

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
«Взаємодія компонентів системи»

Виконала:
студентка групи IA-32
Чайковська С. В.

Перевірив:
Мягкий М. Ю.

Зміст

1 Теоретичні відомості.....	3
2 Тема.....	6
3 Реалізація частини функціоналу робочої програми.....	7
4 Реалізація взаємодії розподілених частин для SOA застосунку.....	10
4.1 Реалізація сервісів, що надає послуги клієнтським застосуванням.....	10
4.2 Використання токенів для передачі даних про автентифікації.....	21
4.3 Двостороннє шифрування.....	22
5 Діаграма класів архітектури «SOA».....	24
6 Висновки.....	25
7 Відповіді на питання.....	26

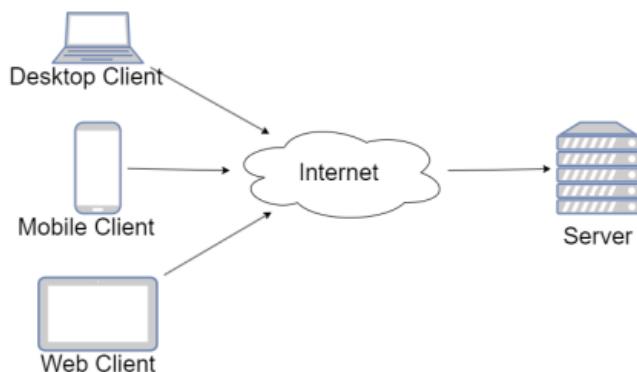
Тема: Взаємодія компонентів системи.

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

1 Теоретичні відомості

Архітектура додатків визначає спосіб організації компонентів програмного забезпечення та їх взаємодії для досягнення функціональних і нефункціональних вимог системи. Вона задає структуру, правила та стандарти для побудови додатків, забезпечуючи масштабованість, ефективність і легкість супроводу. Вибір архітектури залежить від вимог проєкту, таких як масштабованість, надійність, продуктивність і простота супроводу.

Клієнт-серверна архітектура



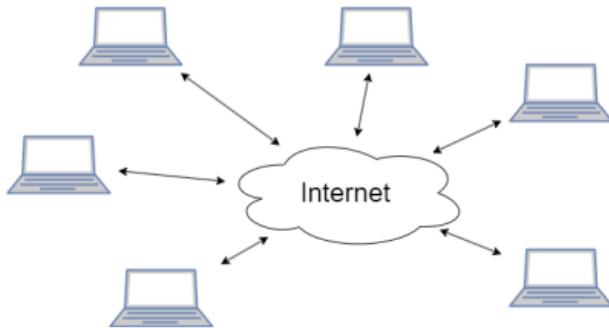
Клієнт-серверні додатки – це розподілені системи, що складаються з клієнтів та серверів. Розрізняють *тонкі* та *тovсті клієнти*.

Тонкий клієнт передає більшість обчислень на сервер, залишаючи на собі лише візуальне відображення даних. Це зменшує вимоги до клієнтських пристройів, але збільшує навантаження на сервер. Така модель актуальна для захищених сценаріїв або для уникнення конфліктів при множинному доступі до даних.

Товстий клієнт виконує більшість обчислень локально, розвантажуючи сервер, який виступає лише як точка доступу до ресурсів.

Для взаємодії використовується модель «підписка/видача», де клієнти підписуються на події, а сервер їх сповіщає. Це зручно для оновлення даних, але створює навантаження на сервер і канали зв'язку.

Peer-to-Peer архітектура



Модель взаємодії додатків типу peer - to - peer має на увазі рівноправ'я клієнтських програм і відсутність серверної програми. Усі клієнтські програми контактирують між собою для виконання спільних цілей. У таких мережах найчастіше виникають наступні проблеми: синхронізація даних, пошук клієнтських застосувань.

Для пошуку клієнтських застосувань може використовуватися одна загальна адреса, що містить набір адрес підключених клієнтів.

Сервіс-орієнтована архітектура

Сервіс-орієнтована архітектура — це підхід до розробки програмного забезпечення, який передбачає використання розподілених, слабо пов'язаних компонентів із стандартизованими інтерфейсами для взаємодії через стандартизовані протоколи. SOA дозволяє створювати масштабовані, керовані системи, незалежні від платформи та інструментів розробки. Програмні комплекси, побудовані за SOA, зазвичай реалізуються як набір веб-служб, що використовують протоколи SOAP або REST. Інтерфейси компонентів інкапсулюють деталі реалізації, що забезпечує можливість багаторазового використання та легкість інтеграції. SaaS (Software as a Service) — це бізнес-модель, у якій постачальник розробляє веб-додаток, самостійно ним керує та надає клієнтам доступ через Інтернет.

Основні риси SaaS:

- Відсутність витрат на встановлення, оновлення й підтримку ПЗ та обладнання.
- Один додаток може обслуговувати кілька клієнтів.
- Оплата здійснюється як абонентська плата або залежить від обсягу використання.
- Оновлення та техпідтримка включені у вартість.
- Оновлення відбуваються автоматично та прозоро для клієнтів.

Мікро-сервісна архітектура

Архітектура мікрослужб — це підхід до розробки додатків, де система складається з невеликих автономних служб. Кожна служба виконується у власному процесі та взаємодіє з іншими через стандартизовані протоколи, такі як HTTP/HTTPS, WebSockets чи AMQP.

