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»

Виконала:
студентка групи ІА-33
Самойленко Анастасія

Перевірив:
асистент кафедри ІСТ
Мягкий Михайло Юрійович

Київ 2025

Тема: Взаємодія компонентів системи
Мета: Вивчити види взаємодії додатків (Client-Server, Peer-to-Peer, Serviceoriented Architecture), та реалізувати в проєктованій системі одну із архітектур.
Тема роботи:
27. Особиста бухгалтерія (state, prototype, decorator, bridge, flyweight, SOA)
Програма повинна бути наочним засобом для ведення особистих фінансів: витрат і прибутку; з можливістю встановлення періодичних витрат / прибутку (зарплата і орендна плата); введення сканованих чеків з відповідними статтями витрат; побудова статистики; експорт/імпорт в Excel, реляційні джерела даних; різні рахунки; ведення єдиного фонду на всі рахунки (всією сім'єю) – на особливі потреби (ремонт, автомобіль, відпустка); можливість введення вкладів / кредитів для контролю банківських рахунків (звірка нарахованих відсотків з необхідними і т.д.).

Вступ

У сучасних інформаційних системах взаємодія компонентів є ключовим елементом для забезпечення обміну даними, координації дій та підтримки розподілених функцій. Основні моделі взаємодії включають клієнт-серверну архітектуру, однорангову (Peer-to-Peer) та сервісно-орієнтовану архітектуру (SOA). У клієнт-серверній моделі клієнти надсилають запити серверу, який обробляє їх і повертає результати. У однорангових мережах кожен вузол одночасно виконує роль клієнта та сервера, забезпечуючи прямий обмін даними між учасниками. У SOA-компонентах реалізуються сервіси, які надають певні функції через стандартизовані інтерфейси, дозволяючи інтегрувати різні додатки та системи.

Лабораторна робота передбачає ознайомлення з теоретичними основами взаємодії компонентів, проектування структури класів та їхньої взаємодії для досягнення конкретного функціоналу, а також реалізацію частини системи у вигляді розподілених компонентів. У звіті подаються діаграма класів, яка відображає архітектуру системи, та фрагменти коду, що демонструють взаємодію основних компонентів і реалізацію обраної архітектури.

Теоретичні відомості

Клієнт-серверна архітектура

Клієнт-серверні додатки являють собою найпростіший варіант розподілених додатків, де виділяється два види додатків: клієнти (представляють додаток користувачеві) і сервери (використовується для зберігання і обробки даних). 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.

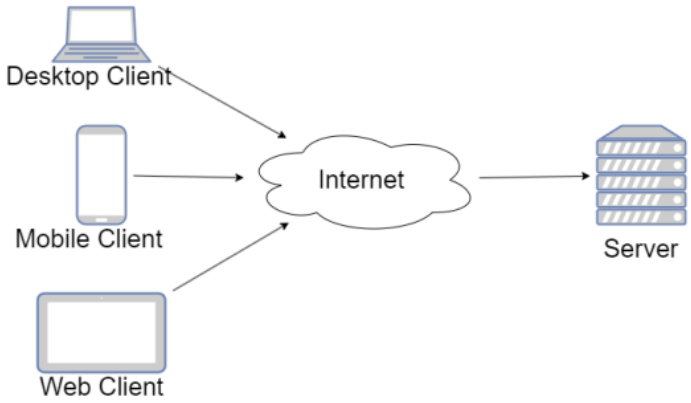


Рисунок 9.1. Клієнт-серверна архітектура

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 (наприклад, бази даних) 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. Наприклад, Evernote, Viber, MS Outlook, комп'ютерні антивіруси, ігри, що потребують інсталяції (The Sims, GTA, …) та інші.

Проміжним варіантом можна назвати SPA (Single Page Application) – це товсті Web-клієнти, які при старті кожен раз завантажуються з сервера, а надалі працюють з сервером через web-API. З одного боку більшу частину логіки вони відпрацьовують на клієнтський стороні, за рахунок чого зменшується серверне навантаження. Також оновлення простіше ніж для товстих клієнтів. Але такі застосунки не працюють, якщо сервер не доступний.

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. Клієнтська частина містить візуальне відображення і логіку обробки дії користувача; код для встановлення сеансу зв'язку з сервером і виконання відповідних викликів.

Серверна частина містить основну логіку роботи програми (бізнес-логіку) або ту її частину, яка відповідає зберіганню або обміну даними між клієнтом і сервером або клієнтами.

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. У цій архітектурі всі вузли мають рівні права та можливості для обміну даними, ресурсами або виконання завдань. 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.

