ЗАТВЕРДЖУЮ
Начальник кафедри інформаційних технологій та систем електронних комунікацій, к.т.н., доцент
Назарій БУРАК
«____» _____ 20___ р.

МЕТОДИЧНА РОЗРОБКА

для проведення лекційного заняття із здобувачами освіти першого (бакалаврського) рівня вищої освіти спеціальності F3 «Комп'ютерні науки»
з дисципліни «WEB-технології та WEB-програмування»

Лекція № 6. Тема 2.2: «Конфігурація та керування залежностями»

Мета: сформувати у здобувачів освіти розуміння принципів зовнішньої конфігурації застосунків Spring Boot, знання про джерела властивостей, профілі середовищ, системи збирання Maven і Gradle, журналювання, а також навички збирання, розгортання і контейнеризації вебзастосунку засобами Docker.

Час: 2 години.

Місце заняття: лекційна аудиторія (або онлайн у MS Teams).

Матеріальне забезпечення: мультимедійний проєктор, ПК з доступом до мережі Internet, IntelliJ IDEA, JDK 21, Maven, Docker Desktop (за наявності), браузер, презентація до лекції, електронний курс у «Віртуальному університеті», тематична вікторина до лекції.

Навчальні питання:

1. Зовнішня конфігурація. Формати application.properties і application.yml.
2. Джерела властивостей і порядок їх застосування. Змінні оточення.
3. Використання властивостей у коді: @Value і @ConfigurationProperties.
4. Профілі середовищ: розроблення, тестування, промислова експлуатація.
5. Керування залежностями засобами Maven.
6. Система збирання Gradle. Порівняння з Maven.
7. Журналювання в Spring Boot. Моніторинг засобами Actuator.
8. Збирання і розгортання застосунку.
9. Основи контейнеризації: Docker, Dockerfile, Docker Compose.

Після лекційного заняття здобувачі освіти повинні:

знати: принцип відокремлення конфігурації від коду; синтаксис properties і YAML; порядок пріоритету джерел властивостей; призначення профілів; структуру POM, області видимості залежностей і фази життєвого циклу Maven; основи Gradle; рівні журналювання; поняття образу і контейнера Docker, багатоетапного збирання;

вміти: описувати конфігурацію у YAML; передавати параметри через змінні оточення і командний рядок; створювати типізовані налаштування з перевіркою значень; створювати і активувати профілі; аналізувати дерево залежностей; налаштовувати журналювання; збирати виконуваний JAR і Docker-образ; запускати застосунок разом з БД через Docker Compose.

Література:

1. Віртуальне навчальне середовище ЛДУ БЖД «Віртуальний університет» [Електронний ресурс]. – Доступний з: https://virt.ldubgd.edu.ua/course/view.php?id=2111
2. Spring Boot Reference Documentation. Externalized Configuration, Profiles, Logging [Електронний ресурс]. – Доступний з: https://docs.spring.io/spring-boot/reference/features/
3. Apache Maven Project. Introduction to the Build Lifecycle; Introduction to the Dependency Mechanism [Електронний ресурс]. – Доступний з: https://maven.apache.org/guides/
4. Gradle User Manual [Електронний ресурс]. – Доступний з: https://docs.gradle.org/
5. Docker Docs. Dockerfile reference; Docker Compose [Електронний ресурс]. – Доступний з: https://docs.docker.com/
6. Wiggins A. The Twelve-Factor App [Електронний ресурс]. – Доступний з: https://12factor.net/
7. Walls C. Spring in Action. 6th ed. – Manning, 2022. – 520 p.

І. Підготовка до заняття

Перед початком заняття викладач перевіряє роботу проєктора, доступ до демо-проєкту (теги `lab-05` і `lab-06`), можливість запуску застосунку командою `mvn spring-boot:run` (профіль dev з БД H2 не потребує PostgreSQL), наявність Docker для демонстрації контейнеризації і готовність тематичної вікторини до лекції (відкривається з «Віртуального університету» на проекторі; проводиться наприкінці заняття, 10–15 хв).

ІІ. Зміст лекції

Вступ

Наприкінці лабораторної роботи № 5 застосунок «Реєстр укриттів» запускався командою `java -jar`, але адреса БД, ім'я користувача і пароль були записані у файлі `application.properties` усередині JAR. Що робити, якщо на ноутбуці розробника працює одна БД, на тестовому сервері – інша, а на промисловому сервері пароль узагалі не можна зберігати в репозиторії? Перезбирати застосунок для кожного середовища? Сучасний підхід інший: **один і той самий артефакт** (JAR або Docker-образ) запускається в усіх середовищах, а відрізняється лише **конфігурація**, яку передають ззовні. Цю ідею сформульовано в методології *The Twelve-Factor App*: «зберігайте конфігурацію в оточенні».

