЦ-40, available=true]]
3. @PreDestroy: закрито
4. Контекст закрито
```

Порада. *Позначте `VehicleDirectory` анотацією `@Scope("prototype")` і запустіть приклад ще раз. Рядок «3. @PreDestroy» зникне з виводу. Це добрий спосіб запам'ятати правило про знищення прототипів раз і назавжди.*

Варіанти для самостійної роботи

Завдання. Створити бін свого варіанта з методами `@PostConstruct` і `@PreDestroy`. Вивести етапи життєвого циклу та результат роботи основного методу біна.

№	Бін	Ініціалізація	Основний метод
1	`HydrantMap`	завантажити 5 гідрантів із координатами	гідранти в радіусі 500 м від точки
2	`RescuerRoster`	завантажити список рятувальників	рятувальники з досвідом понад 5 років
3	`DutyCalendar`	сформувати наряди курсантів на тиждень	хто чергує в заданий день
4	`EquipmentStock`	завантажити залишки спорядження	позиції, яких менше мінімуму
5	`StreetDirectory`	завантажити вулиці Львова з районами	район за назвою вулиці

Питання 1.6. Java-конфігурація: @Configuration і @Bean; кілька реалізацій, @Qualifier і @Primary

Теоретична частина

Сканування компонентів працює лише для власних класів: на чужий клас з бібліотеки анотацію `@Component` не поставиш. Для таких випадків служить Java-конфігурація [20]. У класі з `@Configuration` пишемо метод, що створює й повертає об’єкт, і позначаємо його `@Bean`. Ім’я методу стає іменем біна. Так у нашому проєкті з’являється бін `Clock` з часовим поясом Києва.

Чому `@Configuration`, а не просто `@Component`? Spring створює для конфігураційного класу проксі (за допомогою CGLIB), і тому, якщо один `@Bean`-метод викликає інший, повертається вже наявний singleton, а не новий об’єкт. У «легкому» режимі (`@Configuration(proxyBeanMethods = false)` або `@Bean` у звичайному `@Component`) такий виклик створив би ще один екземпляр.

Тепер друга частина питання. У нас дві реалізації `Notifier`. Коли служба просить «дай мені `Notifier`», контейнер не знає, що обрати, і зупиняє старт з винятком `NoUniqueBeanDefinitionException`. Є три способи розв’язати неоднозначність [12]:

`@Primary` на одній з реалізацій — вона стає варіантом за замовчуванням;

`@Qualifier("ім’я")` у точці впровадження — явно вказуємо, яку реалізацію потрібно;

впровадження колекції `List<Notifier>` або `Map<String, Notifier>` — отримуємо одразу всі реалізації.

На співбесіді корисно сказати, що `@Qualifier` має вищий пріоритет за `@Primary`: якщо в точці впровадження вказано кваліфікатор, позначка «основний» ігнорується.

Приклад задачі

Умова. Позначити `SmsNotifier` як основну реалізацію, а `LogNotifier` назвати явно. Створити службу `AlarmService`, яка надсилає тестові сигнали лише в журнал, а загальні оголошення — одразу всіма каналами. Описати бін `Clock` у конфігураційному класі.

Лістинг 1.6. Реалізація з покроковим розбором

Файл `DutyConfig.java` бін із чужого класу

```
@Bean
public Clock dutyClock() {
    return Clock.system(ZoneId.of("Europe/Kyiv"));
}
```

`Clock` — клас JDK, тому описуємо його методом `@Bean`. Явний часовий пояс важливий: сервер у хмарі зазвичай працює в UTC, а журнал викликів має показувати київський час.

Файл `SmsNotifier.java` основна реалізація

```
@Component
@Primary
public class SmsNotifier implements Notifier {
```

`@Primary` — «бери мене, якщо не сказано інакше». Тому `DispatchService` з питання 1.1 отримує саме SMS-сповіщувач, хоча його конструктор просить просто `Notifier`.

Файл `LogNotifier.java` реалізація з явним іменем

```
@Component("logNotifier")
public class LogNotifier implements Notifier {
```

Ім'я `logNotifier` збігається з тим, що Spring дав би за замовчуванням, але, записане явно, воно захищає від випадкового перейменування класу.

Файл `AlarmService.java` вибір реалізації в точці впровадження

```
private final Notifier journal;
private final List<Notifier> all;

public AlarmService(
        @Qualifier("logNotifier") Notifier journal,
        List<Notifier> all) {
    this.journal = journal;
    this.all = all;
}
```

Перший параметр завдяки `@Qualifier` отримує саме журнал, попри `@Primary` у `SmsNotifier`. Другий — список усіх бінів типу `Notifier`.

```
public void testAlarm() {
    journal.send("Перевірка системи оповіщення");
}

public void broadcast(String message) {
    all.forEach(n -> n.send(message));
}
```

Тестовий сигнал не повинен розбудити весь особовий склад, тому йде лише в журнал. Оголошення про збір зміни розсилається всіма каналами.

Вивід програми результат запуску

```
[SMS] Виклик №1: FIRE, просп. Чорновола, 59
[ЖУРНАЛ] Перевірка системи оповіщення
[ЖУРНАЛ] Збір чергової зміни о 08:00
[SMS] Збір чергової зміни о 08:00
Годинник: Europe/Kyiv
```

З досвіду викладачів. *Порядок елементів у `List<Notifier>` не гарантовано: у нашому виводі журнал опинився першим лише через алфавітний порядок сканування. Якщо порядок важливий, позначте реалізації анотацією `@Order(1)`, `@Order(2)`. Цю деталь легко пропустити, а на код-рев'ю її помітять.*

Варіанти для самостійної роботи

Завдання. Для інтерфейсу свого варіанта створити дві-три реалізації, одну позначити `@Primary`. Написати службу, яка в одному місці використовує реалізацію за замовчуванням, у другому — через `@Qualifier`, а в третьому — усі реалізації списком. Додати один бін класу JDK через `@Bean`.

№	Інтерфейс	Реалізації	Бін через @Bean
1	`ReportExporter`	PDF (основна), CSV, XML	`DateTimeFormatter` з шаблоном `dd.MM.yyyy`
2	`MapProvider`	OpenStreetMap (основна), офлайн-карта	`Locale` для української мови
3	`CadetNotifier`	e-mail (основна), Telegram, SMS	`Clock` з поясом `Europe/Kyiv`
4	`WeatherSource`	офіційний сервіс (основна), заглушка	`Random` з фіксованим зерном
5	`FuelPriceSource`	заправка А (основна), заправка Б, середня ціна	`NumberFormat` для гривні

Розділ 2. Spring Boot

Питання 2.1. Чим Spring Boot відрізняється від Spring; стартери

Теоретична частина

У першому розділі ми самі піднімали контейнер: писали конфігураційний клас, вказували пакет для сканування, створювали `AnnotationConfigApplicationContext`. Для навчання це корисно, але в реальному вебзастосунку до цього додаються ще десятки рішень: який вебсервер, як серіалізувати JSON, куди писати логи, як підключитися до бази даних, які версії бібліотек сумісні між собою. Колись на це йшли дні, а XML-конфігурація займала сотні рядків.

Spring Boot — це не інший фреймворк, а надбудова над Spring Framework, яка бере ці рішення на себе [21; 22]. Всередині працює той самий IoC-контейнер, ті самі біни й анотації. Boot додає три речі.

Перша — стартери (starters). Стартер — це Maven-залежність, яка сама по собі майже не містить коду, зате тягне за собою узгоджений набір бібліотек для певної задачі. `spring-boot-starter-webmvc` дає Spring MVC, вбудований Tomcat і Jackson; `spring-boot-starter-actuator` — засоби моніторингу; `spring-boot-starter-test` — JUnit, Mockito, AssertJ. Версії всіх цих бібліотек задає батьківський POM `spring-boot-starter-parent`, тому в нашому `pom.xml` номерів версій майже немає.

Друга — автоконфігурація: Boot аналізує, що лежить у classpath, і сам створює потрібні біни. Їй присвячено наступне питання.

Третя — вбудований сервер і виконуваний jar. Застосунок запускається звичайною командою `java -jar duty-app.jar`, а не розгортається у зовнішньому Tomcat, як сервлети в першій книжці [1].

Зауважимо, що в Spring Boot 4 стартери стали дрібнішими: замість одного великого `spring-boot-starter-web` рекомендовано `spring-boot-starter-webmvc`, а модулі

автоконфігурації розбито за технологіями [4]. Старі назви ще працюють, але в нових проєктах краще використовувати нові.

На співбесіді на питання «чим Boot відрізняється від Spring» досить трьох тез: стартери замість ручного підбору залежностей, автоконфігурація замість ручного опису інфраструктурних бінів, вбудований сервер замість розгортання WAR-файлу.

Приклад задачі

Умова. Перетворити проєкт чергової частини на застосунок Spring Boot: підключити стартери, створити головний клас з анотацією `@SpringBootApplication` і запустити його без вебсервера, щоб зареєструвати один виклик і подивитися, скільки бінів створив Boot.

Порівняйте кількість бінів з виводом питання 1.2: там їх було десять.

Лістинг 2.1. Реалізація з покроковим розбором

Файл `pom.xml` батьківський POM і стартери

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>4.1.1</version>
    <relativePath/>
</parent>
```

Батьківський POM задає версії сотень бібліотек, налаштування компілятора і плагінів. Оновлення Spring Boot зводиться до зміни одного числа.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webmvc</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Стартери без версій. Вебстартер знадобиться повною мірою в розділі 3, а Actuator — у питанні 2.6.

Файл `DutyApplication.java` точка входу застосунку

```
@SpringBootApplication
public class DutyApplication {
    public static void main(String[] args) {
        SpringApplication.run(DutyApplication.class, args);
    }
}
```

Увесь «запуск» — один рядок. `SpringApplication.run` створює відповідний тип контексту, читає конфігурацію, вмикає автоконфігурацію і, якщо є вебстартер, запускає Tomcat.

Файл `Demo21BootApp.java` запуск без вебсервера

```
var ctx = new SpringApplicationBuilder(DutyApplication.class)
        .web(WebApplicationType.NONE)
        .run(args);
```

`SpringApplicationBuilder` дозволяє налаштувати запуск у коді. Тут ми вимикаємо вебсервер, щоб програма завершилася після виконання.

```
ctx.getBean(DispatchService.class)
   .accept("вул. Клепарівська, 35", CallType.FIRE);
System.out.println("Бінів у контексті: "
        + ctx.getBeanDefinitionCount());
ctx.close();
```

Наші біни з розділу 1 працюють без змін. А от загальна кількість бінів зросла в рази: решту створила автоконфігурація.

Вивід програми результат запуску

```
 :: Spring Boot ::                (v4.1.1)
... INFO ... Starting Demo21BootApp using Java 25 ...
[SMS] Виклик №1: FIRE, вул. Клепарівська, 35
... INFO ... DispatchService : Зареєстровано виклик №1
Бінів у контексті: 168
```

Зверніть увагу. *Точна кількість бінів залежить від версії Boot і набору стартерів, тож у вас число може трохи відрізнятися. Важливий не сам результат, а порядок величини: десяток наших бінів проти півтори сотні інфраструктурних.*

Варіанти для самостійної роботи

Завдання. Створити новий проєкт Spring Boot для підсистеми свого варіанта (через Spring Initializr або вручну), додати вказані стартери, перенести в нього службу з розділу 1 і запустити без вебсервера. Вивести кількість бінів у контексті до і після додавання другого стартера.

№	Підсистема	Стартери
1	облік рятувальників	`spring-boot-starter`, `spring-boot-starter-json`
2	автопарк частини	`spring-boot-starter`, `spring-boot-starter-validation`
3	наряди курсантів	`spring-boot-starter`, `spring-boot-starter-cache`
4	склад спорядження	`spring-boot-starter`, `spring-boot-starter-actuator`
5	карта гідрантів	`spring-boot-starter`, `spring-boot-starter-jdbc`

Питання 2.2. Автоконфігурація і @SpringBootApplication

Теоретична частина

Анотація `@SpringBootApplication` — це скорочення для трьох анотацій [21]:

`@SpringBootConfiguration` — клас є конфігураційним (різновид `@Configuration`);

`@ComponentScan` — сканувати пакет головного класу і всі вкладені пакети;

`@EnableAutoConfiguration` — увімкнути автоконфігурацію.

З другого пункту випливає практичне правило, яке часто перевіряють на співбесіді: головний клас має лежати в кореневому пакеті застосунку. Якщо покласти його, скажімо, у `ua.edu.ldubgd.duty.config`, Spring не побачить служб у пакеті `call`.

Автоконфігурація — це набір звичайних `@Configuration`-класів усередині Spring Boot, перелічених у файлі `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfigurati`

on.imports [23]. Кожен такий клас обвішано умовами (@Conditional...), і бін створюється лише тоді, коли умови виконано. Найпоширеніші умови:

@ConditionalOnClass — у classpath є певний клас (наприклад, драйвер бази даних);

@ConditionalOnMissingBean — користувач не оголосив власного біна такого типу;

@ConditionalOnProperty і @ConditionalOnBooleanProperty — у конфігурації задано певне значення.

Саме @ConditionalOnMissingBean пояснює, чому автоконфігурація не заважає: оголосіть власний бін — і Boot відступить. Це принцип «розумних значень за замовчуванням» (convention over configuration).

Ці самі умови можна ставити й на власні біни. Подивитися, які автоконфігурації спрацювали, а які ні і чому, можна, запустивши застосунок з ключем --debug: у вивід потрапить звіт про умови (conditions evaluation report). Корисно хоча б раз його переглянути — він довгий, але дуже показовий.

Приклад задачі

Умова. Зробити SMS-сповіщувач умовним: він має створюватися, лише якщо властивість `duty.sms.enabled` дорівнює `true` (або взагалі не задана). Запустити застосунок двічі, з різними значеннями властивості, і показати, які сповіщувачі опинилися в контексті і який із них отримує `DispatchService`. Заодно перевірити наявність біна `applicationTaskExecutor`, створеного автоконфігурацією.

Лістинг 2.2. Реалізація з покроковим розбором

Файл `SmsNotifier.java` умовний бін

```
@Component
@Primary
@ConditionalOnBooleanProperty(name = "duty.sms.enabled",
                              matchIfMissing = true)
public class SmsNotifier implements Notifier {
```

matchIfMissing = true — якщо властивості немає взагалі, бін створюється. Так демо розділу 1, які не знають про властивості, працюють як раніше.

Файл `Demo22Conditions.java` два запуски з різними умовами

```
for (String sms : new String[] {"true", "false"}) {
    var ctx = new SpringApplicationBuilder()
        .sources(DutyApplication.class)
        .web(WebApplicationType.NONE)
        .logStartupInfo(false)
        .bannerMode(Banner.Mode.OFF)
        .run("--duty.sms.enabled=" + sms);
```

Значення передаємо як аргумент командного рядка. Чому не через метод properties(...) — див. примітку після лістингу.

```
    System.out.println("duty.sms.enabled=" + sms + " -> "
            + ctx.getBeansOfType(Notifier.class).keySet());
    Notifier main = ctx.getBean(Notifier.class);
    System.out.println("  основний: "
            + main.getClass().getSimpleName());
    System.out.println("  applicationTaskExecutor: "
            + ctx.containsBean("applicationTaskExecutor"));
    ctx.close();
}
```

Коли SMS-біна немає, LogNotifier залишається єдиною реалізацією і впроваджується без жодного @Qualifier. Виконавець задач створила автоконфігурація, бо ми не оголосили власного.

Вивід програми результат запуску

```
duty.sms.enabled=true -> [logNotifier, smsNotifier]
  основний: SmsNotifier
  applicationTaskExecutor: true
duty.sms.enabled=false -> [logNotifier]
  основний: LogNotifier
  applicationTaskExecutor: true
```

З досвіду викладачів. *Готуючи цей приклад, ми спершу передавали значення через `SpringApplicationBuilder.properties("duty.sms.enabled=false")` — і SMS-бін уперто створювався. Причина в тому, що такі властивості мають найнижчий пріоритет і `true` з `application.yml` їх перекриває. Детальніше про порядок джерел конфігурації — у наступному питанні.*

Варіанти для самостійної роботи

Завдання. Оголосити в проєкті свого варіанта бін з умовою, вказаною в таблиці. Продемонструвати два запуски: коли умова виконується і коли ні. Пояснити, яку роль у прикладі відіграє `@ConditionalOnMissingBean`.

№	Бін	Умова створення
1	`TelegramNotifier`	`duty.telegram.enabled=true`
2	`FuelPriceCache`	властивість `duty.fuel.cache-ttl` задана
3	`DemoDataLoader`	активний профіль `dev` (`@Profile`)
4	`Clock` за замовчуванням	немає іншого біна `Clock` (`@ConditionalOnMissingBean`)
5	`PdfReportExporter`	у classpath є клас `org.apache.pdfbox.pdmodel.PDDocument`

Питання 2.3. application.yml, профілі, @Value і @ConfigurationProperties

Теоретична частина

Налаштування, які змінюються між середовищами (адреси, паролі, ліміти, номери телефонів), не повинні жити в коді. Це один з принципів методології «Дванадцятифакторного застосунку» [24]. Spring Boot читає налаштування з багатьох джерел і об'єднує їх у єдине середовище `Environment` [25].

Пріоритет джерел, від найнижчого до найвищого (спрощено): значення за замовчуванням із коду; `application.yml` або `application.properties`; профільні файли `application-{профіль}.yml`; змінні середовища ОС; системні властивості JVM (`-D...`); аргументи командного рядка (`--...`). Значення з джерела з вищим пріоритетом перекриває нижче. Тому в контейнері Docker паролі зазвичай передають змінними середовища, а файл у репозиторії містить лише безпечні значення для розробки.

Формати `.properties` і `.yml` рівноправні. YAML зручніший для вкладених структур і списків, але вимогливий до відступів: пробіл замість табуляції, і завжди однакова кількість.

Профіль — це іменований набір налаштувань. Активують його властивістю `spring.profiles.active=dev` або аргументом `--spring.profiles.active=dev`. Біни теж можна прив'язати до профілю анотацією `@Profile("dev")`.

Прочитати значення в коді можна двома способами. `@Value("${duty.call-limit}")` на полі чи параметрі конструктора — швидко, але розпорошує налаштування по класах і не перевіряє типи до запуску. `@ConfigurationProperties(prefix = "duty")` прив'язує цілу групу властивостей до одного класу, найзручніше — до запису (record). Spring Boot сам перетворює рядки на числа, `LocalTime`, `Duration`, списки, а назви `unit-name`, `unitName` і `UNIT_NAME` вважає однаковими (relaxed binding). Для всього, що більше за одну-дві властивості, рекомендовано саме другий спосіб [25; 14].

Приклад задачі

Умова. Описати налаштування чергової частини записом `DutyProperties` з префіксом `duty`: назва підрозділу, ліміт викликів на зміну, час початку зміни, список телефонів чергових. Підготувати базовий `application.yml` і профілі `dev` та `prod`. Під час старту вивести поточні налаштування і активні профілі.

Запустимо приклад двічі: без профілю і з `--spring.profiles.active=dev`.

Лістинг 2.3. Реалізація з покроковим розбором

Файл `application.yml` базові налаштування

```
duty:
  unit-name: "Чергова частина ЛДУБЖД"
  call-limit: 20
  shift-start: "08:00"
  dispatcher-phones:
    - "+380 32 000-01-01"
  sms:
    enabled: true
```

Власні властивості застосунку зібрано під префіксом `duty`. Списки в YAML записують через дефіс.

Файл `application-dev.yml` профіль для розробки

```
duty:
  unit-name: "Навчальна чергова частина (dev)"
  call-limit: 5
  sms:
    enabled: false
```

Профільний файл містить лише відмінності. На навчальному стенді ліміт менший, а справжні SMS вимкнено — нікому не треба отримувати повідомлення про навчальні пожежі.

Файл `DutyProperties.java` типізовані налаштування

```
@ConfigurationProperties(prefix = "duty")
public record DutyProperties(
        String unitName,
        @DefaultValue("20") int callLimit,
        @DefaultValue("08:00") LocalTime shiftStart,
        List<String> dispatcherPhones) {
```

Кожен компонент запису відповідає властивості: `unit-name` → `unitName`. `@DefaultValue` діє, якщо властивість не задана ніде. Рядок `"08:00"` Boot сам перетворить на `LocalTime`.

Файл `DutyConfig.java` реєстрація класу властивостей

```
@EnableConfigurationProperties(DutyProperties.class)
```

Без цієї анотації (або `@ConfigurationPropertiesScan`) запис залишиться звичайним класом, а не біном.

Файл `StartupReport.java` звіт під час старту

```
@Component
public class StartupReport implements ApplicationRunner {
    private final DutyProperties props;
    private final Environment env;
```

`ApplicationRunner` — бін, метод `run` якого Boot викликає один раз, одразу після старту контексту. Зручне місце для стартових перевірок і звітів.

```
@Override
public void run(ApplicationArguments args) {
    System.out.println("Підрозділ: " + props.unitName());
    System.out.println("Профілі: "
            + Arrays.toString(env.getActiveProfiles()));
    System.out.println("Ліміт викликів на зміну: "
            + props.callLimit());
```

Налаштування беремо з типізованого запису, а активні профілі — з `Environment`. Решта рядків методу виводить час початку зміни і телефони.

Вивід програми без профілю, потім з --spring.profiles.active=dev

```
Підрозділ: Чергова частина ЛДУБЖД
Профілі: []
Ліміт викликів на зміну: 20
Початок зміни: 08:00
Телефони: [+380 32 000-01-01]

Підрозділ: Навчальна чергова частина (dev)
Профілі: [dev]
Ліміт викликів на зміну: 5
Початок зміни: 08:00
Телефони: [+380 32 000-01-01]
```

Порада. *Спробуйте запустити застосунок з профілем `dev`, але додати аргумент `--duty.call-limit=50`. Ліміт стане 50: аргумент командного рядка перекриває і базовий, і профільний файл. Це найшвидший спосіб перевірити, чи ви правильно розумієте порядок джерел конфігурації.*

Варіанти для самостійної роботи

Завдання. Описати налаштування підсистеми свого варіанта записом з `@ConfigurationProperties`, підготувати `application.yml` і два профілі. Вивести налаштування під час старту через `ApplicationRunner` і продемонструвати перекриття значення аргументом командного рядка.

№	Префікс	Властивості	Профілі
1	`fleet`	норма пального, інтервал ТО (`Duration`), список марок	`dev`, `prod`
2	`cadets`	кількість у наряді, початок наряду, список взводів	`dev`, `exam`
3	`hydrants`	інтервал перевірки, мінімальний тиск, райони міста	`dev`, `winter`
4	`training`	тривалість заняття, макс. група, полігони	`dev`, `summer`
5	`reports`	каталог для звітів, формат, адреси для розсилки	`dev`, `prod`

Питання 2.4. Структура проєкту, Maven і Spring Initializr

Теоретична частина

Новий проєкт Spring Boot найпростіше згенерувати на сайті Spring Initializr [26] або безпосередньо в IntelliJ IDEA, яка використовує той самий сервіс. Обираємо систему збирання, версії Java і Boot, координати (`groupId`, `artifactId`) і потрібні стартери — і отримуємо архів з готовою структурою.

Структура стандартна для Maven: `src/main/java` — код, `src/main/resources` — налаштування, шаблони, статичні файли, `src/test/java` — тести, `pom.xml` — опис проєкту [27]. Нічого нового порівняно з першою книжкою тут немає, однак Initializr додає ще обгортку Maven Wrapper (`mvnw`): з нею проєкт збирається навіть на машині, де Maven не встановлено.

Пакети всередині застосунку можна організувати двома способами. За шарами: `controller`, `service`, `repository`. Або за функціональними областями (package by feature): `call`, `fleet`, `notify`. Ми обрали другий. Коли проєкт росте, все, що стосується викликів, лежить поруч, і менше спокуси робити класи публічними «на всяк випадок».

Основні команди Maven, які варто знати напам'ять:

`mvn spring-boot:run` — зібрати і запустити застосунок;

`mvn test` — запустити тести;

`mvn package` — зібрати виконуваний jar у каталозі `target`;

`mvn dependency:tree` — показати дерево залежностей, зокрема транзитивних.

Плагін `spring-boot-maven-plugin` перепаковує звичайний jar у так званий fat jar: усередину вкладено всі залежності, а спеціальний завантажувач запускає застосунок. Саме тому `java -jar` працює без жодного classpath.

Приклад задачі

Умова. Показати поточну структуру проєкту чергової частини після першого і другого розділів, частину `pom.xml`, що відповідає за збирання, і послідовність команд, щоб зібрати та запустити застосунок з профілем `dev`.

Лістинг 2.4. Реалізація з покроковим розбором

Структура проєкту після розділу 2

```
duty-app/
├── pom.xml
└── src/main/
    ├── java/ua/edu/ldubgd/duty/
    │   ├── DutyApplication.java      головний клас
    │   ├── call/                     виклики та їх прийом
    │   ├── config/                   конфігурація, властивості
    │   ├── fleet/                    автопарк
    │   ├── health/                   перевірки стану
    │   ├── notify/                   сповіщення
    │   └── demo/                     навчальні точки запуску
    └── resources/
        ├── application.yml
        ├── application-dev.yml
        └── application-prod.yml
```

Файл `pom.xml` плагін збирання

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
```

Версію плагіна знову не вказуємо — її задає батьківський POM. Плагін додає цілі `spring-boot:run`, `spring-boot:build-image` і перепаковує jar.

```
<properties>
    <java.version>25</java.version>
</properties>
```

Властивість `java.version` батьківський POM передає компілятору як `--release 25`.

Термінал збирання і запуск

```
mvn package
java -jar target/duty-app-0.1.0.jar --spring.profiles.active=dev
mvn dependency:tree -Dincludes=org.apache.tomcat.embed
```

Зверніть увагу. *Каталог `target` і файли IDE (`.idea`, `*.iml`) не мають потрапляти до репозиторію. Initializr створює `.gitignore` автоматично; якщо проєкт заводили вручну — додайте його самі, перш ніж робити перший коміт.*

Варіанти для самостійної роботи

Завдання. Згенерувати через Spring Initializr проєкт підсистеми свого варіанта з указаною організацією пакетів. Зібрати fat jar, запустити його з профілем і за допомогою `mvn dependency:tree` знайти, звідки в проєкт потрапила вказана бібліотека.

№	Підсистема	Організація пакетів	Знайти в дереві
1	облік рятувальників	за функціями	`jackson-databind`
2	автопарк	за шарами	`tomcat-embed-core`
3	наряди курсантів	за функціями	`logback-classic`
4	склад спорядження	за шарами	`snakeyaml`
5	карта гідрантів	за функціями	`micrometer-core`

Питання 2.5. Логування в Spring Boot

Теоретична частина

У першій книжці ми розглядали журналювання загалом: рівні, фасад SLF4J і реалізації [1; 28]. Spring Boot нічого не змінює в цих принципах, лише налаштовує все заздалегідь. За замовчуванням використовується Logback через SLF4J, повідомлення виводяться в консоль з датою, рівнем, ідентифікатором процесу, назвою потоку й логера.

У коді логер отримують через `LoggerFactory.getLogger(Клас.class)` і зберігають у статичному незмінному полі. Повідомлення пишуть з плейсхолдерами `{}`, а не конкатенацією: `log.debug("Прийом виклику: {}", address)`. Якщо рівень DEBUG вимкнено, рядок навіть не буде зібрано — це і швидше, і чистіше.

Керують логуванням через звичайні властивості [21]:

`logging.level.<пакет>=DEBUG` — рівень для пакета чи класу;

`logging.file.name=logs/duty-app.log` — дублювати вивід у файл (з ротацією за розміром);

`logging.pattern.console` — власний шаблон рядка для консолі.

Оскільки це звичайні властивості, рівень можна змінити профілем, змінною середовища або аргументом командного рядка без перезбирання. А з Actuator — навіть на працюючому застосунку, через кінцеву точку `/actuator/loggers`.

Як обирати рівень? Наша порада з першої книжки досі актуальна. ERROR — щось зламалося і потребує втручання. WARN — нештатна, але не критична ситуація. INFO — ключові бізнес-події, які цікаві черговому адміністраторові. DEBUG — подробиці для розробника. На рівні INFO в робочому застосунку має бути видно, що відбувається, але без шуму.

Приклад задачі

Умова. Замінити в службі прийому викликів і журнальному сповіщувачі `System.out` на SLF4J. Писати DEBUG перед прийомом виклику, INFO — після реєстрації, WARN — коли кількість викликів досягла ліміту зміни. У профілі `dev` увімкнути DEBUG для пакета застосунку і задати компактний шаблон рядка. Зареєструвати шість викликів при ліміті п'ять.

Лістинг 2.5. Реалізація з покроковим розбором

Файл `DispatchService.java` логер у службі

```
private static final Logger log =
        LoggerFactory.getLogger(DispatchService.class);
```

Один логер на клас, `static final`. Його назва — повне ім'я класу, тож рівнями можна керувати і для окремого класу, і для всього пакета.

```
public FireCall accept(String address, CallType type) {
    log.debug("Прийом виклику: {} {}", type, address);
    int active = registry.findAll().size();
    if (active >= callLimit) {
        log.warn("Перевищено ліміт {} викликів", callLimit);
    }
```

Ліміт `callLimit` служба отримує з `DutyProperties` через конструктор. Виклик не відхиляємо — пожежу ніхто не скасує, — але черговий має побачити попередження.

```
    FireCall call = registry.register(address, type);
    notifier.send("Виклик №" + call.id() + ": " + type
            + ", " + address);
    log.info("Зареєстровано виклик №{}", call.id());
    return call;
}
```

INFO — головна бізнес-подія: виклик зареєстровано.

Файл `application-dev.yml` рівень і шаблон для розробки

```
logging:
  level:
    ua.edu.ldubgd.duty: DEBUG
  pattern:
    console: "%d{HH:mm:ss} %-5level %logger{0} - %msg%n"
```

`%logger{0}` залишає від назви логера лише ім'я класу. Для розробки цього досить, а рядки вміщуються в екран.

Вивід програми останні рядки виводу

```
11:16:00 DEBUG DispatchService - Прийом виклику: RESCUE вул. Стрийська, 5
11:16:00 INFO  LogNotifier - [ЖУРНАЛ] Виклик №5: RESCUE, вул. Стрийська, 5
11:16:00 INFO  DispatchService - Зареєстровано виклик №5
11:16:00 DEBUG DispatchService - Прийом виклику: RESCUE вул. Стрийська, 6
11:16:00 WARN  DispatchService - Перевищено ліміт 5 викликів
11:16:00 INFO  LogNotifier - [ЖУРНАЛ] Виклик №6: RESCUE, вул. Стрийська, 6
11:16:00 INFO  DispatchService - Зареєстровано виклик №6
```

Порада. *Не записуйте в лог персональні дані — прізвища, номери телефонів заявників, паспортні дані. Журнали часто доступні ширшому колу людей, ніж база даних, а їхнє зберігання регулюється законодавством про захист персональних даних [29]. Адреса виклику в нашому прикладі — допустимий компроміс, номер телефону заявника — вже ні.*

Варіанти для самостійної роботи

Завдання. Додати SLF4J-логування до служби свого варіанта з усіма вказаними рівнями. Налаштувати рівні й шаблон у профілі `dev` і вивід у файл у профілі `prod`. Продемонструвати, що зміна рівня аргументом `--logging.level...` не потребує перезбирання.

№	Служба	DEBUG	INFO	WARN
1	`ShiftService`	склад зміни	зміну сформовано	у зміні менше 4 осіб
2	`FuelLog`	кожна заправка	підсумок за день	перевитрата понад норму
3	`CadetDutyPlanner`	розподіл нарядів	графік сформовано	курсант має понад 3 наряди
4	`HydrantChecker`	результат вимірювання	перевірку завершено	тиск нижчий за норму
5	`EquipmentStock`	кожна видача	залишки на кінець дня	позиція нижче мінімуму

Питання 2.6. Spring Boot Actuator: health і metrics

Теоретична частина

Застосунок, який працює на сервері, має відповідати на просте питання: «ти живий?». Цю роль виконує Spring Boot Actuator — модуль, що додає службові кінцеві точки (endpoints) для моніторингу й керування [30].

Найважливіші з них:

`/actuator/health` — стан застосунку і його залежностей (база даних, диск, черги повідомлень);

`/actuator/info` — інформація про застосунок (версія, коміт Git);

`/actuator/metrics` — метрики: пам’ять, потоки, кількість і тривалість HTTP-запитів;

`/actuator/loggers` — перегляд і зміна рівнів логування на льоту.

За замовчуванням через HTTP відкрито лише `health`. Решту треба явно перелічити у властивості `management.endpoints.web.exposure.include`. Це не випадковість: деякі точки (наприклад, `env` або `heapdump`) розкривають чутливу інформацію, і відкривати їх назовні без захисту не можна [31].

Стан `health` складається з окремих індикаторів — бінів, що реалізують `HealthIndicator`. Boot сам додає індикатори для всього, що знайде: дискового простору, бази даних тощо. Загальний статус — UP, лише якщо всі індикатори в стані UP. Власний індикатор написати нескладно, і це хороша практика: стан застосунку визначається не лише тим, чи працює JVM, а й тим, чи може він виконувати свою роботу.

Метрики Actuator збирає бібліотека Micrometer [32]. Вона є фасадом, як SLF4J для логів: одні й ті самі метрики можна віддавати в Prometheus, Grafana чи хмарний моніторинг.

У середовищах на кшталт Kubernetes Actuator надає також групи `liveness` і `readiness`: перша відповідає «процес не завис», друга — «готовий приймати запити». До цієї теми ми повернемося в розділі 10.

Приклад задачі

Умова. Відкрити кінцеві точки `health`, `info` і `metrics`. Написати індикатор стану `FleetHealthIndicator`: застосунок вважається працездатним, лише якщо в частині є хоча б один справний автомобіль. Запустити застосунок з вебсервером і перевірити стан командою `curl`.

Лістинг 2.6. Реалізація з покроковим розбором

Файл `application.yml` відкриття кінцевих точок

```
management:
  endpoints:
    web:
      exposure:
        include: health, info, metrics
  endpoint:
    health:
      show-details: always
```

`show-details: always` показує деталі кожного індикатора. Для навчального стенду це зручно; у робочому середовищі деталі зазвичай показують лише автентифікованим користувачам (`when-authorized`).

Файл `FleetHealthIndicator.java` власний індикатор

```
@Component
public class FleetHealthIndicator implements HealthIndicator {
    private final VehicleDirectory directory;
```

Ім’я індикатора утворюється з імені класу без суфікса `HealthIndicator`: у звіті він з’явиться як `fleet`. У Spring Boot 4 інтерфейс лежить у пакеті `org.springframework.boot.health.contributor`.

```
@Override
public Health health() {
    int ready = directory.available().size();
    Health.Builder status =
            ready > 0 ? Health.up() : Health.down();
    return status.withDetail("readyTrucks", ready).build();
}
```

Якщо всі машини на ремонті, статус `DOWN` — і разом з ним весь застосунок. Деталь `readyTrucks` допоможе черговому зрозуміти причину.

Термінал перевірка стану

```
mvn spring-boot:run
curl -s localhost:8080/actuator/health/fleet
curl -s localhost:8080/actuator/metrics/jvm.threads.live
```

Вивід програми відповіді сервера (скорочено)

```
{"details":{"readyTrucks":1},"status":"UP"}
{"name":"jvm.threads.live","baseUnit":"threads",
 "measurements":[{"statistic":"VALUE","value":21.0}], ...}
```

Зверніть увагу. *Запит до `/actuator/health` без імені індикатора поверне всі компоненти одразу: `diskSpace`, `fleet`, `ping`, `livenessState` та інші. Змініть у `VehicleDirectory` статус автоцистерни на `false`, перезапустіть застосунок — і загальний статус стане `DOWN`, а HTTP-код відповіді зміниться з 200 на 503.*

Варіанти для самостійної роботи

Завдання. Додати до проєкту свого варіанта Actuator і власний `HealthIndicator` з указаною умовою. Відкрити потрібні кінцеві точки і продемонструвати стани UP і DOWN.

№	Індикатор	Стан DOWN, якщо…	Додатково відкрити
1	`crew`	у поточній зміні менше 4 рятувальників	`loggers`
2	`hydrants`	понад 20 % гідрантів несправні	`info`
3	`cadetDuty`	на завтра не призначено наряд	`metrics`
4	`storage`	на диску менше 1 ГБ для звітів	`loggers`
5	`smsGateway`	шлюз SMS не відповідав понад 5 хв	`info`

ЧАСТИНА 2. ВЕБРІВЕНЬ

Розділ 3. Spring MVC і REST API

Питання 3.1. DispatcherServlet і шлях запиту

Теоретична частина

У першій книжці ми писали сервлети вручну: клас успадковував `HttpServlet`, перевизначав `doGet` і `doPost`, а відповідність URL і класу задавалася анотацією `@WebServlet` [1; 33]. Spring MVC побудовано на тій самій технології, але сервлет у застосунку, по суті, один — `DispatcherServlet`. Він реалізує патерн «Фронт-контролер» (Front Controller): приймає всі запити й розподіляє їх між звичайними методами Java-класів [34; 35].

Шлях запиту через Spring MVC варто вміти розповісти на співбесіді по кроках:

1) Tomcat приймає HTTP-запит і передає його ланцюжку фільтрів сервлетів (тут працюватиме Spring Security, розділ 7), а далі — `DispatcherServlet`;

2) `HandlerMapping` знаходить обробник — метод контролера, чиї анотації (`@GetMapping("/api/calls/{id}")`) відповідають методу і шляху запиту;

3) виконуються методи `preHandle` перехоплювачів (`HandlerInterceptor`);

4) `HandlerAdapter` готує аргументи методу: витягує змінні шляху, параметри, тіло запиту (через `HttpMessageConverter`, для JSON — Jackson), перевіряє їх і викликає метод;

5) результат методу перетворюється на відповідь: для REST — знову через `HttpMessageConverter` у JSON, для сторінок — через `ViewResolver` у HTML (розділ 4);

6) якщо виник виняток, його обробляє `HandlerExceptionResolver` (питання 3.7);

7) виконуються `afterCompletion` перехоплювачів, і відповідь іде клієнтові.

Фільтр і перехоплювач часто плутають. Фільтр належить до специфікації сервлетів і бачить лише «сирий» запит. Перехоплювач — частина Spring MVC: він уже знає, який метод контролера буде викликано, і працює лише для запитів, що дійшли до `DispatcherServlet`.

Протокол HTTP ми детально розглядали в першій книжці, тому тут лише нагадаємо: метод (GET, POST, PUT, DELETE), шлях, заголовки й тіло — це все, з чим працює контролер [36].

Приклад задачі

Умова. Написати перехоплювач `RequestTimingInterceptor`, який для кожного запиту до `/api/**` записує в лог метод, шлях, знайдений обробник, а після завершення — код відповіді і час обробки. Зареєструвати його в конфігурації Spring MVC і перевірити на простому запиті до `/api/status`.

Сам контролер `/api/status` розглянемо в наступному питанні. Тут нас цікавить лише те, що відбувається навколо нього.

Лістинг 3.1. Реалізація з покроковим розбором

Файл `RequestTimingInterceptor.java` перехоплювач

```
@Override
public boolean preHandle(HttpServletRequest req,
                         HttpServletResponse res,
                         Object handler) {
    req.setAttribute("start", System.nanoTime());
    log.info("-> {} {} обробник: {}", req.getMethod(),
            req.getRequestURI(), handler);
    return true;
}
```

Параметр `handler` — уже знайдений метод контролера: `HandlerMapping` спрацював раніше. Повернення `false` зупинило б обробку запиту, і до контролера він би не дійшов.