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.

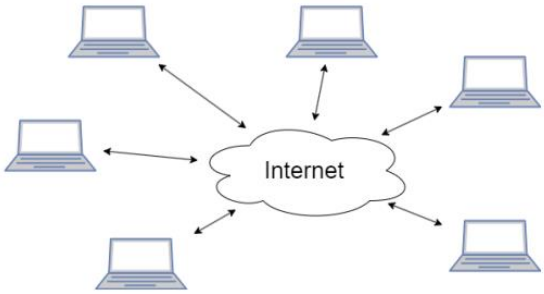


Рисунок 2. Peer-to-Peer архітектура

Основними сферами де peer-to-peer архітектура знайшла широке застосування є файлообмінники (BitTorrent), криптовалюти та інші блокчейнтехнології, інтернет телефонія та відеоконференції (Skype, Zoom), розподілені обчислення (SETI@home, BOINC).

До основних проблемних зон можна віднести безпеку, синхронізацію даних та пошук ресурсів. Через централізацію складно контролювати дані, які передаються. Ефективність пошуку даних знижується зі збільшенням кількості вузлів у мережі і для підвищеня ефективності пошуку потрібно застосовувати спеціальні алгоритми.

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 . Історично сервіс-орієнтована архітектура появилася як альтернатива монолітній архітектурі, в які вся система розроблялася та розгорталася як одне ціле.

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.
Згідно SOA сервіси реєструються на спеціальних сервісах і будь-яка команда розробників, якій потрібен доступ може знайти їх та використовувати.
Часто реалізація SOA покладається на використання централізованого програмного компонента для обміну даними – шину даних (Enterprise Service Bus).
Мікросервісна архітектура є подальшим розвитком сервіс-орієнтованої архітектури з використанням нових напрацювань у інформаційних технологіях.

Мікро-сервісна архітектура.

Сама назва дає зрозуміти, що мікро-сервісна архітектура є підходом до створення серверного додатку як набору малих служб [11]. 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.
Визначення мікросервісів із книги Іраклі Надарейшвілі, Ронні Мітра, Метта Макларті та Майка Амундсена (О'Рейлі) «Архітектура мікросервісів»:
«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.

Хід роботи

1) Діаграма класів частини функціоналу робочої програми. (рис. 1)

