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 № 5. Тема 2.1: «Вступ до Spring Boot»

Мета: сформувати у здобувачів освіти цілісне уявлення про екосистему Spring, розуміння принципів інверсії керування і впровадження залежностей, знання про біни, їх області видимості й життєвий цикл, а також навички створення, структурування і запуску вебзастосунку на Spring Boot.

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, PostgreSQL, браузер, Postman або curl, презентація до лекції, електронний курс у «Віртуальному університеті», тематична вікторина до лекції (HTML, у «Віртуальному університеті»).

Навчальні питання:

1. Екосистема Spring. Spring Framework і Spring Boot.
2. Інверсія керування і впровадження залежностей.
3. IoC-контейнер і біни. Способи оголошення бінів.
4. Області видимості та життєвий цикл бінів.
5. Особливості Spring Boot: стартери, автоконфігурація, вбудований вебсервер.
6. Створення проєкту за допомогою Spring Initializr. Структура проєкту.
7. Доступ до бази даних: DataSource, JdbcClient, транзакції.
8. Перший вебзастосунок: REST-контролер і обробка запитів у Spring MVC.
9. Збирання, запуск і перевірка застосунку.

Після лекційного заняття здобувачі освіти повинні:

знати: склад екосистеми Spring і місце в ній Spring Boot; суть інверсії керування і впровадження залежностей; способи оголошення бінів; області видимості та етапи життєвого циклу біна; призначення стартерів і механізм автоконфігурації; структуру проєкту Spring Boot; шлях HTTP-запиту через `DispatcherServlet` до контролера;

вміти: створювати проєкт у Spring Initializr; оголошувати біни анотаціями і методами `@Bean`; впроваджувати залежності через конструктор; налаштовувати підключення до БД у `application.properties`; виконувати SQL-запити через `JdbcClient`; створювати REST-контролери; збирати і запускати виконуваний JAR.

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 [Електронний ресурс]. – Доступний з: https://docs.spring.io/spring-boot/
3. Spring Framework Reference Documentation. Core Technologies [Електронний ресурс]. – Доступний з: https://docs.spring.io/spring-framework/reference/core.html
4. Spring Guides [Електронний ресурс]. – Доступний з: https://spring.io/guides
5. Walls C. Spring in Action. 6th ed. – Manning, 2022. – 520 p.
6. Spilcă L. Spring Start Here. – Manning, 2021. – 416 p.
7. Fowler M. Inversion of Control Containers and the Dependency Injection pattern [Електронний ресурс]. – 2004. – Доступний з: https://martinfowler.com/articles/injection.html

І. Підготовка до заняття

Перед початком заняття викладач перевіряє роботу проєктора, доступ до демо-проєкту (теги `lab-04` і `lab-05`), наявність запущеної БД PostgreSQL з ЛР 2, можливість запуску застосунку командою `mvn spring-boot:run`, доступність сайту start.spring.io і готовність тематичної вікторини до лекції (відкривається з «Віртуального університету» на проекторі; проводиться наприкінці заняття, 10–15 хв).

ІІ. Зміст лекції

Вступ

У змістовому модулі 1 ми побудували вебзастосунок «Реєстр укриттів» на «чистій» Java: самі відкривали з'єднання з БД через `DriverManager`, закривали `ResultSet` і `PreparedStatement`, керували транзакціями викликами `commit()` і `rollback()`, створювали об'єкти сервісів і репозиторіїв у слухачі `AppContextListener`, писали сервлети з методами `doGet` і `doPost`, збирали WAR-архів і розгортали його на окремому Tomcat. Цей шлях був потрібен, щоб зрозуміти, *що відбувається всередині*. Але в реальному проєкті такого «службового» коду стає більше, ніж бізнес-логіки, і в ньому легко помилитися: забути закрити з'єднання, двічі створити сервіс, неправильно відкотити транзакцію.

Фреймворк **Spring** бере ці обов'язки на себе. Розробник описує, *які* об'єкти потрібні і як вони пов'язані, а Spring сам створює їх, з'єднує між собою, відкриває і закриває ресурси, починає і завершує транзакції, приймає HTTP-запити і перетворює відповіді на JSON. **Spring Boot** додає до цього готові налаштування «за замовчуванням», завдяки яким застосунок запускається однією командою. У модулі 2 ми поступово перенесемо демо-проєкт на Spring Boot, і на цій лекції закладемо основу: розберемо ключові ідеї фреймворку і створимо першу версію застосунку (тег `lab-05`).

1. Екосистема Spring. Spring Framework і Spring Boot

Spring Framework – фреймворк з відкритим кодом для створення застосунків на Java, перша версія якого з'явилася у 2003 р. як легша альтернатива складній платформі J2EE (EJB). Його центральна частина – **IoC-контейнер**, який керує об'єктами застосунку. Навколо ядра сформувалася ціла **екосистема** проєктів, кожен з яких розв'язує окрему задачу (табл. 1). Усі вони використовують одні й ті самі принципи, тому, опанувавши ядро, легко перейти до будь-якого з них.

