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 »

Виконала
студентка групи ІА–22:
Фоменко Альона

Перевірив:
Мягкий Михайло Юрійович

Київ 2024

Зміст

Тема: РІЗНІ ВИДИ ВЗАЄМОДІЇ ДОДАТКІВ: CLIENT-SERVER, PEER-TO-PEER, SERVICE-ORIENTED ARCHITECTURE

Мета: Ознайомитися з короткими теоретичними відомостями. Реалізувати частину функціоналу робочої програми у вигляді класів та їх взаємодій для досягнення конкретних функціональних можливостей. Застосування одного з розглянутих шаблонів при реалізації програми

Теоретичні відомості

Клієнт-серверні додатки

Клієнт-серверні додатки — це найпростіший вид розподілених додатків, які поділяються на:

- Клієнт — забезпечує взаємодію з користувачем.
- Сервер — відповідає за зберігання та обробку даних.

Клієнти бувають:

- Тонкі клієнти — більшість обчислень виконує сервер. Використовуються в сценаріях, де потрібна централізація обробки або захист даних.
- Товсті клієнти — основна обробка даних виконується на стороні клієнта, що розвантажує сервер.

Моделі взаємодії:

- Модель "підписки/видачі" — клієнти отримують повідомлення від сервера про певні події. Зручно для автоматичного оновлення даних.

Клієнт-серверна архітектура часто реалізується за допомогою трирівневої структури:

- Клієнтська частина — візуалізація, логіка користувача.
- Загальна частина (middleware) — спільні класи і функції.
- Серверна частина — бізнес-логіка, управління даними.

Додатки типу "peer-to-peer"

У peer-to-peer (P2P) додатках:

- Всі програми є рівноправними та обмінюються даними напряму.
- Проблеми:
    - Синхронізація даних (використовуються hash-алгоритми або договори синхронізації).
    - Пошук клієнтів (здійснюється через загальну адресну книгу або алгоритми).

Сервіс-орієнтована архітектура (SOA)

SOA — модульний підхід до розробки, базований на використанні незалежних компонентів із стандартизованими інтерфейсами. Характеристики:

- Використання веб-служб (SOAP, REST тощо).
- Інкапсуляція деталей реалізації для забезпечення повторного використання компонентів.
- Незалежність від платформи, масштабованість, легкість керування.

SaaS (Software as a Service):

- Доступ до програм через Інтернет.
- Характеристики:
    - Відсутність витрат на установку та оновлення.
    - Оплата за користування (абонплата або за операції).
    - Модернізація та технічна підтримка включені.
- Заснований на SOA і зазвичай працює у хмарі.

Мікросервісна архітектура

Мікросервіси — це серверні додатки, що складаються з невеликих незалежних служб. Особливості:

- Кожна служба автономна, виконує конкретну бізнес-логіку та взаємодіє через стандартизовані протоколи (HTTP, WebSockets тощо).
- Забезпечують високу гнучкість і масштабованість.
- Підходять для комплексних систем з можливістю незалежного розгортання окремих служб.

Хід роботи

Тема 11: Web crawler (proxy, chain of responsibility, memento, template method, composite, p2p)

Веб-сканер повинен вміти розпізнавати структуру сторінок сайту, переходити за посиланнями, збирати необхідну інформацію про зазначений термін, видаляти не семантичні одиниці (рекламу, об'єкти javascript і т.д.), зберігати знайдені дані у вигляді структурованого набору html файлів вести статистику відвіданих сайтів і метадані.

1. Реалізувати не менше 3-х класів відповідно до обраної теми.