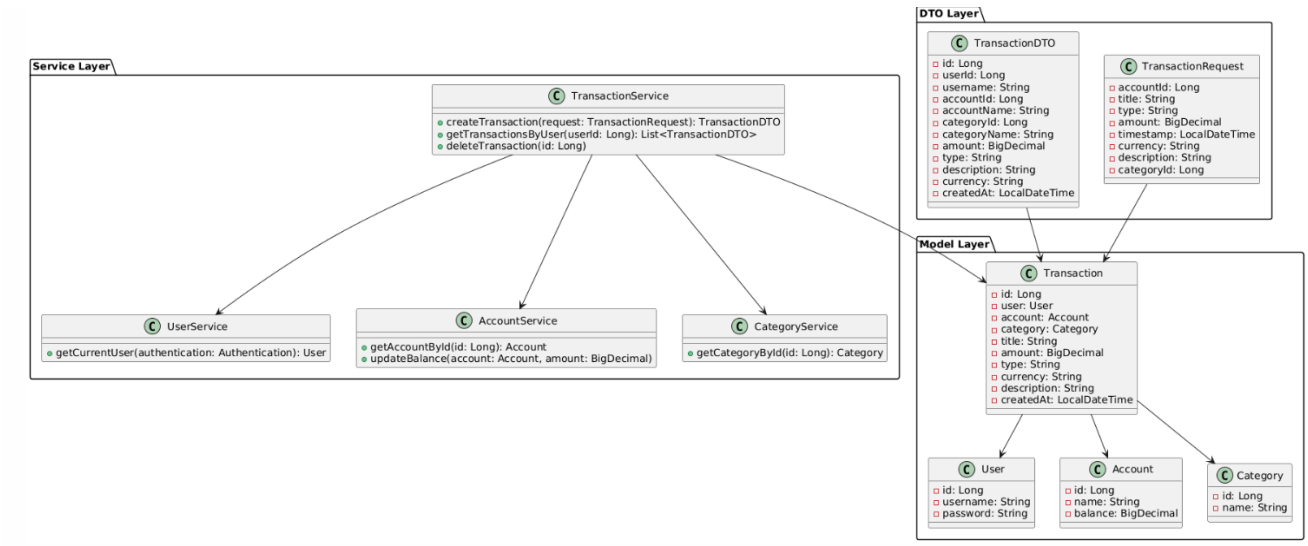
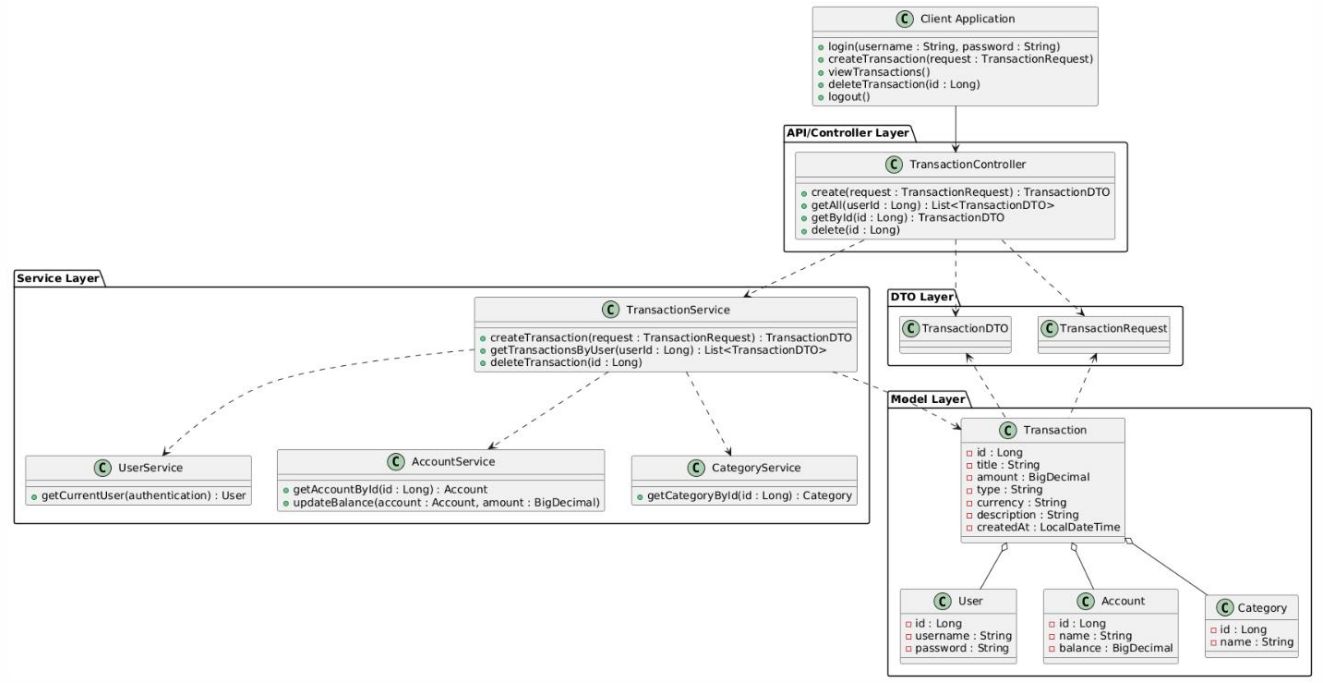


Рисунок 1 – Діаграма класів



2) Реалізація

У проєктованій системі була обрана архітектура Service-Oriented Architecture (SOA), що дозволяє розділити систему на незалежні сервіси, кожен із яких виконує конкретну функціональну задачу та надає стандартизований інтерфейс для взаємодії з іншими компонентами системи.

Для реалізації SOA у нашій системі ми створили окремі сервіси, які відповідають за ключові функції додатку:

1. Сервіс транзакцій — обробляє створення, збереження, отримання та видалення фінансових транзакцій користувача.
2. Сервіс користувачів — забезпечує автентифікацію та отримання даних про користувачів.
3. Сервіс рахунків і категорій — надає доступ до даних про рахунки користувачів та категорії транзакцій.

Взаємодія між сервісами відбувається через стандартизовані методи (інтерфейси REST API), що дозволяє розподілити логіку програми та зробити її більш модульною. Кожен сервіс працює незалежно, а дані передаються у вигляді DTO (Data Transfer Object), що забезпечує контрольовану та безпечну передачу інформації.

