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 № 8. Тема 2.4: «Робота з базами даних у Spring Boot. Безпека та тестування вебзастосунків»

Мета: сформувати у здобувачів освіти знання про об'єктно-реляційне відображення, специфікацію JPA і Hibernate, репозиторії Spring Data JPA, міграції бази даних, оптимізацію роботи з великими обсягами даних, основи Spring Security, а також навички модульного та інтеграційного тестування вебзастосунків 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, браузер, Postman або curl, презентація до лекції, електронний курс у «Віртуальному університеті», тематична вікторина до лекції.

Навчальні питання:

1. Об'єктно-реляційне відображення. JPA і Hibernate.
2. Сутності та зв'язки між ними.
3. Репозиторії Spring Data JPA. Похідні запити та JPQL.
4. Оптимізація роботи з даними: ліниве завантаження, проблема N+1, пагінація.
5. Міграції бази даних. Flyway.
6. Основи Spring Security: автентифікація, авторизація, зберігання паролів.
7. Захист вебсторінок і REST API. CSRF.
8. Модульне тестування: JUnit 5, AssertJ, Mockito.
9. Інтеграційне тестування: Spring Boot Test, MockMvc, @DataJpaTest.

Після лекційного заняття здобувачі освіти повинні:

знати: поняття ORM і невідповідності об’єктної та реляційної моделей; основні анотації JPA; види зв’язків і сторону-власника; можливості Spring Data JPA; ліниве і жадібне завантаження, проблему N+1; принципи версійних міграцій; архітектуру Spring Security (ланцюжок фільтрів); різницю між автентифікацією і авторизацією; призначення BCrypt і CSRF-токенів; піраміду тестування і види тестів у Spring Boot;

вміти: описувати сутності і зв’язки; створювати репозиторії з похідними запитами, JPQL і пагінацією; усувати проблему N+1; створювати міграції Flyway; налаштовувати вхід через форму і HTTP Basic, правила доступу за ролями; зберігати паролі хешами; писати модульні тести з Mockito, тести репозиторіїв і вебшару.

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 Data JPA Reference Documentation [Електронний ресурс]. – Доступний з: https://docs.spring.io/spring-data/jpa/reference/
3. Jakarta Persistence 3.2 Specification; Hibernate ORM User Guide [Електронний ресурс]. – Доступний з: https://hibernate.org/orm/documentation/
4. Flyway Documentation [Електронний ресурс]. – Доступний з: https://documentation.red-gate.com/fd
5. Spring Security Reference [Електронний ресурс]. – Доступний з: https://docs.spring.io/spring-security/reference/
6. Spring Boot Reference. Testing [Електронний ресурс]. – Доступний з: https://docs.spring.io/spring-boot/reference/testing/
7. Mihalcea V. High-Performance Java Persistence. – 2016. – 470 p.
8. OWASP Top 10; Password Storage Cheat Sheet [Електронний ресурс]. – Доступний з: https://owasp.org/

І. Підготовка до заняття

Перед початком заняття викладач перевіряє роботу проєктора, доступ до демо-проєкту (теги `lab-07` і `lab-08`), можливість запуску застосунку командою `mvn spring-boot:run` (профіль dev, H2) і тестів командою `mvn test`, облікові записи `admin`/`admin` та `inspector`/`inspector`, готовність тематичної вікторини до

лекції (відкривається з «Віртуального університету» на проекторі; проводиться наприкінці заняття, 10–15 хв).

ІІ. Зміст лекції

Вступ

Демо-проєкт «Реєстр укриттів» пройшов шлях від статичної сторінки до вебзастосунку з REST API. Але три важливі речі ще не зроблено. По-перше, кожен SQL-запит ми писали вручну і власноруч перетворювали рядки таблиць на об’єкти – у великому застосунку це сотні однотипних запитів. По-друге, будь-хто може змінити або видалити дані: немає входу, ролей, а паролі в ЛР 4 зберігалися відкритим текстом. По-третє, правильність роботи ми перевіряємо очима в браузері й Postman – після кожної зміни коду цю перевірку треба повторювати.

Остання лекція курсу закриває ці прогалини: Spring Data JPA бере на себе доступ до даних, Flyway – версії схеми БД, Spring Security – автентифікацію й авторизацію, а JUnit, Mockito і Spring Boot Test – автоматичну перевірку. Після цього демо-проєкт (тег `lab-08`, версія 1.0.0) має архітектуру, типову для промислових Java-застосунків, і слугує зразком для курсової роботи.

