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

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Взаємодія компонентів системи»

Тема роботи: 3. Текстовий редактор

Виконав
студент групи ІА–33
Супик Андрій Олександрович

Київ 2025

Тема: Взаємодія компонентів системи

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

Посилання на репозиторій: https://github.com/insxlll/trpzlab

Короткі теоретичні відомості

Client-Server

Шаблон "Client-Server" визначає взаємодію між двома типами компонентів: клієнтом, який робить запити, і сервером, який їх обробляє та повертає результати. Використовується для побудови розподілених систем, де клієнт взаємодіє з сервером через мережу. У Java цей шаблон реалізується за допомогою серверного додатка (наприклад, з використанням Java Servlet API чи Spring Boot) і клієнтської програми (через HTTP-клієнти, такі як Apache HttpClient чи RESTTemplate). Сервер обробляє запити клієнта, забезпечуючи централізоване управління даними, а клієнт отримує доступ до функціоналу через чітко визначений API

Peer-to-Peer (P2P)

Шаблон "Peer-to-Peer" забезпечує децентралізовану модель, де кожен вузол може діяти як клієнт і як сервер. Використовується для створення мереж, де учасники обмінюються ресурсами без центрального вузла. У Java цей шаблон реалізується через мережеве програмування (наприклад, з використанням java.net для сокетів або бібліотек для P2P мереж, таких як JXTA). Кожен вузол має механізми для з'єднання з іншими вузлами, надсилання запитів і відповіді на них. Завдяки цьому система стає стійкою до відмови окремих вузлів і легко масштабується.

Service-Oriented Architecture (SOA)

Шаблон "Service-Oriented Architecture" визначає побудову системи з незалежних сервісів, які взаємодіють через стандартизовані протоколи. Використовується для проектування модульних систем, де кожен сервіс відповідає за певну бізнес-логіку. У Java цей шаблон реалізується через технології, такі як SOAP (з використанням JAX-WS) або RESTful сервіси

(з використанням Spring REST). Сервіси спілкуються через API, що дозволяє інтегрувати їх незалежно від платформи або мови програмування. Такий підхід забезпечує повторне використання компонентів і спрощує їх модифікацію чи заміну

Хід роботи

3. **Текстовий редактор** (strategy, command, observer, template method, flyweight, SOA)

Текстовий редактор повинен вміти розпізнавати текстові файли в будь-якій кодуванні, мати розширені функції редагування: макроси, сніппети, підказки, закладки, перехід на рядок / сторінку, підсвічування синтаксису (для однієї мови програмування або розмітки на розсуд студента).

У цій роботі я використовую архітектурний патерн "Сервіс-орієнтована Архітектура" (SOA). Цей патерн було обрано як фундаментальне архітектурне рішення, оскільки він виконує роль мосту для інтеграції із зовнішніми, спеціалізованими сервісами. Це дозволяє текстовому редактору делегувати складні, неядерні завдання зовнішнім постачальникам послуг, забезпечуючи гнучкість і модульність.

У процесі роботи над проєктом стало зрозуміло, що система має ядерні (внутрішня логіка редактора) та неядерні (зовнішні) завдання, які вимагають інтеграції. Без використання SOA інтеграція зі складними зовнішніми сервісами призвела б до катастрофічних наслідків:

- Надмірна залежність: Логіка обробки зовнішніх API була б жорстко вбудована в основний код TextEditorBackend.jar.
- Складнощі розширення: Підключення нових зовнішніх сервісів (інших перекладачів, засобів перевірки) вимагало б модифікації та повного перекомпілювання основного коду.
- Відсутність модульності: Ядро системи було б обтяжене спеціалізованою логікою, що не стосується безпосередньо редагування.

Архітектура SOA вирішує цю проблему елегантно. Вона чітко розділяє відповідальність:

- Розширена функціональність: Дозволяє системі надавати користувачам функціонал, який не реалізовано безпосередньо у внутрішній бізнес-логіці TextEditorBackend.jar.
- Делегування складних завдань: Дозволяє делегувати такі завдання, як перевірка синтаксису або автопереклад, зовнішнім системам.
- Модульність та Гнучкість: Забезпечує архітектурний принцип, за яким підключення нових зовнішніх сервісів здійснюється без модифікації основного коду редактора.

