

Вариант 1

1. Варианты реализации Service Discovery

Service Discovery (Обнаружение сервисов) — автоматическое обнаружение, отслеживание и контроль состояния устройств и сервисов в компьютерной сети. Он создан для того, чтобы с минимальными затратами можно было подключить новое приложение/сервис в уже существующее окружение.

Решаемые задачи

- Реконфигурация системы
- Упрощение администрирования
- Горизонтальная масштабируемость

Варианты реализации:

- Инфраструктурное ПО
 - SD конфигурируется на уровне управления ВМ/контейнеров (e.g. docker, kubernetes)
- Прозрачно силами платформы (например напрямую в Java EE)
- Используя фреймворки (e.g. Spring Cloud)
- Специальное прикладное ПО (Consul, Zookeeper)

Это load balancer! (Или 2 способа использования SD load balancer-ом)

Существует два типа SD:

- client-side discovery - клиент запрашивает информацию о доступе к доступным экземплярам у SD, а дальше сам распределяет нагрузку между ними
- server-side discovery - клиент использует посредника (SD) для запроса каталога услуг и выполнения запросов к ним

Примеры реализации:

- ZooKeeper
 - написан на Java,
 - крепок и стабилен, но тяжеловат в настройке.
 - устарел, почти не используется (в т.ч. требует понижение версии спринга, чтобы все заработало)
- Consul
 - Server-side
 - чуть более тяжелый и медленный, чем ZooKeeper
 - Распределенный Service Discovery (ПО ставится на каждый комп)
 - работает по принципу ключ-значение
 - предлагает веб-интерфейс для работы со списками сервисов

- Eureka Server
 - Clients-side
 - Обязанность — давать имена каждому микросервису.
 - Каждый сервис регистрируется в Eureka и отправляет эхо-запрос серверу Eureka, чтобы сообщить, что он активен.
 - Для этого сервис должен быть помечен как @EnableEurekaClient (или общее - @EnableDiscoveryClient), а сервер @EnableEurekaServer.

2. Структура реестра UDDI

UDDI: Universal Description, Discovery and Integration — централизованное хранилище дескрипторов WSDL со стандартизированным API. Структура состоит из:

- Белые страницы - адрес, контакты и известные идентификаторы;
 - Основная информация о компании и ее бизнесе,
 - Основная контактная информация, включая название компании, адрес, контактный телефон,
 - Уникальные идентификаторы для налоговых идентификаторов компании
- Желтые страницы - категоризация;
 - более подробная информация о компании (описания электронных возможностей, которые компания может предложить любому, кто хочет иметь с ней дело)
 - Желтые страницы используют общепринятые схемы промышленной классификации, отраслевые коды, коды идентификации предприятий
 - Почтовые и географические индексы
- Зеленые страницы - техническая информация о сервисе.
 - Различные интерфейсы
 - API
 - Информация об обнаружении и аналогичные данные, необходимые для поиска, связи и запуска веб-службы

Помимо этого, имеют место четыре типа записей:

- Business Entity — описывает бизнес, предоставивший данный сервис, эта запись включает информацию о категории, помогающую поисковым системам выполнять поиск для определенного типа бизнеса.
- Business Service — класс сервисов внутри бизнеса. Каждый бизнес-сервис принадлежит нескольким бизнес-записям Business Entity.
- Binding Template (шаблон связывания) и Technology Model (технологическая модель) совместно определяют веб-сервис, как это описано в WSDL. Технологическая

модель соответствует абстрактному описанию, а шаблон связывания соответствует протоколу.

3. Сервис на JAX-WS реализующий DNS

```
package dnsservice.endpoint;

import javax.xml.ws.WebService;
import javax.xml.ws.WebMethod;
import javax.xml.ws.WebParam;

@WebService(name = "DNS")
public class DNS {
    private Map<String, String> dnsRecords = new HashMap<>();

    public void DNS () {};

    @WebMethod
    public String getIPAddress(@WebParam(name = "domain") String domain) {
        return dnsRecords.get(domain);
    }

    @WebMethod
    public String updateIPAddress(@WebParam(name = "domain") String domain,
                                @WebParam(name = "ip") String ip) {
        dnsRecord.put(domain, ip);
        return dnsRecords.get(domain);
    }

    @WebMethod
    public String deleteIPAddress(@WebParam(name = "domain") String domain) {
        //...if contains, .remove(domain)..., else, ...
    }

    @WebMethod
    public String addIPAddress(@WebParam(name = "domain") String domain,
                              @WebParam(name = "ip") String ip) {
        //...
    }
}
```

Публикуем сервис:

```
public class DNSServicePublisher {
    public static void main(String[] args) {
        Endpoint.publish(
```

```
        "http://localhost:8080/dns",  
        new DNS()  
    );  
}  
}
```

Вариант 2

1. Service Discovery на Java EE и его реализации

Все сервисы “by design” регистрируются в JNDI и доступны через CDI

- JNDI (Java Naming and Directory Interface) - Java API, которое предоставляет единообразный механизм взаимодействия Java-программы с различными службами имен и каталогов. “Под капотом” интеграция между JNDI и любой конкретной службой осуществляется с помощью интерфейса поставщика услуг. JNDI нужен для того, чтобы мы могли из Java-кода получить Java-объект из некоторого “Registry” объектов (как раз JNDI) по имени объекта, привязанного к этому объекту. То есть в JNDI зарегистрированы все микросервисы, и зная “логическое имя”, можно достать физический URI, и дополнительную метainформацию
- CDI — это Contexts and Dependency Injection. Это спецификация Java EE, описывающая внедрение зависимостей (Dependency Injection) и контексты.

Конфигурация сервера приложений может быть распределённой:

- В виде пула экземпляров сервера.
- В виде кластера.

Если логику “под капотом” реализуют EJB, их можно масштабировать.

<По идее надо бы раскрыть, но хз че можно вписать>

Различные реализации Java EE предоставляют дополнительные механизмы для реализации Service Discovery. Например, Apache Tomcat имеет свой механизм обнаружения служб, а JBoss предлагает JMX для обнаружения служб. Также есть несколько реализаций Service Discovery на основе JAX-WS.

2. Понятие микросервиса. Отличия и сходства от других сервисов.

Микросервисы – это шаблон сервис-ориентированной архитектуры, в котором приложения создаются как совокупность различных наименьших независимых (или слабо связанных), легко изменяемых модулей - микросервисов.

Нет четкой спецификации и стандартна, это больше “идеология”.

Микросервис - разновидность SOA, но есть различия:

- **SOA** ориентирована на **повторное использование сервисов приложений**, а **микросервисы – на разъединение**.
- SOA является полнофункциональным по своей природе, а Microservices – монолитным (поэтому MSA **не** предоставляет явной **поддержки для разработки распределенных приложений**)
- **SOA**-приложения созданы для выполнения **многочисленных бизнес-задач**, а в **MSA выполняет одну бизнес-задачу**. Поэтому **SOA лучше для крупномасштабных интеграций**, а **MSA для небольших веб-приложений**.
- **SOA** предполагает **совместное хранение данных** между службами, тогда как в **микросервисах каждая служба** может иметь **независимое хранилище данных**.
- SOA предназначен для **совместного использования ресурсов** между службами, а микросервисы – для размещения **служб**, которые могут функционировать **независимо** (или хотя бы **ограниченный контекст** для связи).
- В архитектуре **SOA DevOps** и **Непрерывная доставка** становятся популярными, но еще **не стали массовыми**, в то время как **микросервисы уделяют большое внимание DevOps** и **Непрерывной доставке**.
- SOA – это **менее масштабируемая архитектура**, а Microservices – **очень масштабируемая архитектура**.

3. Написать двухтарифный счётчик электроэнергии для интеграции его в ИС-предприятия на JAX-WS

```
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;

@WebService(name = "ElectricityMeterService")
public class ElectricityMeterService {

    @WebMethod
    public double getCurrentPowerConsumption(@WebParam(name = "tariff") int tariff) {
        // Реализация метода
    }

    @WebMethod
    public double getTotalPowerConsumption(@WebParam(name = "tariff") int tariff) {
        // Реализация метода
    }

    @WebMethod
    public String setTariff(@WebParam(name = "tariff") int tariff) {
```

```

    // Реализация метода
}

@WebMethod
public String getCurrentTariff(@WebParam(name = "tariff") int tariff) {
    // Реализация метода
}

@WebMethod
public GetInformationAboutTariffsRes getInformationAboutTariffs() {
    // Реализация метода
}
}

```

Вариант 3

1. Концепция Service Discovery.

Service Discovery (Обнаружение сервисов) — автоматическое обнаружение, отслеживание и контроль состояния устройств и сервисов в компьютерной сети. **SD** создан для того, чтобы с минимальными затратами можно подключить новое приложение в уже существующее наше окружение. Используя Service Discovery, мы можем максимально разделить либо контейнер в виде докера, либо виртуальный сервис от того окружения, в котором он запущен.

Также **Service Discovery** позволяет обнаружить сбой и/или отказы. Service Discovery со своей стороны опрашивает зарегистрированное приложение на факт доступности.

