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 розробки програмного забезпечення»

Взаємодія компонентів системи

Виконав:
студент групи ІА-34
Швець Роман Вадимович

Перевірив:
асистент кафедри ІСТ
Мягкий Михайло Юрійович

Київ - 2025

Тема: Взаємодія компонентів системи

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

Теоретичні відомості

4. **Графічний редактор** (proxy, prototype, decorator, bridge, flyweight, SOA) Графічний редактор повинен вміти створювати / редагувати растрові (або векторні на розсуд студента) зображення в 2-3 основних популярних форматах (bmp, png, jpg), мати панель інструментів для створення графічних примітивів, вибору кольорів, нанесення тексту, додавання найпростіших візуальних ефектів (ч/б растр, інфрачервоний растр, 2-3 на вибір учня), роботи з шарами.

Клієнт-серверна архітектура є базовим варіантом розподілених систем, де чітко розмежовуються ролі клієнтів, що представляють інтерфейс користувача, та серверів, які відповідають за обробку і зберігання даних. В залежності від розподілу навантаження розрізняють «тонкі» клієнти, де основна логіка виконується на сервері (наприклад, веб-застосунки), та «товсті» клієнти, які беруть на себе більшість обчислень, розвантажуючи сервер (наприклад, мобільні додатки або ігри). Така взаємодія зазвичай реалізується через трирівневу структуру, що включає клієнтську частину, проміжний шар (middleware) для загальних даних та серверну частину з бізнес-логікою.

Однорангова (Peer-to-Peer) архітектура представляє собою децентралізовану модель, в якій відсутній головний сервер, а кожен вузол мережі виступає одночасно і клієнтом, і сервером, маючи рівні права на обмін даними. Головними принципами такої побудови є децентралізація, що підвищує стійкість системи до збоїв, рівноправність учасників та безпосередній розподіл ресурсів, таких як обчислювальна потужність чи дисковий простір, між вузлами без посередників. Незважаючи на ефективність у таких сферах, як файлообмінники або блокчейн, ця архітектура має слабкі місця у безпеці, синхронізації та складності контролю даних через відсутність єдиного центру.

Сервіс-орієнтована архітектура (SOA) базується на модульному підході, де

система будується з розподілених, слабко пов'язаних сервісів, що взаємодіють через стандартизовані інтерфейси та протоколи. Цей підхід виник як альтернатива монолітним системам і зазвичай реалізується через веб-служби, які спілкуються за допомогою повідомлень (найчастіше SOAP або REST) без прямих інтеграцій на рівні баз даних. Сервіси в SOA можуть бути як новими бізнес-функціями, так і «обгортками» навколо застарілих систем для їх інтеграції, а для координації обміну даними часто використовується централізована шина даних (Enterprise Service Bus).

Мікросервісна архітектура є еволюційним розвитком SOA і передбачає створення серверного додатку як набору невеликих, автономних служб, кожна з яких виконується у власному процесі та взаємодіє з іншими через легковагі протоколи на кшталт HTTP або AMQP. Кожен мікросервіс реалізує специфічну бізнес-логіку в рамках обмеженого контексту, розробляється та розгортається цілком незалежно від інших, що дозволяє забезпечити високу гнучкість системи та легкість її масштабування. Така архітектура орієнтована на автоматизацію процесів розробки та дозволяє створювати складні системи з компонентів, що мають автономні життєві цикли.

Хід роботи

Для забезпечення взаємодії між розподіленими компонентами системи та захисту даних при передачі мережею використано підхід сервіс-орієнтованої архітектури (SOA). Його основна ідея полягає у розділенні функціоналу на незалежні сервіси, що взаємодіють через стандартизовані протоколи, та використанні механізмів автентифікації для розмежування доступу. У проєкті це реалізовано через серверну частину на базі контролера ImageProcessingController, який надає REST-інтерфейс для обробки зображень, та клієнтський клас-проксі RemoteImageProcessor, що інкапсулює складність мережевих запитів. Клас CryptoUtils забезпечує наскрізне симетричне шифрування даних (payload), гарантуючи їх конфіденційність, а DTO-клас AuthRequest слугує уніфікованим контейнером для передачі облікових даних користувача між клієнтом і сервером.