1. Об’єктно-реляційне відображення. JPA і Hibernate

Java оперує **об’єктами** з полями, посиланнями і колекціями, а реляційна БД – **таблицями** з рядками і зовнішніми ключами. Розбіжність цих моделей (object-relational impedance mismatch) проявляється в багатьох місцях: зв’язок «укриття → район» в об’єкті – посилання, а в таблиці – число `district_id`; колекція перевірок укриття в БД – окрема таблиця; успадкування класів у реляційній моделі прямого відповідника не має. У ЛР 2–7 це перетворення виконував наш код (`RowMapper`).

> **ORM** (Object-Relational Mapping, об’єктно-реляційне відображення) – технологія, що автоматично перетворює об’єкти програми на рядки таблиць і навпаки за описаними правилами відповідності.
> **JPA** (Jakarta Persistence API) – стандартна специфікація ORM для Java: анотації, інтерфейс `EntityManager`, мова запитів JPQL. **Hibernate** – найпоширеніша реалізація JPA, яку використовує Spring Boot.

Центральне поняття JPA – **контекст збереження** (persistence context): набір об’єктів-сутностей, якими в межах транзакції керує `EntityManager`. Кожна сутність перебуває в одному зі станів: *new* (щойно створена через `new`), *managed* (керована – завантажена з БД або збережена), *detached* (від’єднана після закриття контексту), *removed* (позначена до видалення). Для керованих сутностей Hibernate виконує **автоматичне відстеження змін** (dirty checking): якщо в межах транзакції змінити поле завантаженого об’єкта, під час фіксації транзакції Hibernate сам

сформує `UPDATE` – виклик `save()` не потрібен. Контекст також є кешем першого рівня: повторний пошук тієї самої сутності за ідентифікатором у межах транзакції не виконує повторного запиту.

Рівень	**Що пише розробник**	**Де в курсі**
JDBC	SQL, `PreparedStatement`, `ResultSet`, закриття ресурсів	ЛР 2–4
Spring JdbcClient	SQL і `RowMapper`	ЛР 5–7
JPA / Hibernate	сутності з анотаціями; SQL генерується	ЛР 8
Spring Data JPA	інтерфейс репозиторію; реалізацію створює Spring	ЛР 8

Таблиця 1 – Рівні абстракції доступу до даних

ORM не скасовує знання SQL: розробник має розуміти, які запити генерує Hibernate, інакше легко отримати повільний застосунок (див. питання 4). Тому в профілі dev демо-проєкту ввімкнено виведення згенерованого SQL (`spring.jpa.show-sql: true`), а `ddl-auto: none` забороняє Hibernate змінювати схему – нею керує Flyway.

2. Сутності та зв’язки між ними

Сутність (entity) – клас, позначений `@Entity`, екземпляри якого зберігаються в таблиці. Обов’язкові вимоги: поле-ідентифікатор з `@Id` і конструктор без параметрів (може бути `protected`) – Hibernate створює об’єкти через нього. Основні анотації наведено в табл. 2.

Анотація	**Призначення**
`@Entity`, `@Table(name="shelters")`	клас – сутність; ім’я таблиці
`@Id`, `@GeneratedValue(strategy=IDENTITY)`	первинний ключ, що генерує БД (`SERIAL`)
`@Column(nullable, length, unique, name)`	налаштування стовпця
`@Enumerated(EnumType.STRING)`	зберігати перелік як назву константи, а не номер
`@ManyToOne`, `@JoinColumn`	багато → один; стовпець зовнішнього ключа
`@OneToMany(mappedBy=…)`	один → багато; зворотна сторона зв’язку
`@OneToOne`, `@ManyToMany`	один-до-одного; багато-до-багатьох (через проміжну таблицю)
`@Transient`	поле не зберігається в БД

Таблиця 2 – Основні анотації JPA

```
@Entity
@Table(name = "shelters")
public class Shelter {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(nullable = false, unique = true)
    private String address;

    @ManyToOne(fetch = FetchType.LAZY, optional = false)     // багато
укриттів → один район
    @JoinColumn(name = "district_id")
    private District district;

    @Enumerated(EnumType.STRING)
    @Column(nullable = false, length = 20)
    private ShelterStatus status = ShelterStatus.NOT_READY;

    @OneToMany(mappedBy = "shelter", cascade = CascadeType.ALL,
orphanRemoval = true)
    @OrderBy("date DESC")
    private List<Inspection> inspections = new ArrayList<>();

    protected Shelter() { }                                   // для
Hibernate

    public Inspection addInspection(User inspector, LocalDate date,
                                    InspectionResult result, String notes)
{
        Inspection inspection = new Inspection(this, inspector, date,
result, notes);
        inspections.add(inspection);
        this.status = ShelterStatus.from(result);           // бізнес-
правило в сутності
        this.lastInspectionDate = date;
        return inspection;
    }
}
```