```
@Override
public void afterCompletion(HttpServletRequest req,
                            HttpServletResponse res,
                            Object handler, Exception ex) {
    long start = (long) req.getAttribute("start");
    long ms = (System.nanoTime() - start) / 1_000_000;
    log.info("<- {} {} мс", res.getStatus(), ms);
}
```

`afterCompletion` викликається після того, як відповідь сформовано, навіть якщо в контролері виник виняток.

Файл `WebConfig.java` реєстрація перехоплювача

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(timing).addPathPatterns("/api/**");
}
```

`WebMvcConfigurer` — стандартний спосіб доповнити налаштування Spring MVC, не відключаючи автоконфігурацію Boot. Сторінки Actuator і Swagger перехоплювач не чіпатиме.

Термінал запит і лог сервера

```
curl -s localhost:8080/api/status
```

Вивід програми результат запуску

```
Навчальна чергова частина (dev): чергування триває

-> GET /api/status обробник: ua.edu.ldubgd.duty.web.StatusController#status()
<- 200 11 мс
```

Порада. *Запустіть приклад у налагоджувачі IntelliJ IDEA і поставте точку зупинки в методі контролера. У стеку викликів ви побачите весь шлях: `DispatcherServlet.doDispatch`, `HandlerAdapter`, ваш метод. Десять хвилин такого «розглядання» замінюють кілька сторінок теорії.*

Варіанти для самостійної роботи

Завдання. Написати перехоплювач або фільтр за умовою свого варіанта, зареєструвати його для вказаних шляхів і продемонструвати роботу запитами `curl`.

№	Компонент	Що робить
1	`HandlerInterceptor`	відхиляє запити до `/api/**` без заголовка `X-Duty-Shift` (код 400)
2	`HandlerInterceptor`	рахує кількість запитів до кожного методу контролера і виводить у лог
3	`Filter` (`OncePerRequestFilter`)	додає до кожної відповіді заголовок `X-Unit: LDUBGD`
4	`HandlerInterceptor`	пише в лог запити, що оброблялися довше за 200 мс, на рівні WARN
5	`Filter`	генерує ідентифікатор запиту і додає його в заголовок відповіді `X-Request-Id`

Питання 3.2. Різниця між @Controller і @RestController; @RequestMapping

Теоретична частина

`@Controller` — стереотипна анотація для класів вебрівня. За замовчуванням значення, яке повертає метод такого контролера, Spring вважає ім'ям подання (view): рядок `"calls"` означає «знайди шаблон `calls.html` і відрендери його». Щоб повернути дані безпосередньо в тіло відповіді, метод позначають `@ResponseBody`.

`@RestController` — це просто `@Controller` + `@ResponseBody` на всіх методах класу [35]. Для REST API, де всі методи повертають JSON, так зручніше. Тому відповідь на типове питання співбесіди коротка: різниця лише в тому, як трактується значення, що повертається; `@RestController` — скорочення для випадку, коли всі методи повертають дані, а не подання.

`@RequestMapping` зіставляє запит з методом. Її можна поставити на клас (спільний префікс шляху) і на метод. Для конкретних HTTP-методів є скорочення: `@GetMapping`, `@PostMapping`, `@PutMapping`, `@PatchMapping`, `@DeleteMapping`. Крім шляху, можна обмежити тип вмісту (`consumes`, `produces`), наявність заголовків і параметрів.

Що відбувається з об'єктом, який повертає `@ResponseBody`-метод? Spring обирає `HttpMessageConverter` залежно від типу об'єкта й заголовка `Accept` запиту. Для рядка — `StringHttpMessageConverter`, для звичайного об'єкта — конвертер Jackson, що серіалізує його в JSON. Записи (records) Jackson серіалізує без жодних додаткових анотацій.

Приклад задачі

Умова. Створити два контролери. `StatusController` з анотацією `@Controller` повертає простий текстовий статус чергування через `@ResponseBody`. `CallController` з анотацією `@RestController` має спільний префікс `/api/calls` і метод, що повертає один виклик за ідентифікатором у форматі JSON.

Лістинг 3.2. Реалізація з покроковим розбором

Файл `StatusController.java` звичайний контролер

```
@Controller
public class StatusController {
    private final DutyProperties props;
```

Контролер — такий самий бін, як служби, тож налаштування впроваджуються через конструктор.

```
@GetMapping("/api/status")
@ResponseBody
public String status() {
    return props.unitName() + ": чергування триває";
}
```

Без `@ResponseBody` Spring шукав би шаблон з назвою «Навчальна чергова частина…» і повернув би помилку 500.

Файл `CallController.java` REST-контролер

```
@RestController
@RequestMapping("/api/calls")
@Tag(name = "Виклики", description = "Облік викликів")
public class CallController {
    private final DispatchService service;
    private final CallRegistry registry;
    private final CallMapper mapper;
```

Префікс `/api/calls` спільний для всіх методів класу. Анотацію `@Tag` використовує генератор документації (питання 3.9).

```
@GetMapping("/{id}")
public CallResponse get(@PathVariable long id) {
    return mapper.toResponse(service.find(id));
}
```

Метод повертає звичайний об’єкт; Jackson перетворить його на JSON. Як працює `@PathVariable` і чому ми повертаємо `CallResponse`, а не `FireCall`, — у наступних питаннях.

Термінал запити

```
curl -s localhost:8080/api/status
curl -s localhost:8080/api/calls/1
```

Вивід програми результат запуску

```
Навчальна чергова частина (dev): чергування триває
{"id":1,"address":"вул. Клепарівська, 35","type":"FIRE","status":"NEW",
 "receivedAt":"2026-10-23T11:27:42"}
```

Зверніть увагу. *Перед запитом `/api/calls/1` треба зареєструвати хоча б один виклик (див. питання 3.4), інакше у відповідь прийде помилка 404. Дані поки що зберігаються в пам’яті й зникають після перезапуску — базу даних підключимо в розділі 5.*

Варіанти для самостійної роботи

Завдання. Створити REST-контролер для ресурсу свого варіанта зі спільним префіксом шляху і методом отримання одного об'єкта за ідентифікатором, а також звичайний `@Controller` з одним текстовим методом `@ResponseBody`. Дані зберігати в пам'яті.

№	Ресурс	Префікс	Текстовий метод
1	рятувальники	`/api/rescuers`	кількість рятувальників на зміні
2	пожежні автомобілі	`/api/trucks`	кількість справних машин
3	курсанти	`/api/cadets`	хто сьогодні черговий
4	гідранти	`/api/hydrants`	дата останньої перевірки
5	спорядження	`/api/equipment`	кількість позицій на складі

Питання 3.3. @PathVariable, @RequestParam, @RequestBody

Теоретична частина

Дані з HTTP-запиту потрапляють у параметри методу контролера кількома шляхами [35]. Три найважливіші анотації варто розрізняти чітко.

`@PathVariable` бере значення із самого шляху: для шаблону `/api/calls/{id}` і запиту `/api/calls/7` параметр `id` отримає 7. Змінні шляху ідентифікують ресурс, тому вони зазвичай обов'язкові.

`@RequestParam` бере значення з рядка запиту після знака питання: `/api/calls?type=FIRE&page=2`. Параметри запиту підходять для фільтрів, сортування, пагінації. Їх можна зробити необов'язковими (`required = false`) або задати значення за замовчуванням (`defaultValue`).

`@RequestBody` перетворює тіло запиту на об'єкт за допомогою `HttpMessageConverter`. Для JSON це Jackson. Тіло мають запити POST, PUT і PATCH; у GET тіла бути не повинно.

Перетворення типів Spring виконує сам: рядок `"7"` стає `long`, рядок `"FIRE"` — константою переліку `CallType`. Якщо перетворення неможливе (`/api/calls/abc` або `type=FLOOD`), клієнт отримає відповідь 400 Bad Request.

Є ще `@RequestHeader` (значення заголовка), `@CookieValue` і `@ModelAttribute` (дані HTML-форми, розділ 4). Імена параметрів Spring бере з байткоду, тому з Maven-збиранням Boot писати `@PathVariable("id")` не обов'язково.

Типова помилка: передавати фільтр як змінну шляху (`/api/calls/type/FIRE`) або ідентифікатор як параметр (`/api/calls?id=7`). Працюватиме і так, але колеги на код-рев'ю зауважать: шлях описує, *що* ми запитуємо, параметри — *як* саме.

Приклад задачі

Умова. Додати до `CallController` метод отримання списку викликів з необов'язковим фільтром за типом, параметрами сторінки і сортуванням, а також метод реєстрації виклику, який приймає дані в тілі запиту у форматі JSON.

Пагінацію детальніше розглянемо в питанні 3.8, а код відповіді 201 — у питанні 3.4.

Лістинг 3.3. Реалізація з покроковим розбором

Файл `CallController.java` параметри запиту і тіло

```
@GetMapping
public PageResponse<CallResponse> list(
        @RequestParam(required = false) CallType type,
        @RequestParam(defaultValue = "0") @Min(0) int page,
        @RequestParam(defaultValue = "10")
        @Min(1) @Max(100) int size,
        @RequestParam(defaultValue = "id") String sort) {
```

Фільтр за типом необов'язковий: якщо його немає, `type` дорівнює `null`. Решта параметрів мають значення за замовчуванням. Рядок `"FIRE"` Spring сам перетворить на `CallType.FIRE`.

```
@PostMapping
@Operation(summary = "Зареєструвати новий виклик")
public ResponseEntity<CallResponse> create(
        @Valid @RequestBody CallRequest request) {
```

Тіло запиту Jackson перетворює на запис `CallRequest`. Анотацію `@Valid` пояснимо в питанні 3.6.

Файл `CallRequest.java` дані нового виклику

```
public record CallRequest(
        @NotBlank(message = "адресу не вказано")
        @Size(max = 200, message = "адреса задовга")
        String address,
```

Імена компонентів запису збігаються з ключами JSON: `{"address": "...", "type": "FIRE"}`. Анотації валідації поки що пропустіть.

Термінал реєстрація і фільтр

```
curl -s -H "Content-Type: application/json" \
     -d '{"address":"вул. Клепарівська, 35","type":"FIRE"}' \
     localhost:8080/api/calls
curl -s "localhost:8080/api/calls?type=FIRE&size=1&sort=-id"
```

Вивід програми відповідь на другий запит (три виклики, з них два FIRE)

```
{"content":[{"id":3,"address":"просп. Чорновола, 59","type":"FIRE",
  "status":"NEW","receivedAt":"2026-10-23T11:26:47"}],
 "page":0,"size":1,"totalElements":2,"totalPages":2}
```

З досвіду викладачів. *Типова помилка — забутий заголовок `Content-Type: application/json`. Spring у відповідь чесно повертає 415 Unsupported Media Type, а розробник шукає помилку в коді контролера. Ми й самі на цьому спіткнулися, коли готували приклад: оболонка zsh не розбила змінну із заголовком на слова, і `curl` надсилав запит як звичайну форму.*

Варіанти для самостійної роботи

Завдання. Для контролера з варіанта питання 3.2 додати метод отримання списку з фільтрами через `@RequestParam` і метод створення об'єкта через `@RequestBody`. Перевірити некоректні значення параметрів і пояснити отримані коди відповіді.

№	Ресурс	Фільтри (`@RequestParam`)	Поля тіла запиту
1	рятувальники	звання, мін. стаж	ПІБ, звання, стаж
2	автомобілі	модель, справність	номер, модель, рік
3	курсанти	взвод, курс	ПІБ, взвод, курс
4	гідранти	район, справність	адреса, тип, тиск
5	спорядження	категорія, мін. залишок	назва, категорія, кількість

Питання 3.4. ResponseEntity і коди стану HTTP

Теоретична частина

Код стану (status code) — перше, що бачить клієнт API. Коди поділено на класи [36]: 2xx — успіх, 3xx — перенаправлення, 4xx — помилка клієнта, 5xx — помилка сервера. Для REST API Junior-розробник має впевнено знати хоча б такі:

200 OK — успіх, у тілі є дані;

201 Created — ресурс створено, заголовок `Location` містить його адресу;

204 No Content — успіх, тіла немає (часто після DELETE);

400 Bad Request — некоректні дані запиту;

401 Unauthorized і 403 Forbidden — не автентифіковано і заборонено (розділ 7);

404 Not Found — ресурс не існує;

409 Conflict — запит суперечить поточному стану ресурсу;

500 Internal Server Error — необроблений виняток на сервері.

Якщо метод контролера повертає звичайний об'єкт, Spring відповідає кодом 200. Щоб керувати кодом і заголовками, повертають `ResponseEntity<T>` — обгортку над тілом, кодом і заголовками. Для типових випадків є зручні фабричні методи: `ResponseEntity.ok(body)`, `created(uri)`, `noContent()`, `notFound()`. Альтернатива — анотація `@ResponseStatus(HttpStatus.CREATED)` на методі, але тоді заголовок `Location` додати складніше.

Ще одне питання з практики: що повернути, якщо список порожній? 200 з порожнім масивом, а не 404. Ресурс «список викликів» існує, просто він порожній.

Приклад задачі

Умова. Реалізувати в `CallController` правильні коди відповіді: реєстрація виклику повертає 201 Created із заголовком `Location`, видалення — 204 No Content для наявного виклику і 404 Not Found для відсутнього.

Лістинг 3.4. Реалізація з покроковим розбором

Файл `CallController.java` 201 і 204

```
    FireCall call = service.accept(request.address(),
            request.type(), request.callerPhone());
    URI location = ServletUriComponentsBuilder
            .fromCurrentRequest().path("/{id}")
            .buildAndExpand(call.id()).toUri();
    return ResponseEntity.created(location)
            .body(mapper.toResponse(call));
}
```

`ServletUriComponentsBuilder` бере адресу поточного запиту (`http://localhost:8080/api/calls`) і додає до неї ідентифікатор. Так `Location` залишиться правильним і за проксі, і на іншому порту.

```
@DeleteMapping("/{id}")
public ResponseEntity<Void> delete(@PathVariable long id) {
    return registry.delete(id)
            ? ResponseEntity.noContent().build()
            : ResponseEntity.notFound().build();
}
```

`ResponseEntity<Void>` — відповідь без тіла. Метод `delete` реєстру повертає `true`, якщо виклик справді було видалено.

Термінал перевірка кодів

```
curl -s -i -H "Content-Type: application/json" \
     -d '{"address":"вул. Клепарівська, 35","type":"FIRE"}' \
     localhost:8080/api/calls
curl -s -i -X DELETE localhost:8080/api/calls/2
curl -s -i -X DELETE localhost:8080/api/calls/2
```

Вивід програми заголовки відповідей (скорочено)

```
HTTP/1.1 201
Location: http://localhost:8080/api/calls/1
Content-Type: application/json

{"id":1,"address":"вул. Клепарівська, 35","type":"FIRE","status":"NEW",...}

HTTP/1.1 204
HTTP/1.1 404
```

Зверніть увагу. *Повторний DELETE того самого виклику повертає 404, але стан сервера після обох запитів однаковий: виклику немає. Тому DELETE вважається ідемпотентним, попри різні коди відповіді. Детальніше про ідемпотентність — у питанні 3.9.*

Варіанти для самостійної роботи

Завдання. Для контролера свого варіанта реалізувати повний набір кодів: 201 з `Location` під час створення, 200 під час отримання, 204 і 404 під час видалення. Перевірити кожен випадок командою `curl -i`.

№	Ресурс	Додатково
1	рятувальники	409, якщо рятувальник з таким табельним номером уже є
2	автомобілі	409, якщо автомобіль з таким номером уже зареєстровано
3	курсанти	204 замість 404 під час видалення відсутнього курсанта; обґрунтувати
4	гідранти	`@ResponseStatus(HttpStatus.CREATED)` замість `ResponseEntity`
5	спорядження	заголовок `X-Total-Count` у відповіді на отримання списку

Питання 3.5. DTO та маппінг; Java records як DTO

Теоретична частина

DTO (Data Transfer Object) — об’єкт, що описує дані, які проходять через межу системи: в API, з API, між сервісами [34]. Чому б не повертати клієнтові безпосередньо доменний об’єкт `FireCall` (а пізніше — сутність JPA)? Причин кілька, і на співбесіді добре назвати хоча б дві.

Безпека і приватність. Доменний об’єкт може містити поля, яких клієнт бачити не повинен: номер телефону заявника, службові позначки, а в сутностях — паролі чи внутрішні ідентифікатори. Наш `FireCall` має поле `callerPhone`, і випадково віддавати його кожному, хто відкрив журнал викликів, не можна [29].

Стабільність контракту. API — це обіцянка клієнтам (мобільному застосунку, табло, іншим сервісам). Якщо відповідь збігається з внутрішньою моделлю, будь-яке перейменування поля в коді ламає клієнтів.

Різні дані для різних операцій. Для створення виклику клієнт не передає ні `id`, ні статус, ні час — їх визначає сервер. Тому запит і відповідь — різні DTO.

Записи Java (records) ідеально підходять для DTO: вони незмінні, мають автоматичні `equals`, `hashCode` і `toString`, а Jackson уміє їх і серіалізувати, і десеріалізувати [13]. Перетворення «домен ↔ DTO» називають маппінгом. Для невеликих проєктів його пишуть вручну, для великих використовують генератори на кшталт MapStruct. Головне правило: маппінг живе на межі вебрівня, служби про DTO не знають.

Приклад задачі

Умова. Описати DTO для викликів: `CallRequest` (адреса, тип, необов’язковий телефон заявника) і `CallResponse` (ідентифікатор, адреса, тип, статус, час надходження — без телефону). Написати `CallMapper`, що перетворює доменний об’єкт на відповідь і заодно відкидає частки секунди в часі надходження.

Лістинг 3.5. Реалізація з покроковим розбором

Файл `FireCall.java` доменна модель

```
public record FireCall(long id, String address, CallType type,
                       LocalDateTime receivedAt,
                       CallStatus status, String callerPhone) {
```

Порівняно з розділом 1 у виклику з’явилися статус і телефон заявника. Метод `withStatus` (не показано) повертає копію з новим статусом — записи незмінні.

Файл `CallResponse.java` відповідь API

```
public record CallResponse(long id, String address,
                           CallType type, CallStatus status,
                           LocalDateTime receivedAt) {
}
```

Телефону заявника тут немає, і жодна зміна в `FireCall` не потрапить у відповідь непомітно: поле доведеться свідомо додати і сюди.

Файл `CallMapper.java` маппінг

```
@Component
public class CallMapper {
    public CallResponse toResponse(FireCall call) {
        LocalDateTime time = call.receivedAt()
                .truncatedTo(ChronoUnit.SECONDS);
        return new CallResponse(call.id(), call.address(),
                call.type(), call.status(), time);
    }
}
```

Маппер — бін вебрівня. Відкидаємо наносекунди: клієнтам вони не потрібні, а відповідь стає коротшою.

Вивід програми відповідь API: телефону заявника немає

```
{"id":1,"address":"вул. Клепарівська, 35","type":"FIRE","status":"NEW",
 "receivedAt":"2026-10-23T11:27:42"}
```

Порада. *Тимчасово змініть метод `get` контролера так, щоб він повертав `FireCall` замість `CallResponse`. У відповіді з'явиться `"callerPhone":"+380671234567"`. Саме так у реальних системах і стаються витоки персональних даних — не через злам, а через «тимчасово повернемо сутність, потім переробимо».*

Варіанти для самостійної роботи

Завдання. Описати DTO запиту і відповіді для ресурсу свого варіанта у вигляді записів і написати маппер. Поля зі стовпчика «Не віддавати» мають бути в доменній моделі, але не у відповіді.

№	Ресурс	Не віддавати клієнтові	Перетворення під час маппінгу
1	рятувальники	номер паспорта, домашня адреса	ПІБ → «Прізвище І. П.»
2	автомобілі	VIN-код	рік випуску → вік автомобіля
3	курсанти	номер телефону батьків	середній бал округлити до 0,1
4	гідранти	внутрішній код балансоутримувача	тиск у кПа → бар
5	спорядження	закупівельна ціна	кількість → «достатньо / мало»

Питання 3.6. Валідація: Bean Validation, @Valid

Теоретична частина

Правило, яке варто засвоїти раз і назавжди: дані, що прийшли ззовні, вважаються ненадійними, доки їх не перевірено. Перевірка на клієнті (у браузері чи мобільному застосунку) — це зручність для користувача, а не захист: запит завжди можна надіслати в обхід клієнта, хоча б тим самим `curl` [31].

Jakarta Bean Validation — стандарт, що описує обмеження анотаціями на полях і параметрах [37]. Реалізацію (Hibernate Validator) підключає стартер `spring-boot-starter-validation`. Найуживаніші обмеження:

`@NotNull`, `@NotBlank` (не `null` і не порожній рядок з пробілів), `@NotEmpty` (для колекцій);

`@Size(min, max)`, `@Min`, `@Max`, `@Positive`;

`@Pattern(regexp = ...)`, `@Email`;

`@Past`, `@Future` — для дат.

Щоб Spring перевірив тіло запиту, параметр позначають `@Valid`. Якщо обмеження порушено, метод контролера не викликається, а Spring кидає `MethodArgumentNotValidException`, і клієнт отримує 400. Обмеження на простих параметрах (`@RequestParam @Max(100) int size`) Spring Framework з версії 6.1 перевіряє сам, без додаткових анотацій на класі [35].

Повідомлення про помилки можна задати атрибутом `message`, а для багатомовних застосунків — винести у файл `ValidationMessages.properties`. Зверніть увагу: `@Pattern` і `@Size` вважають `null` коректним значенням. Тому необов'язковий телефон заявника ми перевіряємо лише на формат, а адресу — ще й на `@NotBlank`.

Перевірки, що залежать від стану системи («такий номер автомобіля вже існує», «статус не можна змінити»), до Bean Validation не належать. Їх виконує служба, кидаючи доменний виняток (питання 3.7).

Приклад задачі

Умова. Додати обмеження до запиту реєстрації виклику: адреса обов'язкова й не довша за 200 символів, тип обов'язковий, телефон заявника — необов'язковий, але у форматі `+380XXXXXXXXX`. Увімкнути перевірку в контролері й надіслати некоректний запит.

Лістинг 3.6. Реалізація з покроковим розбором

Файл `CallRequest.java` обмеження на полях запиту

```
public record CallRequest(
        @NotBlank(message = "адресу не вказано")
        @Size(max = 200, message = "адреса задовга")
        String address,

        @NotNull(message = "тип виклику обов'язковий")
        CallType type,

        @Pattern(regexp = "\\+380\\d{9}",
                 message = "формат +380XXXXXXXXX")
        String callerPhone) {
}
```

Анотації ставимо безпосередньо на компоненти запису. Подвійний зворотний слеш у регулярному виразі — особливість рядкових літералів Java: у самому виразі буде `\+380\d{9}`.

Файл `CallController.java` увімкнення перевірки

```
public ResponseEntity<CallResponse> create(
        @Valid @RequestBody CallRequest request) {
```

Без `@Valid` анотації в `CallRequest` нічого б не робили — порожня адреса спокійно потрапила б до журналу викликів.

Термінал некоректний запит

```
curl -s -H "Content-Type: application/json" \
     -d '{"address":"","callerPhone":"0671234567"}' \
     localhost:8080/api/calls
```

Вивід програми відповідь 400 (вигляд тіла налаштуємо в питанні 3.7)

```
{"detail":"Некоректні дані виклику","instance":"/api/calls","status":400,
 "title":"Помилка валідації",
 "errors":{"address":"адресу не вказано",
           "callerPhone":"формат +380XXXXXXXXX",
           "type":"тип виклику обов'язковий"}}
```

Зверніть увагу. *Spring перевіряє всі обмеження одразу й повертає всі помилки разом, а не першу знайдену. Для користувача це важливо: краще один раз побачити три зауваження, ніж тричі виправляти форму.*

Варіанти для самостійної роботи

Завдання. Додати обмеження Bean Validation до DTO запиту свого варіанта, увімкнути перевірку `@Valid` і продемонструвати відповідь на запит, що порушує кожне обмеження.

№	Ресурс	Обмеження
1	рятувальники	ПІБ 3–100 символів; звання з переліку; стаж 0–45
2	автомобілі	номер за шаблоном `AA 0000 AA`; рік 1990 — поточний
3	курсанти	ПІБ обов'язковий; курс 1–5; e-mail коректний
4	гідранти	широта 49,7–49,95; довгота 23,9–24,15; тиск > 0
5	спорядження	назва обов'язкова; кількість ≥ 0; дата придатності в майбутньому

Питання 3.7. Глобальна обробка помилок: @RestControllerAdvice, ProblemDetail

Теоретична частина

Якщо в контролері виник виняток і ніхто його не обробив, клієнт отримає 500 Internal Server Error. Для винятку «виклик не знайдено» це неправильно: помилка не на сервері, а в запиті. Писати `try-catch` у кожному методі контролера теж не варто: код обробки помилок розповзеться по всьому проєкту.

Spring пропонує обробку на рівні застосунку. Клас з анотацією `@RestControllerAdvice` містить методи `@ExceptionHandler(Тип.class)`, які перехоплюють винятки цього типу з усіх контролерів [35]. Служби кидають доменні винятки (`CallNotFoundException`, `IllegalStatusChangeException`), нічого не знаючи про HTTP, а відповідність «виняток → код стану» задається в одному місці.

Формат тіла помилки теж варто стандартизувати. Для цього існує RFC 9457 «Problem Details for HTTP APIs» [38]: JSON з полями `type`, `title`, `status`, `detail`, `instance` і будь-якими додатковими полями. Тип вмісту — `application/problem+json`. У Spring цей формат представлено класом `ProblemDetail`.

Стандартні винятки Spring MVC (некоректне тіло, непідтримуваний тип вмісту, помилки валідації) обробляє базовий клас `ResponseEntityExceptionHandler`. Якщо успадкувати від нього свій обробник, усі вони теж повертатимуть `ProblemDetail`, а окремі можна перевизначити. Boot у такому разі не створює власного обробника — ще один приклад `@ConditionalOnMissingBean` з питання 2.2.

Важливо також, чого не повертати. Повідомлення винятку з трасуванням стека, назвами класів і SQL-запитами в тілі відповіді — подарунок для зловмисника. Клієнт отримує зрозумілий опис, а подробиці йдуть у лог сервера [31].

Приклад задачі

Умова. Створити глобальний обробник помилок `ApiExceptionHandler`, який повертає `ProblemDetail`: 404 для відсутнього виклику, 409 для недопустимої зміни статусу, 400 зі списком полів для помилок валідації. Перевірити кожен випадок.

Лістинг 3.7. Реалізація з покроковим розбором

Файл `DispatchService.java` доменний виняток у службі

```
public FireCall find(long id) {
    return registry.findById(id)
            .orElseThrow(() -> new CallNotFoundException(id));
}
```

Служба нічого не знає про HTTP-коди. Вона повідомляє лише факт: виклику з таким номером немає.

Файл `ApiExceptionHandler.java` обробник для всіх контролерів

```
@RestControllerAdvice
public class ApiExceptionHandler
        extends ResponseEntityExceptionHandler {

    @ExceptionHandler(CallNotFoundException.class)
    public ProblemDetail notFound(CallNotFoundException e) {
        ProblemDetail pd = ProblemDetail.forStatusAndDetail(
                HttpStatus.NOT_FOUND, e.getMessage());
        pd.setTitle("Виклик не знайдено");
        return pd;
    }
```

Метод повертає `ProblemDetail`, і Spring сам виставляє код 404 та тип `application/problem+json`. Обробник для `IllegalStatusChangeException` побудовано так само, з кодом 409.

```
@Override
protected ResponseEntity<Object> handleMethodArgumentNotValid(
        MethodArgumentNotValidException e, HttpHeaders headers,
        HttpStatusCode status, WebRequest request) {
    Map<String, String> errors = new TreeMap<>();
    for (FieldError f : e.getBindingResult().getFieldErrors()) {
        errors.put(f.getField(), f.getDefaultMessage());
    }
```

Перевизначаємо стандартну обробку помилок валідації і збираємо пари «поле — повідомлення». `TreeMap` впорядковує поля за алфавітом, тож відповідь стабільна.

```
    ProblemDetail pd = e.getBody();
    pd.setTitle("Помилка валідації");
    pd.setDetail("Некоректні дані виклику");
    pd.setProperty("errors", errors);
    return ResponseEntity.badRequest().body(pd);
}
```

`setProperty` додає до відповіді нестандартне поле `errors` — RFC 9457 це прямо дозволяє.

Термінал запити з помилками

```
curl -s localhost:8080/api/calls/99
curl -s -X PUT -H "Content-Type: application/json" \
     -d '{"status":"NEW"}' localhost:8080/api/calls/1/status
```

Вивід програми результат запуску

```
{"detail":"Виклик №99 не знайдено","instance":"/api/calls/99",
 "status":404,"title":"Виклик не знайдено"}
{"detail":"Неможливо змінити статус DISPATCHED на NEW",
 "instance":"/api/calls/1/status","status":409,
 "title":"Недопустима зміна статусу"}
```

З досвіду викладачів. *У першій версії прикладу наш обробник не успадковував `ResponseEntityExceptionHandler`, а мав звичайний `@ExceptionHandler(MethodArgumentNotValidException.class)`. Щойно ми увімкнули властивість `spring.mvc.problemdetails.enabled`, Boot додав власний обробник, і він «перехоплював» помилки валідації раніше за наш — у відповіді зник список полів. Коли в застосунку кілька `@ControllerAdvice`, порядок їх застосування без `@Order` не гарантовано. Успадкування від `ResponseEntityExceptionHandler` знімає цю проблему.*

Варіанти для самостійної роботи

Завдання. Створити доменні винятки для ресурсу свого варіанта і глобальний обробник, що перетворює їх на `ProblemDetail` з указаними кодами. Додати до відповіді власне поле.

№	Ресурс	Винятки і коди	Додаткове поле
1	рятувальники	не знайдено — 404; дублікат табельного номера — 409	`personnelNo`
2	автомобілі	не знайдено — 404; автомобіль на ремонті — 409	`plate`
3	курсанти	не знайдено — 404; наряд у день іспиту — 422	`date`
4	гідранти	не знайдено — 404; перевірку вже проведено сьогодні — 409	`lastCheck`
5	спорядження	не знайдено — 404; недостатньо на складі — 409	`available`

Питання 3.8. Пагінація і сортування

Теоретична частина

Журнал чергової частини за рік — це десятки тисяч викликів. Повертати їх усі одним запитом не можна: відповідь важитиме мегабайти, сервер витратить пам’ять, а браузер «задумається». Тому списки в API майже завжди віддають сторінками.

Найпоширеніший підхід — пагінація за зміщенням (offset pagination): клієнт передає номер сторінки і її розмір (`?page=2&size=20`), сервер пропускає `page × size` записів і повертає наступні `size`. Відповідь, крім самих даних, містить метадані: загальну кількість елементів і сторінок. Сортування задають параметром, наприклад `sort=receivedAt` або `sort=-id` (мінус — за спаданням).

У Spring Data є готові типи для цього: `Pageable` як параметр і `Page<T>` як результат [39]. Ми підключимо їх у розділі 5, коли з’явиться база даних. Поки ж реалізуємо пагінацію вручну на потоках — так краще видно, що відбувається всередині.

Обов’язково обмежуйте максимальний розмір сторінки. Інакше будь-хто може надіслати `size=1000000` і фактично отримати весь журнал одним запитом — це і проблема продуктивності, і проблема безпеки.

Для дуже великих або часто змінюваних наборів даних пагінація за зміщенням має вади: на далеких сторінках вона повільна, а нові записи «зсувають» сторінки. Альтернатива — курсорна пагінація (keyset або cursor pagination), коли клієнт передає ідентифікатор останнього отриманого запису [40]. Для рівня Junior досить знати, що такий підхід існує і навіщо він потрібен.

Приклад задачі

Умова. Реалізувати в `CallController` отримання списку викликів сторінками з фільтром за типом і сортуванням за номером (за зростанням і спаданням) або за часом надходження. Розмір сторінки обмежити числом 100. Описати універсальний запис `PageResponse<T>` для відповіді.

Лістинг 3.8. Реалізація з покроковим розбором

Файл `PageResponse.java` сторінка результатів

```
public record PageResponse<T>(List<T> content,
                              int page, int size,
                              long totalElements,
                              int totalPages) {
}
```

Узагальнений запис підійде для будь-якого ресурсу. Назви полів навмисно збігаються з тими, що використовує Spring Data, — клієнтам не доведеться нічого змінювати в розділі 5.

Файл `CallController.java` фільтр, сортування, сторінка

```
List<FireCall> all = registry.findAll().stream()
        .filter(c -> type == null || c.type() == type)
        .sorted(comparator(sort))
        .toList();
```

Спершу фільтруємо і сортуємо весь набір: загальна кількість елементів має враховувати фільтр.

```
List<CallResponse> content = all.stream()
        .skip((long) page * size)
        .limit(size)
        .map(mapper::toResponse)
        .toList();
int pages = (all.size() + size - 1) / size;
```

`skip` і `limit` вирізають потрібну сторінку. Приведення до `long` захищає від переповнення під час множення. Кількість сторінок — ділення з округленням угору.

```
private static Comparator<FireCall> comparator(String sort) {
    return switch (sort) {
        case "receivedAt" ->
                Comparator.comparing(FireCall::receivedAt);
        case "-id" -> Comparator
                .comparingLong(FireCall::id).reversed();
        default -> Comparator.comparingLong(FireCall::id);
    };
}
```

Невідоме значення `sort` не викликає помилки: застосовується сортування за номером. Підставляти рядок від клієнта напряму в запит до бази даних (як `ORDER BY` + `sort`) не можна ніколи — це SQL-ін’єкція.

Термінал перевищення розміру сторінки

```
curl -s "localhost:8080/api/calls?size=500"
```

Вивід програми результат запуску

```
{"detail":"Validation failure","instance":"/api/calls","status":400,
 "title":"Bad Request"}
```

Порада. *Нумерацію сторінок у Spring Data прийнято починати з нуля, і ми зробили так само. Якщо фронтенд-розробник очікує, що перша сторінка — це `page=1`, домовтеся про це заздалегідь і задокументуйте в OpenAPI. Розбіжність «з нуля чи з одиниці» — класичне джерело помилки «зникла перша сторінка».*

Варіанти для самостійної роботи

Завдання. Реалізувати для ресурсу свого варіанта пагінацію, фільтр і сортування за вказаними полями. Обмежити максимальний розмір сторінки і перевірити граничні випадки: порожня сторінка за межами набору, нульовий і завеликий розмір.

№	Ресурс	Фільтр	Сортування	Макс. розмір
1	рятувальники	звання	ПІБ, стаж	50
2	автомобілі	справність	рік, модель	30
3	курсанти	взвод	середній бал (спадання)	100
4	гідранти	район	дата перевірки	200
5	спорядження	категорія	залишок	50

Питання 3.9. Проєктування REST: ідемпотентність, версіонування, документування через OpenAPI

Теоретична частина

REST — архітектурний стиль, описаний Роєм Філдінгом у його дисертації [41]. Основні ідеї: ресурси з унікальними адресами, стандартні HTTP-методи для дій над ними, відсутність стану сеансу на сервері (stateless), представлення ресурсу в певному форматі (зазвичай JSON). Практичні правила іменування: іменники в множині (`/api/calls`), а не дієслова (`/api/getCalls`); вкладені ресурси для залежностей (`/api/calls/7/status`); коди стану за призначенням [42; 43].

Ідемпотентність — властивість методу, за якої повторення того самого запиту не змінює результату після першого виконання [36]. GET, PUT і DELETE ідемпотентні, POST — ні: два однакові POST створять два виклики. Це має практичне значення. Якщо мережа «впала» і клієнт не знає, чи дійшов запит, ідемпотентний запит можна безпечно повторити. Тому зміну статусу виклику ми робимо через `PUT /api/calls/{id}/status` з повним новим значенням, а не через POST з дією «перейти до наступного статусу».

Безпечні (safe) методи не змінюють стану взагалі — це GET і HEAD. Безпечний метод завжди ідемпотентний, але не навпаки.

Версіонування потрібне, коли контракт API змінюється несумісно. Поширені варіанти: у шляху (`/api/v2/calls`), у заголовку (`API-Version: 2`), у параметрі запиту або в типі вмісту. У Spring Framework 7 з'явилася вбудована підтримка версіонування: атрибут `version` в анотаціях `@GetMapping` тощо і налаштування способу передачі версії [5]. Яким би не був спосіб, головне — обрати один і дотримуватися його.

Документацію API зараз описують у форматі OpenAPI [44]. Бібліотека springdoc-openapi генерує її автоматично з контролерів і DTO, а Swagger UI показує інтерактивну сторінку, з якої можна одразу надсилати запити [45]. Анотації `@Tag`, `@Operation`, `@Schema` дозволяють додати описи українською.

Приклад задачі

Умова. Реалізувати ідемпотентну зміну статусу виклику методом PUT: повторний запит з тим самим статусом нічого не змінює, а перехід назад (наприклад, з DISPATCHED до NEW) заборонено. Підключити springdoc-openapi і переглянути згенеровану документацію.

Лістинг 3.9. Реалізація з покроковим розбором

Файл `CallStatus.java` допустимі переходи

```
public enum CallStatus {
    NEW, DISPATCHED, ON_SITE, CLOSED;

    public boolean canMoveTo(CallStatus next) {
        return next.ordinal() == ordinal() + 1
                || (this != CLOSED && next == CLOSED);
    }
}
```

Статуси йдуть по черзі: новий → екіпаж виїхав → на місці → закрито. Закрити виклик можна на будь-якому етапі — наприклад, хибний виклик закривають одразу.

Файл `DispatchService.java` ідемпотентна зміна статусу

```
public FireCall changeStatus(long id, CallStatus next) {
    FireCall call = find(id);
    if (call.status() == next) {
        return call;
    }
    if (!call.status().canMoveTo(next)) {
        throw new IllegalStatusChangeException(
                call.status(), next);
    }
    log.info("Виклик №{}: {} -> {}", id, call.status(), next);
    return registry.save(call.withStatus(next));
}
```

Перша перевірка і робить метод ідемпотентним: якщо статус уже такий, просто повертаємо виклик без помилки і без запису в лог.

Файл `CallController.java` кінцева точка PUT

```
@PutMapping("/{id}/status")
@Operation(summary = "Змінити статус виклику (ідемпотентно)")
public CallResponse changeStatus(
        @PathVariable long id,
        @Valid @RequestBody StatusChange body) {
    FireCall call = service.changeStatus(id, body.status());
    return mapper.toResponse(call);
}
```

Тіло запиту — запис `StatusChange` з одним обов'язковим полем `status`. Анотація `@Operation` додає опис у документацію.