Переваги реалізації SOA у системі:

- Модульність і розподіленість: кожен сервіс можна розгортати та масштабувати незалежно від інших.
- Можливість інтеграції: сервіси мають стандартизовані інтерфейси, що полегшує інтеграцію з іншими додатками або зовнішніми сервісами.
- Гнучкість розвитку: нові функціональні можливості можна додавати як окремі сервіси, не змінюючи існуючу систему.
- Підвищена надійність: відмова одного сервісу не призводить до повного зупинення системи, оскільки інші сервіси продовжують працювати.
- Безпека та контроль: завдяки DTO та централізованій обробці автентифікації забезпечується контроль за доступом до ресурсів та безпечна передача даних.

Опис ключових моментів розробки:

```
server.port=8080
server.ssl.enabled=true
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/personal_finance
spring.datasource.username=root
spring.datasource.password=N_030306-a
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.ddl-auto=update
```

У конфігураційному файлі application.properties ми налаштували основні параметри роботи сервера та підключення до бази даних. Параметр server.port=8080 визначає порт, на якому запускається сервер, а server.ssl.enabled=true забезпечує безпечне з'єднання через HTTPS.

Для роботи з базою даних ми вказали URL підключення (spring.datasource.url), логін та пароль користувача (spring.datasource.username та spring.datasource.password), а також драйвер JDBC (spring.datasource.driver-class-name). Це дозволяє Spring Boot встановити зв'язок із MySQL та обмінюватися даними через Hibernate.

Параметр spring.jpa.hibernate.ddl-auto=update автоматично синхронізує структуру бази даних із сутностями моделі (Entity-класи), що полегшує розробку та підтримку системи, оскільки не потрібно вручну створювати або оновлювати таблиці.

Таким чином, цей блок конфігурації забезпечує основу для роботи серверної частини нашої Service-oriented Architecture, дозволяючи обробляти запити клієнтського застосунку, працювати з усіма сутностями та гарантувати безпечне підключення до бази даних.

```java
public class JwtAuthFilter extends OncePerRequestFilter {
    private static final String SECRET = "replace-with-strong-secret";
    private final UserDetailsService userDetailsService;
    public JwtAuthFilter(UserDetailsService userDetailsService) {
        this.userDetailsService = userDetailsService;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {

        String authHeader = request.getHeader("Authorization");

        if (authHeader == null || !authHeader.startsWith("Bearer ")) {
            filterChain.doFilter(request, response);
            return; }

        String token = authHeader.substring(7);
        String userId = null;

        try {
            userId = JWT.require(Algorithm.HMAC256(SECRET))
                    .build()
                    .verify(token)
                    .getSubject();
        } catch (JWTVerificationException ignored) { }

        if (userId != null && SecurityContextHolder.getContext().getAuthentication() == null) {

            UserDetails userDetails = userDetailsService.loadUserByUsername(userId);
            UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(
                    userDetails, null, userDetails.getAuthorities()
            );

            SecurityContextHolder.getContext().setAuthentication(authentication);
        }
        filterChain.doFilter(request, response);
```

```java
@RestController
@RequestMapping("/api/v1/auth")
public class AuthController {

    private static final String SECRET = "replace-with-strong-secret";

    private final UserRepo userRepo;

    public AuthController(UserRepo userRepo) {
        this.userRepo = userRepo;
    }

    @PostMapping("/login")
    public ResponseEntity<Map<String, Object>> login(@RequestBody Map<String, String> body) {
        String username = body.get("username");
        String password = body.get("password");

        Optional<User> userOpt = userRepo.findByUsername(username);
        if (userOpt.isEmpty()) {
            return ResponseEntity.status(404).body(Map.of("error", "User not found"));
        }
        User user = userOpt.get();

        if (user.getPassword() == null || !user.getPassword().equals(password)) {
            return ResponseEntity.status(401).body(Map.of("error", "Invalid credentials"));
        }

        String token = JWT.create()
                .withSubject(user.getId().toString())
                .withIssuer("pa-service")
                .withExpiresAt(new Date(System.currentTimeMillis() + 3600_000))
                .sign(Algorithm.HMAC256(SECRET));

        Map<String, Object> res = new HashMap<>();
        res.put("token", token);
        res.put("user", user);
```

```
    @PostMapping("/register")
    public ResponseEntity<User> register(@RequestBody RegistrationRequest request) {
        if (userRepo.findByUsername(request.getUsername()).isPresent()) {
            return ResponseEntity.status(409).body(null);
        }

        User newUser = new User();
        newUser.setUsername(request.getUsername());
        newUser.setPassword(request.getPassword());
        newUser.setFullName(request.getFullName());
        newUser.setEmail(request.getEmail() != null ? request.getEmail() : "");

        User createdUser = userRepo.save(newUser);
        return ResponseEntity.ok(createdUser);
    }
}
```

У нашій системі для передачі даних про автентифікацію використовується JWT (JSON Web Token). Використання токенів дозволяє реалізувати безпечну та незалежну від стану сесію автентифікації між клієнтським застосунком і сервером, що відповідає принципам архітектури Service-Oriented Architecture (SOA). Під час логіну користувача сервер перевіряє введені облікові дані (username і пароль). У разі успішної автентифікації сервер генерує токен, який містить інформацію про користувача (наприклад, userId) та підписується секретним ключем. Токен має обмежений час життя, після чого його потрібно оновити або заново авторизуватися.