Решаемые задачи

- Реконфигурация системы.
- Упрощение администрирования.
- Горизонтальная масштабируемость.
- Общая концепция универсальна (искать можно не только веб-сервисы!)
- Варианты реализации (не являются прямыми альтернативами!):
 - Инфраструктурное ПО (Kubernetes, nginx...).
 - Специальное прикладное ПО (Consul, ZooKeeper, Etcd...).
 - “Прозрачно” силами платформы (Java EE).
 - Фреймворк (Spring Cloud [Netflix]).

Два типа SD и варианты реализации см. [Вариант 1. Вопрос 1](#)

2. Smart endpoints & Dumb pipes в MCA.

Smart endpoints & Dumb pipes (SE & DP) - паттерн MSA для взаимодействия микросервисов по сети.

Начнем с контрпримера: в ESB реализован подход «умный канал и глупые сервисы»: умный канал — шина ESB, глупые сервисы — конечные системы. В противовес ESB, существует подход «**умные сервисы и глупые каналы**», используемый в MCA. Суть в том, что микросервисы (смарт эндпоинты) должны выполнять свою собственную коммуникационную логику, а надстройка (pipes) по передаче сообщений (“from endpoint to endpoint”) должна иметь как можно более легковесную структуру.

В виде каналов в данном подходе выступают различные брокеры:

- **Брокеры сообщений** (message broker) - выступает посредником: преобразует сообщение по одному протоколу от приложения-источника в сообщение протокола приложения-приёмника. Так же брокер, например, проверяет сообщение на ошибки, маршрутизирует, и выполняет еще ряд функций (но тем не менее не бизнес-логику, оставаясь “dumb”)
- **Событийные брокеры** (event broker) - используется для передачи событий по схеме publish-subscribe.

А в виде сервисов — шлюзы (API Gateway), которые скрывают за собой реализацию и выносят наружу для взаимодействия API.

Плюсы:

- Возможность построения слабосвязанных систем
- Гибкость настройки, взаимозаменяемость инструментов взаимодействия
- Быстрота взаимодействия

Минусы:

- Сложность проектирования и реализации системы
- Требуются практики DevOps по настройке окружения
- В связи с общим трендом на микросервисы в корпоративной разработке, стоимость труда специалистов увеличивается

3. Сервис Jax-ws по менеджменту учащихся.

```
@WebService(endpointInterface = "shit.soa.StudentManagmentService")
public class StudentManagmentServiceImpl implements StudentManagmentService {

    @Inject
    private StudentManagmentRepository rep;

    @WebMethod(operationName = "getAllStudents")
```

```

    @WebResult(name="student_list_dto")
    public StudentsListDTO getAllStudents(@WebParam(name="get_students_req_dto")
GetAllStudentsReqDTO req) {
        return new StudentsListDTO().getStudents().addAll(rep.getStudents());
    }

    @WebMehtod(operationName = "getStudent")
    @WebResult(name="student")
    public Student getStudent(@WebParam(name="get_student_req_dto") GetStudentReqDTO req) {
        return rep.getStudent(req.getId)
    }

    @WebMehtod(operationName = "addStudent")
    @WebResult(name="student")
    public Student addStudent(@WebParam(name="add_student_req_dto") AddStudentReqDTO req) {
        return rep.addStudent(req.getName, req.getGroup, ...);
    }

    @WebMehtod(operationName = "updateStudent")
    @WebResult(name="student")
    public Student updateStudent(@WebParam(name="update_student_req_dto") UpdateStudentReqDTO
req) {
        return rep.updateStudent(req.getId, req.getName, req.getGroup, ...);
    }

    ...
}

```

Вариант 4

1. Service Discovery в решениях на базе Spring Cloud Netflix

Spring Cloud Netflix — бандл для построения COA систем на базе Spring Cloud Netflix и интеграции его в стек Netflix OSS посредством автоматической настройки и привязки к Spring Environment. Шаблоны внутри приложения настраиваются легко и быстро с помощью нескольких простых аннотаций, тем самым позволяя создать большие распределенные системы с проверенными в бою компонентами Netflix. Spring Cloud Netflix предоставляет client-side SD - Eureka.

Напомним, что client-side discovery - клиент запрашивает информацию о доступе к доступным экземплярам у SD, а дальше сам распределяет нагрузку между ними. В Client-side SD единственной «фиксированной точкой» в архитектуре является Service Registry, в котором должна регистрироваться каждая служба. Клиент сам имплементирует логику для взаимодействия со службой.

Каждый микросервис регистрируется на сервере Eureka, и Eureka знает все клиентские приложения, работающие на каждом порту и IP-адресе. Eureka Server также известен как Discovery Server.

С Netflix Eureka каждый клиент действует как “сервер” при подключении к сервису: клиент извлекает список всех подключенных экземпляров в service registry и отправляет все дальнейшие запросы к службам с помощью алгоритма балансировки нагрузки.

Обязанность Eureka — давать имена каждому микросервису. Эврика регистрирует микросервисы и отдает их ip другим микросервисам. Таким образом, каждый сервис регистрируется в Eureka и отправляет эхо-запрос серверу Eureka, чтобы сообщить, что он активен. Для этого сервис должен быть помечен как @EnableEurekaClient, а сервер @EnableEurekaServer.

2. Технология JAX-WS и её использование для реализации веб-сервисов

Java API for XML Web Services (JAX-WS) — это прикладной программный интерфейс языка Java для создания веб-служб, входит в спецификацию Java EE.

JAX-WS является заменой технологии JAX-RPC, предоставляя более документо-ориентированную модель сообщений и упрощая разработку веб-служб за счёт использования аннотаций. Основан на разметке класса аннотациями (метапрограммировании), хорошо совместим с EJB.

В JAX-WS вызов операции веб-сервиса представлен протоколом SOAP — протоколом на основе XML. Спецификация SOAP определяет:

- структуру сообщения,
- правила кодирования
- соглашения для представления вызовов и ответов веб-сервиса.

Эти вызовы и ответы передаются как сообщения SOAP (файлы XML) по HTTP.

На стороне сервера разработчик:

- специфицирует операции веб-сервиса, определяя методы в интерфейсе, написанном на Java.

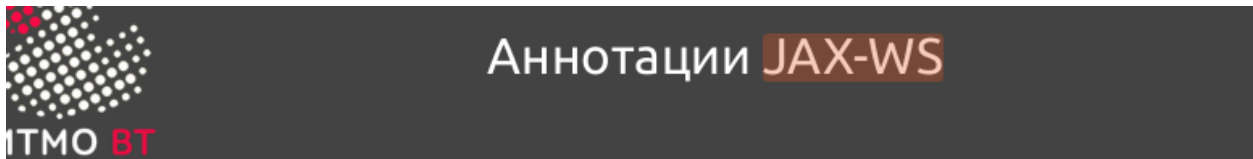
- Пишет классы с реализацией этих методов.

Для реализации клиентских классов, разработчик:

- создает прокси (локальный объект, представляющий сервис),
- затем просто вызывает методы на прокси.

Кроме того, JAX-WS не ограничивает ни одну из сторон в используемых технологиях: клиент JAX-WS может получить доступ к веб-сервису, который не работает в платформе Java, и наоборот. Эта гибкость возможна благодаря использованию в JAX-WS технологий, определённых W3C: HTTP, SOAP и WSDL. WSDL определяет формат XML для описания сервиса как набора конечных точек, работающих с сообщениями.

Можно генерировать: WSDL <-> code (в обе стороны).



- **@WebService** — указывает на то, что Java класс (или интерфейс) является веб-сервисом.
- **@WebMethod** — позволяет настроить то, как будет отображаться метод класса на операцию сервиса.
- **@WebParam** — позволяет настроить то, как будет отображаться конкретный параметр операции на WSDL-часть (part) и XML элемент.
- **@WebResult** — позволяет настроить то, как будет отображаться возвращаемое значение операции на WSDL-часть (part) и XML элемент.
- **@Oneway** — указывает на то, что операция является односторонней, то есть не имеет выходных параметров.
- **@SOAPBinding** — позволяет настроить то, как будет отображаться сервис на протокол SOAP.



Пример сервиса на JAX-WS: интерфейс

```
@WebService
public interface EmployeeService {
    @WebMethod
    Employee getEmployee(int id);
}
```



Пример сервиса на JAX-WS: реализация

```
@WebService(endpointInterface = "com.example.EmployeeService")
public class EmployeeServiceImpl implements EmployeeService {

    @Inject
    private EmployeeRepository employeeRepositoryImpl;

    @WebMethod
    public Employee getEmployee(int id) {
        return employeeRepositoryImpl.getEmployee(id);
    }
}
```



Пример сервиса на JAX-WS: публикация

```
public class EmployeeServicePublisher {
    public static void main(String[] args) {
        Endpoint.publish(
            "http://localhost:8080/employeeservice",
            new EmployeeServiceImpl());
    }
}
```

- 3. Набор микросервисов на Spring MVC, управляющий распределением по аудиториям и преподавателям студентов, ожидающих своей очереди на доп. Должна поддерживаться возможность добавления новых аудиторий и преподавателей и удаления существующих. В одной аудитории может находиться более одного преподавателя.**

Пишем CRUD для аудиторий, это отдельный микросервис.