Реалізація та Інтеграційні Компоненти

SOA у моїй системі представлена двома основними артефактами, які є доступними для Сервера Застосунків через REST API:

- SyntaxCheckAPI – Сервіс перевірки синтаксису
  - Відповідальність: Сервіс, відповідальний за перевірку синтаксису або граматики тексту.
  - Використання: Клієнт викликає цей сервіс, щоб виконати функцію «Перевірити синтаксис Java».
- TranslationAPI – Сервіс автоматичного перекладу
  - Відповідальність: Сервіс, відповідальний за переклад тексту на інші мови.
  - Використання: Використовується для функції «Автопереклад тексту» (через $DeepL$ API).

Зв'язок:

- Зв'язок між ними забезпечується протоколом HTTP/REST.
- Внутрішній Facade ($IExternalServices$) виступає як "middleware", приймаючи запити від логіки редактора і перетворюючи їх на виклики зовнішніх $REST$ сервісів.
- У діаграмі варіантів використання SOA представлений як окремий

актор, з яким відбувається взаємодія для виконання розширених функцій.

Реалізація взаємодії розподілених систем

У цій роботі було реалізовано архітектурний патерн Сервіс-орієнтована Архітектура (SOA) для забезпечення інтеграції із зовнішніми, спеціалізованими сервісами (перевірка синтаксису, переклад). .

Вимоги до завдання могли передбачати використання технологій, таких як SOAP/WSDL або .NET Web Services, для реалізації інтеграції. Оскільки даний проєкт розроблено на платформі Java з використанням фреймворку Spring Boot, було обрано більш сучасний та гнучкий підхід – RESTful API.

Для реалізації SOA-взаємодії та делегування завдань зовнішнім системам було обрано стандартні та сучасні технології з екосистеми Java, які виконують аналогічні функції:

- Зовнішній Сервіс незалежно розгорнуті сервіси, доступні через HTTP/REST. Вони інкапсулюють специфічну бізнес-логіку (переклад, синтаксис).
- Канал Зв'язку Легкий, стандартизований протокол для обміну даними (зазвичай JSON).
- Заглушка IExternalServices виступає як фасад, який приховує складність взаємодії. Внутрішньо він використовує RestTemplate або WebClient для здійснення вихідних HTTP-викликів до зовнішніх REST сервісів.

Механізм SOA-Взаємодії

У системі відбувається чітке розділення відповідальності:

- ❖ Сервер Застосунків (Spring Boot): Містить ядро логіки редактора.
- ❖ Інтерфейс IExternalServices: Викликається сервісним шаром (@Service)

для делегування завдання.

- Процес зв'язку: IExternalServices ініціює HTTP-запит до SyntaxCheckAPI або TranslationAPI, чекає на відповідь та повертає результат назад в основну бізнес-логіку.

Це забезпечує модульність: зовнішні сервіси можуть змінювати свою внутрішню реалізацію (наприклад, переходити з DeepL на Google Translate), не вимагаючи модифікації основного коду $TextEditorBackend$.

