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

Лабораторна робота №9
Технології розроблення програмного забезпечення
Різні види взаємодії додатків: CLIENT-SERVER, PEER-TO-PEER, SERVICE-ORIENTED ARCHITECTURE
CI-server

Виконав:
Студент групи ІА-22
Вдовиченко А.Х.

Перевірив:
Мягкий М. Ю

Київ 2024

Зміст

Тема: Різні види взаємодії додатків CLIENT-SERVER, PEER-TO-PEER, SERVICE-ORIENTED ARCHITECTURE
Мета: ознайомитись з теоретичними відомостями, реалізувати один з видів взаємодії між додатками.

Теоретичні відомості

Клієнт-серверна модель передбачає взаємодію між двома основними компонентами: клієнтом і сервером. Клієнт відповідає за взаємодію з користувачем і надсилання запитів до сервера, а сервер займається обробкою цих запитів, зберіганням даних та виконанням основної логіки. Залежно від розподілу функцій, клієнт може бути тонким або товстим. Тонкий клієнт відправляє більшість обчислювальних операцій на сервер, залишаючи собі лише відображення інформації, тоді як товстий виконує значну частину обчислень локально. Основною перевагою клієнт-серверної архітектури є централізоване управління, що спрощує адміністрування і зменшує ризик конфліктів даних. Проте високі навантаження на сервер і залежність від стабільності з'єднання можуть бути проблемою.

Однорангові мережі (peer-to-peer) побудовані на принципі рівноправності всіх учасників системи. У них немає центрального сервера, і клієнтські програми безпосередньо взаємодіють одна з одною для виконання спільних завдань. Основними викликами таких мереж є пошук клієнтів і синхронізація даних між ними. Для цього використовуються централізовані адресні списки або алгоритми обміну даними. Цей підхід добре підходить для децентралізованих систем, але може викликати труднощі у масштабуванні та забезпеченні стабільності зв'язку.

Сервіс-орієнтована архітектура (SOA) використовує модульний підхід, де програмне забезпечення розробляється як набір взаємодіючих служб. Кожна служба має стандартизований інтерфейс і виконує конкретну функцію. Основною перевагою є гнучкість, що дозволяє інтегрувати різні компоненти незалежно від їхніх платформ чи мов програмування. SOA широко використовується для побудови складних розподілених систем, таких як веб-сервіси або хмарні платформи. Однак розробка таких систем може бути складною через необхідність забезпечення сумісності між компонентами.

Модель SaaS (програмне забезпечення як послуга), яка ґрунтується на принципах SOA, дозволяє користувачам отримувати доступ до програм через Інтернет, спрощуючи управління і знижуючи витрати на обслуговування. Це економічно вигідно, оскільки користувач платить тільки за використання послуги, а не за володіння програмним забезпеченням.

Архітектура мікросервісів, яка є сучасною інтерпретацією SOA, дозволяє створювати серверні додатки як набір незалежних служб, кожна з яких відповідає за виконання специфічної функції. Це забезпечує високу масштабованість і полегшує супроводження великих проектів, але вимагає належного рівня організації обміну даними між компонентами.

Хід роботи

Я використав клієнт-серверну архітектуру для реалізації проєкту, оскільки вона забезпечує простоту реалізації, централізоване управління даними і зручну організацію взаємодії між компонентами. У нашому випадку клієнт відповідає за інтерфейс користувача та передачу запитів до сервера, тоді як сервер обробляє ці запити, виконує бізнес-логіку та працює з базою даних.

Переваги SOA (сервіс-орієнтованої архітектури):

- Гнучкість інтеграції різних компонентів завдяки стандартизованим інтерфейсам.
- Підходить для складних і масштабованих систем із великою кількістю незалежних служб.
- Легкість повторного використання та модифікації окремих сервісів.

Однак у нашому проєкті використання SOA було б недоцільним через кілька причин:

- Відносно невелика складність системи: Наша система не є складною розподіленою системою з багатьма незалежними сервісами. Основна функціональність зосереджена у взаємодії клієнта з сервером, що робить використання SOA надмірним.
- Висока складність впровадження SOA: Розробка та інтеграція сервісів із стандартизованими інтерфейсами вимагає більше ресурсів і часу. Це може бути виправдано для великих проєктів, але не для порівняно простого клієнт-серверного додатка.
- Невиправдані витрати на підтримку: SOA вимагає більш складного налаштування й підтримки, особливо в частині забезпечення сумісності між сервісами. У нашому випадку це б лише ускладнило архітектуру без вагомої користі.