Controller:

```
@RestController
public class AudienceController {
    private final AudienceService audienceService;

    @Autowired
    public AudienceController(AudienceService audienceService) {
    }

    @GetMapping("/audiences")
    public List<Audience> getAudiences() {
        return audienceService.getAudiences();
    }

    @PostMapping("/audiences")
    public Audience addAudience(@RequestBody Audience audience) {
        return audienceService.addAudience(audience);
    }

    @DeleteMapping("/audiences/{id}")
    public void deleteAudience(@PathVariable Long id) {
        audienceService.deleteAudience(id);
    }
}
```

Application:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
```

```

public class AudienceServiceApplication {

    public static void main(String[] args)
    {
        SpringApplication.run(
            AudienceServiceApplication.class,
            args);
    }
}

```

Пишем CRUD для преподавателей, это отдельный микросервис (по своей структуре он аналогичен, в рубеже можно опустить, но упомянуть).

Controller:

```

@RestController
public class TeacherController {

    //...

}

```

Application:

```

@SpringBootApplication
public class TeacherServiceApplication {
    // ...
}

```

АНАЛОГИЧНО пишем микросервис для студентов

//...

Последний шаг - микросервис, добавляющий студентов и преподавателей в аудитории (создает класс)

```

@RestController(path = "/class")
public class ClassController {
    private final ClassService classService;

    @Autowired
    public ClassController(ClassService classService) {
    }
}

```

```
@GetMapping("/student")
public boolean applyStudent(
    @RequestParam(name = "student_id") Long studentId,
    @RequestParam(name = "audience_id") Long audienceId) {
    return classService.applyStudent(studentId, audienceId);
}
```

Ну

```
//...И так же препода добавляем
```

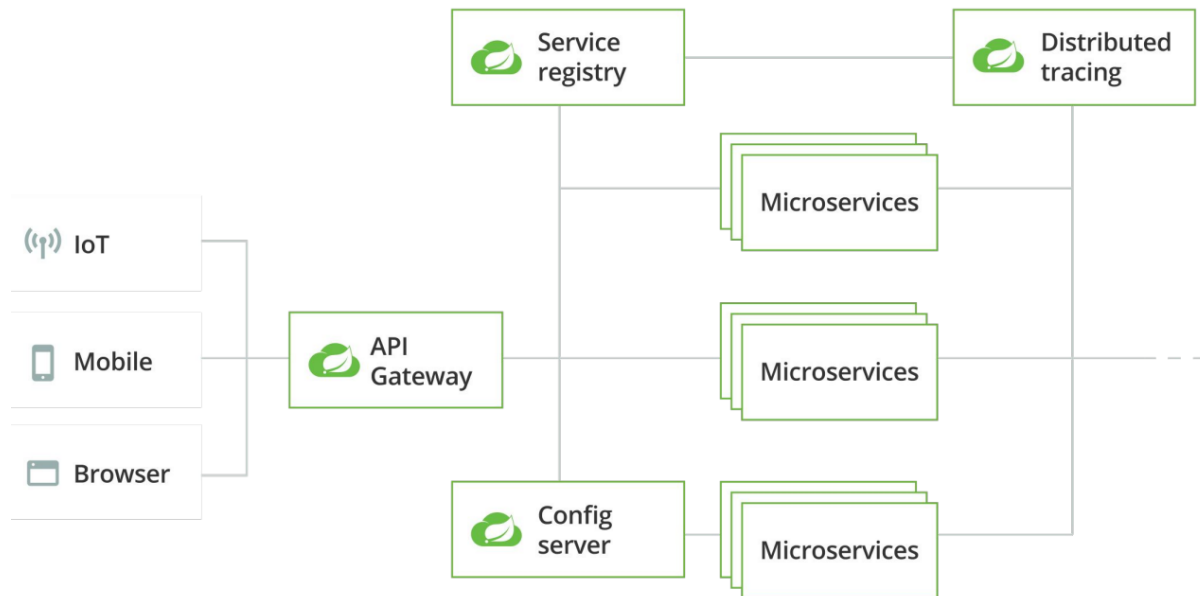
```
}
```

Вариант 5

1. Реализация микросервисов на Spring Cloud.

Spring Cloud

- Инструментарий на базе Spring для построения распределенных систем (а не облако от Spring!)
- Следующий уровень инфраструктурной иерархии (Spring -> Spring Boot -> Spring Cloud).
- Ключевые возможности:
 - Распределённая конфигурация с версионированием.
 - Регистрация и автообнаружение сервисов.
 - Маршрутизация вызовов.
 - Организация взаимного вызова сервисов.



Микросервисы на Spring Cloud

- Решение “из коробки” для реализации MSA.
- Расширяет возможности Spring Boot.
- Включает в себя следующий набор компонентов:
 - Spring Cloud Config Server.
 - Auth Server.
 - API Gateway.
 - Service Discovery.
 - Клиентский балансировщик, Circuit Breaker и HTTP-клиент.
 - Панель мониторинга.

2. Service Discovery и Service Registry.

Service Discovery — см [Вариант 1 вопрос 1](#)

- **Service Discovery** – это процесс, позволяющий клиентам автоматически найти и использовать доступные службы. Примерами Service Discovery могут быть DNS, различные протоколы динамической конфигурации или протоколы обнаружения служб, такие как SLP или UPnP.

Решаемые задачи:

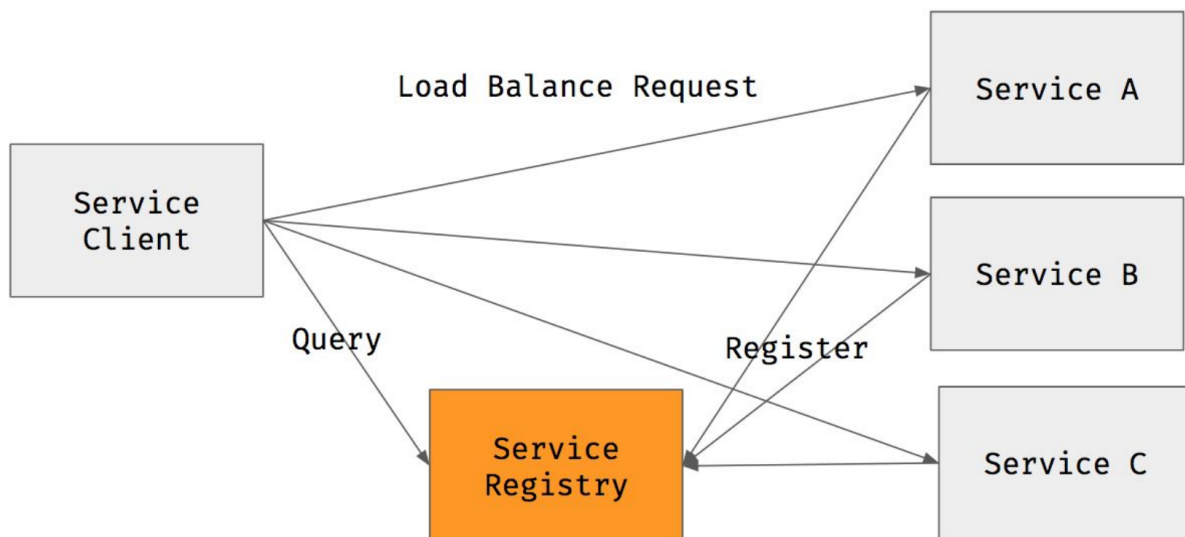
- реконфигурация системы
- упрощение администрирования

- горизонтальная масштабируемость

Service Registry – Ключевой компонент, SD. Это хранилище данных, используемое для хранения информации об услугах, доступных для других систем. Оно может содержать информацию о местоположении служб, метаданные и обнаруженные атрибуты, протоколы доступа и другую полезную информацию. Поэтому его необходимо обновлять и регулярно поддерживать.

Service Registry

- API для взаимодействия с реестром сервисов.
- Есть имплементации для популярных реестров (Eureka, Consul, ZooKeeper, Kubernetes...)
- Базовый интерфейс – DiscoveryClient.



3. Веб-сервис на JAX-WS, управляющий процессом выпечки колобков. Выпечка пирожка состоит из последовательности взаимозависимых этапов, выполнение которых должен обеспечивать сервис - поскребание по сусекам, подготовка теста, лепка заготовки, термическая обработка.