Лістинг 1 – Сутність Shelter (тег lab-08, скорочено)

У двонаправленому зв’язку одна сторона є **власником** (owning side) – та, що містить зовнішній ключ: `Inspection.shelter` з `@ManyToOne` і `@JoinColumn`. Друга сторона (`Shelter.inspections`) лише відображає зв’язок і позначається `mappedBy`. Hibernate записує в БД зміни саме зі сторони-власника, тому обидві сторони узгоджують у допоміжному методі (`addInspection`). Параметр `cascade = ALL` поширює операції (збереження, видалення) з укриття на його перевірки, а `orphanRemoval` видаляє перевірку, вилучену з колекції.

Перелік зберігають як `STRING`: за типового значення `ORDINAL` у БД потрапить номер константи, і додавання нової константи в середину переліку непомітно зіпсує всі наявні дані. Зверніть увагу, що бізнес-правило «результат перевірки змінює стан

укриття» переїхало в метод сутності – так об'єкт захищає власну узгодженість (насичена модель предметної області). У сервісі метод реєстрації перевірки став коротшим: знайти укриття, викликати `addInspection`, а зберігання виконає dirty checking разом з каскадом.

3. Репозиторії Spring Data JPA. Похідні запити та JPQL

Spring Data JPA дозволяє не писати реалізацію репозиторію: достатньо оголосити інтерфейс, що розширює `JpaRepository<Сутність, ТипІдентифікатора>`, і Spring під час запуску створить бін з реалізацією. Інтерфейс одразу містить методи `findById`, `findAll`, `save`, `deleteById`, `existsById`, `count`, а також варіанти з сортуванням і пагінацією. Три класи `Jdbc*Repository` з ЛР 5–7 (понад 200 рядків SQL і відображення) видалено.

Похідні запити (derived queries) – методи, запит яких Spring Data будує з **назви**: префікс (`find…By`, `exists…By`, `count…By`, `delete…By`), умови з назв полів і ключових слів (`And`, `Or`, `Between`, `LessThan`, `Like`, `IsNull`, `True`, `In`), упорядкування (`OrderBy…Asc/Desc`). Помилку в назві (неіснуюче поле) буде виявлено під час запуску застосунку.

```
public interface ShelterRepository extends JpaRepository<Shelter, Integer>
{

    // Похідні запити: SQL будується з назви методу
    List<Shelter> findByDistrictIdOrderByName(Integer districtId);   //
WHERE district_id=? ORDER BY name
    Page<Shelter> findByStatus(ShelterStatus status, Pageable pageable);
    boolean existsByAddress(String address);
    boolean existsByAddressAndIdNot(String address, Integer id);     //
для редагування
    long countByAccessibleTrue();

    // JPQL: запит до сутностей і їхніх полів, а не до таблиць
    @Query("""
           select s from Shelter s join fetch s.district
           where s.lastInspectionDate is null or s.lastInspectionDate <
:border
           order by s.name""")
    List<Shelter> findInspectionOverdue(@Param("border") LocalDate
border);

    @Query("select coalesce(sum(s.capacity), 0) from Shelter s where
s.district.id = :districtId")
    long totalCapacity(@Param("districtId") Integer districtId);
}
```

Лістинг 2 – Репозиторій Spring Data JPA (тег lab-08, скорочено)

Коли умова складна або потрібна агрегація, використовують анотацію `@Query` з мовою **JPQL** (Jakarta Persistence Query Language). JPQL схожа на SQL, але оперує **сутностями та їхніми полями** (`Shelter s`, `s.district.id`), а не таблицями і стовпцями; Hibernate перекладає її на діалект конкретної СУБД. Параметри позначаються `:ім'я` і, як і в `PreparedStatement`, захищають від SQL-ін'єкцій. Для специфічних можливостей СУБД можна написати й «рідний» SQL (`nativeQuery = true`), але тоді втрачається переносність. Запити, що змінюють дані, позначають додатково `@Modifying`.