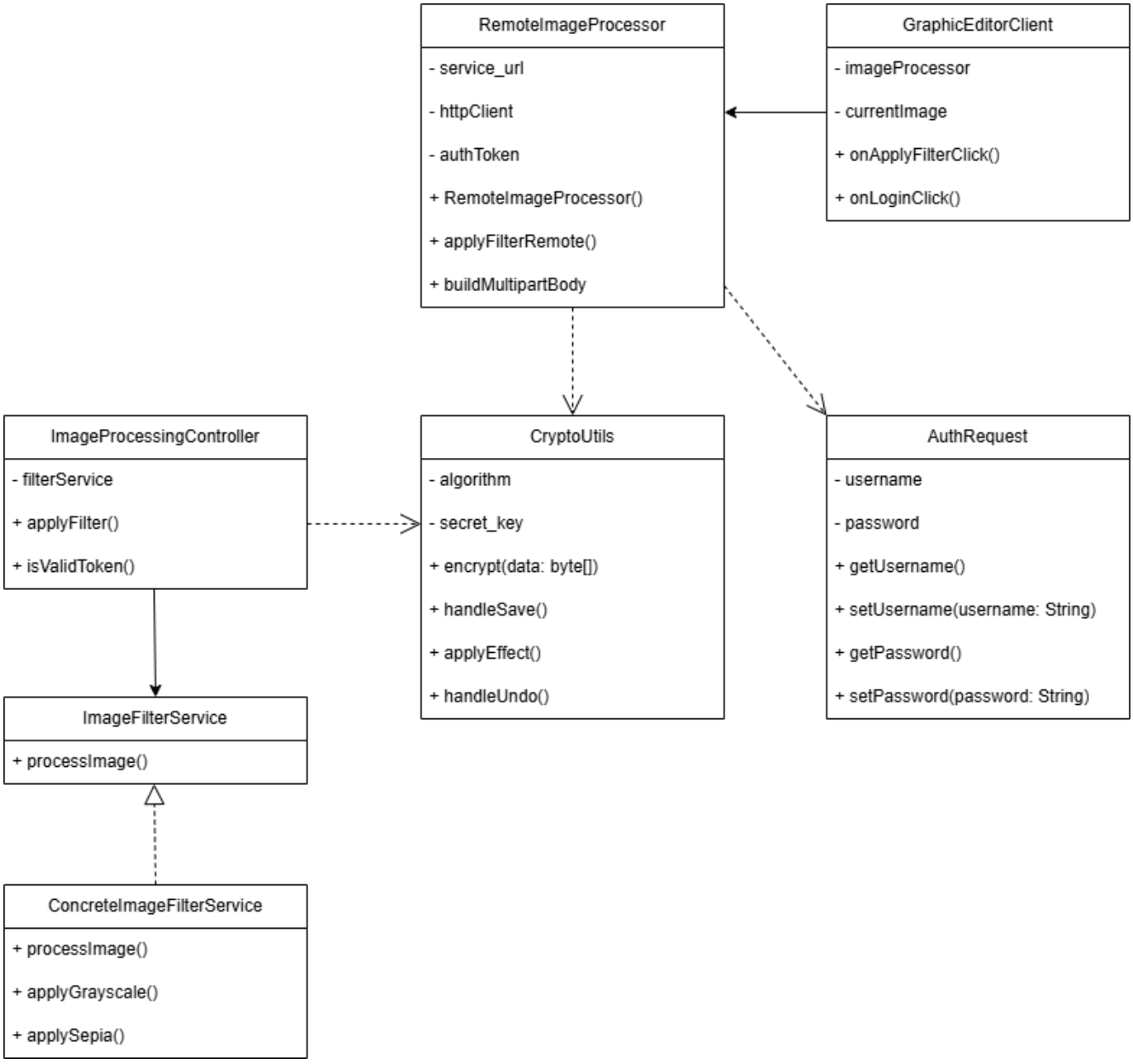

Рисунок 9.1 – Діаграма класів для SOA-архітектури

На поданій діаграмі класів відображено структурну організацію SOA у системі граф. редактора. Ключовим елементом серверної частини є ImageProcessingController, який через залежність використовує CryptoUtils для дешифрування вхідних даних та делегує бізнес-логіку сервісам фільтрації. Клас GraphicEditorClient (клієнтський контекст) пов’язаний із RemoteImageProcessor асоціацією: він не виконує ресурсоємну обробку зображень локально, а лише делегує цей процес віддаленому сервісу, передаючи токен автентифікації та зашифрований файл як аргументи запиту. Такий архітектурний підхід робить систему гнучкою та безпечною – дозволяє масштабувати серверну частину незалежно від клієнтських додатків та захищає дані від перехоплення.

Вихідні коди класів системи

Лістинг 1 – Клас CryptoUtils

```
package com.graphic.editor.utils;

import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;
import java.nio.charset.StandardCharsets;
import java.util.Base64;

public class CryptoUtils {
    private static final String ALGORITHM = "AES";
    // У навчальних цілях ключ хардкодимо. У продакшені це має бути змінна
оточення.
    private static final String SECRET_KEY = "MySuperSecretKey";

    public static byte[] encrypt(byte[] data) throws Exception {
        SecretKeySpec keySpec = new
SecretKeySpec(SECRET_KEY.getBytes(StandardCharsets.UTF_8), ALGORITHM);
        Cipher cipher = Cipher.getInstance(ALGORITHM);
        cipher.init(Cipher.ENCRYPT_MODE, keySpec);
        return cipher.doFinal(data);
    }

    public static byte[] decrypt(byte[] encryptedData) throws Exception {
        SecretKeySpec keySpec = new
SecretKeySpec(SECRET_KEY.getBytes(StandardCharsets.UTF_8), ALGORITHM);
        Cipher cipher = Cipher.getInstance(ALGORITHM);
        cipher.init(Cipher.DECRYPT_MODE, keySpec);
        return cipher.doFinal(encryptedData);
    }
}
```

