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

Лабораторна робота № 9

із дисципліни «Технології розроблення програмного забезпечення»

Тема: «Взаємодія компонентів системи»

Виконала
Студентка групи ІА-31: Перевірив:
Трегуб К. В. Мягкий М. Ю.

Київ 2025

Зміст

1. Мета:

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

2. Теоретичні відомості

Клієнт-серверна архітектура — це модель, де клієнт відповідає за взаємодію з користувачем, а сервер обробляє та зберігає дані. У випадку тонкого клієнта (наприклад, вебдодатки) більшість операцій виконується на сервері, що спрощує оновлення. Товстий клієнт (мобільні або десктопні програми) обробляє логіку локально, зменшуючи навантаження на сервер і дозволяючи працювати офлайн. SPA (Single Page Application) — це проміжний варіант: логіка виконується на клієнті, але робота можлива лише з підключенням до сервера. Типова структура включає три рівні: клієнтський (інтерфейс), проміжний (middleware) і серверний (бізнес-логіка та дані).

Peer-to-Peer (P2P) — децентралізована модель, де кожен вузол одночасно є клієнтом і сервером. Усі учасники рівноправні та обмінюються ресурсами без центрального сервера (наприклад, BitTorrent, блокчейн, Skype). Недоліки: складність забезпечення безпеки, синхронізації та пошуку даних у великих мережах.

Сервіс-орієнтована архітектура (SOA) — модульний підхід, де система складається з незалежних сервісів зі стандартизованими інтерфейсами (HTTP, SOAP, REST). Сервіси виконують конкретні бізнес-функції, обмінюються повідомленнями та можуть інтегруватися через Enterprise Service Bus (ESB). SOA стала основою для мікросервісів.

Мікросервісна архітектура — це створення додатків як набору незалежних малих сервісів, які взаємодіють через HTTP, WebSockets або AMQP. Кожен мікросервіс має власну логіку, життєвий цикл і може розгортатися автономно. Переваги: гнучкість, масштабованість та легке супроводження великих систем.

3. Хід роботи

Тема :

Beauty Salon Booking System (State, Chain of Responsibility, Observer, Facade, Composite, Client-Server) — веб-застосунок для бронювання послуг салону краси, що підтримує управління життєвим циклом запису, проходження валідованого процесу створення бронювання, автоматичне сповіщення користувачів про зміни статусу, централізовану обробку оплати через фасад та роботу як з окремими послугами, так і з пакетними пропозиціями. Система побудована за архітектурою

клієнт–сервер з розмежуванням бізнес-логіки, управління станом і відображенням інтерфейсу.