Клієнтський застосунок зберігає отриманий токен і прикріплює його до заголовків кожного подальшого запиту до серверних API. Сервер перевіряє токен, і якщо він валідний та не прострочений, надає доступ до захищених ресурсів.

Таке рішення має низку переваг:

- Безпека: токен підписаний, що запобігає його підробці; дані автентифікації не передаються у відкритому вигляді.
- Масштабованість: сервер не зберігає стан сесії, що дозволяє легко розширювати систему та використовувати балансувальники навантаження.
- Простота інтеграції: клієнтські застосунки можуть взаємодіяти з сервером незалежно від технології або платформи.

У нашій реалізації приклад використання токенів показано у класі AuthController, де метод login генерує JWT після перевірки облікових даних, а клієнт використовує його для подальших запитів до API. Таким чином, токени

забезпечують надійну і безпечну передачу інформації про автентифікацію між компонентами системи.

Ключові моменти реалізації (без коду):

1. Сервісна логіка та контролери:
    - Контролери (TransactionController, AuthController) приймають запити від клієнтського застосунку, обробляють їх та викликають відповідні сервіси.
    - Сервіси (TransactionService, UserService, AccountService, CategoryService) реалізують основну бізнес-логіку, наприклад, створення транзакцій, оновлення балансу рахунку, обробку категорій та користувачів.
2. Передача даних між клієнтом і сервером:
    - Використовується формат DTO (TransactionDTO, TransactionRequest) для відділення моделі даних від API.
    - Клієнтський застосунок взаємодіє з сервером через REST API, отримуючи структуровані відповіді.
3. Безпека та автентифікація:
    - Реалізовано автентифікацію користувачів за допомогою JWT-токенів. Клієнт отримує токен при логіні і передає його у заголовку кожного запиту.
    - Використання токенів дозволяє бездержавну (stateless) автентифікацію та безпечну взаємодію в розподіленому середовищі.
4. Робота з базою даних:
    - Використовується Spring Data JPA для роботи з MySQL.
    - Моделі (User, Account, Category, Transaction) відображаються на таблиці БД.
    - Операції створення, оновлення та видалення транзакцій автоматично відображаються на балансі рахунків.

Загалом, реалізація демонструє чітке розділення обов’язків між шарами (контролери, сервіси, моделі), масштабовану і безпечну архітектуру, а також готовність до роботи у розподіленому середовищі з різними клієнтськими застосунками.

Відповіді на контрольні запитання:

1. Що таке клієнт-серверна архітектура?

Клієнт-серверна архітектура — це провідна модель розподілених обчислень, у якій функціональність системи розділена між двома основними ролями, які взаємодіють через комп'ютерну мережу: клієнтом та сервером. Клієнт — це ініціатор запиту, як правило, пристрій або програма (наприклад, веббраузер або мобільний додаток), що надає користувачеві інтерфейс і формує запити на отримання певного сервісу чи ресурсу. Сервер — це надавач сервісу; це постійно активна програма або потужний пристрій, який очікує на вхідні запити, обробляє їх, керує спільними ресурсами (наприклад, базами даних) і надсилає відповідь клієнту. Взаємодія відбувається за чітким механізмом "запит-відповідь". Основна цінність цієї архітектури полягає у централізації управління даними та ресурсами на сервері, що забезпечує підвищену безпеку, легше адміністрування та масштабованість системи, дозволяючи незалежно нарощувати потужності сервера або кількість клієнтів.

2. Розкажіть про сервіс-орієнтовану архітектуру.

