

Сервис-ориентированная архитектура

лектор -- Цопа Е.А.
2020/21 уч. год



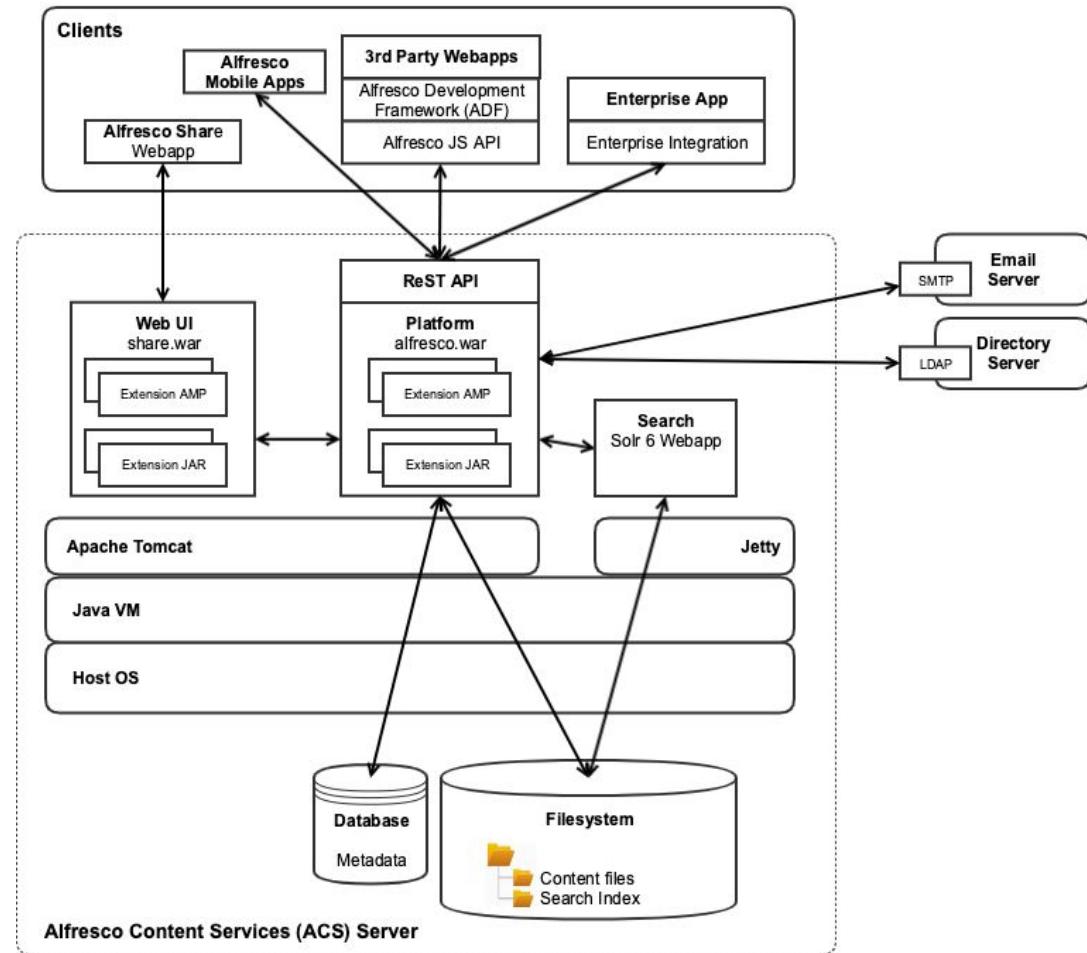
1. Введение



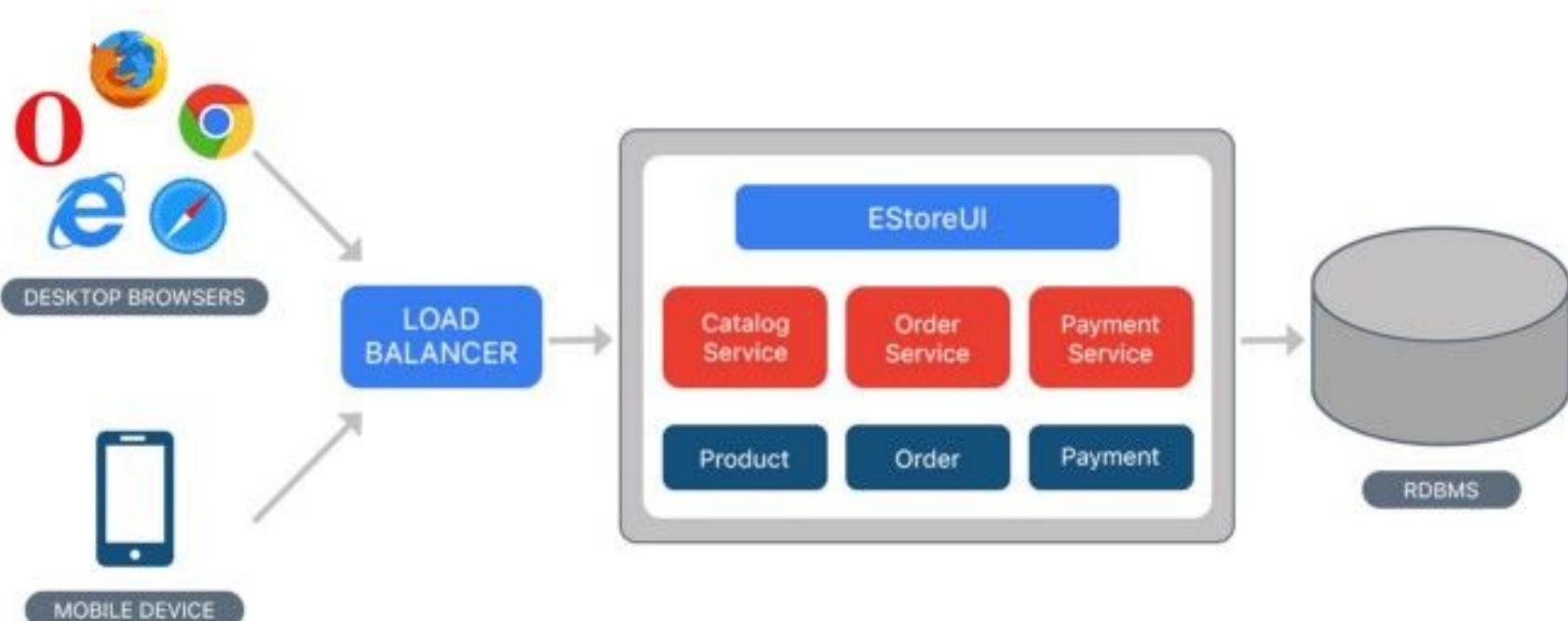
Контекст проблемы

Современные информационные системы:

- Сложные.
- Распределённые и гетерогенные.
- Часто используют “зоопарк” технологий.
- Имеют длительный жизненный цикл.
- Бизнес-требования часто меняются.



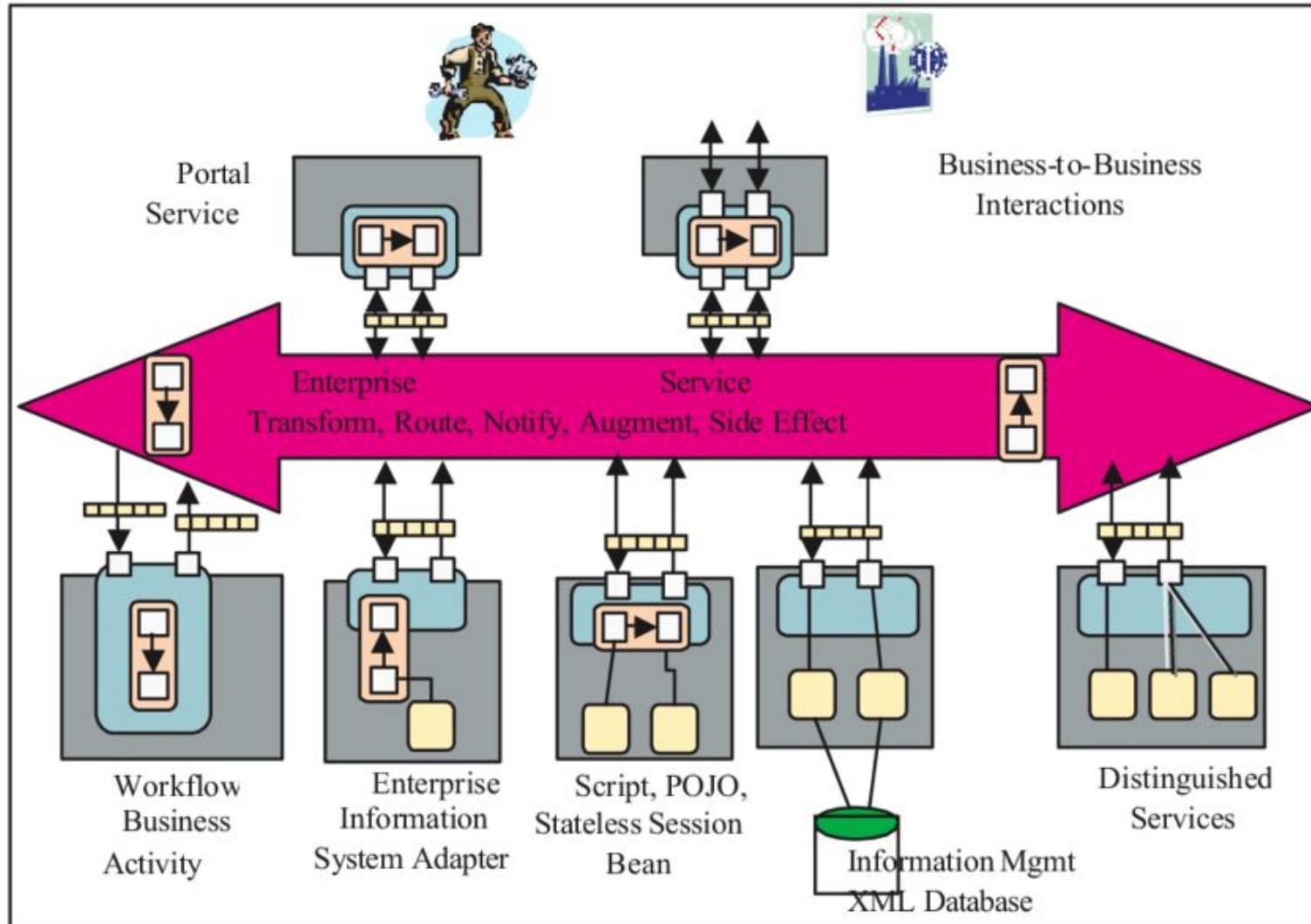
“Монолитная” архитектура





Сервис-ориентированная архитектура (1)

ИТМО ВТ



Сервис-ориентированная архитектура (2)

Приложение делится на модули -- *сервисы*.

