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 «Теорія розробки програмного забезпечення»

Тема: «Взаємодія компонентів системи»

Виконала:

студентка групи ІА-33

Піголь Тетяна

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.

Тема проєкту:

E-mail клієнт (singleton, builder, decorator, template method, interpreter, client-server)

Поштовий клієнт повинен нагадувати функціонал поштових програм Mozilla Thunderbird, The Bat і т.д. Він повинен сприймати і коректно обробляти pop3/smtp/imap протоколи, мати функції автонастройки основних поштових провайдерів для України (gmail, ukr.net, i.ua), розділяти повідомлення на папки/категорії/важливість, зберігати чернетки незавершених повідомлень, прикріплювати і обробляти прикріплені файли.

1. **Короткі теоретичні відомості:**

Клієнт-серверна архітектура

Це мережева архітектура, в якій пристрої розділені на два типи ролей: клієнти та сервери. Клієнт, який ініціює зв'язок, надсилає запит на отримання ресурсу або послуги та сервер – потужний апарат або програма, що приймає запит, обробляє його та надсилає відповідь.

Ключова особливість цієї структури це централізація даних та управління. Вся логіка та дані зазвичай зберігаються на сервері.

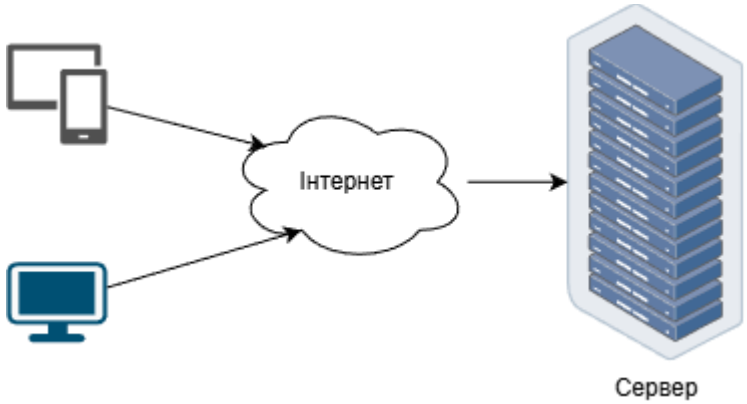


Рис. 1.1 – Клієнт-серверна архітектура

Розрізняють тонкі клієнти і товсті клієнти. Тонкий клієнт – клієнт, який повністю всі операції (або більшість, пов'язаних з логікою роботи програми) передає для обробки на сервер, а сам зберігає лише візуальне уявлення одержуваних від сервера відповідей. Грубо кажучи, тонкий клієнт – набір форм відображення і канал зв'язку з сервером. Прикладом тонкого клієнта є класичні Web-застосунки.

У такому варіанті використання майже все навантаження лягає на сервер або групу серверів.

Перевагою таких моделей є простота розгортання, тому що оновлювати потрібно лише сервери і в результаті клієнти з наступними запитами автоматично будуть працювати з оновленою системою.

Товстий клієнт – антипод тонкого клієнта, більшість логіки обробки даних містить на стороні клієнта. Це сильно розвантажує сервер. Сервер в таких випадках зазвичай працює лише як точка доступу до деякого іншого ресурсу або сполучна ланка з іншими клієнтськими комп'ютерами. Перевагою такого підходу є менші вимоги до серверної частини. Також перевагою, при певному підході до реалізації, є можливість працювати клієнтам без тимчасового доступу до серверу. Прикладом товстого клієнта можна назвати мобільні застосунки, або десктоп застосунки.

Клієнт-серверна взаємодія, як правило, організовується за допомогою 3-х рівневої структури: клієнтська частина, загальна частина, серверна частина. Оскільки велика частина даних загальна (класи, використовувані системою), їх прийнято виносити в загальну частину (middleware) системи.