Сервіс-орієнтована архітектура (СОА), або Service-Oriented Architecture (SOA), є парадигмою організації корпоративних IT-систем, за якої складні бізнес-процеси та функціонал реалізуються не як монолітні застосунки, а як набір слабко зв'язаних, самодостатніх і багаторазово використовуваних компонентів, що називаються сервісами. Основна ідея полягає у декомпозиції великих систем на модульні сервіси, кожен з яких виконує певну бізнес-функцію (наприклад, "перевірка кредитного рейтингу" або "оформлення замовлення"). Взаємодія між цими сервісами, а також між сервісами та зовнішніми застосунками, відбувається виключно через чітко визначені, стандартизовані інтерфейси (контракти), незалежно від технології чи платформи, на якій вони реалізовані. Ключові принципи СОА включають слабку зв'язаність — мінімальну залежність сервісів один від одного, багаторазове використання — можливість застосування одного сервісу у різних бізнес-процесах, та автономність — здатність сервісу самостійно керувати своєю логікою та даними. Така архітектура часто передбачає використання Шини Корпоративних Сервісів (ESB) як центрального посередника

для маршрутизації та трансформації повідомлень, що забезпечує гнучку інтеграцію різнорідних систем, прискорює впровадження нових бізнес-процесів та підвищує загальну адаптивність IT-інфраструктури компанії до змін.

3. Якими принципами керується SOA?

Принципи, якими керується сервіс-орієнтована архітектура (SOA), є ключовими для її успішного впровадження та полягають у забезпеченні максимальної гнучкості та багаторазового використання компонентів. Головний принцип — це слабка зв'язаність (Loose Coupling), що означає, що сервіси мінімально залежать один від одного, взаємодіючи лише через чітко визначені, публічні інтерфейси (контракти), що дозволяє незалежно змінювати внутрішню логіку одного сервісу без порушення роботи інших. Наступний важливий принцип — багаторазове використання (Reusability): сервіси повинні бути спроектовані як самодостатні одиниці, що виконують конкретну бізнес-функцію і можуть бути задіяні у різних застосунках та бізнес-процесах, що зменшує дублювання коду та прискорює розробку. Принцип автономності (Autonomy) вимагає, щоб кожен сервіс повністю контролював свою логіку, оточення та, що дуже важливо, свої дані, забезпечуючи незалежність від інших компонентів. Крім того, СОА керується принципом стандартизованих контрактів (Standardized Contracts), згідно з яким комунікація між сервісами має відбуватися за допомогою відкритих та загальноприйнятих протоколів (як-от SOAP чи REST), що забезпечує сумісність між різнорідними технологічними платформами. Нарешті, принцип абстракції (Abstraction) гарантує, що внутрішня реалізація сервісу є прихованою від споживача, який бачить лише його інтерфейс, що додатково посилює слабку зв'язаність. Усі ці принципи забезпечують гнучку інтеграцію, масштабованість та здатність корпоративної системи швидко адаптуватися до мінливих вимог бізнесу.

4. Як між собою взаємодіють сервіси в SOA?

Взаємодія сервісів у Сервіс-орієнтованій архітектурі відбувається на основі принципів слабкої зв'язаності та стандартизованих контрактів . Споживач Сервісу ініціює взаємодію, надсилаючи запит до Постачальника Сервісу через його чітко визначений інтерфейс. Комунікація використовує стандартизовані протоколи, такі як SOAP (для складних обмінів XML-повідомленнями) або REST (для легких

обмінів через HTTP з використанням JSON/XML). Взаємодія може бути синхронною (з очікуванням миттєвої відповіді) або асинхронною (через черги повідомлень). У великих системах центральним інтеграційним посередником виступає Шина Корпоративних Сервісів (ESB). ESB забезпечує маршрутизацію запитів, трансформацію форматів даних між сервісами, моніторинг та безпеку, ефективно ізолюючи споживача від технологічних деталей реалізації постачальника. Для первинного пошуку потрібних сервісів використовується Реєстр Сервісів.

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

Розробники знаходять існуючі сервіси, шукаючи їх у Реєстрі Сервісів (Service Registry), який слугує каталогом, де Постачальники Сервісів публікують метадані та, найголовніше, Контракт (інтерфейс) сервісу. Для традиційних SOAP-сервісів цим контрактом є файл WSDL, який описує всі операції та формати XML-повідомлень. Для сучасних REST-сервісів це OpenAPI/Swagger специфікація, що описує URL, методи HTTP та структуру JSON-даних. Отримавши Контракт, розробник використовує інструменти для автоматичної генерації клієнтського коду, який дозволяє викликати віддалений сервіс як локальну функцію. Запит формується відповідно до вимог Контракту і надсилається через мережу, часто проходячи через Шину Корпоративних Сервісів (ESB), яка забезпечує маршрутизацію та трансформацію, підтримуючи принцип слабкої зв'язаності.