На цій лекції розглянемо, як Spring Boot читає налаштування з різних джерел, як організувати профілі для розроблення і промислової експлуатації, як Maven і Gradle керують бібліотеками, як правильно вести журнали, і як «упакувати» застосунок разом з усім потрібним у контейнер Docker (тег `lab-06` демо-проєкту).

1. Зовнішня конфігурація. Формати application.properties і application.yml

> **Конфігурація** – усе, що відрізняється між середовищами розгортання при незмінному коді: адреси і облікові дані БД, порти, адреси зовнішніх сервісів, рівні журналювання, параметри бізнес-логіки (наприклад, інтервал перевірок укриттів).
>
> **Зовнішня конфігурація** (externalized configuration) – підхід, за яким конфігурацію не записують у код, а читають під час запуску з файлів, змінних оточення чи аргументів командного рядка.

Spring Boot за замовчуванням читає файл `application.properties` або `application.yml` з каталогу `src/main/resources` (у JAR – з кореня classpath). Формат **properties** – це пари «ключ=значення», де ієрархія задається крапками. Формат **YAML** (YAML Ain't Markup Language) описує ту саму ієрархію відступами, тому спільні префікси не повторюються (табл. 1). Обидва формати рівноправні; у демо-проєкті з етапу `lab-06` використовується YAML.

application.properties	application.yml

spring.application.name=shelter-registry
spring.datasource.url=jdbc:postgresql://…
spring.datasource.username=shelter
server.port=8080
logging.level.root=INFO
	spring:
	  application:
	    name: shelter-registry
	  datasource:
	    url: jdbc:postgresql://…
	    username: shelter
	server:
	  port: 8080

Таблиця 1 – Однакові налаштування у двох форматах

Правила YAML, які найчастіше порушують початківці: відступи робляться **лише пробілами** (табуляція – помилка), кількість пробілів на рівні має бути однаковою (зазвичай два); після двокрапки обов'язковий пробіл; списки позначаються дефісом; рядки зі спеціальними символами (`@`, `:`, `#`) беруть у лапки; коментар починається з `#`. Якщо в проєкті є обидва файли, Spring Boot прочитає обидва, а за однакових ключів пріоритет матиме `.properties`, тому змішувати формати не рекомендується.

Spring Boot підтримує **гнучке зв'язування** (relaxed binding): властивість `registry.inspection-interval-days` можна задати у файлі як `inspection-interval-days` (рекомендована «kebab-case» форма), у змінній оточення як `REGISTRY_INSPECTIONINTERVALDAYS`, а в Java вона потрапить у поле `inspectionIntervalDays`. Значення можуть містити **заповнювачі** (placeholders) `${ім'я:значення_за_замовчуванням}`: запис `port: ${PORT:8080}` означає «взяти змінну оточення PORT, а якщо її немає – 8080». Цей прийом зручний для хмарних платформ, які самі призначають порт.

```
spring:
  application:
    name: shelter-registry
  profiles:
    active: dev                          # профіль за замовчуванням — розроблення
server:
  port: ${PORT:8080}                     # змінна оточення PORT або 8080
registry:                                # власні налаштування застосунку
  city: Львів
  inspection-interval-days: 90
  contact-email: cz@example.com
logging:
  level:
    root: INFO
```

Лістинг 1 – Фрагмент application.yml (тег lab-06)

2. Джерела властивостей і порядок їх застосування. Змінні оточення

Spring Boot збирає властивості з багатьох **джерел** (property sources) і об’єднує їх в одне середовище (`Environment`). Якщо одну властивість задано в кількох джерелах, перемагає джерело з вищим пріоритетом. Спрощений порядок – від найвищого до найнижчого – наведено в табл. 2.

№	Джерело	Приклад
1	Властивості тестів (`@SpringBootTest(properties=…)`, `@TestPropertySource`)	лише під час тестування
2	Аргументи командного рядка	`java -jar app.jar --server.port=9090`
3	Системні властивості Java	`java -Dserver.port=9090 -jar app.jar`
4	Змінні оточення ОС	`SERVER_PORT=9090`, `DB_PASSWORD=…`
5	Файл профілю поза JAR / у JAR	`application-prod.yml`
6	Основний файл поза JAR / у JAR	`application.yml`
7	Значення за замовчуванням у коді	`@Value("${x:10}")`, автоконфігурація

Таблиця 2 – Пріоритет джерел властивостей (спрощено)

Із таблиці випливає практичне правило: у файлі всередині JAR зберігають **розумні значення за замовчуванням**, а все, що залежить від середовища, перевизначають змінними оточення або аргументами. Файл `application.yml`,

розміщений поруч із JAR (або в підкаталозі `config/`), має пріоритет над вбудованим – так адміністратор може змінити налаштування без перезбирання.