Структура програмного застосунку

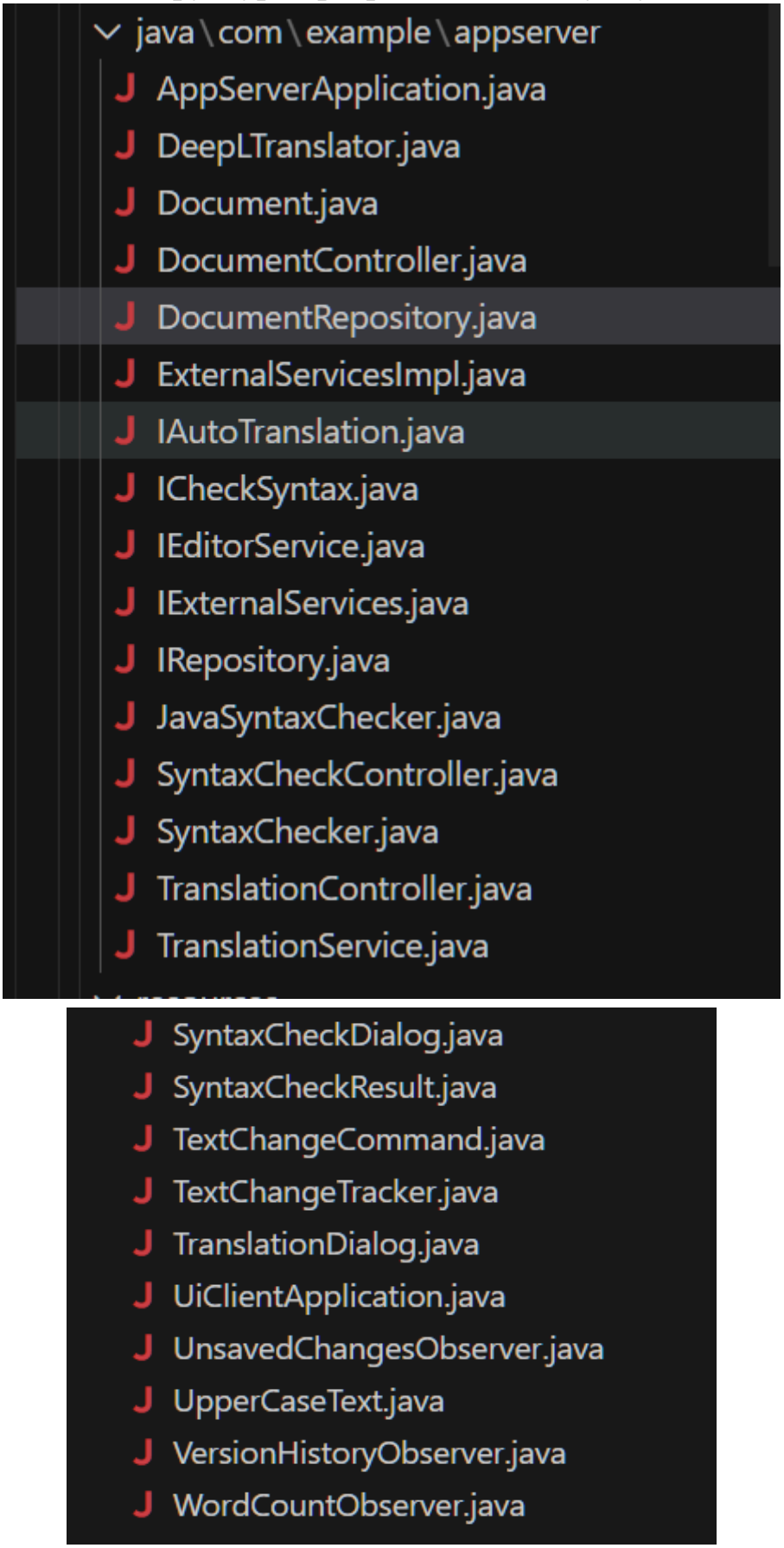

Рисунок 1. - Структура проєкту сервера

Опис архітектури проєкту

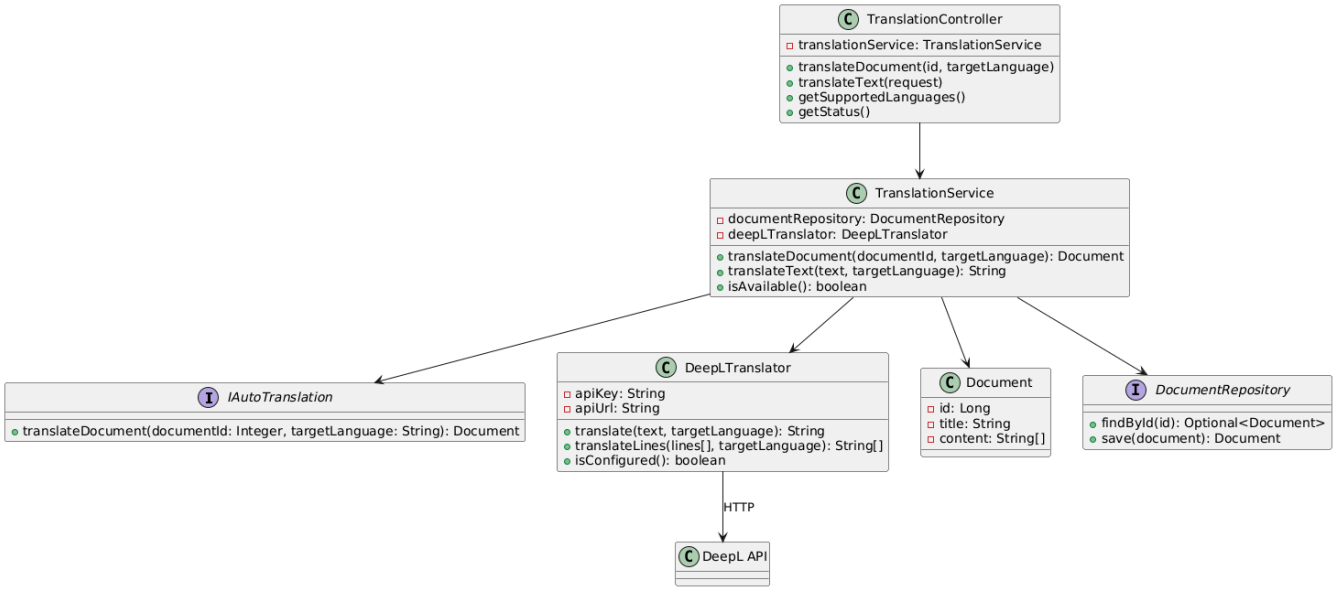

Рисунок 3. - Діаграма класів програмного застосунку

IAutoTranslation (Контракт Сервісу)

```
public interface IAutoTranslation {
	Document translateDocument(Integer documentId, String targetLanguage);
}
```

DeepLTranslator (Адаптер/Обробник API)

```
public class DeepLTranslator {

    private final HttpClient httpClient;
    private final ObjectMapper objectMapper;
    @Value("${deepl.api.key:}")
    private String apiKey;

@Value("${deepl.api.url:https://api-free.deepl.com/v2/translate}")
    private String apiUrl;
```

```
    public DeepLTranslator() {
        this.httpClient = HttpClient.newHttpClient();
        this.objectMapper = new ObjectMapper();
    }

        public String translate(String text, String
targetLanguage) throws Exception {
        if (text == null || text.trim().isEmpty()) {
            return text;
        }

        if (apiKey == null || apiKey.trim().isEmpty()) {
            throw new IllegalStateException("DeepL API ключ
не налаштований. " +
                                    "Додайте deepl.api.key у
application.properties");
        }

        String requestBody = String.format(
                "auth_key=%s&text=%s&target_lang=%s",
                                    URLEncoder.encode(apiKey,
StandardCharsets.UTF_8),
                                      URLEncoder.encode(text,
StandardCharsets.UTF_8),

URLEncoder.encode(targetLanguage.toUpperCase(),
StandardCharsets.UTF_8)
        );

        HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create(apiUrl))
                                    .header("Content-Type",
"application/x-www-form-urlencoded")
```

```
.POST(HttpRequest.BodyPublishers.ofString(requestBody))
                .build();

                        HttpResponse<String> response =
httpClient.send(request,
HttpResponse.BodyHandlers.ofString());

        if (response.statusCode() == 200) {
                                        JsonNode root =
objectMapper.readTree(response.body());
                                    JsonNode translations =
root.get("translations");
                            if (translations != null &&
translations.isArray() && translations.size() > 0) {
                                                    return
translations.get(0).get("text").asText();
            } else {
                    throw new Exception("Неочікуваний формат
відповіді від DeepL API");
            }
        } else if (response.statusCode() == 403) {
             throw new Exception("Невірний API ключ DeepL або