Проєкт	Призначення	Тема курсу
Spring Framework	ядро: IoC-контейнер, AOP, транзакції, JDBC, Spring MVC (веб), WebFlux (реактивний веб)	2.1–2.3
Spring Boot	швидке створення готових до запуску застосунків: стартери, автоконфігурація, вбудований сервер	2.1–2.4
Spring Data	репозиторії для реляційних (JPA, JDBC) і NoSQL-баз без написання реалізацій	2.4
Spring Security	автентифікація, авторизація, захист від типових атак	2.4
Spring Cloud	мікросервіси: конфігурація, виявлення сервісів, шлюзи	–
Spring AI, Spring Batch, Spring Integration	інтеграція з моделями ШІ, пакетна обробка даних, обмін повідомленнями	–

Таблиця 1 – Основні проєкти екосистеми Spring

Важливо розрізняти **Spring Framework** і **Spring Boot**. Spring Framework надає механізми (контейнер, веб, транзакції), але їх потрібно налаштувати: оголосити `DataSource`, менеджер транзакцій, `DispatcherServlet`, конвертери JSON, і розгорнути застосунок на сервері. Spring Boot – це надбудова, яка робить усе це автоматично за принципом «*convention over configuration*» (домовленості замість налаштувань) і при цьому дозволяє змінити будь-яке налаштування. Spring Boot не замінює Spring Framework, а використовує його.

У курсі використовується **Spring Boot 4.1** на основі **Spring Framework 7**. Ці версії вимагають Java 17 або новішої (ми використовуємо Java 21), працюють з пакетами `jakarta.*` (Jakarta EE 11) замість застарілих `javax.*` і вбудованим Tomcat 11. Підтримку версії Spring Boot варто перевіряти на сайті spring.io: кожна мінорна версія безкоштовно підтримується близько одного року, тому для навчальних і реальних проєктів обирають актуальну.

2. Інверсія керування і впровадження залежностей

Розглянемо, як у ЛР 4 створювався сервіс укриттів. Слухач контексту власноруч створював усі об’єкти і передавав їх один одному (лістинг 1).

```
// ЛР 4: AppContextListener — «ручне» складання застосунку
ShelterRepository shelters = new JdbcShelterRepository();
DistrictRepository districts = new JdbcDistrictRepository();
InspectionRepository inspections = new JdbcInspectionRepository();
ShelterService service = new ShelterService(shelters, districts,
inspections);
context.setAttribute("shelterService", service);
```

Лістинг 1 – Ручне створення і зв’язування об’єктів

Поки об’єктів п’ять, це не проблема. Але в реальному застосунку їх сотні, і кожна зміна (новий параметр конструктора, нова реалізація репозиторію) вимагає правити код складання. Ще гірше, якщо клас сам створює свої залежності: `private final ShelterRepository repo = new JdbcShelterRepository();`. Такий клас **жорстко зв’язаний** з конкретною реалізацією: його неможливо протестувати з тестовим репозиторієм у пам’яті або перемкнути на іншу БД без зміни коду.

> **Залежність** (dependency) – об’єкт, який потрібен іншому об’єкту для виконання його роботи. Для `ShelterService` залежностями є репозиторії та годинник `Clock`.
>
> **Інверсія керування** (Inversion of Control, IoC) – принцип, за яким створенням об’єктів і керуванням їх життєвим циклом займається не сам код застосунку, а зовнішній механізм (фреймворк). Керування «перевертається»: не ми викликаємо фреймворк, а фреймворк викликає наш код («принцип Голлівуду»: *не телефонуйте нам, ми вам зателефонуємо*).
>
> **Впровадження залежностей** (Dependency Injection, DI) – спосіб реалізації IoC, за яким об’єкт не створює і не шукає свої залежності, а отримує їх ззовні: через конструктор, сетер або поле.

Ми вже знайомі з IoC: контейнер сервлетів сам створював екземпляри сервлетів і викликав їх методи `init()`, `service()`, `destroy()`. Spring поширює цей підхід на *всі* об’єкти застосунку. Існує три способи впровадження залежностей (табл. 2).

Спосіб	**Приклад**	**Оцінка**
Через конструктор	`public ShelterService(ShelterRepository r) { this.r = r; }`	**рекомендований**: залежності обов’язкові, поля можна зробити `final`, клас легко тестувати без Spring
Через сетер	`@Autowired void setClock(Clock c)`	для необов’язкових залежностей, які можна змінити після створення
Через поле	`@Autowired private ShelterRepository r;`	**не рекомендується**: приховує залежності, не дозволяє `final`, ускладнює тестування; допустимо в тестах

Таблиця 2 – Способи впровадження залежностей

Якщо клас має **єдиний конструктор**, Spring використовує його автоматично, і анотацію `@Autowired` писати не потрібно. Саме так оформлено всі класи демо-проєкту (лістинг 2). Зверніть увагу: код сервісу майже не змінився порівняно з ЛР 3, змінився лише спосіб його створення. Сервіс залежить від *інтерфейсів* репозиторіїв, а не від класів JDBC, тому в тестах йому можна передати інші реалізації.

```
@Service
public class ShelterService {
    private final ShelterRepository shelters;
    private final DistrictRepository districts;
    private final InspectionRepository inspections;
    private final Clock clock;

    public ShelterService(ShelterRepository shelters,
DistrictRepository districts,
                          InspectionRepository inspections, Clock
clock) {
        this.shelters = shelters;          // Spring сам знайде біни
        this.districts = districts;        // потрібних типів і
передасть
        this.inspections = inspections;    // їх у конструктор
        this.clock = clock;
    }
}
```

Лістинг 2 – Впровадження залежностей через конструктор (тег lab-05)

Впровадження залежностей тісно пов’язане з принципом **інверсії залежностей** (літера D у SOLID): модулі верхнього рівня (сервіс) не повинні залежати від модулів нижнього рівня (JDBC-репозиторій), обидва мають залежати від абстракцій (інтерфейсу `ShelterRepository`). Саме тому в ЛР 3 ми виділили інтерфейси репозиторіїв – тепер це дає змогу Spring підставити будь-яку реалізацію.