Змінні оточення – основний спосіб передавання конфігурації в контейнерах і хмарі. Їхні імена пишуть великими літерами, крапки і дефіси замінюють підкресленнями: `spring.datasource.url` → `SPRING_DATASOURCE_URL`, `spring.profiles.active` → `SPRING_PROFILES_ACTIVE`. У демо-проєкті для профілю prod використовуються короткі власні імена через заповнювачі (лістинг 2).

```
spring:
  datasource:
    url: ${DB_URL:jdbc:postgresql://localhost:5432/shelter_registry}
    username: ${DB_USER:shelter}
    password: ${DB_PASSWORD:shelter}

# запуск у Linux/macOS
DB_URL=jdbc:postgresql://db.example:5432/shelters DB_PASSWORD=s3cr3t
java -jar app.jar
# запуск у Windows PowerShell
$env:DB_PASSWORD="s3cr3t"; java -jar app.jar
```

Лістинг 2 – Параметри БД зі змінних оточення (application-prod.yml)

Секрети (паролі, ключі API, токени) ніколи не записують у файли, що потрапляють у Git: публічний репозиторій сканують боти, і викрадений пароль може бути використано за лічені хвилини. Значення за замовчуванням у лістингу 2 допустимі лише для навчальної БД. У реальних системах секрети передають змінними оточення, файлами, змонтованими в контейнер, або спеціальними сховищами (HashiCorp Vault, AWS Secrets Manager, Kubernetes Secrets). Корисна звичка – додати локальний файл з паролями до `.gitignore`.

3. Використання властивостей у коді: @Value і @ConfigurationProperties

Найпростіший спосіб отримати значення властивості – анотація `@Value` на параметрі конструктора або полі біна. У виразі можна вказати значення за замовчуванням через двокрапку. Spring сам перетворить рядок на потрібний тип (число, логічне значення, `Duration`).

```
@Service
public class ReportService {
    private final int interval;
    public ReportService(@Value("${registry.inspection-interval-
days:90}") int interval) {
        this.interval = interval;
    }
}
```

Лістинг 3 – Отримання однієї властивості через @Value

Якщо налаштувань кілька і вони логічно пов'язані, краще використати **типізовані налаштування** – клас (у Java 16+ зручно `record`) з анотацією `@ConfigurationProperties(prefix = "registry")`. Spring Boot зв'яже всі властивості з префіксом `registry.` з компонентами запису і зареєструє об'єкт як бін, який можна впровадити в будь-який сервіс. Анотація `@Validated` разом з обмеженнями Bean Validation (`@NotBlank`, `@Min`, `@Email`) перевіряє значення **під час запуску**: якщо адміністратор помилково задасть інтервал 0, застосунок не запуститься і чітко повідомить, яку властивість задано неправильно, – це краще, ніж незрозуміла помилка через тиждень роботи.

```
@Validated
@ConfigurationProperties(prefix = "registry")
public record RegistryProperties(
        @NotBlank String city,
        @Min(1) int inspectionIntervalDays,
        @Email String contactEmail) {
}

@SpringBootApplication
@ConfigurationPropertiesScan          // знайти класи з
@ConfigurationProperties
public class ShelterRegistryApplication { … }
```

Лістинг 4 – Типізовані налаштування з перевіркою значень

Сервіс отримує налаштування через конструктор, як і будь-яку іншу залежність, і використовує їх у бізнес-логіці. У демо-проєкті з'явився метод пошуку укриттів з простроченою перевіркою і відповідний маршрут `GET /api/shelters/overdue` (лістинг 5). Змінивши лише конфігурацію (`--registry.inspection-interval-days=30`), можна отримати інший результат без зміни коду.

```
public List<ShelterView> findOverdueInspection() {
    LocalDate border =
LocalDate.now(clock).minusDays(properties.inspectionIntervalDays());
    log.debug("Пошук укриттів без перевірки після {}", border);
    return shelters.findAll().stream()
            .filter(s -> s.getLastInspectionDate() == null
                    || s.getLastInspectionDate().isBefore(border))
            .map(ShelterView::from)
            .toList();
}
```

Лістинг 5 – Використання налаштувань у сервісі

Порівняння двох підходів наведено в табл. 3. Також існує інтерфейс `Environment`, через який можна прочитати будь-яку властивість програмно (`env.getProperty("server.port")`), але в прикладному коді його використовують рідко.