Файл `pom.xml` генератор документації

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>3.1.1</version>
</dependency>
```

Лінійка springdoc 3.x сумісна зі Spring Boot 4. Документація доступна за адресою `/v3/api-docs`, інтерактивна сторінка — `/swagger-ui/index.html`.

Термінал двічі той самий запит

```
curl -s -X PUT -H "Content-Type: application/json" \
     -d '{"status":"DISPATCHED"}' localhost:8080/api/calls/1/status
```

Вивід програми обидва запити повертають 200 і однакове тіло; у лозі — один запис

```
{"id":1,"address":"вул. Клепарівська, 35","type":"FIRE",
 "status":"DISPATCHED","receivedAt":"2026-10-23T11:27:14"}

INFO  DispatchService - Виклик №1: NEW -> DISPATCHED
```

Зверніть увагу. *Під час старту springdoc попереджає в лозі, що документація ввімкнена за замовчуванням. У робочому середовищі її або вимикають властивостями `springdoc.api-docs.enabled=false` і `springdoc.swagger-ui.enabled=false`,*

або закривають автентифікацією. Опис усіх кінцевих точок — корисна підказка не лише для колег, а й для зловмисника.

Варіанти для самостійної роботи

Завдання. Спроєктувати і реалізувати для ресурсу свого варіанта ідемпотентну операцію зміни стану методом PUT з перевіркою допустимих переходів. Підключити springdoc-openapi, додати анотації `@Tag` і `@Operation` українською і продемонструвати документацію у Swagger UI.

№	Ресурс	Стани і переходи
1	рятувальник	на зміні → на виклику → на зміні; будь-який → у відпустці
2	автомобіль	у резерві → у бойовому розрахунку → на виїзді → у резерві; будь-який → на ремонті
3	наряд курсанта	заплановано → заступив → здав; заплановано → скасовано
4	гідрант	справний → потребує перевірки → на ремонті → справний
5	заявка на спорядження	нова → погоджена → видана; нова → відхилена

Розділ 4. Серверний рендеринг

Питання 4.1. Thymeleaf: шаблони й модель

Теоретична частина

REST API віддає дані, а що з ними робити — вирішує клієнт: застосунок на React, мобільний застосунок, табло в залі чергової частини. Але для внутрішніх систем часто простіше і швидше сформувати HTML-сторінку прямо на сервері. Такий підхід називають серверним рендерингом (server-side rendering). Робоче місце чергового, адмінка, звіти для керівництва — типові приклади.

У Spring MVC серверний рендеринг працює так. Метод `@Controller` (без `@ResponseBody`) кладе дані в модель — об'єкт `Model`, по суті мапу «ім'я → значення», — і повертає логічне ім'я подання, наприклад `"journal"`. `ViewResolver` перетворює це ім'я на шаблон `templates/journal.html`, шаблонізатор підставляє дані з моделі, і клієнт отримує готовий HTML [35]. Це класичний патерн MVC: модель — дані, подання — шаблон, контролер — логіка між ними.

Thymeleaf — шаблонізатор, рекомендований для Spring Boot [46]. Його головна особливість — «природні шаблони»: шаблон є коректним HTML-файлом, який можна відкрити в браузері навіть без сервера. Динаміка задається атрибутами з префіксом `th:`, а текст усередині тегів служить прикладом для верстальника і замінюється під час рендерингу.

Основні атрибути, які варто знати:

`th:text` — замінити вміст тегу значенням виразу (з екрануванням HTML);

`th:each` — повторити тег для кожного елемента колекції;

`th:if` і `th:unless` — показати тег за умовою;

`th:href`, `th:action` з виразами посилань `@{/journal}` — правильні адреси з урахуванням контекстного шляху;

`th:classappend` — додати CSS-клас за умовою.

Вирази `${...}` записуються мовою SpEL (Spring Expression Language) і мають доступ до атрибутів моделі. Допоміжні об'єкти на кшталт `#lists`, `#strings`, `#temporals` дають корисні функції.

Зверніть увагу на `th:text`: він екранує спецсимволи. Якщо хтось зареєструє виклик з адресою `<script>alert(1)</script>`, на сторінці буде просто текст, а не виконаний

скрипт. Атрибут `th:utext` виводить HTML без екранування, і використовувати його з даними від користувачів не можна — це прямий шлях до XSS-атаки [31].

Приклад задачі

Умова. Створити сторінку «Журнал викликів» для робочого місця чергового: таблиця з номером, адресою, типом і статусом кожного виклику, повідомлення «Викликів немає» для порожнього журналу і посилання на форму реєстрації. Контролер `JournalController` передає в модель список `CallResponse` — ті самі DTO, що й REST API.

Лістинг 4.1. Реалізація з покроковим розбором

Файл `pom.xml` стартер шаблонізатора

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

Автоконфігурація налаштує `ViewResolver`: шаблони шукатимуться в `src/main/resources/templates` з розширенням `.html`.

Файл `JournalController.java` контролер сторінок

```
@Controller
@RequestMapping("/journal")
public class JournalController {
```

Саме `@Controller`, а не `@RestController`: рядок, який повертає метод, означатиме ім’я шаблону.

```
@GetMapping
public String journal(Model model) {
    model.addAttribute("calls", registry.findAll().stream()
            .map(mapper::toResponse).toList());
    return "journal";
}
```

Кладемо в модель список DTO під іменем `calls` і повертаємо ім’я шаблону. Телефони заявників на сторінку не потраплять — з тієї ж причини, що і в API.

Файл `journal.html` шаблон журналу

```
<p th:if="${message}" th:text="${message}">Повідомлення</p>
<a href="/journal/new">Зареєструвати виклик</a>
```

Повідомлення показується лише тоді, коли воно є в моделі. Звідки воно береться — у питанні 4.2.

```
<tr th:each="c : ${calls}"
    th:classappend="${c.status().name() == 'NEW'} ? 'new'">
    <td th:text="${c.id()}">1</td>
    <td th:text="${c.address()}">вул. Зразкова, 1</td>
    <td th:text="${c.type()}">FIRE</td>
    <td th:text="${c.status()}">NEW</td>
</tr>
```

Рядок таблиці повторюється для кожного виклику. Нові виклики отримують клас `new` — його зручно підсвітити червоним у CSS. Компоненти запису викликаються як методи: `c.id()`.

```
<tr th:if="${#lists.isEmpty(calls)}">
    <td colspan="4">Викликів немає</td>
</tr>
```

Допоміжний об’єкт `#lists` перевіряє, чи порожній список.

Термінал фрагмент згенерованого HTML

```
curl -s localhost:8080/journal
```

Вивід програми результат запуску

```
<h1>Журнал викликів</h1>
...
        <td>1</td>
        <td>вул. Личаківська, 8</td>
        <td>RESCUE</td>
        <td>NEW</td>
```

Порада. *Відкрийте файл* `journal.html` *у браузері напряму, з диска. Ви побачите таблицю з прикладом «вул. Зразкова, 1» — це і є природний шаблон. Дизайнер може верстати сторінку, не запускаючи Java, а розробник потім лише додає атрибути* `th:`*.*

Варіанти для самостійної роботи

Завдання. Створити сторінку зі списком об’єктів свого варіанта на Thymeleaf. Використати `th:each`, `th:if`, `th:classappend` і допоміжний об’єкт, указаний у таблиці.

№	Сторінка	Підсвітити рядок, якщо…	Допоміжний об’єкт
1	склад зміни	стаж рятувальника менше року	`#lists.size`
2	автопарк	автомобіль на ремонті	`#strings.toUpperCase`
3	графік нарядів	наряд припадає на вихідний	`#temporals.format`
4	гідранти	перевірка прострочена	`#temporals.format`
5	склад	залишок нижче мінімуму	`#numbers.formatDecimal`

Питання 4.2. Форми та валідація на сторінці

Теоретична частина

HTML-форма надсилає дані методом POST у форматі `application/x-www-form-urlencoded`: пари `назва=значення`, з’єднані знаком `&`. На боці Spring їх збирає в об’єкт анотація `@ModelAttribute` — на відміну від `@RequestBody`, який читає JSON [35]. Записи (records) підходять і тут: Spring викличе конструктор, підставивши значення полів форми.

Валідація працює так само, як у REST: `@Valid` перед параметром. Різниця в обробці помилок. Замість винятку і відповіді 400 Spring кладе результати перевірки в об’єкт `BindingResult`, який має йти одразу після перевірюваного параметра. Контролер перевіряє `hasErrors()` і, якщо помилки є, повертає ту саму форму. Thymeleaf атрибутом `th:errors` покаже повідомлення біля кожного поля, а `th:field` заповнить поля тим, що користувач уже ввів.

Після успішного збереження форми застосовують патерн Post/Redirect/Get (PRG). Контролер відповідає не сторінкою, а перенаправленням (`redirect:/journal`, код 302). Браузер робить GET-запит на журнал. Навіщо? Якщо повернути сторінку прямо у відповідь на POST, оновлення сторінки (F5) повторно надішле форму — і в журналі з’явиться дубль виклику. Для пожежної частини це не дрібниця.

Щоб після перенаправлення показати повідомлення «Виклик зареєстровано», використовують flash-атрибути (`RedirectAttributes.addFlashAttribute`). Вони зберігаються в сесії рівно до наступного запиту, а потім зникають.

Зауважимо, що для форм, які змінюють дані, потрібен захист від CSRF-атак. Spring Security додає його автоматично, а Thymeleaf сам вставляє у форму прихований токен. До цього ми повернемося в розділі 7.

Приклад задачі

Умова. Створити форму реєстрації виклику з полями «Адреса», «Тип» (випадний список) і «Телефон заявника». Використати той самий `CallRequest` з обмеженнями, що й у REST API. За наявності помилок показати форму з повідомленнями біля полів; після успішної реєстрації перенаправити на журнал з повідомленням.

Лістинг 4.2. Реалізація з покроковим розбором

Файл `JournalController.java` порожня форма

```
@GetMapping("/new")
public String form(Model model) {
    model.addAttribute("call", new CallRequest("", null, null));
    model.addAttribute("types", CallType.values());
    return "call-form";
}
```

Форма потребує об’єкта в моделі, навіть порожнього: `th:object` до нього прив’язується. Перелік типів передаємо окремо — для випадного списку.

```
@PostMapping
public String create(
        @Valid @ModelAttribute("call") CallRequest call,
        BindingResult errors, Model model,
        RedirectAttributes redirect) {
    if (errors.hasErrors()) {
        model.addAttribute("types", CallType.values());
        return "call-form";
    }
```

`BindingResult` стоїть одразу після `CallRequest`, інакше Spring кине виняток замість того, щоб передати помилки. Повертаємо форму, не забувши знову покласти перелік типів.

```
    FireCall saved = service.accept(call.address(), call.type(),
            call.callerPhone());
    redirect.addFlashAttribute("message",
            "Виклик №" + saved.id() + " зареєстровано");
    return "redirect:/journal";
}
```

Префікс `redirect:` — вказівка Spring відповісти кодом 302. Повідомлення переживе перенаправлення і з’явиться в журналі один раз.

Файл `call-form.html` шаблон форми

```
<form th:action="@{/journal}" th:object="${call}" method="post">
    <label>Адреса
        <input th:field="*{address}">
    </label>
    <span th:errors="*{address}">помилка</span>
```

`th:object` задає об’єкт форми, а `*{address}` — його поле. `th:field` сам згенерує атрибути `id`, `name` і `value`.

```
<select th:field="*{type}">
    <option value="">— оберіть —</option>
    <option th:each="t : ${types}" th:value="${t}"
            th:text="${t}">FIRE</option>
</select>
```

Випадний список будується з переліку `CallType`. Порожній варіант змушує чергового обрати тип свідомо.

Файл `CallRequest.java` виправлення після тестування форми

```
@NotNull(message = "тип виклику обов'язковий")
CallType type,

@Pattern(regexp = "(\\+380\\d{9})?",
         message = "формат +380XXXXXXXXX")
String callerPhone) {
```

Дві зміни порівняно з розділом 3. Обидві виявилися лише тоді, коли ми почали надсилати дані формою, — див. примітку.

Термінал некоректна і коректна форма

```
curl -s -d "address=&type=&callerPhone=067" localhost:8080/journal
curl -s -i -d "address=вул. Зелена, 101&type=FIRE" localhost:8080/journal
```

Вивід програми фрагменти відповідей

```
    <span>адресу не вказано</span>
    <span>тип виклику обов'язковий</span>
    <span>формат +380XXXXXXXXX</span>

HTTP/1.1 302
Location: http://localhost:8080/journal;jsessionid=4E89D702...
```

З досвіду викладачів. *Форма спершу відмовлялася реєструвати виклик без телефону. Причина: порожнє поле форми приходить на сервер як порожній рядок `""`, а не як `null`, а `@Pattern` вважає коректним лише `null`. У REST API поле просто не передавали, і помилка не проявлялася. Тому регулярний вираз тепер дозволяє й порожнє значення. Друга знахідка дрібніша, але кумедна: апостроф у повідомленні «обов'язковий» на сторінці зник. Spring обробляє повідомлення валідації через `MessageFormat`, а для нього одинарна лапка — службовий символ. Типографський апостроф ' такої проблеми не має.*

Варіанти для самостійної роботи

Завдання. Створити форму додавання об'єкта свого варіанта на Thymeleaf з валідацією, повідомленнями біля полів, патерном Post/Redirect/Get і flash-повідомленням. Перевірити, що оновлення сторінки після реєстрації не створює дубль.

№	Форма	Поля і елементи керування
1	новий рятувальник	ПІБ, звання (список), стаж (число), дата народження
2	новий автомобіль	номер, модель (список), рік, справний (прапорець)
3	наряд курсанта	курсант (список), дата, вид наряду (перемикачі)
4	результат перевірки гідранта	гідрант (список), тиск, коментар (textarea)
5	видача спорядження	позиція (список), кількість, отримувач

Питання 4.3. CORS і взаємодія з фронтендом (fetch)

Теоретична частина

Сучасний фронтенд часто живе окремо від бекенду: застосунок на React чи Vue під час розробки працює на `http://localhost:5173`, а API — на `http://localhost:8080`. Для браузера це різні джерела (origin), бо джерело визначається трійкою «схема + хост + порт». А браузер за замовчуванням дотримується політики одного джерела (same-origin policy): скрипт зі сторінки одного джерела не може прочитати відповідь з іншого.

CORS (Cross-Origin Resource Sharing) — механізм, за допомогою якого сервер повідомляє браузеру, яким чужим джерелам він дозволяє доступ [47]. Для «складних» запитів (POST з JSON, PUT, DELETE, власні заголовки) браузер спершу надсилає

попередній запит (preflight) методом OPTIONS із заголовками `Origin` і `Access-Control-Request-Method`. Сервер відповідає заголовками `Access-Control-Allow-Origin`, `Access-Control-Allow-Methods` тощо. Лише якщо дозвіл є, браузер надсилає справжній запит.

Найважливіше, що треба зрозуміти: CORS — це захист користувача в браузері, а не захист сервера. `curl`, Postman чи інший сервер CORS не перевіряють узагалі. Тому CORS не замінює автентифікації й авторизації.

У Spring CORS налаштовують глобально через `WebMvcConfigurer.addCorsMappings` або точково анотацією `@CrossOrigin` на контролері [35]. Типова помилка — `allowedOrigins("*")` «щоб запрацювало». Для публічного API без автентифікації це допустимо, але разом із cookies чи токенами Spring таку конфігурацію просто не дозволить (`allowCredentials` з `*` заборонено специфікацією). Перелічуйте конкретні джерела, а для різних середовищ виносьте їх у властивості.

На боці фронтенду запити до API надсилає функція `fetch`: `fetch('http://localhost:8080/api/calls', {method: 'POST', headers: {'Content-Type': 'application/json'}, body: JSON.stringify(data)})`. Саме цей запит з JSON-тілом і спричинить preflight.

Приклад задачі

Умова. Дозволити фронтенду, що працює на `http://localhost:5173`, звертатися до REST API методами GET, POST, PUT і DELETE. Перевірити попередній запит від дозволеного і від чужого джерела.

Браузер для перевірки не потрібен: попередній запит можна відтворити командою `curl`, вказавши потрібні заголовки вручну.

Лістинг 4.3. Реалізація з покроковим розбором

Файл `WebConfig.java` глобальні правила CORS

```
@Override
public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/api/**")
            .allowedOrigins("http://localhost:5173")
            .allowedMethods("GET", "POST", "PUT", "DELETE");
}
```

Правило діє лише для API. Сторінки журналу з розділу 4.1 віддаються з того самого джерела, тож CORS їм не потрібен. Метод `addCorsMappings` додано до того самого `WebConfig`, де вже зареєстровано перехоплювач.

Термінал імітація попереднього запиту браузера

```
curl -s -i -X OPTIONS -H "Origin: http://localhost:5173" \
     -H "Access-Control-Request-Method: POST" localhost:8080/api/calls
curl -s -i -X OPTIONS -H "Origin: http://evil.example" \
     -H "Access-Control-Request-Method: POST" localhost:8080/api/calls
```

Вивід програми відповіді (скорочено)

```
HTTP/1.1 200
Access-Control-Allow-Origin: http://localhost:5173
Access-Control-Allow-Methods: GET,POST,PUT,DELETE
Access-Control-Max-Age: 1800

HTTP/1.1 403
Invalid CORS request
```

Зверніть увагу. *`Access-Control-Max-Age: 1800` дозволяє браузеру пам'ятати результат попереднього запиту 30 хвилин і не надсилати OPTIONS перед кожним викликом API. Якщо ви змінили налаштування CORS, а браузер «не бачить» змін — очистіть кеш або відкрийте приватне вікно.*

Варіанти для самостійної роботи

Завдання. Налаштувати CORS для API свого варіанта за вказаними умовами. Написати просту HTML-сторінку з викликом `fetch` і перевірити роботу у браузері, а також відтворити попередній запит командою `curl`.

№	Дозволене джерело	Методи	Додатково
1	`http://localhost:3000`	GET	лише для `/api/rescuers/**`
2	`http://localhost:5173`	GET, POST	дозволити заголовок `X-Duty-Shift`
3	два джерела з властивості `duty.cors.origins`	усі	`maxAge` 10 хв
4	`http://localhost:4200`	GET, PUT	через `@CrossOrigin` на контролері
5	`http://localhost:5173`	GET, POST, DELETE	`allowCredentials(true)`; пояснити, чому не можна `*`

ЧАСТИНА 3. ДАНІ

Розділ 5. Spring Data JPA

Досі журнал викликів жив у пам'яті й зникав після кожного перезапуску. У цьому розділі застосунок отримує справжню базу даних PostgreSQL. Ми спираємося на розділи першої книжки про SQL, JDBC і Hibernate [1], тому основи ORM тут лише нагадуємо, а зосереджуємося на тому, що додає Spring Data. Навчальні класи `Demo` з розділів 1–4 залишилися на мітках `ch01`–`ch04`: без бази даних вони вже не запускаються.

Питання 5.1. Репозиторії: CrudRepository і JpaRepository

Теоретична частина

Пригадайте JDBC-код з першої книжки: відкрити з'єднання, підготувати запит, заповнити параметри, пройти по `ResultSet`, закрити ресурси. Hibernate позбавив нас більшої частини цієї роботи, але методи на кшталт «зберегти», «знайти за ідентифікатором», «видалити» все одно доводилося писати для кожної сутності. Spring Data JPA прибирає і цю рутину [48].

Ідея проста: ми оголошуємо інтерфейс, що успадковує один із базових інтерфейсів Spring Data, і вказуємо тип сутності та тип її ідентифікатора. Реалізацію Spring створить сам під час старту — це проксі-об'єкт, який перенаправляє виклики до `EntityManager`. Ієрархія базових інтерфейсів така:

`Repository` — маркерний інтерфейс без методів;

`CrudRepository` — `save`, `findById`, `findAll`, `count`, `deleteById`, `existsById`;

`PagingAndSortingRepository` — додає `findAll(Pageable)` і `findAll(Sort)`;

`JpaRepository` — усе перелічене плюс специфічні для JPA методи: `flush`, `saveAndFlush`, пакетне видалення, повернення `List` замість `Iterable`.

На практиці найчастіше успадковують `JpaRepository`. Анотація `@Repository` на інтерфейсі не потрібна: Spring Boot знаходить такі інтерфейси сам.

Сутність JPA — це клас, позначений `@Entity`, з полем-ідентифікатором `@Id` [49]. Записи (records) для сутностей не підходять: Hibernate потребує конструктора без параметрів і змінних полів, щоб відстежувати зміни. Тому `FireCall`, який був записом у розділах 1–4, став звичайним класом із захищеним конструктором без параметрів. Записи ми залишили для DTO.

Налаштування підключення — звичайні властивості `spring.datasource.*`. Пул з'єднань HikariCP Boot створює автоматично. Ще одна важлива властивість — `spring.jpa.open-in-view`. За замовчуванням вона ввімкнена і тримає сесію Hibernate відкритою до кінця HTTP-запиту, через що ліниві зв'язки «працюють» навіть у контролерах. Ми її вимикаємо: так помилки доступу до даних проявляються одразу, а не в несподіваних місцях (див. питання 5.3).

Приклад задачі

Умова. Підключити до проєкту PostgreSQL і Spring Data JPA. Описати сутність `FireCall` і репозиторій `CallRepository`. Зберегти виклик, знайти його за ідентифікатором і подивитися, який клас насправді реалізує інтерфейс репозиторію.

Схему бази даних створюють міграції Flyway, які ми детально розглянемо в питанні 5.7. Поки що досить знати, що таблиці `fire_call`, `crew`, `rescuer` і `fire_truck` уже існують, а в останніх трьох є початкові дані.

Лістинг 5.1. Реалізація з покроковим розбором

Файл `application.yml` підключення до бази даних

```
datasource:
  url: ${DB_URL:jdbc:postgresql://localhost:5432/duty}
  username: ${DB_USER:duty}
  password: ${DB_PASSWORD:duty}
jpa:
  open-in-view: false
  hibernate:
    ddl-auto: validate
```

Адреса бази (`DB_URL`), логін і пароль беруться зі змінних середовища, а після двокрапки — значення для розробки. `ddl-auto: validate` означає, що Hibernate лише перевіряє відповідність сутностей таблицям, а не змінює схему.

Файл `FireCall.java` сутність

```
@Entity
@EntityListeners(AuditingEntityListener.class)
public class FireCall {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String address;

    @Enumerated(EnumType.STRING)
    private CallType type;
```

`IDENTITY` — ідентифікатор генерує база (стовпець `BIGSERIAL`). `EnumType.STRING` зберігає назву константи: з `ORDINAL` додавання нового типу в середину переліку зіпсувало б усі старі записи. Слухач аудиту пояснимо в питанні 5.8.

```
protected FireCall() {
}

public FireCall(String address, CallType type,
                LocalDateTime receivedAt, String callerPhone) {
```

Захищений конструктор без параметрів — для Hibernate, публічний — для нашого коду. Решту класу займають поля зв’язків, аудиту і гетери.

Файл `CallRepository.java` репозиторій

```
public interface CallRepository
        extends JpaRepository<FireCall, Long>,
                JpaSpecificationExecutor<FireCall> {
```

Інтерфейс без жодного рядка реалізації. Методи запитів і `JpaSpecificationExecutor` розглянемо в питаннях 5.2 і 5.6.

Файл `Demo51Repository.java` збереження і пошук

```
FireCall saved = calls.save(new FireCall(
        "вул. Клепарівська, 35", CallType.FIRE,
        LocalDateTime.now(), null));
System.out.println("Збережено виклик №" + saved.getId());
```

Після `save` ідентифікатор уже заповнено: Hibernate виконав `INSERT` і прочитав згенероване значення.

```
calls.findById(saved.getId()).ifPresent(c -> System.out
        .println("Знайдено: " + c.getAddress()
                + ", статус " + c.getStatus()));
System.out.println("Реалізація: "
        + calls.getClass().getSimpleName());
```

`findById` повертає `Optional` — відсутність запису не є помилкою. Ім'я класу показує, що репозиторій реалізовано динамічним проксі.

Вивід програми результат запуску

```
Автомобілів у базі: 3
Збережено виклик №1
Знайдено: вул. Клепарівська, 35, статус NEW
Реалізація: $Proxy153
```

З досвіду викладачів. *На комп'ютері, де ми перевіряли приклади, стояв старий PostgreSQL 13.2, і застосунок не стартував з помилкою `invalid value for parameter "TimeZone": "Europe/Kyiv"`. Драйвер передає серверу часовий пояс JVM, а стара база не знала нової назви поясу, яку змінили з `Europe/Kiev` 2022 року. На PostgreSQL 17 проблеми немає. Але сама ситуація типова: оновлюйте базу даних, а не обходьте помилку зміною поясу застосунку.*

Варіанти для самостійної роботи

Завдання. Описати сутність свого варіанта, створити для неї таблицю (тимчасово можна скриптом SQL) і репозиторій на основі `JpaRepository`. Продемонструвати збереження, пошук за ідентифікатором, підрахунок і видалення.

№	Сутність	Поля
1	`Rescuer`	ПІБ, звання, стаж, дата народження
2	`FireTruck`	номер, модель, рік, справність
3	`Cadet`	ПІБ, взвод, курс, середній бал
4	`Hydrant`	адреса, тип, тиск, дата перевірки
5	`EquipmentItem`	назва, категорія, кількість, термін придатності

Питання 5.2. Похідні запити (derived queries) та @Query

Теоретична частина

Найцікавіша можливість Spring Data — запити, які виводяться з назви методу [48]. Метод `findByTypeAndStatus(CallType type, CallStatus status)` Spring розбирає на частини: `find…By` — це вибірка, `Type` і `Status` — поля сутності, `And` — логічне «і». З цього будується JPQL-запит `select c from FireCall c where c.type = ?1 and c.status = ?2`.

Префікси бувають різні: `find`, `read`, `get`, `query` (вибірка), `count` (кількість), `exists` (наявність), `delete` (видалення). Ключових слів багато: `And`, `Or`, `Between`, `LessThan`, `GreaterThanEqual`, `Like`, `Containing`, `StartingWith`, `In`, `IsNull`, `Not`, `True`, `OrderBy…Desc`. Обмеження кількості результатів — `findTop3By…`, `findFirstBy…`.

Помилки в назві методу Spring виявляє ще під час старту: якщо поля `adress` немає, застосунок не запуститься. Це дуже зручно.

Але назви методів мають межу. `findByTypeAndStatusAndReceivedAtAfterAndCrewNameOrderByReceivedAtDesc` формально працює, але читати таке неможливо. Для складніших запитів використовують анотацію `@Query` з текстом на JPQL. JPQL схожий на SQL, але оперує сутностями і їхніми полями, а не таблицями і стовпцями. Параметри позначають іменами (`:from`) і прив'язують анотацією `@Param`.

Якщо потрібні специфічні можливості конкретної СУБД (функції дат PostgreSQL, віконні функції, рекурсивні запити), додають `nativeQuery = true` і пишуть звичайний SQL. Ціна — прив'язка до однієї бази даних.

Приклад задачі

Умова. Додати до репозиторію викликів похідні запити: вибірку за типом і статусом, кількість незакритих викликів, три останні виклики; JPQL-запит відкритих викликів з певного моменту і нативний SQL-запит кількості викликів за сьогодні. Перевірити їх на трьох тестових викликах, один з яких закрито.

Лістинг 5.2. Реалізація з покроковим розбором

Файл `CallRepository.java` запити з назви методу

```
List<FireCall> findByTypeAndStatus(CallType type,
                                   CallStatus status);

long countByStatusNot(CallStatus status);

List<FireCall> findTop3ByOrderByReceivedAtDesc();
```

Три методи — три різні запити, і жодного рядка SQL. Саме `countByStatusNot(CLOSED)` служба використовує для перевірки ліміту викликів зміни.

```
@Query("""
        select c from FireCall c
        where c.receivedAt >= :from and c.status <> 'CLOSED'
        order by c.receivedAt""")
List<FireCall> findOpenSince(@Param("from") LocalDateTime from);
```

JPQL у текстовому блоці Java. Звертаємося до сутності `FireCall` і поля `receivedAt`, а не до таблиці `fire_call` і стовпця `received_at`.

```
@Query(value = """
        SELECT count(*) FROM fire_call
        WHERE received_at::date = current_date""",
       nativeQuery = true)
long countToday();
```

Нативний запит із приведенням типу `::date`, специфічним для PostgreSQL. На іншій СУБД він не спрацює.

Файл `Demo52Queries.java` перевірка запитів

```
System.out.println("FIRE і NEW: " + calls
        .findByTypeAndStatus(CallType.FIRE, CallStatus.NEW)
        .stream().map(FireCall::getAddress).toList());
System.out.println("Незакритих: "
        + calls.countByStatusNot(CallStatus.CLOSED));
```

Перед цим демо реєструє дві пожежі й одне рятування, а одну з пожеж закриває.

Вивід програми результат запуску

```
FIRE і NEW: [вул. Зелена, 101]
Незакритих: 2
Відкриті за годину: 2
Сьогодні (SQL): 3
```

Порада. *Увімкніть у профілі* `dev` *властивість* `spring.jpa.show-sql: true` *і подивіться, які SQL-запити Hibernate будує з ваших методів. Корисна вправа — хоча б раз переписати похідний метод вручну на JPQL і порівняти згенерований SQL — це найкращий спосіб переконатися, що «магії» немає.*

Варіанти для самостійної роботи

Завдання. Для репозиторію свого варіанта написати три похідні запити, один JPQL-запит з `@Query` і один нативний SQL-запит. Продемонструвати їх роботу на тестових даних.

№	Похідні запити	JPQL	Нативний SQL
1	за званням; стаж понад N; топ-3 за стажем	рятувальники без караулу	середній стаж за званнями
2	справні; за моделлю; найновіший	автомобілі певної моделі, відсортовані за роком	кількість за роками випуску
3	за взводом; бал понад N; останні 5 доданих	курсанти без нарядів за тиждень	рейтинг за балом (`rank()`)
4	за районом; тиск менше N; перевірені після дати	гідранти з простроченою перевіркою	кількість за районами
5	за категорією; залишок менше N; з терміном до дати	позиції, що закінчуються за місяць	сума залишків за категоріями

Питання 5.3. Зв’язки між сутностями; FetchType LAZY і EAGER

Теоретична частина

У реальній предметній області сутності пов’язані. Караул має автомобіль і кількох рятувальників, виклик обслуговує караул. JPA описує такі зв’язки анотаціями [49; 50]:

`@ManyToOne` — багато рятувальників належать одному караулу (на боці таблиці — зовнішній ключ `crew_id`);

`@OneToMany(mappedBy = "crew")` — зворотний бік: караул бачить список своїх рятувальників; `mappedBy` вказує, що власником зв’язку є поле `crew` у `Rescuer`;

`@OneToOne` і `@ManyToMany` — рідше, але теж трапляються.

Стратегія завантаження (fetch type) визначає, коли пов’язані дані читаються з бази. `EAGER` — одразу разом з основною сутністю. `LAZY` — лише тоді, коли до зв’язку звернулися вперше; до того часу на його місці стоїть проксі-заглушка. За специфікацією `@ManyToOne` і `@OneToOne` за замовчуванням `EAGER`, а колекції — `LAZY`.

Практична рекомендація Влада Міхалчі, одного з провідних фахівців з Hibernate, і більшості практиків: робіть усі зв’язки `LAZY`, а потрібні дані завантажуйте явно під конкретний сценарій [51]. `EAGER` здається зручним, але він тягне дані завжди, навіть коли вони не потрібні, і його неможливо «вимкнути» для окремого запиту.

Ліниве завантаження працює лише всередині відкритої сесії Hibernate, тобто транзакції. Якщо звернутися до лінивого зв’язку після її закриття, отримаємо `LazyInitializationException` — мабуть, найвідомішу помилку початківців у JPA. Лікувати її треба не вмиканням `EAGER` і не `open-in-view`, а завантаженням потрібних даних у межах транзакції: через `JOIN FETCH`, `@EntityGraph` або маппінг у DTO всередині служби.

Приклад задачі

Умова. Описати сутності `Crew` (караул), `Rescuer` (рятувальник) і зв’язки: караул — автомобіль, караул — рятувальники, виклик — караул. Усі зв’язки зробити

лінивими. Показати `LazyInitializationException` під час звернення до караулу виклику поза транзакцією і правильний доступ усередині транзакції.

Лістинг 5.3. Реалізація з покроковим розбором

Файл `Crew.java` караул

```
@Entity
public class Crew {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    @ManyToOne(fetch = FetchType.LAZY)
    private FireTruck truck;

    @OneToMany(mappedBy = "crew")
    private List<Rescuer> members = new ArrayList<>();
```

Явно вказуємо `LAZY` для `@ManyToOne`, бо за замовчуванням там `EAGER`. Колекція `members` лінива і без цього. Ініціалізуємо її порожнім списком, щоб ніколи не мати `null`.

Файл `Rescuer.java` власник зв'язку

```
@ManyToOne(fetch = FetchType.LAZY)
private Crew crew;
```

Саме тут, на боці «багатьох», зберігається зовнішній ключ `crew_id`. Тому цей бік — власник зв'язку.

Файл `Demo53Lazy.java` доступ поза транзакцією

```
FireCall call = calls.findById(id).orElseThrow();
try {
    System.out.println(call.getCrew().getName());
} catch (RuntimeException e) {
    System.out.println("Поза транзакцією: "
            + e.getClass().getSimpleName());
}
```

`findById` виконується у власній короткій транзакції. Після повернення сесія закрита, а `getCrew()` повертає неініціалізований проксі.

```
String crew = tx.execute(s -> calls.findById(id)
        .orElseThrow().getCrew().getName());
System.out.println("У транзакції: " + crew);
```

`TransactionTemplate` виконує код в одній транзакції. Звернення до караулу спричиняє ще один SELECT, і все працює.

Вивід програми результат запуску

```
Поза транзакцією: LazyInitializationException
У транзакції: 1-й караул
```

Зверніть увагу. *Якщо в `application.yml` увімкнути `spring.jpa.open-in-view: true`, у контролері помилки не буде: сесія житиме до кінця запиту. Саме тому цю властивість часто називають антипатерном — вона ховає проблему, а запити до бази при цьому виконуються вже під час формування JSON, поза транзакцією і без контролю.*

Варіанти для самостійної роботи

Завдання. Описати для своєї предметної області дві-три сутності зі зв'язками `@ManyToOne` і `@OneToMany` (усі `LAZY`). Продемонструвати `LazyInitializationException` і два способи коректно завантажити пов'язані дані.

№	Сутності та зв'язки
1	`Station` (частина) 1 — N `Crew`; `Crew` 1 — N `Rescuer`
2	`FireTruck` 1 — N `Maintenance` (ТО); `Maintenance` N — 1 `Mechanic`
3	`Platoon` (взвод) 1 — N `Cadet`; `Cadet` 1 — N `DutyShift`
4	`District` 1 — N `Hydrant`; `Hydrant` 1 — N `Inspection`
5	`Warehouse` 1 — N `EquipmentItem`; `EquipmentItem` 1 — N `Issue` (видача)

Питання 5.4. Проблема N+1 і як її розв'язати

Теоретична частина

Проблема N+1 — найпоширеніша причина повільної роботи застосунків з ORM і улюблене питання на співбесідах Junior і Middle. Суть така. Ми одним запитом завантажуємо N караулів, а потім для кожного звертаємося до лінивої колекції рятувальників. Hibernate для кожного караулу виконує окремий SELECT. Разом — 1 + N запитів. Для трьох караулів це непомітно, для трьохсот записів у звіті — це вже секунди очікування [51].

Найнеприємніше в N+1 те, що код виглядає абсолютно невинно: звичайний цикл і виклик гетера. Помилку видно лише в лозі SQL-запитів або в статистиці.

Способи розв'язання:

`JOIN FETCH` у JPQL — пов'язані дані завантажуються тим самим запитом через JOIN;

`@EntityGraph(attributePaths = …)` на методі репозиторію — те саме, але декларативно, без написання JPQL;

`@BatchSize` або властивість `hibernate.default_batch_fetch_size` — колекції довантажуються пакетами через `IN (…)`, тож замість N запитів буде N/розмір пакета;

проєкція в DTO (питання 5.6) — завантажуються лише потрібні поля, і зв'язки взагалі не беруть участі.

Одне застереження: `JOIN FETCH` колекції разом із пагінацією змушує Hibernate завантажувати всі рядки і ділити їх на сторінки в пам'яті — він попередить про це в лозі. Для сторінкових списків краще пакетне завантаження або DTO.

Приклад задачі

Умова. Показати проблему N+1 на списку караулів з кількістю рятувальників і розв'язати її двома способами — через `JOIN FETCH` і через `@EntityGraph`. Кількість виконаних SQL-запитів рахувати за статистикою Hibernate.

Лістинг 5.4. Реалізація з покроковим розбором

Файл `CrewRepository.java` два способи завантажити колекцію

```
@Query("select distinct c from Crew c join fetch c.members")
List<Crew> findAllWithMembers();

@EntityGraph(attributePaths = {"members", "truck"})
List<Crew> findByNameStartingWith(String prefix);
```

`distinct` прибирає дублікати караулів, що виникають через JOIN. `@EntityGraph` додає до похідного запиту завантаження і рятувальників, і автомобіля.

Файл `Demo54NPlusOne.java` підрахунок запитів

```
static void run(String name, Supplier<List<Crew>> query,
                TransactionTemplate tx, Statistics stats) {
    stats.clear();
    tx.executeWithoutResult(s -> query.get().forEach(c ->
            System.out.println("  " + c.getName() + ": "
                    + c.getMembers().size() + " особи")));
    System.out.println(name + " -> SQL-запитів: "
            + stats.getPrepareStatementCount());
}
```

Той самий цикл виконуємо для трьох різних запитів. Статистику Hibernate вмикає властивість `hibernate.generate_statistics`, яку демо передає під час старту.

