

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»

Виконав:
Студент групи IA-34
Цап Андрій

Перевірив:
Мягкий Михайло Юрійович

Київ – 2025

Зміст

Зміст	1
Вступ	2
Теоретичні відомості	3
Клієнт-серверна архітектура	4
Peer-to-Peer архітектура	4
Сервіс-орієнтована архітектура	4
Мікро-сервісна архітектура	4
Хід роботи.....	5
Діаграма класів спроектованої архітектури:	6
Вихідні коди.....	7
Висновки.....	11
Контрольні запитання.....	12

Вступ

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

У даній лабораторній роботі було досліджено основні моделі взаємодії застосунків та методи організації обміну даними між компонентами. У межах реалізованої системи керування проєктами було обрано клієнт-серверну архітектуру як найбільш відповідну до структури платформи та характеру її функціональних можливостей.

Серверна частина представлена Spring Boot застосунком, який надає доступ до ресурсів системи через REST-інтерфейс. Клієнтська частина взаємодіє з сервером шляхом виконання HTTP-запитів і отримання даних у форматі JSON. Такий підхід дозволяє чітко відокремити логіку представлення даних від логіки їх обробки та забезпечує можливість подальшого масштабування проєкту.

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.

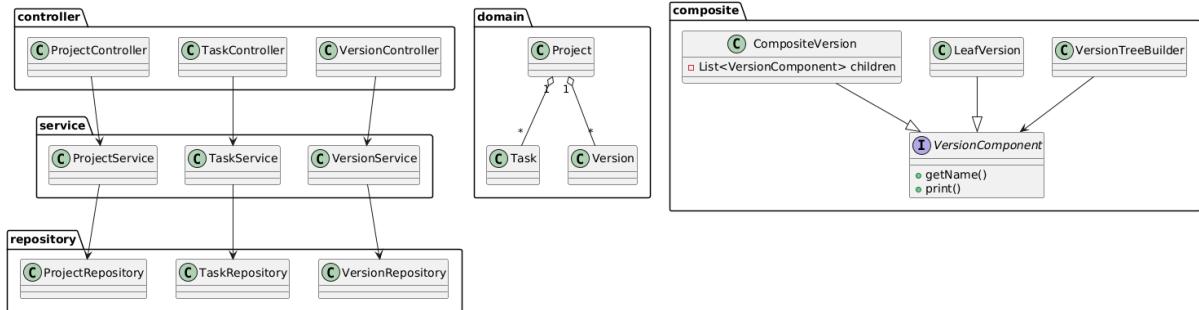
Peer-to-Peer архітектура - Peer-to-Peer (P2P) архітектура – це модель мережевої взаємодії, в якій кожен вузол (комп'ютер або пристрій) є одночасно клієнтом і сервером. У цій архітектурі всі вузли мають рівні права та можливості для обміну даними, ресурсами або виконання завдань. На відміну від клієнт-серверної моделі, де є чітке розділення на клієнти й сервери, P2P-мережа дозволяє учасникам взаємодіяти безпосередньо, без необхідності в централізованому сервері.

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.

Мікро-сервісна архітектура - 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.

Хід роботи

Діаграма класів спроектованої архітектури:



На UML-діаграмі зображене архітектуру багаторівневої системи управління проектами, що складається з чітко розділених компонентів: controller, service, repository, domain. Така структура наслідує класичний підхід MVC/Service-Layer, забезпечує слабку зв'язаність елементів та полегшує розширення функціоналу.

У шарі controller розташовані три основні компоненти: ProjectController, TaskController та VersionController. Кожен з них відповідає за обробку HTTP-запитів, взаємодіє зі своїм відповідним сервісом та передає дані у вигляді моделей до шаблонів інтерфейсу. Контролери не містять бізнес-логіки, виконуючи роль тонких посередників.

