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 № 7. Тема 2.3: «Розроблення вебзастосунків і RESTful API на Spring Boot»

Мета: сформувати у здобувачів освіти знання про архітектуру Spring MVC, маршрутизацію запитів, роботу з параметрами, заголовками, сесіями і cookies, формування вебсторінок засобами Thymeleaf, принципи REST, а також навички розроблення RESTful API з валідацією вхідних даних, централізованою обробкою помилок і документуванням за стандартом OpenAPI.

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 із засобами розробника, Postman або curl, презентація до лекції, електронний курс у «Віртуальному університеті», тематична вікторина до лекції.

Навчальні питання:

1. Spring MVC: контролери, представлення і маршрутизація запитів.
2. Робота з параметрами запиту, заголовками, cookies і сесіями.
3. Шаблонізатор Thymeleaf: синтаксис, ітерації, умови, фрагменти.
4. Форми на вебсторінках: зв'язування з об'єктом, валідація, шаблон PRG.
5. Принципи архітектурного стилю REST.
6. Розроблення RESTful API: CRUD-операції, JSON, ResponseEntity, коди стану.
7. Валідація вхідних даних засобами Bean Validation.
8. Централізована обробка помилок. Формат Problem Details.

9. Документування API: OpenAPI і Swagger UI.

Після лекційного заняття здобувачі освіти повинні:

знати: відмінність `@Controller` від `@RestController`; способи отримання даних запиту; механізми cookies, сесій і flash-атрибутів; основні вирази й атрибути Thymeleaf; обмеження архітектурного стилю REST; семантику HTTP-методів і кодів стану; анотації Bean Validation; призначення `@RestControllerAdvice` і формату Problem Details; структуру документації OpenAPI;

вміти: створювати контролери вебсторінок і REST API; зіставляти запити з методами; формувати сторінки на Thymeleaf з фрагментами; обробляти форми з виведенням помилок біля полів; проєктувати URI ресурсів; повертати правильні коди стану і заголовок `Location`; валідувати вхідні дані; обробляти помилки централізовано; документувати API в Swagger UI.

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 Framework Reference Documentation. Spring Web MVC [Електронний ресурс]. – Доступний з: https://docs.spring.io/spring-framework/reference/web/webmvc.html
3. Thymeleaf Documentation. Tutorial: Using Thymeleaf; Thymeleaf + Spring [Електронний ресурс]. – Доступний з: https://www.thymeleaf.org/documentation.html
4. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : Doctoral dissertation. – University of California, Irvine, 2000.
5. RFC 9110. HTTP Semantics; RFC 9457. Problem Details for HTTP APIs [Електронний ресурс]. – IETF. – Доступний з: https://www.rfc-editor.org/
6. OpenAPI Specification 3.1; springdoc-openapi [Електронний ресурс]. – Доступний з: https://springdoc.org/
7. Jakarta Bean Validation 3.1 Specification [Електронний ресурс]. – Доступний з: https://jakarta.ee/specifications/bean-validation/

І. Підготовка до заняття

Перед початком заняття викладач перевіряє роботу проєктора, доступ до демо-проєкту (теги `lab-06` і `lab-07`), можливість запуску застосунку командою `mvn spring-boot:run` (профіль dev, БД H2), доступність сторінок http://localhost:8080/shelters і http://localhost:8080/swagger-ui.html, наявність Postman або curl і готовність тематичної вікторини до лекції (відкривається з «Віртуального університету» на проекторі; проводиться наприкінці заняття, 10–15 хв).

ІІ. Зміст лекції

Вступ

Після лабораторної роботи № 6 «Реєстр укриттів» уміє лише віддавати дані: кілька GET-маршрутів повертають JSON. Користувачам потрібні зручні вебсторінки з таблицями, фільтрами і формами, а іншим програмам (мобільному застосунку, системі оповіщення) – повноцінний програмний інтерфейс для створення, зміни і видалення даних. Обидва канали мають однаково перевіряти вхідні дані і зрозуміло повідомляти про помилки.

У модулі 1 ми робили це вручну: сервлет, `doGet`/`doPost`, JSP, ручний розбір параметрів, власні сторінки помилок. На цій лекції побачимо, як ці самі задачі розв'язує Spring MVC: контролери з анотаціями, шаблонізатор Thymeleaf, автоматичне зв'язування й валідація даних, централізована обробка помилок і автоматична документація API (тег `lab-07` демо-проєкту).