```

@WebService(endpointInterface = "shit.soa.BakeryService")
public class BakeryServiceImpl extends BakeryService {

    @Inject

```



```

private DoBakeryService doService;

@WebMehtod(operationName = "poskrebanie")
@WebResult(name="poskrebanie_res_dto")
public PoskrebanieResDto poskrebanie(@WebParam(name="poskrebanie_req_dto")
PoskrebanieReqDto req) {
    return new doService.doPoskrebanie(req); //delaet poskrebanie
}

@WebMehtod(operationName = "test_preparing")
@WebResult(name="test_preparing_res_dto")
public TestPreparingResDto test_preparing(@WebParam(name="poskrebanie_req_dto")
TestPreparingReqDto req) {
    return new doService.doTestPreparing(req); //delaet testa preparing
}

@WebMehtod(operationName = "lepka")
@WebResult(name="lepka_res_dto")
public LepkagResDto lepka(@WebParam(name="lepka_req_dto") LepkaReqDto req) {
    return new doService.doLepka(req); //delaet lepka ZagotoVki
}

@WebMehtod(operationName = "term_handling")
@WebResult(name="term_handling_res_dto")
public TermHandlingResDto termHandling(@WebParam(name="term_handling_req_dto")
TermHandlingReqDto req) {
    return new doService.doTermHandling(req); //delaet term handling
}
}

```

Вариант 6

1. Реализация Service Discovery на уровне инфраструктурного ПО

Это один из способов реализации SD в MSA и SOA. Service Discovery на уровне инфраструктурного ПО по сравнению со специальным ПО (по типу Consul) поддерживает не только Client-side SD, но и Server-Side*.

Реализация SD происходит на уровне инфраструктурного ПО происходит с помощью конфигурирования и управления контейнерами и виртуальными машинами.

Контейнеризация.

- Docker: ПО для автоматизации развёртывания и управления приложениями в средах с поддержкой контейнеризации. В отличие от развёртывания виртуалок, достаточно только запустить одно Ядро и докер движок, и все контейнеры ставятся на них.
- Kubernetes. ПО для автоматизации развёртывания, масштабирования и управления контейнеризованными приложениями. Он умеет делать Service Discovery.
 - Под — один или несколько контейнеров, гарантированно запущенные на одном узле (VM с контейнерами приложений), с разделением ресурсов, межпроцессным взаимодействием и уникальным IP-адресом.
 - Service в kubernetes: совокупность логически связанных наборов подов и политик доступа к ним.

При такой реализации возможностей гораздо больше, чем в прозрачном Java EE и специальном ПО том же.

* Что же такое Client side и server-side?

- client-side discovery - клиент запрашивает информацию о доступе к доступным экземплярам у SD, а дальше сам распределяет нагрузку между ними
- server-side discovery - клиент использует посредника (SD) для запроса каталога услуг и выполнения запросов к ним

2. Реализация микросервисов на Java EE

- Все сервисы “by design” регистрируются в JNDI и доступны через CDI.
- Если логику “под капотом” реализуют EJB, их можно масштабировать.
- Конфигурация сервера приложений может быть распределённой:
 - В виде пула экземпляров сервера.

- В виде кластера

Компонент в пуле представляет состояние пула в жизненном цикле EJB. Это означает, что у bean-компонента нет идентификатора. Преимущество наличия bean-компонентов в пуле заключается в том, что время на создание bean-компонента можно сэкономить для запроса. Контейнер имеет механизмы, которые создают объекты пула в фоновом режиме, чтобы сэкономить время создания bean-компонента на пути запроса.

Компоненты сеанса без сохранения состояния и компоненты управления данными используют пул EJB. Помня о том, как вы используете сеансовые компоненты без сохранения состояния и объем трафика, который обрабатывает ваш сервер, настройте размер пула, чтобы предотвратить чрезмерное создание и удаление компонентов.

-
- Подход используется относительно редко
 - Стандартный вариант -- EJB + JAX-RS.
 - “Из коробки” есть Service Discovery с помощью JNDI.
 - Можно выбирать между масштабированием на уровне экземпляров бинов и серверов приложений.
 - Нужно выбирать максимально “лёгкие” серверы приложений (Jetty, Payara Micro etc).

3. Конфигурация в Consul сервиса по выпеканию пирожков с пулом экземпляров сервиса. Сервис реализован на Jakarta EE и развернут в контейнере приложений Wildfly на серверах helios, terra и aqua

```
{
  "services": [{
    "name": "wildfly-bakery-1",
    "address": "helios.se.ifmo.ru",
    "port": "41050",
    "checks": [{
      "id": "bakery-1-check",
      "name": "Bakery 1 Helios check",
      "http": "http://helios.se.ifmo.ru:41050/health",
      "method": "GET",
      "interval": "13s"
    }]
  }, {
    "name": "wildfly-bakery-2",
    "address": "aqua",
```

```
    "port": "41050",
    "checks": [{
      "id": "bakery-2-check",
      "name": "Bakery 1 Aqua check",
      "http": "http://aqua:41050/health",
      "method": "GET",
      "interval": "13s"
    }]
  }, {
    "name": "wildfly-bakery-3",
    "address": "terra",
    "port": "41050",
    "checks": [{
      "id": "bakery-2-check",
      "name": "Bakery 1 Terra check",
      "http": "http://terra:41050/health",
      "method": "GET",
      "interval": "13s"
    }]
  }
}
```

Вариант 7

1. Consul. Назначение, архитектура, возможности

Consul - server-side Система обнаружения сервисов (Service Discovery) с распределенным хранилищем “ключ-значение”. Так как является распределенным, то (ПО ставится на каждый комп). Регистрируются путём создания файлов в формате json

Предлагает веб-интерфейс для работы со списками сервисов.

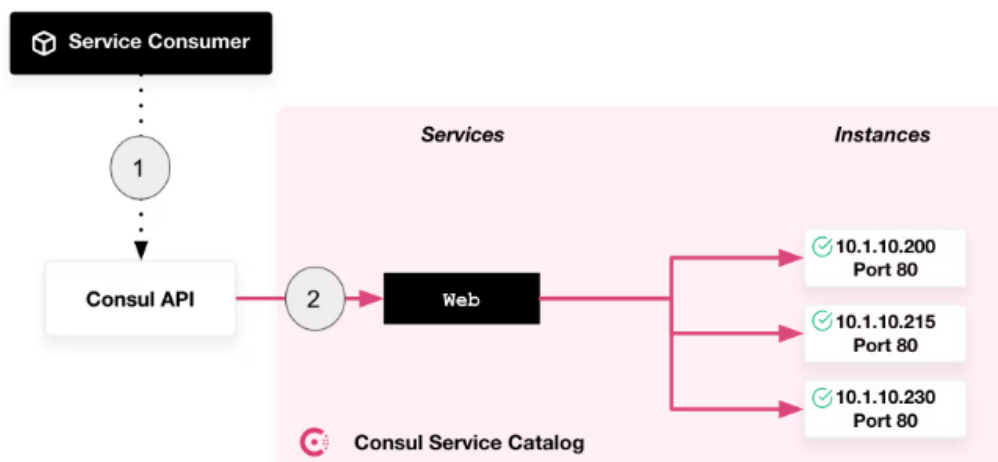
Преимущества Consul для микросервисов

1. Обнаружение сервисов: эта функция покрывается Consul и полезна для интеграции новых технологий в ваши микросервисы.
2. Повышенная прозрачность: Consul полностью прозрачен и может использоваться без каких-либо зависимостей кода.
3. Конфигурация: Consul можно использовать для настройки микросервисов. Могут быть реализованы как обнаружение сервисов, так и их настройка.
4. Балансировка нагрузки: с помощью Consul DNS Consul прозрачно реализует балансировку нагрузки с помощью DNS-сервера.

Регистрация сервисов в консуле производится с помощью функции Consul Connect. Также регистрируются политики взаимодействия, например, мы можем указать, что сервис 1 может взаимодействовать с сервисом 2, но не может взаимодействовать с сервисом 3. Также для регистрации своего приложения доступна HTTP API или конфигурационные файлы самого консула (при наличии поддержки со стороны приложения).

Принцип работы server side SD:

клиент использует посредника (SD - Consul'a) для запроса каталога услуг и выполнения запросов к ним.



2. Микросервисы на spring boot

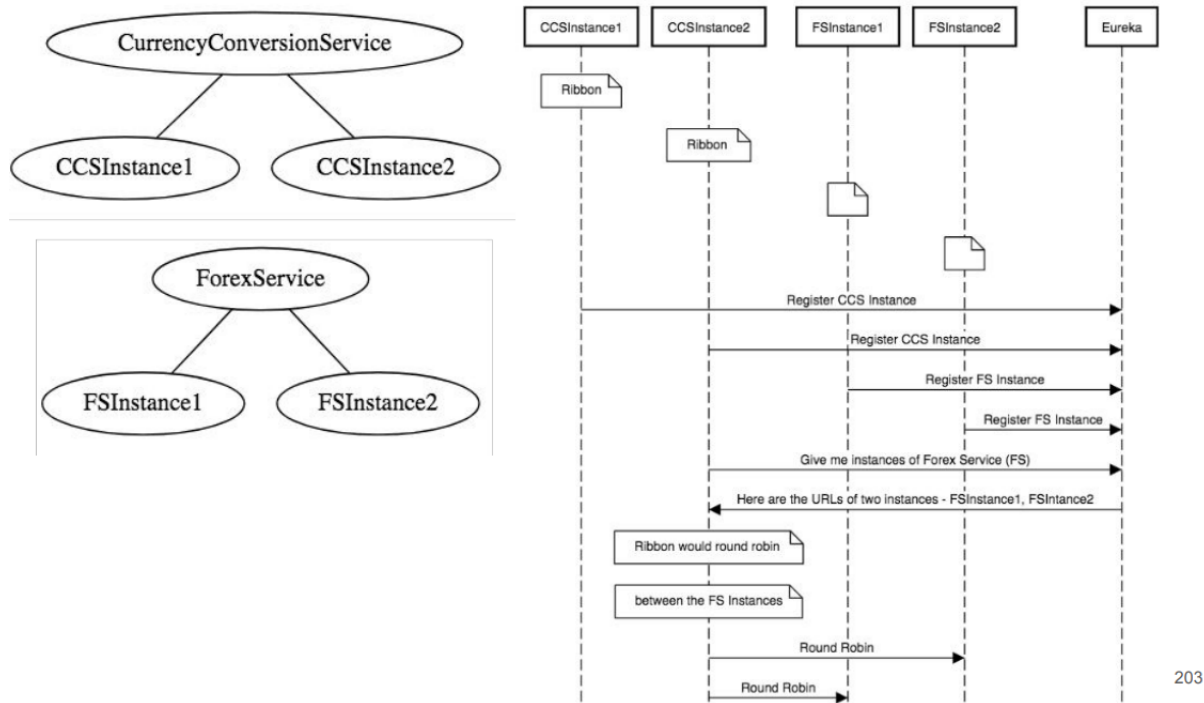
Микросервисы на Spring Boot

- Де-факто – обычные веб-сервисы (Spring Web MVC REST/ Spring Data REST).
- Технология интеграции – на выбор программиста.
- Для того, чтобы управлять пулами микросервисов, нужно интегрироваться с сервером имён (например, Eureka).
- Для того, чтобы распределять нагрузку, нужно интегрироваться с балансировщиком (например, Ribbon).
- Интеграция осуществляется “вручную”.





Балансировка нагрузки и использование индекса



3. Написать на JAX-WS сервис по управлению процессом изготовления резиновых уток