Переваги:

- Масштабованість: кожна служба масштабується незалежно.
- Гнучкість: використання різних технологій для різних завдань.
- Супровідність: код служби зрозумілий і легкий для змін.
- Стійкість: збої в одній службі не впливають на роботу інших.
- Незалежна розробка: команди працюють над різними службами паралельно.

Виклики:

- Управління складністю: потрібні інструменти для моніторингу, оркестрації та забезпечення безпеки.
- Взаємодія між службами: важливо забезпечити надійність і швидкість передачі даних.
- Розподілена природа: ускладнюється тестування та підтримка узгодженості даних.

2 Тема

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).

3 Реалізація частини функціоналу робочої програми

Структура проекту зображена на Рисунку 1:

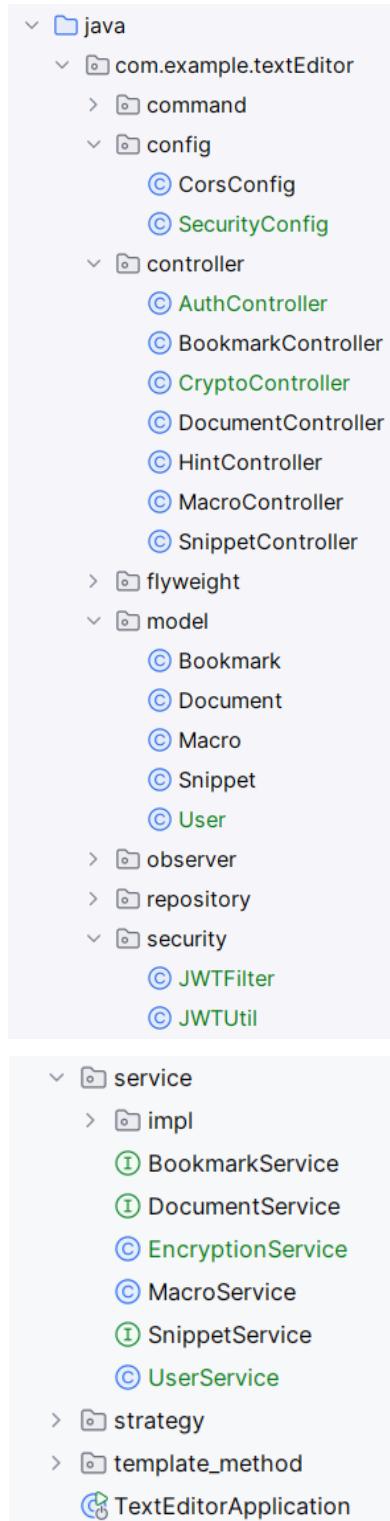


Рис. 1 – Структура проекту

Реалізовано архітектуру SOA:

- *DocumentService* – відповідає за створення, редагування, відкриття, оновлення та збереження документів, а також виконує шифрування та роботу з Flyweight.
- *SyntaxHighlightService* – забезпечує підсвічування синтаксису залежно від типу файлу.
- *FlyweightService* – оптимізує використання пам'яті, обробляючи символи за патерном Flyweight та підраховуючи унікальні символи.
- *MacroService* – обробляє макроси текстового редактора, включаючи форматування та вставку сніппетів.
- *HintGenerator* – генерує підказки автодоповнення на основі словника ключових слів.
- *BookmarkService* – відповідає за роботу із закладками: створення, оновлення та видалення.
- *SnippetService* – забезпечує управління сніппетами, які прив'язані до конкретних документів.
- *UserService* – виконує автентифікацію користувачів, хешування та перевірку паролів.
- *EncryptionService* – забезпечує двостороннє шифрування та розшифрування даних документів.

Реалізовано REST API для взаємодії:

DocumentController

- POST /file/save – створення документа
- GET /file/open/{id} – відкриття документа
- PUT /file/edit/{id} – редагування
- POST /file/upload – завантаження файлу
- POST /file/highlight – підсвічування синтаксису
- GET /file – список документів

MacroController

- POST /macro/format – обробка форматуючого макросу
- POST /macro/snippet – вставка сніппету

HintController

- POST /file/hints – генерація підказок
- POST /bookmarks – створення закладки
- PUT /bookmarks/{id} – оновлення
- DELETE /bookmarks/{id} – видалення

SnippetController

- POST /snippets – створення
- PUT /snippets/{id} – оновлення
- DELETE /snippets/{id} – видалення

AuthController

- POST /auth/register – реєстрація
- POST /auth/login – отримання JWT-токена

CryptoController

- POST /crypto/encrypt – шифрування
- POST /crypto/decrypt – розшифрування

4 Реалізація взаємодії розподілених частин для SOA застосунку

4.1 Реалізація сервісів, що надає послуги клієнтським застосуванням

У розробленому застосунку реалізовано сервісно-орієнтовану архітектуру (SOA), де клієнтський застосунок взаємодіє з бекендом через набір REST-контролерів. Кожен контролер надає окрему групу сервісів, які інкапсулюють бізнес-логіку системи.

Контролери виконують роль вхідних точок сервісів, а основна логіка опрацьовується відповідними сервісними класами. Такий підхід забезпечує чіткий поділ відповідальностей, повторне використання логіки та розширеність системи.