1. Spring MVC: контролери, представлення і маршрутизація запитів

Spring MVC реалізує шаблон **Model–View–Controller**: *контролер* приймає запит, викликає бізнес-логіку і заповнює *модель* (набір іменованих даних), а *представлення* (шаблон сторінки) формує з моделі HTML. Центральний сервлет `DispatcherServlet` (лекція 5) за допомогою `HandlerMapping` знаходить метод контролера, а за допомогою `ViewResolver` – шаблон за іменем, яке повернув метод (рис. 1).

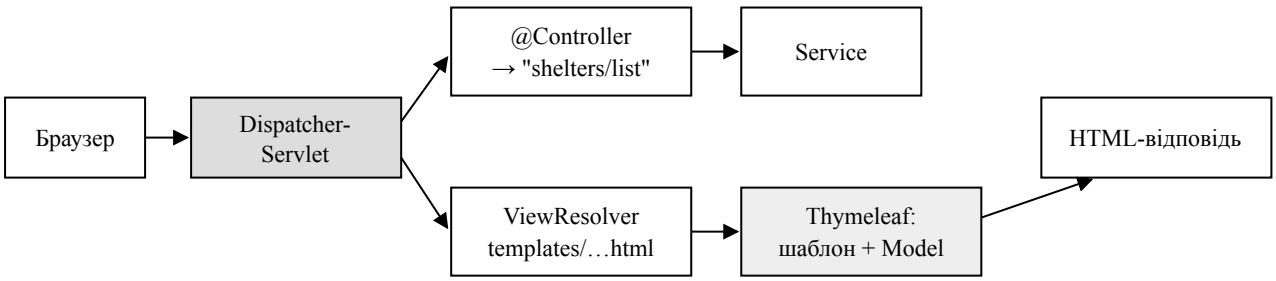

Рисунок 1 – Обробка запиту вебсторінки в Spring MVC

У застосунку зазвичай є два види контролерів (табл. 1). `@Controller` повертає **ім'я представлення**, а дані передає через параметр `Model`; `@RestController` повертає **дані**, які перетворюються на JSON. У демо-проєкті вони розділені за пакетами: `web.ShelterWebController` обслуговує сторінки за адресою `/shelters`, а `api.ShelterApiController` – програмний інтерфейс

`/api/shelters`. Обидва використовують той самий `ShelterService`, тому бізнес-правила не дублюються.

Ознака	**@Controller**	**@RestController**
Що повертає метод	ім'я шаблону: `"shelters/list"` або `"redirect:/…"`	об'єкт або `ResponseEntity` → JSON
Дані для відповіді	`Model`, `@ModelAttribute`	значення, яке повертає метод
Клієнт	браузер користувача	JavaScript, мобільний застосунок, інший сервіс
Помилки	сторінка помилки	JSON з описом помилки і кодом стану

Таблиця 1 – Контролери сторінок і REST API

Маршрутизація задається анотаціями: `@RequestMapping("/shelters")` на класі визначає спільний префікс, `@GetMapping`, `@PostMapping`, `@PutMapping`, `@PatchMapping`, `@DeleteMapping` на методах – HTTP-метод і решту шляху. Шаблони шляху можуть містити змінні (`/{id}`) і обмеження регулярним виразом (`/{id:\d+}`). Атрибути `params`, `headers`, `consumes`, `produces` звужують зіставлення: наприклад, `consumes = "application/json"` приймає лише JSON. Якщо для шляху немає методу з потрібним HTTP-методом, Spring відповідає кодом **405 Method Not Allowed**, а якщо шлях невідомий – **404**.

```
@Controller
@RequestMapping("/shelters")
public class ShelterWebController {
    private final ShelterService service;
    public ShelterWebController(ShelterService service) { this.service
= service; }

    @GetMapping("/{id}")                                    // GET
/shelters/3
    public String detail(@PathVariable int id, Model model) {
        model.addAttribute("shelter", service.findById(id));
        model.addAttribute("inspections", service.findInspections(id));
        return "shelters/detail";                           //
templates/shelters/detail.html
    }

    @ModelAttribute("districts")                            // доступно
на всіх сторінках контролера
    public List<District> districts() { return
service.findAllDdistricts(); }
}
```

Лістинг 1 – Контролер вебсторінок (фрагмент)

Метод з анотацією `@ModelAttribute("districts")` викликається перед кожним методом обробки цього контролера і додає результат у модель – так перелік районів потрапляє і в фільтр списку, і у форму редагування без дублювання коду.

2. Робота з параметрами запиту, заголовками, cookies і сесіями

Spring MVC сам витягує дані з HTTP-запиту і перетворює їх на типи Java за анотаціями параметрів методу (табл. 2). Якщо обов'язкового параметра немає або його неможливо перетворити (наприклад, `?districtId=abc`), Spring відповідає кодом 400 ще до виклику методу.