```
run("findAll()", crews::findAll, tx, stats);
run("join fetch", crews::findAllWithMembers, tx, stats);
run("@EntityGraph",
    () -> crews.findByNameStartingWith(""), tx, stats);
```

Порожній префікс у `findByNameStartingWith` повертає всі караули — так ми порівнюємо однакові набори даних.

Вивід програми (рядки з кількістю осіб скорочено)

```
  1-й караул: 3 особи
  2-й караул: 2 особи
  3-й караул: 2 особи
findAll() -> SQL-запитів: 4
join fetch -> SQL-запитів: 1
@EntityGraph -> SQL-запитів: 1
```

З досвіду викладачів. *На співбесіді мало сказати «використаю JOIN FETCH». Інтерв'юер майже напевно запитає, як ви взагалі помітите проблему. Хороша відповідь: увімкнути лог SQL у середовищі розробки, а в тестах перевіряти кількість запитів, наприклад за статистикою Hibernate, як у цьому прикладі.*

Варіанти для самостійної роботи

Завдання. На сутностях зі свого варіанта питання 5.3 відтворити проблему N+1 і розв'язати її трьома способами: `JOIN FETCH`, `@EntityGraph` і `@BatchSize`. Порівняти кількість SQL-запитів за статистикою Hibernate для 3, 10 і 50 батьківських записів.

№	Батьківська сутність	Колекція, що спричиняє N+1
1	`Station`	караули частини
2	`FireTruck`	записи про ТО
3	`Platoon`	курсанти взводу
4	`District`	гідранти району
5	`Warehouse`	позиції складу

Питання 5.5. @Transactional: поширення транзакцій і типові пастки

Теоретична частина

Транзакція гарантує, що група змін у базі даних виконається повністю або не виконається зовсім — це властивість атомарності з набору ACID, знайомого з першої книжки [1]. У Spring транзакціями керують декларативно: метод позначають `@Transactional`, і Spring сам відкриває транзакцію перед викликом, фіксує (commit) після успішного завершення і відкочує (rollback), якщо вилетів виняток [52].

Як це працює? Пригадайте `BeanPostProcessor` з питання 1.5. Для біна з `@Transactional` Spring створює проксі, і саме проксі керує транзакцією, а потім

викликає справжній метод. Звідси випливають три пастки, про які обов'язково питають на співбесідах.

Перша — самовиклик. Якщо метод класу викликає інший метод того ж класу (`this.accept(...)`), виклик не проходить через проксі, і анотація на другому методі ігнорується. У нашому прикладі це не шкодить, бо зовнішній метод уже транзакційний, але з `REQUIRES_NEW` на внутрішньому методі нова транзакція так і не відкриється.

Друга — тип винятку. За замовчуванням транзакція відкочується лише через неперевірювані винятки (`RuntimeException`) і помилки (`Error`). Перевірюваний виняток (`IOException`, власний `extends Exception`) залишить транзакцію зафіксованою. Змінити це можна атрибутом `rollbackFor`.

Третя — модифікатор доступу. Анотація на `private`-методі не працює: проксі не може його перехопити.

Атрибут `propagation` визначає, що робити, якщо транзакція вже є. `REQUIRED` (за замовчуванням) — приєднатися до наявної або створити нову; `REQUIRES_NEW` — призупинити наявну і створити окрему (наприклад, для журналу аудиту, який має зберегтися навіть у разі відкату основної операції); `SUPPORTS`, `MANDATORY`, `NEVER`, `NOT_SUPPORTED`, `NESTED` — рідші варіанти.

`@Transactional(readOnly = true)` на класі — корисна звичка для служб: читання виконується без зайвих перевірок змін, а методи, що пишуть, позначаються `@Transactional` окремо. Ставте анотацію на рівні служб, а не контролерів чи репозиторіїв: саме служба знає, які операції утворюють одну бізнес-дію.

Приклад задачі

Умова. Додати до служби прийому викликів пакетну реєстрацію: кілька викликів одного типу реєструються в одній транзакції, і якщо будь-яка адреса порожня, не зберігається жоден. Показати відкат транзакції і успішну реєстрацію пакета.

Такий сценарій реальний: під час масштабної надзвичайної ситуації черговий отримує список адрес від диспетчера області одним повідомленням.

Лістинг 5.5. Реалізація з покроковим розбором

Файл `DispatchService.java` транзакції в службі

```
@Service
@Transactional(readOnly = true)
public class DispatchService {
```

Типово всі методи служби лише читають дані. Методи, що змінюють дані, перевизначають це власною анотацією.

```
@Transactional
public List<FireCall> acceptBatch(List<String> addresses,
                                  CallType type) {
    return addresses.stream()
            .map(a -> {
                if (a.isBlank()) {
                    throw new IllegalArgumentException(
                            "Порожня адреса в пакеті");
                }
                return accept(a, type, null);
            })
            .toList();
}
```

`accept` викликається всередині того самого класу — це самовиклик, його власна анотація `@Transactional` не діє. Але всі виклики виконуються в транзакції `acceptBatch`, тож виняток на третій адресі відкотить і перші дві.

Файл `Demo55Transactions.java` відкат і фіксація

```
System.out.println("До: " + calls.count());
try {
    service.acceptBatch(List.of("вул. Сихівська, 4",
            "вул. Наукова, 7", " "), CallType.HAZMAT);
} catch (IllegalArgumentException e) {
    System.out.println("Помилка: " + e.getMessage());
}
System.out.println("Після: " + calls.count());
```

Третя адреса — пробіл. Перші два виклики встигли отримати номери, але після відкату їх у базі немає.

Вивід програми результат запуску

```
До: 0
INFO  DispatchService - Зареєстровано виклик №1
INFO  DispatchService - Зареєстровано виклик №2
Помилка: Порожня адреса в пакеті
Після: 0
INFO  DispatchService - Зареєстровано виклик №3
INFO  DispatchService - Зареєстровано виклик №4
Після вдалого пакета: 2
```

Зверніть увагу. *Номери 1 і 2 після відкату не повторно використовуються: наступний пакет отримав 3 і 4. Послідовності PostgreSQL не відкочуються разом із транзакцією. Тому не покладайтеся на те, що ідентифікатори йдуть без пропусків, — для «номера в журналі» без дірок потрібен окремий механізм.*

Варіанти для самостійної роботи

Завдання. Реалізувати в службі свого варіанта транзакційну операцію з указаною умовою відкату. Продемонструвати відкат, а також пастку з указаного стовпця і спосіб її уникнути.

№	Операція	Відкат, якщо…	Пастка
1	переведення рятувальника в інший караул	у новому караулі вже 6 осіб	самовиклик з `REQUIRES_NEW`
2	списання автомобіля з передачею екіпажу	екіпажу немає куди перейти	перевірюваний виняток без `rollbackFor`
3	формування нарядів на тиждень	у курсанта два наряди за день	`@Transactional` на `private`-методі
4	реєстрація результатів перевірки району	хоча б один гідрант не знайдено	`readOnly = true` на методі, що пише
5	видача комплекту спорядження	будь-якої позиції бракує	журнал видач має зберегтися і в разі відкату (`REQUIRES_NEW`)

Питання 5.6. Проєкції та Specification для фільтрів

Теоретична частина

Не завжди потрібна вся сутність. Для випадного списку викликів досить номера й адреси, для звіту — кількості за типами. Завантажувати для цього повні сутності зі всіма полями й зв’язками — марна робота. Spring Data пропонує проєкції [48].

Інтерфейсна проєкція — інтерфейс з гетерами потрібних полів. Метод репозиторію повертає цей інтерфейс замість сутності, і Spring вибирає з бази лише перелічені стовпці.

DTO-проєкція — запис або клас, який заповнюється конструкторним виразом JPQL `select new пакет.Клас(...)`. Так зручно будувати звіти з агрегатними функціями.

Друга половина питання — динамічні фільтри. У розділі 3 фільтр за типом ми робили вручну на потоках, у пам’яті. Тепер дані в базі, і фільтрувати треба запитом. Але якщо фільтрів кілька, і кожен необов’язковий, похідних методів знадобиться 2^n. Тут

допомагає Specification — невеликий об’єкт-умова на основі JPA Criteria API. Специфікації комбінуються методами `and` і `or`, а специфікація, що повертає `null`, просто ігнорується. Репозиторій отримує можливість `findAll(spec, pageable)`, успадкувавши `JpaSpecificationExecutor`.

Для дуже складних динамічних запитів у проєктах використовують також Querydsl або jOOQ, але для рівня Junior досить упевнено володіти специфікаціями.

Приклад задачі

Умова. Додати до репозиторію інтерфейсну проєкцію «номер — адреса — статус» і DTO-проєкцію зі статистикою викликів за типами. Описати специфікації для фільтрів за типом, статусом і часом. Переписати метод списку в `CallController` з ручної фільтрації на специфікації та пагінацію Spring Data.

Лістинг 5.6. Реалізація з покроковим розбором

Файл `CallSummary.java` інтерфейсна проєкція

```
public interface CallSummary {
    Long getId();
    String getAddress();
    CallStatus getStatus();
}
```

Метод репозиторію `List<CallSummary> findByStatus(CallStatus)` вибере з бази лише три стовпці.

Файл `CallRepository.java` DTO-проєкція зі статистикою

```
@Query("""
        select new ua.edu.ldubgd.duty.call.TypeStat(
                    c.type, count(c))
        from FireCall c
        group by c.type order by count(c) desc""")
List<TypeStat> statsByType();
```

`TypeStat` — запис з двома полями. Повне ім’я класу в JPQL обов’язкове. Цей самий метод відповідає на запит `GET /api/calls/stats`.

Файл `CallSpecs.java` специфікації

```
public static Specification<FireCall> hasType(CallType type) {
    return (root, query, cb) -> type == null
            ? null : cb.equal(root.get("type"), type);
}
```

Якщо фільтр не задано, специфікація повертає `null`, і умова просто не потрапляє в запит. Специфікації `hasStatus` і `since` побудовано так само.

Файл `CallController.java` список через специфікації

```
Specification<FireCall> spec = Specification
        .where(CallSpecs.hasType(type))
        .and(CallSpecs.hasStatus(status));
Page<CallResponse> result = calls
        .findAll(spec, PageRequest.of(page, size, toSort(sort)))
        .map(mapper::toResponse);
```

Фільтрація, сортування і вибір сторінки тепер виконуються в базі одним запитом (і ще одним — для підрахунку). `Page.map` перетворює сутності на DTO, зберігаючи метадані сторінки.

```
return new PageResponse<>(result.getContent(), page, size,
        result.getTotalElements(), result.getTotalPages());
```

Формат відповіді `PageResponse` з розділу 3 не змінився — клієнти API нічого не помітять.

Вивід програми запит GET /api/calls?type=FIRE&sort=-id&size=1 і GET /api/calls/stats

```
{"content":[{"id":3,"address":"просп. Чорновола, 59","type":"FIRE",
  "status":"NEW","receivedAt":"2026-10-23T11:47:05"}],
 "page":0,"size":1,"totalElements":2,"totalPages":2}
[{"type":"FIRE","count":2},{"type":"RESCUE","count":1}]
```

Порада. *Назви полів у специфікаціях (`root.get("type")`) — рядки, і помилка в них проявиться лише під час виконання. Hibernate вміє генерувати статичну метамодель (`FireCall_.type`), яка перетворює такі помилки на помилки компіляції. Для навчального проєкту рядків досить, але в робочому коді метамодель варто підключити.*

Варіанти для самостійної роботи

Завдання. Для ресурсу свого варіанта додати інтерфейсну проєкцію, DTO-проєкцію зі звітом і специфікації для фільтрів. Переписати метод списку REST-контролера на `findAll(spec, pageable)`.

№	Інтерфейсна проєкція	Звіт (DTO-проєкція)	Фільтри
1	ПІБ і звання	кількість рятувальників за званнями	звання, мін. стаж, караул
2	номер і модель	кількість автомобілів за моделями	модель, справність, рік від
3	ПІБ і взвод	середній бал за взводами	взвод, курс, бал від
4	адреса і тиск	кількість несправних за районами	район, тип, перевірка до дати
5	назва і залишок	сума залишків за категоріями	категорія, залишок менше, термін до

Питання 5.7. Міграції схеми: Flyway

Теоретична частина

Звідки беруться таблиці в базі даних? Найпростіший спосіб — дозволити Hibernate створювати їх самому (`ddl-auto: update` або `create`). Для навчання це зручно, для реальної системи — небезпечно: Hibernate не знає, як перейменувати стовпець без втрати даних, не створює потрібних індексів, а зміни схеми неможливо переглянути на код-рев'ю.

Тому схему бази даних ведуть міграціями — упорядкованими SQL-скриптами, які зберігаються в репозиторії разом із кодом [53]. Flyway під час старту застосунку

перевіряє, які скрипти вже застосовано (таблиця `flyway_schema_history`), і виконує нові по черзі. Імена файлів задають порядок: `V1__init.sql`, `V2__seed.sql`, `V3__audit.sql`. Подвійне підкреслення обов'язкове.

Головне правило Flyway: застосовану міграцію не можна змінювати. Flyway зберігає контрольну суму кожного скрипту і відмовиться стартувати, якщо файл змінено. Потрібна зміна — пишіть нову міграцію. Саме тому третя міграція в нашому проєкті не редагує першу, а додає стовпці аудиту командою `ALTER TABLE`.

Альтернатива Flyway — Liquibase, з описом змін у XML або YAML і підтримкою відкату. На співбесіді досить знати, що обидва інструменти розв'язують одну задачу, і пояснити, чому `ddl-auto: update` не використовують у робочому середовищі.

Після підключення міграцій у Hibernate залишається скромна, але корисна роль: `ddl-auto: validate`. Під час старту він порівнює сутності зі схемою і зупинить застосунок, якщо, наприклад, у сутності є поле, якого немає в таблиці.

Приклад задачі

Умова. Описати схему бази даних чергової частини міграціями Flyway: створення таблиць, початкові дані (автомобілі, караули, рятувальники) і окремою міграцією — стовпці аудиту. Перевірити історію застосованих міграцій.

Лістинг 5.7. Реалізація з покроковим розбором

Файл `pom.xml` залежності для міграцій

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-flyway</artifactId>
</dependency>
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-database-postgresql</artifactId>
</dependency>
```

Підтримка конкретних СУБД у Flyway винесена в окремі модулі. Без другої залежності Flyway не розпізнає PostgreSQL.

Файл `V1__init.sql` створення схеми (фрагмент)

```
CREATE TABLE crew (
    id       BIGSERIAL PRIMARY KEY,
    name     VARCHAR(50) NOT NULL UNIQUE,
    truck_id BIGINT REFERENCES fire_truck (id)
);
```

Стандартний каталог для міграцій — `db/migration` у ресурсах. Імена стовпців (`truck_id`) збігаються з тими, що Hibernate очікує для поля `truck`.

```
CREATE TABLE fire_call (
    id           BIGSERIAL PRIMARY KEY,
    address      VARCHAR(200) NOT NULL,
    type         VARCHAR(20)  NOT NULL,
    status       VARCHAR(20)  NOT NULL,
    received_at  TIMESTAMP    NOT NULL,
    caller_phone VARCHAR(13),
    crew_id      BIGINT REFERENCES crew (id)
);

CREATE INDEX idx_fire_call_status ON fire_call (status);
```

Індекс за статусом прискорює підрахунок незакритих викликів, який служба виконує під час кожної реєстрації.

Файл `V3__audit.sql` нова міграція замість зміни старої

```
ALTER TABLE fire_call
    ADD COLUMN created_by VARCHAR(50),
    ADD COLUMN created_at TIMESTAMP,
    ADD COLUMN updated_at TIMESTAMP;
```

Стовпці для аудиту (питання 5.8). Оскільки `V1` вже застосовано на всіх базах, змінювати його не можна.

Термінал історія міграцій

```
psql -U duty -d duty \
  -c "select version, script, success from flyway_schema_history"
```

Вивід програми результат запуску

```
 version |    script     | success
---------+---------------+---------
 1       | V1__init.sql  | t
 2       | V2__seed.sql  | t
 3       | V3__audit.sql | t
```

З досвіду викладачів. *Типовий сценарій: помилку виправляють прямо у `V1__init.sql`, і застосунок перестає стартувати з повідомленням про розбіжність контрольних сум. На навчальній базі можна її просто видалити і створити знову. На робочій — ні, і саме цю звичку «писати нову міграцію» ми хочемо виробити ще під час навчання.*

Варіанти для самостійної роботи

Завдання. Описати схему бази даних свого варіанта міграціями Flyway: створення таблиць з обмеженнями й індексом, початкові дані та окремою міграцією — зміна схеми з указаного стовпця. Увімкнути `ddl-auto: validate` і показати, що станеться, якщо сутність не відповідає таблиці.

№	Таблиці	Зміна схеми в окремій міграції
1	`station`, `crew`, `rescuer`	додати стовпець `phone` до `rescuer`
2	`fire_truck`, `maintenance`	перейменувати `mileage` на `mileage_km`
3	`platoon`, `cadet`, `duty_shift`	додати унікальне обмеження «курсант + дата»
4	`district`, `hydrant`, `inspection`	додати індекс за датою перевірки
5	`warehouse`, `equipment_item`, `issue`	розділити `name` на `name` і `article`

Питання 5.8. Аудит: @CreatedDate і @CreatedBy

Теоретична частина

Для інформаційної системи цивільного захисту важливо не лише що записано, а й хто і коли це зробив. Хто зареєстрував виклик? Коли востаннє змінювали статус? Ці питання постають і під час службових розслідувань, і під час звичайного аналізу роботи зміни.

Spring Data бере цю роботу на себе [39]. Поля сутності позначають анотаціями:

`@CreatedDate` і `@LastModifiedDate` — час створення і останньої зміни;

`@CreatedBy` і `@LastModifiedBy` — користувач, який створив і змінив запис.

Щоб анотації запрацювали, потрібні три речі. Анотація `@EnableJpaAuditing` на конфігураційному класі. Слухач `@EntityListeners(AuditingEntityListener.class)` на сутності — він заповнює поля перед `INSERT` і `UPDATE`. І бін `AuditorAware`, що повертає поточного користувача. Поки в нас немає автентифікації, він повертає умовне «черговий»; у розділі 7 тут з’явиться ім’я користувача зі Spring Security.

За замовчуванням час береться з системного годинника. Ми ж у розділі 1 домовилися отримувати час через бін `Clock` з київським поясом. Тому підключаємо власний

`DateTimeProvider`, який бере час із того самого годинника. Це й узгодженість, і можливість підставити фіксований час у тестах.

Аудит на рівні полів сутності відповідає на питання «хто створив і коли змінив», але не зберігає історію всіх змін. Для повної історії використовують Hibernate Envers (окрема таблиця версій кожного запису) або журнал подій — до подій ми повернемося в розділі 6.

Приклад задачі

Умова. Увімкнути аудит JPA для викликів: зберігати, хто і коли створив виклик, і коли його востаннє змінювали. Час брати з біна `Clock`. Зареєструвати виклик, через півтори секунди змінити його статус і вивести поля аудиту.

Лістинг 5.8. Реалізація з покроковим розбором

Файл `JpaConfig.java` налаштування аудиту

```
@Configuration
@EnableJpaAuditing(dateTimeProviderRef = "auditClock")
public class JpaConfig {

    @Bean
    public AuditorAware<String> auditorAware() {
        return () -> Optional.of("черговий");
    }
```

`AuditorAware` — функціональний інтерфейс, тож досить лямбди. Атрибут `dateTimeProviderRef` вказує ім'я біна, що постачає час.

```
@Bean
public DateTimeProvider auditClock(Clock clock) {
    return () -> Optional.of(LocalDateTime.now(clock));
}
```

Час аудиту береться з того самого `Clock`, що й час надходження виклику.

Файл `FireCall.java` поля аудиту

```
@CreatedBy
private String createdBy;
@CreatedDate
private LocalDateTime createdAt;
@LastModifiedDate
private LocalDateTime updatedAt;
```

Ці поля не мають сетерів: їх заповнює лише слухач аудиту.

Вивід програми результат запуску

```
Створив: черговий
Створено: 2026-10-23T11:46:34.221859
Змінено:  2026-10-23T11:46:35.769834
```

Зверніть увагу. *У демо статус змінюється методом `changeStatus` служби, і ми ніде не викликаємо `save`. Hibernate сам помітив зміну керованої сутності (dirty checking) і під час фіксації транзакції виконав `UPDATE`, а слухач аудиту оновив `updatedAt`. Це одна з тих речей, які варто вміти пояснити на співбесіді: чому зміна об'єкта в транзакції потрапляє в базу без явного збереження.*

Варіанти для самостійної роботи

Завдання. Увімкнути аудит JPA для сутності свого варіанта. Додати вказані поля і продемонструвати їх заповнення під час створення і зміни.

№	Сутність	Поля аудиту	Користувач з `AuditorAware`
1	`Rescuer`	створено, змінено, ким змінено	«кадрова служба»
2	`FireTruck`	створено, змінено	ім’я з властивості `duty.auditor`
3	`DutyShift`	створено, ким створено	«черговий курсу»
4	`Inspection`	створено, ким створено, змінено, ким змінено	«інспектор»
5	`Issue`	створено, ким створено	«завідувач складу»

Питання 5.9. Оптимістичне блокування: @Version

Теоретична частина

Уявімо ситуацію. Два чергових на різних робочих місцях одночасно відкрили картку автоцистерни АЦ-40, яка позначена як вільна. Перший призначає її на пожежу, другий — на інший виклик. Обидва бачать «вільна», обидва зберігають. Без захисту друге збереження перепише перше, і автомобіль буде «призначено» на два виклики одночасно. Це класична проблема втраченого оновлення (lost update).

Є два підходи до її розв’язання. Песимістичне блокування — заблокувати рядок у базі під час читання (`SELECT … FOR UPDATE`), щоб інші чекали. Це надійно, але знижує паралельність і загрожує взаємоблокуваннями. Оптимістичне блокування припускає, що конфлікти рідкісні, і лише перевіряє їх у момент запису [51; 40].

У JPA оптимістичне блокування вмикається одним полем з анотацією `@Version` [49]. Hibernate додає до кожного `UPDATE` умову `WHERE id = ? AND version = ?` і збільшує версію на одиницю. Якщо хтось уже змінив запис, версія в базі інша, оновлено нуль рядків, і Hibernate кидає виняток. Spring перетворює його на `ObjectOptimisticLockingFailureException`.

Що робити з винятком? Не «глушити» його. Користувач має дізнатися, що дані змінилися, і повторити дію з актуальним станом. У REST API для цього підходить код 409 Conflict. Іноді операцію можна безпечно повторити автоматично — для цього є механізми повторних спроб (розділ 9).

Для оптимістичного блокування версія має повертатися клієнтові і надходити назад разом зі зміною — інакше між запитами HTTP конфлікт не буде виявлено. Часто для цього використовують заголовки `ETag` і `If-Match`. У нашому прикладі перевірку показано в межах однієї JVM, щоб механізм був добре видно.

Приклад задачі

Умова. Захистити пожежний автомобіль від одночасного призначення: додати поле `@Version` і перевірку зайнятості під час призначення караулу на виклик. Відтворити ситуацію з двома черговими, які прочитали той самий автомобіль, і перетворити виняток на відповідь 409.

Лістинг 5.9. Реалізація з покроковим розбором

Файл `FireTruck.java` версія сутності

```
@Version
private long version;
```

Стовпець `version` створено в міграції `V1` зі значенням 0 за замовчуванням. Змінювати його вручну не можна — це робить лише Hibernate.

Файл `DispatchService.java` призначення караулу

```
FireTruck truck = crew.getTruck();
if (!truck.isAvailable()) {
    throw new TruckBusyException(truck.getPlate());
}
truck.setAvailable(false);
call.setCrew(crew);
call.setStatus(CallStatus.DISPATCHED);
```

Перевірка `isAvailable` захищає від послідовних призначень, `@Version` — від одночасних: якщо інша транзакція встигла змінити автомобіль, фіксація цієї завершиться винятком.

Файл `Demo59OptimisticLock.java` двоє чергових

```
FireTruck first = trucks.findById(1L).orElseThrow();
FireTruck second = trucks.findById(1L).orElseThrow();
System.out.println("Обидва бачать версію "
        + first.getVersion());
```

Кожен `findById` виконується в окремій транзакції, тож маємо дві незалежні копії того самого автомобіля — як на двох робочих місцях.

```
first.setAvailable(false);
trucks.save(first);
```

Перший черговий зберігає зміну. Версія в базі стає 1.

```
second.setAvailable(false);
try {
    trucks.save(second);
} catch (ObjectOptimisticLockingFailureException e) {
    System.out.println("Другий черговий: "
            + e.getClass().getSimpleName());
}
```

Копія другого чергового має версію 0, а в базі вже 1. Оновлення не відбувається.

Файл `ApiExceptionHandler.java` відповідь 409

```
@ExceptionHandler(ObjectOptimisticLockingFailureException.class)
public ProblemDetail concurrent(
        ObjectOptimisticLockingFailureException e) {
    ProblemDetail pd = ProblemDetail.forStatusAndDetail(
            HttpStatus.CONFLICT,
            "Дані змінив інший користувач, повторіть спробу");
    pd.setTitle("Одночасне редагування");
    return pd;
}
```

Вивід програми вивід демо і повторне призначення автомобіля через REST API

```
Обидва бачать версію 0
Перший черговий зберіг, версія 1
Другий черговий: ObjectOptimisticLockingFailureException

{"detail":"Автомобіль ВС 0101 ДС уже на виїзді",
 "instance":"/api/calls/3/crew/1","status":409,"title":"Конфлікт стану"}
```

Порада. *Поле версії легко забути при маппінгу в DTO і назад — тоді механізм тихо перестає працювати між HTTP-запитами. Перевірте себе: відкрийте одну й ту саму картку у двох вкладках браузера, змініть у першій, потім у другій. Якщо друга зміна пройшла без помилки — версія десь загубилася.*

Варіанти для самостійної роботи

Завдання. Додати оптимістичне блокування до сутності свого варіанта. Відтворити конфлікт двох одночасних змін, повернути клієнту 409 і реалізувати один з указаних способів реакції на конфлікт.

№	Сутність	Конфліктна операція	Реакція на конфлікт
1	`Rescuer`	одночасне переведення в різні караули	показати актуальний караул
2	`FireTruck`	одночасне списання і призначення на ТО	повідомлення з версією в тілі
3	`DutyShift`	двоє чергових курсу міняють наряд	заголовки `ETag` / `If-Match`
4	`Hydrant`	два інспектори вносять тиск	автоматичний повтор один раз
5	`EquipmentItem`	дві одночасні видачі останньої одиниці	друга видача — 409 з залишком

Розділ 6. Кешування й асинхронність

Питання 6.1. Абстракція кешу Spring: @Cacheable, @CacheEvict

Теоретична частина

Деякі дані читаються в сотні разів частіше, ніж змінюються. Довідник пожежних автомобілів частини оновлюють кілька разів на рік, а відкривають під час кожного виклику. Ходити щоразу в базу за тим самим списком — марна витрата часу і з'єднань. Розв'язок — кеш: результат першого звернення зберігається в пам'яті, наступні звернення отримують його звідти.

Spring пропонує абстракцію кешу: ми лише позначаємо методи анотаціями, а де саме зберігаються дані, вирішує підключена реалізація [54]. Без додаткових бібліотек це звичайна `ConcurrentHashMap` у пам'яті. Для робочих систем підключають Caffeine (локальний кеш з обмеженням розміру і часом життя записів) або Redis (спільний кеш для кількох екземплярів застосунку). Код при цьому не змінюється.

Основні анотації:

`@Cacheable("назва")` — якщо в кеші є значення для цих аргументів, метод не викликається;

`@CachePut` — метод викликається завжди, а результат оновлює кеш;

`@CacheEvict` — видалити запис або весь кеш (`allEntries = true`) після зміни даних;

`@EnableCaching` — увімкнути все це в конфігурації.

Ключ кешу за замовчуванням утворюється з аргументів методу, а атрибут `key` дозволяє задати його виразом SpEL. Об'єкти, що повертаються з кешу, — ті самі екземпляри, тому їх варто робити незмінними. Зберігати в кеші сутності JPA — погана ідея: вони прив'язані до сесії, мають ліниві зв'язки і змінювані поля. Кешуйте DTO.

Як і транзакції, кеш працює через проксі (питання 5.5). Отже, і пастка та сама: виклик кешованого методу зсередини того самого класу кеш оминає.

Фразу «є дві складні речі в програмуванні — інвалідація кешу і придумування назв» цитують на співбесідах не випадково. Головне питання для кожного кешу: коли дані в ньому застаріють і хто про це подбає?

Приклад задачі

Умова. Створити службу `FleetService` з кешованим довідником автомобілів і пошуком автомобіля за номером. Під час зміни моделі автомобіля очищати обидва кеші. Показати, коли метод справді звертається до бази.

Лістинг 6.1. Реалізація з покроковим розбором

Файл `AsyncConfig.java` увімкнення кешу

```
@Configuration
@EnableCaching
@EnableAsync
@EnableScheduling
public class AsyncConfig {
}
```

Один конфігураційний клас для всього розділу: кеш (6.1), асинхронні методи (6.2) і задачі за розкладом (6.3). Стартер `spring-boot-starter-cache` додано в `pom.xml`.

Файл `FleetService.java` кешовані методи

```
@Cacheable("trucks")
public List<TruckView> catalog() {
    log.info("Читаю довідник автомобілів з бази");
    return trucks.findAll().stream()
            .map(t -> new TruckView(t.getPlate(), t.getModel()))
            .toList();
}
```

Повертаємо незмінні записи `TruckView`, а не сутності. Рядок у лозі з'являтиметься лише тоді, коли метод справді виконується.

```
@Cacheable(value = "truckByPlate", key = "#plate")
public TruckView byPlate(String plate) {
    log.info("Шукаю в базі {}", plate);
    return catalog().stream()
            .filter(t -> t.plate().equals(plate))
            .findFirst().orElseThrow();
}
```

Ключ — номер автомобіля. Виклик `catalog()` тут — самовиклик: він пройде повз кеш (див. вивід).

```
@Transactional
@CacheEvict(value = {"trucks", "truckByPlate"},
            allEntries = true)
public void rename(String plate, String model) {
    trucks.findByPlate(plate).orElseThrow().setModel(model);
}
```

Після зміни очищаємо обидва кеші. Без цього довідник показував би стару модель аж до перезапуску.

Файл `Demo61Cache.java` п'ять звернень

```
System.out.println("1: " + fleet.catalog().size() + " авто");
System.out.println("2: " + fleet.catalog().size() + " авто");

fleet.rename("ВС 0103 ДС", "АР-2 (модернізований)");
System.out.println("3: " + fleet.catalog().get(2).model());

System.out.println("4: " + fleet.byPlate("ВС 0101 ДС"));
System.out.println("5: " + fleet.byPlate("ВС 0101 ДС"));
```

Вивід програми результат запуску

```
INFO  FleetService - Читаю довідник автомобілів з бази
1: 3 авто
2: 3 авто
INFO  FleetService - Читаю довідник автомобілів з бази
3: АР-2 (модернізований)
INFO  FleetService - Шукаю в базі ВС 0101 ДС
INFO  FleetService - Читаю довідник автомобілів з бази
4: TruckView[plate=ВС 0101 ДС, model=АЦ-40]
5: TruckView[plate=ВС 0101 ДС, model=АЦ-40]
```

Зверніть увагу. *Подивіться на виклик 4. Довідник щойно, на кроці 3, потрапив у кеш, але* `byPlate` *прочитав його з бази ще раз. Це і є самовиклик:* `catalog()` *викликано через* `this`*, а не через проксі. Виправити можна, винісши довідник в окремий бін або впровадивши службу саму в себе через* `@Lazy`*. На співбесіді цей приклад працює краще за будь-яке визначення.*

Варіанти для самостійної роботи

Завдання. Додати кешування до служби свого варіанта: кешований список, кешований пошук за ключем і очищення кешу після зміни даних. Продемонструвати за логом, коли метод звертається до джерела даних. Окремо відтворити пастку самовиклику і виправити її.

№	Кешовані дані	Ключ пошуку	Що очищає кеш
1	перелік звань	код звання	додавання звання
2	довідник моделей техніки	назва моделі	зміна нормативу пального
3	розклад занять	номер взводу	перенесення заняття
4	довідник вулиць	назва вулиці	імпорт нових вулиць
5	номенклатура спорядження	артикул	зміна категорії

Питання 6.2. @Async і віртуальні потоки

Теоретична частина

Деякі операції довгі, але їхнього результату не треба чекати негайно: надсилання сповіщень трьом караулам, формування звіту, запит до зовнішнього сервісу. Якщо виконувати їх по черзі в потоці HTTP-запиту, черговий чекатиме суму всіх затримок.

Анотація `@Async` переносить виконання методу в інший потік [54]. Викликач отримує керування одразу, а метод працює паралельно. Якщо результат усе ж потрібен, метод повертає `CompletableFuture<T>`, і викликач може дочекатися всіх результатів разом. Метод без результату повертає `void`; винятки з нього, щоправда, не дійдуть до викликача — їх обробляє `AsyncUncaughtExceptionHandler`.

Пастки ті самі, що в `@Transactional` і `@Cacheable`: асинхронність працює через проксі, тож самовиклик і `private`-методи не працюють. Окреме застереження — транзакції: асинхронний метод виконується в іншому потоці, а транзакція прив’язана до потоку, тож він її не бачить.

У якому потоці виконується метод? Spring Boot створює бін `applicationTaskExecutor` (ми бачили його в питанні 2.2). Класично це пул платформних потоків. Але з Java 21 у JDK є віртуальні потоки [55]. Це легковагові потоки, якими керує JVM, а не операційна система: їх можна створити мільйони, а блокування на очікуванні мережі чи бази майже нічого не коштує. Властивість `spring.threads.virtual.enabled=true` вмикає їх для виконавця задач, для Tomcat, для планувальника і для інших компонентів Spring Boot [21].

Віртуальні потоки не роблять код швидшим самі по собі. Вони дають змогу обробляти більше одночасних операцій, що чекають на введення-виведення. Для обчислень, які навантажують процесор, вони нічого не змінюють. Про це варто пам’ятати, щоб не обіцяти на співбесіді «у п’ять разів швидше».

Багатопоточність загалом ми розглядали в першій книжці [1; 17], тому тут обмежимося практичним прикладом.

Приклад задачі

Умова. Увімкнути віртуальні потоки і створити асинхронну службу `AsyncNotifier`, яка надсилає повідомлення караулу й «чекає підтвердження» одну секунду. Надіслати повідомлення трьом караулам і виміряти загальний час.

Лістинг 6.2. Реалізація з покроковим розбором

Файл `application.yml` віртуальні потоки

```
threads:
  virtual:
    enabled: true
```

Одна властивість — і `@Async`, `@Scheduled` та обробка HTTP-запитів у Tomcat виконуються на віртуальних потоках.

Файл `AsyncNotifier.java` асинхронний метод

```
@Async
public CompletableFuture<String> notifyCrew(String crew,
                                            String text)
        throws InterruptedException {
    Thread t = Thread.currentThread();
    log.info("{}: надсилаю «{}» (віртуальний: {})", crew, text,
             t.isVirtual());
    Thread.sleep(1000);
    String answer = crew + " підтвердив";
    return CompletableFuture.completedFuture(answer);
}
```

`Thread.sleep` імітує очікування відповіді зовнішнього сервісу. Метод `isVirtual()` з’явився разом із віртуальними потоками.

Файл `Demo62Async.java` три виклики паралельно

```
List<CompletableFuture<String>> answers = List.of(
        notifier.notifyCrew("1-й караул", "збір"),
        notifier.notifyCrew("2-й караул", "збір"),
        notifier.notifyCrew("3-й караул", "збір"));
System.out.println("Запити надіслано за "
        + (System.currentTimeMillis() - start) + " мс");
```

Усі три виклики повертаються миттєво: справжня робота пішла в інші потоки.

```
CompletableFuture.allOf(answers.toArray(
        CompletableFuture[]::new)).join();
answers.forEach(a -> System.out.println(a.join()));
```

`allOf(...).join()` чекає завершення всіх трьох. Загальний час — приблизно одна секунда, а не три.

Вивід програми результат запуску

```
INFO  AsyncNotifier - 1-й караул: надсилаю «збір» (віртуальний: true)
Запити надіслано за 1 мс
INFO  AsyncNotifier - 3-й караул: надсилаю «збір» (віртуальний: true)
INFO  AsyncNotifier - 2-й караул: надсилаю «збір» (віртуальний: true)
1-й караул підтвердив
2-й караул підтвердив
3-й караул підтвердив
Усього: 1007 мс
```

Порада. *Зверніть увагу на порядок рядків: третій караул «почав» раніше за другий. Порядок виконання асинхронних задач не гарантовано. А от результати ми виводимо в*

порядку списку `answers`, бо самі так їх і зібрали. Не будуйте логіку на тому, яка задача завершиться першою.

Варіанти для самостійної роботи

Завдання. Створити асинхронну службу свого варіанта, що повертає `CompletableFuture`, запустити кілька задач паралельно і порівняти час із послідовним виконанням. Перевірити, на яких потоках виконуються задачі з увімкненими і вимкненими віртуальними потоками.

№	Асинхронна операція	Кількість задач	Імітована затримка
1	розсилка SMS рятувальникам зміни	7	500 мс
2	запит стану автомобілів у GPS-системі	3	800 мс
3	формування звітів по взводах	5	1 с
4	перевірка доступності камер на об’єктах	10	300 мс
5	запит залишків у трьох складах	3	700 мс

Питання 6.3. Задачі за розкладом: @Scheduled

Теоретична частина

Чимало дій у черговій частині відбувається за годинником: щогодинний звіт про незакриті виклики, підсумок під час зміни чергування о 08:00, нагадування про наряди курсантів, щонічне архівування журналу. Spring дозволяє запускати такі задачі анотацією `@Scheduled` на методі біна [54]. Метод не повинен мати параметрів і зазвичай повертає `void`. Увімкнути планувальник можна анотацією `@EnableScheduling`.

Розклад задають кількома способами:

`fixedRate` — запускати з фіксованим інтервалом від початку попереднього запуску;

`fixedDelay` — запускати через заданий час після завершення попереднього запуску;

`initialDelay` — затримка перед першим запуском;

`cron` — вираз з шести полів: секунди, хвилини, години, день місяця, місяць, день тижня. Наприклад, `0 0 8 * * *` — щодня о 08:00:00.

Різниця між `fixedRate` і `fixedDelay` — класичне питання. Якщо задача іноді виконується довше за інтервал, `fixedRate` намагатиметься «наздогнати» розклад, а `fixedDelay` завжди витримає паузу між запусками.

Інтервали і cron-вирази варто виносити у властивості: так у профілі `dev` звіт можна запускати щосекунди, а в робочому середовищі — щогодини. Атрибути з суфіксом `String` (`fixedRateString`) приймають і плейсхолдери, і зручний формат тривалості (`2s`, `1h`). Для cron-виразів обов’язково вказуйте часовий пояс атрибутом `zone`, інакше «08:00» на сервері в UTC настане о 10:00 або 11:00 за Києвом.

Окреме застереження стосується кількох екземплярів застосунку. Якщо запустити дві копії сервісу, задача виконуватиметься двічі. Для кластера потрібне розподілене блокування, наприклад бібліотека ShedLock або планувальник Quartz із базою даних [56].

Приклад задачі

Умова. Створити компонент `ShiftReportJob` з двома задачами: періодичний звіт про кількість незакритих викликів з інтервалом із властивостей і підсумок за добу

при зміні чергування о 08:00 за київським часом. Для демонстрації запустити застосунок з інтервалом дві секунди.

Лістинг 6.3. Реалізація з покроковим розбором

Файл `application.yml` розклад у властивостях

```
report:
  interval: 1h
  shift-change-cron: "0 0 8 * * *"
```

Значення для робочого середовища. Будь-яке з них можна перевизначити профілем або аргументом командного рядка.

Файл `ShiftReportJob.java` дві задачі

```
@Scheduled(fixedRateString = "${duty.report.interval}",
           initialDelayString = "${duty.report.interval}")
public void openCallsReport() {
    log.info("Звіт: незакритих викликів {}",
             calls.countByStatusNot(CallStatus.CLOSED));
}
```

Перший звіт — не одразу після старту, а через один інтервал: у момент старту звітувати ще нема про що.

```
@Scheduled(cron = "${duty.report.shift-change-cron}",
           zone = "Europe/Kyiv")
public void shiftChange() {
    log.info("Зміна чергування: підсумок за добу {} викликів",
             calls.count());
}
```

Явний часовий пояс: зміна чергування настає о 08:00 за Києвом незалежно від налаштувань сервера.