Основні контролери й відповідні сервіси:

1. Сервіс *DocumentServiceImpl* представлений на Рисунку 2:

```
14     @Service & Chaikovska Sofiia *
15     public class DocumentServiceImpl implements DocumentService {
16
17         private final DocumentRepository documentRepository;
18         private final SyntaxHighlightService syntaxHighlightService;
19         private final FlyweightService flyweightService;
20         private final EncryptionService encryptionService;
21
22         public DocumentServiceImpl( & Chaikovska Sofiia *
23             DocumentRepository documentRepository,
24             SyntaxHighlightService syntaxHighlightService,
25             FlyweightService flyweightService,
26             EncryptionService encryptionService
27         ) {
28             this.documentRepository = documentRepository;
29             this.syntaxHighlightService = syntaxHighlightService;
30             this.flyweightService = flyweightService;
31             this.encryptionService = encryptionService;
32         }
33
34         @Override & Chaikovska Sofiia *
35         public Document create(Document document) {
36             Timestamp now = new Timestamp(System.currentTimeMillis());
37             document.setCreatedAt(now);
38             document.setUpdatedAt(now);
39
40             String plainText = document.getContent();
41
42             if (plainText != null && !plainText.isEmpty()) {
43                 flyweightService.processText(plainText);
44             }
45         }
46     }
```

```

46     detectExtensionByContent(document);

47

48     String highlighted = syntaxHighlightService.highlight(document);
49     document.setHighlightedContent(highlighted);

50

51     if (plainText != null) {
52         String encrypted = encryptionService.encrypt(plainText);
53         document.setContent(encrypted);
54     }

55

56     return documentRepository.save(document);
57 }

58

59 @Override ➜ Chaikovska Sofiia *
60 public Document getById(int id) {
61     Document doc = documentRepository.findById(id)
62         .orElseThrow(() -> new RuntimeException("Document not found"));

63

64     String decrypted = encryptionService.decrypt(doc.getContent());
65     doc.setContent(decrypted);

66

67     String highlighted = syntaxHighlightService.highlight(doc);
68     doc.setHighlightedContent(highlighted);

69

70     return doc;
71 }

73
74 @Override ➜ Chaikovska Sofiia *
75 public Document update(Document document) {
76     document.setUpdatedAt(new Timestamp(System.currentTimeMillis()));

77

78     String plainText = document.getContent();

79     if (plainText != null && !plainText.isEmpty()) {
80         flyweightService.processText(plainText);
81     }

82

83     detectExtensionByContent(document);

84

85     String highlighted = syntaxHighlightService.highlight(document);
86     document.setHighlightedContent(highlighted);

87

88     String encrypted = encryptionService.encrypt(plainText);
89     document.setContent(encrypted);

90

91     return documentRepository.save(document);
92 }

```

Рис. 2 – Сервіс DocumentServiceImpl

Клас *DocumentServiceImpl*, наведений на рис. 2, реалізує сервісну логіку роботи з документами у системі текстового редактора. Основне призначення цього класу — забезпечити повний цикл обробки документа, включаючи визначення типу файлу, підсвічування синтаксису, підрахунок унікальних символів та шифрування вмісту.

Клас містить ключові методи:

- *create()* — виконує створення документа, аналіз тексту та шифрування перед збереженням.

- `getById()` — отримує документ та розшифрує його для клієнта.
- `update()` — оновлює вміст і повторно шифрує документ.
- `getAll()` — повертає всі документи у розшифрованому вигляді.

2. Контролер *DocumentController* представлений на Рисунку 3:

```

15  @RestController  ↳ Chaikovska Sofiia *
16  @RequestMapping(@RequestMappingPath="/file")
17  public class DocumentController {
18
19      private final DocumentService documentService;  16 usages
20      private final SyntaxHighlightService syntaxHighlightService;  2 usages
21      private final CommandInvoker invoker = new CommandInvoker();  5 usages
22
23  public DocumentController(DocumentService documentService, SyntaxHighlightService syntaxHighlightService)
24      this.documentService = documentService;
25      this.syntaxHighlightService = syntaxHighlightService;
26  }
27
28  @PostMapping(PostMappingPath="/save")  ↳ Chaikovska Sofiia *
29  public ResponseEntity<Map<String, Object>> save(@RequestBody Document document) {
30      try {
31          Document saved = documentService.create(document);
32          Command command = new SaveDocumentCommand(documentService, saved);
33          invoker.executeCommand(command);
34
35          int uniqueCount = documentService.getFlyweightCount();
36
37          Map<String, Object> response = new HashMap<>();
38          response.put("document", saved);
39          response.put("uniqueSymbols", uniqueCount);
40
41          return ResponseEntity.ok(response);
42      } catch (Exception e) {
43          e.printStackTrace();
44          return ResponseEntity.internalServerError().build();
45      }
46  }

```

```

48  @GetMapping(GetMappingPath="/open/{id}")  ↳ Chaikovska Sofiia
49  public ResponseEntity<Document> open(@PathVariable int id) {
50      try {
51          Command command = new OpenDocumentCommand(documentService, id);
52          invoker.executeCommand(command);
53
54          Document doc = documentService.getById(id);
55          return ResponseEntity.ok(doc);
56      } catch (Exception e) {
57          Document newDoc = new Document();
58          newDoc.setFilename("newfile.txt");
59          newDoc.setContent("");
60          Document saved = documentService.create(newDoc);
61
62          Command command = new SaveDocumentCommand(documentService, saved);
63          invoker.executeCommand(command);
64
65          return ResponseEntity.ok(saved);
66      }
67  }
68
69  @PutMapping(PostMappingPath="/edit/{id}")  ↳ Chaikovska Sofiia
70  public ResponseEntity<Map<String, Object>> edit(@PathVariable int id, @RequestBody String content) {
71      try {
72          Document doc = documentService.getById(id);
73
74          Command command = new EditDocumentCommand(documentService, doc, content);
75          invoker.executeCommand(command);
76      }
77  }