Анотація / тип	**Звідки дані**	**Приклад**
`@PathVariable`	частина шляху	`/shelters/{id}` → `int id`
`@RequestParam`	рядок запиту або поле форми	`?districtId=5`; `required`, `defaultValue`
`@RequestHeader`	заголовок	`@RequestHeader("Accept-Language") String lang`
`@CookieValue`	cookie	`@CookieValue("district") Integer d`
`@ModelAttribute`	поля форми → об'єкт	`ShelterForm form`
`@RequestBody`	тіло запиту (JSON) → об'єкт	REST API
`HttpSession`, `@SessionAttribute`	сесія користувача	кошик, поточний користувач
`HttpServletRequest/Response`	повний доступ до Servlet API	встановлення cookie

Таблиця 2 – Отримання даних запиту в Spring MVC

У демо-проєкті сторінка списку **запам'ятовує обраний район у cookie**: якщо користувач обрав район, сервер надсилає заголовок `Set-Cookie: district=5; Max-Age=2592000; HttpOnly`, і під час наступного відкриття сторінки без параметра фільтр відновлюється з cookie (лістинг 2). Значення 0 («усі райони») видаляє cookie встановленням `Max-Age=0`.

```
@GetMapping
public String list(@RequestParam(required = false) Integer districtId,
                   @CookieValue(name = "district", required = false)
Integer savedDistrict,
                   HttpServletResponse response, Model model) {
    Integer filter = districtId != null ? districtId : savedDistrict;
    if (districtId != null) {
        Cookie cookie = new Cookie("district",
String.valueOf(districtId));
        cookie.setPath("/");
        cookie.setHttpOnly(true);
        cookie.setMaxAge(districtId == 0 ? 0 : 30 * 24 * 3600);   // 30
днів або видалити
        response.addCookie(cookie);
    }
    …
    return "shelters/list";
}
```

Лістинг 2 – Читання і встановлення cookie (скорочено)

Сесії в Spring MVC – це ті самі сесії контейнера сервлетів з ЛР 4 (cookie `JSESSIONID`). Отримати сесію можна параметром `HttpSession session`, прочитати атрибут – `@SessionAttribute("user")`. У демо-проєкті властивість `server.servlet.session.tracking-modes: cookie` забороняє передавати ідентифікатор сесії в URL (`;jsessionid=…`), бо такий URL міг би потрапити в журнали чи до інших людей. Окремий випадок сесії – **flash-атрибути**: `RedirectAttributes.addFlashAttribute("message", …)` зберігає повідомлення лише до наступного запиту, що ідеально підходить для шаблону PRG («Укриття додано» після перенаправлення). REST API, на відміну від сторінок, сесій не використовує (див. питання 5).

3. Шаблонізатор Thymeleaf: синтаксис, ітерації, умови, фрагменти

Thymeleaf – сучасний серверний шаблонізатор для Java, що замінює JSP. Його головна ідея – **«природні шаблони»** (natural templates): шаблон є коректним HTML-файлом, який можна відкрити в браузері як макет, бо логіка записується в атрибутах `th:*`, а не у вставках `<% %>`. Під час обробки Thymeleaf замінює вміст елементів даними з моделі. Шаблони розміщують у `src/main/resources/templates`, статичні файли (CSS, зображення) – у `static`. Підключення: стартер `spring-boot-starter-thymeleaf`.

Вираз	Призначення	Приклад
`${…}`	змінна моделі	`th:text="${shelter.name}"`
`*{…}`	поле обраного об’єкта (`th:object`)	`th:field="*{capacity}"`
`@{…}`	URL з урахуванням контексту і параметрів	`@{/shelters/{id} (id=${s.id})}`
`#{…}`	повідомлення з файлу локалізації	`#{shelter.title}`
`~{…}`	фрагмент шаблону	`~{fragments/layout :: header}`
`|…|`	підстановка в рядок	`|Знайдено: ${shelters.size()}|`