3. IoC-контейнер і біни. Способи оголошення бінів

IoC-контейнер Spring – це об’єкт, що реалізує інтерфейс `ApplicationContext`. Він читає **метадані конфігурації** (анотації або Java-класи конфігурації), створює об’єкти, впроваджує в них залежності, зберігає їх і знищує під час зупинки застосунку (рис. 1).

> **Бін** (bean) – об’єкт, яким керує IoC-контейнер Spring. Звичайний об’єкт, створений через `new`, біном не є: Spring про нього нічого не знає і не впроваджує в нього залежності.

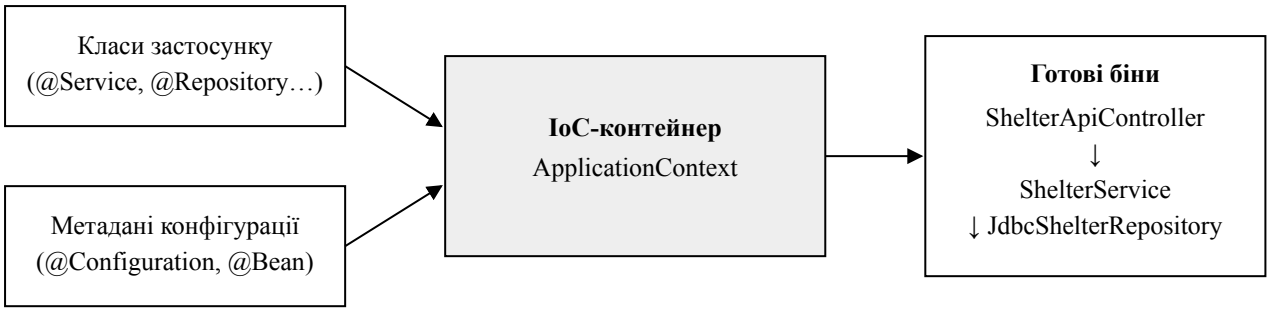

Рисунок 1 – Робота IoC-контейнера Spring

Існують два основні способи оголосити бін. Перший – позначити клас однією зі **стереотипних анотацій** (табл. 3). Під час запуску Spring виконує **сканування компонентів** (component scanning): переглядає пакет головного класу застосунку та всі його підпакети, знаходить позначені класи і створює з них біни. Тому головний клас розміщують у кореневому пакеті (`ua.edu.ldubgd.shelters`), а решту – у підпакетах.

Анотація	**Шар**	**Особливості**
`@Component`	будь-який	базова анотація; решта є її уточненнями
`@Service`	бізнес-логіка	документує роль класу
`@Repository`	доступ до даних	перетворює винятки БД на `DataAccessException`
`@Controller`	веб, HTML-сторінки	методи повертають ім’я представлення
`@RestController`	веб, REST API	= `@Controller` + `@ResponseBody`: результат методу – тіло відповіді (JSON)
`@Configuration`	налаштування	клас з методами `@Bean`

Таблиця 3 – Стереотипні анотації Spring

Другий спосіб – **метод з анотацією `@Bean`** у класі конфігурації. Його застосовують, коли клас не можна позначити анотацією (класи JDK або сторонніх бібліотек) або коли створення об’єкта потребує логіки. Значення, яке повертає метод, стає біном; назва методу – його ім’ям. У демо-проєкті так оголошено годинник `Clock` з часовим поясом Києва (лістинг 3). Сервіс отримує годинник як залежність і не викликає `LocalDate.now()` напряму, тому в тестах йому можна передати «зупинений» годинник з фіксованою датою.

```
@Configuration
public class AppConfig {
    @Bean
    public Clock clock() {
        return Clock.system(ZoneId.of("Europe/Kyiv"));
    }
}
```

Лістинг 3 – Оголошення біна методом @Bean

Під час впровадження Spring шукає бін за **типом** параметра. Якщо бінів потрібного типу немає, запуск завершується помилкою `NoSuchBeanDefinitionException` («Parameter 0 of constructor … required a bean of type … that could not be found»). Якщо бінів кілька (наприклад, дві реалізації `ShelterRepository` – JDBC і в пам'яті), виникає неоднозначність, яку розв'язують анотаціями: `@Primary` позначає реалізацію за замовчуванням, а `@Qualifier("назва")` на параметрі конструктора вказує конкретний бін. Можна також впровадити `List<ShelterRepository>` – тоді Spring передасть усі біни цього типу.

Ще одна поширена помилка – **циклічна залежність**: бін A потребує B, а B потребує A. При впровадженні через конструктор Spring не може створити жоден з них і повідомляє про цикл під час запуску. Це сигнал про погану архітектуру: спільну логіку слід винести в третій бін. Отже, помилки зв'язування виявляються одразу під час запуску, а не під час першого запиту користувача, – ще одна перевага контейнера.