Сервисы:

- Изолированы друг от друга.
- Обладают слабой связанностью (*low coupling*).
- Заменяемы.
- Общаются по стандартизованным протоколам.
- Нет никакого индустриального стандарта -- можно использовать любые технологии.

Принципы SOA (1)

- **Standartized Contract** -- интерфейсы взаимодействия должны быть чётко специфицированы.
- **Reference Autonomy** -- взаимосвязи между сервисами должны быть сведены к минимуму.
- **Location Transparency** -- то, где физически располагается сервис, не должно иметь значения при взаимодействии с ним.
- **Longevity** -- сервисы должны разрабатываться с учётом возможности их длительного использования.

Принципы SOA (2)

- **Abstraction** -- внутренняя логика сервиса должна быть скрыта от клиента.
- **Autonomy** -- сервисы должны самостоятельно контролировать собственную функциональность.
- **Statelessness** -- сервис не должен сохранять состояние между обращениями к нему.
- **Granularity** -- сервис должен реализовывать чётко специфицированный и логически обоснованный набор функций.
- **Normalization** -- сервисы должны быть декомпозированы и нормализованы, чтобы минимизировать избыточность.

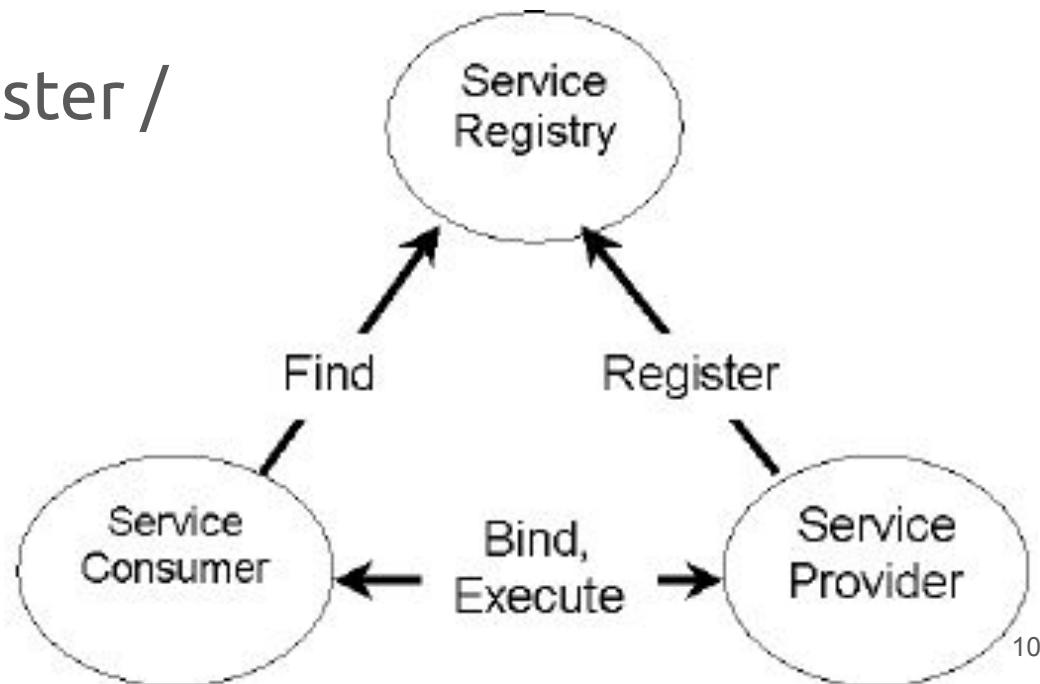
Принципы SOA (3)

- **Composability** -- функциональность сервиса может строиться на базе функциональности других сервисов.
- **Discovery** -- сервисы должны сопровождаться метаданными, позволяющими эффективно идентифицировать и использовать их.
- **Reusability** -- логика приложения разбивается на локальные сервисы, что позволяет повторно использовать код.
- **Encapsulation** -- в сервисы можно “обращивать” функциональность приложений, построенных по принципам, отличным от SOA.

Структура приложения

Любая SOA-система состоит из трёх видов “блоков”:

- Поставщик (service provider).
- Брокер (broker) / реестр (registry) / репозиторий (repository).
- Потребитель (requester / consumer).



Подходы к реализации, стандарты и протоколы

Тысячи их (SOA -- это не только веб-сервисы!):

- Веб-сервисы на базе WSDL и SOAP.
- Системы обмена сообщениями (JMS, ActiveMQ / RabbitMQ).
- RESTful HTTP.
- WCF (by M\$).
- Apache Thrift.
- gRPC (by Google).
- Микросервисы.



Pros & Cons

Достоинства:

- Декомпозиция модулей.
- Можно использовать в разных модулях разные технологии.
- Можно модернизировать модули независимо друг от друга.
- (Теоретически) лучшая масштабируемость.
- Удобная интеграция “из коробки”.

Недостатки:

- Усложнение архитектуры.
- Система теряет целостность.
- Сложнее тестировать.
- Сложнее поддерживать.

2. Веб-сервисы

Что такое веб-сервис

- Сервис, реализуемый одним устройством и используемый другими устройствами, взаимодействующими с первым через интернет.
- Приложение, запущенное на ЭВМ, “слушающее” запросы на определённом порту и возвращающее в ответ на них статический или динамический контент.



Особенности веб-сервисов

- “Жёсткой” спецификации нет.
- Практически любое веб-приложение подходит под определение веб-сервиса.
- В общем случае, веб-сервис более специализирован по сравнению с веб-приложением.

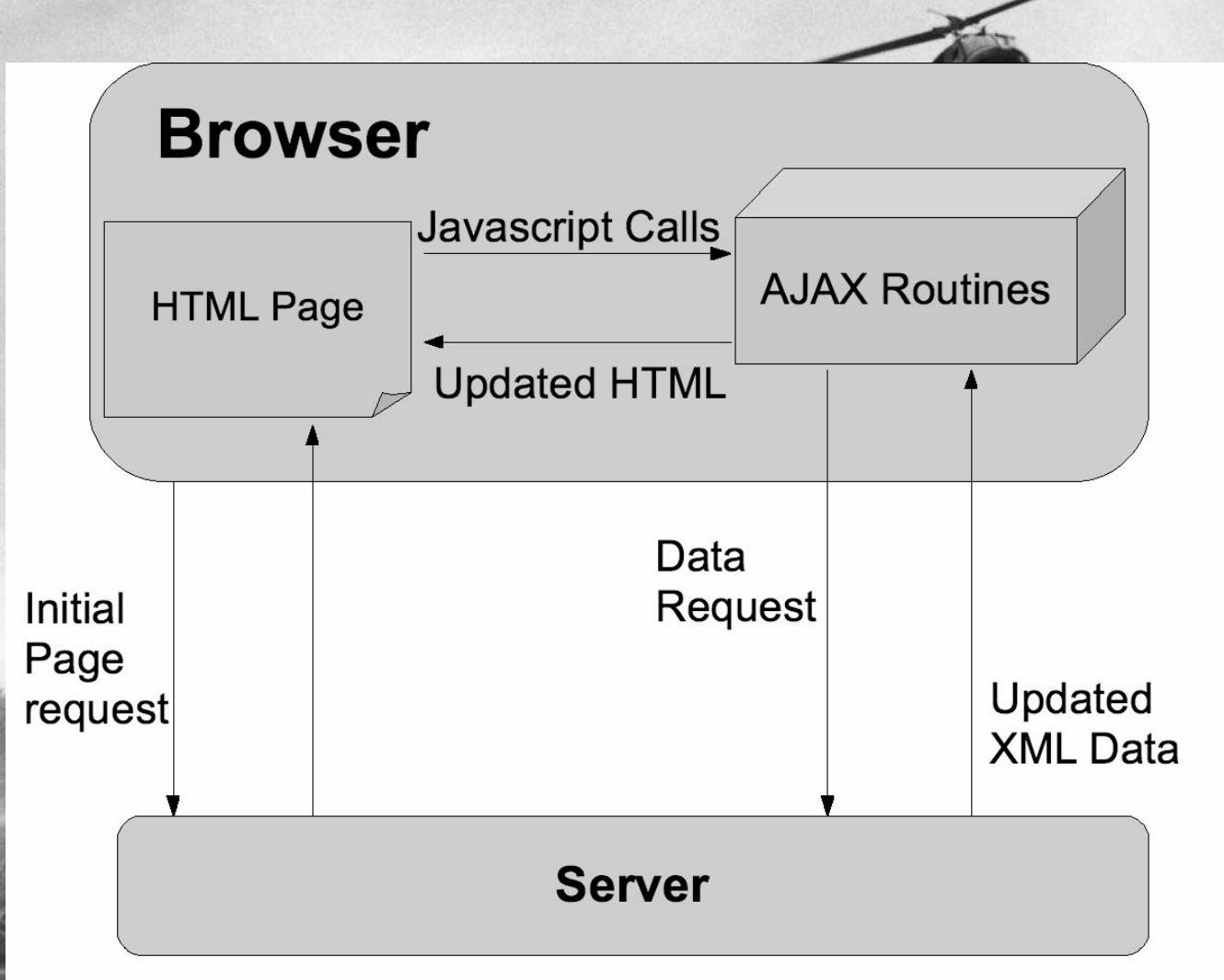


Виды веб-сервисов

- AJAX.
- RESTful.
- Использующие языки спецификации веб-сервисов (WSDL, JSON-RPC etc).
- Стандартизованные W3C (на базе SOAP).



