Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

ЛАБОРАТОРНА РОБОТА №9

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Взаємодія компонентів системи»

Тема роботи: 7. Редактор зображень

Виконав
студент групи ІА–33
Марченко Вадим Олександрович

Київ 2025

Тема: Взаємодія компонентів системи

Мета: ивчити види взаємодії додатків (Client-Server, Peer-to-Peer, Serviceoriented Architecture), та реалізувати в проєктованій системі одну із архітектур.

Посилання на репозиторій: https://github.com/nwu1015/ImageEditor

Короткі теоретичні відомості

Client-Server

Шаблон "Client-Server" визначає взаємодію між двома типами компонентів: клієнтом, який робить запити, і сервером, який їх обробляє та повертає результати. Використовується для побудови розподілених систем, де клієнт взаємодіє з сервером через мережу. У Java цей шаблон реалізується за допомогою серверного додатка (наприклад, з використанням Java Servlet API чи Spring Boot) і клієнтської програми (через HTTP-клієнти, такі як Apache HttpClient чи RESTTemplate). Сервер обробляє запити клієнта, забезпечуючи централізоване управління даними, а клієнт отримує доступ до функціоналу через чітко визначений API

Peer-to-Peer (P2P)

Шаблон "Peer-to-Peer" забезпечує децентралізовану модель, де кожен вузол може діяти як клієнт і як сервер. Використовується для створення мереж, де учасники обмінюються ресурсами без центрального вузла. У Java цей шаблон реалізується через мережеве програмування (наприклад, з використанням java.net для сокетів або бібліотек для P2P мереж, таких як JXTA). Кожен вузол має механізми для з'єднання з іншими вузлами, надсилання запитів і відповіді на них. Завдяки цьому система стає стійкою до відмови окремих вузлів і легко масштабується.

Service-Oriented Architecture (SOA)

Шаблон "Service-Oriented Architecture" визначає побудову системи з незалежних сервісів, які взаємодіють через стандартизовані протоколи. Використовується для проектування модульних систем, де кожен сервіс відповідає за певну бізнес-логіку. У Java цей шаблон реалізується через технології, такі як SOAP (з використанням JAX-WS) або RESTful сервіси

(з використанням Spring REST). Сервіси спілкуються через API, що дозволяє інтегрувати їх незалежно від платформи або мови програмування. Такий підхід забезпечує повторне використання компонентів і спрощує їх модифікацію чи заміну

Хід роботи

7. **Редактор зображень** (state, prototype, memento, facade, composite, client-server)

Редактор зображень має такі функціональні можливості: відкриття/збереження зображень у найпопулярніших форматах (5 на вибір студента), застосування ефектів, наприклад поворот, розтягування, стиснення, кадрування зображення, можливість створення колажів шляхом «нашарування» зображень.

У цій роботі я використовую архітектурний патерн "Клієнт-Сервер" (Client-Server). Цей патерн було обрано як фундаментальне архітектурне рішення, оскільки проєкт є веб-додатком, що вимагає чіткого розділення відповідальності між логікою представлення та логікою обробки даних.

У процесі роботи над проєктом стало зрозуміло, що система має два кардинально різні набори завдань: по-перше, ресурсоємні та чутливі до безпеки операції (обробка зображень, транзакції з базою даних PostgreSQL, збереження файлів на диску), а по-друге, логіка представлення та безпосередньої взаємодії з користувачем (UI).

Без чіткого розділення "Клієнт-Сервер", вся ця складна логіка мала б виконуватися на стороні клієнта (тобто в браузері). Це призвело б до катастрофічних наслідків: по-перше, до вкрай низької продуктивності, оскільки ресурсоємна обробка графіки (BufferedImage, ImageIO) виконувалася б на машині користувача; по-друге, до неможливості централізованого зберігання даних (колажі були б прив'язані до одного

комп'ютера); і по-третє, до повної відсутності безпеки, оскільки логіка доступу до бази даних була б відкритою.

Архітектура "Клієнт-Сервер" вирішує цю проблему елегантно. Вона чітко розділяє відповідальність:

- Серверна частина (Server) у моєму проєкті реалізована за допомогою Spring Boot. Вона виконує всю "важку" та захищену роботу: CollageService та ImageService керують бізнес-логікою (включно з реалізованими патернами), JPA взаємодіє з базою даних, а ImageService інкапсулює складну обробку графіки.