4. Області видимості та життєвий цикл бінів

Область видимості (scope) визначає, скільки екземплярів біна створює контейнер і як довго вони існують (табл. 4). За замовчуванням усі біни – **одинаки** (singleton): на весь застосунок створюється один екземпляр, який спільно використовують усі потоки. Звідси випливає те саме правило, що й для сервлетів: **біни-одинаки не повинні зберігати в полях стан окремого запиту чи користувача**. Поля сервісів і репозиторіїв містять лише залежності (`final`), а дані запиту передаються параметрами методів.

Scope	Скільки екземплярів	Приклад застосування
`singleton`	один на контейнер (за замовчуванням)	сервіси, репозиторії, контролери
`prototype`	новий при кожному запиті біна з контейнера	об’єкти зі змінним станом, будівельники
`request`	один на HTTP-запит	дані поточного запиту
`session`	один на HTTP-сесію	кошик покупця, налаштування користувача
`application`	один на `ServletContext`	спільні дані вебзастосунку

Таблиця 4 – Області видимості бінів

Область видимості задають анотацією `@Scope("prototype")` або спеціалізованими `@RequestScope`, `@SessionScope`. На практиці понад 95 % бінів у вебзастосунку є одинаками.

Життєвий цикл біна-одинака складається з таких етапів: 1) контейнер визначає порядок створення бінів за їх залежностями; 2) викликає конструктор і передає йому залежності; 3) впроваджує залежності через сетери і поля (якщо є); 4) викликає методи *постпроцесорів* (`BeanPostProcessor`), які можуть замінити бін на **проксі** – обгортку, що додає поведінку (транзакції, безпеку, кешування); 5) викликає метод ініціалізації, позначений `@PostConstruct`; 6) бін готовий і використовується; 7) під час закриття контексту викликає метод, позначений `@PreDestroy`. Анотації `@PostConstruct` і `@PreDestroy` належать до пакета `jakarta.annotation`.

У демо-проєкті етапи життєвого циклу показує клас `StartupReport` (лістинг 4), який до того ж реалізує інтерфейс `CommandLineRunner`: його метод `run` викликається один раз, коли застосунок повністю запущено. Під час запуску в журналі послідовно з’являються повідомлення 1–3, а після натискання Ctrl+C – повідомлення 4.

```
@Component
public class StartupReport implements CommandLineRunner {
    private static final Logger log =
LoggerFactory.getLogger(StartupReport.class);
    private final ShelterService service;

    public StartupReport(ShelterService service) {
        this.service = service;
        log.info("1. Конструктор: Spring передав залежність
ShelterService");
    }
    @PostConstruct void init()   { log.info("2. @PostConstruct: усі
залежності впроваджено"); }
    @Override public void run(String... args) {
        log.info("3. Застосунок запущено. Укриттів у реєстрі: {}",
service.findAll().size());
    }
    @PreDestroy void shutdown() { log.info("4. @PreDestroy: контекст
закривається"); }
}
```

Лістинг 4 – Демонстрація життєвого циклу біна

Чому ініціалізаційну логіку переносять з конструктора в `@PostConstruct`? У конструкторі доступні лише залежності, передані параметрами; у `@PostConstruct` – уже всі (включно з упровадженими через поля), і об'єкт повністю сформований. Типові дії в `@PostConstruct`: перевірка налаштувань, завантаження довідників у кеш; у `@PreDestroy` – закриття файлів і з'єднань, завершення фонових потоків. Для бінів `prototype` контейнер не викликає `@PreDestroy`: після передачі об'єкта клієнту він ним більше не керує.

Механізм **проксі** варто запам'ятати вже зараз. Коли контролер викликає метод сервісу, позначеного `@Transactional`, насправді він звертається до проксі-об'єкта, який спочатку відкриває транзакцію, потім викликає справжній метод, а після нього фіксує або відкочує зміни. Наслідок: виклик `@Transactional`-методу *зсередини того самого класу* (через `this`) проходить повз проксі, і транзакція не створюється. Цю особливість часто запитують на співбесідах.

5. Особливості Spring Boot: стартери, автоконфігурація, вбудований вебсервер

Spring Boot – проєкт, який дозволяє створювати самодостатні застосунки Spring, готові до запуску в промисловому середовищі, з мінімумом налаштувань. Його можливості ґрунтуються на чотирьох механізмах.

1. Стартери (starters) – набори узгоджених залежностей для певної задачі. Замість того щоб окремо підключати Spring MVC, Tomcat, бібліотеку Jackson для JSON, валідатор і підбирати їх сумісні версії, достатньо однієї залежності `spring-boot-starter-webmvc`. Версії всіх бібліотек визначає батьківський POM `spring-boot-starter-parent`, тому в залежностях версію не вказують (табл. 5).