Натомість клієнт-серверна архітектура:

- Простіш у реалізації, що дозволило швидко запустити прототип системи.
- Забезпечує централізоване управління даними на сервері, що спрощує адміністрування.

- Дозволяє легко масштабувати серверну частину, якщо зростуть вимоги до продуктивності.

Таким чином, клієнт-серверна архітектура стала оптимальним вибором для реалізації завдань проєкту, оскільки вона відповідає його масштабам, забезпечує потрібний рівень гнучкості та є більш ефективною з точки зору ресурсів.

Для реалізації клієнт-серверної архітектури було створено веб-додаток на основі React для фронтенд-частини та Node.js із використанням Express для серверної частини. Система підтримує управління проектами, їх створення, перегляд та запуск збірок. Для взаємодії між клієнтом і сервером використовується REST API та Websocket.

Спочатку було ініціалізовано React-додаток за допомогою create-react-app. Додано бібліотеки axios для HTTP-запитів та react-router-dom для маршрутизації. Серверна частина налаштована на Node.js з використанням Express для обробки REST-запитів та підключення до бази даних.

Клієнтська частина містить декілька основних компонентів. Компонент ProjectsPage.js відображає список проектів і дозволяє створювати нові проекти, вводячи назву та посилання на репозиторій. Компонент ProjectDetailsPage.js відображає деталі проекту, включаючи список гілок і доступні збірки, а також дозволяє запускати нові збірки. Компонент BuildLogsPage.js показує логи виконання збірок у реальному часі, використовуючи WebSocket. (Рис 1.1.)

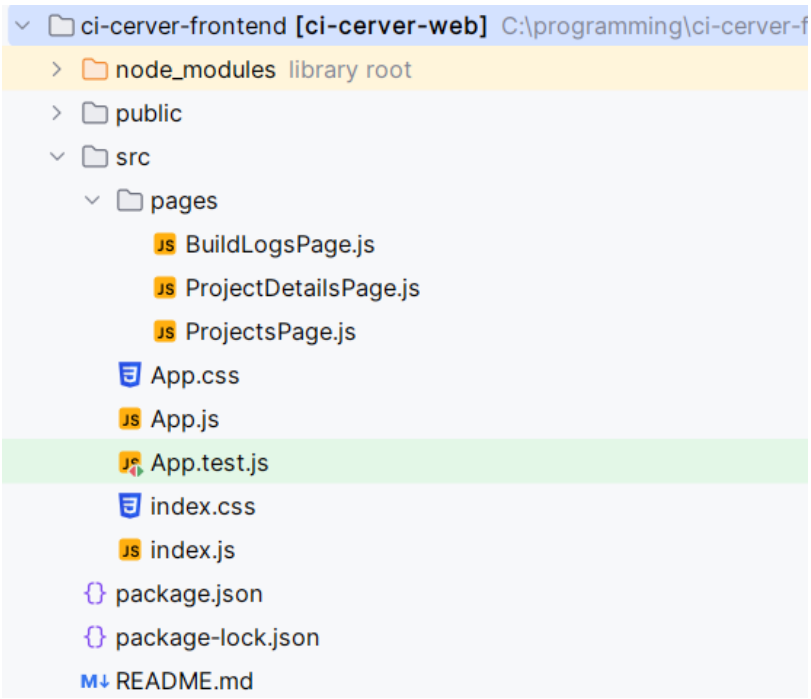

Рис. 1.1. – Структура проекту ci-server-frontend

Обмін даними між клієнтом і сервером здійснюється через HTTP-запити, реалізовані за допомогою axios. Наприклад, для отримання списку проектів використовується метод GET за адресою "http://localhost:8080/projects". Створення нового проекту відбувається через POST-запит до "http://localhost:8080/projects" із передачею JSON-об'єкта з полями name і repositoryUrl. (Рис 1.2.)