- Клієнтська частина (Client) – це веб-браузер користувача. Він відповідає лише за відображення HTML-сторінок (згенерованих за допомогою Thymeleaf) та відправку простих HTTP-запитів (через форми) на сервер.

- Зв'язок між ними забезпечується протоколом HTTP, а Spring MVC @Controller виступає як "middleware", приймаючи прості запити від клієнта (наприклад, "клонувати шар") і перетворюючи їх на виклики складних операцій на сервері.

Реалізація взаємодії розподілених систем

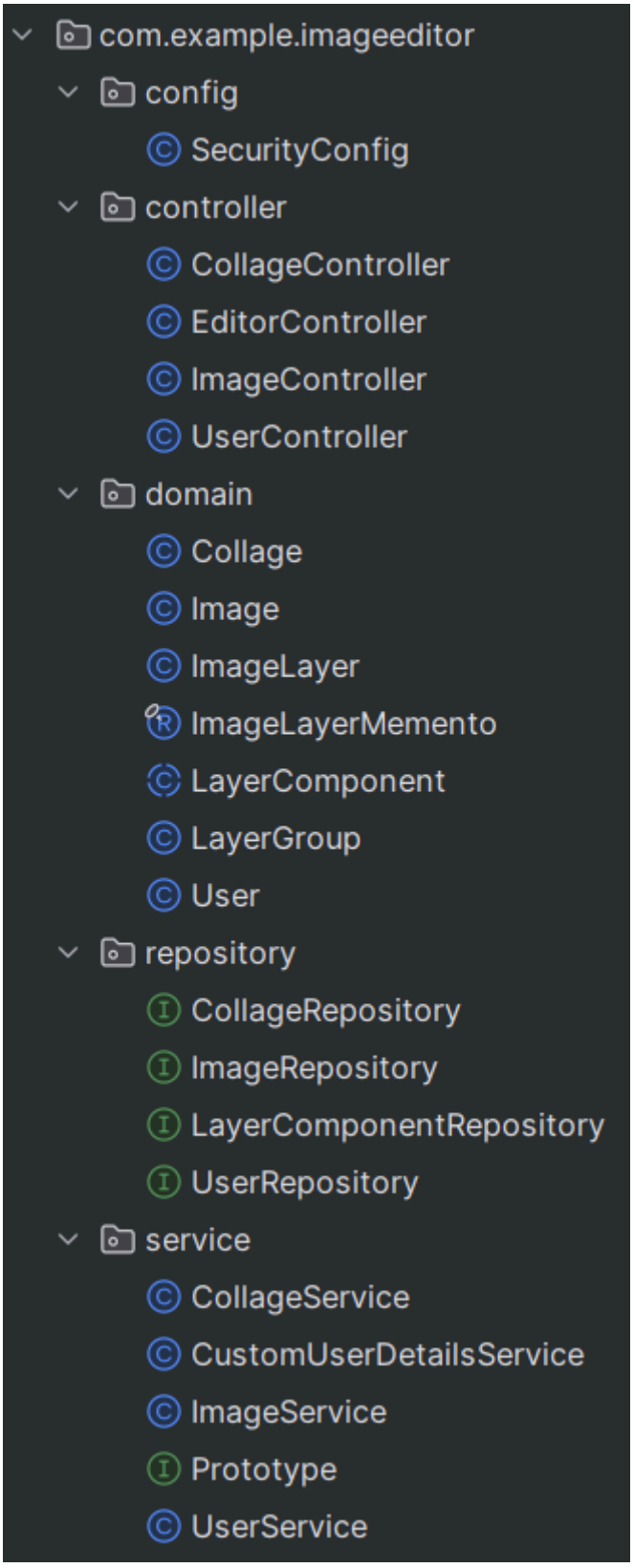
У цій роботі було реалізовано архітектуру "Клієнт-Сервер". Вимоги до завдання вказували на можливі технології зв'язку, такі як WCF, TcpClient або .NET-Remoting. Оскільки даний проєкт розроблено на платформі Java з використанням фреймворку Spring Boot, вищезгадані технології, що є специфічними для стеку Microsoft .NET (і найчастіше використовуються з мовою C#), не могли бути застосовані через фундаментальну несумісність платформ.