Стартер	Що підключає
`spring-boot-starter-webmvc`	Spring MVC, вбудований Tomcat, Jackson (JSON)
`spring-boot-starter-jdbc`	пул з'єднань HikariCP, `JdbcClient`, керування транзакціями
`spring-boot-starter-data-jpa`	Hibernate, Spring Data JPA (тема 2.4)
`spring-boot-starter-thymeleaf`	шаблонізатор HTML-сторінок (тема 2.3)
`spring-boot-starter-validation`	Bean Validation: `@NotBlank`, `@Positive` (тема 2.3)
`spring-boot-starter-security`	Spring Security (тема 2.4)
`spring-boot-starter-actuator`	моніторинг: `/actuator/health`, метрики
`spring-boot-starter-test`	JUnit 5, AssertJ, Mockito, засоби тестування Spring

Таблиця 5 – Поширені стартери Spring Boot

2. Автоконфігурація (auto-configuration) – автоматичне створення бінів залежно від того, які бібліотеки є в classpath і які властивості задано. Наприклад, якщо в classpath є HikariCP і драйвер PostgreSQL, а в `application.properties` задано `spring.datasource.url`, Spring Boot сам створить біни `DataSource`, `JdbcClient` і менеджер транзакцій. Якщо є Spring MVC – створить `DispatcherServlet` і запустить вбудований Tomcat. Класи автоконфігурації містять **умовні анотації**: `@ConditionalOnClass` (клас є в classpath), `@ConditionalOnProperty` (задано властивість), `@ConditionalOnMissingBean` (розробник не оголосив власний бін такого типу). Остання умова означає, що **власний бін завжди має пріоритет**: автоконфігурація лише заповнює прогалини. Звіт про те, які автоконфігурації застосовано і чому, можна побачити, запустивши застосунок з параметром `--debug` (розділ «CONDITIONS EVALUATION REPORT»).

3. Вбудований вебсервер. Tomcat (або Jetty) входить до застосунку як звичайна бібліотека. Застосунок збирається у **виконуваний JAR** («fat jar»), який містить код, усі залежності і сервер, і запускається командою `java -jar`. Окремо встановлювати Tomcat і розгортати WAR (як у ЛР 4) не потрібно. Такий формат зручний для контейнерів Docker і хмарних платформ (тема 2.2).

4. Зовнішня конфігурація. Параметри (порт, адреса БД, рівень журналювання) виносяться у файл `application.properties` або `application.yml` і можуть перевизначатися змінними оточення та параметрами командного рядка без перезбирання застосунку. Докладно конфігурацію і профілі розглянемо в темі 2.2.

Усе це «вмикає» одна анотація на головному класі – `@SpringBootApplication` (лістинг 5). Вона поєднує три анотації: `@Configuration` (клас може містити методи `@Bean`), `@EnableAutoConfiguration` (увімкнути автоконфігурацію) і `@ComponentScan` (сканувати поточний пакет і підпакети). Метод `SpringApplication.run` створює `ApplicationContext`, запускає вбудований сервер і повертає керування, коли застосунок готовий приймати запити.

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

Лістинг 5 – Головний клас застосунку Spring Boot

6. Створення проєкту за допомогою Spring Initializr. Структура проєкту

Spring Initializr (start.spring.io) – вебсервіс, що генерує заготовку проєкту Spring Boot. Він вбудований також в IntelliJ IDEA (File → New → Project → Spring Boot) і VS Code. Під час створення проєкту обирають: систему збирання (**Maven** або **Gradle**), мову (Java), версію Spring Boot (стабільну, без позначок SNAPSHOT чи M), метадані проєкту (*Group* – зворотне доменне ім'я організації, *Artifact* – назва проєкту, з них формується кореневий пакет), спосіб пакування (Jar), версію Java (21) і залежності (стартери). Для демо-проєкту обрано: Group `ua.edu.ldubgd`, Artifact `shelter-registry`, залежності Spring Web, JDBC API, PostgreSQL Driver. Сервіс формує ZIP-архів, який розпаковують і відкривають в IDE як проєкт Maven.

```
shelter-registry/
├── pom.xml                              залежності і плагіни Maven
├── mvnw, mvnw.cmd, .mvn/                Maven Wrapper — збирання без
встановленого Maven
└── src/
    ├── main/
    │   ├── java/ua/edu/ldubgd/shelters/
    │   │   ├── ShelterRegistryApplication.java   головний клас
    │   │   ├── api/          REST-контролери (@RestController)
    │   │   ├── config/       конфігурація (@Configuration)
    │   │   ├── service/      бізнес-логіка (@Service)
    │   │   ├── repository/   інтерфейси і реалізації JDBC
(@Repository)
    │   │   ├── model/, dto/, exception/
    │   └── resources/
    │       ├── application.properties    налаштування
    │       ├── static/                   статичні файли (CSS, JS,
зображення)
    │       └── templates/                шаблони сторінок (тема 2.3)
    └── test/java/…/ShelterRegistryApplicationTests.java
```

Лістинг 6 – Структура проєкту Spring Boot (тег lab-05)

Порівняно з ЛР 4 зникли каталоги `webapp/WEB-INF` і файл `web.xml`: налаштування задаються анотаціями і властивостями, статичні файли розміщуються в `resources/static`. Пакети `model`, `dto`, `repository`, `service` перенесено з модуля 1 майже без змін – це результат правильного розділення на шари в ЛР 3. Фрагмент файлу `pom.xml` наведено в лістингу 7.