Критерій	**@Value**	**@ConfigurationProperties**
Кількість значень	одне	група з одним префіксом
Типізація	просте перетворення типу	вкладені об’єкти, списки, `Duration`, `DataSize`
Перевірка значень	немає	`@Validated` + обмеження
Гнучке зв’язування	обмежене	повне
Підказки в IDE	ні	так (з процесором метаданих)

Таблиця 3 – Порівняння @Value і @ConfigurationProperties

4. Профілі середовищ: розроблення, тестування, промислова експлуатація

> **Профіль** (profile) – іменований набір налаштувань і бінів, що активується для певного середовища: `dev` – розроблення, `test` – автоматизоване тестування, `prod` – промислова експлуатація.

Налаштування профілю розміщують у файлі `application-<профіль>.yml`. Spring Boot спочатку читає спільний `application.yml`, а потім файли активних профілів, значення яких **перевизначають** спільні. Активувати профіль можна властивістю `spring.profiles.active` у файлі (значення за замовчуванням), змінною оточення `SPRING_PROFILES_ACTIVE=prod` або аргументом `--spring.profiles.active=prod`. Одночасно можуть бути активні кілька профілів через кому. У журналі запуску Spring Boot виводить рядок «The following 1 profile is active: "dev"».

Параметр	dev (розроблення)	prod (експлуатація)
База даних	H2 у пам’яті – нічого встановлювати не потрібно	PostgreSQL, параметри зі змінних оточення
Схема і дані	створюються під час кожного запуску (`spring.sql.init.mode: always`)	не змінюються застосунком (`never`); з ЛР 8 – Flyway
Журналювання	DEBUG для нашого коду, SQL-запити в консоль	INFO, журнал у файл
Службові засоби	вебконсоль H2 `/h2-console`	вимкнено

Таблиця 4 – Профілі демо-проєкту (тег lab-06)

```
spring:
  datasource:
    url: jdbc:h2:mem:shelters;MODE=PostgreSQL;DATABASE_TO_LOWER=TRUE
    username: sa
    password:
  sql:
    init:
      mode: always
      schema-locations: classpath:db/schema.sql
      data-locations: classpath:db/data.sql
  h2:
    console:
      enabled: true                          # http://localhost:8080/h2-
console
logging:
  level:
    ua.edu.ldubgd.shelters: DEBUG
    org.springframework.jdbc.core: DEBUG   # показувати SQL-запити
```

Лістинг 6 – Профіль розроблення application-dev.yml

H2 – вбудована реляційна СУБД, написана на Java. У режимі сумісності `MODE=PostgreSQL` вона виконує ту саму схему, що й PostgreSQL, тому новий учасник команди може запустити проєкт однією командою `mvn spring-boot:run` без встановлення сервера БД. Водночас H2 не є повною заміною: перед здачею роботи її завжди перевіряють з тією СУБД, яка буде в експлуатації (профіль prod).

Профілем можна керувати не лише налаштуваннями, а й **бінами**: анотація `@Profile("dev")` на класі або методі `@Bean` означає, що бін створюється лише в цьому профілі. Наприклад, у профілі dev можна підставити заглушку сервісу надсилання листів, яка лише пише повідомлення в журнал, а в prod – справжню реалізацію. Запис `@Profile("!prod")` означає «у всіх профілях, крім prod». Для тестів зручно використовувати `@ActiveProfiles("test")`.

```
public interface Notifier { void send(String to, String text); }

@Component @Profile("dev")
class LogNotifier implements Notifier {           // лише пише в журнал
    public void send(String to, String text) { log.info("Лист {}: {}",
to, text); }
}
@Component @Profile("prod")
class SmtpNotifier implements Notifier { … }      // справжнє
надсилання
```

Лістинг 7 – Різні реалізації біна для різних профілів

5. Керування залежностями засобами Maven

Apache Maven – система збирання і керування залежностями для Java, побудована на принципі «домовленості замість налаштувань»: стандартна структура каталогів (`src/main/java`, `src/test/java`), стандартний життєвий цикл і центральне сховище бібліотек **Maven Central**. Опис проєкту міститься у файлі `pom.xml` (Project Object Model). Кожен артефакт однозначно визначається **координатами** GAV: `groupId` (організація), `artifactId` (назва), `version` (версія, суфікс `-SNAPSHOT` позначає версію в розробці).