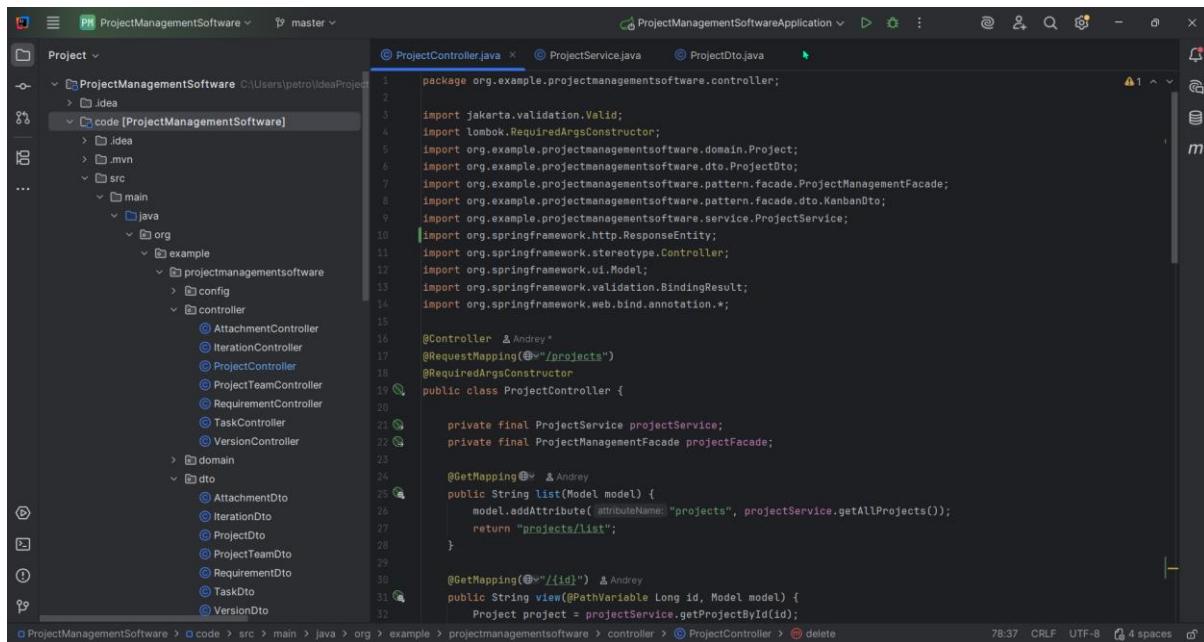
Сервісний шар представлений класами ProjectService, TaskService та VersionService. Саме тут зосереджена бізнес-логіка, перевірка даних, робота з файлами, обробка залежностей та підготовка даних для контролерів. Кожен сервіс звертається до свого відповідного репозиторію, забезпечуючи повне розмежування відповідальностей.

Репозиторій шар (ProjectRepository, TaskRepository, VersionRepository) забезпечує взаємодію з базою даних. Кожен репозиторій інкапсулює операції CRUD для відповідної сутності, зберігаючи систему розширеною та незалежною від конкретної реалізації бази.

У шарі domain зображені основні сутності системи: Project, Task та Version. На діаграмі видно ключові зв'язки: проєкт може містити декілька задач та декілька версій (зв'язок один-до-багатьох). Таке моделювання відображає реальну структуру даних у системі та забезпечує коректну роботу всіх сервісів, які працюють із проєктною інформацією.

Уся система демонструє добре структуровану архітектуру: контролери відповідають за представлення, сервіси за логіку, репозиторії за дані, доменні моделі за структуру інформації, а модуль Composite забезпечує універсальне управління ієрархічними версіями. Така організація робить програму масштабованою, зрозумілою й легко підтримуваною.