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>4.1.2</version>          <!-- визначає версії всіх
бібліотек -->
</parent>
<properties><java.version>21</java.version></properties>
<dependencies>
    <dependency><groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-webmvc</artifactId>
</dependency>
    <dependency><groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jdbc</artifactId></dependency>
    <dependency><groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId><scope>runtime</scope>
</dependency>
    <dependency><groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope></dependency>
</dependencies>
<build><plugins><plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>   <!-- spring-
boot:run, fat jar -->
</plugin></plugins></build>
```

Лістинг 7 – Основні елементи pom.xml

Область `runtime` означає, що драйвер PostgreSQL потрібен лише під час виконання: код застосунку не звертається до його класів напряму, а працює через стандартні інтерфейси JDBC. Плагін `spring-boot-maven-plugin` додає команду `mvn spring-boot:run` і перепаковує звичайний JAR у виконуваний. Gradle – альтернативна система збирання з описом проєкту мовою Groovy або Kotlin; порівняння Maven і Gradle розглянемо в темі 2.2.

7. Доступ до бази даних: DataSource, JdbcClient, транзакції

У ЛР 2 кожен метод репозиторію відкривав нове з’єднання через `DriverManager.getConnection`. Встановлення з’єднання з БД – дорога операція (мережа, автентифікація), тому в серверних застосунках використовують **пул з’єднань**: заздалегідь відкритий набір з’єднань, які видаються на час запиту і повертаються в пул замість закриття. Spring Boot за замовчуванням використовує пул **HikariCP** (до 10 з’єднань) і створює бін `DataSource` за властивостями з лістингу 8.

```
spring.application.name=shelter-registry
spring.datasource.url=jdbc:postgresql://localhost:5432/shelter_registry
spring.datasource.username=shelter
spring.datasource.password=shelter
server.port=8080
```

Лістинг 8 – Файл application.properties

Для виконання SQL-запитів Spring пропонує клас **`JdbcClient`** (з версії Spring Framework 6.1) – сучасну «плавну» обгортку над JDBC. Він сам бере з'єднання з пулу, створює `PreparedStatement`, підставляє параметри, перетворює рядки результату на об'єкти і закриває всі ресурси. Перевірювані винятки `SQLException` перетворюються на неперевірювані винятки ієрархії `DataAccessException`, тому блоки `try-catch-finally` зникають. Іменовані параметри (`:id`) зручніші за позиційні знаки питання і так само захищають від SQL-ін'єкцій. Перетворення рядка на об'єкт описує `RowMapper` – функція, яку можна задати лямбда-виразом (лістинг 9).

```
@Repository
public class JdbcShelterRepository implements ShelterRepository {
    private static final RowMapper<Shelter> MAPPER = (rs, rowNum) -> {
/* rs → Shelter */ };
    private final JdbcClient jdbc;

    public JdbcShelterRepository(JdbcClient jdbc) { this.jdbc = jdbc; }

    public Optional<Shelter> findById(int id) {
        return jdbc.sql(SELECT + " WHERE s.id = :id")
                   .param("id", id)
                   .query(MAPPER)
                   .optional();
    }
    public boolean existsByAddress(String address) {
        return jdbc.sql("SELECT COUNT(*) FROM shelters WHERE address =
:address")
                   .param("address", address)
                   .query(Integer.class).single() > 0;
    }
}
```

Лістинг 9 – Репозиторій на JdbcClient

Порівняйте обсяг коду: метод `findById` у ЛР 2 займав близько 20 рядків з урахуванням відкриття і закриття ресурсів, тепер – п'ять. Метод `query` може повернути список (`list()`), один об'єкт (`single()`), `Optional` (`optional()`) або потік; для змін використовують `update()`, а згенерований ключ отримують через `KeyHolder`.

Транзакції. У ЛР 2 реєстрація перевірки укриття (вставлення запису в `inspections` і зміна статусу в `shelters`) виконувалася в транзакції з ручними викликами `setAutoCommit(false)`, `commit()` і `rollback()`. У Spring достатньо позначити метод сервісу анотацією **`@Transactional`**: проксі відкриє транзакцію перед викликом методу, зафіксує її після успішного завершення і відкотить, якщо метод завершився неперевірюваним винятком (лістинг 10). Усі запити `JdbcClient` усередині методу автоматично використовують одне з'єднання цієї транзакції.