Завантажені бібліотеки зберігаються в **локальному сховищі** `~/.m2/repository` і повторно не завантажуються. Залежності бувають **прямими** (оголошені в `pom.xml`) і **транзитивними** (залежності залежностей): стартер `spring-boot-starter-webmvc` автоматично приносить Spring MVC, Jackson, вбудований Tomcat та ще кілька десятків бібліотек. Переглянути повне дерево можна командою `mvn dependency:tree`. Якщо дві бібліотеки потребують різних версій однієї залежності, Maven обирає «найближчу» до проєкту в дереві, що інколи спричиняє конфлікти; Spring Boot розв’язує цю проблему через **BOM** (Bill of Materials) – узгоджений перелік версій у батьківському POM `spring-boot-starter-parent`.

Scope	**Компіляція / тести / виконання**	**Приклад**
`compile`	так / так / так	стартери Spring Boot (за замовчуванням)
`runtime`	ні / так / так	драйвери PostgreSQL і H2
`test`	лише тести	`spring-boot-starter-test` (JUnit, Mockito)
`provided`	так / так / ні (надає середовище)	`jakarta.servlet-api` у WAR (ЛР 4)

Таблиця 5 – Області видимості залежностей Maven

```
<!-- ЛР6: БД H2 для профілю dev -->
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>                <!-- версію визначає BOM
Spring Boot -->
</dependency>
<!-- ЛР6: перевірка @ConfigurationProperties -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
<!-- ЛР6: службові ендпоінти /actuator/health, /actuator/info -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Лістинг 8 – Залежності, додані на етапі lab-06

Життєвий цикл Maven складається з фаз, які виконуються послідовно: виклик пізнішої фази запускає всі попередні (табл. 6). Конкретну роботу на кожній фазі виконують **плагіни** (компілятор, Surefire для тестів, `spring-boot-maven-plugin` для перепакування JAR). **Maven Wrapper** (`mvnw`, `mvnw.cmd`) завантажує потрібну версію Maven автоматично, тому проєкт збирається однаково на будь-якому комп'ютері і в CI.

Фаза	**Що відбувається**
`validate`	перевірка коректності POM
`compile`	компіляція `src/main/java` у `target/classes`; обробка ресурсів (підстановка `@project.version@`)
`test`	компіляція і запуск модульних тестів
`package`	створення JAR/WAR у `target/`; Spring Boot-плагін робить його виконуваним
`verify`	інтеграційні перевірки
`install`	копіювання артефакту в локальне сховище `~/.m2`
`deploy`	публікація артефакту у віддалене сховище

Таблиця 6 – Основні фази життєвого циклу Maven

Корисні команди: `mvn clean package` – видалити `target` і зібрати заново; `mvn package -DskipTests` – зібрати без запуску тестів (лише для швидкої перевірки, не для здачі); `mvn -q` – тихий режим; `mvn dependency:tree` – дерево

залежностей; `mvn versions:display-dependency-updates` – доступні оновлення бібліотек.

6. Система збирання Gradle. Порівняння з Maven

Gradle – альтернативна система збирання, у якій опис проєкту – не XML, а програма мовою Kotlin (`build.gradle.kts`) або Groovy (`build.gradle`). Gradle використовує ті самі сховища (Maven Central) і ті самі координати бібліотек, але інші назви «областей» залежностей – **конфігурації**: `implementation` (≈ compile), `runtimeOnly` (≈ runtime), `testImplementation` (≈ test), `compileOnly` (≈ provided). Завдяки інкрементальному збиранню і кешуванню результатів Gradle зазвичай швидший для великих багатомодульних проєктів; він є стандартом для Android. Spring Initializr однаково генерує проєкти обох типів.