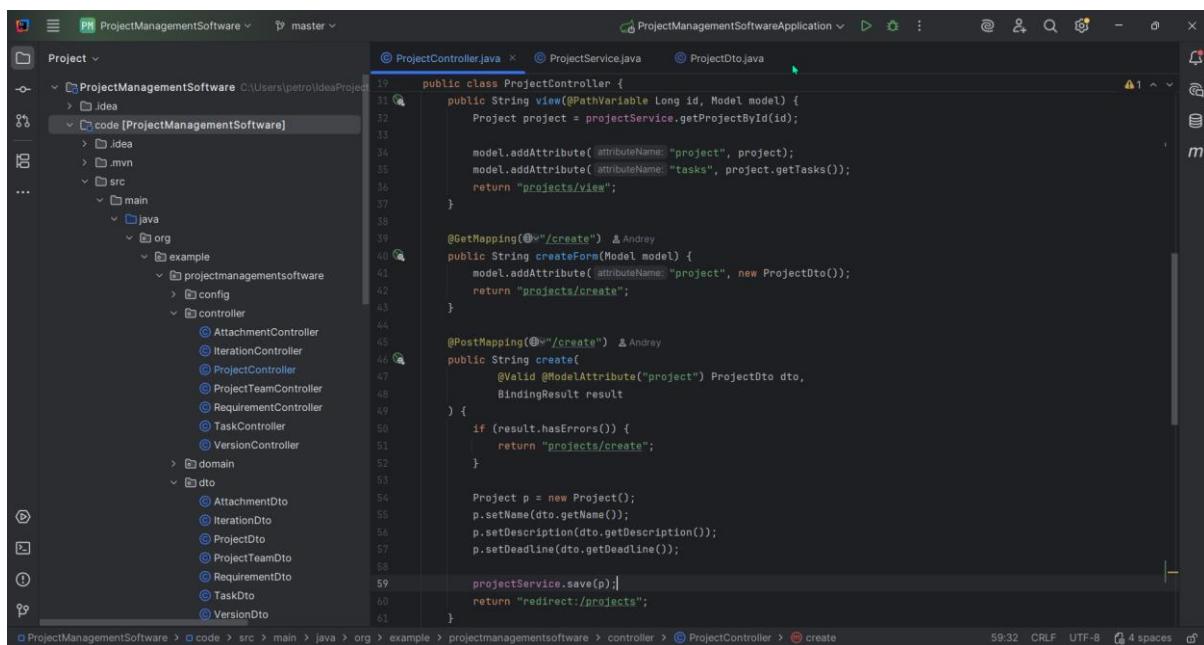
Вихідні коди



```

1 package org.example.projectmanagementsoftware.controller;
2
3 import jakarta.validation.Valid;
4 import lombok.RequiredArgsConstructor;
5 import org.example.projectmanagementsoftware.domain.Project;
6 import org.example.projectmanagementsoftware.dto.ProjectDto;
7 import org.example.projectmanagementsoftware.pattern.facade.ProjectManagementFacade;
8 import org.example.projectmanagementsoftware.service.ProjectService;
9 import org.springframework.http.ResponseEntity;
10 import org.springframework.stereotype.Controller;
11 import org.springframework.ui.Model;
12 import org.springframework.validation.BindingResult;
13 import org.springframework.web.bind.annotation.*;
14
15 @Controller
16 @Andrey
17 @RequestMapping("/projects")
18 @RequiredArgsConstructor
19 public class ProjectController {
20
21     private final ProjectService projectService;
22     private final ProjectManagementFacade projectFacade;
23
24     @GetMapping
25     public String list(Model model) {
26         model.addAttribute("projects", projectService.getAllProjects());
27         return "projects/list";
28     }
29
30     @GetMapping("/{id}")
31     public String view(@PathVariable Long id, Model model) {
32         Project project = projectService.getProjectById(id);
33         model.addAttribute("project", project);
34         model.addAttribute("tasks", project.getTasks());
35         return "projects/view";
36     }
37
38     @GetMapping("/create")
39     public String createForm(Model model) {
40         model.addAttribute("project", new ProjectDto());
41         return "projects/create";
42     }
43
44     @PostMapping("/create")
45     public String create(
46         @Valid @ModelAttribute("project") ProjectDto dto,
47         BindingResult result
48     ) {
49
50         if (result.hasErrors()) {
51             return "projects/create";
52         }
53
54         Project p = new Project();
55         p.setName(dto.getName());
56         p.setDescription(dto.getDescription());
57         p.setDeadline(dto.getDeadline());
58
59         projectService.save(p);
60         return "redirect:/projects";
61     }
62
63 }

```



```

public class ProjectController {
    ...
    @GetMapping("/edit/{id}")
    public String editForm(@PathVariable Long id, Model model) {
        model.addAttribute("project", projectService.getProjectById(id));
        return "projects/edit";
    }

    @PostMapping("/edit/{id}")
    public String update(@PathVariable Long id, @ModelAttribute Project project) {
        projectService.update(id, project);
        return "redirect:/projects/" + id;
    }

    @PostMapping("/delete/{id}")
    public String delete(@PathVariable Long id) {
        projectService.delete(id);
        return "redirect:/projects";
    }

    @GetMapping("/{projectId}/board")
    public String kanban(@PathVariable Long projectId, Model model) {
        KanbanDto board = projectFacade.getKanban(projectId);
        model.addAttribute("board", board);
        return "projects/board";
    }

    @GetMapping("/api/projects/{id}")
    public ResponseEntity<Project> getProject(@PathVariable Long id) {
        return ResponseEntity.ok(projectService.getProjectById(id));
    }
}