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

Переваги Клієнт-Серверної Моделі

1. Централізоване управління (Centralized Control): Усі ключові ресурси, дані та бізнес-логіка зберігаються і контролюються на сервері. Це значно спрощує адміністрування системи, управління безпекою, забезпечення цілісності даних та проведення резервного копіювання.
2. Масштабованість (Scalability): Модель дозволяє незалежно нарощувати потужність. За потреби можна оновити або замінити сервер на більш потужний, не втручаючись у конфігурацію клієнтів, або ж просто додати більше клієнтів.

3. Спільне використання ресурсів: Сервер забезпечує одночасний доступ багатьох клієнтів до спільних ресурсів, таких як бази даних, принтери чи обчислювальна потужність.
4. Ефективність та Економічність: Клієнтські машини можуть бути менш потужними ("тонкі клієнти"), оскільки основні обчислення та обробка даних виконуються на централізованому сервері.
5. Надійність: Дані зберігаються на професійному серверному обладнанні з резервуванням та контролем, що зазвичай вища, ніж на окремих клієнтських робочих станціях.

Недоліки Клієнт-Серверної Моделі

1. Висока вартість сервера: Необхідність у потужному та надійному центральному сервері, а також спеціалізованому програмному забезпеченні та кваліфікованому персоналі для його обслуговування, вимагає значних початкових інвестицій.
2. Центральна точка відмови (Single Point of Failure): Якщо центральний сервер виходить з ладу, усі клієнти та, відповідно, вся система перестають функціонувати, доки сервер не буде відновлено. Це критичний недолік, який вимагає високої відмовостійкості.
3. Залежність від мережі: Взаємодія повністю залежить від стабільності та пропускної здатності мережевого з'єднання між клієнтом і сервером. Повільна або непрацююча мережа унеможливлює роботу системи.
4. Перевантаження сервера: Якщо кількість клієнтів або інтенсивність їхніх запитів значно зростає, центральний сервер може бути перевантажений, що призводить до падіння продуктивності системи для всіх користувачів.

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

Переваги Однорангової Моделі (P2P)

1. Стійкість та Надійність (Robustness): Модель не має єдиної точки відмови (Single Point of Failure). Вихід з ладу одного чи навіть кількох вузлів не

призводить до зупинки всієї мережі, оскільки функціональність розподілена.

2. Масштабованість (Scalability) та Простота розгортання: Додавання нових рангів у мережу є простим і не вимагає складної центральної інфраструктури. Продуктивність і доступність ресурсів можуть зростати пропорційно збільшенню кількості учасників.
3. Ефективне використання ресурсів: P2P-мережі ефективно використовують сукупну обчислювальну потужність, пропускну здатність та дисковий простір усіх учасників. Замість покладатися на один центральний сервер, навантаження розподіляється.
4. Низька вартість: Немає потреби у дорогому, високопродуктивному серверному обладнанні та складному адмініструванні централізованої системи.

Недоліки Однорангової Моделі (P2P)

1. Складність управління та безпеки: Відсутність центрального управління ускладнює забезпечення безпеки, застосування єдиної політики доступу, моніторинг та централізоване резервне копіювання даних.
2. Ненадійність ресурсів: Оскільки кожен ранг є автономним і може довільно залишати мережу, доступність певних ресурсів не гарантована. Ресурс може бути доступний лише доти, доки ранг, який його зберігає, знаходиться онлайн.
3. Проблеми з пошуком: У чисто децентралізованих P2P-мережах пошук потрібного ресурсу може бути повільним і неефективним, оскільки запит, можливо, доведеться розсилати по багатьом вузлам.
4. Продуктивність (Performance): Продуктивність часто залежить від якості інтернет-з'єднання та обчислювальної потужності кожного окремого рангу, які можуть сильно відрізнятися.

8. Що таке мікро-сервісна архітектура?

Мікросервісна архітектура (МСА) – це сучасний підхід до розробки програмного забезпечення, при якому великий застосунок розбивається на набір невеликих,

незалежних сервісів, кожен з яких реалізує одну конкретну бізнес-функцію і розгортається автономно. Ключовими характеристиками МСА є: технологічна гетерогенність (сервіси можуть бути написані на різних мовах програмування), децентралізоване управління даними (кожен сервіс має свою базу даних) та слабка зв'язаність. Сервіси спілкуються між собою за допомогою легких протоколів, таких як REST/HTTP. На відміну від моноліту, МСА забезпечує вищу швидкість розробки, незалежне масштабування окремих компонентів і кращу відмовостійкість, оскільки збій одного сервісу не призводить до зупинки всієї системи.