Спосіб	**Коли застосовувати**	**Приклад**
Методи `JpaRepository`	CRUD за ідентифікатором	`findById`, `save`
Похідний запит	прості умови за полями	`findByStatus`
`@Query` JPQL	складні умови, `join fetch`, агрегати	`findInspectionOverdue`
`@Query(nativeQuery=true)`	специфічні функції СУБД	повнотекстовий пошук PostgreSQL

Таблиця 3 – Способи формування запитів у Spring Data JPA

4. Оптимізація роботи з даними: ліниве завантаження, проблема N+1, пагінація

Зв'язок можна завантажувати **жадібно** (EAGER – разом з основною сутністю) або **ліниво** (LAZY – окремим запитом лише під час першого звернення). За замовчуванням `@ManyToOne` і `@OneToOne` – жадібні, а колекції (`@OneToMany`, `@ManyToMany`) – ліниві. Рекомендована практика – робити **усі зв'язки лінивими** (`fetch = FetchType.LAZY`) і явно вказувати, що завантажити разом, у конкретному запиті. Замість справжнього об'єкта Hibernate підставляє проксі, який звертається до БД під час першого виклику геттера; якщо транзакція вже закрита, виникне `LazyInitializationException`.

Проблема N+1 запитів – найпоширеніша причина повільної роботи ORM. Список з N укриттів завантажується одним запитом, а потім для виведення назви району кожного укриття виконується ще по одному запиту – разом N+1. Для 6 укриттів це непомітно, для 10 000 – катастрофа. Розв'язання – завантажити пов'язані дані тим самим запитом: `join fetch` у JPQL або анотація `@EntityGraph(attributePaths = "district")` на методі репозиторію (лістинг 3). Перевіряють результат у журналі SQL: замість N+1 рядків `select` має бути один із `join`.

```
// без оптимізації: 1 запит до shelters + 6 запитів до districts
select … from shelters s
select … from districts d where d.id=?      -- × 6

// з @EntityGraph(attributePaths = "district") або join fetch: 1 запит
select s.*, d.* from shelters s join districts d on d.id = s.district_id

@Override
@EntityGraph(attributePaths = "district")
List<Shelter> findAll();
```

Лістинг 3 – Проблема N+1 і її розв'язання

У демо-проєкті також вимкнено режим `spring.jpa.open-in-view`, увімкнений у Spring Boot за замовчуванням. Цей режим тримає з'єднання з БД відкритим до кінця формування сторінки, щоб ліниві зв'язки «чарівно» завантажувалися прямо з шаблону. Це приховує проблему N+1 і утримує з'єднання з пулу довше, ніж потрібно. Без нього сервіс повертає DTO з усіма потрібними даними, а помилки лінивого завантаження виявляються одразу під час розроблення.

Пагінація – повернення великої колекції частинами (сторінками). Spring Data приймає параметр `Pageable` (номер сторінки, розмір, сортування) і повертає `Page<T>` з даними поточної сторінки та загальною кількістю записів і сторінок; Hibernate формує SQL з `LIMIT/OFFSET` і окремий запит `count`. Spring MVC сам створює `Pageable` з параметрів запиту `?page=0&size=5&sort=capacity,desc` (лістинг 4). Віддавати клієнту необмежений список з тисяч записів не можна: це навантаження на БД, мережу і браузер.

```
@GetMapping("/shelters")
public PageResponse<ShelterView> list(
        @ParameterObject @PageableDefault(size = 20, sort = "name")
Pageable pageable) {
    return PageResponse.of(service.findPage(pageable));
}

GET /api/shelters?page=0&size=4&sort=capacity,desc
{"content":[{"name":"Підземний паркінг ТЦ","capacity":450,…}, …],
 "page":0,"size":4,"totalElements":6,"totalPages":2}
```

Лістинг 4 – Пагінація в REST API

5. Міграції бази даних. Flyway

Схема БД змінюється разом із кодом: додаються таблиці, стовпці, індекси. Якщо зміни виконувати вручну, бази розробників, тестового і промислового серверів швидко розійдуться. Автоматичне створення схеми Hibernate (`ddl-auto: update`) для промислового використання непридатне: воно не видаляє і не

перейменовує стовпці, не переносить дані й може непередбачувано змінити таблиці.

> **Міграція БД** – версійований скрипт зміни схеми або даних, що зберігається в репозиторії разом із кодом і застосовується автоматично в певному порядку рівно один раз.