Для реалізації клієнт-серверної взаємодії було обрано стандартні та сучасні технології з екосистеми Java, які виконують аналогічні функції:

- Роль TcpClient (низькорівневе управління з'єднанням) виконує вбудований веб-сервер Tomcat, який автоматично запускається разом із Spring Boot. Він керує всіма TCP/IP-з'єднаннями та обробляє життєвий цикл протоколу HTTP.

- Роль WCF / .NET-Remoting (фреймворк для зв'язку та "загальна частина") виконує Spring MVC. "Клієнтом" виступає веб-браузер користувача, а "сервером" – сам додаток Spring Boot. Зв'язок відбувається за протоколом HTTP. Класи-контролери (@Controller) приймають HTTP-запити (POST-запити з HTML-форм), перетворюють їх на виклики методів сервісного шару (@Service) і повертають клієнту згенеровані Thymeleaf HTML-сторінки.

Структура програмного застосунку


com.example.imageeditor
config
SecurityConfig
controller
CollageController
EditorController
ImageController
UserController
domain
Collage
Image
ImageLayer
ImageLayerMemento
LayerComponent
LayerGroup
User
repository
CollageRepository
ImageRepository
LayerComponentRepository
UserRepository
service
CollageService
CustomUserDetailsService
ImageService
Prototype
UserService