Таблиця 3 – Вирази Thymeleaf

Основні атрибути: `th:text` – вивести текст з **екрануванням** HTML (захист від XSS, як `c:out` у JSP); `th:utext` – без екранування (лише для довіреного HTML); `th:each` – цикл; `th:if`/`th:unless` – умова; `th:href`, `th:action`, `th:value`, `th:selected`, `th:class` – значення атрибутів. Для виразів доступні службові об’єкти: `#strings`, `#temporals` (форматування дат), `#lists`, `#fields` (помилки форми).

```
<table>
  <caption th:text="|Знайдено укриттів: ${shelters.size()}, місткість:
${totalCapacity} осіб|">
    Укриття</caption>
  <tr th:each="s : ${shelters}">
    <td><a th:href="@{/shelters/{id}(id=${s.id})}"
th:text="${s.name}">Назва</a></td>
    <td th:text="${s.address}">Адреса</td>
    <td th:text="${s.capacity}">0</td>
  </tr>
  <tr th:if="${shelters.isEmpty()}">
    <td colspan="3">У цьому районі укриттів ще не зареєстровано.</td>
  </tr>
</table>
```

Лістинг 3 – Таблиця укриттів (templates/shelters/list.html, скорочено)

Текст «Назва», «Адреса», «0» у лістингу 3 – це вміст-заглушка, який видно під час відкриття шаблону як файлу; під час обробки його замінять дані. Спільні частини сторінок (заголовок `head`, шапка з навігацією, блок повідомлень) винесено у **фрагменти** файлу `fragments/layout.html` і підключено атрибутом `th:replace`. Фрагменти можуть мати параметри – так кожна сторінка передає

свій заголовок (лістинг 4). Це аналог `<%@ include %>` з JSP, але без порушення структури HTML.

```
<!-- fragments/layout.html -->
<head th:fragment="head(title)">
  <meta charset="UTF-8">
  <title th:text="${title} + ' — Реєстр укриттів'">Реєстр
укриттів</title>
  <link rel="stylesheet" th:href="@{/css/style.css}">
</head>
<div th:fragment="flash">
  <p class="message" role="status" th:if="${message}"
th:text="${message}">Повідомлення</p>
</div>

<!-- shelters/list.html -->
<head th:replace="~{fragments/layout :: head(title='Укриття')}"></head>
<div th:replace="~{fragments/layout :: header}"></div>
```

Лістинг 4 – Фрагменти з параметрами

4. Форми на вебсторінках: зв’язування з об’єктом, валідація, шаблон PRG

Thymeleaf і Spring MVC зв’язують поля HTML-форми з полями Java-об’єкта (**об’єкта форми**, form backing object). Атрибут `th:object="${form}"` обирає об’єкт, а `th:field="*{name}"` на полі введення одночасно встановлює атрибути `id`, `name` і `value` (для перемикачів – `checked`). Після надсилання форми Spring створює об’єкт `ShelterForm`, заповнює його значеннями полів і перетворює типи (рядок «250» → `Integer`, «BASEMENT» → перелік `ShelterType`).

```
<form th:action="@{/shelters}" th:object="${form}" method="post"
novalidate>
  <label for="name">Назва</label>
  <input type="text" th:field="*{name}"
         th:aria-describedby="${#fields.hasErrors('name')} ? 'name-
error'">
  <p class="field-error" id="name-error"
     th:if="${#fields.hasErrors('name')}" th:errors="*
{name}">Помилка</p>

  <select th:field="*{districtId}">
    <option value="">— оберіть район —</option>
    <option th:each="d : ${districts}" th:value="${d.id}"
th:text="${d.name}">Район</option>
  </select>
  <button type="submit">Зберегти</button>
</form>
```

Лістинг 5 – Форма з виведенням помилок біля поля (templates/shelters/form.html)

Обробник форми (лістинг 6) приймає об’єкт з анотаціями `@Valid` і `@ModelAttribute("form")`. Наступним параметром **обов’язково** йде `BindingResult` – сюди Spring складає помилки перетворення типів і валідації; якщо цього параметра немає, у разі помилок метод не викличеться, а користувач отримає сторінку 400. Якщо помилки є, метод повертає ту саму форму: Thymeleaf покаже введені значення і повідомлення біля полів (`th:errors`). Бізнес-помилки сервісу (адреса вже зареєстрована) теж додаються до `BindingResult` методом `rejectValue`, тому показуються однаково. У разі успіху виконується перенаправлення `redirect:/shelters/7` з flash-повідомленням – шаблон **Post/Redirect/Get** з ЛР 4 у Spring записується одним рядком.

```
@PostMapping
public String create(@Valid @ModelAttribute("form") ShelterForm form,
BindingResult binding,
                     RedirectAttributes redirect) {
    if (binding.hasErrors()) {
        return "shelters/form";                          // помилки біля
полів
    }
    try {
        ShelterView created = service.create(form);
        redirect.addFlashAttribute("message", "Укриття «" +
created.name() + "» додано");
        return "redirect:/shelters/" + created.id();    // PRG
    } catch (ValidationException e) {
        e.getErrors().forEach((field, msg) ->
binding.rejectValue(field, "business", msg));
        return "shelters/form";
    }
}
```

Лістинг 6 – Обробка форми створення укриття

Атрибут `novalidate` у формі вимикає вбудовану перевірку браузера, щоб на лекції було видно серверну валідацію; у робочому застосунку атрибути `required`, `min`, `max` залишають для зручності користувача, але **сервер перевіряє дані завжди** – браузерну перевірку легко обійти. Атрибути `aria-invalid` і `aria-describedby` пов’язують поле з повідомленням про помилку для екранних читачів – вимоги доступності з ЛР 3 продовжують діяти.