```

Рис 1,2,3 - Код ProjectController

```

package org.example.projectmanagementsoftware.service;

import lombok.RequiredArgsConstructor;
import org.example.projectmanagementsoftware.domain.Project;
import org.example.projectmanagementsoftware.exception.NotFoundException;
import org.example.projectmanagementsoftware.pattern.observer.enums.ProjectEventType;
import org.example.projectmanagementsoftware.pattern.observer.interfaces.ObservableProject;
import org.example.projectmanagementsoftware.pattern.observer.interfaces.ProjectObserver;
import org.example.projectmanagementsoftware.pattern.observer.utils.ProjectUpdateChecker;
import org.example.projectmanagementsoftware.repository.ProjectRepository;
import org.springframework.stereotype.Service;

import java.util.List;

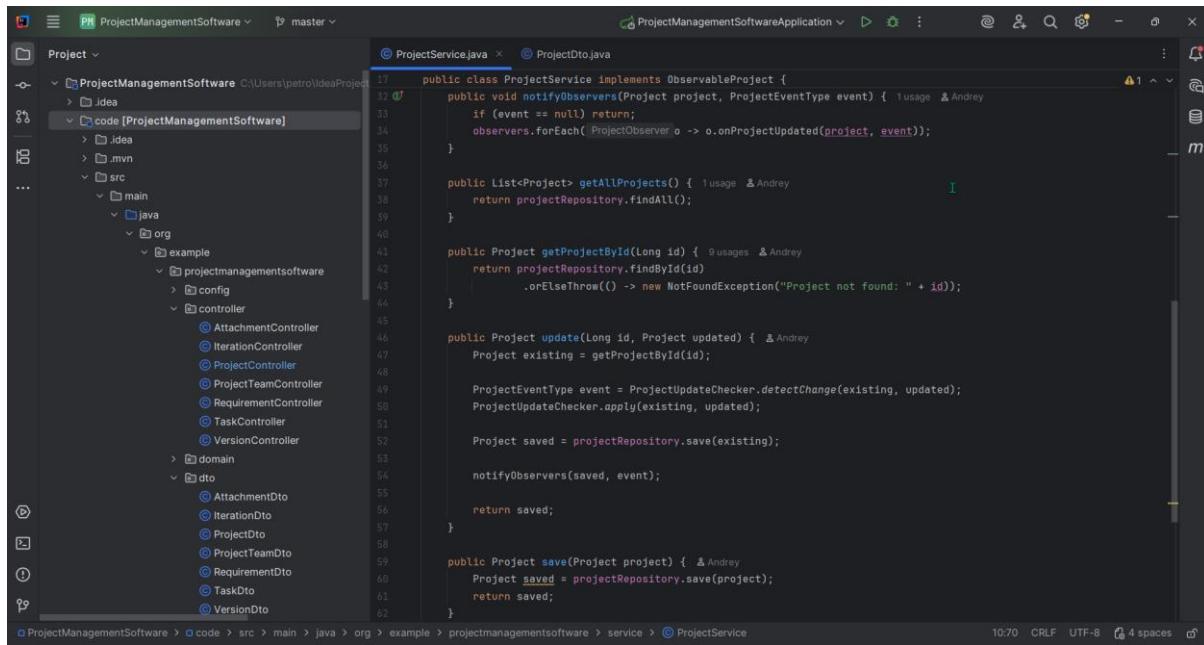
@Service
@RequiredArgsConstructor
public class ProjectService implements ObservableProject {
    ...
    private final ProjectRepository projectRepository;
    private final List<ProjectObserver> observers;

    @Override
    public void registerObserver(ProjectObserver observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(ProjectObserver observer) {
        observers.remove(observer);
    }

    public void notifyObservers(Project project, ProjectEventType event) {
    }
}