AJAX



RESTful Web Services (1)

Веб-сервисы, построенные на основе
идеологии REST:

- Идеология предложена Роем Филдингом (Roy Fielding) в 2000 г.
- Официального стандарта нет.
- Обычно строится “поверх” протокола http.

{REST*ful* API}

RESTful Web Services (2)

Ключевая идея -- каждый запрос клиента содержит в себе исчерпывающую информацию о желаемом ответе сервера.

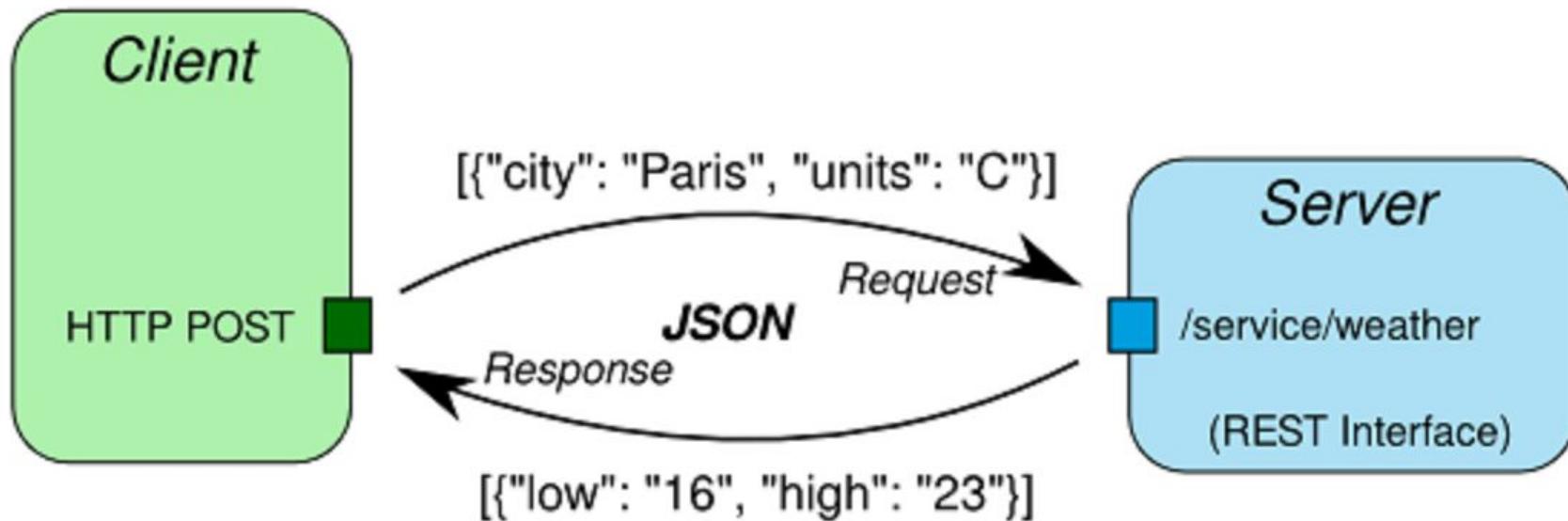
Особенности:

- Сервисы оперируют *ресурсами*.
- На ресурсы указывают URI.
- Тип операции определяется методом http.
- В ответ на запрос сервис может возвращать “полезную нагрузку” (payload).
- Состояние не сохраняется (statelessness).



Пример

RESTful Web Service in Java



Преимущества подхода

- Эффективное взаимодействие компонентов системы.
- Масштабируемость.
- Надёжность.
- Простые и унифицированные интерфейсы.
- Возможность ограничения доступа к отдельным сервисам без остановки всей системы.

Языки спецификации веб-сервисов

- Их много.
- Позволяют декларативно описать, что “умеет” веб-сервис.
- Существуют как для RESTful, так и для SOAP.
- Могут быть использованы для построения реестров веб-сервисов.
- Могут быть использованы для автогенерации кода сервиса и / или клиента.



Пример WSDL

```
><message name="ExecuteCoffeeRequest">...</message>
><message name="ExecuteCoffeeRequestResponse">...</message>
><message name="ExecuteDeliveryRequest">...</message>
▼<message name="ExecuteDeliveryRequestResponse">
  <part name="parameters" element="tns:ExecuteDeliveryRequestResponse"/>
</message>
▼<portType name="TurtlebotPublishersInterface">
  ▶<operation name="ExecuteCoffeeRequest">...</operation>
  ▼<operation name="ExecuteDeliveryRequest">
    <input
      wsam:Action="http://soap.turtlebot.mybot.org/TurtlebotPublishersInterface/ExecuteDeliveryRequest"
      message="tns:ExecuteDeliveryRequest"/>
    <output
      wsam:Action="http://soap.turtlebot.mybot.org/TurtlebotPublishersInterface/ExecuteDeliveryRequest"
      message="tns:ExecuteDeliveryRequestResponse"/>
  </operation>
</portType>
><binding name="TurtlebotPublishersWebServicePortBinding"
  type="tns:TurtlebotPublishersInterface">...</binding>
▼<service name="TurtlebotPublishersWebService">
  ▼<port name="TurtlebotPublishersWebServicePort"
    binding="tns:TurtlebotPublishersWebServicePortBinding">
    <soap:address location="http://192.168.100.11:5555/turtlesim_publisher_ws"/>
  </port>
</service>
```

SOAP

- Позволяет специфицировать интерфейсы веб-сервисов.
- Предложен в 1998 г. для сервисов Microsoft.
- Изначально расшифровывался как Simple Object Access Protocol, сейчас не расшифровывается никак.
- Стандартизирован W3C в 2003 г.

Особенности SOAP

- Основан на XML, является расширением стандарта XML-RPC.
- Обычно работает “поверх” http.
- Обычно используется совместно с дескрипторами веб-сервисов.

SOAP vs RESTful



SOAP (WS-*)

POX

RSS

JSON

...

SMTP

HTTP
POST

MQ...

HTTP
GET

HTTP
POST

HTTP
PUT

HTTP
DEL

Endpoint URI

Resource URI

Application

Application

SOAP

REST



SOAP: пример запроса

```
POST /Quotation HTTP/1.0
Host: www.xyz.org
Content-Type: text/xml; charset = utf-8
Content-Length: nnn

<?xml version = "1.0"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV = "http://www.w3.org/2001/12/soap-envelope"
    SOAP-ENV:encodingStyle = "http://www.w3.org/2001/12/soap-encoding">

    <SOAP-ENV:Body xmlns:m = "http://www.xyz.org/quotations">
        <m:GetQuotation>
            <m:QuotationsName>MiscroSoft</m:QuotationsName>
        </m:GetQuotation>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```



SOAP: пример ответа

HTTP/1.0 200 OK

Content-Type: text/xml; charset = utf-8

Content-Length: nnn

```
<?xml version = "1.0"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV = "http://www.w3.org/2001/12/soap-envelope"
    SOAP-ENV:encodingStyle = "http://www.w3.org/2001/12/soap-encoding">

    <SOAP-ENV:Body xmlns:m = "http://www.xyz.org/quotation">
        <m:GetQuotationResponse>
            <m:Quotation>Here is the quotation</m:Quotation>
        </m:GetQuotationResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Преимущества подхода

- Есть чёткая спецификация.
- Есть готовые инфраструктурные решения.
- Удобен для RPC-систем.

3. RESTful Web Services

Особенности подхода

- RESTful – идеология, а не стандарт.
- Любой обработчик запросов, соответствующий идеологии, можно считать RESTful веб-сервисом.
- Можно использовать универсальные технологии (сервлеты, php-скрипты и т.д.).
- Есть специально предназначенные фреймворки.

Взаимодействие с ресурсом

- На ресурс указывает URI.
- Два вида ресурсов:
 - Осуществляющие манипуляции с данными.
 - Выполняющие какие-либо операции.

Интерпретация методов HTTP

Метод	Ресурс, манипулирующий данными <code>https://api.example.com/collection</code>	Ресурс, выполняющий операции <code>https://api.example.com/clusters/1234/create-vm</code>
POST		Вызывает операцию, интерфейс к которой предоставляет ресурс
GET	Возвращает объект в теле ответа	Возвращает статус асинхронной операции в теле ответа
PUT	Загружает объект из тела запроса на ресурс	
PATCH	Обновляет какую-либо часть содержимого ресурса в соответствии с данными в теле запроса	
DELETE	Удаляет содержимое ресурса. Последующий запрос GET вернёт HTTP 404.	Отменяет асинхронную операцию

RESTful Naming Conventions (1)

Требований нет, есть рекомендации.

- URL формируются иерархически.
- Управляемые сущности именуются во множественном числе.
- Обращение без параметра возвращает массив объектов.
- Обращение с ИД возвращает конкретный объект.