```

```

77         int uniqueCount = documentService.getFlyweightCount();
78
79         Map<String, Object> response = new HashMap<>();
80         response.put("document", doc);
81         response.put("uniqueSymbols", uniqueCount);
82
83         return ResponseEntity.ok(response);
84     } catch (Exception e) {
85         return ResponseEntity.internalServerError().build();
86     }
87 }
88
89 @PostMapping(@RequestMapping("/upload")) & Chaikovska Sofiia
90 public ResponseEntity<Map<String, Object>> upload(@RequestParam("file") MultipartFile file) {
91     try {
92         String content = new String(file.getBytes(), java.nio.charset.StandardCharsets.UTF_8);
93         String filename = file.getOriginalFilename();
94         if (filename == null || filename.isBlank()) {
95             return ResponseEntity.badRequest().build();
96         }
97
98         Document doc = new Document();
99         doc.setFilename(filename);
100        doc.setContent(content);
101
102        Document saved = documentService.create(doc);
103
104        Command command = new UploadDocumentCommand(documentService, saved);
105        invoker.executeCommand(command);

```

```

107         int uniqueCount = documentService.getFlyweightCount();
108
109         Map<String, Object> response = new HashMap<>();
110         response.put("document", saved);
111         response.put("uniqueSymbols", uniqueCount);
112
113         return ResponseEntity.ok(response);
114     } catch (Exception e) {
115         e.printStackTrace();
116         return ResponseEntity.internalServerError().build();
117     }
118 }
119
120 @GetMapping(@RequestMapping("/flyweight/stats")) & Chaikovska Sofiia
121 public ResponseEntity<Map<String, Object>> getFlyweightStats() {
122     Map<String, Object> stats = new HashMap<>();
123     stats.put("uniqueSymbols", documentService.getFlyweightCount());
124     return ResponseEntity.ok(stats);
125 }
126
127
128 @PostMapping(@RequestMapping("/highlight")) & Chaikovska Sofiia
129 public ResponseEntity<String> highlight(@RequestBody Document document) {
130     String highlighted = syntaxHighlightService.highlight(document);
131     return ResponseEntity.ok(highlighted);
132 }

```

Рис. 3 – Контролер DocumentController

Клас *DocumentController*, наведений на рис. 3, визначає веб-сервісний інтерфейс для клієнтських застосунків. Основне призначення цього контролера — надати REST API для виконання операцій з документами: створення, редагування, відкриття, завантаження та аналізу. *DocumentController* надає

доступ до всіх сервісних можливостей через HTTP і виступає основним Web Service для клієнта у контексті SOA.

3. Сервіс *MacroService* представлений на Рисунку 4:

```
1 package com.example.textEditor.service;
2
3 > import ...
4
5
6
7 @Service 3 usages  ↳ Chaikovska Sofia
8
9 public class MacroService {
10
11     public String formatText(String text, String formatType) {
12         MacroProcessor processor = new FormattingMacroProcessor();
13         return processor.process( inputText: "@" + formatType + "(" + text + ")");
14     }
15
16     public String insertSnippet(String snippetType) {
17         MacroProcessor processor = new SnippetMacroProcessor();
18         return processor.process( inputText: "@snippet(" + snippetType + ")");
19     }
20 }
```

Рис. 4 – Сервіс *MacroService*

Клас *MacroService*, наведений на рис. 4, реалізує сервіс для роботи з макросами у системі текстового редактора. Основне призначення цього класу — обробка текстових макросів та генерація готових форматувальних або шаблонних конструкцій.

Клас містить два основні методи:

- *formatText()* — створює процесор форматувальних макросів та виконує операції над текстом (uppercase, bold, italic тощо).
- *insertSnippet()* — генерує готові шаблонні конструкції, такі як цикл, функція чи клас.

4. Контролер *MacroController* представлений на Рисунку 5:

```
1 package com.example.textEditor.controller;
2
3 > import ...
4
5
6
7 @RestController  ↳ Chaikovska Sofia
8 @RequestMapping("/macro")
9 public class MacroController {
10
11     private final MacroService macroService; 3 usages
12
13     public MacroController(MacroService macroService) {
14         this.macroService = macroService;
15     }
16 }
```

```

17     @PostMapping("/{format}")  ↳ Chaikovska Sofia
18     public ResponseEntity<String> formatMacro(
19         @RequestParam String text,
20         @RequestParam String type
21     ) {
22         String result = macroService.formatText(text, type);
23         return ResponseEntity.ok(result);
24     }
25
26     @PostMapping("/{snippet}")  ↳ Chaikovska Sofia
27     public ResponseEntity<String> snippetMacro(
28         @RequestParam String type
29     ) {
30         String result = macroService.insertSnippet(type);
31         return ResponseEntity.ok(result);
32     }
33 }
```

Рис. 5 – Контролер MacroController

Клас *MacroController*, наведений на рис. 5, визначає веб-інтерфейс для роботи з макросами. Основне призначення цього контролера — надати клієнтським застосункам REST API для форматування тексту та вставки кодових сніппетів.