```
@Transactional
public ShelterView registerInspection(int shelterId, int inspectorId,
                                      InspectionResult result, String
notes) {
    Shelter shelter = getShelter(shelterId);
    LocalDate today = LocalDate.now(clock);
    inspections.save(new Inspection(null, shelterId, inspectorId,
today, result, notes));
    shelter.applyInspection(result, today);
    shelters.save(shelter);              // помилка тут → відкат обох
змін
    return ShelterView.from(shelter);
}
```

Лістинг 10 – Декларативна транзакція

Транзакції оголошують на рівні **сервісу**, а не репозиторію, бо саме сервіс знає, які операції утворюють одну бізнес-дію. Для методів, що лише читають дані, використовують `@Transactional(readOnly = true)`.

8. Перший вебзастосунок: REST-контролер і обробка запитів у Spring MVC

Spring MVC – вебфреймворк Spring Framework, побудований на Servlet API. Його центральний елемент – **`DispatcherServlet`**, звичайний сервлет, зіставлений з адресою `/`, тобто з усіма запитами. Він реалізує шаблон *Front Controller*: приймає кожен запит і передає його потрібному методу контролера. Spring Boot реєструє `DispatcherServlet` у вбудованому Tomcat автоматично. Шлях запиту показано на рис. 2.

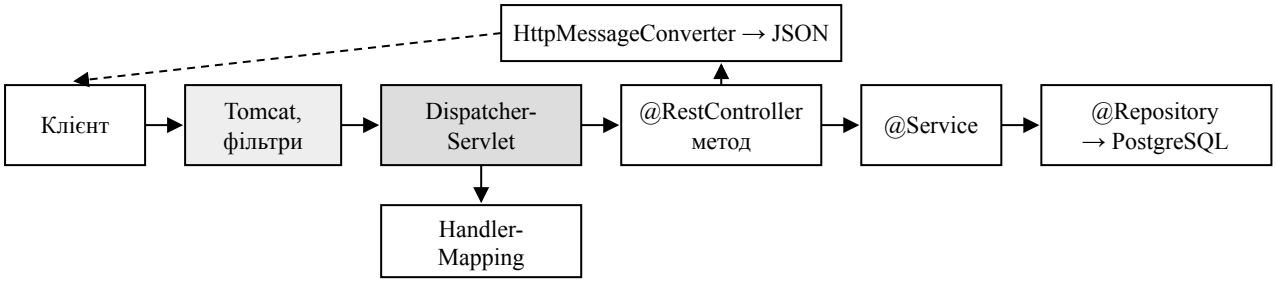

Рисунок 2 – Шлях HTTP-запиту в застосунку Spring MVC

Послідовність обробки: 1) Tomcat приймає з'єднання і пропускає запит через ланцюжок фільтрів; 2) `DispatcherServlet` запитує в `HandlerMapping`, який метод якого контролера відповідає HTTP-методу і шляху запиту; 3) з параметрів запиту, змінних шляху і тіла формуються аргументи методу; 4) метод контролера викликає сервіс, той – репозиторій; 5) значення, яке повернув метод, `HttpMessageConverter` (бібліотека Jackson) перетворює на JSON і записує в тіло відповіді з заголовком `Content-Type: application/json`.

Найпростіший контролер для перевірки роботи застосунку наведено в лістингу 11. Анотація `@GetMapping("/hello")` зіставляє метод із запитом `GET /hello`, а `@RequestParam` зв'язує параметр методу з параметром рядка запиту.

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello(@RequestParam(defaultValue = "світ") String
name) {
        return "Привіт, " + name + "! Реєстр укриттів працює.";
    }
}
```

Лістинг 11 – Найпростіший REST-контролер

Основні анотації Spring MVC для зіставлення запитів наведено в табл. 6. Повний набір операцій CRUD (POST, PUT, DELETE), валідацію і документування API розглянемо в темі 2.3, а на цьому етапі обмежимося читанням даних.

Анотація	Призначення
`@RequestMapping("/api")`	на класі: спільний префікс шляху для всіх методів
`@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`	зіставлення методу з HTTP-методом і шляхом
`@PathVariable`	значення з шаблону шляху: `/shelters/{id}` → `int id`
`@RequestParam`	параметр рядка запиту: `?districtId=6`; `required = false` – необов'язковий
`@RequestBody`	тіло запиту (JSON) → об'єкт Java (тема 2.3)
`@ExceptionHandler`, `@ResponseStatus`	перетворення винятку на відповідь з потрібним кодом стану

Таблиця 6 – Анотації зіставлення запитів

Контролер реєстру укриттів (лістинг 12) отримує сервіс через конструктор і повертає дані у форматі JSON. Порівняйте його з `ShelterServlet` з ЛР 4: немає методів `doGet`, об'єктів `request` і `response`, ручного розбору параметрів і перетворення типів (рядок `"6"` автоматично стає числом `6`; якщо перетворення неможливе, Spring відповідає кодом 400). Записи Java (`record`), як-от DTO `ShelterView`, перетворюються на JSON без додаткових налаштувань: імена компонентів запису стають ключами об'єкта JSON.

```
@RestController
@RequestMapping("/api")
public class ShelterApiController {
    private final ShelterService service;
    public ShelterApiController(ShelterService service) { this.service
= service; }

    @GetMapping("/shelters")                        // GET
/api/shelters?districtId=6
    public List<ShelterView> shelters(@RequestParam(required = false)
Integer districtId) {
        return districtId == null ? service.findAll() :
service.findByDistrict(districtId);
    }
    @GetMapping("/shelters/{id}")                   // GET
/api/shelters/3
    public ShelterView shelter(@PathVariable int id) { return
service.findById(id); }

    @ExceptionHandler(NotFoundException.class)      // неіснуючий id →
404
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public Map<String, String> notFound(NotFoundException e) {
        return Map.of("error", e.getMessage());
    }
}
```

Лістинг 12 – REST-контролер реєстру укриттів

Відповідь на запит `GET /api/shelters/3` матиме вигляд:

```
HTTP/1.1 200
Content-Type: application/json

{"id":3,"name":"Школа № 12","address":"вул. Наукова, 7",
 "district":"Франківський","type":"Захисна споруда ЦЗ","capacity":300,
 "accessibility":"Ні","status":"Обмежено
готове","statusCode":"LIMITED"}
```

Лістинг 13 – Відповідь REST API у форматі JSON

Отже, REST-контролер повертає *дані*, а не HTML-сторінку. Такий API може використовувати будь-який клієнт: односторінковий вебзастосунок на JavaScript, мобільний застосунок, інший сервіс. Принципи проєктування REST API (ресурси, HTTP-методи, коди стану) докладно розглядатимуться в темі 2.3.