Flyway під час запуску застосунку шукає скрипти в `src/main/resources/db/migration` з іменами за шаблоном `V<версія>__<опис>.sql` (два підкреслення), порівнює їх зі службовою таблицею `flyway_schema_history` і виконує нові за зростанням версії. Для кожної застосованої міграції зберігається контрольна сума: **змінювати вже застосовану міграцію не можна** – Flyway зупинить запуск. Будь-яка зміна оформлюється новою міграцією. Альтернативний інструмент – Liquibase (описи у XML/YAML).

```
src/main/resources/db/migration/
├── V1__schema.sql        таблиці districts, users, shelters, inspections, індекси
├── V2__data.sql          тестові дані; паролі — хеші BCrypt
└── V3__add_phone.sql     наступна зміна: ALTER TABLE shelters ADD COLUMN phone VARCHAR(20);

flyway_schema_history:
 installed_rank | version | description | checksum   | success
 1              | 1       | schema      | -158294... | t
 2              | 2       | data        | 902117...  | t
```

Лістинг 5 – Міграції Flyway демо-проєкту

Для профілю prod, де БД з ЛР 2 вже існує, задано `flyway.baseline-on-migrate: true` і `baseline-version: 2`: Flyway позначить наявну схему як версію 2 і застосовуватиме лише нові міграції. Підключення: стартер `spring-boot-starter-flyway` і модуль `flyway-database-postgresql`.

6. Основи Spring Security: автентифікація, авторизація, зберігання паролів

Spring Security – фреймворк безпеки застосунків Spring. Його архітектура знайома з ЛР 4: це **ланцюжок фільтрів сервлетів** (`SecurityFilterChain`), що обробляє кожен запит до `DispatcherServlet`. Фільтри витягують облікові дані (з форми, заголовка `Authorization` або сесії), перевіряють їх, зберігають результат у `SecurityContext` і вирішують, чи пропустити запит. Лише підключення стартера `spring-boot-starter-security` закриває *усі* адреси і створює тимчасового користувача з паролем у журналі – далі доступ налаштовують явно.

Поняття	Питання	Помилка
Автентифікація	хто ви? (логін і пароль, токен)	401 Unauthorized або перенаправлення на сторінку входу
Авторизація	що вам дозволено? (роль, права)	403 Forbidden

Таблиця 4 – Автентифікація і авторизація

Користувачів Spring Security отримує через інтерфейс `UserDetailsService` – у демо-проєкті його реалізація читає таблицю `users` через JPA-репозиторій. **Паролі ніколи не зберігають відкритим текстом** і не шифрують зворотно: зберігають результат повільної односторонньої **хеш-функції з сіллю** – BCrypt, Argon2, PBKDF2. Сіль (випадкове значення в кожному хеші) робить однакові паролі різними хешами і знешкоджує заздалегідь обчислені таблиці, а навмисна повільність (параметр вартості `$10$`) робить перебір дорогим. Швидкі хеш-функції MD5 і SHA-256 для паролів непридатні. `DelegatingPasswordEncoder` розпізнає алгоритм за префіксом `{bcrypt}`, що дозволяє згодом перейти на інший алгоритм без втрати старих паролів. У ЛР 4 паролі мали префікс `{noop}` (без хешування) – тепер його замінено.

```
@Bean
UserDetailsService userDetailsService(UserRepository users) {
    return username -> users.findByUsername(username)
            .map(u -> User.withUsername(u.getUsername())
                    .password(u.getPasswordHash())        //
{bcrypt}$2y$10$oaUim9…
                    .roles(u.getRole().name())            // ADMIN або
INSPECTOR
                    .build())
            .orElseThrow(() -> new UsernameNotFoundException(username));
}

@Bean
PasswordEncoder passwordEncoder() {
    return PasswordEncoderFactories.createDelegatingPasswordEncoder();
}
```

Лістинг 6 – Користувачі з БД і кодувальник паролів

7. Захист вебсторінок і REST API. CSRF

Вебсторінки і REST API мають різних клієнтів, тому в демо-проєкті налаштовано **два ланцюжки фільтрів** (табл. 5). Для сторінок – вхід через форму (`formLogin`) і сесія з cookie `JSESSIONID`; для API – **HTTP Basic** (логін і пароль у заголовку `Authorization` кожного запиту) без сесії, що відповідає принципу REST «без стану». У реальних системах для API частіше використовують токени (JWT, OAuth 2.0), але принцип той самий.

