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

із дисципліни «**Технології розроблення програмного забезпечення**»

Тема **«Різні види взаємодії додатків: Client-Server, Peer-to-Peer, Service-Oriented Architecture»**

Виконав:

студент групи ІА–24

Яблонський Д.Б.

Перевірив:

Мягкий М.Ю.

Київ 2024

# Зміст

**Мета:** метою виконання лабораторної роботи є вивчення та практичне застосування таких архітектурних рішень, як Client-Server, Peer-to-Peer та Service-Oriented Architecture, для створення ефективних і гнучких програмних рішень. Ця лабораторна робота спрямована на розвиток навичок у використанні різних видів взаємодії додатків при розробці програмного забезпечення.

## Теоретичні відомості

Нижче подано теоретичні відомості щодо базових архітектурних рішень, які широко застосовуються у сфері програмної інженерії та розподілених систем. Такі архітектури визначають, яким чином компоненти системи взаємодіють між собою, як розподіляються ресурси, обчислювальні завдання та інформаційні потоки. Серед поширених рішень можна виділити архітектуру типу Client-Server, Peer-to-Peer (P2P) та Service-Oriented Architecture (SOA).

### Архітектура "Client-Server"

У цій архітектурі система поділяється на дві основні складові: клієнти (Client) та сервер(и) (Server). Сервер виступає центральним вузлом, що надає певний набір послуг (наприклад, доступ до баз даних, веб-сторінок, обчислювальних ресурсів), тоді як клієнти звертаються до нього із запитами. Такий підхід характеризується чітким розподілом ролей: сервер управляє ресурсами та логікою, а клієнт ініціює запити та отримує результати. Перевагами є централізований контроль, зручність адміністрування та безпеки, а недоліком може стати залежність від доступності та продуктивності сервера.

### Архітектура "Peer-to-Peer" (P2P)

На відміну від моделі клієнт-сервер, в P2P усі вузли мережі мають однаковий статус: кожен вузол може бути як клієнтом, так і сервером одночасно. Кожен "пір" може надавати та отримувати ресурси напряму від інших. Така децентралізована структура підвищує стійкість до відмов окремих вузлів, оскільки немає єдиного "центрального" сервера. Це сприяє масштабованості та зниженню навантаження на окремі точки, але ускладнює питання безпеки, пошуку ресурсів та керування версіями даних.

### Service-Oriented Architecture (SOA)

SOA ґрунтується на ідеї створення незалежних програмних компонентів, які реалізують конкретні послуги (сервіси) та взаємодіють через чітко визначені інтерфейси. Сервіси можуть бути розгорнуті на різних платформах та технологіях, але завдяки стандартизованим протоколам (наприклад, веб-сервісам) їх можна легко об’єднати для побудови складніших бізнес-процесів. SOA сприяє гнучкості та повторному використанню компонентів, знижує зв’язаність системи та дозволяє ефективно реагувати на зміни вимог шляхом додавання або заміни сервісів.

## Хід роботи

1. Ознайомитися з короткими теоретичними відомостями.
2. Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.
3. Застосування одного з розглянутих шаблонів при реалізації програми.

## Варіант 20

## Тема: Mind-mapping software

### Реалізація архітектури Client-Server

Побудова застосунку Mind-mapping software на основі архітектури «клієнт-сервер» є доцільним рішенням у випадку, коли внутрішня логіка роботи застосунку спирається на щільно пов’язані моделі. Якщо у застосунку невелика кількість моделей, але вони активно взаємодіють одна з одною, має сенс зупинитись на архітектурі «клієнт-сервер», оскільки буде простіше налагодити взаємодію між моделями в межах одного серверу, а розбиття застосунку на декілька сервісів призведе до ускладнення структури цього застосунку. Також винесення складної логіки й взаємодії цих моделей на серверну сторону допомагає уникнути надмірного навантаження на пристрої користувачів, потенційного зростання затримок та зниження продуктивності. Зі свого боку, клієнтський інтерфейс легкий, оперативний та інтерактивний, адже більшість опрацювань перенесено на серверну частину. Це означає, що кожен користувач, незалежно від свого пристрою чи географічного розташування, може зручно працювати з ментальними картами, отримуючи

стабільний і захищений доступ до актуальних даних.