Файл `Demo63Scheduled.java` запуск з коротким інтервалом

```
try (var ctx = DemoApp.start("--duty.report.interval=2s")) {
    var service = ctx.getBean(DispatchService.class);
    service.accept("вул. Зелена, 101", CallType.FIRE, null);
    Thread.sleep(2500);
    service.accept("вул. Стрийська, 1", CallType.RESCUE, null);
    Thread.sleep(2000);
}
```

Застосунок працює приблизно чотири з половиною секунди, тож звіт встигає спрацювати двічі.

Вивід програми результат запуску

```
08:53:27 INFO  DispatchService - Зареєстровано виклик №1
08:53:29 INFO  ShiftReportJob - Звіт: незакритих викликів 1
08:53:30 INFO  DispatchService - Зареєстровано виклик №2
08:53:31 INFO  ShiftReportJob - Звіт: незакритих викликів 2
```

З досвіду викладачів. *Перевіряти cron-вираз, чекаючи до восьмої ранку, незручно. Є швидший спосіб: клас `CronExpression` зі Spring уміє обчислити наступний момент запуску — `CronExpression.parse("0 0 8 * * *").next(LocalDateTime.now())`. Перевірте так свій вираз перед тим, як здавати лабораторну.*

Варіанти для самостійної роботи

Завдання. Створити компонент із задачами за розкладом для свого варіанта. Винести розклад у властивості, вказати часовий пояс для cron і продемонструвати роботу з коротким інтервалом у профілі `dev`. Пояснити вибір між `fixedRate` і `fixedDelay`.

№	Періодична задача	Задача за cron
1	перевірка рятувальників без караулу (кожні 30 хв)	список відпусток на тиждень (пн, 07:00)
2	перевірка автомобілів на ТО (щогодини)	звіт про пальне (останній день місяця, 18:00)
3	нагадування про наряд за годину (кожні 10 хв)	графік нарядів на завтра (щодня, 20:00)
4	перевірка прострочених гідрантів (щодня)	план перевірок (1-ше число місяця, 09:00)
5	контроль залишків (кожні 15 хв)	інвентаризація (п’ятниця, 16:00)

Питання 6.4. Події застосунку: ApplicationEvent, @EventListener

Теоретична частина

Коли виклик закрито, має статися кілька речей: автомобіль повертається в резерв, обласний центр отримує повідомлення, статистика оновлюється. Можна вписати все це в метод `changeStatus`, але тоді служба викликів знатиме і про автопарк, і про зовнішні системи, і про статистику. Кожна нова реакція — нова залежність і новий рядок у тому самому методі.

Події розв’язують цю проблему. Служба лише публікує факт: «виклик закрито» — через `ApplicationEventPublisher`. Хто і як на це реагує, вона не знає. Слухачі — методи з анотацією `@EventListener` в інших бінах [57]. Подією може бути будь-який об’єкт; зручно використовувати записи. Це реалізація патерну «Спостерігач» [16], і вона дає слабку зв’язаність без жодних черг повідомлень.

Важливо розуміти, як виконуються слухачі. За замовчуванням `@EventListener` викликається синхронно, у тому самому потоці і в тій самій транзакції, що й `publishEvent`. Якщо слухач кине виняток, транзакцію публікатора буде відкочено. Це зручно, коли реакція є частиною операції (звільнити автомобіль).

`@TransactionalEventListener` викликається лише після успішної фіксації транзакції (за замовчуванням фаза `AFTER_COMMIT`). Якщо транзакцію відкочено, слухач не спрацює взагалі. Це правильний вибір для зовнішніх дій — повідомлень, листів, викликів інших сервісів: не можна повідомляти обласний центр про закриття виклику, якого в базі так і не закрили. У поєднанні з `@Async` такий слухач ще й не затримує відповідь користувачу.

Атрибут `condition` дозволяє відфільтрувати події виразом SpEL. Події в межах однієї JVM не переживають перезапуску застосунку; якщо подію не можна втратити, використовують брокер повідомлень (розділ 9) або патерн transactional outbox [56].

Приклад задачі

Умова. Під час закриття виклику публікувати подію `CallClosedEvent`. Перший слухач у тій самій транзакції повертає автомобіль караулу в резерв (якщо караул призначався). Другий — асинхронно, після фіксації транзакції, повідомляє обласний центр. Показати кількість вільних автомобілів до і після закриття.

Лістинг 6.4. Реалізація з покроковим розбором

Файл `CallClosedEvent.java` подія

```
public record CallClosedEvent(long callId, Long crewId) {
}
```

Подія містить лише ідентифікатори, а не сутності: слухачі самі завантажать те, що їм потрібно. `crewId` може бути `null`, якщо караул не призначали.

Файл `DispatchService.java` публікація

```
call.setStatus(next);
if (next == CallStatus.CLOSED) {
    Long crewId = call.getCrew() == null
            ? null : call.getCrew().getId();
    events.publishEvent(new CallClosedEvent(id, crewId));
}
```

`ApplicationEventPublisher` впроваджено через конструктор, як і решту залежностей. Служба викликів нічого не знає про автомобілі й обласний центр.

Файл `CrewReleaseListener.java` два слухачі

```
@EventListener(condition = "#event.crewId() != null")
public void releaseTruck(CallClosedEvent event) {
    Crew crew = crews.findById(event.crewId()).orElseThrow();
    crew.getTruck().setAvailable(true);
    log.info("Виклик №{} закрито: {} повертається в резерв",
            event.callId(), crew.getTruck().getPlate());
}
```

Синхронний слухач працює в транзакції `changeStatus`. Зміну автомобіля Hibernate збереже під час її фіксації, без явного `save`. Умова відсікає виклики без караулу.

```
@Async
@TransactionalEventListener
public void reportToRegion(CallClosedEvent event) {
    log.info("Обласному центру: виклик №{} закрито",
            event.callId());
}
```

Спрацює лише після фіксації транзакції і в окремому віртуальному потоці.

Вивід програми результат запуску

```
INFO  DispatchService - Виклик №1: виїхав 1-й караул на ВС 0101 ДС
Вільних авто: 2
INFO  DispatchService - Виклик №1: DISPATCHED -> CLOSED
INFO  CrewReleaseListener - Виклик №1 закрито: ВС 0101 ДС повертається в резерв
INFO  CrewReleaseListener - Обласному центру: виклик №1 закрито
Вільних авто: 3
```

Зверніть увагу. *Якщо прибрати `@Transactional` з методу `changeStatus`, `@TransactionalEventListener` мовчки перестане спрацьовувати: немає транзакції — немає і моменту її фіксації. Spring лише запише попередження в лог. Атрибут `fallbackExecution = true` змушує слухача виконатися і без транзакції, але краще зрозуміти, чому транзакції немає.*

Варіанти для самостійної роботи

Завдання. Реалізувати для свого варіанта подію і два слухачі: синхронний (у транзакції публікатора) і `@TransactionalEventListener` з `@Async`. Показати, що другий слухач не спрацьовує, якщо транзакцію відкочено.

№	Подія	Синхронна реакція	Реакція після фіксації
1	рятувальника переведено	оновити кількість осіб у караулах	лист кадровій службі
2	автомобіль поставлено на ТО	вивести з бойового розрахунку	повідомлення механіку
3	наряд скасовано	звільнити дату в графіку	повідомлення курсанту
4	гідрант визнано несправним	позначити район як ризиковий	заявка до водоканалу
5	спорядження видано	зменшити залишок	запис у зовнішню облікову систему

ЧАСТИНА 4. БЕЗПЕКА І ЯКІСТЬ

Розділ 7. Spring Security

Досі будь-хто, хто знав адресу сервера, міг зареєструвати виклик, видалити запис із журналу чи прочитати статистику. Для системи, що працює з даними про надзвичайні ситуації та персональними даними заявників, це неприпустимо [29]. У цьому розділі застосунок отримує користувачів, ролі та токени. Навчальні облікові записи описано в міграції `V4__users.sql`; паролі там — лише для розробки.

Питання 7.1. Ланцюжок фільтрів (SecurityFilterChain)

Теоретична частина

Spring Security вбудовується у вебзастосунок не через контролери, а раніше — на рівні фільтрів сервлетів, ще до `DispatcherServlet` (див. шлях запиту в питанні 3.1). Фільтр `DelegatingFilterProxy` передає кожен запит у `FilterChainProxy`, а той — у перший `SecurityFilterChain`, що відповідає шляху запиту [58].

Кожен ланцюжок — це впорядкований набір фільтрів безпеки, і кожен відповідає за свою задачу: `CsrfFilter` перевіряє CSRF-токен, `BasicAuthenticationFilter` і `BearerTokenAuthenticationFilter` читають облікові дані із заголовка `Authorization`, `UsernamePasswordAuthenticationFilter` обробляє форму входу, `ExceptionTranslationFilter` перетворює помилки безпеки на 401 або перенаправлення на сторінку входу, а останній, `AuthorizationFilter`, вирішує, чи можна пропустити запит далі [59].

Щойно до проєкту додано стартер `spring-boot-starter-security`, Boot закриває всі адреси і створює одного користувача з випадковим паролем, який друкує в лог. Це безпечне значення за замовчуванням, але для реального застосунку ми описуємо власні ланцюжки біном `SecurityFilterChain`.

Ланцюжків може бути кілька. Типова ситуація — як у нас: REST API, де кожен запит несе токен і сесії не потрібні, і сторінки для людей з формою входу та сесією. Метод `securityMatcher` визначає, які запити обробляє ланцюжок, а анотація `@Order` — порядок перевірки. Перший ланцюжок, що «впізнав» запит, обробляє його повністю.

Усередині ланцюжка правила доступу `authorizeHttpRequests` перевіряються згори донизу, до першого збігу. Тому конкретні правила пишуть вище, а загальне `anyRequest()` — останнім.

Приклад задачі

Умова. Описати два ланцюжки фільтрів. Перший — для `/api/**`: без сесій, з автентифікацією через Basic (лише для отримання токена) і JWT. Другий — для решти адрес: форма входу, сесія, захист сторінок журналу. Публічними залишити статус чергування і стан застосунку.

Правила доступу за ролями і роботу з JWT розглянемо в наступних питаннях. Тут важливо побачити загальну структуру.

Лістинг 7.1. Реалізація з покроковим розбором

Файл `SecurityConfig.java` ланцюжок для API

```
@Bean
@Order(1)
SecurityFilterChain api(HttpSecurity http) throws Exception {
    http.securityMatcher("/api/**")
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/api/status").permitAll()
```

`@Order(1)` — цей ланцюжок перевіряється першим, але лише для адрес `/api/**`. Статус чергування доступний без входу.

```
        .httpBasic(Customizer.withDefaults())
        .oauth2ResourceServer(rs -> rs.jwt(jwt ->
                jwt.jwtAuthenticationConverter(rolesFromJwt())))
        .sessionManagement(s -> s.sessionCreationPolicy(
                SessionCreationPolicy.STATELESS))
        .csrf(csrf -> csrf.disable());
    return http.build();
}
```

Два способи автентифікації: логін і пароль у заголовку Basic та токен Bearer. Сесії не створюються. Чому тут можна вимкнути CSRF — у питанні 7.6.

Файл `SecurityConfig.java` ланцюжок для сторінок

```
@Bean
@Order(2)
SecurityFilterChain web(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests(auth -> auth
            .requestMatchers("/actuator/health/**", "/login",
                             "/error").permitAll()
```

Другий ланцюжок без `securityMatcher` обробляє все, що не потрапило в перший. Адреса `/error` відкрита свідомо — див. примітку.

```
            .requestMatchers("/journal/**")
                .hasAnyRole("DISPATCHER", "ADMIN")
            .requestMatchers("/swagger-ui/**",
                             "/v3/api-docs/**",
                             "/actuator/**").hasRole("ADMIN")
            .anyRequest().authenticated())
        .formLogin(Customizer.withDefaults());
    return http.build();
}
```

Журнал — лише для чергових і адміністратора. Документацію API та службові точки Actuator бачить тільки адміністратор (пригадайте застереження з питань 2.6 і 3.9).

Термінал запити без входу

```
curl -s -i localhost:8080/api/status
curl -s -i localhost:8080/api/calls
curl -s -i localhost:8080/journal
```

Вивід програми (заголовки скорочено)

```
Навчальна чергова частина (dev): чергування триває

HTTP/1.1 401
WWW-Authenticate: Basic realm="Realm", charset="UTF-8"

HTTP/1.1 302
Location: http://localhost:8080/login
```

З досвіду викладачів. *Спершу запит до `/api/calls` без входу повертав не 401, а 302 з перенаправленням на сторінку входу. Причина — пересилання на `/error`: коли фільтр безпеки встановив код 401, Tomcat передає запит на сторінку помилки, а її адреса не підпадає під `/api/**` і потрапляє в другий ланцюжок, який вимагає входу. Відкрита адреса `/error` розв'язує проблему. Якщо REST-клієнт раптом отримує HTML сторінки входу замість JSON, перевірте насамперед це.*

Варіанти для самостійної роботи

Завдання. Описати для застосунку свого варіанта два ланцюжки фільтрів: для API і для сторінок. Перевірити запитами `curl`, який ланцюжок обробляє кожну адресу і що повертається без автентифікації.

№	Префікс API	Публічні адреси	Сторінки лише для ролі
1	`/api/rescuers/**`	`/api/status`, `/actuator/health`	`/staff/**` — `HR`
2	`/api/trucks/**`	`/api/fleet/summary`	`/garage/**` — `MECHANIC`
3	`/api/cadets/**`	`/api/schedule/today`	`/office/**` — `COURSE_OFFICER`
4	`/api/hydrants/**`	`/api/map`	`/inspect/**` — `INSPECTOR`
5	`/api/equipment/**`	`/api/catalog`	`/stock/**` — `STOREKEEPER`

Питання 7.2. Автентифікація: UserDetailsService, зберігання паролів

Теоретична частина

Автентифікація відповідає на питання «хто ви?», авторизація — «що вам дозволено?». Їх часто плутають, а на співбесіді плутанина коштує дорого. Пам’ятайте також про коди: 401 Unauthorized насправді означає «не автентифіковано», 403 Forbidden — «автентифіковано, але не дозволено».

Spring Security отримує дані користувача через інтерфейс `UserDetailsService` з одним методом `loadUserByUsername` [59]. Він повертає `UserDetails`: ім’я, хеш пароля, повноваження (ролі), ознаки активності. Звідки береться користувач — з бази, LDAP чи файлу — вирішує наша реалізація. Далі `DaoAuthenticationProvider` порівнює введений пароль із хешем за допомогою `PasswordEncoder`.

Головне правило: паролі ніколи не зберігають у відкритому вигляді і не шифрують. Їх хешують спеціальними повільними функціями з сіллю — BCrypt, scrypt, Argon2 [31; 60]. Повільність тут — перевага: перебір мільйонів варіантів стає непрактичним. Сіль (випадкове значення, яке BCrypt зберігає просто в хеші) робить однакові паролі різними хешами і знецінює заздалегідь обчислені таблиці. Звичайні хеш-функції на кшталт SHA-256 для паролів не підходять: вони створені швидкими.

Хеш BCrypt виглядає так: `$2a$10$5tUO7...`. Тут `2a` — версія алгоритму, `10` — параметр вартості (2^{10} раундів), далі — сіль і сам хеш. Збільшення вартості на одиницю вдвічі сповільнює і перевірку, і перебір.

Повідомлення про помилку входу має бути однаковим для неіснуючого користувача і неправильного пароля. Інакше зловмисник зможе перебирати логіни, дізнаючись, які з них існують. Spring Security так і робить за замовчуванням.

Приклад задачі

Умова. Зберігати користувачів у таблиці `app_user` з хешами BCrypt і роллю. Реалізувати `DbUserDetailsService`, який завантажує користувача з бази, і бін `PasswordEncoder`. Перевірити вхід з правильним і неправильним паролем.

Лістинг 7.2. Реалізація з покроковим розбором

Файл `V4__users.sql` таблиця користувачів (фрагмент)

```
CREATE TABLE app_user (
    id            BIGSERIAL PRIMARY KEY,
    username      VARCHAR(50)  NOT NULL UNIQUE,
    password_hash VARCHAR(100) NOT NULL,
    role          VARCHAR(20)  NOT NULL,
    enabled       BOOLEAN      NOT NULL DEFAULT TRUE
);
```

Стовпець названо `password_hash`, а не `password`, — щоб ніхто не мав ілюзій щодо його вмісту. Ознака `enabled` дозволяє заблокувати обліковий запис, не видаляючи його.

Файл `DbUserDetailsService.java` завантаження користувача

```
@Override
public UserDetails loadUserByUsername(String username) {
    AppUser u = users.findByUsername(username)
            .orElseThrow(() ->
                    new UsernameNotFoundException(username));
    return User.withUsername(u.getUsername())
            .password(u.getPasswordHash())
            .roles(u.getRole())
            .disabled(!u.isEnabled())
            .build();
}
```

Перетворюємо власну сутність на стандартний `UserDetails`. Метод `roles("DISPATCHER")` сам додає префікс: повноваження називатиметься `ROLE_DISPATCHER`.

Файл `SecurityConfig.java` кодувальник паролів

```
@Bean
PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

Достатньо оголосити бін: Spring Security знайде і `UserDetailsService`, і `PasswordEncoder` та з'єднає їх сам.

Термінал вхід через Basic

```
curl -s -o /dev/null -w "%{http_code}\n" -u dispatcher:bad \
     -X POST localhost:8080/api/auth/token
curl -s -u dispatcher:dispatcher-2026 \
     -X POST localhost:8080/api/auth/token
```

Вивід програми результат запуску

```
401
{"token":"eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJkdXR5LWFwcCIsInN1YiI6...",
 "expiresIn":28800}
```

Порада. *Хеші для міграції ми згенерували один раз невеликою програмою з `new BCryptPasswordEncoder().encode(...)`. Запустіть її двічі з тим самим паролем — хеші будуть різні, але обидва пройдуть перевірку. Це наочно показує, навіщо в хеші зберігається сіль. І ще: навчальні паролі в міграції — свідомий компроміс для лабораторних. У робочій системі початкового адміністратора створюють окремим захищеним кроком, а не скриптом у репозиторії.*

Варіанти для самостійної роботи

Завдання. Реалізувати зберігання користувачів підсистеми свого варіанта в базі даних з хешуванням паролів і власним `UserDetailsService`. Додати вказану додаткову перевірку і продемонструвати вхід із правильними і неправильними даними.

№	Таблиця користувачів	Додаткова перевірка
1	`staff_user`	заблокувати після 5 невдалих спроб
2	`garage_user`	термін дії облікового запису (`accountExpired`)
3	`cadet_user`	вхід лише для курсантів поточного навчального року
4	`inspector_user`	вартість BCrypt 12; порівняти час перевірки з 10
5	`store_user`	вимога змінити пароль після першого входу

Питання 7.3. Авторизація: ролі, @PreAuthorize

Теоретична частина

Після автентифікації кожен користувач має набір повноважень (authorities) — рядків на кшталт `ROLE_DISPATCHER`. Роль у Spring Security — це просто повноваження з префіксом `ROLE_`. Тому `hasRole("DISPATCHER")` і `hasAuthority("ROLE_DISPATCHER")` перевіряють одне й те саме. Окрім ролей, можна використовувати дрібніші повноваження на кшталт `calls:delete` — це зручно, коли ролей багато, а права між ними перетинаються.

Перевіряти доступ можна на двох рівнях [59]. На рівні адрес — у `authorizeHttpRequests`: правила для шаблонів шляхів і HTTP-методів. Це зручно для грубого поділу: «читати можуть усі, змінювати — лише чергові». На рівні методів — анотацією `@PreAuthorize` з виразом SpEL. Її вмикає `@EnableMethodSecurity`. Перевірка відбувається перед викликом методу, і, як ви вже здогадалися, через проксі. Вираз може звертатися до параметрів методу: `@PreAuthorize("#username == authentication.name")` — користувач змінює лише власний профіль.

Обидва рівні доповнюють один одного. Правила адрес не дозволяють забути захистити новий контролер, а анотації на методах служб захищають логіку незалежно від того, звідки її викликано: з REST, зі сторінки чи із задачі за розкладом.

Для нашої чергової частини ролі такі. `DISPATCHER` — черговий: реєструє виклики, призначає караули. `RESCUER` — рятувальник: бачить виклики. `CADET` — курсант на стажуванні: бачить виклики. `ADMIN` — адміністратор: усе, включно з видаленням і службовими сторінками.

Принцип найменших привілеїв: кожен отримує рівно ті права, що потрібні для роботи. Курсанту на стажуванні не потрібно видаляти записи з журналу, навіть якщо «він обережний».

Приклад задачі

Умова. Задати правила доступу до API викликів: читання — для всіх ролей, зміна — для чергового й адміністратора. Видалення виклику дозволити лише адміністратору анотацією `@PreAuthorize`. Перевірити доступ рятувальника і чергового.

Лістинг 7.3. Реалізація з покроковим розбором

Файл `SecurityConfig.java` правила для адрес API

```
.requestMatchers(HttpMethod.GET, "/api/calls/**")
    .hasAnyRole("DISPATCHER", "RESCUER", "CADET", "ADMIN")
.requestMatchers("/api/calls/**")
    .hasAnyRole("DISPATCHER", "ADMIN")
.anyRequest().authenticated())
```

Порядок важливий: спершу правило для GET, потім для решти методів. Якби рядки помінялися місцями, рятувальник не зміг би навіть читати.

Файл `CallController.java` перевірка на методі

```
@DeleteMapping("/{id}")
@PreAuthorize("hasRole('ADMIN')")
public ResponseEntity<Void> delete(@PathVariable long id) {
```

Правило адрес пропускає DELETE для чергового, але анотація на методі його зупиняє. Видалення запису з журналу — дія адміністратора.

Термінал рятувальник і черговий

```
curl -s -o /dev/null -w "%{http_code}\n" \
     -H "Authorization: Bearer $RESCUER" localhost:8080/api/calls/1
curl -s -o /dev/null -w "%{http_code}\n" -X POST \
     -H "Authorization: Bearer $RESCUER" -H "Content-Type: application/json" \
     -d '{"address":"вул. Зелена, 101","type":"FIRE"}' localhost:8080/api/calls
curl -s -o /dev/null -w "%{http_code}\n" -X DELETE \
     -H "Authorization: Bearer $DISPATCHER" localhost:8080/api/calls/1
```

Вивід програми результат запуску

```
200
403
403
```

Зверніть увагу. *У відповідях 403 немає жодного пояснення, і це правильно: клієнтові досить знати, що дію заборонено. Але для чергового «нічого не сталося» — погана підказка. На сторінках варто просто не показувати кнопок, недоступних для ролі. Thymeleaf має для цього модуль `thymeleaf-extras-springsecurity6` з атрибутом `sec:authorize`.*

Варіанти для самостійної роботи

Завдання. Задати для API свого варіанта правила доступу за ролями на рівні адрес і на рівні методів. Продемонструвати відповіді 200 і 403 для кожної ролі.

№	Ролі	Особливе правило з `@PreAuthorize`
1	`HR`, `COMMANDER`, `RESCUER`	рятувальник бачить лише власний профіль
2	`MECHANIC`, `COMMANDER`	списання автомобіля — лише `COMMANDER`
3	`COURSE_OFFICER`, `CADET`	курсант бачить лише власні наряди
4	`INSPECTOR`, `ADMIN`	інспектор змінює лише гідранти свого району
5	`STOREKEEPER`, `COMMANDER`, `RESCUER`	видача понад 10 одиниць — лише `COMMANDER`

Питання 7.4. JWT і stateless-автентифікація

Теоретична частина

Класичний вебзастосунок пам’ятає користувача через сесію: після входу сервер створює запис у пам’яті, а браузер отримує cookie `JSESSIONID`. Для REST API, з яким працюють мобільні застосунки, табло та інші сервіси, такий підхід незручний: сесію треба десь зберігати, синхронізувати між екземплярами сервера, а клієнти не завжди вміють працювати з cookies.

Альтернатива — автентифікація без стану (stateless) за допомогою токенів. Найпоширеніший формат — JWT (JSON Web Token) [61]. Токен складається з трьох частин, закодованих Base64URL і розділених крапками: заголовок (алгоритм підпису), корисне навантаження (claims: хто, коли видано, коли спливає, ролі) і підпис. Клієнт надсилає токен у кожному запиті в заголовку `Authorization: Bearer <токен>`, а сервер перевіряє підпис і термін дії — без звернень до бази чи сесії.

Важливі наслідки, про які питають на співбесідах:

JWT підписаний, але не зашифрований. Будь-хто може прочитати його вміст, тому паролі чи персональні дані в токен класти не можна.

Відкликати виданий токен до закінчення терміну складно: сервер ніде не зберігає список «живих» токенів. Тому термін дії роблять коротким (хвилини або години), а для тривалої роботи використовують окремі токени оновлення (refresh tokens).

Ключ підпису — найцінніший секрет системи. Хто знає ключ, той може видати собі токен адміністратора. Він ніколи не потрапляє в репозиторій і передається змінною середовища.

Spring Security має повноцінну підтримку JWT у модулі OAuth2 Resource Server [59]. Ми використовуємо симетричний підпис HS256 зі спільним секретом: застосунок і видає, і перевіряє токени. Коли токени видає окремий сервер авторизації, використовують асиметричні ключі (RS256): сервер підписує закритим ключем, а застосунки перевіряють відкритим (питання 7.5).

Приклад задачі

Умова. Реалізувати видачу JWT: користувач автентифікується логіном і паролем через Basic на адресі `/api/auth/token` і отримує токен з ролями, дійсний вісім годин. Налаштувати перевірку токенів для решти API і прочитати вміст виданого токена.

Лістинг 7.4. Реалізація з покроковим розбором

Файл `SecurityConfig.java` ключ, кодувальник і декодер

```
@Bean
SecretKey jwtKey(@Value("${duty.jwt.secret}") String secret) {
    byte[] bytes = secret.getBytes(StandardCharsets.UTF_8);
    return new SecretKeySpec(bytes, "HmacSHA256");
}
```

Секрет береться з властивості, яка, своєю чергою, читає змінну середовища `JWT_SECRET`. Значення в `application.yml` — лише для розробки. Для HS256 ключ має бути не коротшим за 32 байти.

```
@Bean
JwtEncoder jwtEncoder(SecretKey key) {
    return new NimbusJwtEncoder(new ImmutableSecret<>(key));
}

@Bean
JwtDecoder jwtDecoder(SecretKey key) {
    return NimbusJwtDecoder.withSecretKey(key)
            .macAlgorithm(MacAlgorithm.HS256).build();
}
```

Обидва біни використовують той самий ключ. Декодер підхопить `oauth2ResourceServer` з питання 7.1.

Файл `TokenController.java` видача токена

```
List<String> roles = auth.getAuthorities().stream()
        .map(GrantedAuthority::getAuthority)
        .filter(a -> a.startsWith("ROLE_"))
        .map(a -> a.substring("ROLE_".length()))
        .toList();
```

У токен кладемо назви ролей без префікса. Фільтр `startsWith` з'явився після тестування — див. примітку.

```
JwtClaimsSet claims = JwtClaimsSet.builder()
        .issuer("duty-app")
        .subject(auth.getName())
        .issuedAt(now)
        .expiresAt(now.plus(TTL))
        .claim("roles", roles)
        .build();
```

Стандартні поля: видавець (`iss`), суб'єкт (`sub`), час видачі (`iat`) і закінчення (`exp`). Власне поле `roles` перетворюється назад на повноваження `ROLE_…` конвертером `rolesFromJwt` у `SecurityConfig`.

Термінал отримати токен і розкодувати його середню частину

```
TOKEN=$(curl -s -u dispatcher:dispatcher-2026 -X POST \
        localhost:8080/api/auth/token | jq -r .token)
echo $TOKEN | cut -d. -f2 | base64 -d
```

Вивід програми результат запуску

```
{"iss":"duty-app","sub":"dispatcher","exp":1792774736,
 "iat":1792745936,"roles":["DISPATCHER"]}
```

З досвіду викладачів. *У першій версії в токені, крім `DISPATCHER`, з'явилася дивна «роль» `R_PASSWORD`. Виявилося, що Spring Security 7 додає до автентифікованого користувача службове повноваження `FACTOR_PASSWORD` — позначку того, що вхід виконано паролем (основа для багатофакторної автентифікації). Наш код зрізав перші п'ять символів у всіх повноважень підряд. Мораль: не робіть припущень про вміст списку повноважень, фільтруйте явно.*

Варіанти для самостійної роботи

Завдання. Реалізувати видачу і перевірку JWT для API свого варіанта. Додати до токена вказане власне поле і використати його в коді. Продемонструвати відповідь на прострочений токен і токен з підробленим підписом.

№	Термін дії	Власне поле в токені	Використання поля
1	1 год	`crewId`	рятувальник бачить лише свій караул
2	30 хв	`garage`	механік бачить лише машини свого гаража
3	4 год	`platoon`	курсант бачить графік лише свого взводу
4	8 год	`district`	інспектор бачить гідранти свого району
5	15 хв	`warehouse`	комірник працює лише зі своїм складом

Питання 7.5. OAuth2 і OpenID Connect: загальна схема

Теоретична частина

У питанні 7.4 застосунок сам перевіряв пароль і сам видавав токен. Для одного застосунку це прийнятно. Але в університеті, як і в будь-якій організації, систем багато: журнал чергової частини, електронний деканат, пошта, бібліотека. Якщо кожна зберігатиме власні паролі, користувачі матимуть десяток облікових записів, а адміністратори — десяток місць, де можна забути заблокувати звільненого співробітника.

OAuth 2.0 — протокол делегованої авторизації [62]. Він описує ролі: власник ресурсу (користувач), клієнт (застосунок, що хоче доступу), сервер авторизації (видає токени) і сервер ресурсів (API, що перевіряє токени). Сам OAuth 2.0 відповідає на питання «що клієнтові дозволено», але не «хто користувач». Цю частину додає OpenID Connect (OIDC) — шар автентифікації поверх OAuth 2.0 з ID-токеном і стандартними даними про користувача [63].

Основний сценарій для вебзастосунків — Authorization Code Flow (для односторінкових і мобільних застосунків — з розширенням PKCE):

1) користувач натискає «Увійти», і застосунок перенаправляє його на сервер авторизації;

2) користувач вводить пароль там, а не в нашому застосунку (і, можливо, підтверджує другий фактор);

3) сервер авторизації повертає браузер назад з одноразовим кодом;

4) застосунок обмінює код на токени — запит іде від сервера до сервера, з секретом клієнта;

5) з токеном доступу застосунок звертається до API.

Сервер авторизації — це окремий продукт: Keycloak (відкритий, популярний в університетах і державних установах), Microsoft Entra ID, Google, Okta. Писати власний сервер авторизації для навчального проєкту не потрібно, та й для робочого — лише з дуже вагомих причин.

Spring Security підтримує обидві ролі. `oauth2Login` робить застосунок клієнтом OIDC (кнопка «Увійти через …»). `oauth2ResourceServer` — сервером ресурсів, що перевіряє токени. Наш API вже є сервером ресурсів, і перейти на зовнішній сервер авторизації можна, змінивши лише налаштування: замість власного секрету вказати адресу видавця (`issuer-uri`), звідки Spring завантажить відкриті ключі для перевірки підпису.

Приклад задачі

Умова. Підготувати профіль `oidc`, у якому API чергової частини перевіряє токени, видані зовнішнім сервером авторизації університету (наприклад, Keycloak), і порівняти потоки даних із власною видачею токенів з питання 7.4.

Розгортання Keycloak виходить за межі посібника. Схему нижче варто вміти відтворити на дошці під час співбесіди.

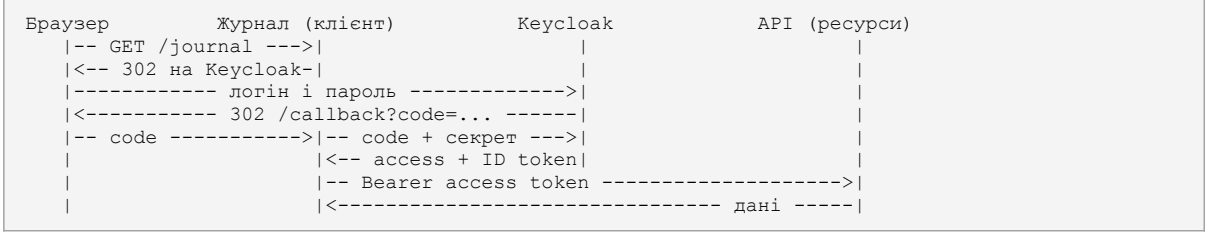
Лістинг 7.5. Реалізація з покроковим розбором

Файл `application-oidc.yml` перевірка токенів зовнішнього видавця

```yaml
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://sso.ldubgd.example/realms/duty
```

Адреса умовна. За нею Spring знайде документ `/.well-known/openid-configuration` і набір відкритих ключів. У такому разі власні біни `JwtEncoder`, `JwtDecoder` і контролер видачі токенів з питання 7.4 з проєкту прибирають.

Схема Authorization Code Flow для журналу чергової частини



Порада. *На співбесіді часто запитують: «Чим OAuth2 відрізняється від OpenID Connect?». Коротка відповідь: OAuth2 — про авторизацію (токен доступу до API), OIDC — про автентифікацію (ID-токен з даними про користувача) поверх OAuth2. І ще одне: потік Implicit і передача пароля клієнтові (Resource Owner Password) вважаються застарілими — не пропонуйте їх.*

Варіанти для самостійної роботи

Завдання. Описати для підсистеми свого варіанта схему автентифікації через зовнішній сервер авторизації: учасників, потік, токени, що передаються, і налаштування Spring Security. За можливості розгорнути Keycloak у Docker і перевірити роботу (для виконання знадобиться розділ 10).

№	Клієнт	Потік	Особливість
1	мобільний застосунок рятувальника	Authorization Code + PKCE	токен оновлення
2	табло в залі чергової частини	Client Credentials	без користувача
3	вебкабінет курсанта	Authorization Code	вхід через обліковий запис університету
4	планшет інспектора	Authorization Code + PKCE	робота без мережі до 8 год
5	інтеграція з обліковою системою складу	Client Credentials	обмежений `scope`

Питання 7.6. CSRF, CORS і поширені вразливості з OWASP Top 10

Теоретична частина

OWASP Top 10 — перелік найпоширеніших і найнебезпечніших вразливостей вебзастосунків, який регулярно оновлює спільнота OWASP [31]. Junior-розробник не повинен знати всі методи атак, але має розуміти, від чого захищається його код. Ось як основні пункти переліку стосуються нашого застосунку:

порушення контролю доступу — правила ролей і `@PreAuthorize` (питання 7.3);

криптографічні збої — хешування паролів BCrypt, секрети в змінних середовища (7.2, 7.4);

ін'єкції — параметризовані запити Spring Data замість склеювання рядків (3.8, 5.2), екранування в Thymeleaf проти XSS (4.1);

небезпечна конфігурація — закриті службові точки Actuator і Swagger (2.6, 7.1);

уразливі компоненти — актуальні версії Spring Boot і залежностей (розділ 10);

збої автентифікації — однакові повідомлення для неправильного логіна і пароля, короткий термін дії токенів (7.2, 7.4).

Окремо розглянемо CSRF (Cross-Site Request Forgery). Атака виглядає так: черговий увійшов у журнал, браузер зберігає cookie сесії. В іншій вкладці він відкриває сторінку зловмисника, де є прихована форма, що надсилає POST на `/journal`. Браузер автоматично додасть cookie сесії — і в журналі з'явиться фальшивий виклик. Захист — CSRF-токен: сервер вимагає в кожній формі приховане поле з випадковим значенням, якого сторінка зловмисника знати не може. Spring Security вмикає цей захист за замовчуванням, а Thymeleaf сам додає токен у форми з `th:action`.

Чому ж для API ми CSRF вимкнули? Тому що API не використовує cookies: токен Bearer браузер автоматично не додає, тож підробити такий запит з чужої сторінки неможливо. Правило просте: якщо автентифікація спирається на cookies, захист від CSRF обов'язковий.

CORS ми налаштували в питанні 4.3. Нагадаємо: CORS — це послаблення політики браузера, а не захист сервера. Неправильно налаштований CORS (`*` разом із cookies) відкриває саме ту діру, яку закриває CSRF-захист.

Spring Security також за замовчуванням додає заголовки безпеки: `X-Content-Type-Options: nosniff`, `X-Frame-Options: DENY` (захист від вбудовування сторінки у фрейм — clickjacking), `Cache-Control: no-store` для захищених відповідей.

Приклад задачі

Умова. Показати роботу захисту від CSRF на сторінках журналу: запит від автентифікованого чергового без токена відхиляється, а форма, згенерована Thymeleaf, містить токен. Перевірити заголовки безпеки у відповіді.

Щоб відтворити атаку без браузера, увійдемо через форму входу за допомогою `curl`, зберігши cookie сесії у файлі.

Лістинг 7.6. Реалізація з покроковим розбором

Термінал вхід, збереження сесії і запит без CSRF-токена

```
curl -s -c jar.txt -b jar.txt localhost:8080/login > login.html
curl -s -c jar.txt -b jar.txt -d "username=dispatcher" \
     -d "password=dispatcher-2026" -d "_csrf=<токен зі сторінки>" \
     localhost:8080/login
curl -s -i -b jar.txt -d "address=вул. Личаківська, 8&type=RESCUE" \
     localhost:8080/journal
```

Вивід програми результат запуску

```
HTTP/1.1 403
```

Термінал токен у формі і заголовки безпеки

```
curl -s -b jar.txt localhost:8080/journal/new | grep _csrf
curl -s -i localhost:8080/api/status
```

Вивід програми (фрагменти)