Peer-to-Peer архітектура

Peer-to-Peer (P2P) архітектура – це модель мережевої взаємодії, в якій кожен вузол (комп'ютер або пристрій) є одночасно клієнтом і сервером. Кожен вузол виступає одночасно і як клієнт, так і сервер. Вузли обмінюються ресурсами (файлами, обчислювальною потужністю) безпосередньо між собою, минаючи центральний сервер. Ключова особливість такої системи є відсутність виділеного центрального

сервера. На відміну від клієнт-серверної моделі, де є чітке розділення на клієнти й сервери, P2P-мережа дозволяє учасникам взаємодіяти безпосередньо, без необхідності в централізованому сервері.

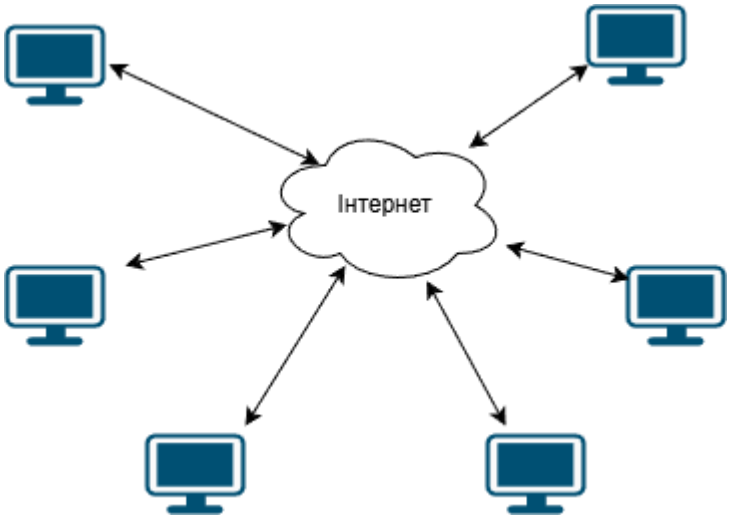


Рис. 1.1 – Peer-to-Peer архітектура

Основними принципами P2P-архітектури є:

- Децентралізація – відсутність центрального сервера, що зменшує залежність від одного вузла, підвищуючи стійкість мережі до збоїв і атак.
- Рівноправність вузлів – кожен вузол може виконувати одночасно функції клієнта (отримувати ресурси) і сервера (надавати ресурси).
- Розподіл ресурсів – вузли надають доступ до своїх власних ресурсів, таких як обчислювальна потужність, дисковий простір або файли.

Сервіс-орієнтована архітектура

Сервіс-орієнтована архітектура (SOA, англ. service-oriented architecture) – модульний підхід до розробки програмного забезпечення, заснований на використанні розподілених, слабо пов’язаних (англ. Loose coupling) сервісів або служб, оснащених стандартизованими інтерфейсами для взаємодії за стандартизованими протоколами. Архітектурний підхід спрямований на побудову корпоративних систем шляхом інтеграції розподілених, слабко пов'язаних компонентів (сервісів).

Програмні комплекси, розроблені відповідно до сервіс-орієнтованою архітектурою, зазвичай реалізуються як набір веб-служб (або веб-сервісів), які, як правило, взаємодіють по HTTP з використанням SOAP або REST. Ці служби надають певні бізнес-функції, наприклад, отримання інформації про наявність матеріалів на складі.

Сервіси взаємодіють між собою тільки за рахунок обміну повідомленнями, без створення спеціальних інтеграцій для доступу до однієї інформації, наприклад, до однієї бази даних.

Сервіси також можуть бути реалізовані як обгортки навколо застарілої системи. Це робиться для зменшення вартості переробки системи, а також спрощення інтеграції існуючих монолітних систем в нову архітектуру.

Мікросервісна архітектура є подальшим розвитком сервіс-орієнтованої архітектури з використанням нових напрацювань у інформаційних технологіях.