Параметр	Вебсторінки	REST API (/api/**)
Вхід	форма `/login`	HTTP Basic (Postman, curl `-u admin:admin`)
Стан	сесія, cookie `JSESSIONID`	без сесії (`STATELESS`)
CSRF	увімкнено, токен у кожній формі	вимкнено (немає cookie-автентифікації)
Гість	перегляд списку і карток	усі GET-запити
INSPECTOR	реєстрація перевірок	POST `/api/shelters/*/inspections`
ADMIN	створення, редагування, видалення	усі інші зміни

Таблиця 5 – Ланцюжки безпеки демо-проєкту (тег lab-08)

```
@Bean @Order(1)
SecurityFilterChain apiSecurity(HttpSecurity http) throws Exception {
    http.securityMatcher("/api/**")
        .authorizeHttpRequests(auth -> auth
            .requestMatchers(HttpMethod.GET, "/api/**").permitAll()
            .requestMatchers(HttpMethod.POST,
"/api/shelters/*/inspections")
                .hasAnyRole("INSPECTOR", "ADMIN")
            .anyRequest().hasRole("ADMIN"))
        .httpBasic(Customizer.withDefaults())
        .sessionManagement(s ->
s.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .csrf(csrf -> csrf.disable());
    return http.build();
}

@Bean @Order(2)
SecurityFilterChain webSecurity(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests(auth -> auth
            .requestMatchers("/css/**", "/login", "/error").permitAll()
            .requestMatchers("/shelters/new",
"/shelters/*/edit").hasRole("ADMIN")
            .requestMatchers(HttpMethod.GET, "/shelters",
"/shelters/{id}").permitAll()
            .requestMatchers(HttpMethod.POST,
"/shelters/*/inspections").hasAnyRole("INSPECTOR", "ADMIN")
            .requestMatchers("/shelters/**").hasRole("ADMIN")
            .anyRequest().authenticated())
        .formLogin(form ->
form.loginPage("/login").defaultSuccessUrl("/shelters").permitAll())
        .logout(logout -> logout.logoutSuccessUrl("/shelters"));
    return http.build();
}
```

Лістинг 7 – Конфігурація SecurityConfig (скорочено)

Правила перевіряються **за порядком**, і спрацьовує перше, що збіглося. Тому правило для `/shelters/new` стоїть раніше за загальне `GET /shelters/{id}` –

інакше шаблон `{id}` збігся б зі словом «new», і сторінка створення стала б доступною гостю. Завершальне правило `anyRequest()` закриває все, що не дозволено явно (принцип «заборонено все, що не дозволено»). Крім правил за URL, доступ можна обмежувати на рівні методів сервісу анотацією `@PreAuthorize("hasRole('ADMIN')")` (вмикається `@EnableMethodSecurity`).

CSRF (Cross-Site Request Forgery, міжсайтова підробка запиту) – атака, за якої сторінка зловмисника змушує браузер автентифікованого користувача надіслати запит на наш сайт: браузер автоматично додасть cookie сесії, і сервер виконає, наприклад, видалення укриття. Захист – **CSRF-токен**: випадкове значення, яке сервер вставляє в кожну форму як приховане поле і перевіряє під час POST; сайт зловмисника цього значення не знає. Thymeleaf додає токен у форми з `th:action` автоматично. Для API з HTTP Basic захист вимкнено, бо браузер не надсилає облікові дані такого API автоматично. Ім'я поточного користувача сервіс отримує зі `SecurityContext` (параметр `Principal`), тому інспектор, що зареєстрував перевірку, тепер визначається за входом, а не передається в тілі запиту, де його можна було б підробити.

8. Модульне тестування: JUnit 5, AssertJ, Mockito

Автоматичні тести перевіряють поведінку коду після кожної зміни за секунди і є виконуваною документацією вимог. Їх співвідношення описує **піраміда тестування**: у основі – багато швидких **модульних** (unit) тестів окремих класів, вище – менше **інтеграційних** тестів взаємодії шарів (з БД, вебшаром), на вершині – небагато повільних наскрізних (end-to-end) тестів через браузер. Стартери `spring-boot-starter-*-test` підключають усе потрібне (табл. 6).