5. Принципи архітектурного стилю REST

REST (Representational State Transfer, передавання стану представлення) – архітектурний стиль розподілених систем, описаний Роєм Філдінгом у 2000 р.

API, що дотримується його обмежень, називають **RESTful**. Основні обмеження: **клієнт-сервер** (розділення інтерфейсу і даних); **відсутність стану** (stateless) – кожен запит містить усю потрібну інформацію, сервер не зберігає сесію клієнта; **кешованість** – відповіді позначаються як такі, що можна кешувати; **єдиний інтерфейс** – ресурси ідентифікуються URI, а дії – стандартними методами HTTP; **багаторівнева система** – між клієнтом і сервером можуть бути проксі, балансувальники.

> **Ресурс** – будь-яка сутність, до якої клієнт має доступ (укриття, перевірка, район), що ідентифікується URI. **Представлення** ресурсу – його поточний стан у певному форматі, найчастіше JSON.

Правила проєктування URI: ресурси – **іменники у множині** (`/api/shelters`), окремий елемент – за ідентифікатором (`/api/shelters/3`), вкладені ресурси – через ієрархію (`/api/shelters/3/inspections`), фільтрація – параметрами (`?districtId=5`). Дієслова в URI не використовують: не `/api/createShelter` чи `/api/deleteShelter?id=3`, а `POST /api/shelters` і `DELETE /api/shelters/3`. Семантику методів наведено в табл. 4.

Метод	Дія	Безпечний	Ідемпотентний	Типова відповідь
GET	отримати ресурс або колекцію	так	так	200 OK
POST	створити ресурс у колекції	ні	ні	201 Created + `Location`
PUT	замінити ресурс повністю	ні	так	200 OK або 204
PATCH	змінити частину ресурсу	ні	не обов’язково	200 OK
DELETE	видалити ресурс	ні	так	204 No Content

Таблиця 4 – Методи HTTP у RESTful API

Безпечний метод не змінює стан сервера; **ідемпотентний** – дає той самий результат за повторного виконання: двічі виконаний `DELETE /api/shelters/3` залишить ресурс видаленим, а двічі виконаний `POST` створить два укриття. Від цього залежить, чи можна безпечно повторити запит після мережевої помилки. Коди стану, що використовуються в API, згруповано в табл. 5.

Код	Коли повертати
200 OK	успішне читання або оновлення з тілом відповіді
201 Created	ресурс створено; адреса – у заголовку `Location`
204 No Content	успіх без тіла (видалення)
400 Bad Request	некоректні дані: помилки валідації, не JSON
401 / 403	не автентифіковано / немає прав (тема 2.4)
404 Not Found	ресурсу з таким ідентифікатором немає
405 Method Not Allowed	метод не підтримується для цього URI
409 Conflict	конфлікт зі станом даних (порушення цілісності, дублікат)
500 Internal Server Error	непередбачена помилка сервера – у правильному API трапляється лише через дефекти

Таблиця 5 – Коди стану HTTP у REST API

6. Розроблення RESTful API: CRUD-операції, JSON, ResponseEntity, коди стану

Повний набір операцій **CRUD** (Create, Read, Update, Delete) для укриттів наведено в лістингу 7. Тіло запиту у форматі JSON зв’язується з об’єктом анотацією `@RequestBody`; перетворення JSON ↔ Java виконує бібліотека **Jackson** через `HttpMessageConverter`. Коли потрібно керувати не лише тілом, а й кодом стану і заголовками, метод повертає **`ResponseEntity<T>`**: `ResponseEntity.created(uri).body(x)` дає 201 із заголовком `Location`, `ResponseEntity.noContent().build()` – 204.

```
@RestController
@RequestMapping("/api")
public class ShelterApiController {
    @GetMapping("/shelters")
    public List<ShelterView> list(@RequestParam(required = false)
Integer districtId) { … }

    @GetMapping("/shelters/{id}")
    public ShelterView get(@PathVariable int id) { return
service.findById(id); }

    @PostMapping("/shelters")
    public ResponseEntity<ShelterView> create(@Valid @RequestBody
ShelterForm form) {
        ShelterView created = service.create(form);
        URI location = ServletUriComponentsBuilder.fromCurrentRequest()
                .path("/{id}").buildAndExpand(created.id()).toUri();
        return ResponseEntity.created(location).body(created);
// 201 + Location
    }

    @PutMapping("/shelters/{id}")
    public ShelterView update(@PathVariable int id, @Valid @RequestBody
ShelterForm form) {
        return service.update(id, form);
    }

    @DeleteMapping("/shelters/{id}")
    public ResponseEntity<Void> delete(@PathVariable int id) {
        service.delete(id);
        return ResponseEntity.noContent().build();
// 204
    }
}
```