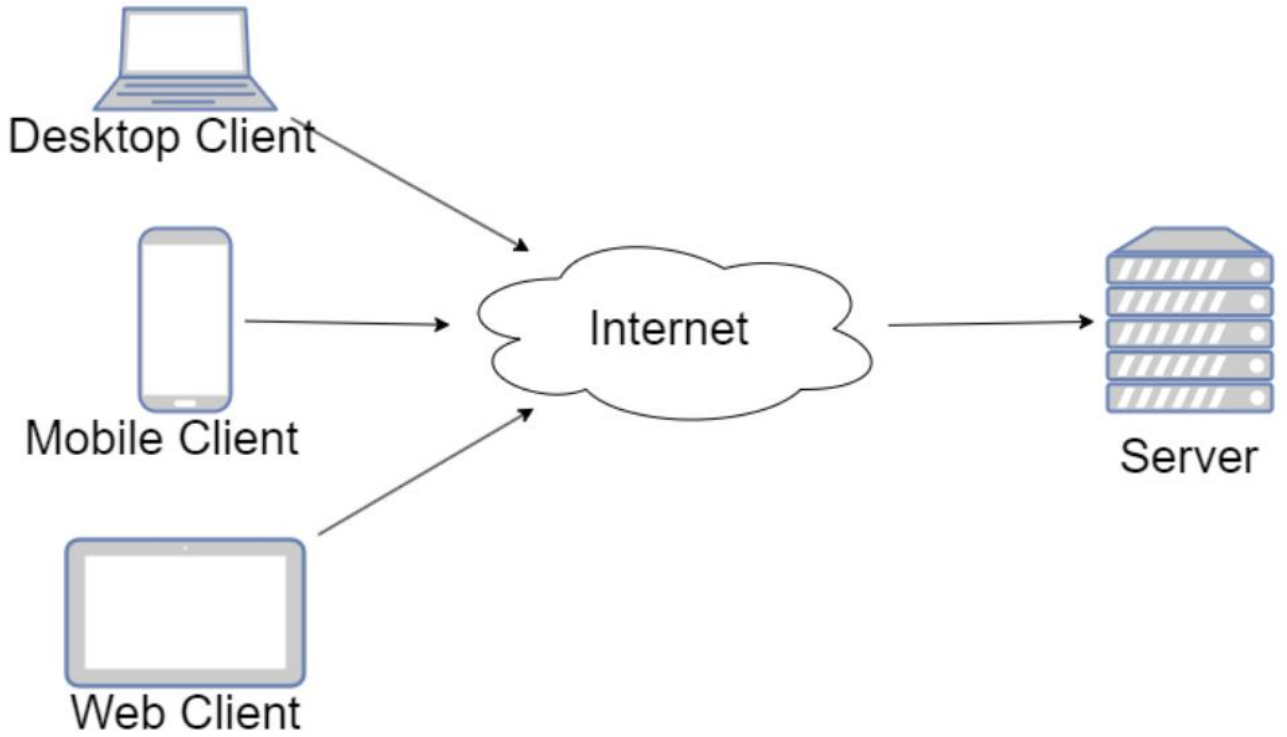

Рис.1 – Архітектура «клієнт-сервер»

Архітектура системи "Beauty Salon Booking"

У даному проєкті реалізовано класичну клієнт-серверну архітектуру (Client-Server Architecture) на базі протоколу HTTP. Система спроєктована як монолітний веб-застосунок, де сервер виконує основну обчислювальну роботу (Server-Side Rendering), а клієнт відповідає за відображення інтерфейсу та взаємодію з користувачем.

Серверна частина

Серверна частина побудована на базі фреймворку Spring Boot (Java). Вона виступає центральним вузлом системи і реалізує багатошарову структуру:

1. Presentation Layer (Контролери): BookingWebController, AuthController, AdminWebController. Вони приймають HTTP-запити від клієнта, обробляють вхідні параметри та визначають, який HTML-шаблон (View) повернути.
2. Business Logic Layer (Сервіси): Тут зосереджена основна логіка та реалізація патернів проєктування (BookingService, PaymentFacade, MasterService). Цей шар не залежить від вебінтерфейсу.

3. Data Access Layer (Репозиторії): Забезпечує доступ до бази даних MySQL через Spring Data JPA (Hibernate).

Клієнтська частина

Клієнтська частина реалізована як "тонкий клієнт" за допомогою технологій HTML5, CSS3 та шаблонізатора Thymeleaf.

- Відображення: HTML-сторінки генеруються на сервері (Server-Side Rendering) і надсилаються браузеру вже готовими.
- Інтерактивність: Для динамічних елементів (наприклад, фільтрація майстрів або завантаження вільних слотів часу) використовується JavaScript (Fetch API), який надсилає асинхронні запити на сервер без перезавантаження сторінки.

Middleware (Посередницький шар)

Цей прошарок інкапсулює всю складність системи. Контролери звертаються до нього, не знаючи деталей реалізації.

- Facade: PaymentFacade приховує складний процес оплати.
- Chain of Responsibility: Ланцюжок хендлерів у BookingService перевіряє можливість створення запису.
- State: Клас Booking та його стани контролюють життєвий цикл замовлення.
- Observer: BookingEventPublisher розсилає сповіщення при зміні даних.