5. Сервіс *HintGenerator* представлений на Рисунку 6:

```

1  package com.example.textEditor.service.impl;
2
3  > import ...
4
5
6  @Service 6 usages  ↳ Chaikovska Sofia
7
8  public class HintGenerator {
9
10
11     private final List<String> dictionary = new ArrayList<>(Arrays.asList( 1 usage
12         "print", "println", "printf", "for", "while", "if", "else",
13         "int", "float", "double", "String", "class", "public", "private", "static"
14     ));
15
16     public List<String> generateHints(String input) { 2 usages  ↳ Chaikovska Sofia
17         List<String> hints = new ArrayList<>();
18         if (input == null || input.isEmpty()) return hints;
19         String lowerInput = input.toLowerCase();
20         for (String word : dictionary) {
21             if (word.startsWith(lowerInput)) {
22                 hints.add(word);
23             }
24         }
25         return hints;
26     }
27 }
```

Рис. 6 – Сервіс HintGenerator

Клас *HintGenerator*, наведений на рис. 6, реалізує сервіс автоматичної генерації підказок у системі текстового редактора. Основне призначення цього класу — формувати список можливих слів-доповнень на основі введеного фрагмента, допомагаючи користувачу при наборі тексту.

6. Контролер *HintController* представлений на Рисунку 7:

```

8  @RestController  & Chaikovska Sofia*
9  @RequestMapping(@"/file")
10 public class HintController {
11
12     private final HintGenerator hintGenerator;  2 usages
13
14     public HintController(HintGenerator hintGenerator) {
15         this.hintGenerator = hintGenerator;
16     }
17
18     @PostMapping(@"/hints")  & Chaikovska Sofia
19     public HintsResponse getHints(@RequestBody TextRequest request) {
20         List<String> hints = hintGenerator.generateHints(request.getText());
21         return new HintsResponse(hints);
22     }
23
24     public static class TextRequest {
25         private String text;  2 usages
26         public String getText() { return text; }
27         public void setText(String text) { this.text = text; }
28     }
29
30     public static class HintsResponse {
31         private List<String> hints;  1 usage
32         public HintsResponse(List<String> hints) { this.hints = hints; }
33     }
}

```

Рис. 7 – Контролер HintController

Клас *HintController*, наведений на рис. 7, визначає веб-інтерфейс для отримання текстових підказок. Основне призначення цього контролера — надати REST API, який дозволяє клієнтським застосункам отримувати автодоповнення під час введення тексту.

7. Сервіс *BookmarkServiceImpl* представлений на Рисунку 8:

```

11  @Service  & Chaikovska Sofia*
12  public class BookmarkServiceImpl implements BookmarkService {
13
14      private final BookmarkRepository bookmarkRepository;
15
16      public BookmarkServiceImpl(BookmarkRepository bookmarkRepository) {
17          this.bookmarkRepository = bookmarkRepository;
18      }
19
20      @Override  & Chaikovska Sofia
21      public Bookmark create(Bookmark bookmark) {
22          return bookmarkRepository.save(bookmark);
23      }
24
25      @Override  & Chaikovska Sofia
26      public Bookmark update(Bookmark bookmark) {
27          return bookmarkRepository.save(bookmark);
28      }
29
30      @Override  1 usage  & Chaikovska Sofia
31      public void delete(int id) {
32          bookmarkRepository.deleteById(id);
33      }
}

```

Рис. 8 – Сервіс BookmarkServiceImpl

Клас *BookmarkServiceImpl*, наведений на рис. 8, реалізує сервісну логіку управління закладками у системі текстового редактора. Основне призначення цього класу — забезпечити створення, оновлення та видалення закладок, що прив'язуються до конкретних документів.

Клас містить основні методи:

- *create()* — зберігає нову закладку у базу даних.
- *update()* — оновлює параметри існуючої закладки.
- *delete()* — видаляє закладку за ідентифікатором.

8. Контролер *BookmarkController* представлений на Рисунку 9:

```
11  @RestController  ↳ Chaikovska Sofia *
12  @RequestMapping(@GetMapping("/bookmarks"))
13  public class BookmarkController {
14
15      private final BookmarkService bookmarkService;  4 usages
16      private final DocumentService documentService;  3 usages
17
18      public BookmarkController(BookmarkService bookmarkService, DocumentService documentService) {
19          this.bookmarkService = bookmarkService;
20          this.documentService = documentService;
21      }
22
23      @PostMapping(PostMapping("/bookmarks"))
24      public Bookmark create(@RequestBody Bookmark bookmark) {
25          Document doc = documentService.getById(bookmark.getDocument().getId());
26          bookmark.setDocument(doc);
27          return bookmarkService.create(bookmark);
28      }
29
30      @PutMapping(PathVariable("/{id}"))
31      public Bookmark update(@PathVariable int id, @RequestBody Bookmark bookmark) {
32          bookmark.setId(id);
33          Document doc = documentService.getById(bookmark.getDocument().getId());
34          bookmark.setDocument(doc);
35          return bookmarkService.update(bookmark);
36      }
37
38      @DeleteMapping(PathVariable("/{id}"))
39      public void delete(@PathVariable int id) {
40          bookmarkService.delete(id);
41      }

```

Рис. 9 – Контролер *BookmarkController*

Клас *BookmarkController*, наведений на рис. 9, визначає веб-інтерфейс для роботи із закладками. Основне призначення цього контролера — надати REST API для створення, зміни та видалення закладок, які відносяться до певних документів.