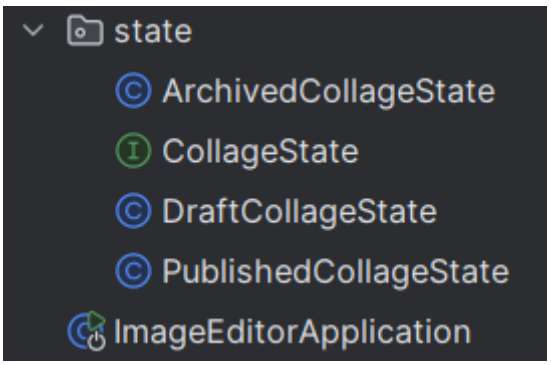


Рисунок 1. - Структура проєкту сервера

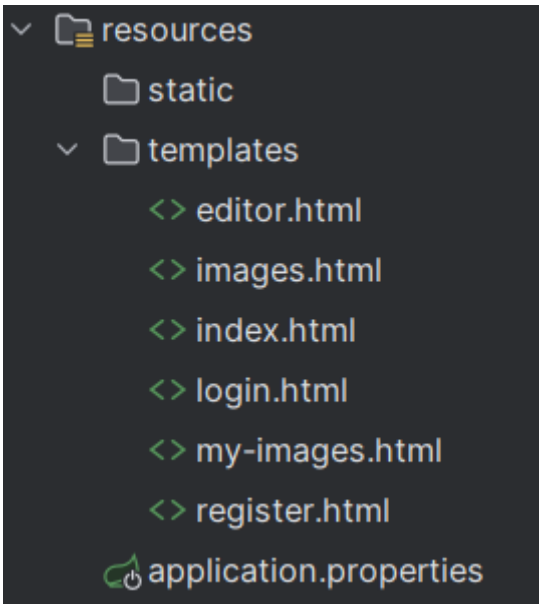


Рисунок 2. - Структура проєкту клієнта

Опис архітектури проєкту

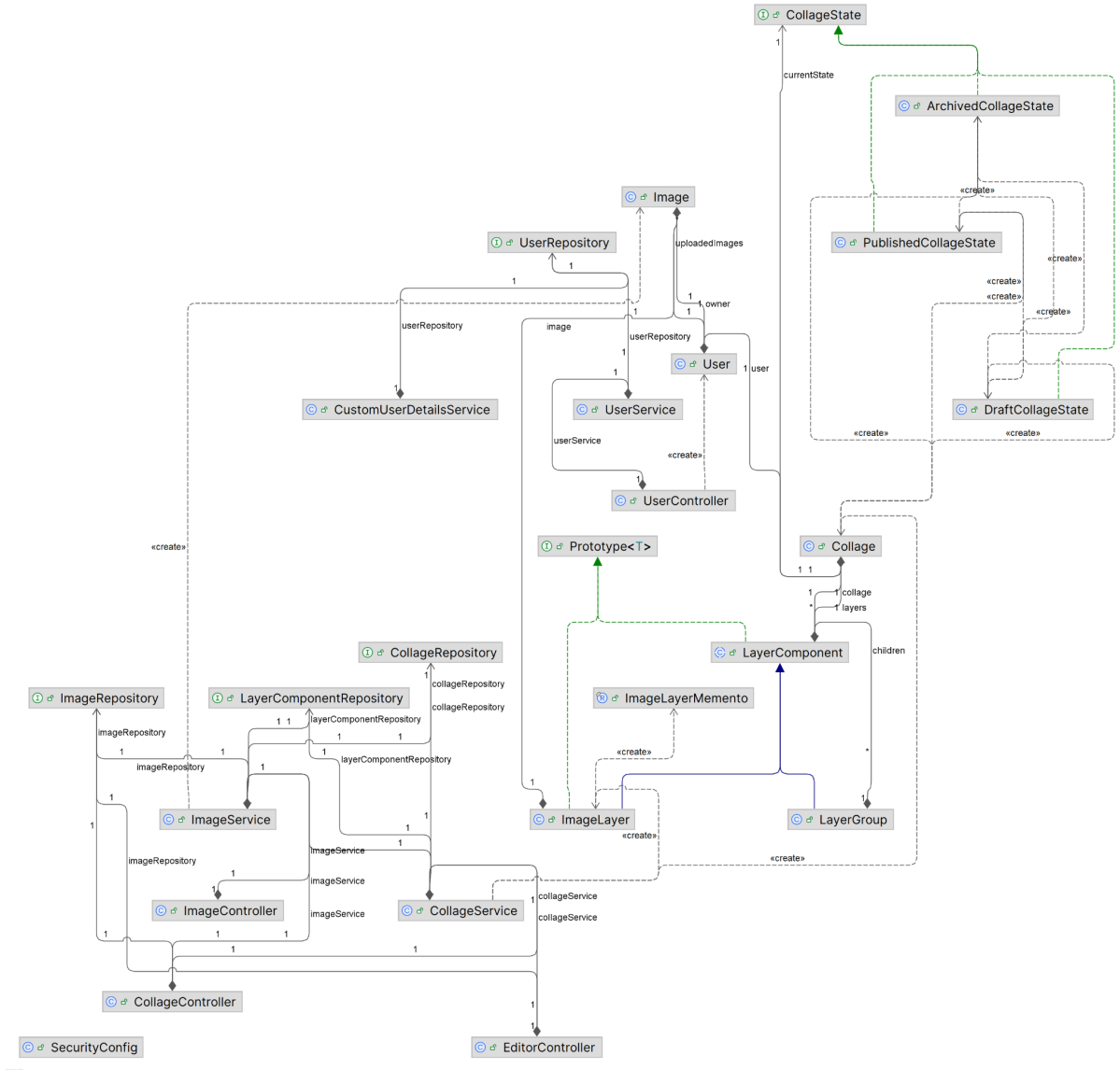


Рисунок 3. - Діаграма класів програмного застосунку

Клієнтська Частина (Client)

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org" xmlns:sec="http://www.thymeleaf.org/extras/spring-security">
<head>
    <meta charset="UTF-8">
    <title th:text="'Редактор колажу: ' + ${collage.name}">Редактор колажу</title>
    <style...>
</head>
<body>

<div class="container">
    <h1 th:text="'Редактор: ' + ${collage.name}">Редактор: Назва колажу</h1>

    <div class="render-section">
        <h2>Завершення роботи</h2>
        <p>Коли ви закінчите редагування, ви можете зберегти всі шари в одне фінальне зображення.</p>
        <form th:action="@{'/collages/' + ${collage.id} + '/render'}" method="post"...>
    </div>

    <div class="upload-section">
        <h2>Додати нове зображення</h2>
        <form th:action="@{'/collages/' + ${collage.id} + '/layers/add'}" method="post" enctype="multipart/f
    </div>

    <div class="layers-section">
        <h2>Шари колажу</h2>

        <div th:each="layer : ${collage.layers}" class="layer">

            <div th:if="${layer instanceof T(com.example.imageeditor.domain.ImageLayer)}">
                <div th:with="imgLayer=${layer}"...> </div> <div th:if="${layer instanceof T(com.example.ima
            <div class="layer-group-info"...>
        </div>
```