вичерпано ліміт");
        } else if (response.statusCode() == 456) {
               throw new Exception("Досягнуто ліміт символів
DeepL API");
        } else {
             throw new Exception("DeepL API помилка (код " +
response.statusCode() + "): " + response.body());
        }
    }
```

```
        public String[] translateLines(String[] lines, String
targetLanguage) throws Exception {
          if (lines == null || lines.length == 0) {
              return lines;
          }

          List<String> translatedLines = new ArrayList<>();
          for (String line : lines) {
              if (line == null || line.trim().isEmpty()) {
                  translatedLines.add(line);
              } else {
                          String translated = translate(line,
targetLanguage);
                  translatedLines.add(translated);
              }
          }

          return translatedLines.toArray(new String[0]);
      }

      public boolean isConfigured() {
          return apiKey != null && !apiKey.trim().isEmpty();
      }

      public static String[] getSupportedLanguages() {
          return new String[]{
                      "BG", "CS", "DA", "DE", "EL", "EN", "ES",
"ET", "FI", "FR",
                      "HU", "ID", "IT", "JA", "KO", "LT", "LV",
"NB", "NL", "PL",
                      "PT", "RO", "SK", "SL", "SV", "TR", "UK",
"ZH"
          };
```

```
}

public static String getLanguageName(String code) {
    switch (code.toUpperCase()) {
        case "BG": return "Болгарська";
        case "CS": return "Чеська";
        case "DA": return "Данська";
        case "DE": return "Німецька";
        case "EL": return "Грецька";
        case "EN": return "Англійська";
        case "ES": return "Іспанська";
        case "ET": return "Естонська";
        case "FI": return "Фінська";
        case "FR": return "Французька";
        case "HU": return "Угорська";
        case "ID": return "Індонезійська";
        case "IT": return "Італійська";
        case "JA": return "Японська";
        case "KO": return "Корейська";
        case "LT": return "Литовська";
        case "LV": return "Латвійська";
        case "NB": return "Норвезька";
        case "NL": return "Голландська";
        case "PL": return "Польська";
        case "PT": return "Португальська";
        case "RO": return "Румунська";
        case "SK": return "Словацька";
        case "SL": return "Словенська";
        case "SV": return "Шведська";
        case "TR": return "Турецька";
        case "UK": return "Українська";
        case "ZH": return "Китайська";
```

```
            default: return code;
        }
    }
}
```