Лістинг 7 – CRUD-операції REST API укриттів (тег lab-07, скорочено)

```
POST /api/shelters HTTP/1.1
Content-Type: application/json

{"name":"Укриття № 7","address":"вул. Зелена, 109","districtId":4,
 "type":"BASEMENT","capacity":150,"accessible":true}

HTTP/1.1 201 Created
Location: http://localhost:8080/api/shelters/7
Content-Type: application/json

{"id":7,"name":"Укриття № 7","address":"вул. Зелена,
109","district":"Сихівський",
 "type":"Підвал","capacity":150,"accessibility":"Так","status":"Не
готове","statusCode":"NOT_READY"}
```

Лістинг 8 – Створення ресурсу: запит і відповідь

Зверніть увагу на два різні об’єкти: **вхідний** DTO (`ShelterForm`) містить лише поля, які клієнт має право задати (без `id` і стану), а **вихідний** (`ShelterView`) – дані для показу. Приймати і віддавати безпосередньо сутності моделі небезпечно: клієнт міг би змінити поле, яке йому не належить (вразливість *mass assignment*), а внутрішня структура БД «просочилася» б в API. Той самий `ShelterForm` використовується і HTML-формою, і JSON-запитом, тому правила валідації спільні. Для перевірки API використовують Postman, curl або HTTP Client в IntelliJ IDEA.

7. Валідація вхідних даних засобами Bean Validation

Дані від клієнта завжди вважаються ненадійними. **Jakarta Bean Validation** дозволяє описати обмеження анотаціями на полях класу, а Spring перевіряє їх автоматично, якщо параметр контролера позначено `@Valid` (стартер `spring-boot-starter-validation`). Повідомлення задається атрибутом `message`.

Анотація	**Перевірка**
`@NotNull`	значення не `null`
`@NotBlank`	рядок не `null` і містить не лише пробіли
`@NotEmpty`	рядок або колекція не порожні
`@Size(min, max)`	довжина рядка, розмір колекції
`@Min`, `@Max`, `@Positive`	межі числа
`@Email`, `@Pattern(regexp)`	формат адреси, регулярний вираз
`@Past`, `@Future`	дата в минулому / майбутньому
`@Valid` на полі	каскадна перевірка вкладеного об’єкта

Таблиця 6 – Основні анотації Bean Validation

```
public class ShelterForm {
    @NotBlank(message = "Вкажіть назву укриття")
    @Size(max = 150, message = "Назва — не довше 150 символів")
    private String name;

    @NotNull(message = "Оберіть тип укриття")
    private ShelterType type;

    @NotNull(message = "Вкажіть місткість")
    @Min(value = 1, message = "Місткість має бути не менше 1")
    @Max(value = 10_000, message = "Місткість — не більше 10 000 осіб")
    private Integer capacity;
    …
}

public record InspectionRequest(
        @NotNull(message = "Вкажіть інспектора") @Positive Integer
inspectorId,
        @NotNull(message = "Вкажіть результат перевірки")
InspectionResult result,
        @Size(max = 1000) String notes) { }
```

Лістинг 9 – Обмеження на вхідних DTO

Валідацію розподіляють на два рівні. **Перевірка формату** (поле заповнене, число в межах, рядок не задовгий) виконується анотаціями на DTO в контролері. **Бізнес-правила**, що потребують звернення до БД (адреса ще не зареєстрована, район існує), перевіряє сервіс і кидає `ValidationException` з переліком помилок за полями. Остання лінія захисту – обмеження самої БД (`NOT NULL`, `CHECK`, `UNIQUE`, зовнішні ключі): навіть за помилки в коді некоректні дані не потраплять у таблиці. Для полів-обгорток використовують `Integer`, а не `int`: інакше відсутнє в JSON поле стане нулем, і `@NotNull` не спрацює.

