



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: “SOA”

Виконав:

студент групи ІА-14

Рисаков Богдан

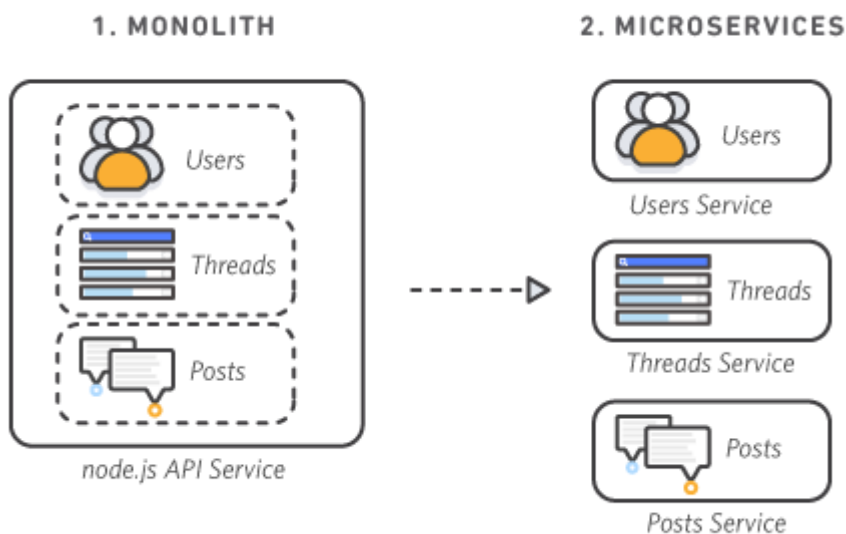
Перевірив:

Мягкий Михайло Юрійович

Київ, 2023

Мета: Реалізувати SOA у проєкті на тему System Activity Monitor

Сервіс-орієнтована архітектура (SOA) - це метод розроблення програмного забезпечення, який використовує програмні компоненти, звані сервісами, для створення бізнес-додатків. Кожен сервіс надає бізнес-можливості, і сервіси також можуть взаємодіяти один з одним на різних платформах і мовах. Розробники застосовують SOA для багаторазового використання сервісів у різних системах або об'єднання кількох незалежних сервісів для виконання складних завдань.



Приклад даного підходу до архітектури. [Reference](#)

Переваги:

- Гнучкість та Масштабованість
- Модульність
- Повторне Використання Компонентів
- Відокремлення Інтерфейсу від Виконання
- Мовна Незалежність

Недоліки:

- Складність Управління
- Залежність від Мережі
- Вартість інфраструктури та Управління
- Проблеми Продуктивності

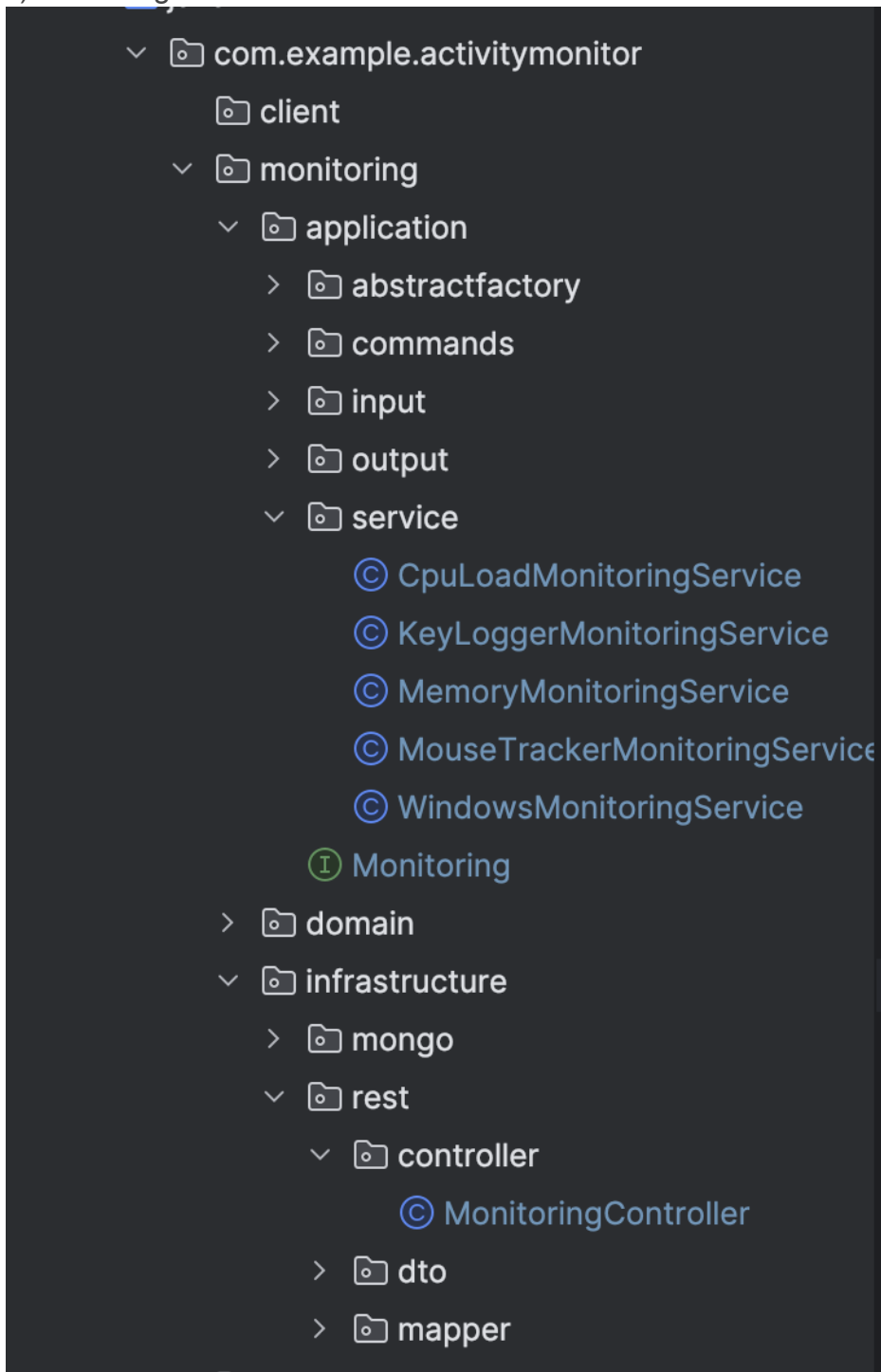
Реалізація:

Так як SOA - архітектурне рішення то у рамках реалізації я надам структуру проекту та укажу як саме сервіси повинні спілкуватися між собою.

Тож архітектура:

Відповідно до бізнес-задачі програма розділена на 2 частини: report та monitoring

1)Monitoring



Тут під кожну задачу є свій сервіс. Щоб використати їх треба звернутися до Monitoring Controller:

```
@RestController("/monitoring")
public class MonitoringController {

    private final Map<String, AbstractMonitor> monitorMap;

    public MonitoringController(@Autowired List<AbstractMonitor> monitorList) {
        this.monitorMap = monitorList
            .stream()
            .collect(Collectors.toMap(AbstractMonitor::getOsType,
Function.identity()));
    }

    @PostMapping
    public ResponseEntity<Void> startMonitoring(@RequestBody MonitoringRequestDto
monitoringRequestDto) {
        String monitoringType = monitoringRequestDto.getMonitoringType();
        String osType = monitoringRequestDto.getOsType();

        AbstractMonitor abstractMonitor = monitorMap.get(osType);

        abstractMonitor.startMonitoring(monitoringType);

        return ResponseEntity.ok().build();
    }
}
```

Цей контролер отримує Request :

```
@Getter
@Setter
public class MonitoringRequestDto {

    private String monitoringType;
    private String osType;
















    public MonitoringRequestDto(String monitoringType, String osType) {
        this.monitoringType = monitoringType;
        this.osType = osType;
    }
}
```

Та по ньому визначає який саме сервіс використати.

Сам сервіс напряму використаний бути не може -тільки через клас - Monitor який містить в собі сервіси для конкретної ОС.

Чому це гарний підхід: цей контролер більше не буде змінюватись, (звідно зараз він нічого не робить та зміни будуть під час курсової) тобто при додаванні нового сервісу чи Монітору під ОС усі інші компоненти програми цього не помітять.

Зі сторони звітів все буде теж саме:

- ▼  report
 - ▼  application
 - >  bridge
 - >  cpuload.model
 - >  iterator
 - >  keylogger.model
 - >  memory.model
 - >  mousetracker.model
 - ▼  visitor
 - © ReportByTimeGenerator
 - © ReportStartStopGenerator
 - ⓘ ReportVisitor
 - © ScheduledReportGenerator
 - >  windows.model
 - ⓘ ReportRequest
 - >  domain
 - ▼  infrastructure.rest
 - ▼  controller
 - © ReportController
 - ▼  dto
 - © ReportRequestDto
 - >  mapper

Тільки для формування звітів ми використали паттерн Visitor:

```
@RestController("/report")
public class ReportController {

    private Map<String, ReportVisitor> reportVisitorMap;

    private Map<String, AbstractMonitor> monitorMap;

    public ReportController(List<ReportVisitor> reportVisitorMap,
List<AbstractMonitor> monitorList) {

        this.reportVisitorMap = reportVisitorMap.stream()
.collect(java.util.stream.Collectors.toMap(ReportVisitor::getReportName,
        java.util.function.Function.identity()));

        this.monitorMap = monitorList
            .stream()
            .collect(Collectors.toMap(AbstractMonitor::getOsType,
Function.identity()));
    }

    @PostMapping("")
    public ResponseEntity<Report> getReport(@RequestBody ReportRequestDto
reportRequestDto) {
        String reportType = reportRequestDto.getReportType();

        AbstractMonitor currentMonitor =
monitorMap.get(reportRequestDto.getOsType());

        ReportVisitor reportVisitor = reportVisitorMap.get(reportType);

currentMonitor.getConcreteMonitoring(reportRequestDto).accept(reportVisitor);

        return ResponseEntity.noContent().build();
    }
}
```

Тож тут також:

- Внедряється Monitor для ОС.
- Знаходиться Visitor для конкретного репорту
- Конкретний моніторинг приймає відвідувача та віддає звіт

Без використання Spring Dependency Injection через класи - біни ці контролери не будуть працювати, але SOA не патерн а більш специфікація архітектури.

Тож без використання Spring будуть або порушені принципи ООП або написання такої архітектури буде ДУЖЕ складної та по суті буде використана погана копія Spring

Висновок: Я переписав архітектуру проекту зробивши її SOA