TranslationService (Обробник Бізнес-логіки)

```
public class TranslationService implements IAutoTranslation
{

    @Autowired
    private DocumentRepository documentRepository;

    @Autowired
    private DeepLTranslator deepLTranslator;

    @Override
      public Document translateDocument(Integer documentId,
String targetLanguage) {
        if (documentId == null) {
             throw new IllegalArgumentException("ID документа
не може бути null");
        }

                                    Document document =
documentRepository.findById(documentId.longValue())
                                     .orElseThrow(() -> new
IllegalArgumentException("Документ з ID " + documentId + " не
знайдено"));

        try {
            if (!deepLTranslator.isConfigured()) {
                   throw new IllegalStateException("DeepL API
не налаштований");
            }
```

```
                                String[] originalContent =
document.getContent();
                                String[] translatedContent =
deepLTranslator.translateLines(originalContent,
targetLanguage);

                                String translatedTitle =
deepLTranslator.translate(document.getTitle(),
targetLanguage);

            Document translatedDocument = new Document();
              translatedDocument.setTitle(translatedTitle + "
(" + DeepLTranslator.getLanguageName(targetLanguage) + ")");

translatedDocument.setContent(translatedContent);

                                                    return
documentRepository.save(translatedDocument);

          } catch (Exception e) {
                    throw new RuntimeException("Помилка при
перекладі документа: " + e.getMessage(), e);
          }
      }

           public String translateText(String text, String
targetLanguage) throws Exception {
                    return deepLTranslator.translate(text,
targetLanguage);
      }

      public boolean isAvailable() {
```

```
        return deepLTranslator.isConfigured();
    }
}
```