Взаємодія клієнта і сервера

Сценарій 1: Створення бронювання (Клієнт -> Сервер)

1. Клієнт: Користувач заповнює форму на сторінці booking_create.html (обирає послугу, майстра, час). JavaScript динамічно підвантажує вільні слоти через AJAX-запит.
2. Запит: Браузер надсилає POST запит на /web/bookings/create.
3. Сервер (Контролер): BookingWebController приймає дані та передає їх у сервіс.
4. Сервер (Logic - Chain of Responsibility): BookingService запускає ланцюжок валідації (validationChain). Послідовно перевіряється наявність клієнта, майстра та їх сумісність. Якщо помилка — ланцюжок переривається, і сервер повертає сторінку з помилкою.
5. Сервер (Logic - Composite): Якщо валідація успішна, сервіс використовує патерн Composite для розрахунку кінцевої вартості (враховуючи базову послугу та додаткові опції пакету).

6. Сервер (Logic - Observer): Після збереження в БД спрацьовує BookingEventPublisher, який надсилає email-сповіщення.
7. Відповідь: Сервер повертає код перенаправлення (Redirect) на сторінку дашборду.

Сценарій 2: Оплата замовлення (Клієнт -> Сервер -> Facade)

1. Клієнт: На дашборді користувач натискає "Оплатити". Відкривається модальне вікно, вводяться дані картки.
2. Запит: Браузер надсилає POST запит на /web/bookings/{id}/pay.
3. Сервер (Контролер): Контролер делегує обробку об'єкту PaymentFacade.
4. Сервер (Logic - Facade + State + Strategy):
   - Фасад перевіряє через патерн State, чи знаходиться замовлення в статусі CONFIRMED (інакше оплата заборонена).
   - Фасад обирає стратегію оплати (Strategy) і проводить транзакцію.
   - Фасад зберігає чек у БД.
5. Відповідь: Сервер перенаправляє клієнта назад, де вже відображається оновлений статус PAID.