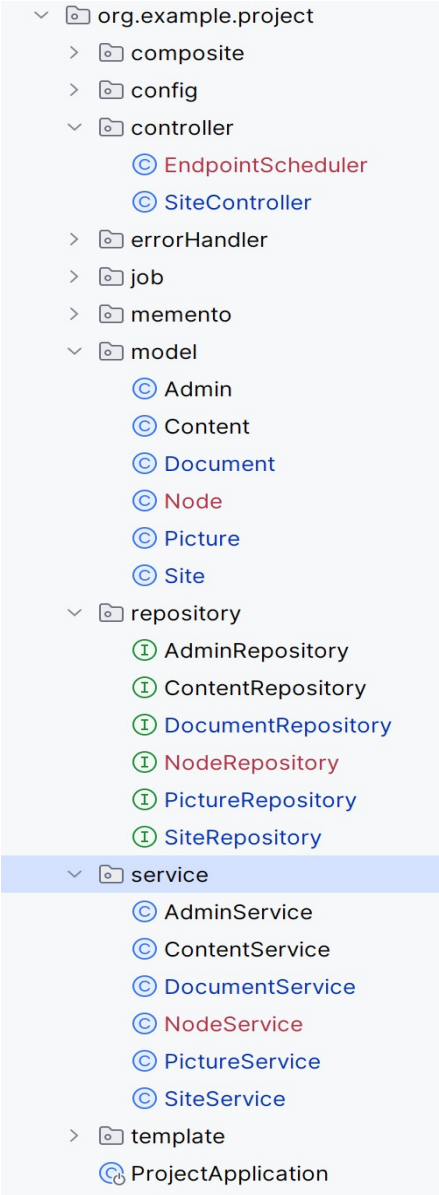


Рис. 1: Структура проекту

У ході лабораторної роботи була реалізована система кравлера, де основним завданням було використання p2p.

2. Реалізація P2P

Призначення P2P (peer-to-peer) полягає в організації децентралізованої мережі, де всі вузли мають рівноправний статус і можуть виступати як споживачами, так і постачальниками ресурсів. Це забезпечує розподіл завдань, масштабованість, балансування навантаження та відмовостійкість, оскільки мережа не залежить від центрального вузла. У контексті краулінгу P2P дозволяє створювати розподілені системи для збору та обробки даних, де вузли обмінюються інформацією безпосередньо один з одним.

P2P ідеально підходить для розподілених краулерів, забезпечуючи ефективну організацію взаємодії між вузлами, балансування навантаження та синхронізацію даних. Завдяки цьому патерну, краулер стає масштабованим, стійким до збоїв та гнучким у розширенні.

Клас EndpointScheduler

```java
@Component
public class EndpointScheduler {

    5 usages
    private final RestTemplate restTemplate;

    @Autowired
    DocumentService documentService;
    @Autowired
    SiteService siteService;
    @Autowired
    NodeService nodeService;
    @Autowired
    PictureService pictureService;

    @Autowired
    public EndpointScheduler(RestTemplate restTemplate) { this.restTemplate = restTemplate; }
```

Рисунок 1: Клас EndpointScheduler частина 1

```java
1 usage
public void syncSites(Node node){
    String lastSyncSiteDate = node.getSyncSiteDate().toString().replace( target: " ", replacement: "%20");
    String url1 = "http://" + node.getIp() + ":" + node.getPort() + "/getsitesfromnode/" + lastSyncSiteDate ;
    System.out.println(url1);
    try {
        String response = restTemplate.getForObject(url1, String.class);
        //   System.out.println("Response: " + response);
        ObjectMapper objectMapper = new ObjectMapper();
        List<Site> sites = objectMapper.readValue(response, new TypeReference<List<Site>>() {});
        Date lastDate = node.getSyncSiteDate();
        for (Site site : sites) {
            siteService.addSyncSite(site.getId(), site.getUrl(), site.getTitle(),
                    site.getInsertDate().toString(), site.getDocumentCount(), site.getPictureCount());
            if (lastDate == null || site.getInsertDate().compareTo(lastDate) > 0){
                lastDate = site.getInsertDate();
            }

        }
        System.out.println("lastDate" + lastDate);
        nodeService.updateSiteSyncDate(node.getId(), lastDate);

    } catch (Exception e) {
        System.err.println("Error to call endpoint: " + e.getMessage());
    }
}
```

Рисунок 2: Клас EndpointScheduler частина 2

```
public void syncDocs (Node node){
    String lastSyncDate = node.getSyncDocDate().toString().replace( target: " ", replacement: "%20");
    System.out.println(lastSyncDate + "replaced /////");
    String url = "http://" + node.getIp() + ":" + node.getPort() + "/getdocumentsfromnode/" + lastSyncDate;
    System.out.println(url);
    try {
        String response = restTemplate.getForObject(url, String.class);
        //   System.out.println("Response: " + response);
        ObjectMapper objectMapper = new ObjectMapper();
        List<Document> documents = objectMapper.readValue(response, new TypeReference<List<Document>>() {
        });
        Date lastDate = node.getSyncDocDate();

        for (Document document : documents) {
            //  System.out.println("document ----- " + document.getUrl());
            documentService.addSyncDoc(document.getSiteId(), document.getUrl(),
                    document.getParentUrl(), document.getStatus(), document.getLevel(), document.getInsertDate().toString(),
            if (lastDate == null || document.getInsertDate().compareTo(lastDate) > 0) {
                //   System.out.println("document.getInsertDate()  " + document.getInsertDate());
                lastDate = document.getInsertDate();
            }

        }
        System.out.println("lastDate" + lastDate);
        nodeService.updateDocSyncDate(node.getId(), lastDate);

    } catch (Exception e) {
        System.err.println("Error to call endpoint: " + e.getMessage());
    }
}
```