Продемонструвати логіку архітектури можна на прикладі реєстрації користувача та завантаження його даних на головну сторінку.

```
return (
  <div className="register-page">
    <h2>Register</h2>
    <form onSubmit={handleSubmit} className="register-form">
      <input
        type="text"
        name="name"
        placeholder="Name"
        value={newUser.name}
        onChange={handleInputChange}
      />
      <input
        type="email"
        name="email"
        placeholder="Email"
        value={newUser.email}
        onChange={handleInputChange}
      />
      <input
        type="password"
        name="password"
        placeholder="Password"
        value={newUser.password}
        onChange={handleInputChange}
      />
      <button
        type="submit"
      >
        Register
      </button>
      {showHelperText && <p className="helper-text">Fields must not be empty</p>}
    </form>
    <div className="navigation">
      <Link href={"/"}>Back to main</Link>
      <Link href={"/login"}>Login</Link>
    </div>
  </div>
);
```

Рис. 1 - Форма реєстрації в компоненті RegisterPage

На рис. 1 зображено компонент, в якому користувач вводить дані для реєстрації

```
const handleSubmit = async (event: React.FormEvent) => {  Show usages
  event.preventDefault();

  try {
    for (let field of Object.values(newUser)) {
      if (field === "") {
        setShowHelperText(true);
        return;
      }
    }
    const res = await fetch('http://localhost:8080/api/register', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      credentials: 'include',
      body: JSON.stringify(newUser)
    });

    if (res.ok) {
      userContext?.login(+document.cookie.split("=")[1])
      setNewUser({
        name: "",
        email: "",
        password: "",
      });
      router.push('/');
    } else {
      console.log('Registration error');
    }
  } catch (error) {
    console.error('Network error:', error);
  }
};
```

Рис. 2 - Обробка даних реєстрації та надсилання запиту на сервер

Далі дані, введені користувачем обробляються та надсилаються на сервер у вигляді http запиту з використанням методу POST

```java
@RestController  yablonya
@RequestMapping("/api/")
@CrossOrigin(origins = "http://localhost:3000", allowCredentials = "true")
public class HomeController {
    private final UserService userService;  6 usages
    private static final Logger logger = LoggerFactory.getLogger(HomeController.class);  4 usages

    @Autowired  yablonya
    public HomeController(UserService userService) {
        this.userService = userService;
    }

    @PostMapping("/register")  yablonya
    public ResponseEntity<?> registerUser(@RequestBody UserRegistrationRequest request) {
        try {
            User registeredUser = userService.registerUser(request.getName(), request.getEmail(), request.getPassword());
            HttpHeaders newHeaders = userService.addUserIdToCookie(registeredUser);

            logger.info("User registered with email: {}", registeredUser.getEmail());

            return ResponseEntity.status(HttpStatus.CREATED).headers(newHeaders).body(registeredUser);
        } catch (IllegalArgumentException e) {
            logger.error("Registration error: {}", e.getMessage());
            return ResponseEntity.badRequest().body(e.getMessage());
        }
    }
```

Рис. 3 - Ендпоінт реєстрації на сервері

```java
@Service  10 usages  yablonya
public class UserService {
    private final UserRepository userRepository;  5 usages
    private final CookiePrototype userCookiePrototype = new CookiePrototype( name: "userId",  maxAge: 24 * 60 * 60);
    private static final Logger logger = LoggerFactory.getLogger(UserService.class);  7 usages

    @Autowired  yablonya
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public User registerUser(String name, String email, String password) {  1 usage  yablonya
        if (userRepository.findByEmail(email) != null) {
            throw new IllegalArgumentException("User with this email already exists.");
        }

        User user = createUser(name, email, password);
        User savedProfile = userRepository.save(user);
        logger.info("User profile created: {}", savedProfile.getEmail());

        return savedProfile;
    }
```

Рис. 4 - Метод сервісу, що виконує реєстрацію користувача

На сервері цей запит обробляється спочатку контролером, який визначає що це за запит, які дані були передані та як далі опрацьовувати ці дані. Потім сервіс виконує всю бізнес логіку і повертає результат, якщо це потрібно. Цю відповідь контролер повертає клієнту у вигляді об’єкта HTTP Response.

```
useEffect(() => {
  const fetchUser = async () => {  Show usages  yablonya
    try {
      const res = await fetch(`http://localhost:8080/api/user/${userContext?.user}`, {
        method: 'GET',
        headers: {
          'Content-Type': 'application/json'
        },
        credentials: 'include',
      });

      if (res.ok) {
        const user = await res.json();
        setUser(user);
      } else {
        console.log('Error fetching user');
      }
    } catch (error) {
      console.error('Network error:', error);
    }
  }

  const fetchUserMindMaps = async () => {  Show usages  new *
    try {
      const res = await fetch(`http://localhost:8080/api/mind-map/all`, {
        method: 'GET',
        headers: {
          'Content-Type': 'application/json'
        },
        credentials: 'include',
      });

      if (res.ok) {
        const mindMaps = await res.json();
        setMindMaps(mindMaps);
      } else {
        console.log('Error fetching user mind maps');
      }
    } catch (error) {
      console.error('Network error:', error);
    }
  }

  if (userContext?.user) {
    fetchUser();
    fetchUserMindMaps();
  }
}, [userContext]);
```