9. Сервіс *UserService* представлений на Рисунку 10:

```

9   @Service 4 usages new *
10  @RequiredArgsConstructor
11  public class UserService {
12
13    private final UserRepository repo;
14    private final BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();
15
16    @
17      public void register(User user) { 1 usage new *
18        user.setPassword(encoder.encode(user.getPassword()));
19        repo.save(user);
20      }
21
22      public User getByUsername(String username) { 1 usage new *
23        return repo.findByUsername(username).orElse( other: null);
24      }
25
26      public boolean checkPassword(String raw, String encoded) {
27        return encoder.matches(raw, encoded);
28    }

```

Рис. 10 – Сервіс UserService

Клас *UserService*, наведений на рис. 10, реалізує сервісну логіку автентифікації користувачів у системі. Основне призначення цього класу — забезпечити реєстрацію користувачів, пошук облікових записів та перевірку коректності введеного пароля. *UserService* забезпечує ключові операції автентифікації та ізолює логіку роботи з користувачами від контролера.

Клас містить основні методи:

- *register()* — виконує збереження нового користувача із хешованим паролем (BCrypt).
- *getByUsername()* — отримує користувача за логіном з бази даних.
- *checkPassword()* — порівнює введений пароль із збереженим хешем.

10. Контролер *AuthController* представлений на Рисунку 11:

```

9   @RestController new *
10  @RequestMapping(@"/auth")
11  @RequiredArgsConstructor
12  public class AuthController {
13
14    private final UserService userService;
15    private final JWTUtil jwtUtil;
16
17    @PostMapping(@"/register") new *
18    public String register(@RequestBody User user) {
19      userService.register(user);
20      return "OK";
21    }

```

```

23     @PostMapping(value="/login") new *
24     public String login(@RequestBody User user) {
25         User dbUser = userService.getByUsername(user.getUsername());
26
27         if (dbUser == null || !userService.checkPassword(user.getPassword(), dbUser.getPassword()))
28             return "INVALID CREDENTIALS";
29
30         return jwtUtil.generateToken(dbUser.getUsername());
31     }
32 }
```

Рис. 11 – Контролер AuthController

Клас *AuthController*, наведений на рис. 11, визначає веб-інтерфейс для виконання операцій автентифікації. Основне призначення цього контролера — надати REST API для реєстрації користувачів та отримання JWT-токена при вході в систему. *AuthController* виконує роль точки входу для користувачів та забезпечує обмін токенами автентифікації у рамках SOA-архітектури.

Контролер містить два основні ендпоїнти:

- */register* — приймає дані нового користувача та викликає сервіс для його реєстрації.
- */login* — перевіряє правильність логіна та пароля та повертає JWT-токен у разі успішної автентифікації.

11. Сервіс *EncryptionService* представлений на Рисунку 12:

```

13  public class EncryptionService {
14
15      private static final String ALGO = "AES"; 1 usage
16      private final SecretKeySpec keySpec; 3 usages
17
18      public EncryptionService(@Value("MySecretKeyForAES") String secretKey) { new *
19          byte[] key = Arrays.copyOf(secretKey.getBytes(StandardCharsets.UTF_8), newLength: 16);
20          this.keySpec = new SecretKeySpec(key, ALGO);
21      }
22
23      public String encrypt(String plain) { 3 usages new *
24          try {
25              Cipher cipher = Cipher.getInstance( transformation: "AES/ECB/PKCS5Padding");
26              cipher.init(Cipher.ENCRYPT_MODE, keySpec);
27              byte[] enc = cipher.doFinal(plain.getBytes(StandardCharsets.UTF_8));
28              return Base64.getEncoder().encodeToString(enc);
29          } catch (Exception e) {
30              throw new RuntimeException("Encrypt error", e);
31          }
32      }
33
34      public String decrypt(String encoded) { 2 usages new *
35          try {
36              Cipher cipher = Cipher.getInstance( transformation: "AES/ECB/PKCS5Padding");
37              cipher.init(Cipher.DECRYPT_MODE, keySpec);
38              byte[] dec = cipher.doFinal(Base64.getDecoder().decode(encoded));
39              return new String(dec, StandardCharsets.UTF_8);
40          } catch (Exception e) {
41              throw new RuntimeException("Decrypt error", e);
42          }
43      }
44 }
```

Рис. 12 – Сервіс EncryptionService

Клас *EncryptionService*, наведений на рис. 12, реалізує механізм двостороннього шифрування у системі текстового редактора. Основне призначення цього класу — забезпечити зашифроване збереження даних та можливість їх подальшого розшифрування за допомогою алгоритму AES. *EncryptionService* відповідає за захист даних документа та реалізує вимогу SOA-додатків щодо двостороннього шифрування.

Клас містить два головні методи:

- *encrypt()* — виконує шифрування тексту з використанням AES та повертає результат у вигляді Base64.
- *decrypt()* — здійснює зворотну операцію та відновлює вихідний текст із зашифрованого рядка.

12. Контролер *CryptoController* представлений на Рисунку 13:

```
7  @RestController new *
8  @RequestMapping("/crypto")
9  @RequiredArgsConstructor
10 public class CryptoController {
11
12     private final EncryptionService encryptionService;
13
14     @PostMapping("/encrypt") new *
15     public String encrypt(@RequestBody String text) {
16         return encryptionService.encrypt(text);
17     }
18
19     @PostMapping("/decrypt") new *
20     public String decrypt(@RequestBody String encrypted) {
21         return encryptionService.decrypt(encrypted);
22     }
23 }
```

Рис. 13 – Контролер *CryptoController*

Клас *CryptoController*, наведений на рис. 13, визначає веб-інтерфейс для роботи з шифруванням. Основне призначення цього контролера — надати REST API, який дозволяє клієнтським застосункам виконувати операції шифрування та розшифрування тексту. *CryptoController* забезпечує доступ до сервісу шифрування та може використовуватися будь-яким клієнтом у рамках SOA-архітектури.