Рис 1.2 – Сторінка доступних проектів

Для реалізації роботи з логами збірок було налаштовано WebSocket-з'єднання. Це забезпечує отримання логів у режимі реального часу, що дозволяє оперативно відстежувати процес виконання збірок. Логи відображаються у вигляді прокручуваного списку для зручності перегляду. (Рис 1.3)

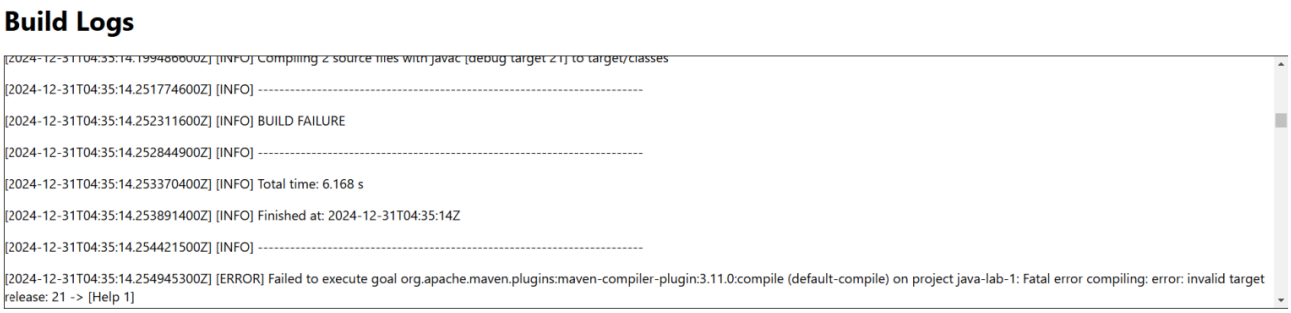

Рис 1.3 – Сторінка перегляду логів збірки

Серверна частина використовує PostgreSQL для зберігання даних. Підключення до бази даних здійснюється через Panache ORM, що забезпечує зручність роботи з таблицями та записами. На сервері обробляються запити для створення, отримання та оновлення проектів і збірок. Крім того, реалізовано потокове передавання даних логів через WebSocket.

Робота додатку тестувалася для перевірки функціоналу. Тестування основних функцій, таких як створення проектів, перегляд деталей, запуск збірок і перегляд логів, дозволило виявити та виправити помилки. Для тестування компонентів React використовувалася бібліотека Jest. Особлива увага приділялася налагодженню роботи WebSocket для забезпечення коректного обміну даними в режимі реального часу.

У результаті розроблено та протестовано клієнт-серверний веб-додаток для управління проектами та їх збірками. Використання сучасних технологій, таких як React, Node.js, WebSocket та REST API, забезпечує масштабованість і зручність у користуванні.

Висновки

У цій лабораторній роботі було реалізовано клієнт-серверну архітектуру для CI серверу з використанням REST API та Websocket. У цій роботі досліджено різні моделі взаємодії додатків. Клієнт-серверна модель забезпечує централізоване управління даними, P2P пропонує децентралізований підхід, а SOA дозволяє інтегрувати модулі в складні системи. Робота з цими архітектурами допомогла зрозуміти, як вибирати підхід залежно від складності системи, забезпечуючи гнучкість, масштабованість і ефективність.