Інструмент	Призначення
JUnit 5 (Jupiter)	запуск тестів: `@Test`, `@BeforeEach`, `@DisplayName`, `@ParameterizedTest`
AssertJ	«плавні» перевірки: `assertThat(x).isEqualTo(…)`, `assertThatThrownBy`
Mockito	заглушки (mocks) залежностей: `when(…).thenReturn(…)`, `verify(…)`
Spring Boot Test	контекст для тестів: `@SpringBootTest`, `@WebMvcTest`, `@DataJpaTest`
MockMvc	імітація HTTP-запитів до контролерів без запуску сервера
Spring Security Test	запити від імені користувача: `with(user("admin").roles("ADMIN"))`

Таблиця 6 – Інструменти тестування

Модульний тест перевіряє один клас ізольовано, без Spring і без БД: залежності замінюються **заглушками Mockito**, яким задають поведінку, а після виклику перевіряють, які методи було викликано. Тест будують за схемою **Arrange – Act – Assert** (підготувати – виконати – перевірити). Саме тут окупається впровадження залежностей через конструктор (лекція 5) і бін `Clock`: сервіс створюється звичайним `new` з фіксованим годинником, тож тест не залежить від дати запуску.

```
@ExtendWith(MockitoExtension.class)
class ShelterServiceTest {
    @Mock ShelterRepository shelters;
    @Mock DistrictRepository districts;
    final Clock clock = Clock.fixed(Instant.parse("2027-03-01T10:00:00Z"),
ZoneId.of("Europe/Kyiv"));
    ShelterService service;
    // @BeforeEach: service = new ShelterService(shelters, districts, …,
clock, properties);

    @Test
    @DisplayName("Не можна зареєструвати два укриття за однією адресою")
    void createRejectsDuplicateAddress() {
        when(shelters.existsByAddress("вул. Наукова,
7")).thenReturn(true);       // Arrange
        when(districts.findById(5)).thenReturn(Optional.of(district));

        assertThatThrownBy(() -> service.create(form("вул. Наукова, 7")))
// Act + Assert
                .isInstanceOf(ValidationException.class);
        verify(shelters, never()).save(any());
// нічого не збережено
    }
}
```

Лістинг 8 – Модульний тест сервісу з Mockito

9. Інтеграційне тестування: Spring Boot Test, MockMvc, @DataJpaTest

Модульні тести не перевіряють, чи правильно написано JPQL, чи працюють правила безпеки, чи коректно контролер перетворює JSON. Для цього Spring Boot пропонує **тести зрізів** (test slices), що піднімають лише потрібну частину контексту, і повний інтеграційний тест (табл. 7).

Анотація	Що піднімається	Що перевіряють
`@DataJpaTest`	JPA, репозиторії, БД (H2), Flyway; кожен тест відкочується	похідні запити, JPQL, пагінацію, відображення сутностей
`@WebMvcTest(X.class)`	Spring MVC з одним контролером; сервіси – `@MockitoBean`	маршрути, JSON, валідацію, коди стану, безпеку
`@SpringBootTest`	увесь застосунок	наскрізні сценарії через усі шари

Таблиця 7 – Види тестів Spring Boot

```
@DataJpaTest
@AutoConfigureTestDatabase(replace =
AutoConfigureTestDatabase.Replace.NONE)
class ShelterRepositoryTest {
    @Autowired ShelterRepository repository;

    @Test
    void pagination() {
        Page<Shelter> page = repository.findAll(PageRequest.of(0, 4,
Sort.by("capacity").descending()));
        assertThat(page.getTotalElements()).isEqualTo(6);
        assertThat(page.getTotalPages()).isEqualTo(2);

assertThat(page.getContent().getFirst().getCapacity()).isEqualTo(450);
    }
}

@WebMvcTest(ShelterApiController.class)
@Import(SecurityConfig.class)
class ShelterApiControllerTest {
    @Autowired MockMvc mvc;
    @MockitoBean ShelterService service;

    @Test
    void inspectorCannotCreate() throws Exception {

mvc.perform(post("/api/shelters").with(user("inspector").roles("INSPECTOR"))

.contentType(MediaType.APPLICATION_JSON).content(VALID_JSON))
            .andExpect(status().isForbidden());
    }

    @Test
    void invalidBodyIs400WithFieldErrors() throws Exception {

mvc.perform(post("/api/shelters").with(user("admin").roles("ADMIN"))
                        .contentType(MediaType.APPLICATION_JSON)
                        .content("
{\"name\":\"\",\"districtId\":4,\"type\":\"BASEMENT\",\"capacity\":0}"))
            .andExpect(status().isBadRequest())
            .andExpect(jsonPath("$.errors.name").value("Вкажіть назву
укриття"));
    }
}
```