```
        </div>

            <div class="layer-controls">
                <div class="control-group">
                    <h4>Загальні дії (для Шару або Групи)</h4>
                    <div><strong>ID:</strong> <span th:text="${layer.id}"></span></div>
                    <div><strong>Загальний поворот:</strong> <span th:text="${layer.rotationAngle}">0</span

                    <form th:action="@{'/collages/' + ${collage.id} + '/layers/' + ${layer.id} + '/action'}
                    <form th:action="@{'/collages/' + ${collage.id} + '/layers/' + ${layer.id} + '/action'}
                    <form th:action="@{'/collages/' + ${collage.id} + '/layers/' + ${layer.id} + '/action'}
                    <form th:action="@{'/collages/' + ${collage.id} + '/layers/' + ${layer.id} + '/clone'}"
                </div>
            </div>

        </div> <div th:if="${collage.layers.isEmpty()}">
        <p>У цьому колажі ще немає жодного шару. Завантажте зображення, щоб почати.</p>
    </div>
    </div>
</div>

</body>
</html>
```

Рисунок 4. - Частина з реалізації клієнтської сторони. editor.html

Цей файл є повноцінним прикладом клієнтської частини архітектури. Хоча він динамічно генерується на сервері за допомогою шаблонізатора

Thymeleaf, кінцевий HTML-код виконується та відображається у веб-браузері користувача.

Цей файл відповідає виключно за представлення (UI) та взаємодію з користувачем. Він не містить жодної складної бізнес-логіки; він не знає, як саме потрібно повертати шар, як застосовується кадрування, як працює патерн Memento (Undo/Redo) чи що таке база даних.

Його єдина відповідальність – надати користувачу інтерфейс (кнопки "Повернути", "Клонувати", форми для зміни розміру) та ініціювати зв'язок із сервером. Коли користувач натискає кнопку, відповідна HTML-форма (<form>) відправляє HTTP POST-запит (через th:action) на вказану URL-адресу, передаючи серверу команди та дані.

Серверна Частина (Server)

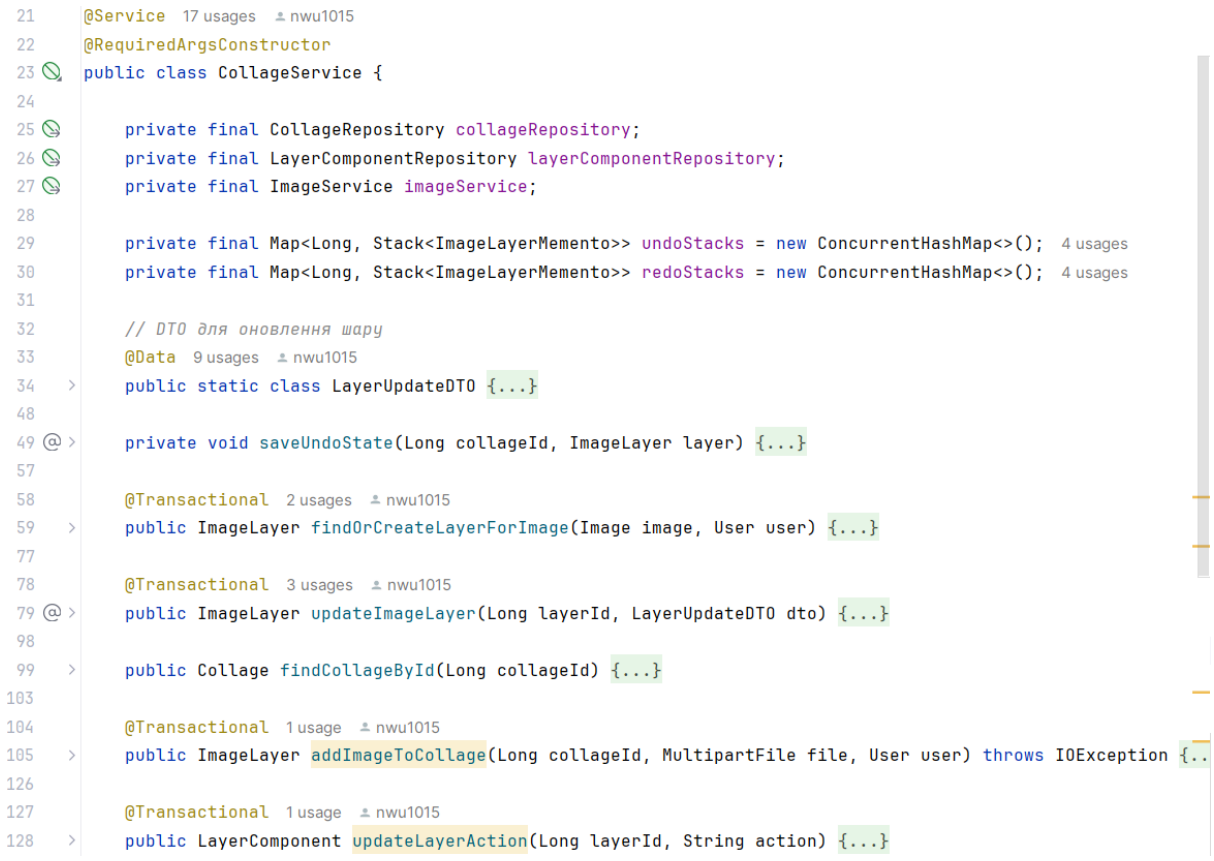


Рисунок 5. - Частина з реалізації серверної сторони. CollageService.java

Клас CollageService виступає як "мозок" програми. Він безпосередньо реалізує складні бізнес-правила та керує станом додатку:

- Він керує життєвим циклом об'єктів, реалізуючи патерн State (через collage.getCurrentState().checkCanEdit(...)).

- Він виступає "Опікуном" для патерну Memento (через saveUndoState(...) та undo()/redo()).

- Він є "Клієнтом" для патернів Composite (component.render(...)) та Prototype (prototypeComponent.clone()).

- Він виконує транзакційні (@Transactional) операції з базою даних, забезпечуючи цілісність даних, що є виключно серверною задачею.

Middleware