Мікро-сервісна архітектура

Це підхід до розробки, при якому єдина програма будується як набір невеликих сервісів, кожен з яких працює у власному процесі. Сама назва дає зрозуміти, що мікро-сервісна архітектура є підходом до створення серверного додатку як набору малих служб. Це означає, що архітектура мікро-сервісів головним чином орієнтована на серверну частину, не дивлячись на те, що цей підхід так само використовується для зовнішнього інтерфейсу, де кожна служба виконується в своєму процесі і взаємодіє з іншими службами за такими протоколами, як HTTP/HTTPS, WebSockets чи AMQP. Кожен мікросервіс реалізує специфічні можливості в предметній області і свою бізнес-логіку в рамках конкретного обмеженого контексту, повинна розроблятись автономно і розвертатись незалежно.

«Мікросервіс – це компонент із чітко визначеними межами, який можна розгортати незалежно, і підтримує взаємодію за допомогою зв’язку на основі повідомлень. Архітектура мікросервісів – це стиль розробки високоавтоматизованих систем

програмного забезпечення, що легко розвивати та яке складається з мікросервісів, орієнтованих на певні можливості».

Мікросервіси забезпечують чудові можливості супроводження в величезних комплексних системах з високою масштабуємістю за рахунок створення додатків, заснованих на множині незалежно розгортуючих служб з автономними життєвими циклами.

2. Хід роботи:

Система, реалізована як вебзастосунок, має гібридну архітектуру і виконує подвійну роль залежно від контексту взаємодії. Для кінцевого користувача (веб-браузера) додаток виступає сервером, який обробляє HTTP-запити та відображає HTML-сторінки. Для зовнішнього поштового сервісу (Google Gmail API) додаток виступає авторизованим OAuth2-клієнтом, який споживає REST API для отримання та відправки листів.

Серверна частина реалізована за допомогою патерну MVC. Клас-контролер приймає запити від браузера, взаємодіє з моделлю та повертає відображення.

У класі MailController (рис. 2.1) реалізовано обробку вхідних запитів. Анотація @Controller перетворює клас на веб-компонент, що слухає певні URL-адреси.

```
@GetMapping("/inbox")
public String viewInbox(Model model, Authentication auth) {
    model.addAttribute( attributeName: "mails", mailService.getInbox(getUsername(auth)));
    model.addAttribute( attributeName: "folderName", attributeValue: "Вхідні");
    model.addAttribute( attributeName: "folderType", attributeValue: "inbox");
    return "mailbox";
}
```

Рис. 2.1 – Частина структури файлу MailController.java

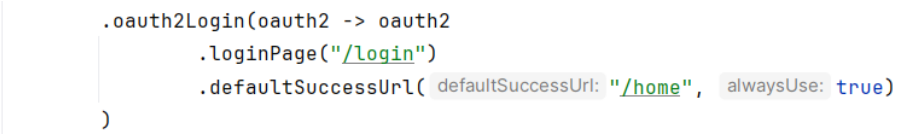
Для виконання бізнес-логіки додаток не використовує локальний SMTP-сервер, а звертається до хмарної інфраструктури Google. Це реалізовано через інтеграцію з Gmail API. Додаток діє як споживач сервісу. Клас MailService (рис. 2.2) інкапсулює логіку клієнта. Він створює об'єкт Gmail, який виконує реальні HTTP-запити до серверів Google.

```
private List<Mail> fetchMessages(String query) {
    List<Mail> mails = new ArrayList<>();
    try {
        Gmail gmail = getGmailService();
        ListMessagesResponse response = gmail.users().messages().list( userId: "me")
                .setQ(query).setMaxResults(20L).execute();

        if (response.getMessages() != null) {
            for (Message msg : response.getMessages()) {
                Message fullMsg = gmail.users().messages().get( userId: "me", msg.getId()).setFormat("full").execute();
                mails.add(convertMessageToMail(fullMsg));
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return mails;
}
```

Рис. 2.2 – Частина структури файлу MailService.java

Відповідно до вимог розподілених систем, автентифікація виконується не шляхом передачі пароля, а за допомогою токенів доступу, використовуючи протокол OAuth2.0. Налаштування SecurityConfig (рис. 2.3) делегують процес входу провайдеру Google.



```
        .oauth2Login(oauth2 -> oauth2
                .loginPage("/login")
                .defaultSuccessUrl( defaultSuccessUrl: "/home",  alwaysUse: true)
        )
```

Рис. 2.3 – Частина структури файлу SecurityConfig.java

Посилання на репозиторій: https://github.com/iatan33/EmailApp9

Контрольні запитання:

1. Що таке клієнт-серверна архітектура?