Контролер містить два основні ендпоїнти:

- */encrypt* — приймає відкритий текст та повертає його зашифрований варіант.

- */decrypt* — приймає зашифрований рядок та повертає розшифрований текст.

4.2 Використання токенів для передачі даних про автентифікації

Для забезпечення автентифікації у розробленому сервісі використовується механізм JWT. Після успішного входу користувача сервер генерує токен та повертає його клієнтському застосунку. У подальших запитах клієнт надсилає цей токен у заголовку *Authorization*, що дозволяє серверу ідентифікувати користувача без створення сесій.

У застосунку реалізовано:

- клас *JWTUtil*, представлений на Рис. 14, який генерує та перевіряє JWT-токени;
- фільтр *JWTFilter*, представлений на Рис. 15, який перехоплює кожен HTTP-запит та виконує валідацію токена.

Таким чином, токен слугує засобом передачі інформації про автентифікацію між клієнтським застосунком та сервером, що повністю відповідає вимогам SOA-архітектури.

```

10  @Component 3 usages new *
11  public class JWTUtil {
12
13      private final Key secret = Keys.secretKeyFor(SignatureAlgorithm.HS256); 3 usages
14
15      public String generateToken(String username) { 1 usage new *
16          return Jwts.builder()
17              .setSubject(username)
18              .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60))
19              .signWith(secret)
20              .compact();
21      }
22
23      public String extractUsername(String token) { 1 usage new *
24          return Jwts.parserBuilder() JwtParserBuilder
25              .setSigningKey(secret)
26              .build() JwtParser
27              .parseClaimsJws(token) Jws<Claims>
28              .getBody() Claims
29              .getSubject();
30      }
31
32      public boolean validateToken(String token) { 1 usage new *
33          try {
34              Jwts.parserBuilder().setSigningKey(secret).build().parse(token);
35              return true;
36          } catch (Exception e) {
37              return false;
38          }
39      }

```

Рис. 14 – Клас JWTUtil

```

18     @Component new *
19     @RequiredArgsConstructor
20     public class JWTFilter extends OncePerRequestFilter {
21
22     private final JWTUtil jwtUtil;
23
24     @Override no usages new *
25     protected void doFilterInternal(HttpServletRequest request,
26                                     HttpServletResponse response,
27                                     FilterChain filterChain) throws ServletException, IOException {
28
29         String authHeader = request.getHeader("Authorization");
30
31         String username = null;
32         String token = null;
33
34         if (authHeader != null && authHeader.startsWith("Bearer ")) {
35             token = authHeader.substring(7);
36             username = jwtUtil.extractUsername(token);
37         }
38         if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {
39
40             if (jwtUtil.validateToken(token)) {
41                 UsernamePasswordAuthenticationToken auth =
42                     new UsernamePasswordAuthenticationToken(username, null, new ArrayList<>());
43
44                 auth.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
45
46                 SecurityContextHolder.getContext().setAuthentication(auth);
47             }
48         }
49         filterChain.doFilter(request, response);

```

Рис. 15 – Клас JWTFilter

4.3 Двостороннє шифрування

Для забезпечення безпеки даних у розробленому застосунку реалізовано механізм двостороннього шифрування на основі симетричного алгоритму AES. Даний механізм використовується для шифрування та розшифрування вмісту документів перед збереженням у базі даних та під час їх отримання.

Шифрування реалізовано у вигляді окремого сервісу *EncryptionService*, представленого на Рис. 16, який:

- шифрує текст методом *encrypt()*
- виконує зворотну операцію в методі *decrypt()*
- використовує симетричний ключ, що задається в конфігурації

застосунку

Перед збереженням документа вміст перетворюється у зашифрований вигляд, а при відкритті — автоматично розшифровується. Таким чином забезпечується конфіденційність збережених даних навіть у разі отримання несанкціонованого доступу до бази. Двостороннє шифрування повністю

відповідає вимогам до SOA-додатків щодо захисту переданих і збережених даних.

```
13  public class EncryptionService {  
14  
15      private static final String ALGO = "AES";  1 usage  
16      private final SecretKeySpec keySpec;  3 usages  
17  
18      public EncryptionService(@Value("MySecretKeyForAES") String secretKey) {  new *  
19          byte[] key = Arrays.copyOf(secretKey.getBytes(StandardCharsets.UTF_8),  newLength: 16);  
20          this.keySpec = new SecretKeySpec(key, ALGO);  
21      }  
22  
23      public String encrypt(String plain) {  3 usages  new *  
24          try {  
25              Cipher cipher = Cipher.getInstance( transformation: "AES/ECB/PKCS5Padding");  
26              cipher.init(Cipher.ENCRYPT_MODE, keySpec);  
27              byte[] enc = cipher.doFinal(plain.getBytes(StandardCharsets.UTF_8));  
28              return Base64.getEncoder().encodeToString(enc);  
29          } catch (Exception e) {  
30              throw new RuntimeException("Encrypt error", e);  
31          }  
32      }  
33  
34      public String decrypt(String encoded) {  2 usages  new *  
35          try {  
36              Cipher cipher = Cipher.getInstance( transformation: "AES/ECB/PKCS5Padding");  
37              cipher.init(Cipher.DECRYPT_MODE, keySpec);  
38              byte[] dec = cipher.doFinal(Base64.getDecoder().decode(encoded));  
39              return new String(dec, StandardCharsets.UTF_8);  
40          } catch (Exception e) {  
41              throw new RuntimeException("Decrypt error", e);  
42          }  
43      }  
44  }
```

Рис. 16 – Сервіс EncryptionService

5 Діаграма класів архітектури «SOA»

Діаграма класів для архітектури «SOA» зображена на Рисунку 17:



Рис. 17 – Діаграма класів для архітектури «SOA»

Представлена діаграма класів демонструє архітектуру серверної частини текстового редактора, спроектовану відповідно до принципів SOA. Система складається з кількох незалежних сервісів, кожен з яких надає окремі функціональні можливості, та набору контролерів, що забезпечують доступ до цих сервісів через REST API.