```java
@Controller  nwu1015
@RequestMapping("/collages")
@RequiredArgsConstructor
public class CollageController {

    private final CollageService collageService;

    private final ImageRepository imageRepository;

    private final ImageService imageService;

    /**
     * Transition method: finds the collage by image ID and redirects to the main editor.
     */
    @GetMapping("/from-image/{imageId}")  nwu1015
    public String editCollageFromImage(@PathVariable Long imageId, @AuthenticationPrincipal User user) {...}

    /**
     * The main method that displays the editor page for a specific collage.
     */
    @GetMapping("/{collageId}")  nwu1015
    public String showEditorPage(@PathVariable Long collageId, Model model) {...}
```

```java
    @PostMapping("/{collageId}/layers/add")  nwu1015
    public String handleImageUpload(@PathVariable Long collageId,
                                    @RequestParam("imageFile") MultipartFile file,
                                    @AuthenticationPrincipal User user,
                                    RedirectAttributes redirectAttributes) {...}

    /**
     * Processes actions on an existing layer (rotate, delete, etc.).
     */
    @PostMapping("/{collageId}/layers/{layerId}/action")  nwu1015
    public String handleLayerAction(@PathVariable Long collageId,
                                    @PathVariable Long layerId,
                                    @RequestParam String action) {...}

    @GetMapping("/layers/{layerId}/transformed")  nwu1015
    @ResponseBody
    public ResponseEntity<Resource> getTransformedLayer(@PathVariable Long layerId) {...}