RESTful Naming Conventions (2)

- Получить список поставщиков:

GET <http://www.example.com/customers>

- Добавить нового поставщика:

POST <http://www.example.com/customers>

- Получить поставщика с ИД=12345:

GET <http://www.example.com/customers/12345>

- Обновить данные о поставщике с ИД=12345:

PUT <http://www.example.com/customers/12345>

- Получить все заказы поставщика с ИД=12345:

GET <http://www.example.com/customers/12345/orders>



RESTful Naming Conventions (3)

- Сложносоставные слова рекомендуется заменять иерархией.
- Если без них всё-таки не обойтись, использовать snake-case (или hyphen-case, но не camel-case).
- Семантика осуществляемого действия располагается в методе, а не в URL.



RESTful Naming Conventions (4)

- Совсем плохо:

GET <http://www.example.com/getAllInvoicesForCustomer/12345>

- Всё ещё плохо:

GET <http://www.example.com/get-all-invoices-for-customer/12345>

- Уже лучше:

GET <http://www.example.com/customers/12345/get-all-invoices>

- Совсем хорошо:

GET <http://www.example.com/customers/12345/invoices>



Веб-сервис на базе сервлета

ИТМО BT

```
@WebServlet(value="/customers")
public class CustomerService extends HttpServlet {
    // Return all customers
    @Override
    public void doGet(...) { ... }
    // Create new customer
    @Override
    public void doPost(...) { ... }
    // Update existing customer
    @Override
    public void doPut (...) { ... }
    ...
}
```

4. JAX-RS

JAX-RS

- Спецификация API для разработки веб-сервисов.
- Аббревиатура от Jakarta RESTful Web Services.
- Разработана в составе проекта Apache Jakarta.
- Включена в спецификации Java SE 7+ и Java EE 6+.
- Есть эталонная реализация -- Jersey.



Особенности JAX-RS

- Позволяет создавать REST API к компонентам с помощью аннотаций.
- Часть Java EE -- работает на любом сервере приложений.
- Внутри сервисов доступны API всех компонентов Java EE.
- Аннотации можно применять внутри любых компонентов.



Аннотации JAX-RS

- `@Path` -- путь (URL) к ресурсу или методу.
- `@GET`, `@PUT`, `@POST`, `@DELETE` и `@HEAD` -- метод HTTP-запроса, который будет обработан ресурсом.
- `@Produces` -- тип возвращаемого контента (`text/html` etc).
- `@Consumes` -- тип обрабатываемого контента (`text/json` etc).

Вспомогательные аннотации JAX-RS (1)

- `@PathParam` -- отображает элемент иерархии URL на параметр метода.
- `@QueryParam` -- отображает параметр из URL на параметр метода.
- `@MatrixParam` -- отображает матричный параметр HTTP-запроса на параметр метода.
- `@HeaderParam` -- отображает заголовок HTTP-запроса на параметр метода.

Вспомогательные аннотации JAX-RS (2)

- `@CookieParam` -- отображает cookie на параметр метода.
- `@FormParam` – отображает параметр POST-запроса на параметр метода.
- `@DefaultValue` -- определяет значение по умолчанию для параметра метода.
- `@Context` -- позволяет получить контекстно-связанный объект (например, `@Context HttpServletRequest request`).



Пример сервиса JAX-RS (1)

```
import javax.ws.rs.core.MediaType;

@Path("/todo")
public class TodoResource {

    // This method is called if XML is requested
    @GET
    @Produces({MediaType.APPLICATION_XML})
    public Todo getXML() {
        Todo todo = new Todo();
        todo.setSummary("Application XML Todo Summary");
        todo.setDescription("Application XML Todo Description");
        return todo;
    }

    // This method is called if JSON is requested
    @GET
    @Produces({MediaType.APPLICATION_JSON})
    public Todo getJSON() {
```



Пример сервиса JAX-RS (2)

```
Todo todo = new Todo();
todo.setSummary("Application JSON Todo Summary");
todo.setDescription("Application JSON Todo Description");
return todo;
}

// This can be used to test the integration with the browser
@GET
@Produces({ MediaType.TEXT_XML })
public Todo getHTML() {
    Todo todo = new Todo();
    todo.setSummary("XML Todo Summary");
    todo.setDescription("XML Todo Description");
    return todo;
}

}
```



Пример клиента JAX-RS (1)

```
public class TodoTest {  
  
    public static void main(String[] args) {  
        ClientConfig config = new ClientConfig();  
        Client client = ClientBuilder.newClient(config);  
  
        WebTarget target = client.target(getBaseURI());  
        // Get XML  
        String xmlResponse = target.path("rest").path("todo").request()  
            .accept(MediaType.TEXT_XML).get(String.class);  
        // Get XML for application  
        String xmlAppResponse = target.path("rest").path("todo").request()  
            .accept(MediaType.APPLICATION_XML).get(String.class);
```



Пример клиента JAX-RS (2)

```
// Get JSON for application
String jsonResponse = target.path("rest").path("todo").request()
    .accept(MediaType.APPLICATION_JSON).get(String.class);

System.out.println(xmlResponse);
System.out.println(xmlAppResponse);
System.out.println(jsonResponse);
}

private static URI getBaseURI() {
    return UriBuilder.fromUri(
        "http://localhost:8080/com.vogella.jersey.jaxb").build();
}

}
```



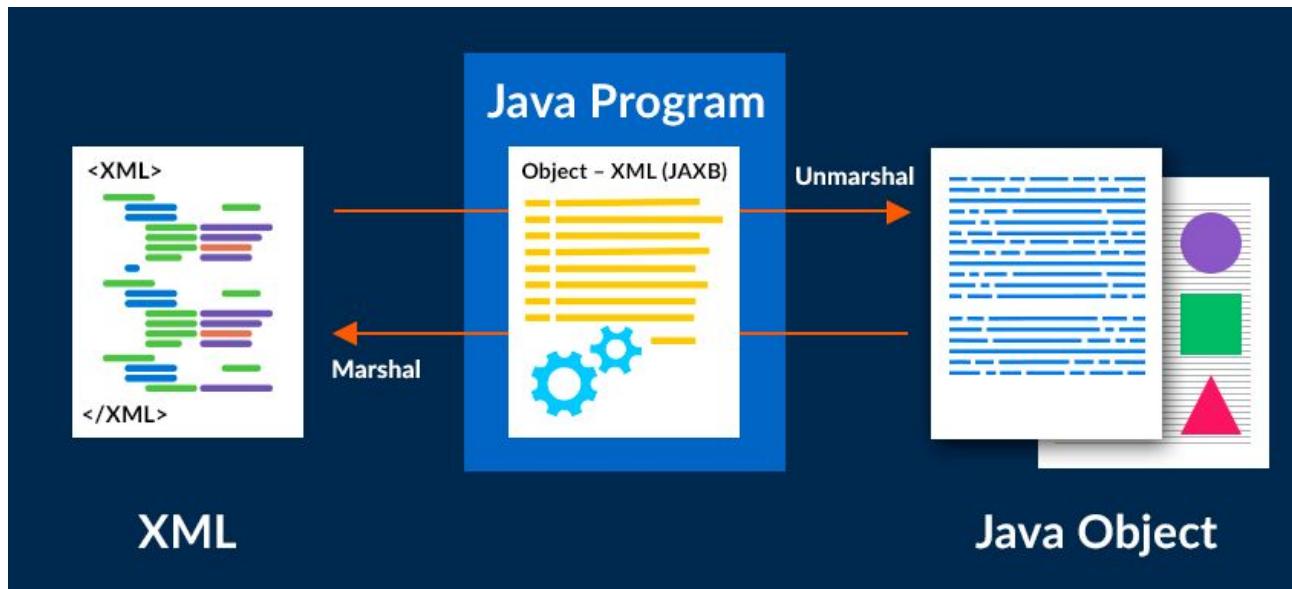
JAX-RS: разработка сервиса

Порядок действий:

1. Создаём проект (в случае Maven можно использовать архетип maven-archetype-webapp).
2. Добавляем зависимости JAX-RS (если версия JDK < 7).
3. Создаём описание представления ресурса (например, с помощью JAXB).
4. Создаём ресурс REST (сам веб-сервис).
5. Регистрируем ресурс.

JAX-RS: описание представления ресурса

- Опционально -- в принципе, можем передавать что угодно.
- В каноничном варианте реализуется с помощью аннотаций JAXB -- `@XmlRootElement`, `@XmlAttribute`, `@XmlElement` и т.д.





Описание представления ресурса

(. . .)

```
@XmlRootElement(name = "student")  
@XmlAttributeType(XmlAccessType.FIELD)  
public class Student {  
    @XmlAttribute  
    private Integer id;  
  
    @XmlElement  
    private String name;  
    ( . . . )  
}
```

Описание представления коллекции

(. . .)

```
@XmlRootElement(name = "student")  
@XmlAttributeType(XmlAccessType.FIELD)  
public class Configurations {  
    @XmlAttribute  
    private Integer size;  
  
    @XmlElement  
    private List<Student> students;  
  
( . . . )  
}
```



Ещё один пример сервиса (1)

```
@Path("/students")
@Produces("application/xml")
public class StudentResource {

    @GET
    public Students getStudents() {
        List<Student> list =
            StudentDB.getAllStudents();
        Students s = new Students();
        students.setStudents(list);
        return students;
    }
}
```

Ещё один пример сервиса (2)

```
@GET  
@Path("/{id}")  
public Response getStudentById(  
    @PathParam("id") Integer id) {  
    Student s = StudentDB.getStudent(id);  
    if(s == null) {  
        Return Response.status(NOT_FOUND)  
            .build();  
    } else {  
        return Response.status(OK).entity(s)  
            .build();  
    }  
(...)  
}
```



Регистрация сервиса (1)

(. . .)

```
@ApplicationPath("/student-management")
public class StudentApplication
        extends Application {
    private Set<Object> singletons =
        new HashSet<Object>();
    public StudentApplication() {
        singletons.add(
            new StudentResource()) ;
    }
}
```