8. Централізована обробка помилок. Формат Problem Details

Якщо кожен метод контролера сам перехоплюватиме винятки, код стане громіздким, а формат помилок – різним. Spring дозволяє винести обробку в окремий клас з анотацією **`@RestControllerAdvice`** (для REST) або `@ControllerAdvice` (для сторінок). Його методи з `@ExceptionHandler` перехоплюють винятки з усіх контролерів (або лише з указаних в `assignableTypes`) і формують відповідь (табл. 7).

Виняток	Код	Причина
`MethodArgumentNotValidException`	400	не пройдено `@Valid` для `@RequestBody`
`ValidationException` (власний)	400	порушено бізнес-правила в сервісі
`HttpMessageNotReadableException`	400	тіло не JSON або невідоме значення переліку
`NotFoundException` (власний)	404	ресурсу немає
`DataIntegrityViolationException`	409	порушено обмеження БД (неіснуючий зовнішній ключ)

Таблиця 7 – Обробники помилок демо-проєкту (ApiExceptionHandler)

Для тіла відповіді з помилкою існує стандарт **RFC 9457 «Problem Details for HTTP APIs»**: JSON-об'єкт з полями `type`, `title`, `status`, `detail`, `instance` і довільними розширеннями. У Spring його представляє клас `ProblemDetail`, а тип вмісту відповіді – `application/problem+json`. Клієнтові не потрібно вгадувати формат помилок кожного API.

```
@RestControllerAdvice(assignableTypes = ShelterApiController.class)
public class ApiExceptionHandler {
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ProblemDetail invalid(MethodArgumentNotValidException e) {
        Map<String, String> errors = new LinkedHashMap<>();
        for (FieldError fe : e.getBindingResult().getFieldErrors()) {
            errors.putIfAbsent(fe.getField(), fe.getDefaultMessage());
        }
        ProblemDetail pd =
ProblemDetail.forStatusAndDetail(HttpStatus.BAD_REQUEST,
                                                          "Перевірте
поля запиту");
        pd.setTitle("Некоректні дані");
        pd.setProperty("errors", errors);
        return pd;
    }

    @ExceptionHandler(NotFoundException.class)
    public ProblemDetail notFound(NotFoundException e) {
        return ProblemDetail.forStatusAndDetail(HttpStatus.NOT_FOUND,
e.getMessage());
    }
}
```

Лістинг 10 – Централізований обробник помилок REST API

```
HTTP/1.1 400 Bad Request
Content-Type: application/problem+json

{"type":"about:blank","title":"Некоректні дані","status":400,
 "detail":"Перевірте поля запиту","instance":"/api/shelters",
 "errors":{"name":"Вкажіть назву укриття","capacity":"Місткість має
бути не менше 1"}}
```

Лістинг 11 – Відповідь з помилкою у форматі Problem Details

Для вебсторінок у демо-проєкті є окремий `WebExceptionHandler` з анотацією `@ControllerAdvice`: для `NotFoundException` він повертає шаблон `error/404.html` з кодом 404 замість стандартної «Whitelabel Error Page». Важливе правило безпеки: відповідь про помилку не повинна містити стек викликів, SQL-запити чи імена таблиць – ця інформація допомагає зловмиснику. Деталі пишуть у журнал сервера, а клієнт отримує зрозуміле повідомлення.

9. Документування API: OpenAPI і Swagger UI

API без документації майже неможливо використовувати: розробники клієнта мають знати маршрути, параметри, формати тіла і можливі коди відповіді. **OpenAPI Specification** (раніше Swagger Specification) – стандартний машинозчитуваний формат опису HTTP API у JSON або YAML. З нього генерують інтерактивну документацію (**Swagger UI**), клієнтські бібліотеки, тести і макети серверів.

Бібліотека **springdoc-openapi** створює опис OpenAPI автоматично, аналізуючи контролери Spring: маршрути, параметри, типи DTO і обмеження Bean Validation. Достатньо додати залежність `springdoc-openapi-starter-webmvc-ui`, і стануть доступні `/v3/api-docs` (опис у JSON) та `/swagger-ui.html` (інтерактивна сторінка, де кожен запит можна виконати кнопкою «Try it out»). Анотації `@Tag`, `@Operation`, `@ApiResponse` доповнюють опис людськими поясненнями (лістинг 12).