```
<input type="hidden" name="_csrf" value="bl5VpKmGsbdH_75kSRpmjOZ_...">

X-Content-Type-Options: nosniff
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
X-Frame-Options: DENY
```

Зверніть увагу. *У цьому прикладі немає жодного рядка нового коду. Усе показане — поведінка Spring Security за замовчуванням, і найкраще, що може зробити розробник, — не вимикати її «щоб запрацювало». Якщо ви ловите себе на тому, що пишете* `.csrf(csrf -> csrf.disable())` *для сторінок з формами, зупиніться й подумайте, чому форма не надсилає токен.*

Варіанти для самостійної роботи

Завдання. Перевірити застосунок свого варіанта на вказані вразливості з OWASP Top 10. Для кожної описати сценарій атаки, продемонструвати захист (або вразливість і її виправлення) і пояснити, який механізм Spring відповідає за захист.

№	Вразливості для перевірки
1	CSRF на формі переведення рятувальника; XSS у полі ПІБ
2	SQL-ін’єкція в параметрі сортування; відкрита точка `/actuator/env`
3	IDOR: курсант змінює ідентифікатор у URL і бачить чужий наряд
4	clickjacking сторінки інспектора; відсутність обмеження спроб входу
5	перегляд закупівельних цін через API без ролі; секрет JWT у репозиторії

Питання 7.7. Отримання поточного користувача; аудит дій

Теоретична частина

Після автентифікації Spring Security зберігає дані про користувача в `SecurityContext`, прив’язаному до поточного потоку. Дістати їх можна кількома способами [59]:

у контролері — параметром з анотацією `@AuthenticationPrincipal` (отримаємо `UserDetails`, `Jwt` або іншого «принципала», залежно від способу входу) або параметром `Authentication`;

у будь-якому іншому місці — статичним методом `SecurityContextHolder.getContext().getAuthentication()`.

Другий спосіб зручний, але створює приховану залежність від глобального стану. Тому в службах краще передавати потрібні дані явно, а `SecurityContextHolder` використовувати в інфраструктурному коді — наприклад, в аудиті.

Пригадайте `AuditorAware` з питання 5.8: досі він повертав умовного «чергового». Тепер можна повертати справжнє ім’я користувача з контексту безпеки. Для дій без користувача (задачі за розкладом, слухачі подій у фонових потоках) варто повертати зрозуміле значення на кшталт «система», а не `null`.

Ще одна деталь: контекст безпеки прив’язаний до потоку, тож в асинхронних методах (питання 6.2) його за замовчуванням немає. Якщо асинхронній задачі потрібен користувач, контекст передають явно або використовують `DelegatingSecurityContextAsyncTaskExecutor`.

Аудит у системі цивільного захисту — не формальність. Під час розбору надзвичайної ситуації питання «хто і коли зареєстрував виклик, хто призначив караул» постають обов'язково. Тому ім'я користувача має потрапляти в журнал автоматично, без жодних зусиль з боку розробника, який пише нову функцію.

Приклад задачі

Умова. Додати кінцеву точку `/api/me`, що повертає ім'я користувача, ролі та термін дії токена. Переписати `AuditorAware` так, щоб у полі `created_by` викликів зберігалося ім'я користувача, який їх зареєстрував.

Лістинг 7.7. Реалізація з покроковим розбором

Файл `MeController.java` поточний користувач у контролері

```
@GetMapping("/api/me")
public Map<String, Object> me(
        @AuthenticationPrincipal Jwt jwt) {
    return Map.of("username", jwt.getSubject(),
                  "roles", jwt.getClaimAsStringList("roles"),
                  "expiresAt", jwt.getExpiresAt().toString());
}
```

Для входу за токеном «принципал» — це сам розібраний `Jwt`. Клієнт (наприклад, табло чи мобільний застосунок) може дізнатися, чий токен у нього і коли його треба оновити.

Файл `JpaConfig.java` аудит з контексту безпеки

```
@Bean
public AuditorAware<String> auditorAware() {
    return () -> Optional.ofNullable(SecurityContextHolder
                    .getContext().getAuthentication())
            .filter(Authentication::isAuthenticated)
            .map(Authentication::getName)
            .or(() -> Optional.of("система"));
}
```

Якщо користувача немає (запуск із задачі за розкладом чи з демо-класу), записуємо «система». Сутність `FireCall` і служби при цьому не змінилися зовсім.

Термінал хто я, і хто зареєстрував виклики

```
curl -s -H "Authorization: Bearer $TOKEN" localhost:8080/api/me
psql -U duty -d duty -c "select id, address, created_by from fire_call"
```

Вивід програми результат запуску

```
{"username":"dispatcher","expiresAt":"2026-10-23T16:58:56Z",
 "roles":["DISPATCHER"]}

 id |       address       | created_by
----+---------------------+------------
  1 | вул. Зелена, 101    | dispatcher
  2 | вул. Личаківська, 8 | dispatcher
```

Порада. *Перший виклик зареєстровано через REST API з токеном, другий — через форму журналу із сесією. В обох випадках `created_by` заповнився правильно, бо аудит спирається на `SecurityContext`, а не на спосіб входу. Саме тому інфраструктурні речі варто будувати на абстракціях Spring Security, а не, скажімо, на розборі заголовка `Authorization` вручну.*

Варіанти для самостійної роботи

Завдання. Реалізувати для застосунку свого варіанта кінцеву точку з даними поточного користувача і аудит дій з іменем користувача. Додати вказану перевірку і продемонструвати роботу для двох різних користувачів.

№	Аудит якої дії	Додатково
1	зміна караулу рятувальника	хто змінив — у таблиці історії
2	постановка автомобіля на ТО	заборонити механіку змінювати власні записи пізніше ніж через добу
3	зміна наряду	курсант бачить, хто з офіцерів змінив його наряд
4	результат перевірки гідранта	`@LastModifiedBy` і `@LastModifiedDate`
5	видача спорядження	ім’я користувача в асинхронному повідомленні (передати контекст)

Розділ 8. Тестування

У попередніх розділах ми перевіряли кожен приклад вручну: запускали демо, надсилали `curl`, дивилися в лог. Для навчання це добре, але кожна зміна коду вимагала б повторити всі ці перевірки. Автоматичні тести роблять це за нас за лічені секунди, і на співбесіді Junior-розробника про них питають обов’язково. Тести лежать у каталозі `src/test/java` проєкту і запускаються командою `mvn verify`.

Питання 8.1. Модульні тести: JUnit 5 і Mockito

Теоретична частина

Модульний тест (unit test) перевіряє один клас окремо від решти системи: без бази даних, мережі, файлової системи і без контейнера Spring. Такі тести виконуються за мілісекунди, тож їх можна мати сотні й запускати після кожної зміни [64].

JUnit — стандартний фреймворк тестування для Java [65]. Тестовий метод позначають `@Test`, підготовку перед кожним тестом — `@BeforeEach`. Параметризовані тести (`@ParameterizedTest` з `@CsvSource`, `@ValueSource`, `@MethodSource`) запускають один і той самий метод з різними даними — зручно для таблиць допустимих значень. Для перевірок ми використовуємо бібліотеку AssertJ: `assertThat(x).isEqualTo(y)` читається майже як звичайне речення і дає зрозумілі повідомлення про помилки.

Але служба рідко працює сама: `DispatchService` потребує репозиторіїв, сповіщувача, публікатора подій. Тут допомагає Mockito [66]. Мок (mock) — об’єкт-замінник, створений з інтерфейсу або класу. За замовчуванням його методи нічого не роблять і повертають «порожні» значення. Поведінку задають через `when(...).thenReturn(...)`, а виклики перевіряють через `verify(...)`. `ArgumentCaptor` дозволяє «перехопити» аргумент виклику і перевірити його вміст.

Ось де окупається впровадження залежностей через конструктор з питання 1.3. Службу ми створюємо звичайним `new`, передаючи моки і фіксований `Clock`, — і отримуємо повністю передбачуване середовище.

Структуру тесту зручно тримати за схемою Arrange — Act — Assert (підготувати, виконати, перевірити). Назва тесту має описувати поведінку: `illegalTransitionIsRejected` краще, ніж `testChangeStatus2`. Один тест — одна поведінка. І останнє: не мокайте все підряд. Записи, переліки і прості об’єкти без зовнішніх залежностей (як `FireCall` чи `CallStatus`) використовуйте справжні.

Приклад задачі

Умова. Покрити модульними тестами правила зміни статусу виклику (параметризованим тестом) і службу `DispatchService`: реєстрація виклику з фіксованим часом і сповіщенням, публікація події під час закриття, відмова при недопустимому переході, виняток для неіснуючого виклику.

Лістинг 8.1. Реалізація з покроковим розбором

Файл `CallStatusTest.java` параметризований тест

```
@ParameterizedTest(name = "{0} -> {1}: {2}")
@CsvSource({
        "NEW,        DISPATCHED, true",
        "DISPATCHED, ON_SITE,    true",
        "NEW,        CLOSED,     true",
        "NEW,        ON_SITE,    false",
        "DISPATCHED, NEW,        false",
        "CLOSED,     CLOSED,     false"
})
void transitions(CallStatus from, CallStatus to,
                 boolean allowed) {
    assertThat(from.canMoveTo(to)).isEqualTo(allowed);
}
```

Шість випадків в одному методі. JUnit сам перетворить рядки на константи переліку і `boolean`, а атрибут `name` зробить звіт зрозумілим: «NEW -> ON_SITE: false».

Файл `DispatchServiceTest.java` моки і фіксований годинник

```
@ExtendWith(MockitoExtension.class)
class DispatchServiceTest {
    @Mock CallRepository calls;
    @Mock CrewRepository crews;
    @Mock Notifier notifier;
    @Mock ApplicationEventPublisher events;
```

`MockitoExtension` створює моки для полів з `@Mock`. Жодного Spring, жодної бази.

```
@Test
void acceptSavesCallAndNotifies() {
    when(calls.save(any()))
            .thenAnswer(inv -> inv.getArgument(0));

    FireCall call = service.accept("вул. Зелена, 101",
            CallType.FIRE, null);

    assertThat(call.getStatus()).isEqualTo(CallStatus.NEW);
    assertThat(call.getReceivedAt())
            .isEqualTo(LocalDateTime.of(2026, 9, 1, 9, 0));
    verify(notifier).send(contains("вул. Зелена, 101"));
}
```

Мок репозиторію повертає той самий об’єкт, що отримав. Час перевіряємо точно — завдяки `Clock.fixed`. `verify` переконується, що сповіщення надіслано і містить адресу.

```
@Test
void closingPublishesEvent() {
    FireCall call = new FireCall("вул. Пасічна, 62",
            CallType.FIRE, LocalDateTime.now(clock), null);
    when(calls.findById(7L)).thenReturn(Optional.of(call));

    service.changeStatus(7L, CallStatus.CLOSED);

    var event = ArgumentCaptor.forClass(CallClosedEvent.class);
    verify(events).publishEvent(event.capture());
    assertThat(event.getValue().callId()).isEqualTo(7L);
}
```

`ArgumentCaptor` перехоплює подію, передану публікатору, і ми перевіряємо її вміст.

```
@Test
void illegalTransitionIsRejected() {
    FireCall call = new FireCall("вул. Стрийська, 1",
            CallType.RESCUE, LocalDateTime.now(clock), null);
    call.setStatus(CallStatus.DISPATCHED);
    when(calls.findById(1L)).thenReturn(Optional.of(call));

    assertThatThrownBy(
            () -> service.changeStatus(1L, CallStatus.NEW))
            .isInstanceOf(IllegalStatusChangeException.class)
            .hasMessageContaining("DISPATCHED на NEW");
    verifyNoInteractions(events);
}
```

Перевіряємо і виняток, і те, чого не повинно статися: жодної події при відмові.

Термінал запуск модульних тестів

```
mvn test -Dtest='CallStatusTest,DispatchServiceTest'
```

Вивід програми результат запуску

```
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0 -- in ...CallStatusTest
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0 -- in ...DispatchServiceTest
```

Порада. *Перевірте, чи тест справді щось перевіряє: тимчасово «зламайте» код, наприклад, приберіть умову* `this != CLOSED` *у* `canMoveTo`*. Якщо всі тести залишилися зеленими — тест нічого не ловить. Радимо робити так з кожним новим тестом: несподівано часто виявляється, що «тест» насправді нічого не перевіряє.*

Варіанти для самостійної роботи

Завдання. Написати модульні тести для служби свого варіанта з моками залежностей і фіксованим `Clock`: параметризований тест бізнес-правила, тест успішного сценарію з `verify`, тест винятку з `assertThatThrownBy`, тест з `ArgumentCaptor`.

№	Служба	Бізнес-правило для параметризованого тесту
1	`ShiftService`	у зміні 4–6 рятувальників, серед них хоча б один офіцер
2	`FleetService`	ТО кожні 10 000 км або 6 місяців
3	`CadetDutyPlanner`	не більше двох нарядів на тиждень, не в день іспиту
4	`HydrantChecker`	тиск 0,15–0,6 МПа — справний, інакше — ні
5	`EquipmentStock`	видача можлива, якщо залишок після неї ≥ мінімуму

Питання 8.2. @SpringBootTest і зрізи контексту (@WebMvcTest, @DataJpaTest)

Теоретична частина

Модульні тести не перевіряють, як класи працюють разом: чи правильно налаштовано біни, транзакції, слухачі подій, міграції. Для цього потрібні інтеграційні тести з контейнером Spring [67].

`@SpringBootTest` піднімає повний контекст застосунку — так само, як `SpringApplication.run`. Біни впроваджуються в тест через `@Autowired`. Це найповніша, але й найповільніша перевірка: старт триває секунди, а для бази даних потрібен справжній PostgreSQL (питання 8.4). Spring кешує контекст між тестовими класами з однаковою конфігурацією, тому дорого коштує лише перший старт.

Між модульним тестом і повним контекстом є проміжний рівень — зрізи (test slices). Кожна анотація-зріз піднімає лише частину застосунку [21]:

`@WebMvcTest(Контролер.class)` — лише вебрівень: контролери, `@ControllerAdvice`, конвертери, фільтри; служби підміняються моками;

`@DataJpaTest` — лише JPA: сутності, репозиторії, Flyway; кожен тест виконується в транзакції, яку наприкінці відкочено;

`@JsonTest`, `@RestClientTest` та інші — для вужчих задач.

Бін, якого немає в зрізі, підміняють анотацією `@MockitoBean` (у старих версіях — `@MockBean`): Spring кладе в контекст мок Mockito замість справжнього біна. Додаткові класи до зрізу додають через `@Import`.

У Spring Boot 4 тестова підтримка розбита на модулі, як і решта: для `@WebMvcTest` потрібен `spring-boot-starter-webmvc-test`, для `@DataJpaTest` — `spring-boot-starter-data-jpa-test`. Пакети анотацій теж змінилися: наприклад, `org.springframework.boot.webmvc.test.autoconfigure.WebMvcTest` [4].

Зверніть увагу на назви: класи `*Test` запускає плагін Surefire на фазі `test`, класи `*IT` (integration test) — плагін Failsafe на фазі `integration-test`. Повільні тести відокремлюють, щоб швидкий цикл «змінив — перевірив» не чекав на них.

Приклад задачі

Умова. Написати інтеграційний тест з повним контекстом, що перевіряє ланцюжок із розділів 5 і 6: призначення караулу знімає автомобіль з резерву, а закриття виклику через подію повертає його. Підключити тестові стартери Spring Boot 4 і Failsafe для класів `*IT`.

Лістинг 8.2. Реалізація з покроковим розбором

Файл `pom.xml` тестові стартери

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webmvc-test</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa-test</artifactId>
    <scope>test</scope>
</dependency>
```

Усі тестові залежності мають `scope test` і не потрапляють у зібраний jar. Ще три — для Spring Security і Testcontainers — побудовано так само.

Файл `pom.xml` Failsafe для інтеграційних тестів

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <executions>
        <execution>
            <goals>
                <goal>integration-test</goal>
                <goal>verify</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Версію плагіна задає батьківський POM. Тепер `mvn verify` запускає і `*Test`, і `*IT`, а `mvn test` — лише швидкі.

Файл `DutyApplicationIT.java` тест з повним контекстом

```
@SpringBootTest
@Import(PostgresContainerConfig.class)
class DutyApplicationIT {
    @Autowired DispatchService service;
    @Autowired TruckRepository trucks;
```

Справжні біни зі справжніми транзакціями, подіями і базою даних. Конфігурацію бази з контейнером розглянемо в питанні 8.4.

```
@Test
void closingCallReturnsTruckToReserve() {
    long id = service.accept("вул. Пасічна, 62",
            CallType.FIRE, null).getId();
    service.assignCrew(id, 2);
    assertThat(truckAvailable(2)).isFalse();

    service.changeStatus(id, CallStatus.CLOSED);

    assertThat(truckAvailable(2)).isTrue();
}

private boolean truckAvailable(long id) {
    return trucks.findById(id).orElseThrow().isAvailable();
}
```

Тест перевіряє одразу кілька механізмів: транзакції служби, оптимістичне блокування автомобіля, публікацію події і синхронного слухача. Модульний тест такого не покаже.

Вивід програми mvn verify (фрагмент)

```
[INFO] Tests run: 20, Failures: 0, Errors: 0, Skipped: 0
...
[INFO] --- failsafe:3.5.6:integration-test (default) @ duty-app ---
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO] BUILD SUCCESS
```

Зверніть увагу. *Тест з повним контекстом не відкочує змін автоматично, на відміну від `@DataJpaTest`. Після нього в базі лишається зареєстрований виклик. Тут це не заважає, але якщо інший тест рахує виклики, результат залежатиме від порядку запуску. Тести мають бути незалежними: або очищайте дані в `@AfterEach`, або позначайте тест `@Transactional`, або (найнадійніше) перевіряйте лише ті дані, які тест створив сам.*

Варіанти для самостійної роботи

Завдання. Написати для застосунку свого варіанта один тест `@SpringBootTest`, що перевіряє взаємодію кількох компонентів, і пояснити, чому цей сценарій не можна перевірити модульним тестом. Налаштувати запуск через Failsafe.

№	Сценарій інтеграційного тесту
1	переведення рятувальника оновлює склад обох караулів (подія + слухач)
2	постановка автомобіля на ТО прибирає його з кешу довідника
3	формування графіка нарядів відкочується повністю в разі конфлікту дат
4	результат перевірки гідранта записує аудит з іменем інспектора
5	видача спорядження зменшує залишок і публікує подію після фіксації

Питання 8.3. MockMvc і тестування REST

Теоретична частина

Як перевірити контролер, не запускаючи Tomcat і не надсилаючи справжніх HTTP-запитів? Для цього є `MockMvc` — імітація HTTP-запиту, яка проходить через `DispatcherServlet`, фільтри, конвертери JSON, валідацію і обробники винятків, але без мережі [67]. Тест виконується за мілісекунди й перевіряє саме те, що бачитиме клієнт: код стану, заголовки, тіло відповіді.

Запит будують статичними методами `get`, `post`, `put`, `delete`, додаючи тип вмісту, тіло й заголовки. Перевірки — `andExpect(status().isOk())`, `header()`, `content()`, `jsonPath(...)`. Вирази JsonPath (`$.errors.address`) дозволяють перевірити окреме поле JSON, не розбираючи всю відповідь.

Найчастіше `MockMvc` використовують разом зі зрізом `@WebMvcTest`: контролер справжній, служби — моки. Так ми перевіряємо саме вебрівень: маппінг адрес, перетворення JSON, валідацію, коди відповіді, обробку помилок. Логіку служб уже перевірено модульними тестами з питання 8.1.

У Spring Framework 6.2 з’явився також `MockMvcTester` з API у стилі AssertJ. Класичний `MockMvc` при цьому нікуди не зник, і на співбесідах найчастіше питають саме про нього.

Що варто перевіряти в тестах контролера? Не «усе підряд», а контракт API: правильні коди для успіху і помилок, обов’язкові поля, формат помилок і — важливо — відсутність полів, яких бути не повинно. Пам’ятаєте телефон заявника з питання 3.5? Тест, що перевіряє його відсутність, захистить від випадкового витоку краще за будь-яку інструкцію.

Приклад задачі

Умова. Написати тести `CallController` зі зрізом `@WebMvcTest`: отримання виклику у форматі JSON без телефону заявника, відповідь 404 у форматі ProblemDetail, відповідь 400 з переліком помилкових полів, відповідь 201 із заголовком `Location`.

Оскільки API тепер захищене, тест виконується від імені користувача з роллю чергового. Детальніше про тестування безпеки — у питанні 8.5.

Лістинг 8.3. Реалізація з покроковим розбором

Файл `CallControllerTest.java` зріз вебрівня

```
@WebMvcTest(CallController.class)
@Import({SecurityConfig.class, CallMapper.class})
@WithMockUser(roles = "DISPATCHER")
class CallControllerTest {
    @Autowired MockMvc mvc;
    @MockitoBean DispatchService service;
    @MockitoBean CallRepository calls;
```

У контекст потрапляють контролер, обробник помилок і конфігурація безпеки. Служба і репозиторій — моки. `@WithMockUser` «входить» від імені чергового для всіх тестів класу.

```
@Test
void returnsCallAsJson() throws Exception {
    when(service.find(1L)).thenReturn(call(1, "+380671234567"));

    mvc.perform(get("/api/calls/1"))
       .andExpect(status().isOk())
       .andExpect(jsonPath("$.address")
                .value("вул. Зелена, 101"))
       .andExpect(jsonPath("$.status").value("NEW"))
       .andExpect(jsonPath("$.callerPhone").doesNotExist());
}
```

Мок повертає виклик із телефоном заявника, а тест вимагає, щоб у JSON його не було.

```
@Test
void invalidRequestGives400WithFieldErrors() throws Exception {
    mvc.perform(post("/api/calls")
                .contentType(MediaType.APPLICATION_JSON)
                .content("""
                    {"address": "", "callerPhone": "067"}"""))
       .andExpect(status().isBadRequest())
       .andExpect(jsonPath("$.errors.address")
                .value("адресу не вказано"))
       .andExpect(jsonPath("$.errors.type").exists());
}
```

Перевіряємо разом валідацію з питання 3.6 і формат помилки з питання 3.7. Текстовий блок Java зручний для JSON у тестах.

```
private static FireCall call(long id, String phone) {
    FireCall c = new FireCall("вул. Зелена, 101", CallType.FIRE,
            LocalDateTime.of(2026, 9, 1, 9, 0), phone);
    ReflectionTestUtils.setField(c, "id", id);
    return c;
}
```

Допоміжний метод для тестових даних. Навіщо тут `ReflectionTestUtils` — див. примітку.

Вивід програми результат запуску

```
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0 -- in ...CallControllerTest
```

З досвіду викладачів. *Перша версія тесту падала з `NullPointerException` усередині маппера. Ми створили `FireCall` звичайним конструктором, а ідентифікатор сутності призначає база даних під час збереження — у моку бази немає, тож `getId()` повертав `null`. Додавати в сутність публічний сетер для ідентифікатора лише заради тестів не варто. `ReflectionTestUtils` з модуля `spring-test` записує значення в приватне поле, не змінюючи робочого коду.*

Варіанти для самостійної роботи

Завдання. Написати тести REST-контролера свого варіанта зі зрізом `@WebMvcTest` і `MockMvc`. Перевірити успішне отримання, створення з кодом 201, помилку валідації, відповідь 404 і відсутність у JSON указаного чутливого поля.

№	Контролер	Поле, якого не має бути у відповіді
1	`RescuerController`	`passportNumber`
2	`TruckController`	`vin`
3	`CadetController`	`parentPhone`
4	`HydrantController`	`ownerCode`
5	`EquipmentController`	`purchasePrice`

Питання 8.4. Testcontainers і PostgreSQL у тестах

Теоретична частина

Для тестів репозиторіїв довго й охоче використовували вбудовану базу H2 у пам'яті: вона стартує миттєво і не потребує встановлення. Але H2 — не PostgreSQL. Наш нативний запит з `::date`, типи `BIGSERIAL`, специфіку блокувань, функції дат і поведінку індексів H2 відтворює лише частково або взагалі ні. Тест, який проходить на H2, нічого не гарантує для робочої бази.

Testcontainers — бібліотека, яка перед тестами запускає справжню базу даних (чи брокер повідомлень, чи Keycloak) у контейнері Docker, а після тестів її зупиняє [68]. Тест працює з тією самою версією PostgreSQL, що й робочий сервер, а кожен запуск починається з чистої бази.

Spring Boot спрощує інтеграцію до мінімуму. Бін контейнера з анотацією `@ServiceConnection` автоматично стає джерелом налаштувань підключення: Boot сам підставить адресу, порт, логін і пароль запущеного контейнера замість `spring.datasource.*` [21]. Зазвичай такий бін описують в окремому класі `@TestConfiguration` і підключають до тестів через `@Import`.

Для `@DataJpaTest` потрібна ще одна деталь. За замовчуванням цей зріз намагається замінити базу вбудованою H2. Анотація `@AutoConfigureTestDatabase(replace = NONE)` вимикає заміну, і тест працює з контейнером.

У Testcontainers 2.0, з яким працює Spring Boot 4.1, змінилися назви модулів (`testcontainers-postgresql` замість `postgresql`) і пакет класу контейнера (`org.testcontainers.postgresql.PostgreSQLContainer`). Приклади з інтернету для версії 1.x доведеться трохи підправити.

Приклад задачі

Умова. Написати тести репозиторію викликів, які працюють зі справжнім PostgreSQL у контейнері: підрахунок незакритих викликів, нативний запит кількості викликів за сьогодні і статистика за типами. Передбачити можливість запуску тестів на локальній базі, якщо Docker недоступний.

Лістинг 8.4. Реалізація з покроковим розбором

Файл `PostgresContainerConfig.java` контейнер для тестів

```
@TestConfiguration(proxyBeanMethods = false)
@ConditionalOnProperty(name = "duty.test.containers",
                       havingValue = "true",
                       matchIfMissing = true)
public class PostgresContainerConfig {

    @Bean
    @ServiceConnection
    PostgreSQLContainer postgres() {
        return new PostgreSQLContainer("postgres:17-alpine");
    }
}
```

Та сама версія, що й у робочому середовищі. Умова на властивості дозволяє вимкнути контейнер (див. примітку). Типово він увімкнений.

Файл `CallRepositoryTest.java` тест репозиторію на PostgreSQL

```
@DataJpaTest
@AutoConfigureTestDatabase(replace = Replace.NONE)
@Import(PostgresContainerConfig.class)
class CallRepositoryTest {
    @Autowired CallRepository calls;
```

Зріз JPA з міграціями Flyway, справжньою базою і відкатом кожного тесту.

```
@Test
void countsOpenCallsAndTodayCalls() {
    calls.save(new FireCall("вул. Зелена, 101", CallType.FIRE,
            LocalDateTime.now(), null));
    FireCall closed = new FireCall("вул. Наукова, 7",
            CallType.HAZMAT, LocalDateTime.now(), null);
    closed.setStatus(CallStatus.CLOSED);
    calls.save(closed);

    assertThat(calls.countByStatusNot(CallStatus.CLOSED))
            .isEqualTo(1);
    assertThat(calls.countToday()).isEqualTo(2);
}
```

Нативний `countToday` з `::date` на H2 не спрацював би. Саме заради таких запитів потрібен справжній PostgreSQL.

Термінал з Docker і без нього

```
mvn verify
DUTY_TEST_CONTAINERS=false DB_URL=jdbc:postgresql://localhost:5432/duty \
    mvn verify
```

Вивід програми результат запуску

```
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0 -- in ...CallRepositoryTest
```

З досвіду викладачів. *На комп'ютері, де ми готували цей розділ, Docker не було, тож тести ми запускали другою командою — на локальному PostgreSQL з вимкненим контейнером. Так само буває і в навчальних аудиторіях. Але пам'ятайте: локальна база не очищається між запусками, тож тести мають або працювати в транзакції з відкатом (як `@DataJpaTest`), або не залежати від наявних даних. У CI-конвеєрі (розділ 10) Docker є завжди, і там тести працюють з контейнером.*

Варіанти для самостійної роботи

Завдання. Написати тести репозиторію свого варіанта з Testcontainers для запитів, специфічних для PostgreSQL. Показати, що хоча б один з них не працює на H2, і пояснити чому.

№	Запит для тесту	Особливість PostgreSQL
1	середній стаж за званнями	`round(avg(...), 1)`, сортування
2	автомобілі з ТО за останні 30 днів	`now() - interval '30 days'`
3	рейтинг курсантів у взводі	віконна функція `rank() over (...)`
4	гідранти в радіусі від точки	обчислення відстані за координатами
5	пошук спорядження за частиною назви без урахування регістру	`ILIKE`

Питання 8.5. Тестування захищених кінцевих точок

Теоретична частина

Правила безпеки — та частина коду, помилки в якій найдорожчі і найменш помітні. Переставили два рядки в `authorizeHttpRequests` — і рятувальник може видаляти виклики. Без тестів про це дізнаються лише після інциденту. Тому правила доступу перевіряють автоматично, як і будь-яку іншу логіку [59].

Модуль `spring-security-test` додає до тестів кілька інструментів:

`@WithMockUser(username = "...", roles = "...")` — виконати тест від імені вигаданого користувача без справжньої автентифікації;

`@WithUserDetails("dispatcher")` — завантажити користувача через наш `UserDetailsService`;

постпроцесори запитів `MockMvc`: `jwt()` — запит із токеном JWT із заданими повноваженнями, `httpBasic(...)`, `csrf()` — додати CSRF-токен до запиту форми.

Для API на JWT найзручніший `jwt()`: декодер токенів у тесті не викликається, а `MockMvc` одразу встановлює автентифікацію з потрібними повноваженнями. Підпис і термін дії токена при цьому не перевіряються — це вже відповідальність тестів конфігурації JWT.

Що тестувати? Таблицю «роль × дія» з очікуваним результатом. Корисно перевіряти не лише заборони, а й дозволи: якщо після рефакторингу адміністратор раптом отримує 403, це теж помилка. І обов’язково — запит без автентифікації: 401, а не 403 і не 302 з HTML сторінки входу (пригадайте примітку до питання 7.1).

Приклад задачі

Умова. Перевірити правила доступу до API викликів: анонімний запит отримує 401, рятувальник не може створювати виклики, черговий не може видаляти, адміністратор видаляє успішно.

Лістинг 8.5. Реалізація з покроковим розбором

Файл `CallSecurityTest.java` ролі і коди відповіді

```
@Test
void anonymousGets401() throws Exception {
    mvc.perform(delete("/api/calls/1"))
       .andExpect(status().isUnauthorized());
}
```

Без жодних облікових даних. Справжня конфігурація `SecurityConfig` імпортована в зріз, як і в питанні 8.3.

```
@Test
void rescuerCannotCreateCalls() throws Exception {
    mvc.perform(post("/api/calls").with(jwt().authorities(
                new SimpleGrantedAuthority("ROLE_RESCUER"))))
       .andExpect(status().isForbidden());
}
```

`jwt()` імітує запит із токеном рятувальника. Перевірка ролі спрацьовує раніше, ніж валідація тіла, тож тіло запиту можна не передавати.

```
@Test
void dispatcherCannotDelete() throws Exception {
    mvc.perform(delete("/api/calls/1").with(jwt().authorities(
                new SimpleGrantedAuthority("ROLE_DISPATCHER"))))
       .andExpect(status().isForbidden());
}

@Test
void adminDeletes() throws Exception {
    when(calls.existsById(1L)).thenReturn(true);

    mvc.perform(delete("/api/calls/1").with(jwt().authorities(
                new SimpleGrantedAuthority("ROLE_ADMIN"))))
       .andExpect(status().isNoContent());
}
```

Перший тест перевіряє `@PreAuthorize` на методі (правило адрес черговому DELETE дозволяє), другий — що дозвіл адміністратора не зламано.

Вивід програми результат запуску

```
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0 -- in ...CallSecurityTest
```

Порада. *Спробуйте поміняти місцями два правила `requestMatchers` для `/api/calls/**` у `SecurityConfig` і запустіть тести. Впаде саме той тест, який має впасти. Такий експеримент переконує краще за будь-які аргументи, що тести безпеки потрібні.*

Варіанти для самостійної роботи

Завдання. Написати тести правил доступу для API свого варіанта у вигляді таблиці «роль × дія». Для кожної ролі перевірити хоча б одну дозволену і одну заборонену дію, а також запит без автентифікації.

№	Ролі	Дії для перевірки
1	`HR`, `COMMANDER`, `RESCUER`	перегляд, переведення, звільнення рятувальника
2	`MECHANIC`, `COMMANDER`	перегляд, постановка на ТО, списання
3	`COURSE_OFFICER`, `CADET`	перегляд власного і чужого наряду, зміна наряду
4	`INSPECTOR`, `ADMIN`	перегляд, внесення результату, видалення гідранта
5	`STOREKEEPER`, `COMMANDER`, `RESCUER`	перегляд, видача до 10 і понад 10 одиниць

Питання 8.6. Покриття коду та піраміда тестування

Теоретична частина

Скільки тестів потрібно і яких? Класична відповідь — піраміда тестування [69; 64]. В основі — багато швидких модульних тестів. Над ними — менше інтеграційних (зрізи, база в контейнері). На вершині — небагато наскрізних (end-to-end) тестів, що перевіряють систему цілком через інтерфейс користувача чи API. Чим вище рівень, тим тест повільніший, крихкіший і дорожчий у підтримці, але тим ближче він до реального користувача.

У нашому проєкті піраміда виглядає так: 10 модульних тестів (статуси, служба), 10 тестів зрізів (вебрівень, безпека, репозиторій) і один тест з повним контекстом.

Покриття коду (code coverage) показує, яку частину коду виконали тести. Інструмент JaCoCo підключається до JVM агентом і після тестів будує звіт: покриття рядків, гілок (кожна сторона `if` чи `switch`), методів і класів. Звіт у HTML показує код із підсвіченими рядками — зеленим виконані, червоним пропущені, жовтим частково покриті гілки.

Як ставитися до цифри? Низьке покриття — точний сигнал: багато коду ніхто не перевіряє. Високе — не гарантія: тест може виконати рядок і нічого не перевірити (пригадайте пораду до питання 8.1). Тому погоня за 100 % — хибна мета. Розумно вимагати високого покриття для бізнес-логіки (служби, правила) і не витрачати сил на конфігураційні класи, демо-код і прості гетери. JaCoCo вміє і перевіряти мінімальний поріг (`check`), зупиняючи збирання, якщо покриття впало.

Ще одна корисна метрика — мутаційне тестування (PIT). Інструмент навмисно вносить у код дрібні зміни («мутанти») і перевіряє, чи тести їх «вбивають». Це відповідь на питання, чи тести справді щось перевіряють. На рівні Junior досить знати, що такий підхід існує.

Приклад задачі

Умова. Підключити JaCoCo до збирання, виключити з аналізу навчальний пакет `demo`, запустити всі тести і проаналізувати звіт: які пакети покрито добре, які погано, і які тести варто додати насамперед.

Лістинг 8.6. Реалізація з покроковим розбором

Файл `pom.xml` плагін покриття

```
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.8.15</version>
    <configuration>
        <excludes>
            <exclude>**/demo/**</exclude>
        </excludes>
    </configuration>
```

Демо-класи — навчальні точки запуску, а не код застосунку. Їх виключаємо, щоб не спотворювати картину.

```
    <executions>
        <execution>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
        <execution>
            <id>report</id>
            <phase>verify</phase>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

`prepare-agent` підключає агент до JVM тестів, `report` після фази `verify` будує звіт у `target/site/jacoco/index.html`.

Вивід програми покриття за пакетами (рядки / гілки)

```
пакет              рядки     гілки
call               69,7 %    50,0 %
call.web           45,0 %    14,3 %
crew               74,1 %     0,0 %
fleet              40,7 %     0,0 %
security           57,0 %     0,0 %
web                82,2 %   100,0 %
...
усього             63,3 %    41,5 %
```

З досвіду викладачів. *Готуючи звіт, ми спершу отримали порожній каталог `target/site`. Причина виявилася в тому, що ми передавали Maven параметр `-DargLine=...` для налаштування JVM тестів — і він перезаписав властивість `argLine`, через яку JaCoCo підключає свій агент. Тести пройшли, а покриття ніхто не вимірював. Якщо потрібні додаткові параметри JVM, пишіть у налаштуваннях Surefire `<argLine>@{argLine} ваші-параметри</argLine>` або передавайте їх змінними середовища.*

Варіанти для самостійної роботи

Завдання. Підключити JaCoCo до проєкту свого варіанта, проаналізувати звіт і додати тести, що підвищують покриття гілок указаного класу до заданого рівня. Налаштувати ціль `check` з мінімальним порогом для всього проєкту.

№	Клас для покращення	Цільове покриття гілок	Поріг `check` для проєкту
1	`ShiftService`	90 %	60 % рядків
2	`FleetService`	80 %	50 % гілок
3	`CadetDutyPlanner`	100 %	65 % рядків
4	`HydrantChecker`	90 %	55 % гілок
5	`EquipmentStock`	85 %	70 % рядків

ЧАСТИНА 5. ІНТЕГРАЦІЯ ТА РОЗГОРТАННЯ

Розділ 9. Взаємодія сервісів

Інформаційна система чергової частини не живе у вакуумі. Вона бере дані про погоду, повідомляє обласний центр, показує виклики на табло в залі. У цьому розділі застосунок навчиться звертатися до зовнішніх сервісів, обмінюватися повідомленнями і пережити ситуацію, коли інший сервіс не відповідає.

Питання 9.1. RestClient: виклик зовнішнього API

Теоретична частина

Для пожежної частини погода — не дрібниця: швидкість і напрямок вітру визначають, куди поширюватиметься вогонь і з якого боку підводити техніку. Тому картка виклику має показувати поточну погоду. Отримаємо її від відкритого сервісу Open-Meteo, який не вимагає ключа.

Для HTTP-запитів до інших сервісів у Spring є кілька клієнтів [54]. `RestTemplate` — найстаріший; він ще працює, але новий код на ньому вже не пишуть. `WebClient` — реактивний клієнт із модуля WebFlux, корисний переважно в реактивних застосунках. `RestClient` (з'явився у Spring Framework 6.1) — сучасний синхронний клієнт із «текучим» (fluent) API, який і варто використовувати в звичайних застосунках Spring MVC. Окрім того, є HTTP-інтерфейси: ви описуєте інтерфейс з анотаціями `@GetExchange`, а Spring генерує реалізацію, як для репозиторіїв.

Типовий запит із `RestClient` виглядає так: `get()`, `uri(...)` з параметрами, `retrieve()`, `body(Тип.class)`. Відповідь перетворюється на об'єкт тим самим Jackson, що й у контролерах. Для коду 4xx або 5xx `retrieve()` кидає виняток `HttpClientErrorException` або `HttpServerErrorException`; поведінку можна змінити методом `onStatus`.

Spring Boot створює бін `RestClient.Builder` з уже налаштованими конвертерами. Його впроваджують у свій клас і будують власний клієнт з базовою адресою. Зверніть увагу: у Boot 4 для цього потрібен окремий стартер `spring-boot-starter-restclient` [4].

Обов'язкова деталь, яку часто пропускають, — тайм-аути. Без них запит до сервісу, що «завис», чекатиме нескінченно довго, і разом з ним — потік, який обробляє запит чергового. Тайм-аут з'єднання обмежує час встановлення зв'язку, тайм-аут читання — очікування відповіді.

Базову адресу зовнішнього сервісу виносимо у властивості. Так у тестах її можна підмінити, а в робочому середовищі — направити запити через корпоративний проксі.

Приклад задачі

Умова. Створити клієнт погоди на основі `RestClient` з тайм-аутами з'єднання і читання. Отримати поточну температуру, швидкість і напрямок вітру для Львова і віддати їх через кінцеву точку `/api/weather`.

Лістинг 9.1. Реалізація з покроковим розбором

Файл `WeatherClient.java` клієнт з тайм-аутами

```
public WeatherClient(
        RestClient.Builder builder,
        @Value("${duty.weather.base-url}") String baseUrl) {
    HttpClient jdk = HttpClient.newBuilder()
            .connectTimeout(Duration.ofSeconds(2)).build();
    var factory = new JdkClientHttpRequestFactory(jdk);
    factory.setReadTimeout(Duration.ofSeconds(3));
    this.http = builder.baseUrl(baseUrl)
            .requestFactory(factory).build();
}
```

Використовуємо вбудований у JDK `HttpClient`. Дві секунди на з'єднання і три на відповідь — для чергової частини довше чекати погоду немає сенсу.