Вихідний код

```
import React, { useState, useEffect, useRef } from "react";
import { useParams } from "react-router-dom";
import axios from "axios";

const BuildLogsPage = () => {
    const { id, buildId } = useParams();
    const [logs, setLogs] = useState([]);
    const [status, setStatus] = useState("");
    const socketRef = useRef(null);

    useEffect(() => {
        axios.get(`http://localhost:8080/builds/${buildId}/logs`).then(response
=> setLogs(response.data.split("\n")));

        socketRef.current = new WebSocket(`ws://localhost:8080/project-
builds/${buildId}`);
        socketRef.current.onmessage = event => {
            setLogs(prevLogs => [...prevLogs, event.data]);
        };

        socketRef.current.onopen = () => {
            axios.get(`http://localhost:8080/builds/${buildId}`).then(response
=> setStatus(response.data.buildStatus));
        };

        return () => socketRef.current.close();
    }, [id, buildId]);

    return (
        <div>
            <h1>Build Logs</h1>
            <div style={{ height: "300px", overflowY: "scroll", border: "1px
solid black" }}>
                {logs.map((log, index) => (
                    <p key={index}>{log}</p>
                ))}
            </div>
        </div>
    );
};

export default BuildLogsPage;


import React, { useState, useEffect } from "react";
import { useParams, useNavigate } from "react-router-dom";
import axios from "axios";

const ProjectDetailsPage = () => {
    const { id } = useParams();
    const [branches, setBranches] = useState([]);
    const [selectedBranch, setSelectedBranch] = useState("");
```

```
    const [builds, setBuilds] = useState([]);
    const navigate = useNavigate();

    useEffect(() => {
        axios.get(`http://localhost:8080/projects/${id}/branches`).then(response
=> setBranches(response.data));
        axios.get(`http://localhost:8080/projects/${id}/builds`).then(response
=> setBuilds(response.data));
    }, [id]);

    const createBuildAndStart = () => {
        axios.post(`http://localhost:8080/projects/${id}/builds`, null, {
params: { branch: selectedBranch } }).then(response => {
            const buildId = response.data.id;
            const socket = new WebSocket(`ws://localhost:8080/project-
builds/${buildId}`);
            socket.onopen = () => {
                socket.send("start");
                navigate(`/projects/${id}/builds/${buildId}`);
            };
        });
    };

    return (
        <div>
            <h1>Project Details</h1>
            <h2>Branches</h2>
            <select value={selectedBranch} onChange={e =>
setSelectedBranch(e.target.value)}>
                <option value="" disabled>
                    Select a branch
                </option>
                {branches.map(branch => (
                    <option key={branch} value={branch}>
                        {branch}
                    </option>
                ))}
            </select>
            <button onClick={createBuildAndStart}
disabled={!selectedBranch}>Start Build</button>

            <h2>Builds</h2>
            <ul>
                {builds.map(build => (
                    <li key={build.id}>
                        <span>Build #{build.id} - {build.buildStatus}</span>
                        <button onClick={() =>
navigate(`/projects/${id}/builds/${build.id}`)}>View Logs</button>
                    </li>
                ))}
            </ul>
        </div>
    );
};

export default ProjectDetailsPage;


import React, { useState, useEffect } from "react";
import { useNavigate } from "react-router-dom";
import axios from "axios";

const ProjectsPage = () => {
    const [projects, setProjects] = useState([]);
    const [newProject, setNewProject] = useState({ name: "", repositoryUrl: ""
```

```
});
    const navigate = useNavigate();

    useEffect(() => {
        axios.get("http://localhost:8080/projects").then(response =>
setProjects(response.data));
    }, []);

    const createProject = () => {
        axios.post("http://localhost:8080/projects", newProject).then(response
=> {
            setProjects([...projects, response.data]);
            setNewProject({ name: "", repositoryUrl: "" });
        });
    };

    return (
        <div>
            <h1>Projects</h1>
            <ul>
                {projects.map(project => (
                    <li key={project.id}>
                        <span>{project.name}</span>
                        <button onClick={() =>
navigate(`/projects/${project.id}`)}>View</button>
                    </li>
                ))}
            </ul>
            <h2>Create Project</h2>
            <input
                type="text"
                placeholder="Project Name"
                value={newProject.name}
                onChange={e => setNewProject({ ...newProject, name:
e.target.value })}
            />
            <input
                type="text"
                placeholder="Repository URL"
                value={newProject.repositoryUrl}
                onChange={e => setNewProject({ ...newProject, repositoryUrl:
e.target.value })}
            />
            <button onClick={createProject}>Create</button>
        </div>
    );
};

export default ProjectsPage;


// App.js
import React from "react";
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
import ProjectsPage from "./pages/ProjectsPage";
import ProjectDetailsPage from "./pages/ProjectDetailsPage";
import BuildLogsPage from "./pages/BuildLogsPage";

const App = () => {
    return (
        <Router>
            <Routes>
                <Route path="/" element={<ProjectsPage />} />
                <Route path="/projects/:id" element={<ProjectDetailsPage />} />
                <Route path="/projects/:id/builds/:buildId"
```

```
element={<BuildLogsPage />} />
            </Routes>
        </Router>
    );
};

export default App
```