```
plugins {
    java
    id("org.springframework.boot") version "4.1.2"
    id("io.spring.dependency-management") version "1.1.7"
}
group = "ua.edu.ldubgd"
version = "0.6.0"
java { toolchain { languageVersion = JavaLanguageVersion.of(21) } }
repositories { mavenCentral() }
dependencies {
    implementation("org.springframework.boot:spring-boot-starter-
webmvc")
    implementation("org.springframework.boot:spring-boot-starter-jdbc")
    runtimeOnly("org.postgresql:postgresql")
    runtimeOnly("com.h2database:h2")
    testImplementation("org.springframework.boot:spring-boot-starter-
test")
}
```

Лістинг 9 – Аналогічний опис проєкту в Gradle (build.gradle.kts)

Критерій	Maven	Gradle
Опис проєкту	декларативний XML (`pom.xml`)	код на Kotlin/Groovy
Гнучкість	жорсткий життєвий цикл, розширення плагінами	довільні задачі (tasks) і логіка
Швидкість	повне збирання	інкрементальне, кеш збирання, демон
Збирання / запуск	`./mvnw package`, `./mvnw spring-boot:run`	`./gradlew build`, `./gradlew bootRun`
Результат	`target/*.jar`	`build/libs/*.jar`
Поширеність	більшість корпоративних Java-проєктів	Android, великі багатомодульні проєкти

Таблиця 7 – Порівняння Maven і Gradle

Вибір системи збирання не впливає на код застосунку. У курсі використовується Maven як найпоширеніший у вакансіях Java-розробника; здобувачі, які обрали Gradle, можуть використовувати його в лабораторних роботах за умови, що проєкт збирається командою Wrapper.

7. Журналювання в Spring Boot. Моніторинг засобами Actuator

Журналювання (logging) – запис повідомлень про роботу застосунку для діагностики помилок, аудиту і моніторингу. Виводити повідомлення через `System.out.println` у серверному застосунку не можна: такі повідомлення не мають рівня важливості, часу, імені потоку і класу, їх не можна вимкнути чи перенаправити у файл. Spring Boot використовує фасад **SLF4J** (Simple Logging Facade for Java) з реалізацією **Logback**: код залежить лише від інтерфейсу `Logger`, а реалізацію можна замінити без зміни коду.

Рівень	Призначення і приклад
`ERROR`	помилка, що потребує втручання: не вдалося з’єднатися з БД
`WARN`	підозріла ситуація, робота триває: відхилено некоректну форму
`INFO`	важливі бізнес-події: створено або видалено укриття, застосунок запущено
`DEBUG`	деталі для розробника: параметри пошуку, SQL-запити
`TRACE`	найдетальніша трасування роботи бібліотек

Таблиця 8 – Рівні журналювання

Повідомлення виводиться, якщо його рівень не нижчий за рівень, налаштований для логера (пакета). За замовчуванням це INFO. Рівні задають у конфігурації окремо для кожного пакета

(`logging.level.ua.edu.ldubgd.shelters: DEBUG`), а властивість `logging.file.name` вмикає запис у файл з автоматичною ротацією за розміром.

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

log.info("Створено укриття id={} «{}»", saved.getId(),
saved.getName());   // ✓ параметри {}
log.warn("Відхилено створення укриття: {}", errors);
log.debug("Пошук укриттів без перевірки після {}", border);
log.error("Помилка імпорту файлу {}", fileName, exception);
// виняток — останнім

log.info("Створено " + shelter);                 // ✗ рядок збирається
навіть за вимкненого INFO
log.info("Вхід: {} / {}", login, password);    // ✗ НІКОЛИ не
журналюйте паролі й персональні дані
```

Лістинг 10 – Правила журналювання (ShelterService, тег lab-06)

Рядок журналу Spring Boot містить дату і час, рівень, ідентифікатор процесу, ім’я потоку (`[nio-8080-exec-1]` – потік Tomcat, що обробляє запит), скорочене ім’я класу і повідомлення. Під час розслідування інциденту саме за цими полями шукають події одного запиту. У великих системах журнали збирають централізовано (ELK-стек, Grafana Loki) і часто записують у форматі JSON.

Spring Boot Actuator додає службові ендпоінти для моніторингу і керування застосунком. Найважливіший – `/actuator/health`, що повертає `{"status":"UP"}`, якщо застосунок і його БД працюють; його опитують балансувальники навантаження, Docker і Kubernetes, щоб перезапустити «завислий» екземпляр. Ендпоінт `/actuator/info` показує версію застосунку, `/actuator/metrics` – метрики (пам’ять, кількість запитів). Оскільки ендпоінти розкривають внутрішню інформацію, назовні вмикають лише потрібні: у демо-проєкті – `health` та `info` (властивість `management.endpoints.web.exposure.include`). Запис `version: '@project.version@'` у `info.app` Maven замінює на версію з POM під час збирання.

8. Збирання і розгортання застосунку

Результат збирання Spring Boot-проєкту – **виконуваний JAR**, який містить скомпільований код, ресурси, усі бібліотеки (каталог `BOOT-INF/lib`) і завантажувач Spring Boot. Для запуску на сервері достатньо встановленого JRE

відповідної версії. Типовий сценарій розгортання на віртуальному сервері: зібрати артефакт у CI (GitHub Actions), скопіювати його на сервер, задати змінні оточення і запустити як службу ОС (у Linux – через systemd), щоб застосунок автоматично перезапускався після збою чи перезавантаження сервера.

```
mvn clean package                                                    #
target/shelter-registry-0.6.0.jar
java -jar target/shelter-registry-0.6.0.jar                          #
профіль dev (H2)
SPRING_PROFILES_ACTIVE=prod DB_PASSWORD=… java -jar target/shelter-
registry-0.6.0.jar
java -jar target/shelter-registry-0.6.0.jar --registry.inspection-
interval-days=30 --server.port=9090

curl http://localhost:8080/actuator/health        # {"status":"UP"}
curl http://localhost:8080/api/shelters/overdue
```

Лістинг 11 – Збирання і запуск з різною конфігурацією

Проте такий підхід вимагає, щоб на кожному сервері була встановлена правильна версія Java, а адміністратор налаштував службу. Розбіжності середовищ породжують класичну проблему «на моєму комп'ютері працює». Її розв'язує контейнеризація.