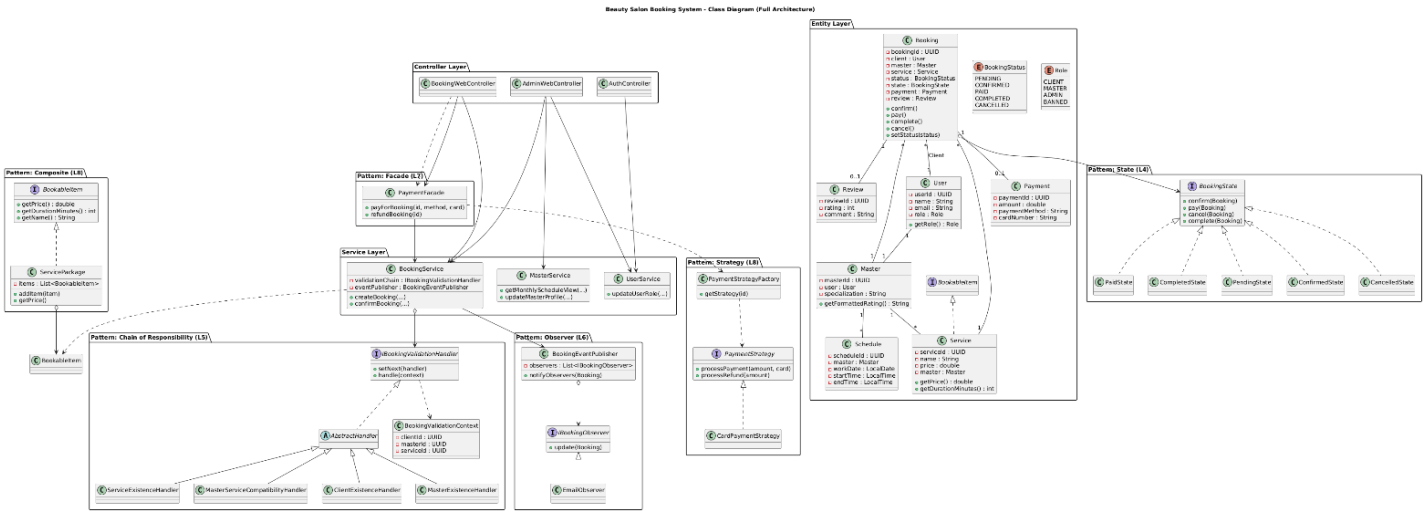

Рис. 2 - Діаграма класів

Структура рішення

Domain Layer (com.beautysalon.booking.entity) — Шар сутностей:

- Містить основні сутності предметної області (User, Booking, Master, Service, Schedule, Payment, Review).
- Визначає зв'язки між об'єктами (One-to-One, One-to-Many) та анотації JPA для об'єктно-реляційного відображення (ORM) на таблиці бази даних.

Data Access Layer (com.beautysalon.booking.repository) — Шар доступу до даних:

- Відповідає за безпосередню взаємодію з базою даних.
- Містить інтерфейси репозиторіїв (IUserRepository, IBookingRepository, IMasterRepository тощо), що розширюють стандартний інтерфейс JpaRepository.
- Забезпечує абстракцію від SQL-запитів, надаючи готові методи для CRUD-операцій та кастомних вибірок (наприклад, пошук вільних слотів часу).

Service Layer (com.beautysalon.booking.service, com.beautysalon.booking.payment, com.beautysalon.booking.validation) — Сервісний шар:

- Виступає проміжною ланкою, де зосереджена вся бізнес-логіка системи.
- Реалізує ключові патерни проєктування:
    - Chain of Responsibility: Валідація даних при створенні бронювання (validation package).
    - Observer: Система асинхронних сповіщень через BookingEventPublisher.
    - Facade: Клас PaymentFacade для спрощення складного процесу оплати.
    - Strategy: Гнучка обробка різних типів оплати (PaymentStrategy).
    - Composite: Робота з ієрархічними пакетами послуг (ServicePackage).
    - State: Управління життєвим циклом бронювання (BookingState).
- Координує роботу репозиторіїв та забезпечує цілісність даних за допомогою транзакцій (@Transactional).

Presentation Layer (com.beautysalon.booking.controller) — Шар представлення:

- Містить веб-контролери (BookingWebController, AdminWebController, AuthController), які приймають HTTP-запити від клієнта.
- Використовує DTO (MasterOptionDto, ScheduleDayDto) для безпечної передачі даних між клієнтом і сервером, уникаючи розкриття внутрішніх сутностей та проблем з рекурсією.
- Відповідає за вибір відповідного HTML-шаблону Thymeleaf та наповнення його даними (Model).

Infrastructure / Config (com.beautysalon.booking.config) — Інфраструктурний шар:

- Містить конфігураційні класи, такі як DatabaseInitializer (для автоматичного початкового наповнення БД при старті) та ObserverConfig (для налаштування підписок).

Frontend Resources (src/main/resources) — Ресурси:

- Templates: HTML-шаблони Thymeleaf для динамічного відображення сторінок.
- Static: Статичні ресурси (CSS-стилі beauty-salon.css, зображення).
- Application Properties: Файл конфігурації з налаштуваннями підключення до БД, поштового сервера та параметрів запуску.

Взаємодія Патернів

Розглянемо типовий процес "Створення та Оплата замовлення" і подивимося, де вступає в гру кожен патерн.

1. Валідація (Chain of Responsibility)

Момент: Коли Клієнт натискає кнопку "Забронювати". Дія: Перш ніж створити запис у базі, система має переконатися, що дані коректні.