Лістинг 2 – клас AuthRequest

```
package com.graphic.editor.dto;

import java.io.Serializable;

public class AuthRequest implements Serializable {
    private String username;
    private String password;

    public AuthRequest() {}

    public AuthRequest(String username, String password) {
        this.username = username;
        this.password = password;
    }

    public String getUsername() { return username; }
    public void setUsername(String username) { this.username = username; }

    public String getPassword() { return password; }
    public void setPassword(String password) { this.password = password; }
}
```

Лістинг 3 – клас ImageProcessingController

```
package com.graphic.editor.service.controller;
```

```
import com.graphic.editor.utils.CryptoUtils;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;

@RestController
@RequestMapping("/api/v1/image")
public class ImageProcessingController {

    @PostMapping(value = "/apply-filter", consumes =
MediaType.MULTIPART_FORM_DATA_VALUE)
    public ResponseEntity<byte[]> applyFilter(
            @RequestHeader(HttpHeaders.AUTHORIZATION) String token,
            @RequestParam("file") MultipartFile file,
            @RequestParam("filterType") String filterType) {

        if (!isValidToken(token)) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();
        }

        try {
            byte[] encryptedContent = file.getBytes();

            byte[] originalImage = CryptoUtils.decrypt(encryptedContent);

            byte[] processedImage = originalImage;

            byte[] encryptedResponse = CryptoUtils.encrypt(processedImage);

            return ResponseEntity.ok()
                    .contentType(MediaType.APPLICATION_OCTET_STREAM)
                    .body(encryptedResponse);

        } catch (Exception e) {
            e.printStackTrace();
            return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
        }
    }

    private boolean isValidToken(String token) {
        return token != null && token.startsWith("Bearer ");
    }
}
```

Лістинг 4 – клас RemoteImageProcessor

```
package com.graphic.editor.client.proxy;

import com.graphic.editor.utils.CryptoUtils;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.file.Files;
import java.nio.file.Path;
import java.time.Duration;
import java.util.UUID;

public class RemoteImageProcessor {
```

```
    private static final String SERVICE_URL =
"http://localhost:8080/api/v1/image/apply-filter";
    private final HttpClient httpClient;
    private final String authToken;

    public RemoteImageProcessor(String authToken) {
        this.authToken = authToken;
        this.httpClient = HttpClient.newBuilder()
                .connectTimeout(Duration.ofSeconds(10))
                .build();
    }

    public byte[] applyFilterRemote(Path imagePath, String filterType) throws
Exception {
        byte[] fileBytes = Files.readAllBytes(imagePath);
        byte[] encryptedBytes = CryptoUtils.encrypt(fileBytes);
        String boundary = "Boundary-" + UUID.randomUUID().toString();
        byte[] body = buildMultipartBody(encryptedBytes, filterType, boundary);
        HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create(SERVICE_URL))
                .header("Authorization", "Bearer " + authToken)
                .header("Content-Type", "multipart/form-data; boundary=" +
boundary)
                .POST(HttpRequest.BodyPublishers.ofByteArray(body))
                .build();

        HttpResponse<byte[]> response = httpClient.send(request,
HttpResponse.BodyHandlers.ofByteArray());

        if (response.statusCode() == 200) {
            byte[] encryptedResponse = response.body();
            return CryptoUtils.decrypt(encryptedResponse);
        } else {
            throw new RuntimeException("Server error code: " +
response.statusCode());
        }
    }

    private byte[] buildMultipartBody(byte[] fileData, String filterType, String
boundary) throws Exception {
        var byteStream = new java.io.ByteArrayOutputStream();

        byteStream.write(("--" + boundary + "\r\n").getBytes());
        byteStream.write("Content-Disposition: form-data;
name=\"filterType\"\r\n\r\n".getBytes());
        byteStream.write((filterType + "\r\n").getBytes());

        byteStream.write(("--" + boundary + "\r\n").getBytes());
        byteStream.write(("Content-Disposition: form-data; name=\"file\";
filename=\"secure_image.dat\"\r\n").getBytes());
        byteStream.write("Content-Type: application/octet-
stream\r\n\r\n".getBytes());
        byteStream.write(fileData);
        byteStream.write("\r\n".getBytes());

        byteStream.write(("--" + boundary + "--\r\n").getBytes());

        return byteStream.toByteArray();
    }
}
```