    @PostMapping("/{collageId}/render")  nwu1015
    public String renderCollage(@PathVariable Long collageId, @AuthenticationPrincipal User user,
                                RedirectAttributes redirectAttributes) {
        try {...} catch (IOException e) {
            e.printStackTrace();
            redirectAttributes.addFlashAttribute( attributeName: "errorMessage",
                    attributeValue: "Не вдалося зберегти колаж.");
            return "redirect:/collages/" + collageId;
        }
    }
```

Рисунок 6. - Частина з реалізації контролера програми.

CollageController.java

CollageController, є ключовим прикладом "загальної частини" (Middleware), або шару зв'язку, в архітектурі "Клієнт-Сервер". Він діє як "міст" або "перекладач" між клієнтською частиною (HTML-сторінками, що виконуються у браузері) та серверною частиною (класами @Service, що виконують бізнес-логіку).

Використовуючи анотації Spring MVC, такі як @PostMapping, @GetMapping, цей клас "слухає" вхідні HTTP-запити від клієнта на конкретних URL-адресах.

Наприклад, коли користувач натискає кнопку "Повернути праворуч" на editor.html (Клієнт), форма надсилає HTTP POST-запит. Метод handleLayerAction у контролері перехоплює цей запит. Він розбирає URL (@PathVariable) та дані форми (@RequestParam), а потім перетворює цей простий запит на конкретний виклик методу Java на серверній частині – collageService.updateLayerAction(...).

Після того, як сервер (CollageService) виконав всю складну роботу, контролер формує HTTP-відповідь (у даному випадку return "redirect:..."), яка наказує клієнту (браузеру) оновити сторінку, тим самим завершуючи цикл "запит-відповідь".

Висновок: Виконуючи цю лабораторну роботу, я ознайомився з такими архітектурними підходами, як «Client-Server», «Peer-to-Peer» та «Service-oriented Architecture».

Особливу увагу я приділив архітектурі "Клієнт-Сервер" (Client-Server), оскільки вона є фундаментальною для розробки сучасних веб-додатків, подібних до мого проєкту. Ця архітектура була обрана для чіткого розділення відповідальності між логікою представлення даних та логікою їх обробки.

Під час вивчення цієї архітектури я зрозумів, наскільки елегантно вона вирішує проблему розподілу завдань у системі. Вона чітко розмежовує ролі: Клієнт (Client), у моєму випадку – веб-браузер, відповідає виключно за представлення інтерфейсу користувача та ініціювання запитів. У свою чергу Сервер (Server), реалізований на Spring Boot, інкапсулює всю складну бізнес-логіку, керування станами, транзакції та взаємодію з базою даних.

Завдяки цьому досвіду я краще зрозумів переваги патерну "Клієнт-Сервер" у контексті побудови масштабованих та безпечних додатків. Я усвідомив його ключову роль у централізації бізнес-логіки та даних: це не тільки захищає чутливі операції від прямого доступу, але й значно спрощує підтримку та оновлення системи, оскільки вся логіка знаходиться в одному місці (на сервері).

Контрольні запитання

1. Що таке клієнт-серверна архітектура?

Клієнт-серверна архітектура – це модель взаємодії, у якій програма поділена на два основні компоненти: клієнт і сервер.

Клієнт відповідає за запити та відображення даних користувачу, а сервер – за обробку цих запитів, збереження інформації та виконання логіки. Клієнт надсилає запит, сервер його опрацьовує і повертає результат. Такий підхід дозволяє розділити відповідальності, спростити масштабування й забезпечити зручне обслуговування системи.

2. Розкажіть про сервіс-орієнтовану архітектуру.

Сервіс-орієнтована архітектура (SOA) – це підхід, у якому система складається з окремих незалежних сервісів. Кожен сервіс виконує чітко визначену функцію, має власний інтерфейс і може взаємодіяти з іншими сервісами через стандартизовані протоколи.

Сервіси не залежать від конкретної мови програмування чи платформи, тому їх легко комбінувати, оновлювати та повторно використовувати. Така архітектура спрощує масштабування системи, підвищує гнучкість і дозволяє змінювати окремі частини без впливу на всю систему.

3. Якими принципами керується SOA?

Основні принципи, якими керується сервіс-орієнтована архітектура (SOA):

- Слабке зв’язування. Сервіси мінімально залежать один від одного, що дозволяє їх змінювати окремо.

- Повторне використання. Сервіси створюються так, щоб їх можна було застосовувати в різних системах і сценаріях.

- Чіткі контракти. Кожен сервіс має визначений інтерфейс (контракт), через який клієнти з ним взаємодіють.

- Автономність. Сервіси самостійні: вони управляють власними даними та логікою.

- Незалежність технологій. Сервіси можуть бути реалізовані на різних мовах і працювати на різних платформах.