Регистрация сервиса (2)

```
private Set<Class<?>> empty =  
        new HashSet<Class<?>>();  
  
@Override  
public Set<Class<?>> getClasses() {  
    return empty;  
}  
  
@Override  
public Set<Object> getSingletons() {  
    return singletons;  
}  
}
```

5. RESTful@Spring

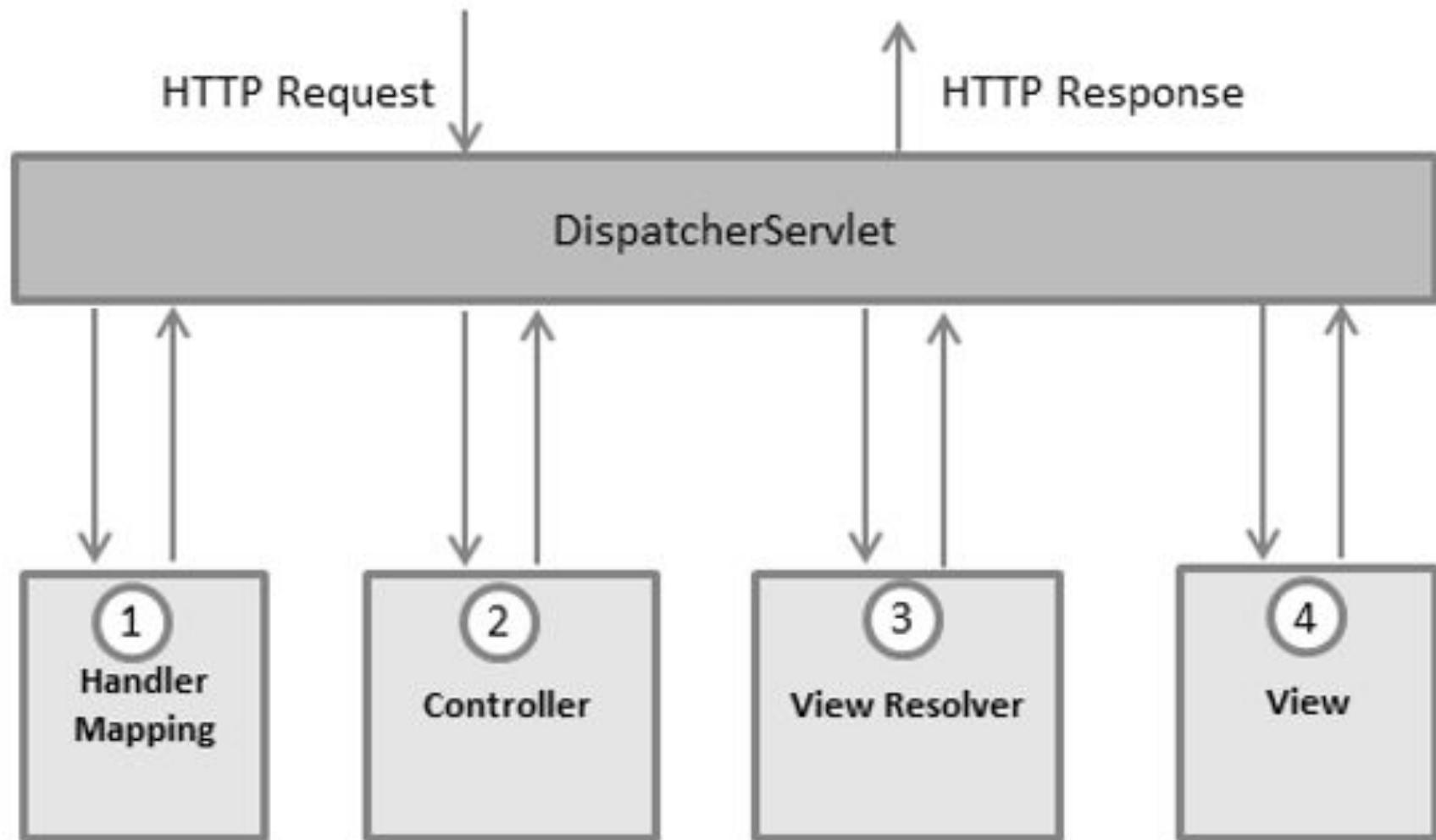
RESTful-сервисы на Spring WebMVC

- Spring Web MVC -- “базовый” фреймворк в составе Spring для разработки веб-приложений.
- Универсальный, на клиентской стороне интегрируется с популярными JS-фреймворками.
- Удобен и для разработки веб-сервисов.





Архитектура Spring MVC (1)



Из чего состоит приложение

- **Model** – инкапсулирует данные приложения (состоит из POJO или бинов).
- **View** -- отвечает за отображение данных модели.
- **Controller** -- обрабатывает запрос пользователя, создаёт соответствующую модель и передаёт её для отображения в представление.

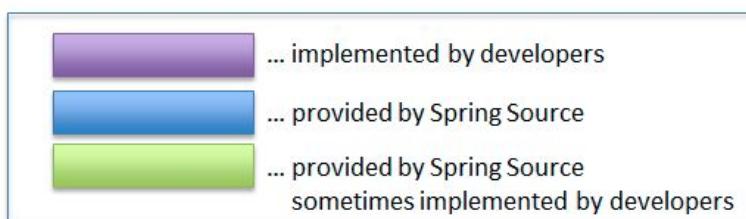
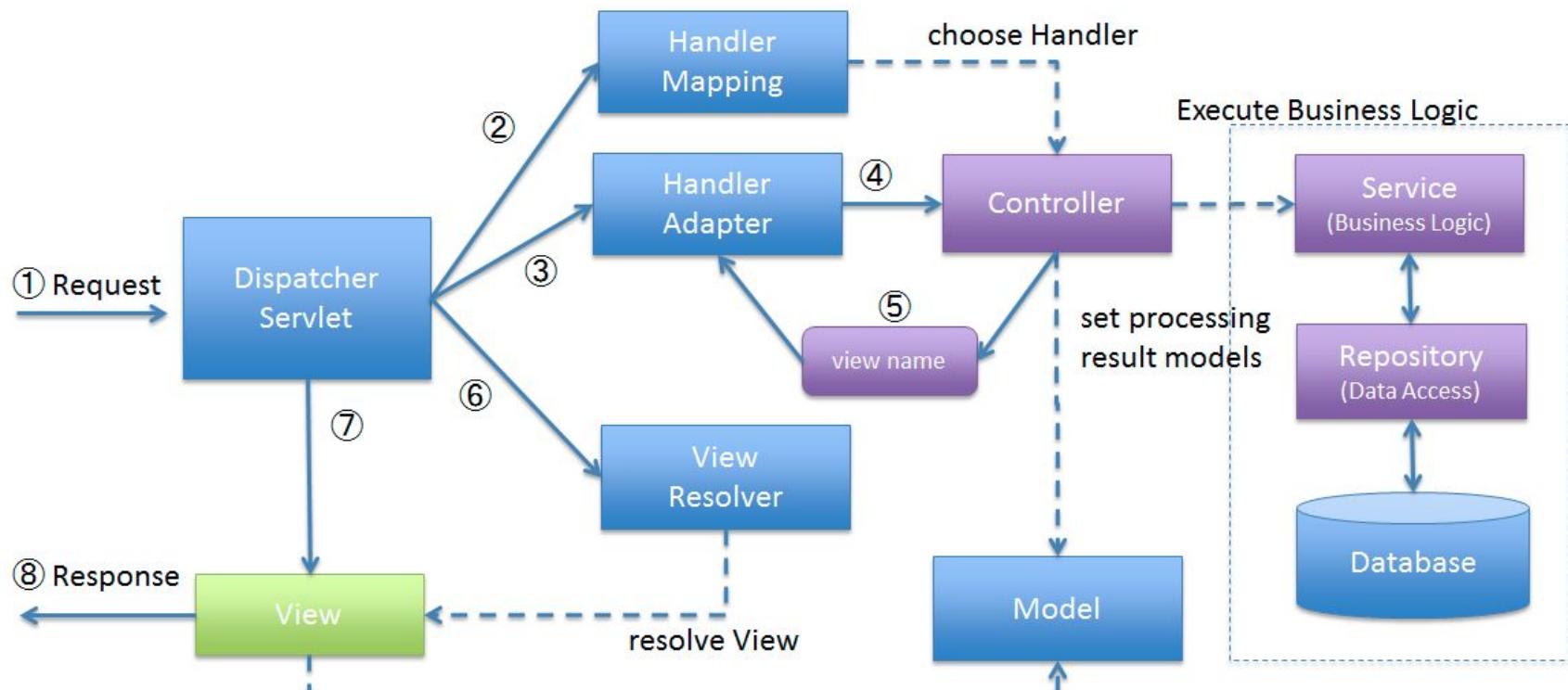


Dispatcher Servlet

- Обрабатывает все запросы и формирует ответы на них.
- Связывает между собой все элементы архитектуры Spring MVC.
- Обычный сервлет -- конфигурируется в web.xml.