```



The screenshot shows the IntelliJ IDEA interface with the ProjectManagementSoftware project open. The left sidebar displays the project structure, including modules like ProjectManagementSoftware, sub-modules like code, and source code packages such as main.java.org.example.projectmanagementsoftware.controller. The right pane shows the code editor for ProjectService.java, which implements ObservableProject. The code includes methods for notifying observers, getting all projects, getting a project by ID, updating a project, saving a project, and deleting a project. The code editor shows syntax highlighting and some annotations.

```

public class ProjectService implements ObservableProject {
    public void notifyObservers(Project project, ProjectEventType event) { usage & Andrey
        if (event == null) return;
        observers.forEach( ProjectObserver o -> o.onProjectUpdated(project, event));
    }

    public List<Project> getAllProjects() { usage & Andrey
        return projectRepository.findAll();
    }

    public Project getProjectById(Long id) { usage & Andrey
        return projectRepository.findById(id)
            .orElseThrow(() -> new NotFoundException("Project not found: " + id));
    }

    public Project update(Long id, Project updated) { & Andrey
        Project existing = getProjectById(id);

        ProjectEventType event = ProjectUpdateChecker.detectChange(existing, updated);
        ProjectUpdateChecker.apply(existing, updated);

        Project saved = projectRepository.save(existing);

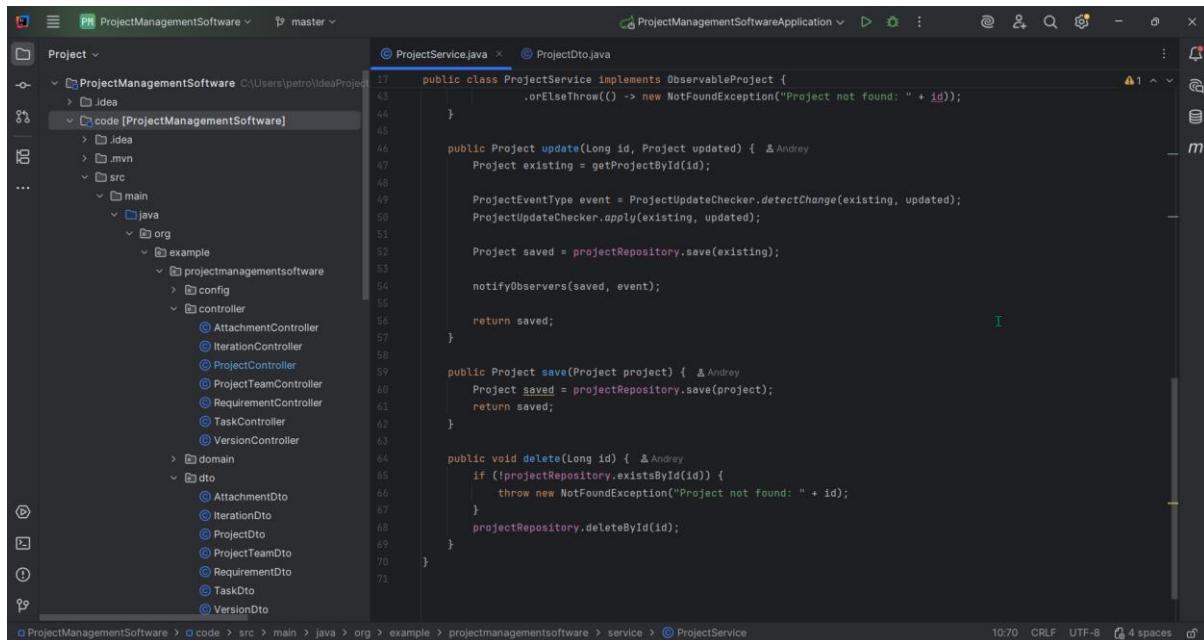
        notifyObservers(saved, event);

        return saved;
    }

    public Project save(Project project) { & Andrey
        Project saved = projectRepository.save(project);
        return saved;
    }

    public void delete(Long id) { & Andrey
        if (!projectRepository.existsById(id)) {
            throw new NotFoundException("Project not found: " + id);
        }
        projectRepository.deleteById(id);
    }
}

```

This screenshot shows the same IntelliJ IDEA interface as the first one, but the code editor is displaying the ProjectService.java code again. The code is identical to the one shown in the first screenshot, implementing ObservableProject and providing methods for project management operations.