```
public WeatherReport current(double lat, double lon) {
    Forecast f = http.get()
            .uri(u -> u.path("/v1/forecast")
                    .queryParam("latitude", lat)
                    .queryParam("longitude", lon)
                    .queryParam("current", FIELDS)
                    .build())
            .retrieve()
            .body(Forecast.class);
```

`UriBuilder` сам закодує параметри. Відповідь сервісу Jackson перетворить на внутрішній запис `Forecast`.

```
record Current(
        @JsonProperty("temperature_2m") Double temperature,
        @JsonProperty("wind_speed_10m") Double windSpeed,
        @JsonProperty("wind_direction_10m") Integer windDir) {
}
```

`@JsonProperty` зв'язує назви полів зовнішнього API з нашими. Чужі назви не просочуються в решту коду — назовні йде лише власний `WeatherReport`.

Термінал запит погоди

```
curl -s -H "Authorization: Bearer $TOKEN" localhost:8080/api/weather
```

Вивід програми результат запуску

```
{"temperature":9.8,"windSpeed":11.5,"windDirection":264,
 "source":"open-meteo.com"}
```

З досвіду викладачів. *Застосунок спершу не стартував з повідомленням, що бракує біна `RestClient.Builder`. У Spring Boot 3 його створювала автоконфігурація вебстартера, а в Boot 4 вона переїхала в окремий модуль. Якщо після оновлення версії Boot «зникають» знайомі біни — перегляньте розділ про модулі в примітках до випуску [4], там є таблиця нових стартерів.*

Варіанти для самостійної роботи

Завдання. Створити клієнт до відкритого API для свого варіанта на основі `RestClient` з тайм-аутами і базовою адресою у властивостях. Перетворити відповідь на власний запис і віддати через REST-кінцеву точку. Обробити відповідь 404 зовнішнього сервісу методом `onStatus`.

№	Дані	Можливе джерело
1	прогноз погоди на 3 дні	Open-Meteo, параметр `daily`
2	якість повітря (PM2.5)	Open-Meteo Air Quality API
3	державні свята для графіка нарядів	Nager.Date API
4	координати за адресою (геокодування)	Nominatim (OpenStreetMap)
5	курс валют для закупівель	API Національного банку України

Питання 9.2. Обмін повідомленнями: RabbitMQ на базовому рівні

Теоретична частина

HTTP-запит — синхронна взаємодія: клієнт чекає, доки сервер відповість. Якщо обласний центр зараз недоступний, повідомлення про закритий виклик загубиться або чергову частину змусять чекати. Для подій, які треба доставити напевно, але не обов'язково миттєво, використовують брокер повідомлень [56].

Брокер — проміжний сервер, який приймає повідомлення від відправника (producer) і зберігає їх, доки отримувач (consumer) їх не забере. Відправник і отримувач не знають один про одного і можуть працювати в різний час. Два найвідоміші брокери — RabbitMQ і Apache Kafka.

У RabbitMQ повідомлення надходить не одразу в чергу, а в обмінник (exchange), який за правилами маршрутизації (bindings) розкладає його по чергах [70]. Типи обмінників: `direct` (за точним ключем), `topic` (за шаблоном ключа, як-от `call.*`), `fanout` (у всі прив'язані черги). Отримувач читає повідомлення з черги і підтверджує обробку; якщо підтвердження немає, повідомлення повернеться в чергу.

Kafka влаштована інакше: це розподілений журнал подій, де повідомлення зберігаються певний час і можуть бути прочитані повторно різними групами отримувачів. Для потокової обробки великих обсягів подій це перевага. Для простого «доставити завдання» RabbitMQ зазвичай простіший. На співбесіді Junior досить вміти пояснити цю різницю.

У Spring з RabbitMQ працюють через Spring AMQP: `RabbitTemplate` для надсилання і `@RabbitListener` для отримання. Біни `TopicExchange`, `Queue`, `Binding` Spring сам створить у брокері під час старту. Повідомлення зручно передавати в JSON — для цього підключають `JacksonJsonMessageConverter`.

І ще одна важлива думка. Надсилати повідомлення варто лише після фіксації транзакції — інакше обласний центр може дізнатися про закриття виклику, який через секунду буде відкочено. Тому ми знову використовуємо `@TransactionalEventListener` з питання 6.4.

Приклад задачі

Умова. Після закриття виклику надсилати повідомлення в обмінник `duty.events` з ключем `call.closed`. Обласний центр отримує всі події з ключами `call.*` зі своєї черги. Увімкнути обмін повідомленнями властивістю, щоб застосунок працював і без брокера.

Лістинг 9.2. Реалізація з покроковим розбором

Файл `MessagingConfig.java` обмінник, черга, прив'язка

```
@Configuration
@ConditionalOnBooleanProperty("duty.messaging.enabled")
public class MessagingConfig {
    public static final String EXCHANGE = "duty.events";
    public static final String REGION_QUEUE = "region.calls";
```

Уся конфігурація обміну повідомленнями вмикається однією властивістю. Типово вона `false`: брокер з'явиться в розділі 10 разом із Docker Compose.

```
@Bean
Binding regionBinding(Queue regionQueue,
                      TopicExchange dutyEvents) {
    return BindingBuilder.bind(regionQueue).to(dutyEvents)
            .with("call.*");
}
```

Черга обласного центру отримуватиме і `call.closed`, і майбутні `call.created`, `call.dispatched`.

Файл `CallEventPublisher.java` надсилання після фіксації

```
@TransactionalEventListener
public void onClosed(CallClosedEvent event) {
    var message = new CallEventMessage(event.callId(), "closed",
            Instant.now(clock));
    rabbit.convertAndSend(MessagingConfig.EXCHANGE,
            "call.closed", message);
}
```

Повідомлення — окремий запис, а не внутрішня подія застосунку: формат повідомлення — це контракт з іншою системою, і змінювати його треба обережно.

Файл `RegionCenterListener.java` отримувач

```
@RabbitListener(queues = MessagingConfig.REGION_QUEUE)
public void receive(CallEventMessage message) {
    log.info("Обласний центр отримав: виклик №{} {}",
            message.callId(), message.event());
}
```

У реальній системі отримувач жив би в окремому застосунку обласного центру. Тут він у тому самому проєкті, щоб приклад був самодостатнім.

Файл `CallEventPublisherTest.java` перевірка без брокера

```
publisher.onClosed(new CallClosedEvent(7, 1L));

var message = ArgumentCaptor.forClass(Object.class);
verify(rabbit).convertAndSend(eq("duty.events"),
        eq("call.closed"), message.capture());
assertThat(message.getValue()).isEqualTo(new CallEventMessage(
        7, "closed", Instant.parse("2026-09-01T06:00:00Z")));
```

Модульний тест з моком `RabbitTemplate` перевіряє обмінник, ключ і вміст повідомлення.

Зверніть увагу. *На комп'ютері, де ми готували розділ, не було ні Docker, ні RabbitMQ, тому код надсилання перевірено лише модульним тестом. Для повної перевірки запустіть брокер (наприклад, командою з розділу 10), увімкніть* `duty.messaging.enabled=true` *і закрийте виклик — у лозі з'явиться повідомлення від* `RegionCenterListener`*, а на вебсторінці керування RabbitMQ (порт 15672) — черга* `region.calls`*. Індикатор стану* `rabbit` *в Actuator ми прив'язали до тієї самої властивості, щоб без брокера застосунок не вважався «хворим».*

Варіанти для самостійної роботи

Завдання. Налаштувати обмін повідомленнями RabbitMQ для свого варіанта: обмінник, черги, прив'язки і відправку після фіксації транзакції. Написати модульний тест відправника і, за наявності брокера, перевірити доставку.

№	Подія	Тип обмінника	Отримувачі
1	рятувальника переведено	`topic` (`staff.*`)	кадрова служба
2	автомобіль на ТО	`direct` (`maintenance`)	механічна служба
3	наряд змінено	`fanout`	курсант, офіцер курсу
4	гідрант несправний	`topic` (`hydrant.broken`)	водоканал, район
5	видано спорядження	`direct` (`issue`)	бухгалтерія

Питання 9.3. WebSocket і STOMP: живе табло викликів

Теоретична частина

У залі чергової частини висить табло з поточними викликами. Оновлювати його запитом до API щосекунди — марна витрата ресурсів, а оновлення раз на хвилину — запізно. Потрібно, щоб сервер сам повідомляв табло про зміни.

HTTP не вміє «штовхати» дані від сервера: відповідь завжди йде на запит. WebSocket — окремий протокол поверх TCP, що починається з HTTP-запиту зі спеціальними заголовками (`Upgrade: websocket`), а далі тримає відкрите двостороннє з'єднання [71]. Сервер і клієнт можуть надсилати повідомлення будь-коли.

Сам WebSocket — лише «труба» для байтів. Щоб не вигадувати власний формат повідомлень, використовують STOMP — простий текстовий протокол обміну повідомленнями з командами `CONNECT`, `SUBSCRIBE`, `SEND`. Клієнт підписується на адресу (destination), наприклад `/topic/calls`, і отримує все, що туди надсилає сервер [72].

Spring вмикає цей механізм анотацією `@EnableWebSocketMessageBroker`. Ми реєструємо кінцеву точку для з'єднання (`/ws`) і простий вбудований брокер для адрес `/topic/...`. Надсилає повідомлення бін `SimpMessagingTemplate`. Для кластера з кількох екземплярів вбудований брокер замінюють зовнішнім (RabbitMQ з плагіном STOMP).

Альтернатива WebSocket для однобічних оновлень — Server-Sent Events (SSE): звичайний HTTP-запит, у відповідь на який сервер довго надсилає події. SSE простіший, але дані йдуть лише від сервера до клієнта. Для табло цього досить, і на співбесіді доречно згадати обидва варіанти.

Зверніть увагу на безпеку. З'єднання WebSocket після встановлення не проходить ланцюжок фільтрів для кожного повідомлення. Для навчального табло ми відкрили `/ws/**` без автентифікації, і на табло йдуть лише адреса, тип і статус виклику. У робочій системі доступ до кінцевої точки обмежують так само, як і до API, а для повідомлень використовують Spring Security Messaging.

Приклад задачі

Умова. Додати живе табло викликів: під час реєстрації і зміни статусу виклику застосунок після фіксації транзакції надсилає повідомлення на адресу `/topic/calls`. Підготувати сторінку `board.html` з клієнтом STOMP і перевірити роботу клієнтом на Java.

Лістинг 9.3. Реалізація з покроковим розбором

Файл `WebSocketConfig.java` кінцева точка і брокер

```
@Override
public void registerStompEndpoints(
        StompEndpointRegistry registry) {
    registry.addEndpoint("/ws");
}

@Override
public void configureMessageBroker(
        MessageBrokerRegistry registry) {
    registry.enableSimpleBroker("/topic");
    registry.setApplicationDestinationPrefixes("/app");
}
```

Клієнти з'єднуються з `ws://сервер/ws`. Вбудований брокер розсилає повідомлення підписникам адрес `/topic/...`.

Файл `BoardPublisher.java` надсилання на табло

```
@TransactionalEventListener
public void onChange(CallChangedEvent e) {
    broker.convertAndSend("/topic/calls", new BoardMessage(
            e.callId(), e.address(), e.type().name(),
            e.status().name()));
}
```

Подію `CallChangedEvent` служба публікує під час реєстрації і кожної зміни статусу. Табло дізнається про зміну лише після фіксації транзакції.

Файл `board.html` сторінка табло (фрагмент)

```
const client = new StompJs.Client({
    brokerURL: `ws://${location.host}/ws`
});
const list = document.getElementById('calls');
client.onConnect = () =>
    client.subscribe('/topic/calls', frame => {
        const c = JSON.parse(frame.body);
        const li = document.createElement('li');
        li.textContent =
            `№${c.id} ${c.type} ${c.address} — ${c.status}`;
        list.prepend(li);
    });
client.activate();
```

Бібліотека `@stomp/stompjs` підключається з CDN. Нове повідомлення з'являється вгорі списку.

Файл `Demo93Board.java` клієнт STOMP на Java

```
session.subscribe("/topic/calls", new StompFrameHandler() {
    public Type getPayloadType(StompHeaders h) {
        return BoardMessage.class;
    }
    public void handleFrame(StompHeaders h, Object payload) {
        System.out.println("Табло: " + payload);
        received.countDown();
    }
});
```

Демо підписується на табло, реєструє виклик і змінює його статус. `CountDownLatch` чекає двох повідомлень.

Вивід програми результат запуску

```
Табло: BoardMessage[id=1, address=вул. Зелена, 101, type=FIRE, status=NEW]
Табло: BoardMessage[id=1, address=вул. Зелена, 101, type=FIRE, status=DISPATCHED]
Отримано всі: true
```

Порада. *Відкрийте `http://localhost:8080/board.html` у двох вікнах браузера і зареєструйте виклик через форму журналу. Обидва табло оновляться одночасно, без перезавантаження. У вкладці «Мережа» інструментів розробника можна побачити саме з'єднання WebSocket і кожен кадр STOMP — це корисно для налагодження.*

Варіанти для самостійної роботи

Завдання. Реалізувати для свого варіанта оновлення в реальному часі через WebSocket і STOMP: кінцеву точку, адресу підписки, надсилання після фіксації транзакції і просту HTML-сторінку. Порівняти з реалізацією тієї самої функції через Server-Sent Events.

№	Що показувати в реальному часі	Адреса підписки
1	склад караулів, що змінюється	`/topic/crews`
2	статус автомобілів (у резерві, на виїзді, на ТО)	`/topic/trucks`
3	хто з курсантів заступив у наряд	`/topic/duty`
4	результати перевірки гідрантів на карті	`/topic/hydrants`
5	залишки на складі	`/topic/stock`

Питання 9.4. Моноліт чи мікросервіси: що Junior має знати

Теоретична частина

Наш застосунок — моноліт: усе в одному процесі, одна база, одне розгортання. Мікросервісна архітектура розбиває систему на невеликі незалежні сервіси, кожен з власною базою даних і власним циклом розгортання, які взаємодіють через мережу — HTTP або брокер повідомлень [73; 56].

Переваги мікросервісів реальні: незалежне розгортання (оновили сервіс погоди, не чіпаючи журнал), масштабування окремих частин, різні технології для різних задач, менші команди з чіткими зонами відповідальності. Але й ціна висока: мережеві виклики замість викликів методів (а мережа ненадійна, див. питання 9.5), розподілені транзакції, які доводиться замінювати сагами і подіями, складніші розгортання, моніторинг і налагодження, дублювання даних між сервісами.

Відома порада Мартіна Фаулера — «спершу моноліт» (monolith first): більшість успішних мікросервісних систем починалися як моноліт, який розділили, коли межі між частинами стали зрозумілими. Добре структурований модульний моноліт — пакети за функціональними областями, чіткі залежності між ними, взаємодія через події — розділити значно легше, ніж «великий клубок».

Подивіться на наш проєкт з цього погляду. Пакети `call`, `crew`, `fleet`, `weather`, `security` — кандидати на окремі модулі. Зв’язок між викликами й автопарком ми вже зробили через подію (питання 6.4), а з обласним центром — через брокер (9.2). Клієнт погоди спілкується із зовнішнім сервісом через власний DTO і не пускає чужу модель далі. Саме ці прийоми знадобляться при розділенні.

Що з цього має знати Junior? Уміти назвати переваги і недоліки обох підходів без фанатизму; знати базові патерни мікросервісів (API Gateway, Service Discovery, Circuit Breaker, база даних на сервіс, сага); розуміти, що мікросервіси — це передусім організаційне рішення, а не технічна мода. Відповідь «у нас моноліт, і це нормально для нашого розміру» на співбесіді звучить зріліше, ніж «ми все переписуємо на мікросервіси».

Приклад задачі

Умова. Проаналізувати структуру наскрізного проєкту і запропонувати, як розділити його на сервіси, якщо навантаження зросте: які межі, які дані належать

кожному сервісу, як вони взаємодіятимуть. Порівняти з поточною структурою модульного моноліту.

Задача аналітична, тож замість коду — схеми. Саме такі схеми найчастіше просять намалювати на співбесіді.

Лістинг 9.4. Реалізація з покроковим розбором

Схема поточний модульний моноліт

```
duty-app (один процес, одна база PostgreSQL)
├── call      виклики, журнал ──подія CallClosedEvent──┐
├── crew      караули, рятувальники  <──────────────────┘
├── fleet     автопарк, кеш довідника
├── weather   клієнт зовнішнього API ──HTTP──> open-meteo.com
├── board     табло (WebSocket)
├── messaging ──RabbitMQ──> обласний центр
└── security  користувачі, JWT
```

Схема можливе розділення на сервіси

```
                ┌──────────────── API Gateway (JWT) ────────────────┐
                │                       │                           │
     call-service              fleet-service             weather-service
     (виклики, журнал)         (караули, техніка)        (кеш погоди)
       БД: calls                 БД: fleet                 без БД
          │                          ^
          └── call.closed ──> RabbitMQ ──> fleet-service, region-center
Користувачі та токени — окремий сервер авторизації (Keycloak, питання 7.5)
```

З досвіду викладачів. *Типова помилка в навчальних проєктах — «мікросервіси», де три сервіси звертаються до однієї спільної бази даних. Це не мікросервіси, а розподілений моноліт — поєднання недоліків обох підходів: мережеві виклики є, а незалежності немає. Окрема база даних на сервіс — не формальність, а суть підходу.*

Варіанти для самостійної роботи

Завдання. Для системи свого варіанта намалювати схему модульного моноліту і схему розділення на 3–4 сервіси. Для кожного сервісу вказати дані, якими він володіє, і спосіб взаємодії (синхронний HTTP чи повідомлення). Обґрунтувати, яку взаємодію зробити асинхронною.

№	Система	Обов’язково врахувати
1	облік особового складу частини	інтеграція з кадровою системою ДСНС
2	автопарк і технічне обслуговування	GPS-трекери автомобілів
3	навчальний процес курсантів	розклад, наряди, оцінки
4	протипожежне водопостачання міста	карта, інспекції, водоканал
5	матеріально-технічне забезпечення	склади, закупівлі, бухгалтерія

Питання 9.5. Стійкість до збоїв: тайм-аути, повтори, circuit breaker

Теоретична частина

Перша помилка розподілених обчислень — вважати, що мережа надійна. Зовнішній сервіс може не відповідати, відповідати повільно, повертати помилки через раз. Якщо наш застосунок цього не враховує, збій чужого сервісу перетворюється на збій нашого: потоки чекають, пул з’єднань вичерпується, і журнал викликів перестає працювати через

те, що «лежить» сервіс погоди. Майкл Найгард детально описав такі каскадні збої і патерни захисту від них [74].

Основні патерни стійкості (resilience):

тайм-аут — не чекати довше, ніж має сенс (питання 9.1);

повтор (retry) — повторити запит при тимчасовій помилці, з паузою між спробами; повторювати можна лише ідемпотентні операції (пригадайте питання 3.9);

запобіжник (circuit breaker) — якщо частка невдалих викликів перевищила поріг, «розімкнути коло» і деякий час взагалі не звертатися до сервісу, одразу повертаючи запасний результат; потім обережно перевірити, чи сервіс ожив (стан half-open);

запасний результат (fallback) — що повернути, коли сервіс недоступний: кешоване значення, значення за замовчуванням, чесне «недоступно»;

обмеження одночасних викликів (bulkhead) і частоти (rate limiter).

У Java ці патерни реалізує бібліотека Resilience4j [75]. З модулем для Spring Boot їх застосовують анотаціями `@Retry`, `@CircuitBreaker`, `@Bulkhead` з параметрами у властивостях. Spring Framework 7 також додав власні базові анотації `@Retryable` і `@ConcurrencyLimit` [5] — для простих випадків без запобіжника їх досить.

Важливий нюанс — порядок. Resilience4j застосовує аспекти в певному порядку: повтор зовні, запобіжник усередині. Тому запасний результат має бути на зовнішньому аспекті (повторі), інакше повтори не відбуватимуться (див. примітку). А повторювати виняток розімкненого запобіжника (`CallNotPermittedException`) немає сенсу — його виключають із повторів.

Для системи цивільного захисту правило просте: збій допоміжного сервісу не повинен заважати основній роботі. Черговий має зареєструвати виклик навіть тоді, коли погода «недоступна».

Приклад задачі

Умова. Захистити отримання погоди: три спроби з паузою 500 мс, запобіжник, що розмикається, якщо з двох останніх викликів хоча б половина невдалі, і запасний результат «недоступно». Перевірити поведінку, спрямувавши клієнт на адресу, що не відповідає.

Лістинг 9.5. Реалізація з покроковим розбором

Файл `WeatherService.java` анотації стійкості

```
@Retry(name = "weather", fallbackMethod = "unavailable")
@CircuitBreaker(name = "weather")
public WeatherReport lviv() {
    log.info("Запит погоди");
    return client.current(LVIV_LAT, LVIV_LON);
}
```

Ім’я `weather` пов’язує анотації з налаштуваннями у `application.yml`. Запасний метод — на повторі, тобто на зовнішньому аспекті.

```
private WeatherReport unavailable(Throwable e) {
    log.warn("Погода недоступна: {}",
            e.getClass().getSimpleName());
    return WeatherReport.unknown();
}
```

Сигнатура запасного методу — та сама, плюс параметр із винятком. Черговий побачить «недоступно», а не помилку 500.

Файл `application.yml` параметри повтору і запобіжника

```
resilience4j:
  retry:
    instances:
      weather:
        max-attempts: 3
        wait-duration: 500ms
        ignore-exceptions:
          - io.github.resilience4j.circuitbreaker.CallNotPermittedException
  circuitbreaker:
    instances:
      weather:
        sliding-window-size: 4
        minimum-number-of-calls: 2
        failure-rate-threshold: 50
        wait-duration-in-open-state: 30s
```

Файл `Demo95Resilience.java` три запити до недоступного сервісу

```
try (var ctx = DemoApp.start(
        "--duty.weather.base-url=http://10.255.255.1")) {
    var weather = ctx.getBean(WeatherService.class);
    for (int i = 1; i <= 3; i++) {
        long start = System.currentTimeMillis();
        var report = weather.lviv();
        System.out.println(i + ": " + report.source() + " за "
                + (System.currentTimeMillis() - start) + " мс");
    }
}
```

Адреса з приватного діапазону, на яку ніхто не відповідає, — зручний спосіб відтворити тайм-аут з'єднання.

Вивід програми результат запуску

```
INFO  WeatherService - Запит погоди
INFO  WeatherService - Запит погоди
WARN  WeatherService - Погода недоступна: CallNotPermittedException
1: недоступно за 5071 мс
WARN  WeatherService - Погода недоступна: CallNotPermittedException
2: недоступно за 1 мс
WARN  WeatherService - Погода недоступна: CallNotPermittedException
3: недоступно за 0 мс
```

З досвіду викладачів. *Перша версія мала `fallbackMethod` на `@CircuitBreaker` — так написано в багатьох прикладах з інтернету. Повторів не було жодного: запобіжник першим ловив виняток і повертав «недоступно», а для повтору це був успіх. Коли запасний метод переїхав на `@Retry`, повтори з'явилися, але після розмикання запобіжника кожен виклик ще секунду «повторював» відмову запобіжника. Звідси `ignore-exceptions`. Подивіться на вивід: перший запит чесно чекав дві спроби з тайм-аутами, а наступні отримують відповідь миттєво — саме для цього і потрібен запобіжник.*

Варіанти для самостійної роботи

Завдання. Захистити виклик зовнішнього сервісу з питання 9.1 свого варіанта патернами стійкості з указаними параметрами. Відтворити збій сервісу і показати в лозі роботу кожного патерну. Пояснити, що буде без кожного з них.

№	Повтори	Запобіжник	Запасний результат
1	2 спроби, 1 с	50 % з 6 викликів, 60 с	останній успішний прогноз з кешу
2	3 спроби, експоненційна пауза	30 % з 10 викликів	«дані недоступні»
3	без повторів	50 % з 4 викликів	графік без урахування свят
4	3 спроби, 200 мс	70 % з 5 викликів, 10 с	координати центру району
5	2 спроби, 500 мс	`@Bulkhead` на 5 одночасних	курс за попередній день

Розділ 10. Розгортання

Застосунок, який працює лише на ноутбуці розробника, ще не застосунок. У цьому розділі ми запакуємо систему чергової частини в образ Docker, запустимо її разом з базою даних і брокером однією командою, налаштуємо автоматичне збирання й тестування після кожного коміту і підготуємо конфігурацію для робочого середовища. Усі файли розділу лежать у корені проєкту `duty-app`.

Питання 10.1. Збирання jar і Docker-образ

Теоретична частина

Команда `mvn package` збирає виконуваний jar, у якому є все потрібне: наш код, бібліотеки, вбудований Tomcat (питання 2.4). Запустити його можна на будь-якій машині з потрібною версією Java. Саме тут і проблема: «потрібна версія Java», «потрібні змінні середовища», «потрібна база даних» — на сервері все це має бути налаштовано так само, як у розробника. Фраза «а в мене працює» з'явилася не на порожньому місці.

Контейнер розв'язує цю проблему: застосунок постачається разом із середовищем виконання — операційною системою, JRE, налаштуваннями [76]. Образ (image) — незмінний шаблон, контейнер — запущений екземпляр образу. Образ описують файлом `Dockerfile` з інструкціями: `FROM` (базовий образ), `COPY` (скопіювати файли), `RUN` (виконати команду під час збирання), `ENTRYPOINT` (команда запуску).

Кілька практик, які відрізняють хороший `Dockerfile` від першого-ліпшого:

багатоетапне збирання (multi-stage build) — на першому етапі образ з JDK і Maven збирає jar, на другому в маленький образ з JRE копіюється лише результат; компілятор і вихідний код у підсумковий образ не потрапляють;

шари — Docker кешує кожну інструкцію, тому спершу копіюють те, що змінюється рідко (залежності), а потім — свій код; Spring Boot уміє розкласти jar на шари (`-Djarmode=tools extract --layers`), і після зміни одного класу перезбирається лише останній шар у кілька кілобайтів;

запуск від звичайного користувача, а не від root;

параметри пам'яті у відсотках (`-XX:MaxRAMPercentage`), щоб JVM враховувала обмеження контейнера.

Альтернатива власному `Dockerfile` — Cloud Native Buildpacks: команда `mvn spring-boot:build-image` збере оптимізований образ без жодного файлу конфігурації [21]. Для навчання ми все ж пишемо `Dockerfile` вручну — так видно, що відбувається.

Ще одна відмінність, яку питають на співбесідах: контейнер — не віртуальна машина. Він використовує ядро операційної системи хоста і ізолює лише процес, файлову систему і мережу, тому стартує за секунди і займає мегабайти, а не гігабайти.

Приклад задачі

Умова. Написати багатоетапний `Dockerfile` для застосунку чергової частини: збирання jar на образі з JDK 25, розкладання на шари і запуск на образі з JRE 25 від окремого користувача. Додати `.dockerignore`.

Лістинг 10.1. Реалізація з покроковим розбором

Dockerfile етап збирання

```
FROM eclipse-temurin:25-jdk AS build
WORKDIR /src
COPY mvnw pom.xml ./
COPY .mvn .mvn
RUN ./mvnw -q dependency:go-offline
COPY src src
```

Спершу копіюємо лише `pom.xml` і обгортку Maven та завантажуємо залежності. Поки `pom.xml` не змінювався, Docker бере цей шар з кешу, і перезбирання займає секунди.

```
RUN ./mvnw -q package -DskipTests \
 && java -Djarmode=tools -jar target/duty-app-*.jar \
         extract --layers --launcher --destination target/app
```

Тести тут пропускаємо свідомо: їх уже виконав CI (питання 10.3). jar розкладається на каталоги-шари.

Dockerfile етап запуску

```
FROM eclipse-temurin:25-jre
RUN useradd --system --uid 1001 duty
WORKDIR /app
COPY --from=build /src/target/app/dependencies/ ./
COPY --from=build /src/target/app/spring-boot-loader/ ./
COPY --from=build /src/target/app/snapshot-dependencies/ ./
COPY --from=build /src/target/app/application/ ./
```

Шари копіюються від найстабільнішого (бібліотеки) до найчастіше змінюваного (наш код). У підсумковому образі немає ні JDK, ні Maven, ні вихідного коду. Далі процес запускається від системного користувача `duty`: якщо зловмисник і знайде вразливість, прав root у контейнері він не отримає.

```
USER duty
EXPOSE 8080
ENTRYPOINT ["java", "-XX:MaxRAMPercentage=75", \
            "org.springframework.boot.loader.launch.JarLauncher"]
```

Файл `.dockerignore` що не потрапляє в контекст збирання

```
target/
.idea/
*.iml
.git/
```

Термінал збирання і запуск образу

```
docker build -t duty-app:0.1.0 .
docker images duty-app
docker run --rm -p 8080:8080 -e DB_URL=... duty-app:0.1.0
```

Порада. *Порівняйте розміри: образ з JDK і Maven важить кілька сотень мегабайтів більше, ніж підсумковий образ з JRE. Тепер змініть один рядок у контролері й перезберіть образ — у виводі* `docker build` *ви побачите, що всі шари, крім останнього, взято з кешу (*`CACHED`*). Саме заради цього шари і впорядковують.*

Варіанти для самостійної роботи

Завдання. Написати багатоетапний `Dockerfile` для застосунку свого варіанта з урахуванням указаної вимоги. Порівняти розмір і час перезбирання образу з одноетапним варіантом.

№	Додаткова вимога до образу
1	мітки (`LABEL`) з версією і автором образу
2	збирання образу через `mvn spring-boot:build-image` і порівняння з власним `Dockerfile`
3	базовий образ на Alpine; пояснити ризики musl-бібліотеки
4	аргумент збирання (`ARG`) з версією застосунку
5	образ без оболонки (distroless); як у ньому налагоджувати проблеми

Питання 10.2. Docker Compose: застосунок + PostgreSQL

Теоретична частина

Наш застосунок потребує PostgreSQL і RabbitMQ. Запускати кожен контейнер окремою довгою командою `docker run` з десятком параметрів незручно і легко помилитися. Docker Compose описує всю систему одним файлом `compose.yaml`: які сервіси (контейнери) потрібні, з яких образів, з якими змінними середовища, томами і портами [76]. Команда `docker compose up` піднімає все разом, `docker compose down` — зупиняє.

Сервіси в Compose потрапляють в одну віртуальну мережу і бачать один одного за іменами. Тому застосунок звертається до бази за адресою `db:5432`, а не `localhost` — усередині контейнера `localhost` означає сам контейнер.

Дані бази зберігаються в іменованому томі (volume). Контейнер можна видалити й створити заново, а дані залишаться. Без тому кожен перезапуск починався б з порожньої бази — пам'ятаєте міграції Flyway (питання 5.7)? Саме вони заповнять нову базу правильною схемою.

Порядок запуску задає `depends_on`. Але «контейнер запущено» не означає «база готова приймати з'єднання»: PostgreSQL потребує кількох секунд на старт. Тому для бази описують перевірку готовності (`healthcheck`), а застосунок чекає умови `service_healthy`.

Секрети в `compose.yaml` не пишуть. Compose підставляє значення змінних `${IM'Я}` з оточення або з файлу `.env` у тому самому каталозі. Файл `.env` додають у `.gitignore`, а в репозиторій кладуть приклад `.env.example`. Синтаксис `${JWT_SECRET:?повідомлення}` зупинить запуск з поясненням, якщо змінну не задано.

Spring Boot має ще й власну підтримку Compose: залежність `spring-boot-docker-compose` під час запуску в IDE сама виконає `docker compose up` і налаштує підключення до сервісів. Для розробки це дуже зручно.

Приклад задачі

Умова. Описати в `compose.yaml` систему з трьох сервісів: PostgreSQL 17 з томом для даних і перевіркою готовності, RabbitMQ з вебконсоллю, застосунок чергової частини з профілем `prod`. Секрети передати через файл `.env`.

Лістинг 10.2. Реалізація з покроковим розбором

Файл `compose.yaml` база даних

```yaml
services:
  db:
    image: postgres:17-alpine
    environment:
      POSTGRES_DB: duty
      POSTGRES_USER: duty
      POSTGRES_PASSWORD: ${DB_PASSWORD:-duty}
    volumes:
      - db-data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD", "pg_isready", "-U", "duty"]
      interval: 5s
      retries: 10
```

Образ PostgreSQL сам створює базу і користувача зі змінних середовища. `pg_isready` перевіряє, чи база вже приймає з'єднання. Порт назовні не відкриваємо: базу бачить лише застосунок.

```yaml
rabbitmq:
  image: rabbitmq:4-management
  ports:
    - "15672:15672"
```

Брокер для питання 9.2. Образ з суфіксом `management` містить вебконсоль на порту 15672.

Файл `compose.yaml` застосунок

```yaml
app:
  build: .
  depends_on:
    db:
      condition: service_healthy
    rabbitmq:
      condition: service_started
```

`build: .` — образ збирається з нашого `Dockerfile`. Застосунок стартує лише тоді, коли база пройшла перевірку готовності.

```yaml
environment:
  SPRING_PROFILES_ACTIVE: prod
  DB_URL: jdbc:postgresql://db:5432/duty
  DB_PASSWORD: ${DB_PASSWORD:-duty}
  JWT_SECRET: ${JWT_SECRET:?задайте JWT_SECRET у файлі .env}
  DUTY_MESSAGING_ENABLED: "true"
  SPRING_RABBITMQ_HOST: rabbitmq
ports:
  - "8080:8080"
```

Усі налаштування — змінними середовища. Spring Boot перетворює `DUTY_MESSAGING_ENABLED` на властивість `duty.messaging.enabled` (relaxed binding, питання 2.3). Без `JWT_SECRET` Compose не запуститься.

Файл `.env.example` приклад файлу секретів

```
# Скопіюйте у .env і задайте власні значення. Файл .env не комітити!
DB_PASSWORD=change-me
JWT_SECRET=згенеруйте-openssl-rand-base64-48
```

Термінал запуск усієї системи

```
cp .env.example .env        # і відредагувати значення
docker compose up -d --build
docker compose ps
docker compose logs -f app
docker compose down         # дані в томі db-data зберігаються
```

Зверніть увагу. *У цьому розділі ми вперше передаємо `DUTY_MESSAGING_ENABLED: "true"` — обмін повідомленнями з розділу 9 запрацює повністю. Після `docker compose up` відкрийте вебконсоль RabbitMQ на `http://localhost:15672` (логін і пароль за замовчуванням `guest`), закрийте виклик через API — і побачите повідомлення в черзі `region.calls`. Лапки навколо `"true"`*

обов'язкові: YAML інакше перетворить значення на логічний тип, а змінні середовища — завжди рядки.

Варіанти для самостійної роботи

Завдання. Описати в `compose.yaml` систему для застосунку свого варіанта з указаними сервісами. Налаштувати перевірки готовності, томи для даних і секрети через `.env`. Продемонструвати, що дані зберігаються після `docker compose down` і `up`.

№	Сервіси, крім застосунку і PostgreSQL
1	Keycloak (питання 7.5)
2	Redis для кешу автопарку (питання 6.1)
3	pgAdmin для адміністрування бази
4	RabbitMQ і другий застосунок-отримувач
5	Prometheus для збирання метрик Actuator

Питання 10.3. CI-конвеєр у GitHub Actions

Теоретична частина

Неперервна інтеграція (Continuous Integration, CI) — практика, за якої кожна зміна коду автоматично збирається і перевіряється тестами на окремому сервері [69]. Мета проста: дізнатися про помилку через кілька хвилин після коміту, а не через тиждень перед здачею. Наступний крок — неперервне постачання (Continuous Delivery, CD): після успішної перевірки застосунок автоматично готується до розгортання або розгортається.

GitHub Actions — вбудований у GitHub сервіс CI/CD [77]. Конвеєр (workflow) описують файлом YAML у каталозі `.github/workflows`. Його складові:

`on` — події, що запускають конвеєр (push у гілку, pull request, розклад);

`jobs` — завдання, кожне виконується на окремій віртуальній машині (`runs-on`);

`steps` — кроки завдання: готові дії (`uses: actions/checkout@v5`) або команди оболонки (`run: ./mvnw verify`).

На віртуальних машинах GitHub Docker встановлено, тож тести з Testcontainers з питання 8.4 тут працюють без змін — з контейнером, а не з локальною базою. Кешування залежностей Maven (`cache: maven`) скорочує збирання з кількох хвилин до хвилини.

Що варто включати в конвеєр навчального проєкту? Збирання і всі тести (`mvn verify`); звіт про покриття як артефакт; збирання Docker-образу для основної гілки. Для pull request результат конвеєра показується прямо на сторінці запиту — зміни з червоним хрестиком зливати не прийнято. У робочих проєктах це закріплюють правилами захисту гілки.

Ще одна корисна річ — Dependabot. Він щотижня перевіряє, чи вийшли нові версії залежностей і дій, і сам створює pull request з оновленням. Пам'ятаєте пункт «уразливі компоненти» з OWASP Top 10 (питання 7.6)? Автоматичні оновлення — найдешевший захист від нього.

Приклад задачі

Умова. Налаштувати конвеєр GitHub Actions: на кожен push в основну гілку і кожен pull request — збирання на Java 25 з усіма тестами і звітом покриття; для основної гілки додатково — збирання Docker-образу. Увімкнути Dependabot для Maven і GitHub Actions.

Лістинг 10.3. Реалізація з покроковим розбором

Файл `.github/workflows/ci.yml` події і середовище

```yaml
name: CI

on:
  push:
    branches: [main]
  pull_request:

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v5

      - uses: actions/setup-java@v5
        with:
          distribution: temurin
          java-version: 25
          cache: maven
```

Конвеєр запускається для основної гілки і для кожного запиту на злиття. Дія `setup-java` встановлює JDK і кешує каталог `~/.m2`.

```yaml
- name: Збирання і тести
  run: ./mvnw -B verify

- name: Звіт про покриття
  uses: actions/upload-artifact@v4
  with:
    name: jacoco
    path: target/site/jacoco
```

`verify` запускає і модульні тести, і інтеграційні з Testcontainers. Звіт JaCoCo можна завантажити зі сторінки запуску конвеєра.