9. Основи контейнеризації: Docker, Dockerfile, Docker Compose

> **Контейнер** – ізольований процес ОС із власною файловою системою, мережею і обмеженнями ресурсів. На відміну від віртуальної машини, контейнер використовує ядро ОС хоста, тому запускається за секунди і займає мегабайти, а не гігабайти.
>
> **Образ** (image) – незмінний шаблон контейнера: файлова система з ОС, середовищем виконання і застосунком, що складається з **шарів**. Контейнер – запущений екземпляр образу.

Docker – найпоширеніша платформа контейнеризації. Образи зберігаються в реєстрах (Docker Hub, GitHub Container Registry), звідки їх завантажують командою `docker pull`. Готові офіційні образи існують для PostgreSQL, JDK (`eclipse-temurin`), Maven тощо. Порівняння способів розгортання наведено в табл. 9.

Критерій	JAR на сервері	Віртуальна машина	Контейнер
Що постачається	лише застосунок	ОС + усе ПЗ	застосунок + JRE + бібліотеки ОС
Розмір	десятки МБ	гігабайти	сотні МБ
Запуск	секунди	хвилини	секунди
Однаковість середовищ	залежить від сервера	висока	висока

Таблиця 9 – Способи розгортання застосунку

Образ описують у текстовому файлі **Dockerfile**. Кожна інструкція створює шар; якщо файли, від яких залежить шар, не змінилися, Docker бере його з кешу. У демо-проєкті використано **багатоетапне збирання** (multi-stage build): на першому етапі в образі з Maven і JDK компілюється JAR, на другому в мінімальний образ лише з JRE копіюється готовий JAR. Підсумковий образ не містить Maven, вихідного коду і компілятора – він менший і безпечніший (лістинг 12).

```
# Етап 1: збирання jar
FROM maven:3.9-eclipse-temurin-21 AS build
WORKDIR /src
COPY pom.xml .
RUN mvn -q dependency:go-offline        # залежності — окремим шаром
(кеш)
COPY src ./src
RUN mvn -q package -DskipTests

# Етап 2: мінімальний образ з JRE
FROM eclipse-temurin:21-jre
WORKDIR /app
RUN useradd --system --uid 1001 app     # не від імені root
COPY --from=build /src/target/shelter-registry-*.jar app.jar
USER app
EXPOSE 8080
ENV SPRING_PROFILES_ACTIVE=prod
ENTRYPOINT ["java", "-jar", "/app/app.jar"]
```

Лістинг 12 – Dockerfile демо-проєкту (тег lab-06)

Зверніть увагу на порядок інструкцій: `pom.xml` копіюється і залежності завантажуються *до* копіювання вихідного коду. Код змінюється часто, а залежності – рідко, тому під час повторного збирання шар із бібліотеками береться з кешу і збирання триває секунди. Файл `.dockerignore` виключає з контексту збирання каталоги `target`, `.git`, `.idea`. Застосунок запускається від імені непривілейованого користувача, щоб у разі зламу зловмисник не отримав права root у контейнері. Альтернатива власному Dockerfile – команда `mvn`

`spring-boot:build-image`, яка створює оптимізований образ засобами Cloud Native Buildpacks без жодного Dockerfile.

Команда	**Призначення**
`docker build -t shelter-registry:0.6 .`	зібрати образ з Dockerfile у поточному каталозі
`docker images`	переглянути локальні образи
`docker run -d -p 8080:8080 -e DB_URL=… shelter-registry:0.6`	запустити контейнер у фоні, відкрити порт, передати змінну
`docker ps`, `docker logs -f <id>`	запущені контейнери; журнал контейнера
`docker stop <id>`, `docker rm <id>`	зупинити і видалити контейнер