4. Як між собою взаємодіють сервіси в SOA?

У SOA сервіси взаємодіють між собою через чітко визначені інтерфейси, використовуючи стандартні протоколи обміну повідомленнями. Зазвичай це відбувається так: 1) один сервіс надсилає запит іншому сервісу; 2) інший сервіс обробляє запит і повертає відповідь у стандартизованому форматі (наприклад, XML або JSON); 3) зв’язок відбувається через мережу, здебільшого за допомогою HTTP, SOAP або REST.

5. Як розробники взнають про існуючі сервіси і як робити до них запити?

У SOA розробники дізнаються про існуючі сервіси через реєстр (каталог) сервісів. Це спеціальне місце, де зберігається опис кожного сервісу: його назва, призначення, інтерфейси, формати даних і спосіб виклику. Такий каталог дозволяє швидко знайти потрібний сервіс і зрозуміти, як з ним працювати.

Щоб виконати запит до сервісу, розробник використовує його контракт – опис доступних операцій і параметрів. На основі цього контракту формується стандартний запит (наприклад, HTTP, SOAP або REST), який сервіс приймає й обробляє. Таким чином, маючи документацію або запис у каталозі сервісів, розробник легко може підключитися та використовувати будь-який сервіс у системі.

6. У чому полягають переваги та недоліки клієнт-серверної моделі?

Переваги клієнт-серверної моделі:

- Централізоване зберігання даних. Сервер контролює доступ, безпеку та оновлення інформації.

- Легка підтримка. Оновлення робляться на сервері, а клієнти отримують актуальні дані без змін у своїх програмах.

- Масштабованість. Можна підсилювати сервер або додавати нові сервери під навантаження.

- Кращий контроль безпеки. Дані та логіка зберігаються в одному місці.

Недоліки клієнт-серверної моделі:

- Залежність від сервера. Якщо сервер виходить з ладу, система перестає працювати для всіх клієнтів.

- Велике навантаження на сервер. Усі запити надходять до одного місця, що може вимагати потужного обладнання.

7. У чому полягають переваги та недоліки однорангової моделі взаємодії?

Переваги однорангової (peer-to-peer, P2P) моделі:

- Відсутність центрального сервера. Кожен вузол одночасно є клієнтом і сервером, що зменшує залежність від одного центру.

- Хороша масштабованість. Чим більше вузлів, тим більше ресурсів (обчислювальних і мережевих) у мережі.

- Висока стійкість. Вихід окремих вузлів не зупиняє роботу всієї системи.

- Ефективність у розподілі навантаження. Дані та операції розподілені між учасниками.

Недоліки однорангової моделі:

- Складність безпеки. Немає єдиного центру контролю, тому важче забезпечити захист і перевірку даних.

- Нестабільність вузлів. Користувачі можуть довільно підключатися й відключатися, що ускладнює стабільність роботи.

- Обмежені ресурси. Вузли часто мають різні можливості, і слабкі пристрої можуть стримувати швидкість передачі.

8. Що таке мікро-сервісна архітектура?

Мікросервісна архітектура – це підхід, у якому система складається з великої кількості дрібних, незалежних сервісів. Кожен мікросервіс відповідає за одну конкретну функцію, має власну логіку, власні дані та може оновлюватися окремо від інших.

Сервіси спілкуються між собою через легкі мережеві протоколи (наприклад, HTTP або повідомлення). Такий підхід спрощує масштабування, дозволяє швидко оновлювати частини системи й робить її більш гнучкою та стійкою до відмов.

9. Які протоколи використовуються для обміну даними в мікросервісній архітектурі?

У мікросервісній архітектурі найчастіше використовують такі протоколи обміну даними: HTTP/HTTPS, AMQP (RabbitMQ) (Протокол для асинхронного обміну повідомленнями через брокера черг), Kafka Protocol (Використовується в Apache Kafka для стрімінгової передачі подій.), WebSockets (Для двостороннього постійного з’єднання, якщо потрібна швидка реакція або push-повідомлення.), MQTT (Легковаговий протокол для IoT-пристроїв та мікросервісів з малим ресурсом)

10. Чи можна назвати підхід сервіс-орієнтованою архітектурою, коли ми в проєкті між шаром веб-контролерів та шаром доступу до даних реалізуємо шар бізнес-логіки у вигляді сервісів?

Ні, це не можна назвати сервіс-орієнтованою архітектурою. Якщо між веб-контролерами та доступом до даних створено шар бізнес-логіки у

вигляді сервісів, то це просто шарова (трирівнева) архітектура одного застосунку.

У цьому випадку сервіси – це частини одного проєкту, вони не працюють як окремі автономні системи і не взаємодіють через мережеві протоколи. SOA ж передбачає, що сервіси є незалежними, можуть розгортатися окремо, спілкуватися між собою через стандартизовані інтерфейси і використовуватися різними клієнтами.