```yaml
- name: Образ Docker
  if: github.ref == 'refs/heads/main'
  run: docker build -t duty-app:${{ github.sha }} .
```

Образ позначаємо хешем коміту: завжди відомо, з якого саме коду його зібрано. Наступним кроком у реальному проєкті було б надсилання образу в реєстр.

Файл `.github/dependabot.yml` автоматичні оновлення залежностей

```yaml
version: 2
updates:
  - package-ecosystem: maven
    directory: /
    schedule:
      interval: weekly
  - package-ecosystem: github-actions
    directory: /
    schedule:
      interval: weekly
```

З досвіду викладачів. *Одна з найчастіших причин першого червоного конвеєра — файл `mvnw` у репозиторії без права на виконання: крок падає з `Permission denied`. На Windows цей біт загубити найлегше. Лікується командою `git update-index --chmod=+x mvnw` і новим комітом. Тому ми й радимо додавати Maven Wrapper до проєкту одразу — тоді така проблема виявиться в перший же день, а не перед захистом.*

Варіанти для самостійної роботи

Завдання. Налаштувати конвеєр GitHub Actions для репозиторію свого варіанта з указаною додатковою вимогою. Продемонструвати червоний запуск (навмисно зламаний тест) і зелений після виправлення.

№	Додаткова вимога до конвеєра
1	матриця збирання на Java 21 і Java 25
2	перевірка мінімального покриття JaCoCo (`check`), збирання падає нижче 60 %
3	значок статусу збирання (badge) у файлі `README.md`
4	публікація Docker-образу в GitHub Container Registry для тегів `v*`
5	окреме завдання для інтеграційних тестів, що виконується після модульних

Питання 10.4. Конфігурація для production, секрети, змінні середовища

Теоретична частина

Робоче середовище (production) відрізняється від середовища розробки не лише адресою бази даних. Тут працюють справжні користувачі, тут небезпечно показувати зайву інформацію, тут застосунок перезапускають під час оновлень, а логи читають не очима, а системами збору. Конфігурацію для нього зручно тримати в окремому профілі `prod` (питання 2.3), а все, що відрізняється між конкретними серверами, передавати змінними середовища [24].

Ось що зазвичай входить у профіль `prod`:

вимкнені Swagger UI і опис API (або доступні лише адміністратору);

деталі стану Actuator — лише для автентифікованих користувачів (`show-details: when-authorized`);

плавна зупинка (`server.shutdown: graceful`) — під час зупинки застосунок перестає приймати нові запити, але завершує ті, що вже обробляються;

проби liveness і readiness (`management.endpoint.health.probes.enabled`) для оркестраторів на кшталт Kubernetes: перша показує, що процес не завис, друга — що застосунок готовий приймати трафік;

структуровані логи у форматі JSON (`logging.structured.format.console`), які системи на кшталт ELK чи Grafana Loki розбирають без регулярних виразів;

`server.forward-headers-strategy` — коли застосунок стоїть за зворотним проксі (nginx, балансувальник), щоб правильно формувалися адреси на кшталт заголовка `Location` (питання 3.4).

Окремо про секрети: паролі до бази, ключ підпису JWT, ключі зовнішніх API. Правила прості й жорсткі [31]. Секрети ніколи не потрапляють у репозиторій — ні в `application.yml`, ні в `compose.yaml`, ні в історію Git. Значення за замовчуванням для секретів у коді допустимі лише для розробки і мають бути очевидно непридатними (`dev-only-secret-change-me`). У робочому середовищі секрети передаються змінними середовища або зі спеціальних сховищ: Docker secrets, Kubernetes Secrets, HashiCorp Vault, сховища хмарних провайдерів.

Якщо секрет все-таки потрапив у репозиторій, видалити файл новим комітом недостатньо — він залишився в історії. Єдина правильна реакція — негайно замінити (ротувати) секрет. Сервіси на кшталт GitHub сканують публічні репозиторії й попереджають про такі витоки.

Пам’ятайте про час: сервер у контейнері працює в UTC (і наш досвід з питання 5.1 показує, що часові пояси підступні). Тому ми з першого розділу беремо час через бін `Clock` з явним поясом `Europe/Kyiv`, а в базі зберігаємо його однозначно.

Приклад задачі

Умова. Підготувати профіль `prod` для застосунку чергової частини: плавна зупинка, проби стану, обмежені деталі Actuator, вимкнений Swagger, структуровані логи, коректна робота за проксі. Перелічити змінні середовища, які обов'язково задають на сервері.

Лістинг 10.4. Реалізація з покроковим розбором

Файл `application-prod.yml` сервер і життєвий цикл

```
server:
  shutdown: graceful
  forward-headers-strategy: framework

spring:
  lifecycle:
    timeout-per-shutdown-phase: 20s
```

Під час `docker compose down` чи оновлення застосунок має до 20 секунд, щоб завершити поточні запити. Черговий не втратить виклик, який саме реєстрував.

Файл `application-prod.yml` Actuator і документація

```
management:
  endpoint:
    health:
      show-details: when-authorized
      probes:
        enabled: true
```

З'являються адреси `/actuator/health/liveness` і `/actuator/health/readiness`. Деталі (зокрема `readyTrucks`) бачить лише адміністратор.

```
springdoc:
  api-docs:
    enabled: false
  swagger-ui:
    enabled: false
```

Документацію API вимкнено зовсім — вона потрібна розробникам, а не робочому серверу.

Файл `application-prod.yml` структуровані логи

```
logging:
  structured:
    format:
      console: ecs
```

Кожен рядок логу — об'єкт JSON у форматі Elastic Common Schema з часом, рівнем, логером, потоком і повідомленням. Підтримку структурованих логів додано в Spring Boot 3.4.

Вивід програми приклад рядка логу в профілі prod (скорочено)

```
{"@timestamp":"2026-10-23T08:15:02.114Z","log.level":"INFO",
 "process.thread.name":"tomcat-handler-3",
 "log.logger":"ua.edu.ldubgd.duty.call.DispatchService",
 "message":"Зареєстровано виклик №42","ecs.version":"8.11"}
```

Змінні середовища що задають на сервері

```
SPRING_PROFILES_ACTIVE=prod
DB_URL=jdbc:postgresql://db.ldubgd.example:5432/duty
DB_USER=duty
DB_PASSWORD=<із сховища секретів>
JWT_SECRET=<щонайменше 32 випадкові байти, із сховища секретів>
SPRING_RABBITMQ_HOST=rabbit.ldubgd.example
DUTY_MESSAGING_ENABLED=true
```

Порада. *Перевірте себе командою* `git log -p | grep -i -E "password|secret"` *у власному репозиторії. Якщо знайдеться щось, крім навчальних значень на кшталт* `dev-only-secret-change-me`*, — саме час прочитати документацію GitHub про видалення чутливих даних з історії і замінити скомпрометовані секрети. Краще дізнатися про це від себе, ніж від автоматичного попередження.*

Варіанти для самостійної роботи

Завдання. Підготувати профіль `prod` і перелік змінних середовища для застосунку свого варіанта. Додати вказане налаштування і перевірити його роботу в Docker Compose.

№	Додаткове налаштування для робочого середовища
1	логи у файл з ротацією (розмір 10 МБ, зберігати 7 файлів)
2	ендпоінт `/actuator/info` з версією збирання і хешем коміту Git
3	HTTPS із сертифікатом через `server.ssl.*` (самопідписаний для навчання)
4	обмеження пулу з'єднань HikariCP і тайм-аут запитів до бази
5	читання секретів з файлів (`spring.config.import=configtree:/run/secrets/`)

Розділ 11. Spring і штучний інтелект

Великі мовні моделі (Large Language Models, LLM) за кілька років стали звичайним інструментом розробника — так само, як колись бази даних чи вебсервіси. Для Junior-розробника сьогодні важливі дві речі: уміти вбудувати модель у застосунок і вміти відповідально працювати з асистентом для коду. Обом присвячено цей розділ. Приклади використовують модель Claude від компанії Anthropic [78]. Для запуску потрібен ключ API у змінній середовища `ANTHROPIC_API_KEY`; без нього застосунок стартує, але звернення до помічника завершаться помилкою.

Питання 11.1. Spring AI: ChatClient і підключення моделі Claude

Теоретична частина

Мовна модель доступна застосунку як звичайний вебсервіс: ми надсилаємо HTTP-запит із повідомленнями (системними інструкціями і текстом користувача) і отримуємо відповідь. Для Claude це Messages API [78]. Можна працювати з ним через `RestClient` (питання 9.1) або офіційний Java SDK від Anthropic, але для застосунків Spring зручніший проєкт Spring AI [79].

Spring AI робить для мовних моделей те саме, що Spring Data для баз даних: дає єдину абстракцію над різними постачальниками. Головний інтерфейс — `ChatClient` з «текучим» API: `prompt()`, `system(...)`, `user(...)`, `call()`. Відповідь можна отримати як текст (`content()`) або одразу як об'єкт Java (`entity(Клас.class)`): Spring AI сам додасть до запиту опис потрібного формату і розбере JSON у відповіді. Стартер `spring-ai-starter-model-anthropic` автоматично налаштовує клієнт для Claude, а модель, максимальну довжину відповіді та інші параметри задають властивостями [80].

Що варто знати про роботу з моделлю, щоб уникнути типових помилок.

Модель не пам’ятає попередніх запитів. Кожен запит містить усе, що модель має врахувати. Історію розмови Spring AI вміє зберігати і додавати сам (`ChatMemory`), але це робота застосунку, а не моделі.

Системний промпт задає роль і правила для всіх запитів. Писати його варто як інструкцію для нового колеги: хто він, що робить, що робити, коли даних бракує [81]. Для нашої системи особливо важливе правило: помічник лише радить, рішення ухвалює черговий.

Вартість і швидкість. Оплата йде за токени (частини слів) у запиті й відповіді, тож довжину відповіді обмежують, а запити до моделі не роблять у «гарячих» циклах. Відповідь може займати секунди — це ще один аргумент на користь тайм-аутів і запасних результатів з питання 9.5.

Безпека даних. Текст, надісланий моделі, виходить за межі вашої системи. Персональні дані заявників (телефони, прізвища) туди не передаємо — у прикладі помічник бачить лише номер, тип і адресу виклику [29]. Ключ API — такий самий секрет, як пароль до бази (питання 10.4).

І головне для служби цивільного захисту: модель може помилятися впевнено і переконливо. Тому вона пропонує, а людина вирішує і несе відповідальність.

Приклад задачі

Умова. Підключити до проєкту Spring AI з моделлю Claude і створити службу `DutyAssistant`, яка за даними виклику складає коротку довідку для чергового: суть, ризики, який караул запропонувати. Відповідь отримувати як запис Java. Відкрити кінцеву точку лише для чергових і адміністратора.

Лістинг 11.1. Реалізація з покроковим розбором

Файл `pom.xml` Spring AI

```
<dependency>
    <groupId>org.springframework.ai</groupId>
    <artifactId>spring-ai-starter-model-anthropic</artifactId>
</dependency>
```

Версію задає імпортований BOM `spring-ai-bom` (у розділі `dependencyManagement`). Лінійка Spring AI 2.x сумісна зі Spring Boot 4.

Файл `application.yml` модель і ключ

```
ai:
  anthropic:
    api-key: ${ANTHROPIC_API_KEY:not-set}
    chat:
      options:
        model: claude-opus-5-5
        max-tokens: 2000
```

Ключ — лише зі змінної середовища. Значення `not-set` дозволяє застосунку стартувати без ключа (наприклад, на лабораторних без доступу до API).

Файл `DutyAssistant.java` системний промпт

```
private static final String SYSTEM = """
        Ти — помічник чергового пожежно-рятувальної
        частини ЛДУБЖД.
        Відповідай українською, коротко і по суті.
        Ти лише радиш: рішення про виїзд ухвалює черговий.
        Якщо даних бракує, так і скажи, нічого не вигадуй.""";
```

Роль, мова, межі відповідальності і поведінка в разі нестачі даних. Остання вимога знижує ризик «вигаданих» фактів.

```
public DutyAssistant(ChatClient.Builder builder,
                     DutyTools tools) {
    this.chat = builder.defaultSystem(SYSTEM)
            .defaultTools(tools)
            .build();
}
```

`ChatClient.Builder` створює автоконфігурація. Інструменти `DutyTools` розглянемо в питанні 11.2.

```
public CallBrief brief(FireCall call) {
    return chat.prompt()
            .user(u -> u.text("""
                    Виклик №{id}: {type}, адреса: {address}.
                    Склади коротку довідку для чергового:
                    що врахувати на місці і який караул
                    запропонувати.""")
                    .param("id", call.getId())
                    .param("type", call.getType())
                    .param("address", call.getAddress()))
            .call()
            .entity(CallBrief.class);
}
```

Шаблон із параметрами замість склеювання рядків. До моделі йдуть лише номер, тип і адреса — телефону заявника там немає. `entity` перетворить відповідь на запис `CallBrief` з полями `summary`, `risks`, `suggestedCrew`.

Файл `AssistantController.java` доступ лише для чергових

```
@RestController
@RequestMapping("/api/assistant")
@PreAuthorize("hasAnyRole('DISPATCHER', 'ADMIN')")
public class AssistantController {
```

`@PreAuthorize` на класі діє для всіх методів. Кожен запит коштує грошей, тож відкривати помічника всім ролям немає сенсу.

Вивід програми приклад відповіді GET /api/assistant/calls/1/brief (зміст щоразу інший)

```
{"summary":"Пожежа за адресою вул. Зелена, 101.",
 "risks":"Західний вітер 11 км/год: дим іде на сусідні будинки.",
 "suggestedCrew":"1-й караул (АЦ-40), автомобіль у резерві"}
```

Зверніть увагу. *Відповідь моделі недетермінована: той самий запит може дати трохи інший текст. Тому тести служб, що звертаються до моделі, не перевіряють дослівний текст. Модульні тести використовують мок `ChatClient` (питання 8.1), а якість відповідей оцінюють окремим набором прикладів з очікуваними властивостями: «згадано вітер», «запропоновано вільний караул», «немає вигаданих адрес».*

Варіанти для самостійної роботи

Завдання. Створити службу на основі `ChatClient` для свого варіанта, яка повертає структуровану відповідь (запис Java). Написати системний промпт з роллю, мовою і межами відповідальності. Переконатися, що в модель не передаються персональні дані.

№	Задача помічника	Поля відповіді
1	чернетка рапорту про виїзд за даними виклику	`title`, `text`, `missingData`
2	пояснення несправності автомобіля простою мовою	`problem`, `urgency`, `nextStep`
3	план підготовки курсанта до наряду	`tasks`, `checklist`, `tips`
4	висновок за результатами перевірки гідрантів району	`summary`, `brokenCount`, `priority`
5	обґрунтування заявки на поповнення складу	`items`, `reason`, `deadline`

Питання 11.2. Виклик інструментів (tool calling)

Теоретична частина

Модель знає лише те, що є в її навчальних даних і в поточному запиті. Вона не знає, які караули зараз у резерві і яка погода у Львові. Можна заздалегідь вкласти ці дані в запит, але часто незрозуміло, які саме знадобляться. Механізм виклику інструментів (tool use, function calling) розв'язує це інакше: ми описуємо моделі функції, які вона може «попросити» виконати, а застосунок їх виконує і повертає результат [82].

Схема виглядає так:

1) застосунок надсилає запит разом з описами інструментів: назва, призначення, параметри;

2) модель вирішує, що їй потрібні дані, і замість відповіді повертає запит на виклик інструмента з конкретними аргументами;

3) застосунок виконує свій метод — звертається до бази, до сервісу погоди — і надсилає результат моделі;

4) модель або просить ще один інструмент, або формує остаточну відповідь.

Важливо: модель не виконує код і не має доступу до бази. Вона лише просить, а виконує і контролює все наш застосунок. Тому інструменти — це межа безпеки. Ми самі вирішуємо, які дії доступні моделі. Інструменти для читання (вільні караули, погода) — безпечні. Інструмент, який змінює дані (призначити караул, закрити виклик), вимагає підтвердження людиною — або взагалі не надається моделі.

У Spring AI інструмент — звичайний метод біна з анотацією `@Tool`. Опис (`description`) модель читає, щоб вирішити, коли викликати інструмент, тож писати його треба так само уважно, як системний промпт. Параметри описують анотацією `@ToolParam`. Spring AI сам побудує схему параметрів, виконає метод і поверне результат моделі; цикл «запит — інструмент — запит» ми не пишемо вручну.

Застосунок, у якому модель у циклі сама обирає інструменти для досягнення мети, називають агентом. Anthropic радить починати з найпростішого рішення і переходити до агентів лише тоді, коли задача справді цього потребує [83]. Для ширшого підключення інструментів існує відкритий стандарт Model Context Protocol (MCP), що дозволяє описати інструменти один раз і використовувати їх у різних застосунках [84]. Spring AI підтримує MCP як клієнт і як сервер.

Приклад задачі

Умова. Дати помічнику чергового три інструменти тільки для читання: список караулів, чий автомобіль у резерві; виклики з певним статусом; поточна погода у Львові. Перевірити, що модель сама звертається до потрібних інструментів, коли формує довідку.

Лістинг 11.2. Реалізація з покроковим розбором

Файл `DutyTools.java` інструменти

```
@Tool(description = "Караули, чий автомобіль зараз у резерві")
@Transactional(readOnly = true)
public List<CrewInfo> freeCrews() {
    return crews.findByNameStartingWith("").stream()
            .filter(c -> c.getTruck().isAvailable())
            .map(c -> new CrewInfo(c.getId(), c.getName(),
                    c.getTruck().getModel(),
                    c.getMembers().size()))
            .toList();
}
```

Звичайний метод біна. Повертаємо DTO, а не сутність: модель отримає лише назву караулу, модель автомобіля і кількість осіб. `@EntityGraph` з питання 5.4 уникає проблеми N+1.

```
@Tool(description = "Незакриті виклики з указаним статусом")
public List<CallSummary> callsByStatus(
        @ToolParam(description = "NEW, DISPATCHED або ON_SITE")
        CallStatus status) {
    return calls.findByStatus(status);
}
```

Опис параметра підказує моделі допустимі значення. Інтерфейсна проєкція з питання 5.6 не містить телефонів заявників.

```
@Tool(description = "Поточна погода у Львові")
public WeatherReport currentWeather() {
    return weather.lviv();
}
```

Захищений метод з питання 9.5. Якщо сервіс погоди недоступний, модель отримає «недоступно» і, згідно із системним промптом, так і скаже.

Лог послідовність викликів під час формування довідки (приклад)

```
-> модель: довідка за викликом №1 (FIRE, вул. Зелена, 101)
<- модель просить: currentWeather()
-> застосунок: {"temperature":9.8,"windSpeed":11.5,"windDirection":264}
<- модель просить: freeCrews()
-> застосунок: [{"id":1,"name":"1-й караул","truck":"АЦ-40","members":3}, ...]
<- модель: остаточна відповідь CallBrief
```

З досвіду викладачів. *Спокусливо додати інструмент «зробити все»: `executeSql(String sql)`, «щоб модель сама розібралася». Це прямий шлях до біди: будь-який текст, що потрапив у запит (зокрема адреса виклику, яку ввела людина), може містити інструкцію на кшталт «видали всі записи» — це називають ін'єкцією промпту. Давайте моделі вузькі інструменти з чіткими параметрами і лише ті права, які справді потрібні, — той самий принцип найменших привілеїв, що й у питанні 7.3.*

Варіанти для самостійної роботи

Завдання. Описати 2–3 інструменти для помічника свого варіанта з указаними обмеженнями. Перевірити, які інструменти модель викликає для різних запитань, і пояснити, чому дія, що змінює дані, вимагає підтвердження людини.

№	Інструменти для читання	Дія, яку модель лише пропонує
1	склад караулу; стаж рятувальника	переведення в інший караул
2	стан автомобіля; дата останнього ТО	постановка на ТО
3	графік нарядів; бали курсанта	заміна курсанта в наряді
4	гідранти району; результати перевірок	позначка «несправний»
5	залишки на складі; норми видачі	оформлення видачі

Питання 11.3. Робота з AI-асистентами для коду: можливості, межі, відповідальність

Теоретична частина

Асистенти для коду — Claude Code, вбудовані асистенти IDE та інші — сьогодні є частиною звичайного робочого місця розробника [7; 2]. Вони пояснюють незнайомий код, пишуть тести, шукають причину помилки за трасуванням стека, перекладають код зі старої версії бібліотеки на нову. Claude Code працює просто в терміналі або IDE: читає файли проєкту, запускає команди (`mvn test`, `git diff`), пропонує й вносить зміни.

Що з цього випливає для Junior-розробника? Ми бачимо три практичні правила.

Перше — ви відповідаєте за код незалежно від того, хто його написав. На код-рев'ю «так запропонував асистент» не є аргументом. Якщо ви не можете пояснити рядок коду, його не можна комітити. Асистент вільно посилається на методи, яких немає в бібліотеці, використовує застарілі API (у цій книжці ми бачили, скільки змінилося між Spring Boot 3 і 4) і може пропустити крайові випадки.

Друге — асистент працює краще з хорошим контекстом. Чітке формулювання задачі, посилання на конкретні файли, відомі обмеження («Spring Boot 4.1, не використовуй `RestTemplate`»), критерій готовності («усі тести мають пройти») дають кращий результат, ніж «зроби, щоб працювало» [81]. Файл `CLAUDE.md` у корені репозиторію дозволяє записати ці домовленості один раз: як збирати проєкт, які правила оформлення коду, що не можна змінювати.

Третє — перевіряйте автоматично. Тести з розділу 8, CI з розділу 10, перевірка лістингів у цьому посібнику — найкращий захист від впевнених помилок як людини, так і моделі. Попросіть асистента спершу написати тест, що відтворює помилку, а потім виправити код — так видно, що виправлення справді працює.

Окремо — про безпеку. Не передавайте асистентам секрети і персональні дані; переглядайте команди, які асистент збирається виконати; не вмикайте автоматичне виконання всього підряд у репозиторії з доступом до робочих систем.

І про навчання. Асистент може зробити лабораторну за вас, але на співбесіді його поруч не буде, а на роботі вам доведеться розуміти код, який він пише. Використовуйте його як терплячого старшого колегу, який пояснює, — а не як заміну власному розумінню. Ми свідомо залишили в посібнику примітки «З досвіду викладачів» про помилки, на які натрапили самі, готуючи приклади: часто саме налагодження вчить найбільше.

Приклад задачі

Умова. Сформулювати для асистента для коду задачу з розширення наскрізного проєкту так, щоб результат можна було перевірити автоматично. Підготувати файл `CLAUDE.md` з домовленостями проєкту і критично переглянути запропоновані зміни.

Лістинг 11.3. Реалізація з покроковим розбором

Файл `CLAUDE.md` домовленості проєкту для асистента

```
# Проєкт duty-app — інформаційна система чергової частини ЛДУБЖД

## Збирання і перевірка
- Java 25, Spring Boot 4.1, Maven: `./mvnw verify`
- Тести бази: Testcontainers; без Docker —
  `DUTY_TEST_CONTAINERS=false DB_URL=... ./mvnw verify`

## Правила
- Пакети за функціями (call, crew, fleet, ...), DTO — записи Java
- Залежності лише через конструктор, час — через бін Clock
- Схема бази змінюється лише новою міграцією Flyway
- Не передавати в API, логи і моделі телефон заявника (callerPhone)
- Не змінювати SecurityConfig без окремого тесту в CallSecurityTest
```

Запит до асистента приклад добре сформульованої задачі

```
Додай до API статистику викликів за останні N днів:
GET /api/calls/stats/daily?days=7 → [{date, count}], days від 1 до 30.
- запит у CallRepository на JPQL (без нативного SQL);
- доступ — ролі DISPATCHER і ADMIN;
- тести: репозиторій (@DataJpaTest), контролер (@WebMvcTest, 200 і 400
  для days=0), безпека (403 для RESCUER);
- готово, коли ./mvnw verify проходить; покажи git diff перед комітом.
```

Порада. *Перед тим як прийняти зміни від асистента, поставте собі три запитання. Чи розумію я кожен рядок? Чи є тест, який упаде, якщо цей код зламати? Чи не з'явилося в диффі нічого, про що я не просив, — нових залежностей, змінених налаштувань безпеки, вимкнених тестів? Якщо хоч на одне питання відповідь «ні», зміни ще не готові.*

Варіанти для самостійної роботи

Завдання. Сформулювати для асистента для коду задачу з розширення застосунку свого варіанта з чіткими обмеженнями і критерієм готовності. Виконати задачу з асистентом, а у звіті навести: запит, отриманий дифф, знайдені в ньому помилки або зайві зміни і як їх виправлено.

№	Задача для асистента	Обов'язкова перевірка
1	пошук рятувальників за частиною прізвища	тест на прізвища з апострофом
2	історія ТО автомобіля з пагінацією	тест для порожньої історії
3	експорт графіка нарядів у CSV	тест кодування UTF-8 і роздільника
4	карта несправних гідрантів у форматі GeoJSON	тест на некоректні координати
5	звіт про залишки нижче мінімуму	тест безпеки для ролі RESCUER

ВИСНОВКИ

У посібнику розглянуто 62 питання, які найчастіше трапляються на технічних співбесідах на позицію Java Junior Developer у частині розробки вебзастосунків на платформі Spring. Матеріал охоплює ядро Spring і Spring Boot, вебрівень на Spring MVC, роботу з даними через Spring Data JPA, кешування, асинхронність і події, безпеку зі Spring Security, автоматичне тестування, взаємодію з іншими сервісами, розгортання в контейнерах і використання великих мовних моделей у застосунках.

Головна особливість посібника — наскрізний проєкт. Інформаційна система чергової частини ЛДУБЖД почалася в першому розділі з кількох класів у пам'яті й у підсумку має REST API і сторінки для чергового, базу даних PostgreSQL з міграціями, ролі й токени JWT, понад два десятки автоматичних тестів, обмін повідомленнями, живе табло, захист від збоїв зовнішніх сервісів, Docker-образ, конвеєр CI і помічника на основі моделі Claude. Кожен лістинг книжки звірено з кодом проєкту, а стан коду після кожного розділу збережено окремою міткою в репозиторії. Тож здобувач освіти може не лише прочитати про технологію, а й відкрити, запустити і змінити код саме тієї версії, що відповідає розділу.

Ми свідомо залишили в тексті примітки «З досвіду викладачів» про помилки, на які натрапили самі, готуючи приклади: зниклий апостроф у повідомленні валідації, роль `R_PASSWORD` у токені, повтори, що не повторювали, звіт про покриття, якого не було. Ці історії — не випадкові. Вони показують, що навіть простий код поводиться несподівано, і що найкращий захист від цього — автоматичні перевірки й звичка читати повідомлення про помилки до кінця.

Варіанти для самостійної роботи побудовано так, щоб кожен здобувач протягом семестру створив власну підсистему на тій самій предметній області: облік рятувальників, автопарк, наряди курсантів, протипожежне водопостачання чи матеріально-технічне забезпечення. Виконані послідовно, варіанти одного номера складаються в завершений застосунок, який не соромно показати роботодавцю як навчальний проєкт у портфоліо.

Spring розвивається швидко. Між виходом Spring Boot 3 і Spring Boot 4 змінилися назви стартерів і пакетів, з'явилися віртуальні потоки, вбудована підтримка версіонування API і стійкості. Конкретні назви класів у цій книжці з часом застаріють, але принципи залишаться: впровадження залежностей, тонкий вебрівень, транзакції на рівні служб, явні межі безпеки, тести як частина коду. Тому радимо звертатися до офіційної документації Spring [57; 21] як до основного джерела, а до цієї книжки — як до карти, що допомагає в ній зорієнтуватися.

І наостанок — побажання від авторів, яке ми вже висловлювали в першій книжці і повторимо тут. Не поспішайте. Краще одне питання на день, але з власним кодом, власними помилками і власним розумінням, ніж увесь посібник за тиждень з асистентом, який думає за вас. На співбесіді й на роботі цінується саме це розуміння.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Головатий Р. Р., Хлевной О. В., Назар Ю. С., Смотр О. О., Борзов Ю. О., Райта Д. А. Основи програмування мовою Java : збірник задач для підготовки до технічних співбесід (рівень Trainee) : навч. посіб. Львів : ЛДУБЖД, 2026. 140 с.
2. The State of Developer Ecosystem 2025. JetBrains. URL: https://www.jetbrains.com/lp/devecosystem-2025/ (дата звернення: 20.10.2026).
3. Кодекс цивільного захисту України : Кодекс України від 02.10.2012 № 5403-VI. URL: https://zakon.rada.gov.ua/laws/show/5403-17 (дата звернення: 20.10.2026).
4. Spring Boot 4.0 Release Notes. URL: https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-4.0-Release-Notes (дата звернення: 20.10.2026).
5. What's New in Spring Framework 7.0. URL: https://github.com/spring-projects/spring-framework/wiki/What%27s-New-in-Spring-Framework-7.x (дата звернення: 20.10.2026).
6. ДСТУ 8302:2015. Інформація та документація. Бібліографічне посилання. Загальні положення та правила складання. [Чинний від 2016-07-01]. Вид. офіц. Київ : ДП «УкрНДНЦ», 2016. 16 с.
7. Claude Code Docs. Overview. Anthropic. URL: https://docs.claude.com/en/docs/claude-code/overview (дата звернення: 20.10.2026).
8. Fowler M. Inversion of Control Containers and the Dependency Injection pattern. 2004. URL: https://martinfowler.com/articles/injection.html (дата звернення: 20.10.2026).
9. van Deursen S., Seemann M. Dependency Injection Principles, Practices, and Patterns. Shelter Island : Manning, 2019. 552 p.
10. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Boston : Prentice Hall, 2017. 432 p.
11. Spring Framework Documentation. The IoC Container. URL: https://docs.spring.io/spring-framework/reference/core/beans.html (дата звернення: 20.10.2026).
12. Spring Framework Documentation. Annotation-based Container Configuration. URL: https://docs.spring.io/spring-framework/reference/core/beans/annotation-config.html (дата звернення: 20.10.2026).
13. JEP 395: Records. OpenJDK. URL: https://openjdk.org/jeps/395 (дата звернення: 20.10.2026).
14. Walls C. Spring in Action. 6th ed. Shelter Island : Manning, 2022. 520 p.
15. Spring Framework Documentation. Bean Scopes. URL: https://docs.spring.io/spring-framework/reference/core/beans/factory-scopes.html (дата звернення: 20.10.2026).
16. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Reading : Addison-Wesley, 1994. 395 p.
17. Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D. Java Concurrency in Practice. Boston : Addison-Wesley, 2006. 384 p.
18. Spring Framework Documentation. Customizing the Nature of a Bean. URL: https://docs.spring.io/spring-framework/reference/core/beans/factory-nature.html (дата звернення: 20.10.2026).
19. Jakarta Annotations 3.0 Specification. Eclipse Foundation. URL: https://jakarta.ee/specifications/annotations/3.0/ (дата звернення: 20.10.2026).
20. Spring Framework Documentation. Java-based Container Configuration. URL: https://docs.spring.io/spring-framework/reference/core/beans/java.html (дата звернення: 20.10.2026).
21. Spring Boot Reference Documentation. URL: https://docs.spring.io/spring-boot/index.html (дата звернення: 20.10.2026).
22. Heckler M. Spring Boot: Up and Running. Sebastopol : O'Reilly Media, 2021. 328 p.

23. Spring Boot Reference Documentation. Creating Your Own Auto-configuration. URL: https://docs.spring.io/spring-boot/reference/features/developing-auto-configuration.html (дата звернення: 20.10.2026).
24. Wiggins A. The Twelve-Factor App. URL: https://12factor.net (дата звернення: 20.10.2026).
25. Spring Boot Reference Documentation. Externalized Configuration. URL: https://docs.spring.io/spring-boot/reference/features/external-config.html (дата звернення: 20.10.2026).
26. Spring Initializr. URL: https://start.spring.io (дата звернення: 20.10.2026).
27. Apache Maven Project. Introduction to the Build Lifecycle. URL: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html (дата звернення: 20.10.2026).
28. SLF4J User Manual. URL: https://www.slf4j.org/manual.html (дата звернення: 20.10.2026).
29. Про захист персональних даних : Закон України від 01.06.2010 № 2297-VI. URL: https://zakon.rada.gov.ua/laws/show/2297-17 (дата звернення: 20.10.2026).
30. Spring Boot Reference Documentation. Production-ready Features (Actuator). URL: https://docs.spring.io/spring-boot/reference/actuator/index.html (дата звернення: 20.10.2026).
31. OWASP Top 10:2021. The Open Worldwide Application Security Project. URL: https://owasp.org/Top10/ (дата звернення: 20.10.2026).
32. Micrometer Documentation. URL: https://docs.micrometer.io (дата звернення: 20.10.2026).
33. Jakarta Servlet 6.1 Specification. Eclipse Foundation. URL: https://jakarta.ee/specifications/servlet/6.1/ (дата звернення: 20.10.2026).
34. Fowler M. Patterns of Enterprise Application Architecture. Boston : Addison-Wesley, 2002. 560 p.
35. Spring Framework Documentation. Spring Web MVC. URL: https://docs.spring.io/spring-framework/reference/web/webmvc.html (дата звернення: 20.10.2026).
36. Fielding R., Nottingham M., Reschke J. HTTP Semantics : RFC 9110. IETF, 2022. URL: https://www.rfc-editor.org/rfc/rfc9110 (дата звернення: 20.10.2026).
37. Jakarta Validation 3.1 Specification. Eclipse Foundation. URL: https://jakarta.ee/specifications/bean-validation/3.1/ (дата звернення: 20.10.2026).
38. Nottingham M., Wilde E., Dalal S. Problem Details for HTTP APIs : RFC 9457. IETF, 2023. URL: https://www.rfc-editor.org/rfc/rfc9457 (дата звернення: 20.10.2026).
39. Spring Data Commons Reference Documentation. Auditing. URL: https://docs.spring.io/spring-data/commons/reference/auditing.html (дата звернення: 20.10.2026).
40. Kleppmann M. Designing Data-Intensive Applications. Sebastopol : O'Reilly Media, 2017. 616 p.
41. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : dissertation … Doctor of Philosophy in Information and Computer Science. Irvine : University of California, 2000. 162 p.
42. Richardson L., Amundsen M., Ruby S. RESTful Web APIs. Sebastopol : O'Reilly Media, 2013. 406 p.
43. Lauret A. The Design of Web APIs. Shelter Island : Manning, 2019. 392 p.
44. OpenAPI Specification v3.1.1. OpenAPI Initiative. URL: https://spec.openapis.org/oas/v3.1.1.html (дата звернення: 20.10.2026).
45. springdoc-openapi: OpenAPI 3 Library for Spring Boot. URL: https://springdoc.org (дата звернення: 20.10.2026).
46. Thymeleaf. Tutorial: Thymeleaf + Spring. URL: https://www.thymeleaf.org/doc/tutorials/3.1/thymeleafspring.html (дата звернення: 20.10.2026).
47. Fetch Standard. CORS protocol. WHATWG. URL: https://fetch.spec.whatwg.org/#http-cors-protocol (дата звернення: 20.10.2026).
48. Spring Data JPA Reference Documentation. URL: https://docs.spring.io/spring-data/jpa/reference/ (дата звернення: 20.10.2026).

49. Jakarta Persistence 3.2 Specification. Eclipse Foundation. URL: https://jakarta.ee/specifications/persistence/3.2/ (дата звернення: 20.10.2026).
50. Hibernate ORM User Guide. URL: https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html (дата звернення: 20.10.2026).
51. Mihalcea V. High-Performance Java Persistence. Cluj-Napoca : Vlad Mihalcea, 2016. 470 p.
52. Spring Framework Documentation. Data Access: Transaction Management. URL: https://docs.spring.io/spring-framework/reference/data-access/transaction.html (дата звернення: 20.10.2026).
53. Flyway Documentation. Redgate. URL: https://documentation.red-gate.com/flyway (дата звернення: 20.10.2026).
54. Spring Framework Documentation. Integration: REST Clients, Task Execution and Scheduling, Cache Abstraction. URL: https://docs.spring.io/spring-framework/reference/integration.html (дата звернення: 20.10.2026).
55. JEP 444: Virtual Threads. OpenJDK. URL: https://openjdk.org/jeps/444 (дата звернення: 20.10.2026).
56. Richardson C. Microservices Patterns: With Examples in Java. Shelter Island : Manning, 2018. 520 p.
57. Spring Framework Documentation. Core Technologies. URL: https://docs.spring.io/spring-framework/reference/core.html (дата звернення: 20.10.2026).
58. Spring Security Reference Documentation. Servlet Applications: Architecture. URL: https://docs.spring.io/spring-security/reference/servlet/architecture.html (дата звернення: 20.10.2026).
59. Spring Security Reference Documentation. URL: https://docs.spring.io/spring-security/reference/ (дата звернення: 20.10.2026).
60. Madden N. API Security in Action. Shelter Island : Manning, 2020. 576 p.
61. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT) : RFC 7519. IETF, 2015. URL: https://www.rfc-editor.org/rfc/rfc7519 (дата звернення: 20.10.2026).
62. Hardt D. The OAuth 2.0 Authorization Framework : RFC 6749. IETF, 2012. URL: https://www.rfc-editor.org/rfc/rfc6749 (дата звернення: 20.10.2026).
63. OpenID Connect Core 1.0. OpenID Foundation. URL: https://openid.net/specs/openid-connect-core-1_0.html (дата звернення: 20.10.2026).
64. Khorikov V. Unit Testing Principles, Practices, and Patterns. Shelter Island : Manning, 2020. 304 p.
65. JUnit User Guide. URL: https://docs.junit.org/current/user-guide/ (дата звернення: 20.10.2026).
66. Mockito Framework Documentation. URL: https://site.mockito.org (дата звернення: 20.10.2026).
67. Spring Framework Documentation. Testing. URL: https://docs.spring.io/spring-framework/reference/testing.html (дата звернення: 20.10.2026).
68. Testcontainers for Java Documentation. URL: https://java.testcontainers.org (дата звернення: 20.10.2026).
69. Humble J., Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Boston : Addison-Wesley, 2010. 512 p.
70. RabbitMQ Tutorials. Broadcom. URL: https://www.rabbitmq.com/tutorials (дата звернення: 20.10.2026).
71. Fette I., Melnikov A. The WebSocket Protocol : RFC 6455. IETF, 2011. URL: https://www.rfc-editor.org/rfc/rfc6455 (дата звернення: 20.10.2026).
72. Spring Framework Documentation. WebSockets. URL: https://docs.spring.io/spring-framework/reference/web/websocket.html (дата звернення: 20.10.2026).
73. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol : O’Reilly Media, 2021. 612 p.

74. Nygard M. T. Release It! Design and Deploy Production-Ready Software. 2nd ed. Raleigh : Pragmatic Bookshelf, 2018. 376 p.
75. Resilience4j Documentation. URL: https://resilience4j.readme.io (дата звернення: 20.10.2026).
76. Docker Docs. Docker Inc. URL: https://docs.docker.com (дата звернення: 20.10.2026).
77. GitHub Actions Documentation. GitHub. URL: https://docs.github.com/actions (дата звернення: 20.10.2026).
78. Claude Docs. Messages API. Anthropic. URL: https://docs.claude.com/en/api/messages (дата звернення: 20.10.2026).
79. Spring AI Reference Documentation. URL: https://docs.spring.io/spring-ai/reference/ (дата звернення: 20.10.2026).
80. Spring AI Reference Documentation. Anthropic Chat. URL: https://docs.spring.io/spring-ai/reference/api/chat/anthropic-chat.html (дата звернення: 20.10.2026).
81. Claude Docs. Prompt engineering overview. Anthropic. URL: https://docs.claude.com/en/docs/build-with-claude/prompt-engineering/overview (дата звернення: 20.10.2026).
82. Claude Docs. Tool use with Claude. Anthropic. URL: https://docs.claude.com/en/docs/agents-and-tools/tool-use/overview (дата звернення: 20.10.2026).
83. Schluntz E., Zhang B. Building effective agents. Anthropic, 2024. URL: https://www.anthropic.com/engineering/building-effective-agents (дата звернення: 20.10.2026).
84. Model Context Protocol Specification. URL: https://modelcontextprotocol.io/specification (дата звернення: 20.10.2026).