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

Для обміну даними в мікросервісній архітектурі використовується набір різноманітних протоколів, які можна розділити на дві основні категорії: синхронні (для негайної відповіді) та асинхронні (для комунікації через повідомлення), що забезпечує гнучкість та відмовостійкість системи.

1. Синхронні Протоколи (Запит-Відповідь)

Ці протоколи використовуються для прямої взаємодії, коли сервіс-споживач очікує негайної відповіді від сервісу-постачальника:

- REST (Representational State Transfer) over HTTP: Це найпоширеніший протокол. Він є легким, простим і використовує стандартні методи HTTP (GET, POST, PUT, DELETE) для маніпуляції ресурсами. Дані зазвичай передаються у форматі JSON (через його компактність і простоту) або іноді XML.
- gRPC (Google Remote Procedure Call): Це сучасний фреймворк для віддаленого виклику процедур. На відміну від REST, gRPC використовує протокол HTTP/2 для високої продуктивності та Protocol Buffers як механізм серіалізації даних. Він забезпечує двонаправлену потокову передачу даних і є ефективнішим для комунікації між сервісами.

2. Асинхронні Протоколи (Керована Подіями)

Ці протоколи використовуються для комунікації, керованої подіями (Event-Driven Architecture), де сервіс надсилає повідомлення про подію, не очікуючи негайної відповіді:

- AMQP (Advanced Message Queuing Protocol): Це відкритий стандартний протокол для обміну повідомленнями. Він використовується в брокерах повідомлень, таких як RabbitMQ, для забезпечення надійної та безпечної асинхронної комунікації між сервісами.
- Kafka Protocol: Протокол, який використовується в розподіленій потоковій платформі Apache Kafka. Він ідеально підходить для високонавантажених сценаріїв, таких як обробка потоків подій, збору логів і обміну великими обсягами даних між мікросервісами.
- MQTT (Message Queuing Telemetry Transport): Легкий протокол, оптимізований для передачі даних від пристроїв з обмеженими ресурсами (наприклад, IoT-пристроїв), але також може використовуватися для внутрішньої асинхронної комунікації між сервісами.

Вибір протоколу залежить від конкретних потреб сервісу: синхронні для транзакційних операцій, де потрібна негайна відповідь, і асинхронні для забезпечення високої відмовостійкості, слабкої зв'язаності та реакції на події.

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, такий підхід не є Сервіс-орієнтованою архітектурою (СОА). Реалізація шару бізнес-логіки у вигляді класів, які ви називаєте "сервісами", між шарами веб-контролерів та доступу до даних, є типовим прикладом Багаторівневої (або Трирівневої) Архітектури і використовується для внутрішньої структуризації коду в межах одного монолітного застосунку. У цьому випадку "сервіси" є локальними, тісно зв'язаними класами, які викликаються безпосередньо. Натомість, СОА — це архітектурний стиль корпоративного масштабу, де кожен "сервіс" є незалежним, автономним застосунком, який розгортається окремо,

доступний для зовнішніх систем через мережеві протоколи (REST/SOAP) і має слабку зв'язаність з іншими компонентами.

Висновок:

У ході виконання лабораторної роботи я реалізувала систему з архітектурою Service-Oriented Architecture (SOA), що дозволило чітко розділити функціональні шари та забезпечити гнучку взаємодію між серверною та клієнтською частинами застосунку. Застосування SOA надало кілька ключових переваг: підвищену масштабованість і підтримуваність коду завдяки розділенню контролерів, сервісів і моделей, спрощену інтеграцію нових компонентів, можливість багаторазового використання сервісів різними клієнтськими застосунками, а також забезпечення безпечної автентифікації через використання JWT-токенів.

До основних позитивних результатів можна віднести: централізовану обробку бізнес-логіки, зменшення дублювання коду, гнучкість у розширенні функціоналу та легкість підтримки та тестування окремих компонентів. Використання DTO дозволило ізолювати внутрішню модель даних від зовнішнього API, що підвищує безпеку та стабільність взаємодії.

Серед потенційних недоліків варто відзначити підвищену складність налаштування та конфігурації сервісів, необхідність продуманого управління транзакціями та обробки помилок у розподіленому середовищі, а також потребу в належному моніторингу та логуванні для відстеження взаємодії між сервісами.

Загалом, реалізація SOA в цій системі дозволила створити масштабовану, безпечну та гнучку платформу, яка готова до розширення та інтеграції з різними клієнтськими застосунками, забезпечуючи при цьому надійну і структуровану організацію коду та бізнес-логіки.