Рисунок 3: Клас EndpointScheduler частина 3

```
1 usage
public void syncContent (Node node){
    String lastSyncDate = node.getSyncContentDate().toString().replace( target: " ", replacement: "%20");
    System.out.println(lastSyncDate + "replaced /////");
    String url = "http://" + node.getIp() + ":" + node.getPort() + "/getcontentfromnode/" + lastSyncDate;
    System.out.println(url);
    try {
        String response = restTemplate.getForObject(url, String.class);
        ObjectMapper objectMapper = new ObjectMapper();
        List<Document> documentList = objectMapper.readValue(response, new TypeReference<List<Document>>() {
        });
        Date lastDate = node.getSyncContentDate();
        for (Document document : documentList) {
            documentService.addSyncContent( document.getUrl(), document.getTitle(), document.getScanDate().toString(),
            if (lastDate == null || document.getScanDate().compareTo(lastDate) > 0) {
                lastDate = document.getScanDate();
            }

        }
        System.out.println("lastDate" + lastDate);
        nodeService.updateContentSyncDate(node.getId(), lastDate);

    } catch (Exception e) {
        System.err.println("Error to call endpoint: " + e.getMessage());
    }
}
```

Рисунок 4: Клас EndpointScheduler частина 4

```
1 usage
public void syncPics (Node node){
    String lastSyncDate = node.getSyncPicDate().toString().replace( target: " ", replacement: "%20");
    String url = "http://" + node.getIp() + ":" + node.getPort() + "/getpicsfromnode/" + lastSyncDate;
    System.out.println(url);
    try {
        String response = restTemplate.getForObject(url, String.class);
        ObjectMapper objectMapper = new ObjectMapper();
        List<Picture> pictureList = objectMapper.readValue(response, new TypeReference<List<Picture>>() {
        });
        Date lastDate = node.getSyncPicDate();

        for (Picture picture : pictureList) {
            pictureService.addSyncPics(picture.getSiteId(), picture.getUrl(),picture.getParentUrl(),picture.getInsertDate()
              if (lastDate == null || picture.getInsertDate().compareTo(lastDate) > 0) {
                lastDate = picture.getInsertDate();
            }

        }
        System.out.println("lastDate" + lastDate);
        nodeService.updatePicSyncDate(node.getId(), lastDate);

    } catch (Exception e) {
        System.err.println("Error to call endpoint: " + e.getMessage());
    }
}
```

Рисунок 5: Клас EndpointScheduler частина 5

```java
@Async
@Scheduled(fixedDelay = 60000)
public void getDocumentsFromNode() {
    System.out.println("nodesync");
    List<Node> nodes  = nodeService.getAllNodes();
    for (Node node : nodes) {

        syncSites(node);

        syncDocs(node);

        syncContent(node);

        syncPics(node);

    }
}
```

Рисунок 6: Клас EndpointScheduler
частина 6

- Призначення: Це компонент, який відповідає за синхронізацію даних між вузлами (Node) та основною системою, використовуючи HTTP-запити до API вузлів.
- Основні методи:
    - syncSites(Node node) — отримує сайти із зазначеного вузла та зберігає їх у базі.
    - syncDocs(Node node) — отримує документи із вузла.
    - syncContent(Node node) — отримує вміст документів із вузла.
    - syncPics(Node node) — отримує зображення із вузла.

- Технології: Використовує RestTemplate для виконання HTTP-запитів і ObjectMapper для десеріалізації JSON-даних.