TranslationController (API Контролер)

```
public class TranslationController {

    @Autowired
    private TranslationService translationService;

    @PostMapping("/document/{id}")
    public ResponseEntity<?> translateDocument(
            @PathVariable Integer id,
            @RequestBody TranslationRequest request) {
        try {
                                    String targetLanguage =
request.getTargetLanguage();
                            if (targetLanguage == null ||
targetLanguage.trim().isEmpty()) {
                return ResponseEntity.badRequest()
                            .body(Map.of("error", "Цільова мова
не вказана"));
            }

                                Document translatedDocument =
translationService.translateDocument(id, targetLanguage);
            return ResponseEntity.ok(translatedDocument);

        } catch (IllegalArgumentException e) {
            return ResponseEntity.badRequest()
                    .body(Map.of("error", e.getMessage()));
        } catch (Exception e) {
            return ResponseEntity.internalServerError()
```

```
                        .body(Map.of("error", "Помилка при
перекладі: " + e.getMessage()));
            }
        }

        @PostMapping("/text")
            public ResponseEntity<?> translateText(@RequestBody
TranslationRequest request) {
            try {
                String text = request.getText();
                                    String targetLanguage =
request.getTargetLanguage();

                if (text == null || text.trim().isEmpty()) {
                    return ResponseEntity.badRequest()
                                .body(Map.of("error", "Текст для
перекладу не вказано"));
                }

                                if (targetLanguage == null ||
targetLanguage.trim().isEmpty()) {
                    return ResponseEntity.badRequest()
                                .body(Map.of("error", "Цільова мова
не вказана"));
                }

                                    String translatedText =
translationService.translateText(text, targetLanguage);
                Map<String, String> response = new HashMap<>();
                response.put("originalText", text);
                response.put("translatedText", translatedText);
                response.put("targetLanguage", targetLanguage);
                return ResponseEntity.ok(response);
```

```
        } catch (Exception e) {
            return ResponseEntity.internalServerError()
                    .body(Map.of("error", "Помилка при перекладі: " + e.getMessage()));
        }
    }

    @GetMapping("/languages")
    public ResponseEntity<?> getSupportedLanguages() {
        String[] languages = DeepLTranslator.getSupportedLanguages();
        Map<String, String> languageMap = new HashMap<>();
        for (String code : languages) {
            languageMap.put(code, DeepLTranslator.getLanguageName(code));
        }
        return ResponseEntity.ok(languageMap);
    }

    @GetMapping("/status")
    public ResponseEntity<?> getStatus() {
        boolean available = translationService.isAvailable();
        Map<String, Object> status = new HashMap<>();
        status.put("available", available);
        status.put("message", available
                ? "DeepL API налаштований і готовий до роботи"
                : "DeepL API не налаштований. Додайте API ключ у application.properties");
        return ResponseEntity.ok(status);
    }
```

```
    public static class TranslationRequest {
        private String text;
        private String targetLanguage;

        public String getText() {
            return text;
        }

        public void setText(String text) {
            this.text = text;
        }

        public String getTargetLanguage() {
            return targetLanguage;
        }

         public void setTargetLanguage(String targetLanguage)
{
            this.targetLanguage = targetLanguage;
        }
    }
  }
```

Висновок: Виконуючи цю роботу, я ознайомився з ключовими архітектурними підходами, зокрема "Сервіс-орієнтована Архітектура" (SOA), та освоїв її практичне застосування.
Особливу увагу я приділив SOA, оскільки вона дозволила мені елегантно вирішити проблему інтеграції неядерних, але критично важливих функцій у мою систему.

Під час вивчення цієї архітектури я зрозумів, наскільки ефективно вона вирішує проблему розподілу завдань у системі. SOA чітко розмежовує ролі: Ядро системи (TextEditorBackend) інкапсулює основну бізнес-логіку, а Зовнішні сервіси (SyntaxCheckAPI, TranslationAPI) відповідають за спеціалізовану обробку.

Завдяки цьому досвіду я краще зрозумів переваги патерну SOA у контексті побудови модульних та гнучких додатків. Я усвідомив його ключову роль у делегуванні завдань (наприклад, переклад, перевірка синтаксису) та мінімізації залежностей в основному коді, що значно спрощує підтримку та оновлення системи.

Контрольні запитання

1. Що таке клієнт-серверна архітектура?

Клієнт-серверна архітектура – це модель взаємодії, у якій програма поділена на два основні компоненти: клієнт і сервер.

Клієнт відповідає за запити та відображення даних користувачу, а сервер – за обробку цих запитів, збереження інформації та виконання логіки. Клієнт надсилає запит, сервер його опрацьовує і повертає результат. Такий підхід дозволяє розділити відповідальності, спростити масштабування й забезпечити зручне обслуговування системи.

2. Розкажіть про сервіс-орієнтовану архітектуру.

Сервіс-орієнтована архітектура (SOA) – це підхід, у якому система складається з окремих незалежних сервісів. Кожен сервіс виконує чітко визначену функцію, має власний інтерфейс і може взаємодіяти з іншими сервісами через стандартизовані протоколи.

Сервіси не залежать від конкретної мови програмування чи платформи, тому їх легко комбінувати, оновлювати та повторно використовувати. Така архітектура спрощує масштабування системи, підвищує гнучкість і дозволяє змінювати окремі частини без впливу на всю систему.

3. Якими принципами керується SOA?

Основні принципи, якими керується сервіс-орієнтована архітектура (SOA):

- Слабке зв’язування. Сервіси мінімально залежать один від одного, що дозволяє їх змінювати окремо.
- Повторне використання. Сервіси створюються так, щоб їх можна було застосовувати в різних системах і сценаріях.
- Чіткі контракти. Кожен сервіс має визначений інтерфейс (контракт), через який клієнти з ним взаємодіють.

- Автономність. Сервіси самостійні: вони управляють власними даними та логікою.
- Незалежність технологій. Сервіси можуть бути реалізовані на різних мовах і працювати на різних платформах.