Це модель взаємодії, де навантаження розподілене між постачальниками ресурсів (серверами) та замовниками послуг (клієнтами). Клієнт надсилає запит, а сервер його обробляє та повертає результат.

2. Розкажіть про сервіс-орієнтовану архітектуру.

Це підхід до проєктування ПЗ, де додаток будується як набір розподілених сервісів, що взаємодіють через мережу (часто через корпоративну шину ESB). Головна мета – повторне використання бізнес-логіки в масштабах підприємства.

3. Якими принципами керується SOA?

Основні принципи: слабка зв'язність (loose coupling), повторне використання сервісів, автономність, абстракція (приховування внутрішньої реалізації) та стандартизація контрактів взаємодії.

4. Як між собою взаємодіють сервіси в SOA?

Зазвичай через Enterprise Service Bus (ESB) – проміжне програмне забезпечення, яке керує маршрутизацією та перетворенням форматів даних. Часто використовується протокол SOAP та формат XML.

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

Через Service Registry (реєстр сервісів) або репозиторій. Структура запитів описується у формальних контрактах (наприклад, WSDL для SOAP), які пояснюють, які методи доступні та які дані потрібні.

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

Переваги: Централізоване керування даними, простіше забезпечення безпеки, легкість оновлення серверної частини.

Недоліки: Сервер є єдиною точкою відмови (SPOF) та вузьким місцем продуктивності при великій кількості запитів.

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

Переваги: Висока відмовостійкість (мережа працює, навіть якщо частина вузлів вимкнеться), легка масштабованість, дешевизна (не потрібен потужний сервер).

Недоліки: Складність адміністрування, проблеми з безпекою даних, відсутність централізованого контролю.

8. Що таке мікро-сервісна архітектура?

Це стиль розробки, при якому єдиний додаток розділяється на набір невеликих, незалежних сервісів. Кожен сервіс виконує одну бізнес-функцію, має власну базу даних і може бути розгорнутий окремо.

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

Синхронні: HTTP/REST, gRPC.

Асинхронні (через брокери повідомлень): AMQP (RabbitMQ), протоколи Kafka.

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 архітектурний патерн Service Layer (Шар сервісів) у межах монолітної архітектури. Справжня SOA (і мікросервіси) передбачає, що сервіси фізично відокремлені, працюють як різні процеси та спілкуються через мережу, а не через виклик методів класу в пам'яті однієї програми.

Висновки: у ході виконання даної лабораторної роботи було ознайомлено зі сучасними структурами програмного забезпечення та опрацьовано найкращі випадки для використання відповідних архітектур. Було успішно спроєктовано та реалізовано розподілену програмну систему для роботи з електронною поштою, побудовану на принципах сервіс-орієнтованої архітектури та багаторівневої клієнт-серверної архітектури. Для кінцевого користувача система є сервером, що надає зручний веб-інтерфейс для керування листами. Одночасно з цим, для виконання певної бізнес-логіки, додаток діє як клієнт, взаємодіючи з хмарним сервісом Google Gmail API. Реалізовано сучасний механізм автентифікації та авторизації за стандартом OAuth2.0. Використання токенів доступу замість збереження паролів користувачів дозволило підвищити рівень безпеки додатку та відповідає вимогам до сучасних розподілених систем. У проєкті також використано патерн проєктування MVC (Model-View-Controller) для розділення логіки відображення та обробки даних.