Клас Node

```java
import ...

12 usages
@Data
@Entity
public class Node {
    @Id
    private  int id;
    private  String ip;
    private  String port;
    private Date syncDocDate;
    private  String status;
    private Date syncPicDate;
    private Date syncContentDate;
    private Date syncSiteDate;

    2 usages
    public Date getSyncSiteDate() { return syncSiteDate; }

    no usages
    public void setSyncSiteDate(Date syncSiteDate) { this.syncSiteDate = syncSiteDate; }

    public int getId() { return id; }

    public void setId(int id) { this.id = id; }

    4 usages
    public String getIp() { return ip; }

    no usages
    public void setIp(String ip) { this.ip = ip; }

    4 usages
    public String getPort() { return port; }
```

Рисунок 7: Клас Node

- Призначення: Модель бази даних для представлення вузла системи.
- Основні поля:
    - id — ідентифікатор вузла.
    - ip та port — адреса вузла.
    - syncDocDate, syncSiteDate, syncContentDate, syncPicDate — останні дати синхронізації для різних типів даних.
    - status — статус вузла.
- Анотації: Використовує @Entity для зв’язку з базою даних і @Data для автоматичної генерації гетерів/сетерів.
- Методи: Гетери/сетери для всіх полів.

Інтерфейс NodeRepository

```java
@RepositoryRestResource
public interface NodeRepository extends JpaRepository<Node, Long> {

    1 usage
    @Query(value = "SELECT * FROM trpz.get_all_nodes()", nativeQuery = true)
    List<Node> getAllNodes();

    1 usage
    @Query(value = "SELECT * FROM trpz.update_doc_sync_date(?1, ?2)", nativeQuery = true)
    void updateDocSyncDate(int nodeId, Date lastDate);

    1 usage
    @Query(value = "SELECT * FROM trpz.update_site_sync_date(?1, ?2)", nativeQuery = true)
    void updateSiteSyncDate(int nodeId, Date lastDate);

    1 usage
    @Query(value = "SELECT * FROM trpz.update_pic_sync_date(?1, ?2)", nativeQuery = true)
    void updatePicSyncDate(int nodeId, Date lastDate);

    1 usage
    @Query(value = "SELECT * FROM trpz.update_content_sync_date(?1, ?2)", nativeQuery = true)
    void updateContentSyncDate(int nodeId, Date lastDate);
```

Рисунок 8: Інтерфейс NodeRepository

- Призначення: Виконує взаємодію з базою даних для вузлів.
- Основні методи:
    - getAllNodes() — отримує список усіх вузлів.
    - updateDocSyncDate(int nodeId, Date lastDate) — оновлює дату синхронізації документів для вузла.
    - updateSiteSyncDate(int nodeId, Date lastDate) — оновлює дату синхронізації сайтів.
    - updatePicSyncDate(int nodeId, Date lastDate) — оновлює дату синхронізації зображень.
    - updateContentSyncDate(int nodeId, Date lastDate) — оновлює дату синхронізації контенту.

Клас NodeService

Рисунок 9: Клас NodeService

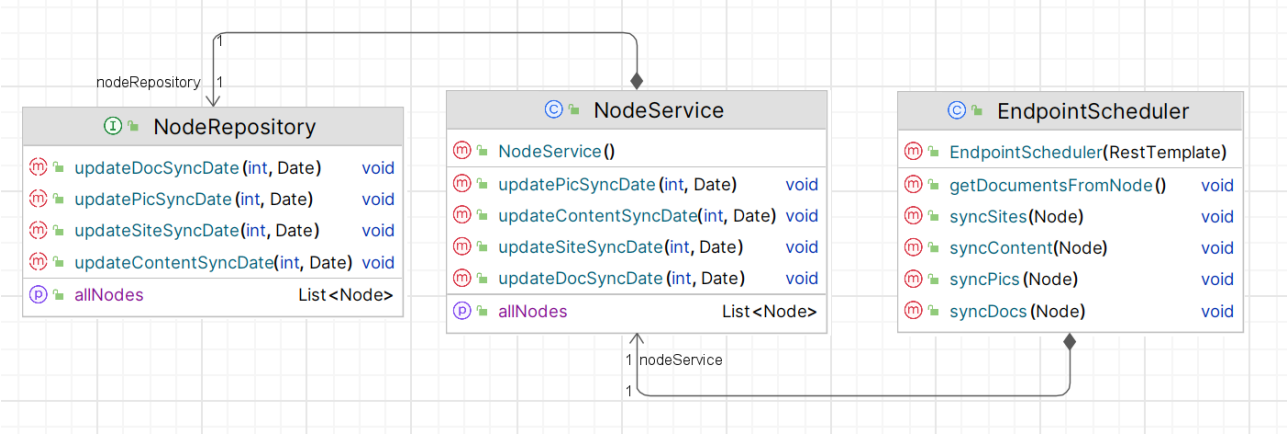
- Призначення: Сервісний клас для бізнес-логіки, що стосується вузлів.
- Основні методи:
    - getAllNodes() — отримує всі вузли з репозиторію.
    - updateDocSyncDate, updateSiteSyncDate, updateContentSyncDate, updatePicSyncDate — викликають відповідні методи репозиторію для оновлення дат синхронізації.