```
@WebService(endpointInterface = "shit.soa.DuckService")
public class DuckServiceImpl extends DuckService {

    @Inject
    private DoDuckService makingService;

    @WebMethod(operationName = "makeNewDuck")
    public MakeNewDuckResDto makeNewDuck(MakeNewDuckReqDto req) {
        return new makingService.makeNewDuck(req.getDuckParams());
        //delat' utka po params
    }

    @WebMethod(operationName = "remakeOldDuck")
    public TestPreparingResDto remakeOldDuck(TestPreparingReqDto req) {
```

```

        return new makingService remakeOldDuck(req.getOldDuck(). req.getDuckParams());
        //peredelat' old utka with new params
    }

    @WebMehtod(operationName = "destroyBadDuck")
    public DestroyBadDuckResDto destroyBadDuck(DestroyBadDuckReqDto req) {
        return new makingService.destroyBadDuck(req.getDeffectiveDuck());
        //ubit' utka with wrong params
    }
    ... //fantasize!
}

```

Вариант 8

1. Service Discovery

Описание подхода

- Обнаружение сервисов (Service Discovery) — автоматическое определение устройств и сервисов, предоставляемых этими устройствами в компьютерной сети.
- Протоколы обнаружения сервисов (Service Discovery Protocol) — сетевые протоколы, реализующие SD.
- Таких протоколов много: DNS SD , Dynamic Host Configuration Protocol (DHCP), Service Location Protocol (SLP), Universal Description Discovery and Integration (UDDI) для веб-сервисов, Web Proxy Autodiscovery Protocol (WPAD)...

SD создан для того, чтобы с минимальными затратами можно подключить новое приложение в уже существующее наше окружение. Используя Service Discovery, мы можем максимально разделить либо контейнер в виде докера, либо виртуальный сервис от того окружения, в котором он запущен.

Решаемые задачи

- Реконфигурация системы.
- Упрощение администрирования.
- Горизонтальная масштабируемость.
- Общая концепция универсальна (искать можно не только веб-сервисы!)
- Варианты реализации (не являются прямыми альтернативами!):
 - Инфраструктурное ПО (Kubernetes, nginx...).
 - Специальное прикладное ПО (Consul, ZooKeeper, Etcd...).
 - “Прозрачно” силами платформы (Java EE).
 - Фреймворк (Spring Cloud [Netflix]).

Два типа SD и варианты реализации см. [Вариант 1. Вопрос 1](#)

2. Smart Endpoints & Dumb pipes

Smart endpoints & Dumb pipes (SE & DP) - паттерн MSA для взаимодействия микросервисов по сети.

Начнем с контрпримера: в ESB реализован подход «умный канал и глупые сервисы»: умный канал — шина ESB, глупые сервисы — конечные системы. В противовес ESB, существует подход «**умные сервисы и глупые каналы**», используемый в MCA. Суть в том, что микросервисы (смарт эндпоинты) должны выполнять свою собственную коммуникационную логику, а надстройка (pipes) по передаче сообщений (“from endpoint to endpoint”) должна иметь как можно более легковесную структуру.

В виде каналов в данном подходе выступают различные брокеры:

- **Брокеры сообщений** (message broker) - выступает посредником: преобразует сообщение по одному протоколу от приложения-источника в сообщение протокола приложения-приёмника. Так же брокер, например, проверяет сообщение на ошибки, маршрутизирует, и выполняет еще ряд функций (но тем не менее не бизнес-логику, оставаясь “dumb”)
- **Событийные брокеры** (event broker) - используется для передачи событий по схеме publish-subscribe.

А в виде сервисов — шлюзы (API Gateway), которые скрывают за собой реализацию и выносят наружу для взаимодействия API.

Плюсы:

- Возможность построения слабосвязанных систем
- Гибкость настройки, взаимозаменяемость инструментов взаимодействия
- Быстрота взаимодействия

Минусы:

- Сложность проектирования и реализации системы
- Требуется практика DevOps по настройке окружения
- В связи с общим трендом на микросервисы в корпоративной разработке, стоимость труда специалистов увеличивается

3. Jax-ws сервис по управлению контингентом в университете

```
@WebService(endpointInterface = "shit.soa.StudentManagmentService")
public class StudentManagmentServiceImpl extends StudentManagmentService {

    @Inject
    private StudentManagmentRepository rep;

    @WebMethod(operationName = "getAllStudents")
```

```

    @WebResult(name="student_list_dto")
    public StudentsListDTO getAllStudents(@WebParam(name="get_students_req_dto")
GetAllStudentsReqDTO req) {
        return new StudentsListDTO().getStudents().addAll(rep.getStudents());
    }

    @WebMehtod(operationName = "getStudent")
    @WebResult(name="student")
    public Student getStudent(@WebParam(name="get_student_req_dto") GetStudentReqDTO req) {
        return rep.getStudent(req.getId)
    }

    @WebMehtod(operationName = "addStudent")
    @WebResult(name="student")
    public Student addStudent(@WebParam(name="add_student_req_dto") AddStudentReqDTO req) {
        return rep.addStudent(req.getName, req.getGroup, ...);
    }

    @WebMehtod(operationName = "updateStudent")
    @WebResult(name="student")
    public Student updateStudent(@WebParam(name="update_student_req_dto") UpdateStudentReqDTO
req) {
        return rep.updateStudent(req.getId, req.getName, req.getGroup, ...);
    }

    ...
}

```

Вариант 9

1. Основные проблема коммуникации микросервисов и способы их решения

Проблемы:

- **Проблема доступности.** Мы не знаем, какой из микросервисов сейчас доступен для общения, а какой упал или потерял коннект.
- **Задержка передачи, потери, дублирование пакетов.** Мы думаем, что отправили пакет, но получатель его не принял или получил в двойном экземпляре, и нам об этом неизвестно.

- **Нагрузка на трафик и память.** Сервисы бывают нагруженные, поэтому общение приходится оптимизировать. От этого все становится сложнее. Если использовать какие-то асинхронные системы общения, придется хранить информацию какое-то время, а значит появляется вопрос к утилизации памяти или диска.
- **Отказоустойчивость.** Часто бывает, что сервисы падают каскадом - один упал, запросы не обрабатывает. Вслед за ним валятся другие сервисы, которые его вызывали - все от того, что они не могут получить ответ на свой запрос.
- Решения:

Асинхронные способы общения - мы отправляем сообщение, а ответ придет когда-нибудь потом или он в принципе не предусмотрен:

Месседжинг - RabbitMQ, ZeroMQ, ActiveMQ, Kafka. По сути берут на себя ответственность за доставку сообщения. Также Мессенджинги могут быть с балансировкой нагрузки.

Синхронные способы общения - мы делаем вызов и ждем получения ответа:


REST API - неплохое решение, когда нет высоких нагрузок и получается хорошо контролировать доступность микросервисов. Взаимодействия по REST API проще отслеживать, чем асинхронный обмен, - сразу видно, что ответ не приходит или приходит не в том формате.

Минусы: Если какой-то простой batch должен передать сообщение и завершиться, но получатель лежит, будут проблемы

2. Circuit Breaker. Принципы, реализация, примеры использования.

- ПО, предотвращающее “заведомо обречённые” запросы к сервисам.
- “Каноничный” вариант -- Hystrix из стека Netflix.
- Конфигурируется аннотациями (пример для Hystrix):
 - `@EnableCircuitBreaker` - разрешить имплементацию CircuitBreaker
 - `@EnableHystrix` enables Hystrix capabilities in your Spring Boot application
 - `@EnableHystrixDashboard` - will give a dashboard view of Hystrix stream.

- В отличие от Retry (когда если маленький шанс, что ошибка возникнет снова - мы не выбрасываем ошибку) паттерна, паттерн Circuit Breaker рассчитан на менее ожидаемые ошибки, которые могут длиться намного дольше: обрыв сети, отказ сервиса, оборудования. В этих ситуациях при повторной попытке отправить аналогичный запрос с большой долей вероятности мы получим аналогичную ошибку.

 Circuit Breaker: пример использования



3. JAX-WS сервис по управлению авиаперелётами