Таблиця 10 – Основні команди Docker

Застосунку потрібна ще й БД. **Docker Compose** описує кілька пов’язаних контейнерів (сервісів) у файлі `docker-compose.yml` і запускає їх однією командою `docker compose up --build` (лістинг 13). Сервіси бачать один одного в спільній мережі за іменем, тому адреса БД для застосунку – `db:5432`, а не `localhost`. **Том** (volume) `db-data` зберігає дані PostgreSQL поза контейнером, і вони не зникають після його перезапуску. Перевірка стану (`healthcheck`) разом з `depends_on … condition: service_healthy` гарантує, що застосунок стартує лише тоді, коли БД готова приймати з’єднання. Скрипти з каталогу `/docker-entrypoint-initdb.d` офіційний образ PostgreSQL виконує під час першого запуску – так створюються схема і тестові дані з ЛР 2.

```
services:
  db:
    image: postgres:17
    environment:
      POSTGRES_DB: shelter_registry
      POSTGRES_USER: shelter
      POSTGRES_PASSWORD: shelter
    volumes:
      - db-data:/var/lib/postgresql/data
      - ./src/main/resources/db/schema.sql:/docker-entrypoint-
initdb.d/01-schema.sql:ro
      - ./src/main/resources/db/data.sql:/docker-entrypoint-
initdb.d/02-data.sql:ro
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U shelter -d shelter_registry"]
      interval: 5s
  app:
    build: .
    depends_on:
      db: { condition: service_healthy }
    environment:
      SPRING_PROFILES_ACTIVE: prod
      DB_URL: jdbc:postgresql://db:5432/shelter_registry
    ports:
      - "8080:8080"
volumes:
  db-data:
```

Лістинг 13 – Застосунок і PostgreSQL у Docker Compose (скорочено)

Отже, конфігурація, профілі і контейнеризація працюють разом: той самий образ `shelter-registry:0.6` запускається на ноутбуці, тестовому і промисловому сервері, а відмінності задаються лише змінними оточення. Подальший крок – оркестрація контейнерів (Kubernetes) і автоматичне збирання та розгортання (CI/CD), які виходять за межі курсу.

Висновки

Конфігурацію відокремлюють від коду: значення за замовчуванням зберігають у `application.yml`, а залежні від середовища параметри і секрети передають змінними оточення або аргументами, які мають вищий пріоритет. Типізовані налаштування `@ConfigurationProperties` з `@Validated` виявляють помилки конфігурації під час запуску. Профілі (`dev`, `test`, `prod`) дозволяють мати різні налаштування і біни для різних середовищ з одного артефакту. Maven і Gradle керують прямими і транзитивними залежностями, а BOM Spring Boot узгоджує їх версії. Журналювання через SLF4J з рівнями і параметрами замінює `System.out`, а Actuator надає перевірку стану. Docker пакує застосунок разом із

середовищем виконання в образ, а Docker Compose запускає його разом із БД однією командою.

Контрольні запитання

1. Що таке зовнішня конфігурація і навіщо відокремлювати її від коду?
2. Чим відрізняються формати properties і YAML? Які типові помилки в YAML?
3. Назвіть джерела властивостей у порядку пріоритету. Як задати властивість змінною оточення?
4. Що означає запис `${PORT:8080}`?
5. Порівняйте `@Value` і `@ConfigurationProperties`. Навіщо `@Validated`?
6. Що таке профіль? Як його активувати? Для чого `@Profile`?
7. Які області видимості залежностей Maven ви знаєте? Що таке транзитивні залежності і BOM?
8. Назвіть основні фази життєвого циклу Maven. Що робить `mvn clean package`?
9. Чим Gradle відрізняється від Maven? Які аналоги має область `runtime`?
10. Які рівні журналювання існують? Чому не можна журналювати паролі і використовувати конкатенацію рядків?
11. Для чого потрібен ендпоінт `/actuator/health`?
12. Чим контейнер відрізняється від віртуальної машини? Що таке образ і шар?
13. Навіщо багатоетапне збирання і чому залежності копіюють у Dockerfile до вихідного коду?
14. Як сервіси в Docker Compose знаходять один одного? Для чого потрібні томи?

Завдання для самостійної роботи

Опрацювати матеріал лекції. Ознайомитися з методологією The Twelve-Factor App (розділи III «Config» і XI «Logs»). Встановити Docker Desktop і виконати посібник «Spring Boot with Docker» на spring.io/guides. Завершити та оформити лабораторну роботу № 6. Пройти тест № 6 у «Віртуальному університеті».

III. Підведення підсумків заняття

Провести тематичну вікторину до лекції (20 питань, режим «усі разом» або «дві команди», 10–15 хв), розібрати питання, на які найбільше здобувачів відповіли неправильно. Узагальнити основні положення лекції, відповісти на запитання здобувачів освіти. Наголосити на зв’язку матеріалу з лабораторною роботою № 6 і на правилах роботи із секретами. Видати завдання для самостійної роботи.

Методичну розробку розробив:

Доцент кафедри ІТтаСЕК, к.т.н. Роман ГОЛОВАТИЙ

Методична розробка обговорена на засіданні кафедри ІТтаСЕК
Протокол № ____ від «____» _____ 20___ р.