Рис 4,5,6 - Код ProjectService

The screenshot shows the IntelliJ IDEA interface with the ProjectManagementSoftware application open. The left sidebar displays the project structure under 'Project'. The 'src' directory contains 'main', 'java', 'org', and 'example'. In the 'example' directory, there is a 'projectmanagementsoftware' package which contains several controller classes: AttachmentController, IterationController, ProjectController, ProjectTeamController, RequirementController, TaskController, and VersionController. Below these are 'domain', 'dto', and other sub-directories. The 'dto' directory contains 'AttachmentDto', 'IterationDto', 'ProjectDto', 'ProjectTeamDto', 'RequirementDto', 'TaskDto', and 'VersionDto'. The 'ProjectDto.java' file is currently selected and open in the main editor window. The code defines a class 'ProjectDto' with fields: 'name' (with validation annotations @NotBlank and @Setter), 'description' (with validation annotations @NotBlank and @Setter), and 'deadline' (with validation annotations @NotNull, @Future, and @DateTimeFormat). The code uses Lombok annotations @Getter and @Setter.

```
1 package org.example.projectmanagementsoftware.dto;
2
3 import jakarta.validation.constraints.Future;
4 import jakarta.validation.constraints.NotBlank;
5 import jakarta.validation.constraints.NotNull;
6 import lombok.Getter;
7 import lombok.Setter;
8 import org.springframework.format.annotation.DateTimeFormat;
9
10 import java.time.LocalDate;
11
12 @Setter 3 usages & Andrey
13 @Setter
14 public class ProjectDto {
15
16     @NotBlank(message = "Название не может быть пустым")
17     private String name;
18
19     @NotBlank(message = "Описание не может быть пустым")
20     private String description;
21
22     @NotNull(message = "Дата дедлайна с обозначением")
23     @Future(message = "Дедлайн не может быть в прошлом")
24     @DateTimeFormat(pattern = "yyyy-MM-dd")
25     private LocalDate deadline;
26
27 }
28
29 }
```

Рис 7 - Код ProjectDto

Висновки

У процесі виконання лабораторної роботи були розглянуті принципи взаємодії застосунків у розподілених системах та проаналізовані три основні архітектурні підходи: Client–Server, P2P та Service-Oriented Architecture. На основі структури розробленої системи було визначено, що клієнт-серверна модель є найбільш доцільною для впровадження.

Реалізація REST-взаємодії дозволила створити чіткий розподіл між клієнтською та серверною частинами, організувати обмін даними у стандартизованому форматі та забезпечити гнучкість у розширенні функціоналу. Додавання окремих ендпоїнтів та демонстрація їхньої роботи підтвердили відповідність системи вимогам розподіленого середовища.

Контрольні запитання

1. Що таке клієнт-серверна архітектура?

Модель, де клієнт надсилає запити, а сервер виконує логіку, працює з даними і відповідає.

2. Розкажіть про сервіс-орієнтовану архітектуру.

SOA — підхід, де система складається з незалежних сервісів. Кожен сервіс виконує окрему функцію, має чіткий інтерфейс і взаємодіє з іншими через стандартизовані протоколи.

3. Якими принципами керується SOA?

- слабке зв'язування
- повторне використання сервісів
- чітко визначені контракти
- незалежність реалізації
- доступність через стандартизовані інтерфейси
- можливість комбінування сервісів у більші процеси

4. Як між собою взаємодіють сервіси в SOA?

Через стандартизовані протоколи: SOAP, REST, XML/JSON. Один сервіс надсилає запит іншому через його публічний інтерфейс.

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

Через реєстр сервісів, документацію або опис контрактів. Виклик відбувається через URL або ендпоїнт сервісу.

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

Переваги:

- централізоване управління даними
- безпека
- масштабованість
- розділення обов'язків

Недоліки:

- залежність від сервера
- навантаження концентрується в одній точці
- може бути "вузьке місце" при великому трафіку

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

Переваги:

- немає центральної точки відмови
- краща масштабованість у плані ресурсів
- кожен вузол може бути і клієнтом, і сервером

Недоліки:

- складніше забезпечити безпеку
- важко контролювати мережу
- непередбачуваність доступності вузлів

8. Що таке мікро-сервісна архітектура?

Підхід, де додаток розбивається на дуже дрібні незалежні сервіси. Кожен мікросервіс реалізує вузьку функцію, має свою базу даних, розгортається і масштабується окремо.

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

- HTTP/REST
- gRPC
- WebSockets
- AMQP, Kafka

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

Ні, це просто шар бізнес-логіки всередині моноліту. SOA — це архітектура між окремими сервісами, а не між шарами в одному проекті.