```
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;

@WebService(name = "FlightService")
public class FlightService {
    @WebMethod
    public List<Flight> getFlights() {
        // Реализация метода
    }

    @WebMethod
    public void addFlight(@WebParam(name = "flight") Flight flight) {
        // Реализация метода
    }

    @WebMethod
    public void deleteFlight(@WebParam(name = "flight") Flight
flight) {
        // Реализация метода
    }

    @WebMethod
    public List<Airport> getAirports() {
        // Реализация метода
    }

    @WebMethod
    public void addAirport(@WebParam(name = "airport") Airport
airport) {
        // Реализация метода
    }

    @WebMethod
    public void deleteAirport(@WebParam(name = "airport") Airport
airport) {
        // Реализация метода
    }
}
```

Вариант 10

1. Load Balancer в Spring Cloud.

Библиотека Spring Cloud Load Balancer позволяет нам создавать приложения, которые взаимодействуют с другими приложениями с балансировкой нагрузки. Используя любой алгоритм, который мы хотим, мы можем легко реализовать балансировку нагрузки при выполнении удаленных вызовов службы.

Load Balancer в Spring Cloud предоставляет возможность распределения нагрузки между несколькими экземплярами служб или приложений. Он поддерживает различные алгоритмы балансировки, включая балансировку по нагрузке, балансировку по расстоянию и балансировку рандомом. Он может использоваться для маршрутизации трафика в микросервисных архитектурах. Это помогает избежать перегрузки одного экземпляра, а также снизить время отклика. Он также предлагает маршрутизацию по меткам (применяемым к отдельным экземплярам служб) и равномерное распределение нагрузки между ними.

Для использования Load Balancer в Spring Cloud необходимо иметь следующие компоненты:

1. Клиентское приложение, использующее балансировщик нагрузки для отправки запросов на сервер.
2. Сервер, который принимает запросы и отправляет ответы клиентскому приложению.
3. Фронт-контроллер, который отвечает за обработку запросов и управляет потоком трафика между клиентом и сервером.
4. Алгоритм балансировки, который определяет, какие серверы будут использоваться для обработки запросов и каким образом будет происходить маршрутизация трафика.

Spring Cloud предоставляет Load Balancer:

1. Ribbon – это клиентский Load Balancer, который помогает приложениям маршрутизировать запросы к доступным службам.

2. Mule ESB, основные свойства и понятия.

Mule ESB - реализация сервисной шины. Сервисная шина предприятия (Enterprise Service Bus, ESB) — ПО, обеспечивающее обмен сообщениями между различными ИС на принципах COA.



- › Синхронный и асинхронный вызов сервисов.
- › Использование защищённого транспорта, с гарантированной доставкой сообщений, поддерживающего транзакционную модель.
- › Маршрутизация сообщений.
- › Доступ к данным из сторонних ИС с помощью адаптеров.
- › Обработка и преобразование сообщений.
- › Оркестровка и хореография сервисов.
- › Разнообразные механизмы контроля и управления (аудиты, протоколирование).

Mule ESB - платформа, одна из реализаций ESB, дающая возможность объединять различные информационные системы на основе принципов обмена сообщениями, контроля надежности, сопоставления данных, оркестровки, а также масштабированием между узлами.



Mule ESB

- Платформа ESB и фреймворк для интеграции сервисов от MuleSoft.
- Написана на Java, может выступать в качестве брокера и для сервисов на других платформах.
- Есть открытая и коммерческая версии.
- Есть адаптеры для разных видов сервисов.
- Есть Anypoint Studio -- IDE на базе Eclipse для разработки проектов на базе Mule.





Mule: основные понятия

- Каждое *сообщение (message)* делится на две части -- *заголовок (header)* и *полезную нагрузку (payload)*.
- Обмен сообщениями реализуется через *потоки (flows)*. Каждое приложение содержит один или несколько потоков.
- Есть два способа конфигурации потоков:
 - Через дескрипторы XML.
 - Графический -- с помощью Anypoint Studio.

3. Конфигурация на Consul пула серверов WildFly, на которых развёрнута Jakarta EE система по выпеканию пирожков.

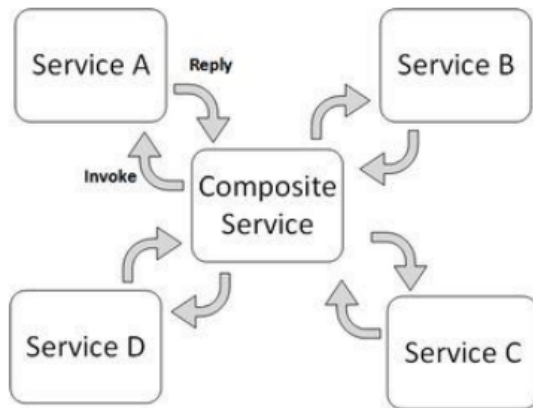
```
{
  "services": [{
    "name": "wildfly-bakery-1",
    "address": "14.88.228.42",
    "port": "14141",
    "checks": [{
      "id": "bakery-1-check",
      "name": "Bakery 1 HTTP check",
      "http": "http://14.88.228.42/14141",
      "method": "GET",
      "interval": "14s"
    }]
  }, {
    "name": "wildfly-bakery-2",
    "address": "14.88.228.24",
    "port": "14141",
    "checks": [{
      "id": "bakery-2-check",
      "name": "Bakery 2 script check",
      "script": "/petuxon.py --host 14.88.228.24 -- port 14141",
      "interval": "88s"
    }]
  }]
}
```

Вариант 11

1. Оркестровка и хореография микросервисов

Оба термина описывают два аспекта разработки бизнес-процессов на основе объединения Web-сервисов.

Оркестровка микросервисов – добавление подсистемы (дирижер - orchestrator), которая будет координировать взаимодействие между сервисами:



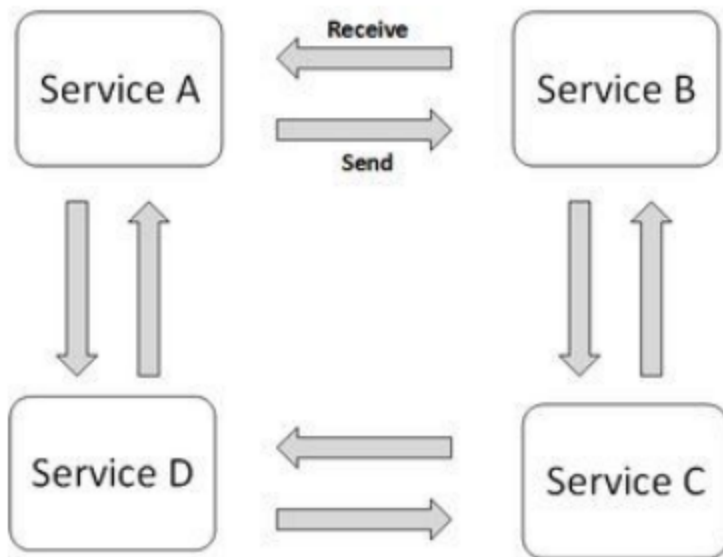
При оркестровке мы настраиваем и интегрируем существующие микросервисы для реализации бизнес-целей (в т.ч. планирование распределенной системы, интеграцию существующих микросервисов, обновление микросервисов и мониторинг производительности).

Есть готовые решения для оркестровки микросервисов:

- Kubernetes;
- Azure Kubernetes Service (ACS);
- Apache Mesos;
- Amazon Elastic Container Service (ECS).



Хореография микросервисов – это процесс проектирования связи между микросервисами и их взаимодействий. Она может включать в себя проектирование архитектуры, создание мостов между микросервисами, планирование взаимодействия между микросервисами и мониторинг и анализ межсервисной связи. В отличие от оркестровки “центрального” сервиса нет: Административная логика “размазана” по сервисам:



Хореография более честно ложится на Микросервисную архитектуру, но сложнее в реализации. Для имплементации могут использоваться брокеры событий

2. Top-down и bottom-up стратегии при использовании аннотаций wsdl

Top-down стратегия включает в себя создание WSDL-документа, используя в качестве основы схему XML. Процесс предполагает использование аннотаций, таких как `@WebService` и `@WebMethod`, для определения интерфейса web-службы.

Bottom-up стратегия предполагает начало с существующего WSDL-документа. Для реализации данной стратегии используется `@WebServiceProvider` аннотация для создания интерфейса web-службы.

Top-down стратегия предполагает сначала создание WSDL-документа и затем использование его для генерации кода. Это полезно, когда вам нужно предоставить доступ к вашим службам другим людям, которые не знакомы с вашим кодом.

Bottom-up стратегия предполагает сначала создание кода и затем генерацию WSDL-документа на основе этого кода. Это полезно, когда вы создаете службы для использования ими других людей, но вы остаетесь ответственным за их реализацию.

3. Конфигурация Spring Boot сервиса для подключения Consul, он живет на localhost на 8500 порту

```
spring.application.name=main-service
spring.cloud.consul.host=localhost
spring.cloud.consul.port=8500
```