Обработка запроса





Конфигурация web.xml

```
<web-app id = "WebApp_ID" version = "2.4"
    xmlns = "http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation = "http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <display-name>Spring MVC Application</display-name>

    <servlet>
        <servlet-name>HelloWeb</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloWeb</servlet-name>
        <url-pattern>*.jsp</url-pattern>
    </servlet-mapping>

</web-app>
```



Конфигурация HelloWeb-servlet.xml

```
<beans xmlns = "http://www.springframework.org/schema/beans"
       xmlns:context = "http://www.springframework.org/schema/context"
       xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation = "http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:component-scan base-package = "com.tutorialspoint" />

    <bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name = "prefix" value = "/WEB-INF/jsp/" />
        <property name = "suffix" value = ".jsp" />
    </bean>

</beans>
```



Создание контроллера

```
@Controller  
@RequestMapping("/hello")  
public class HelloController {  
    @RequestMapping(method = RequestMethod.GET)  
    public String printHello(ModelMap model) {  
        model.addAttribute("message", "Hello Spring MVC Framework!");  
        return "hello";  
    }  
}
```

Возвращаемое
представление



Атрибуты модели



@RequestBody & @ResponseBody

```
1  @Controller
2  @RequestMapping("/post")
3  public class ExamplePostController {
4
5      @Autowired
6      ExampleService exampleService;
7
8      @PostMapping("/response")
9      @ResponseBody
10     public ResponseTransfer postResponseController(
11         @RequestBody LoginForm loginForm) {
12         return new ResponseTransfer("Thanks For Posting!!!");
13     }
14 }
```

LoginForm будет
десериализован из
JSON

ResponseTransfer
будет сериализован в
JSON



ИТМО ВТ

@GetMapping

```
@GetMapping("/books")
public void book() {
    //
}

/* these two mappings are identical */

@RequestMapping(value = "/books", method = RequestMethod.GET)
public void book2() {

}
```

Есть аналогичные аннотации для Post, Put, Delete и Patch



@RequestParam

```
@PostMapping("/users")
/* First Param is optional */
public User createUser(
    @RequestParam(required = false)
        Integer age,
    @RequestParam String name) {
    // does not matter
}
```

Автоматическое преобразование параметров запроса в объект

```
@PostMapping("/users")
/* Spring преобразует userDto
автоматически, если в классе есть
getters and setters */
public User createUser(
    UserDto userDto) {
    //
}
```



@PathVariable

```
@GetMapping("/users/{userId}")
public User getUser(
    @PathVariable(required = false)
        String userId) {
    //...
    return user;
}
```

Two red arrows point to specific parts of the code: one arrow points to the `@PathVariable` annotation, and another arrow points to the `userId` parameter name.



Автоматическое преобразование параметров URL в объект

```
@GetMapping(  
    "/users/{userId}/{userName}")  
public User getUser(UserDto  
                      userDto) {  
  
    /* Will set "userId" &  
     * "userName" properties  
     * Automatically */  
    return user;  
}
```



Отображение методов на URL

```
@RestController  
@RequestMapping("/api/users") ←  
public class UserController {  
    @GetMapping(params = {"user_id"}) ←  
    public ResponseEntity<?> getUserId(  
        @RequestParam(name = "user_id") String userId) {  
        // Doesn't matter  
        return new ResponseEntity<>(user, HttpStatus.OK);  
    }  
  
    @GetMapping(params = {"email"}) ←  
    public ResponseEntity<?> getUserByEmail(  
        @RequestParam(name = "email") String email) {  
        // Doesn't matter  
        return new ResponseEntity<>(dtos, HttpStatus.OK);  
    }  
}
```

Один и тот же URL,
одинаковый тип
параметра



@RestController

```
1 @Controller  
2 @RequestMapping("books")  
3 public class SimpleBookController {  
4  
5     @GetMapping("/{id}", produces = "application/json")  
6     public @ResponseBody Book getBook(@PathVariable int id) {  
7         return findBookById(id);  
8     }  
9  
10    private Book findBookById(int id) {  
11        // ...  
12    }  
13 }
```

@Controller
+
@ResponseBody
=

@RestController

```
1 @RestController  
2 @RequestMapping("books-rest")  
3 public class SimpleBookRestController {  
4  
5     @GetMapping("/{id}", produces = "application/json")  
6     public Book getBook(@PathVariable int id) {  
7         return findBookById(id);  
8     }  
9  
10    private Book findBookById(int id) {  
11        // ...  
12    }  
13 }
```

Accept Header

- Клиент сообщает серверу, какой формат ответа он хочет получить от контроллера.
- Используется заголовок Accept:

```
GET http://localhost:8080  
      /transactions/{userid}
```

Accept: application/json



Ожидаемый формат ответа

HttpMessageConverter (1)

- Spring сам не умеет в сериализацию / маршалинг.
- Сериализация / маршалинг реализуются сторонними библиотеками.
- HttpMessageConverter -- адаптер для сторонних библиотек.
- Содержит 4 метода -- canRead (MediaType), canWrite (MediaType), read (Object, InputStream, MediaType) и write (Object, OutputStream, MediaType).



HttpMessageConverter (2)

Есть готовые конвертеры “из коробки”:

```
static {
    ClassLoader classLoader =
        AllEncompassingFormHttpMessageConverter
            .class.getClassLoader();
    jaxb2Present = ClassUtils.isPresent(
        "javax.xml.bind.Binder", classLoader);
    jackson2Present = /*...*/
    jackson2XmlPresent = /*...*/
    jackson2SmilePresent = /*...*/
    gsonPresent = /*...*/
    jsonbPresent = /*...*/
}
```

Использование Spring Boot (1)

Позволяет упростить конфигурацию приложения.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>
    Spring-boot-starter-parent
  </artifactId>
  <version>2.1.3.RELEASE</version>
  <relativePath/>
  <!-- lookup parent from repository -->
</parent>
( . . . )
```



Использование Spring Boot (2)

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>
            Spring-boot-starter-jdbc
        </artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>
            Spring-boot-starter-web
        </artifactId>
    </dependency>
    ( . . . )
</dependencies>
```

6. Spring Data REST

Spring Data REST

- Часть проекта Spring Data.
- Позволяет быстро создать REST API к репозиторию Spring Data.



Особенности Spring Data REST

- Ресурсы описываются в формате HAL.
- 3 основных вида ресурсов -- коллекция (collection), элемент (item) и ассоциация (association).
- Поддерживается постраничный вывод.
- Для коллекций поддерживается динамическая фильтрация.
- Специальный вид ресурсов -- поисковый (search resources) для вызова методов, формирующих поисковые запросы.
- Поддерживаются JPA, MongoDB, Neo4j, GemFire и Cassandra.

Hypertext Application Language (HAL)

- “Work-in-progress” стандарт для описания гипермедиа-ресурсов (hypermedia resources).
- Гипермедиа -- расширение гипертекста (+ графика, видео, звук и т.д.).
- Две нотации -- JSON и XML.



Пример ресурса HAL

```
{  
    "_links": {  
        "self": {  
            "href": "http://example.com/api/book/hal-cookbook"  
        }  
    },  
    "_embedded": {  
        "author": {  
            "_links": {  
                "self": {  
                    "href": "http://example.com/api/author/shahadat"  
                }  
            },  
            "id": "shahadat",  
            "name": "Shahadat Hossain Khan",  
            "homepage": "http://author-example.com"  
        }  
    },  
    "id": "hal-cookbook",  
    "name": "HAL Cookbook"  
}
```



Использование Spring Data REST

1. Добавляем зависимость в Maven / Gradle.
2. Конфигурируем.
3. Выбираем стратегию экспорта репозитория.
4. Выбираем базовый URI.
5. Запускаем приложение.



Конфигурация Maven / Gradle

- Зависимость в Maven:

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-rest-webmvc</artifactId>
    <version>3.3.4.RELEASE</version>
</dependency>
```

- Зависимость в Gradle:

```
dependencies {
    ... other project dependencies
    compile("org.springframework.data
        :spring-data-rest-webmvc
        :3.3.4.RELEASE")}
```

Конфигурация Spring Data REST

- Не требуется, если используем Spring Boot.
- Задаётся в классе `RepositoryRestMvcConfiguration`, который необходимо импортировать в конфигурацию приложения.
- Изменяется путём регистрации своего конфигуратора `RepositoryRestConfigurer` или наследования от класса адаптера `RepositoryRestConfigurerAdapter`.

Стратегия экспорта репозитория

Наименование	Описание
DEFAULT	Открывает наружу все публичные интерфейсы репозитория, но учитывает флаг <code>exported</code> в аннотациях <code>@(Repository) RestResource</code> .
ALL	Открывает наружу все интерфейсы репозитория без учёта модификаторов доступа и аннотаций.
ANNOTATION	Открывает наружу только ресурсы, помеченные аннотациями <code>@(Repository) RestResource</code> с учётом значения флага <code>exported</code> .
VISIBILITY	Открывает наружу только публичные аннотированные ресурсы.



Изменение базового URI

- Задаётся в application.properties:

```
spring.data.rest.basePath=/api
```

- Может быть задано в RepositoryRestConfigurer:

```
@Component
```

```
public class CustomizedRestMvcConfiguration extends  
RepositoryRestConfigurerAdapter {
```

```
@Override
```

```
public void configureRepositoryRestConfiguration(  
    RepositoryRestConfiguration config) {
```

```
    config.setBasePath("/api");
```

```
}
```

```
}
```



Пример ресурса

```
@RepositoryRestResource(collectionResourceRel = "people", path = "people")
public interface PersonRepository extends MongoRepository<Person, String> {

    // Prevents GET /people/:id
    @Override
    @RestResource(exported = false)
    public Person findOne(String id);

    // Prevents GET /people
    @Override
    @RestResource(exported = false)
    public Page<Person> findAll(Pageable pageable);

    // Prevents POST /people and PATCH /people/:id
    @Override
    @RestResource(exported = false)
    public Person save(Person s);

    // Prevents DELETE /people/:id
    @Override
    @RestResource(exported = false)
    public void delete(Person t);