4. Як між собою взаємодіють сервіси в SOA?

У SOA сервіси взаємодіють між собою через чітко визначені інтерфейси, використовуючи стандартні протоколи обміну повідомленнями. Зазвичай це відбувається так: 1) один сервіс надсилає запит іншому сервісу; 2) інший сервіс обробляє запит і повертає відповідь у стандартизованому форматі (наприклад, XML або JSON); 3) зв'язок відбувається через мережу, здебільшого за допомогою HTTP, SOAP або REST.

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

У SOA розробники дізнаються про існуючі сервіси через реєстр (каталог) сервісів. Це спеціальне місце, де зберігається опис кожного сервісу: його назва, призначення, інтерфейси, формати даних і спосіб виклику. Такий каталог дозволяє швидко знайти потрібний сервіс і зрозуміти, як з ним працювати.

Щоб виконати запит до сервісу, розробник використовує його контракт – опис доступних операцій і параметрів. На основі цього контракту формується стандартний запит (наприклад, HTTP, SOAP або REST), який сервіс приймає й обробляє. Таким чином, маючи документацію або запис у каталозі сервісів, розробник легко може підключитися та використовувати будь-який сервіс у системі.

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

- Централізоване зберігання даних. Сервер контролює доступ, безпеку та оновлення інформації.
- Легка підтримка. Оновлення робляться на сервері, а клієнти отримують актуальні дані без змін у своїх програмах.
- Масштабованість. Можна підсилювати сервер або додавати нові сервери під навантаження.
- Кращий контроль безпеки. Дані та логіка зберігаються в одному місці. Недоліки клієнт-серверної моделі:
- Залежність від сервера. Якщо сервер виходить з ладу, система перестає працювати для всіх клієнтів.
- Велике навантаження на сервер. Усі запити надходять до одного місця, що може вимагати потужного обладнання.

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

Переваги однорангової (peer-to-peer, P2P) моделі:

- Відсутність центрального сервера. Кожен вузол одночасно є клієнтом і сервером, що зменшує залежність від одного центру.
- Хороша масштабованість. Чим більше вузлів, тим більше ресурсів (обчислювальних і мережевих) у мережі.
- Висока стійкість. Вихід окремих вузлів не зупиняє роботу всієї системи.
- Ефективність у розподілі навантаження. Дані та операції розподілені між учасниками.

Недоліки однорангової моделі:

- Складність безпеки. Немає єдиного центру контролю, тому важче забезпечити захист і перевірку даних.
- Нестабільність вузлів. Користувачі можуть довільно підключатися й відключатися, що ускладнює стабільність роботи.

- Обмежені ресурси. Вузли часто мають різні можливості, і слабкі пристрої можуть стримувати швидкість передачі.

8. Що таке мікро-сервісна архітектура?

Мікросервісна архітектура – це підхід, у якому система складається з великої кількості дрібних, незалежних сервісів. Кожен мікросервіс відповідає за одну конкретну функцію, має власну логіку, власні дані та може оновлюватися окремо від інших.

Сервіси спілкуються між собою через легкі мережеві протоколи (наприклад, HTTP або повідомлення). Такий підхід спрощує масштабування, дозволяє швидко оновлювати частини системи й робить її більш гнучкою та стійкою до відмов.

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

У мікросервісній архітектурі найчастіше використовують такі протоколи обміну даними: HTTP/HTTPS, AMQP (RabbitMQ) (Протокол для асинхронного обміну повідомленнями через брокера черг), Kafka Protocol (Використовується в Apache Kafka для стрімінгової передачі подій.), WebSockets (Для двостороннього постійного з’єднання, якщо потрібна швидка реакція або push-повідомлення.), MQTT (Легковаговий протокол для IoT-пристроїв та мікросервісів з малим ресурсом)

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, це не можна назвати сервіс-орієнтованою архітектурою. Якщо між веб-контролерами та доступом до даних створено шар бізнес-логіки у

вигляді сервісів, то це просто шарова (трирівнева) архітектура одного застосунку.

У цьому випадку сервіси – це частини одного проєкту, вони не працюють як окремі автономні системи і не взаємодіють через мережеві протоколи. SOA ж передбачає, що сервіси є незалежними, можуть розгортатися окремо, спілкуватися між собою через стандартизовані інтерфейси і використовуватися різними клієнтами.