```
spring.cloud.consul.config.enabled=true
#health check
spring.cloud.consul.config.import-check.enabled=false
spring.cloud.consul.discovery.health-check-path=/health
spring.cloud.consul.discovery.health-check-interval=8s
#if tls cert self-signed
spring.cloud.consul.discovery.health-check-tls-skip-verify=true
#if https needed
spring.cloud.consul.discovery.scheme=https
```

Вариант 12

1. Свойства и ключевые особенности микросервиса.

Это архитектурный стиль разработки, который позволяет создавать приложения в виде набора небольших автономных **микросервисов**, разработанных для бизнес-сферы.



Особенности MSA

- Модули легко заменить в любое время: акцент на простоту, независимость развёртывания и обновления.
- Микросервис [по возможности] выполняет только одну элементарную функцию.
- Модули могут быть реализованы с использованием разных стеков технологий и работать на разных платформах.
- Архитектура симметричная, а не иерархическая: зависимости между микросервисами одноранговые.

Свойства микросервиса (1)

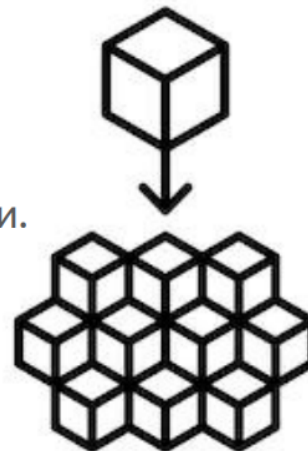
1. Небольшой.
2. Независимый.
3. Строится вокруг бизнес-потребности и использует ограниченный контекст (Bounded Context).
4. Взаимодействует с другими микросервисами по сети на основе паттерна Smart endpoints & dumb pipes.



172

Свойства микросервиса (2)

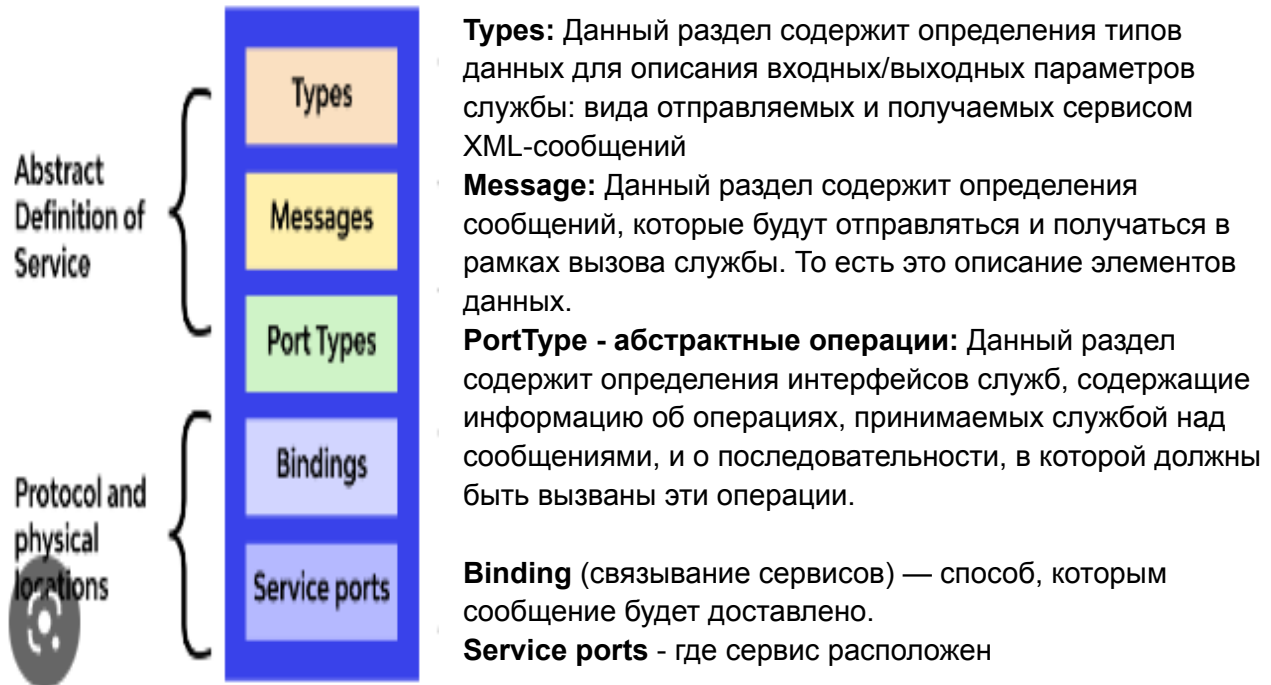
5. Его распределенная суть обязывает использовать подход Design for failure.
6. Централизация ограничена сверху на минимуме.
7. Процессы его разработки и поддержки требуют автоматизации.
8. Его развитие итерационное.



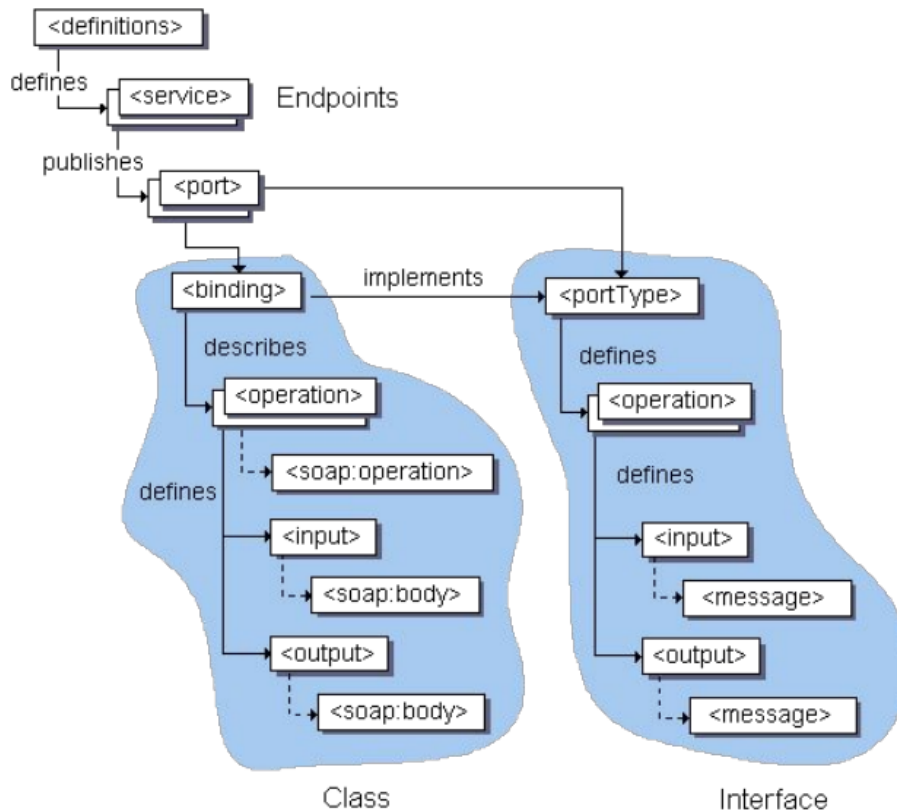
2. Структура документа WSDL

WSDL - Язык спецификации SOAP веб-сервисов, который Базируется на XML.

Структура версии 1.1:



Структура документа WSDL 1.1, пример:



Отличия WSDL 2.0 от 1.1:

- Типы портов переименованы в интерфейсы. Поддержка наследования интерфейса достигается за счет использования атрибута `extends` в элементе интерфейса.
- Порты переименованы в конечные точки

3. Конфигурация, регистрирующая в Consul пул экземпляров СУБД MySQL. Сервисы расположены на машинах `ra`, `helios` и `anubis`, экземпляры занимают порт 3306 на каждой из машин. Конфигурация должна обеспечивать проверку того, запущены ли экземпляры каждого из сервисов.

```

{
  "services": [{
    "name": "mysql-ra",
    "address": "ra-address",
    "port": "3306",
    "checks": [{
      "name": "RA MySQL check",
      "script": "/mysql-check-script.py --host ra-address -- port 3306",
      "interval": "88s"
    }]
  }],
  {
    "name": "mysql-helios",

```