Сервісний шар:

Сервіси відповідають за бізнес-логіку програми. Кожен з них інкапсулює свої операції та не залежить від UI чи інших компонентів.

- DocumentService та DocumentServiceImpl

- MacroService
- BookmarkService & BookmarkServiceImpl
- SnippetService & SnippetServiceImpl
- HintGenerator
- UserService
- EncryptionService
- FlyweightService

Контролерний шар:

Контролери виконують роль SOA-веб-сервісів, які приймають HTTP-запити й передають їх у відповідні сервіси.

- DocumentController
- MacroController
- BookmarkController
- SnippetController
- HintController
- AuthController
- CryptoController

Діаграма відображає модульну, сервіс-орієнтовану архітектуру, де кожен компонент має чітко визначену відповіальність і може бути використаний незалежно через REST API. Такий підхід повністю відповідає принципам SOA та спрощує масштабування застосунку.

Посилання на GitHub: <https://github.com/chaikovsska/textEditor>

6 Висновки

У ході виконання лабораторної роботи було реалізовано сервіс-орієнтовану архітектуру у вигляді набору автономних REST-сервісів, кожен з яких відповідає за окрему функціональність текстового редактора. Розроблена архітектура продемонструвала ключові властивості SOA: слабке зв'язування, повторне використання сервісів, можливість незалежного розгортання та чітко визначені контракти взаємодії. Робота дозволила закріпити

розуміння відмінностей між різними підходами побудови розподілених систем та практично застосувати принципи сервіс-орієнтованого проєктування.

7 Відповіді на питання

1. Що таке клієнт-серверна архітектура?

Клієнт-серверна архітектура — це модель організації комп’ютерних систем, де клієнти роблять запити на сервер, який обробляє ці запити і повертає результати. Клієнт відповідає за інтерфейс користувача та ініціацію запитів, сервер — за обробку даних, зберігання та логіку.

2. Розкажіть про сервіс-орієнтовану архітектуру.

SOA — це архітектурний підхід, у якому система складається з набору незалежних сервісів, які:

- виконують конкретні функції,
- доступні через мережеві протоколи (HTTP, SOAP),
- можуть використовуватися різними клієнтами.

Кожен сервіс автономний і надає чітко визначені можливості через API.

3. Якими принципами керується SOA?

Основні принципи SOA:

- Автономність сервісів — сервіси працюють незалежно.
- Стандартизований інтерфейси — сервіси взаємодіють через визначені API.
- Повторне використання — сервіси можна використовувати в різних системах.
- Легко інтегрувати — сервіси повинні легко поєднуватись у складні процеси.
- Слабке зв’язування — зміни в одному сервісі мінімально впливають на інші.

4. Як між собою взаємодіють сервіси в SOA?

Сервіси взаємодіють через стандартизовані повідомлення або API, наприклад через SOAP або REST. Кожен сервіс публікує свій інтерфейс (WSDL,

OpenAPI), і інші сервіси можуть викликати його методи, обмінюючись даними у формі XML, JSON або інших форматах.

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

- Через документацію (Swagger/OpenAPI, Postman Collections, WSDL).
- Через Service Registry (якщо використовується) – каталог доступних сервісів.
- Через внутрішні API-портали компанії.
- Через узгоджений контракт між командами.

Після цього запити виконуються через REST-клієнти, HTTP або SDK.

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

Переваги:

- централізоване зберігання даних;
- висока безпека;
- просте адміністрування;
- клієнти можуть бути різними (мобільні, десктопні).

Недоліки:

- сервер — єдина точка відмови;
- потрібні ресурси для підтримки сервера;
- велике навантаження зменшує продуктивність.

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

Переваги:

- відсутність центрального сервера;
- висока масштабованість;
- економічність (кожен комп'ютер розподіляє навантаження).

Недоліки:

- складніше забезпечити безпеку;
- складно адмініструвати;

- низька надійність через зміну стану вузлів.

8. Що таке мікро-сервісна архітектура?

Мікросервісна архітектура — це підхід, де додаток складається з малих, незалежних сервісів, кожен з яких реалізує одну бізнес-функцію і може розгортатися окремо. Вона є розвитком SOA, але з більш дрібними і автономними компонентами.

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

Основні протоколи:

- HTTP/HTTPS + REST
- gRPC
- SOAP
- Message brokers: RabbitMQ, Kafka, MQTT для асинхронної взаємодії
- WebSockets для реального часу

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?

Це не повноцінна SOA, а локальне використання сервісів всередині одного додатку. SOA передбачає автономні, незалежні сервіси, доступні через стандартизовані протоколи для інших систем. Такий підхід — це архітектурний патерн «сервісний шар» (Service Layer), який організовує бізнес-логіку всередині одного проєкту.