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>3.2.0</version>
</dependency>

@Tag(name = "Укриття", description = "Реєстр укриттів і перевірок")
public class ShelterApiController {
    @Operation(summary = "Створити укриття")
    @ApiResponse(responseCode = "201", description = "Створено; адреса
— у заголовку Location")
    @ApiResponse(responseCode = "400", description = "Некоректні дані")
    @PostMapping("/shelters")
    public ResponseEntity<ShelterView> create(@Valid @RequestBody
ShelterForm form) { … }
}
```

Лістинг 12 – Підключення springdoc-openapi і анотації документації

Підхід, за якого документація генерується з коду, називають *code-first*; протилежний підхід *API-first* передбачає, що спочатку узгоджують файл OpenAPI, а потім генерують за ним каркас контролерів. Для навчального проєкту зручніший перший. У промисловому середовищі Swagger UI часто вимикають або закривають автентифікацією, щоб не розкривати структуру API (`springdoc.swagger-ui.enabled: false` у профілі prod).

Підсумкову карту застосунку на етапі `lab-07` наведено в табл. 8: два канали доступу до тих самих даних і бізнес-правил.

Вебсторінки (ShelterWebController)	**REST API (ShelterApiController)**
`GET /shelters?districtId=5` – список, фільтр у cookie	`GET /api/shelters` – 200
`GET /shelters/{id}` – картка, історія перевірок	`GET /api/shelters/{id}` – 200, 404
`GET /shelters/new`, `POST /shelters` – створення	`POST /api/shelters` – 201 + Location, 400
`GET /shelters/{id}/edit`, `POST /shelters/{id}`	`PUT /api/shelters/{id}` – 200, 400, 404
`POST /shelters/{id}/delete`	`DELETE /api/shelters/{id}` – 204, 404
`POST /shelters/{id}/inspections`	`POST /api/shelters/{id}/inspections` – 200, 400, 404, 409

Таблиця 8 – Маршрути демо-проєкту (тег lab-07)

Чому сторінки використовують `POST /shelters/{id}/delete`, а не `DELETE`? HTML-форми підтримують лише методи GET і POST, тому для

сторінок дія записується в шлях. REST API, яким користуються програми, застосовує повний набір методів HTTP.

Висновки

Spring MVC розділяє контролери сторінок (`@Controller` + Thymeleaf) і REST API (`@RestController` + JSON), які спираються на спільний сервіс. Дані запиту отримуються анотаціями `@PathVariable`, `@RequestParam`, `@RequestHeader`, `@CookieValue`, `@ModelAttribute`, `@RequestBody`. Thymeleaf формує сторінки з природних HTML-шаблонів, екранує дані й підтримує фрагменти; `th:object`/`th:field` разом з `BindingResult` показують помилки біля полів, а `RedirectAttributes` реалізує PRG. RESTful API моделює ресурси іменниками, дії – методами HTTP, результат – кодами стану (201 + `Location`, 204, 400, 404, 409). Вхідні дані перевіряються анотаціями Bean Validation і бізнес-правилами сервісу, помилки обробляються централізовано в `@RestControllerAdvice` у форматі Problem Details, а springdoc-openapi автоматично документує API в Swagger UI.

Контрольні запитання

1. Чим `@Controller` відрізняється від `@RestController`? Що повертають їхні методи?
2. Як отримати в методі контролера змінну шляху, параметр, заголовок, cookie?
3. Що таке flash-атрибути і навіщо вони в шаблоні PRG?
4. Що таке «природні шаблони» Thymeleaf? Чим `th:text` відрізняється від `th:utext`?
5. Для чого потрібні вирази `${}`, `*{}`, `@{}` і фрагменти?
6. Як працюють `th:object`, `th:field` і `BindingResult`?
7. Назвіть обмеження архітектурного стилю REST. Що таке ресурс і представлення?
8. Які методи HTTP є безпечними та ідемпотентними? Чому це важливо?
9. Які коди стану повертає API під час створення і видалення ресурсу? Навіщо заголовок `Location`?
10. Навіщо розділяти вхідні і вихідні DTO? Що таке mass assignment?
11. Як розподілити валідацію між DTO, сервісом і БД?
12. Як працює `@RestControllerAdvice`? Які поля містить Problem Details?

13. Що таке OpenAPI і Swagger UI? Чим code-first відрізняється від API-first?

Завдання для самостійної роботи

Опрацювати матеріал лекції. Задокументувати REST API власного вебсервісу засобами OpenAPI/Swagger. Реалізувати функціональність вебсервісу курсової роботи: сторінки на Thymeleaf і CRUD API з валідацією. Ознайомитися з RFC 9457 і посібниками «Serving Web Content with Spring MVC», «Validating Form Input» на spring.io/guides. Пройти тест № 7 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 № 7 і курсовою роботою. 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.