```

    "address": "helios.se.ifmo.ru",
    "port": "3306",
    "checks": [{
        "name": "Helios MySQL check",
        "script": "/mysql-check-script.py --host helios.se.ifmo.ru -- port 3306",
        "interval": "88s"
    }]
}, {
    "name": "mysql-anubis",
    "address": "anubis-address",
    "port": "3306",
    "checks": [{
        "name": "Anubis MySQL check",
        "script": "/mysql-check-script.py --host anubis-address -- port 3306",
        "interval": "88s"
    }]
}]
}

```

Вариант 13

1. Преимущества и недостатки MSA по сравнению с “обычной” SOA.

Преимущества MSA по сравнению с SOA:

- Уменьшение размера кода: MSA может быть более легкой и модульной архитектурой, поскольку она может быть ориентирована на компоненты и более маленькие микросервисы.
- Легче реализовывать масштабирование: MSA может быть легко масштабируемой архитектурой, поскольку она может быть разделена на множество маленьких и модульных компонентов.
- Ускорение разработки: Для создания и поддержания приложений в рамках MSA можно использовать более легкие инструменты и фреймворки.

Недостатки MSA по сравнению с SOA:

- Потенциальные проблемы с согласованностью данных: В некоторых случаях возникает проблема установления согласованности данных по всем микросервисам.
- Отсутствие поддержки для платформ: В некоторых случаях микросервисы могут быть сложно портировать на новые платформы.

- Потенциальные проблемы с безопасностью: В некоторых случаях возникают проблемы с безопасностью, поскольку для некоторых приложений требуется более высокий уровень безопасности, чем тот, который предоставляется в рамках МСА.

Преимущества микросервисов

- Более простой шаблон архитектуры, который легко понять разработчикам
- IDE быстрее делает разработчиков быстрее и продуктивнее
- Веб-контейнер запускается быстрее; это помогает ускорить процесс развертывания и разработки.
- Это позволяет команде разрабатывать, развертывать и масштабировать свой сервис независимо от всех других команд.

Недостатки микросервисов

- Он разработан для построения монолитных приложений, поэтому он не предоставляет явной поддержки для разработки распределенных приложений.
- Тестирование сложнее
- Разработчики должны реализовать механизм межсервисных коммуникаций.
- Реализация вариантов использования, охватывающих несколько служб, требует координации между командами.
- Микросервис стоит дорого, так как вам всегда нужно поддерживать различное серверное пространство для разных бизнес-задач

2. Аннотации JAX-WS

- `@WebService` — указывает на то, что Java класс (или интерфейс) является веб-сервисом.
- `@WebMethod` — позволяет настроить то, как будет отображаться метод класса на операцию сервиса.
- `@WebParam` — позволяет настроить то, как будет отображаться конкретный параметр операции на WSDL-часть (part) и XML элемент.
- `@WebResult` — позволяет настроить то, как будет отображаться возвращаемое значение операции на WSDL-часть (part) и XML элемент.
- `@Oneway` — указывает на то, что операция является односторонней, то есть не имеет выходных параметров.
- `@SOAPBinding` — позволяет настроить то, как будет отображаться сервис на протокол SOAP. (`@SOAPBinding(style=Style.RPC)`)

3. Конфигурация, регистрирующая в Consul пул сервисов libreoffice ... обработкой документов. Сервисы расположены на машинах aqua, ..., по пути /usr/bin/soffice. Конфигурация должна обеспечивать проверку запущены ли экземпляры сервисов.

```
{
  "service": {
    "name": "libreoffice-service",
    "port": 8080,
    "check": {
      "script": "curl helios:8080/health",
      "interval": "10s"
    },
    "tags": ["libreoffice", "documents"],
    "enable_tag_override": false
  },
  "checks": [
    {
      "id": "libreoffice-service-1",
      "name": "Libreoffice service on aqua",
      "command": "/usr/bin/soffice",
      "args": ["--listen",
"--accept=socket,host=aqua,port=8080"],
      "interval": "10s"
    },
    {
      "id": "libreoffice-service-2",
      "name": "Libreoffice service on terra",
      "command": "/usr/bin/soffice",
      "args": ["--listen",
"--accept=socket,host=terra,port=8080"],
      "interval": "10s"
    }
  ]
}
```

Вариант 14

1. UDDI

UDDI: Universal Description, Discovery and Integration — централизованное хранилище дескрипторов WSDL со стандартизированным API. Спецификация UDDI задает способ публикации Web-служб и поиска информации о них.

Спецификация UDDI выполняет две функции:

- Она представляет собой протокол на основе SOAP, описывающий способ взаимодействия клиентов с реестрами UDDI.
- Она выполняет роль набора реестров с глобальной репликацией.

Структура UDDI состоит из:

- Белые страницы - адрес, контакты и известные идентификаторы;
- Желтые страницы - категоризация, описание возможностей, разбиение по географии;
- Зеленые страницы - техническая информация о сервисе: интерфейсы, API.

В состав UDDI входит схема XML для сообщений SOAP, в которой задан набор документов с описанием бизнес-записей и служб, общий набор API для поиска и публикации информации в каталогах, а также API для репликации записей каталогов между равноправными узлами UDDI.

UDDI управляет поиском Web-служб в распределенном реестре бизнес-записей и связанных описаний служб в общем формате XML. Перед публикацией бизнес-объекта и Web-службы в общедоступном реестре необходимо зарегистрировать бизнес-объект в реестре UDDI.

2. Design for failure

Design for Failure - Одно из ключевых свойств микросервисной архитектуры. У MSA есть ряд проблем:

- Ненадёжность сети (а практически все вызовы -- удалённые!).
- Сложность реализации событийной архитектуры.

Эти проблемы сложно и дорого устранять, поэтому мы считаем, что наличие не критичных ошибок при работе -- нормальная ситуация. Свойство же design for failure заключается в том, что с самого первого этапа, начиная строить микросервисную архитектуру, вы должны исходить из предположения, что ваши сервисы не работают. Другими словами, ваш сервис должен понимать, что ему могут не ответить никогда, если он ожидает каких-то данных. Таким образом, вы сразу должны исходить из ситуации, что что-то у вас может не работать.

При этом повышаются требования к инфраструктуре: мы должны уметь оперативно выявлять и исправлять проблемы.

Например, для этого компания Netflix разработала Chaos Monkey — инструмент, который ломает сервисы, хаотически их выключает и рвет соединения. Это нужно, чтобы оценить надежность системы.

3. конфиг для консула чтобы зарегать 3 бд

```
{
  "services": [
    {
      "name": "mysql-service"
      "address": "mysql-address"
      "port": "1010"
      "checks": [
        {
          "name": "check is mysql alive"
          "script": "/check-is-alive.py --host mysql-address --port 1010"
          "interval": "20s"
        }
      ]
    },
    {
      "name": "postgresql-service"
      "address": "postgresql-address"
      "port": "1010"
      "checks": [
        {
          "name": "check is postgresql alive"
          "script": "/check-is-alive.py --host postgresql-address --port 1010"
          "interval": "20s"
        }
      ]
    },
    {
      "name": "redis-service"
      "address": "redis-address"
      "port": "1010"
      "checks": [
        {
          "name": "check is redis alive"
          "script": "/check-is-alive.py --host redis-address --port 1010"
          "interval": "20s"
        }
      ]
    }
  ]
}
```

```

    }
  ],
}

```

Вариант 15

1. Service Discovery в решениях на базе Spring Cloud Netflix

Spring Cloud Netflix -- бандл для построения COA систем на базе Spring Cloud Netflix и интеграции его в стек Netflix OSS посредством автоматической настройки и привязки к Spring Environment. Шаблоны внутри приложения настраиваются легко и быстро с помощью нескольких простых аннотаций, тем самым позволяя создать большие распределенные системы с проверенными в бою компонентами Netflix. Spring Cloud Netflix предоставляет client-side SD - Eureka.s

Напомним, что client-side discovery - клиент запрашивает информацию о доступе к доступным экземплярам у SD, а дальше сам распределяет нагрузку между ними. В Client-side SD единственной «фиксированной точкой» в архитектуре является Service Registry, в котором должна регистрироваться каждая служба. Клиент сам имплементирует логику для взаимодействия со службой.

Каждый микросервис регистрируется на сервере Eureka, и Eureka знает все клиентские приложения, работающие на каждом порту и IP-адресе. Eureka Server также известен как Discovery Server.

С Netflix Eureka каждый клиент действует как “сервер” при подключении к сервису: клиент извлекает список всех подключенных экземпляров в service registry и отправляет все дальнейшие запросы к службам с помощью алгоритма балансировки нагрузки.

Обязанность Eureka — давать имена каждому микросервису. Эврика регистрирует микросервисы и отдает их ip другим микросервисам. Таким образом, каждый сервис регистрируется в Eureka и отправляет эхо-запрос серверу Eureka, чтобы сообщить, что он активен. Для этого сервис должен быть помечен как @EnableEurekaClient, а сервер @EnableEurekaServer.

2. Сервисные шины особенности, преимущества и недостатки, использование в решениях на базе SOA

Основным направлением архитектуры ESB является разделение систем друг от друга и обеспечение их устойчивой и управляемой связи.

Состав ESB:

- Брокер сообщений — обеспечивает управление очередностью сообщений и выступает посредником между приложением-источником и приложением-приемником;
- Комплект адаптеров — программных компонентов, которые служат для связи приложений с ESB и преобразуют один интерфейс в другой. Чем больше различных адаптеров заложено в интеграционную шину, тем шире ее функционал;
- В современных ESB-решениях реализованы принципы микросервисной архитектуры. В соответствии с ними весь функционал системы распределяется между микросервисами, каждый из которых может работать независимо от других;
- Средства для контроля и мониторинга.

Преимущества:

- Через интеграционный шлюз происходит быстрый обмен данными с использованием разных форматов и протоколов.
- ESB позволяет преобразовать сообщения в нужный формат, контролировать транзакции, проводить маршрутизацию с учетом смысла, равномерно распределять нагрузку на отдельные сервисы и гарантировать безопасность обмена данными

Недостатки:

- Сложность реализации
- Требуется больших ресурсов.

Mule([см. Вариант 10 вопрос 2](#))

3. Набор микросервисов на Spring MVC, реализующий параллельную сортировку элементов массива. Алгоритм сортировки можно выбрать любой подходящий.

Похоже на [вариант 4. Вопрос 3](#) (потому что тоже именно набор микросервисов, а не 1 микр./осервис)