Рис 5 - Завантаження даних користувача та його карт

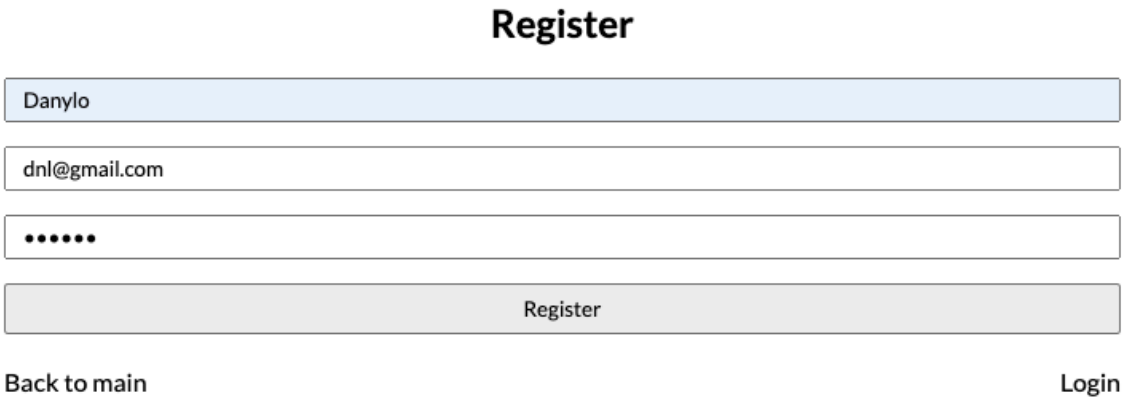


Рис 6 - Форма реєстрації, яку бачить користувач

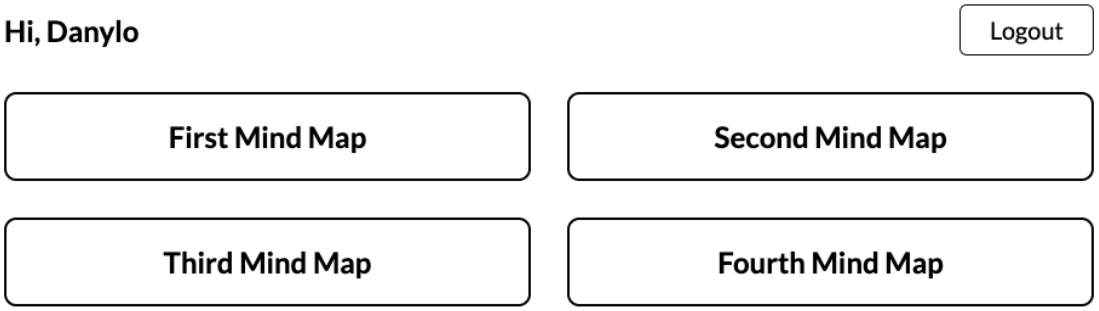


Рис 7 - Головна сторінка, яку бачить зареєстрований користувач

Для користувача це виглядатиме так, що він заповнює форму, після вдалої реєстрації він потрапляє на головну сторінку де будуть відображені як його особисті дані, так і дані про його карти пам’яті.

Весь оновлений код можна переглянути в даній директорії репозиторію web-частини проєкту: https://github.com/yablonya/TRPZ_labs_WEB_yablonskyi_ia-24/tree/main/src а також в директорії серверної частини проекту: https://github.com/yablonya/TRPZ_labs_yablonskyi_ia-24/tree/lab-9/src/main/java/org/example/mindmappingsoftware

**Висновок:** У процесі виконання роботи я реалізував взаємодію клієнта та сервера згідно архітектурі «клієнт-сервер». Основна увага приділялася надсиланню коректних запитів та правильна обробка цих запитів на сервері з подальшою відправкою відповіді на клієнт. Завдяки поточному архітектурному рішенню вдалося чітко розділити відповідальність між двома складовими: клієнт забезпечує зручне та інтуїтивне управління ментальними картами, тоді як серверна логіка відповідає за централізоване зберігання, обробку та захист даних.

Такий підхід дозволив оптимізувати ресурси та підвищити стабільність системи: серверна частина може бути масштабована при збільшенні навантаження, а клієнт залишатиметься легким та швидким. Крім того, централізований контроль даних спростив управління доступом та покращив безпечність, оскільки всі операції над ментальними картами контролюються в одному місці. Отже, фінальний результат підтвердив доцільність використання «клієнт-сервер» архітектури для доопрацювання вже існуючого рішення, що забезпечує гнучкість, ефективність та легке адміністрування системи.