- BookingService отримує запит і передає його в Ланцюжок (validationChain).
- Ланка 1 (ClientExistence): "Чи існує такий клієнт?" -> Так, йдемо далі.
- Ланка 2 (MasterExistence): "Чи існує такий майстер?" -> Так, йдемо далі.
- Ланка 3 (ServiceExistence): "Чи існує така послуга?" -> Так, йдемо далі.
- Ланка 4 (Compatibility): "Чи надає цей майстер цю послугу?" -> Так, все добре. Результат: Ланцюжок пройдено, можна створювати об'єкт.

```
IBookingValidationHandler clientHandler = new ClientExistenceHandler(userRepository);
IBookingValidationHandler masterHandler = new MasterExistenceHandler(masterRepository);
IBookingValidationHandler serviceHandler = new ServiceExistenceHandler(serviceRepository);
IBookingValidationHandler compatibilityHandler = new MasterServiceCompatibilityHandler();

clientHandler.setNext(masterHandler);
masterHandler.setNext(serviceHandler);
serviceHandler.setNext(compatibilityHandler);
this.validationChain = clientHandler;
```

Рис. 3 Збірка ланцюжка в конструкторі(BookingService.java)

```
public Booking createBooking(UUID clientId, UUID serviceId, UUID masterId, LocalDateTime desiredDateTime, boolean allInclusive) {
    BookingValidationContext context = new BookingValidationContext(clientId, masterId, serviceId, desiredDateTime);
    validationChain.handle(context);
    if (context.hasError()) {
        throw new RuntimeException(context.getErrorMessage());
    }
```

Рис. 4 Виклик у методі createBooking (BookingService.java)

2. Розрахунок ціни (Composite)

Момент: Безпосередньо перед збереженням нового бронювання. Дія: Системі треба знати, скільки коштує замовлення.

- Якщо клієнт обрав просто послугу, патерн працює як Листок (Service) і повертає ціну з бази (наприклад, 500 грн).
- Якщо клієнт обрав "All Inclusive", створюється Компонувальник (ServicePackage). Він містить основну послугу (500 грн) + додаткові послуги (200 грн).
- BookingService викликає метод getPrice() у спільного інтерфейсу BookableItem, не знаючи, що саме там всередині. Результат: Отримана фінальна ціна (700 грн).

```
BookableItem finalItem;
BookableItem baseService = context.getService();

if (allInclusive) {
    ServicePackage vipPackage = new ServicePackage("VIP-пакет: " + baseService.getName());
    vipPackage.addItem(baseService);

    com.beautysalon.booking.entity.Service addons =
        new com.beautysalon.booking.entity.Service(name: "VIP-додатки (Косметика, Масаж, Напої)", description: "All Inclusive", price: 200, durationMinutes: 15);

    vipPackage.addItem(addons);
    finalItem = vipPackage;
} else {
    finalItem = baseService;
}

Booking newBooking = new Booking();
newBooking.setClient(context.getClient());
newBooking.setMaster(context.getMaster());
newBooking.setService(context.getService());
newBooking.setBookingDate(context.getDateTime().toLocalDate());
newBooking.setBookingTime(context.getDateTime().toLocalTime());
newBooking.setTotalPrice(finalItem.getPrice());
newBooking.setStatus(BookingStatus.PENDING);

Booking savedBooking = bookingRepository.save(newBooking);
eventPublisher.notifyObservers(savedBooking);
return savedBooking;
```

Рис. 5 ServicePackage агрегує основну та додаткові послуги (BookingService.java)

3. Сповіщення про створення (Observer)

Момент: Одразу після збереження бронювання в БД (статус PENDING). Дія: BookingService викликає метод notifyObservers() у Видавця (BookingEventPublisher).