У поданих лістингах реалізовано взаємодію компонентів системи графічного редактора на основі сервіс-орієнтованої архітектури (SOA). Клас CryptoUtils

забезпечує наскрізний рівень безпеки, реалізуючи механізми симетричного шифрування та дешифрування даних за алгоритмом AES, що є необхідним для захисту графічного контенту під час передачі мережею. Клас ImageProcessingController виступає як постачальник послуг (Service Provider), експортуючи RESTful інтерфейс для зовнішніх звернень. Він відповідає за перевірку токена авторизації в заголовках запиту, дешифрування вхідного потоку даних та ініціювання бізнес-логіки обробки зображення. На стороні клієнта клас RemoteImageProcessor виконує роль проксі-об'єкта, який інкапсулює низькорівневі деталі HTTP-комунікації та формування multipart-запитів. Така організація коду забезпечує слабку зв'язність (loose coupling) між клієнтським додатком та сервером, дозволяючи винести ресурсоємні операції обробки зображень на віддалений вузол, зберігаючи при цьому високий рівень безпеки даних завдяки двосторонньому шифруванню.

Висновок:

У ході виконання лабораторної роботи було реалізовано сервіс-орієнтовану архітектуру (SOA) у контексті системи графічного редактора. Цей архітектурний стиль дозволив вирішити проблему високого навантаження на клієнтський додаток та забезпечити безпечну взаємодію компонентів у розподіленому середовищі. У проєкті створено систему, що розділяє відповідальність між сервером (Service Provider), який виконує ресурсоємну обробку зображень, та клієнтом (Service Consumer), який лише делегує ці задачі. Клас ImageProcessingController забезпечує надання послуг через REST-інтерфейс, а клієнтський проксі-об'єкт RemoteImageProcessor інкапсулює складність мережевої комунікації, використовуючи токени для автентифікації та двостороннє шифрування даних. Такий підхід спростив масштабування системи, дозволивши розвивати серверну бізнес-логіку незалежно від клієнтського інтерфейсу та гарантувати конфіденційність даних.

Відповіді на контрольні запитання:

1. Що таке клієнт-серверна архітектура?

Це найпростіший варіант розподілених додатків, у якому виділяються два види компонентів: клієнти (представляють додаток користувачеві, інтерфейс) і

сервери (відповідають за збереження та обробку даних).

2. Розкажіть про сервіс-орієнтовану архітектуру.

Сервіс-орієнтована архітектура (SOA) – це модульний підхід до розробки ПЗ, що базується на використанні розподілених, слабко пов’язаних сервісів зі стандартизованими інтерфейсами. Вона виникла як альтернатива монолітній архітектурі.

3. Якими принципами керується SOA?

SOA керується принципами модульності, використання розподілених сервісів, слабкої пов'язаності (Loose coupling), наявності стандартизованих інтерфейсів та взаємодії за стандартизованими протоколами.

4. Як між собою взаємодіють сервіси в SOA?

Сервіси взаємодіють виключно шляхом обміну повідомленнями (зазвичай по HTTP з використанням SOAP або REST). Вони не використовують спеціальні інтеграції для прямого доступу до спільних даних (наприклад, однієї БД).

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

Сервіси реєструються на спеціальних сервісах (реєстрах), де розробники можуть їх знайти для використання. Для обміну даними та доступу часто використовується централізована шина даних (Enterprise Service Bus).

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

Переваги: Для «тонких» клієнтів – простота розгортання та оновлення (оновлюється лише сервер). Для «товстих» клієнтів – менше навантаження на сервер, можливість роботи офлайн (без доступу до сервера).

Недоліки: Для «тонких» клієнтів –майже все навантаження лягає на сервер. Для «товстих» – необхідність інсталяції та складніший процес оновлення.

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

Переваги: Децентралізація (відсутність головного сервера), стійкість до збоїв (зменшена залежність від одного вузла), рівноправність вузлів та розподіл ресурсів.

Недоліки: Проблеми з безпекою, складність контролю даних через децентралізацію, зниження ефективності пошуку ресурсів зі збільшенням кількості вузлів.

8. Що таке мікро-сервісна архітектура?

Це стиль розробки, при якому серверний додаток створюється як набір малих служб (мікросервісів). Кожен мікросервіс є компонентом із чітко визначеними межами, розгортається незалежно і реалізує специфічні можливості бізнес-логіки.

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

Використовуються такі протоколи, як HTTP/HTTPS, WebSockets або AMQP.

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, не можна. SOA передбачає використання розподілених сервісів , які часто взаємодіють через мережу (HTTP/SOAP/REST). Якщо сервіси реалізовані просто як класи всередині одного проекту (моноліту), це є лише структурним розділенням коду, а не сервіс-орієнтованою архітектурою, яка протиставляється монолітній.