9. Збирання, запуск і перевірка застосунку

Застосунок Spring Boot можна запустити трьома способами: з IDE (кнопка Run біля методу `main`), командою Maven або як виконуваний JAR (лістинг 14). Перед запуском має працювати PostgreSQL з базою з ЛР 2.

```
git checkout lab-05
mvn spring-boot:run                          # запуск у режимі розроблення
mvn package                                  # target/shelter-registry-
0.5.0.jar
java -jar target/shelter-registry-0.5.0.jar --server.port=9090

curl "http://localhost:8080/hello?name=Олена"
curl http://localhost:8080/api/shelters?districtId=6
curl -i http://localhost:8080/api/shelters/999      # HTTP/1.1 404
```

Лістинг 14 – Збирання, запуск і перевірка

Під час запуску в консолі виводиться журнал: банер Spring, версії Java і Spring Boot, ініціалізація пулу HikariCP, повідомлення життєвого циклу біна `StartupReport` і рядок «Tomcat started on port 8080». Найчастіші помилки запуску та їх причини: «Port 8080 was already in use» – порт зайнято іншим процесом (наприклад, застосунком з ЛР 4); «Connection refused» або «password authentication failed» – не запущено PostgreSQL або неправильні облікові дані в `application.properties`; «required a bean of type … that could not be found» – клас не позначено стереотипною анотацією або він розміщений поза пакетом головного класу.

Для зручності розроблення підключають модуль **Spring Boot DevTools**, який автоматично перезапускає застосунок після компіляції змінених класів. Перевірити, що контекст піднімається і всі біни створюються, дозволяє інтеграційний тест з анотацією `@SpringBootTest` (лістинг 15). У тестах залежності впроваджують через поле з `@Autowired`, бо тестовий клас створює JUnit, а не Spring. Детально тестування розглянемо в темі 2.4.

```
@SpringBootTest
class ShelterRegistryApplicationTests {
    @Autowired ShelterService service;

    @Test
    void contextLoads() {
        assertThat(service).isNotNull();
    }
}
```

Лістинг 15 – Перевірка контексту застосунку (mvn test)

Підсумкове порівняння реалізації демо-проєкту в модулях 1 і 2 наведено в табл. 7. У темі 2.2 ми продовжимо: винесемо налаштування в профілі середовищ, налаштуємо журналювання і зберемо Docker-образ застосунку.

ЛР 3–4 (Java, Servlet API)	ЛР 5 (Spring Boot)
`new ShelterService(new JdbcShelterRepository(), …)` у слухачі	`@Service`, `@Repository` + впровадження через конструктор
`DriverManager.getConnection(...)`	`DataSource` (пул HikariCP) з `application.properties`
`PreparedStatement`, `ResultSet`, `try-with-resources`	`JdbcClient` + `RowMapper`
`setAutoCommit(false)` / `commit()` / `rollback()`	`@Transactional`
`HttpServlet.doGet` + JSP	`@RestController` + JSON
WAR + окремий Tomcat, `web.xml`	виконуваний JAR з вбудованим Tomcat

Таблиця 7 – Що змінилося після переходу на Spring Boot

Висновки

Spring Framework реалізує інверсію керування: об’єкти застосунку (біни) створює і зв’язує IoC-контейнер, а класи отримують залежності через конструктор. Біни оголошують стереотипними анотаціями або методами `@Bean`; за замовчуванням вони є одинаками, тому не повинні зберігати стан окремого запиту. Життєвий цикл біна включає створення, впровадження залежностей, обгортання проксі, `@PostConstruct` і `@PreDestroy`. Spring Boot спрощує використання Spring завдяки стартерам, автоконфігурації, вбудованому вебсерверу і зовнішній конфігурації; проєкт зручно створювати в Spring Initializr. `JdbcClient` і `@Transactional` замінюють ручну роботу з JDBC, а `@RestController`, що працює через `DispatcherServlet`, – ручне написання сервлетів.

Контрольні запитання

1. Які проєкти входять до екосистеми Spring? Чим Spring Boot відрізняється від Spring Framework?
2. Що таке інверсія керування і впровадження залежностей? Наведіть приклад з демо-проєкту.
3. Порівняйте способи впровадження залежностей. Чому рекомендують конструктор?
4. Що таке бін? Якими способами можна оголосити бін?
5. Як працює сканування компонентів? Чому головний клас розміщують у кореневому пакеті?

6. Як Spring розв’язує неоднозначність, якщо є кілька бінів одного типу?
7. Які області видимості бінів ви знаєте? Чому біни-одинаки не повинні мати змінного стану?
8. Опишіть етапи життєвого циклу біна. Для чого потрібні `@PostConstruct` і `@PreDestroy`?
9. Що таке стартер і автоконфігурація? Яку роль відіграє `@ConditionalOnMissingBean`?
10. Які анотації поєднує `@SpringBootApplication`?
11. Навіщо потрібен пул з’єднань? Які переваги має `JdbcClient` порівняно з «чистим» JDBC?
12. Як працює `@Transactional`? Чому виклик через `this` не створює транзакцію?
13. Опишіть шлях HTTP-запиту від клієнта до методу `@RestController`.

Завдання для самостійної роботи

Опрацювати матеріал лекції. Створити в Spring Initializr проєкт для предметної області своєї курсової роботи і спроєктувати архітектуру вебсервісу (шари, пакети, перелік бінів і їх залежностей). Ознайомитися з посібником «Building a RESTful Web Service» на spring.io/guides. Пройти тест № 5 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 № 5. 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.