}
```



Метаданные о ресурсе

ИТМО ВТ

```
curl -v http://localhost:8080/
```

```
< HTTP/1.1 200 OK
```

```
< Content-Type: application/hal+json
```

```
{ "_links" : {  
    "orders" : {  
        "href" : "http://localhost:8080/orders"  
    },  
    "profile" : {  
        "href" : "http://localhost:8080/api/alps"  
    }  
}
```

Коллекция -- Collection Resource

- URL ресурса -- имя класса в нижнем регистре и множественном числе -- /api/orders.
- Поддерживаемые методы HTTP -- только GET и POST.
- Все остальные методы HTTP возвращают 405 Method Not Allowed.
- “Открываться наружу” будут следующие методы:
 - findAll (Pageable)
 - findAll (Sort)
 - findAll ()

Элемент -- Item Resource

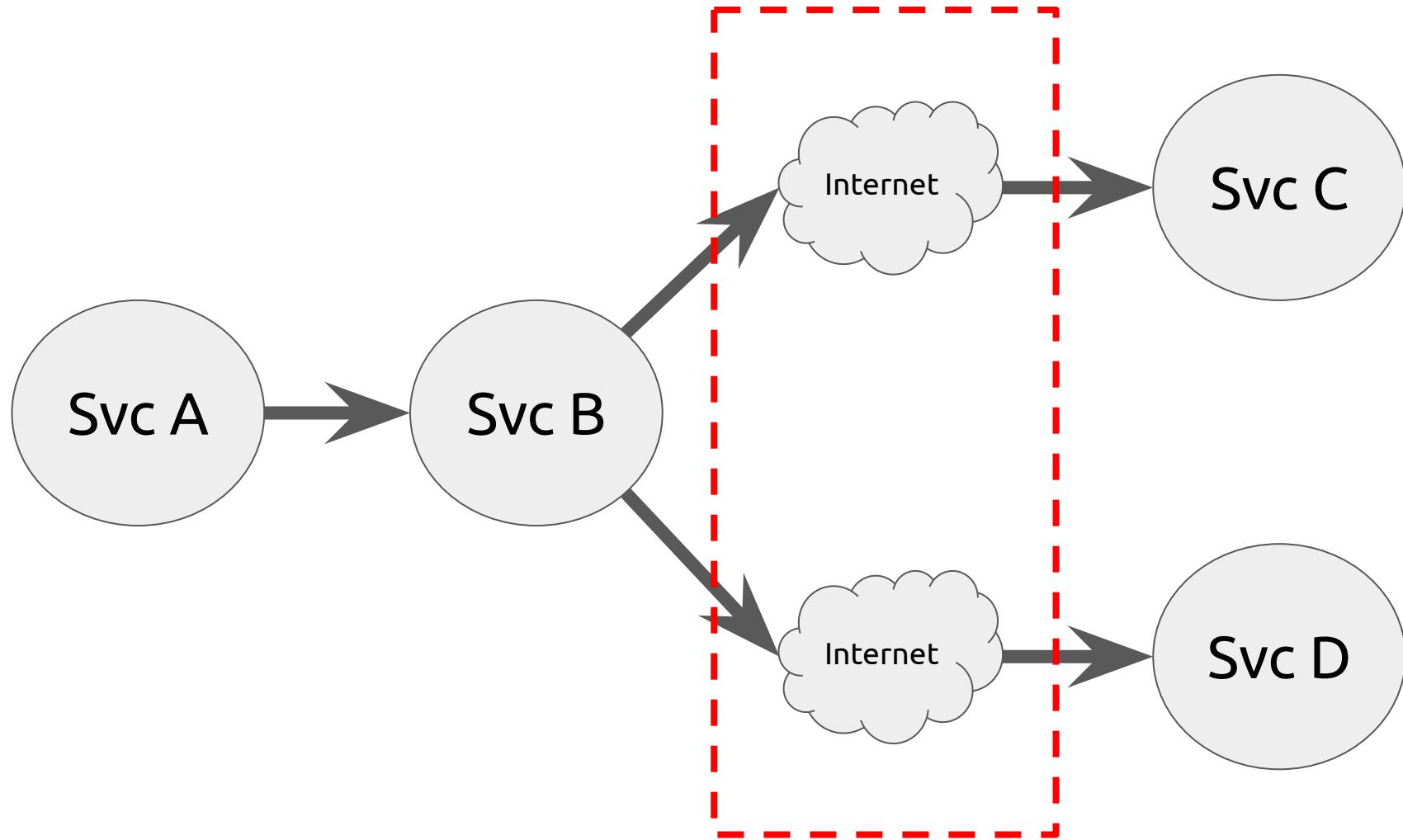
- Ресурс для взаимодействия с отдельными элементами коллекции
- Поддерживаемые методы HTTP -- GET, PUT, PATCH и DELETE.
- “Открываться наружу” будут следующие методы:
 - findById(...) // GET
 - save(...) // PUT, PATCH
 - delete(...) // DELETE

Ассоциация -- Association Resource

- Ресурс для взаимодействия с ресурсами, “вложенными” в свойства основного.
- Поддерживаемые методы HTTP -- GET, PUT, POST и DELETE.
- Метод POST поддерживается только в ситуации, когда в ассоциации находится коллекция.

7. Введение в криптографию