- Підписник (EmailObserver): Отримує об'єкт бронювання, формує лист ("Ваш запис створено") і відправляє його на пошту клієнта. Результат: Клієнт миттєво дізнається про успішний запис.

```
    Booking savedBooking = bookingRepository.save(newBooking);
    eventPublisher.notifyObservers(savedBooking);
    return savedBooking;
}
```

Рис. 6 ServicePackage агрегує основну та додаткові послуги (BookingService.java)

4. Управління статусом (State)

Момент: Адміністратор натискає "Підтвердити", а потім Клієнт натискає "Оплатити". Дія: Об'єкт Booking не просто змінює поле в базі. Він делегує цю дію своєму поточному Стану.

- Поточний стан: PendingState. Виклик confirm() переводить його в ConfirmedState.
- Поточний стан: ConfirmedState. Виклик pay() дозволений і переводить його в PaidState.

```
public void confirm() {
    state.confirm(this);
}

public void pay() {
    state.pay(this);
}

public void cancel() {
    state.cancel(this);
}
```

Рис. 7 Делегування поточному об'єкту стану (Booking.java)

```
public void setStatus(BookingStatus status) {
    this.status = status;
    initState();
}
```

Рис. 8 Зміна стану (Booking.java)

- Спроба порушення: Якщо клієнт спробує оплатити PendingState (непідтверджене), об'єкт стану кине помилку "Оплата заборонена". Результат: Чіткий контроль бізнес-логіки переходів.

5. Проведення оплати (Facade + Strategy)

Момент: Клієнт вводить дані картки і натискає "Підтвердити оплату". Дія:

- Контролер звертається до Фасаду (PaymentFacade), який є єдиною точкою входу для фінансів.
- Фасад перевіряє через State, чи можна платити.
- Фасад звертається до Фабрики Стратегій і отримує потрібну Стратегію (CardPaymentStrategy).

- Стратегія виконує "списання" (логіку оплати).
- Фасад зберігає чек у БД.
- Фасад знову смикає Observer, щоб надіслати лист "Оплата успішна". Результат: Складна операція виконана одним простим викликом.

```
@Transactional
public Booking payForBooking(UUID bookingId, String paymentMethod, String cardNumber) {
    Booking booking = bookingRepository.findById(bookingId)
            .orElseThrow(() -> new RuntimeException(message: "Бронювання не знайдено."));

    booking.pay();

    PaymentStrategy strategy = strategyFactory.getStrategy(paymentMethod);

    if (!strategy.processPayment(booking.getTotalPrice(), cardNumber)) {
        throw new RuntimeException("Зовнішній платіж " + strategy.getId() + " не вдалося виконати.");
    }

    Payment payment = new Payment();
    payment.setBooking(booking);
    payment.setAmount(booking.getTotalPrice());
    payment.setPaymentMethod(strategy.getId());
    payment.setPaymentStatus(paymentStatus: "PAID");
    payment.setPaymentDate(LocalDateTime.now());
    payment.setCardNumber(cardNumber);
    paymentRepository.save(payment);

    Booking savedBooking = bookingRepository.save(booking);
    bookingService.notifyPaymentObservers(savedBooking);
    return savedBooking;
}
```

Рис. 9 PaymentFacade.java

Реалізація патерну Facade та Strategy. Клас PaymentFacade надає спрощений інтерфейс для оплати, координуючи роботу репозиторіїв, станів та спостерігачів. Вибір конкретного алгоритму оплати здійснюється через патерн Strategy (paymentStrategyFactory).

4. Висновок

У ході лабораторної роботи було реалізовано клієнт-серверну архітектуру в проєкті Beauty Salon Booking System на базі Spring Boot. Серверна частина виконує всю бізнес-логіку та обробку даних (MySQL, Spring Data JPA), успішно застосовуючи патерни Chain of Responsibility (валідація бронювання), Composite (розрахунок вартості пакетів), State (життєвий цикл запису), Observer (email-сповіщення), Facade та Strategy (оплата). Клієнтська частина — тонкий клієнт з Server-Side Rendering (Thymeleaf) та мінімальним JavaScript. Завдяки чіткому розподілу відповідальностей та багатошаровій структурі отримано повноцінний, зручний у супроводі веб-додаток для бронювання послуг салону краси, що

демонструє ефективне використання клієнт-серверної моделі та класичних патернів проєктування.