Проблема, яку вирішує P2P

Проблема, яку вирішує **P2P**, полягає в усуненні залежності від центрального сервера та забезпеченні децентралізації. У традиційних клієнт-серверних системах центральний сервер обробляє всі запити, що створює кілька серйозних проблем. По-перше, сервер стає єдиною точкою відмови: якщо він виходить з ладу, вся система стає недоступною.

По-друге, масштабованість обмежена, оскільки зі збільшенням кількості клієнтів навантаження на сервер зростає, що може сповільнювати роботу системи. По-третє, утримання та забезпечення продуктивності такого сервера вимагає значних фінансових витрат. Додатково, централізовані системи легко піддаються цензурі та контролю з боку організацій або влади, а також є мішенню для атак, таких як DDoS, які можуть паралізувати систему.

Діаграма класів



Чому саме p2p підходить для кравлера

Патерн P2P (peer-to-peer) є ідеальним вибором для організації краулера завдяки своїй здатності забезпечувати розподілену та децентралізовану архітектуру. У контексті веб-краулінгу цей підхід дозволяє об'єднати вузли в мережу, де кожен вузол не лише виконує свої локальні завдання, але й взаємодіє з іншими вузлами для синхронізації даних. Це забезпечує гнучкість, масштабованість і стійкість системи до збоїв.

Переваги використання патерна P2P у краулері:

1. Розподіленість та децентралізація. У P2P кожен вузол є рівноправним, тобто може бути як постачальником, так і споживачем даних. Це дає змогу уникнути перевантаження центрального сервера, що є критичним для великих краулінгових задач. Обробка даних розподіляється між численними вузлами, завдяки чому система стає більш масштабованою та стійкою.

2. Синхронізація даних між вузлами. У системі P2P дані передаються безпосередньо між вузлами. Для краулера це означає можливість синхронізації вузлів через HTTP-запити для обміну інформацією про:

- Сайти.
- Документи.
- Контент.
- Зображення.

Ноди дозволяють передавати лише нові або оновлені дані, ґрунтуючись на останній даті синхронізації. Це оптимізує обсяг передачі даних та гарантує їх актуальність.

3. Гнучкість і адаптація до розподіленого середовища. HTTP-ендпойнти використовуються як основний механізм взаємодії між вузлами, що забезпечує:

- Надійність: HTTP дозволяє отримувати чіткі статуси успішності або помилки.
- Універсальність: Вузли можуть мати різні характеристики, але взаємодіяти через спільний протокол.
- Простоту розширення: Додавання нових вузлів не вимагає значних змін у системі.

4. Простота реалізації через HTTP-ендпойнти. HTTP-запити забезпечують стандартизований спосіб передачі даних. Використання REST-запитів дозволяє:

- Передавати лише оновлені дані за допомогою параметрів, таких як lastSyncDate.

- Використовувати формати, зручні для обміну даними, наприклад, JSON.
- Гарантувати передачу даних, навіть якщо процес переривається: при повторній спробі синхронізація відновиться.

5. Стійкість до збоїв. Розподілена природа P2P дозволяє системі продовжувати роботу навіть у разі виходу з ладу одного або кількох вузлів. Інші вузли мережі зберігатимуть актуальність даних та забезпечуватимуть синхронізацію без участі недоступних вузлів.

Висновок

У даній лабораторній роботі я реалізувала архітектуру на основі P2P для вирішення проблем децентралізації та підвищення стійкості системи. Завдяки використанню рівноправних вузлів було усунуто залежність від центрального сервера, що забезпечило розподіл навантаження, стійкість до збоїв і атак, а також масштабованість системи. Це зробило систему більш гнучкою, ефективною та незалежною, що є ключовими перевагами при роботі з сучасними розподіленими мережами. Реалізація P2P-підходу покращила надійність і доступність системи, спростила її розвиток і підтримку, створюючи базу для побудови стійких і динамічних рішень.