Лістинг 9 – Тест репозиторію і тест вебшару з безпекою

Тести запускаються командою `mvn test` (або на фазі `package`), а також автоматично в CI (наприклад, GitHub Actions) під час кожного `push`, щоб помилка не потрапила в основну гілку. Добрий тест незалежний від інших і від порядку запуску, детермінований (фіксований час, власні дані), перевіряє одну поведінку і має назву, що її описує. Для перевірки з тією самою СУБД, що в експлуатації,

використовують бібліотеку **Testcontainers**, яка запускає PostgreSQL у Docker на час тестів.

Отже, демо-проєкт пройшов повний шлях курсу (табл. 8). Цей самий шлях здобувачі освіти пройшли у власних проєктах, які стають основою курсової роботи.

Тег	Що додано
`lab-01–lab-02`	статична сторінка, HTTP; схема PostgreSQL, JDBC, транзакції
`lab-03–lab-04`	шари MVC, доступність; сервлети, JSP, сесії, фільтри, Tomcat
`lab-05–lab-06`	Spring Boot, IoC/DI, JdbcClient; YAML, профілі, журнали, Docker
`lab-07`	Thymeleaf, RESTful API, валідація, Problem Details, Swagger
`lab-08`	Spring Data JPA, Flyway, Spring Security, автоматичні тести

Таблиця 8 – Еволюція демо-проєкту «Реєстр укриттів»

Висновки

JPA і Hibernate відображають сутності на таблиці, а Spring Data JPA створює реалізацію репозиторіїв з похідних запитів і JPQL. Зв’язки описуються анотаціями `@ManyToOne`/`@OneToMany` із чітко визначеною стороною-власником; усі зв’язки роблять лінивими, а потрібні дані завантажують `join fetch` чи `@EntityGraph`, уникаючи проблеми N+1; великі колекції віддають посторінково. Схемою БД керують версійні міграції Flyway. Spring Security реалізує автентифікацію й авторизацію ланцюжком фільтрів, паролі зберігаються хешами BCrypt, форми захищені CSRF-токенами, а API – HTTP Basic без сесій. Модульні тести з Mockito перевіряють бізнес-логіку ізольовано, тести `@DataJpaTest` і `@WebMvcTest` – шари доступу до даних і веб, разом утворюючи піраміду тестування.

Контрольні запитання

1. Що таке ORM і JPA? Яке місце Hibernate?
2. Що таке контекст збереження і автоматичне відстеження змін?
3. Які вимоги до класу-сутності? Чому переліки зберігають як STRING?
4. Що таке сторона-власник зв’язку? Для чого `mappedBy`, `cascade`, `orphanRemoval`?
5. Як Spring Data JPA формує запит з назви методу? Наведіть приклади.
6. Чим JPQL відрізняється від SQL?
7. Що таке ліниве завантаження? У чому полягає проблема N+1 і як її усунути?

8. Як реалізується пагінація в Spring Data і REST API?
9. Що таке міграція БД? Як працює Flyway і чому не можна змінювати застосовану міграцію?
10. Чим автентифікація відрізняється від авторизації? Які коди стану їм відповідають?
11. Як правильно зберігати паролі? Навіщо сіль і повільна хеш-функція?
12. Що таке CSRF і як від нього захищає токен? Чому для API з HTTP Basic захист вимикають?
13. Що таке піраміда тестування? Чим модульний тест відрізняється від інтеграційного?
14. Для чого `@DataJpaTest`, `@WebMvcTest`, `@SpringBootTest` і MockMvc?

Завдання для самостійної роботи

Опрацювати матеріал лекції. Перевести доступ до даних власного вебсервісу курсової роботи на Spring Data JPA, налаштувати міграції, автентифікацію та авторизацію, написати модульні та інтеграційні тести. Ознайомитися з OWASP Top 10 і посібниками «Accessing Data with JPA», «Securing a Web Application», «Testing the Web Layer» на spring.io/guides. Пройти тест № 8 у «Віртуальному університеті». Підготуватися до захисту курсової роботи та екзамену.

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 і курсу загалом, відповісти на запитання здобувачів освіти. Наголосити на зв’язку матеріалу з лабораторною роботою № 8, курсовою роботою і питаннями екзамену. 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.