5. Контрольні питання

1. Що таке клієнт-серверна архітектура? Клієнт-серверна архітектура — це модель взаємодії, де клієнтські програми або пристрої надсилають запити до сервера, який обробляє їх і повертає результат. Клієнт відповідає за інтерфейс користувача, а сервер — за обробку даних, логіку та зберігання.

2. Розкажіть про сервіс-орієнтовану архітектуру (SOA). SOA — це архітектурний стиль, за якого програма складається з окремих сервісів, кожен з яких виконує певну бізнес-функцію. Сервіси взаємодіють між собою через стандартизовані інтерфейси (наприклад, HTTP, SOAP, REST), що забезпечує гнучкість та масштабованість системи.

3. Якими принципами керується SOA? Основні принципи SOA:

- Автономність: Сервіси функціонують незалежно один від одного.
- Стандартизація: Використання уніфікованих інтерфейсів для взаємодії.
- Повторне використання: Сервіси можуть застосовуватися в різних частинах системи.
- Інтеграція: Легкість об’єднання сервісів у складні процеси.
- Слабке зв’язування: Зміни в одному сервісі мінімально впливають на інші.

4. Як між собою взаємодіють сервіси в SOA? Сервіси спілкуються через стандартизовані протоколи (SOAP, REST) або повідомлення. Кожен сервіс надає опис свого інтерфейсу (наприклад, WSDL або OpenAPI), що дозволяє іншим сервісам викликати його методи та обмінюватися даними у форматах XML, JSON тощо.

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

- Реєстр сервісів: Централізована база даних, де зареєстровані всі доступні сервіси та їхні інтерфейси.
- Документація API: Наприклад, OpenAPI/Swagger, яка описує методи та формати даних.
- Запити: Відбуваються через стандартні протоколи (HTTP, SOAP, REST) за адресою сервісу.

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

- Централізоване управління даними та безпекою.
- Легкість масштабування серверної частини.
- Прості клієнти, оскільки основна логіка знаходиться на сервері.

Недоліки:

- Сервер може стати «вузьким місцем» (single point of failure).
- Високі вимоги до потужності сервера при великому навантаженні.
- Залежність клієнтів від сервера — без зв’язку система не працює.

7. У чому полягають переваги та недоліки однорангової (peer-to-peer) моделі?
Переваги:

- Відсутність центрального сервера, що підвищує стійкість системи.
- Прямий обмін ресурсами між учасниками мережі.
- Горизонтальне масштабування — додавання вузлів збільшує потужність.

Недоліки:

- Складність забезпечення безпеки та контролю доступу.
- Кожен вузол відповідає за управління власними ресурсами.
- Важко синхронізувати дані та координувати оновлення.

8. Що таке мікросервісна архітектура? Мікросервісна архітектура — це підхід, за якого додаток складається з дрібних, незалежних сервісів, кожен з яких виконує одну бізнес-функцію. Сервіси можуть розгортатися окремо, що забезпечує гнучкість та масштабованість системи.

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

- HTTP/HTTPS + REST
- gRPC
- SOAP
- Message brokers: RabbitMQ, Kafka, MQTT (для асинхронної взаємодії)
- WebSockets (для роботи в реальному часі)

10. Чи можна назвати підхід сервіс-орієнтованою архітектурою, якщо в проєкті між веб-контролерами та шаром доступу до даних реалізовано шар бізнес-логіки у вигляді сервісів? Це не повноцінна SOA, а внутрішня організація коду за допомогою сервісного шару (Service Layer). SOA передбачає автономні сервіси, доступні для

зовнішніх систем через стандартизовані протоколи, тоді як ваш підхід стосується лише внутрішньої структури одного додатку.