Взаимодействие веб-сервисов



Особенности решения

- Протоколы -- TLS / SSL.
- Клиентом и сервером являются веб-сервисы.
- Используется инфраструктура JRE / JDK и сервера приложений.

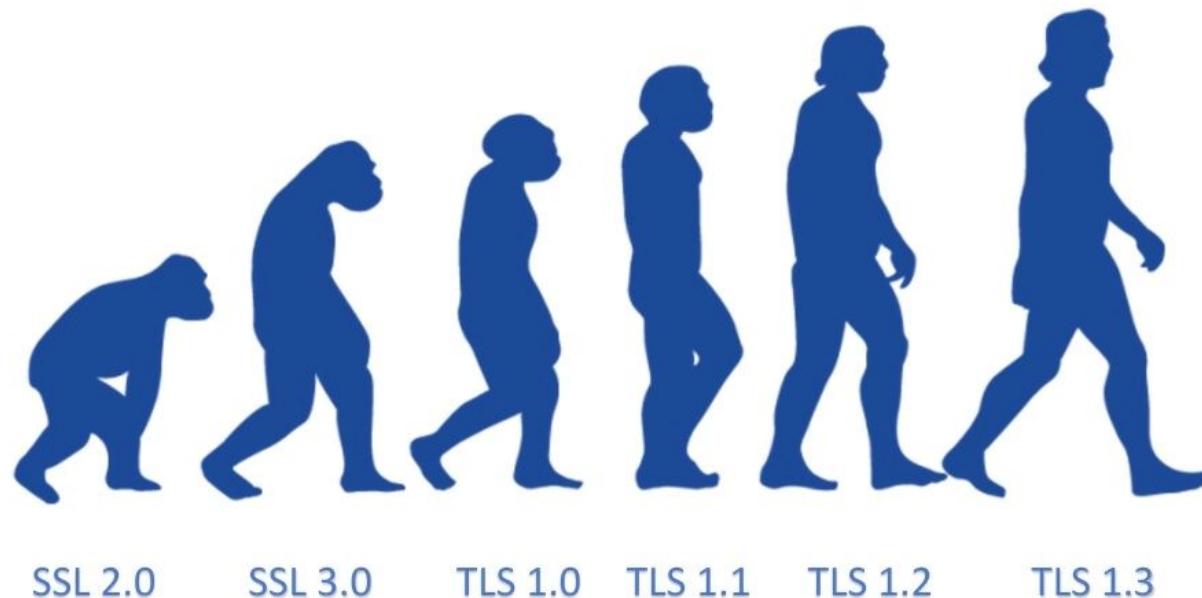


SSL -- Secure Sockets Layer

- Криптографический протокол.
- Предложен в 1995 г. в браузере Netscape Navigator.
- Использует асимметричную криптографию для аутентификации ключей обмена, симметричное шифрование для сохранения конфиденциальности, коды аутентификации сообщений для целостности сообщений.
- Со временем должен быть исключён в пользу TLS.

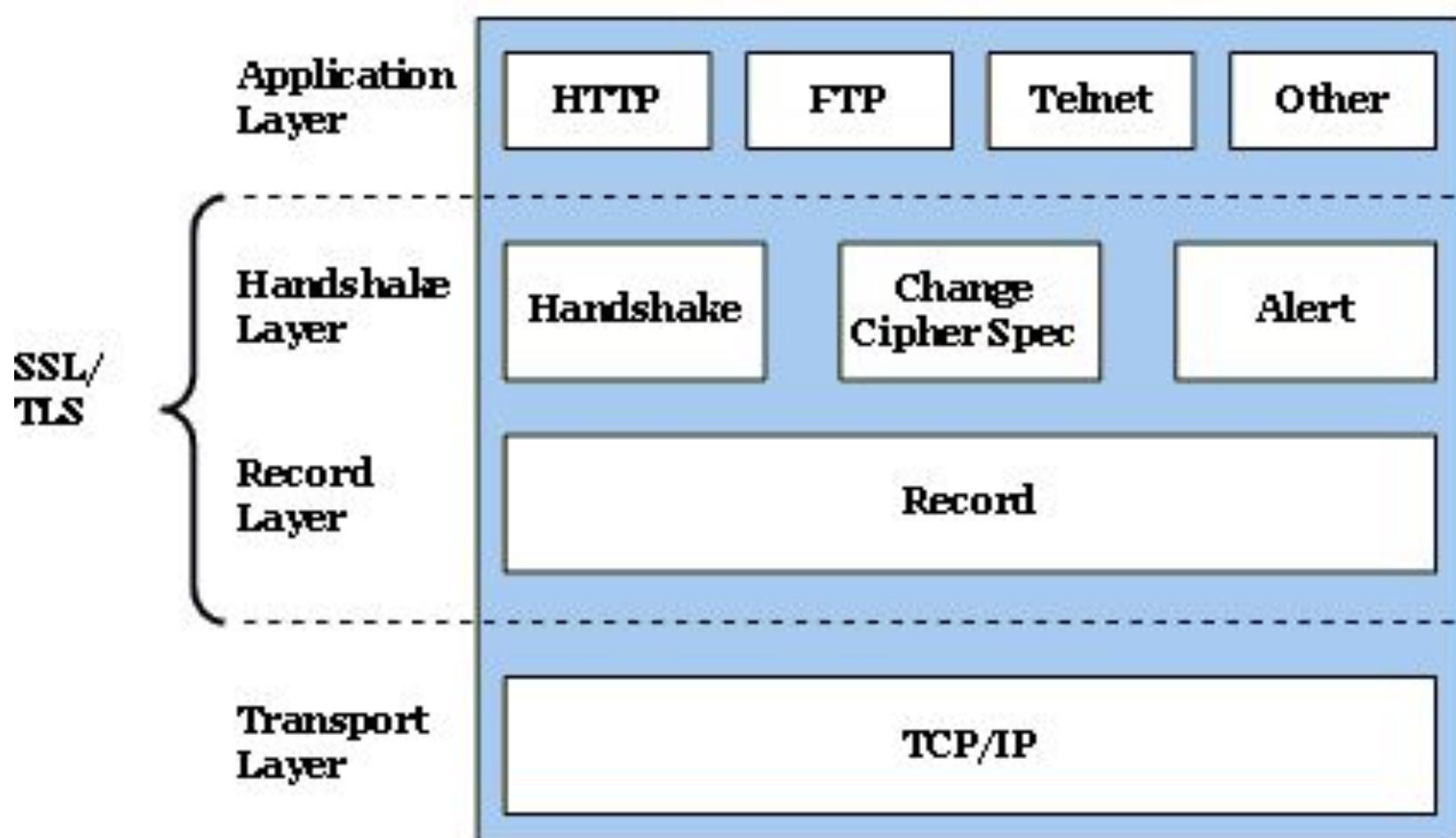
TLS -- Transport Layer Security

- Основан на SSL 3.0.
- Предложен в 1999 г.
- Актуальная версия -- 1.3 (2018 г.).
- Обратно совместим с SSL v3.



TLS / SSL в иерархии протоколов

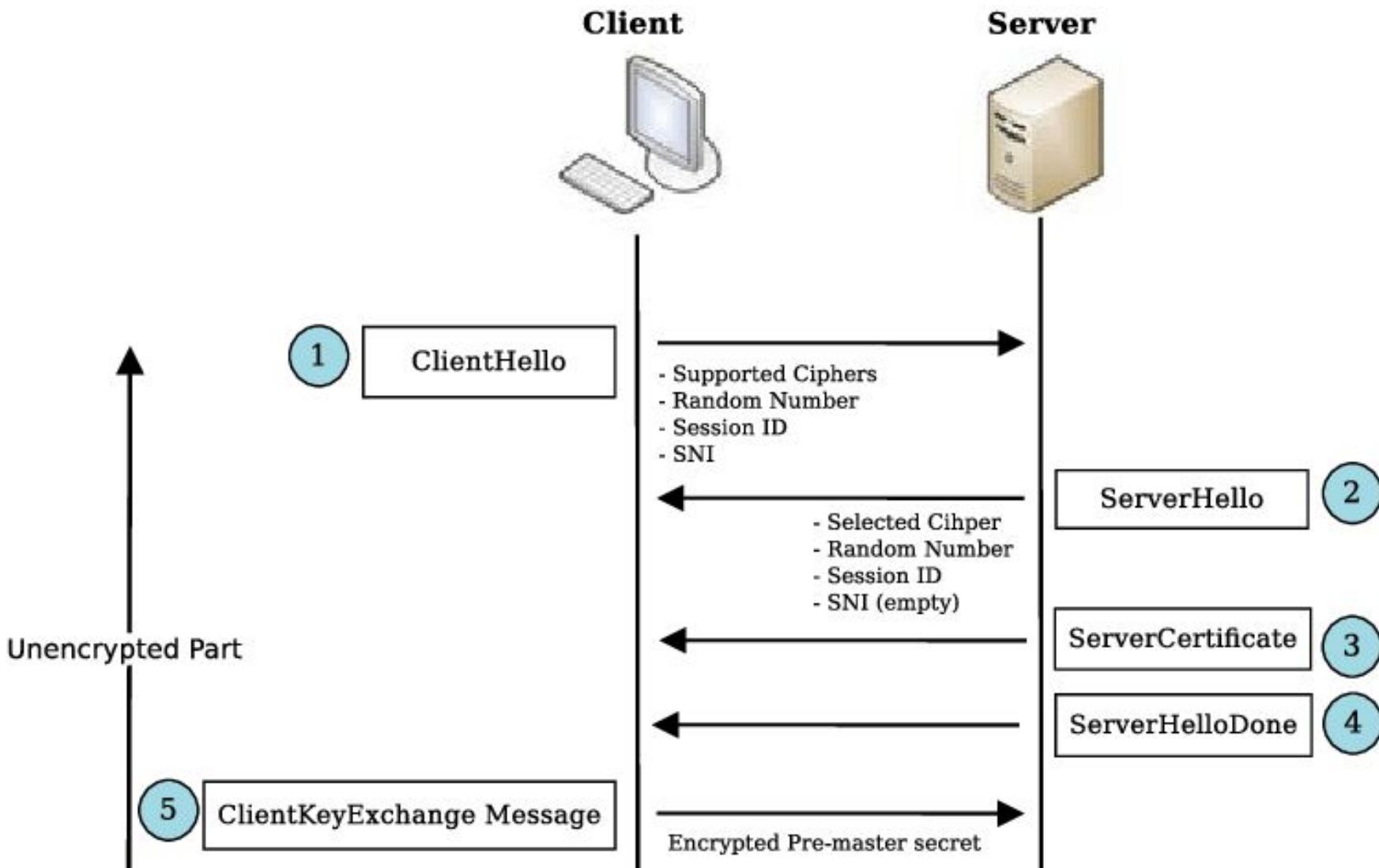
SSL/TLS Protocol Layers



Внутренняя иерархия TLS / SSL

- Слой протокола подтверждения подключения (Handshake Protocol Layer):
 - Протокол подтверждения подключения (Handshake Protocol).
 - Протокол изменения параметров шифра (Cipher Spec Protocol).
 - Предупредительный протокол (Alert Protocol).
- Слой протокола записи (Record Protocol Layer).

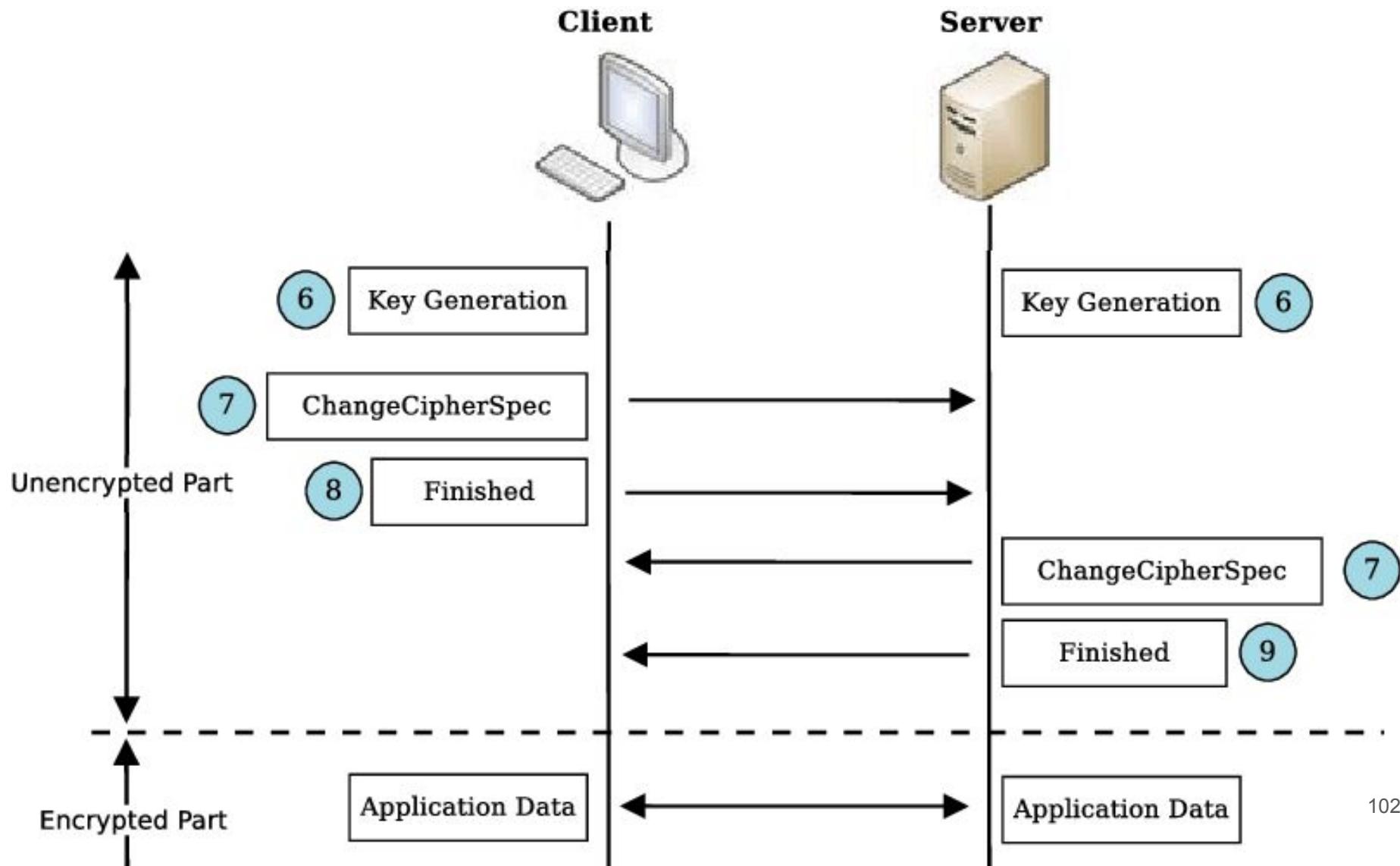
Подтверждение связи (Handshake) (1)





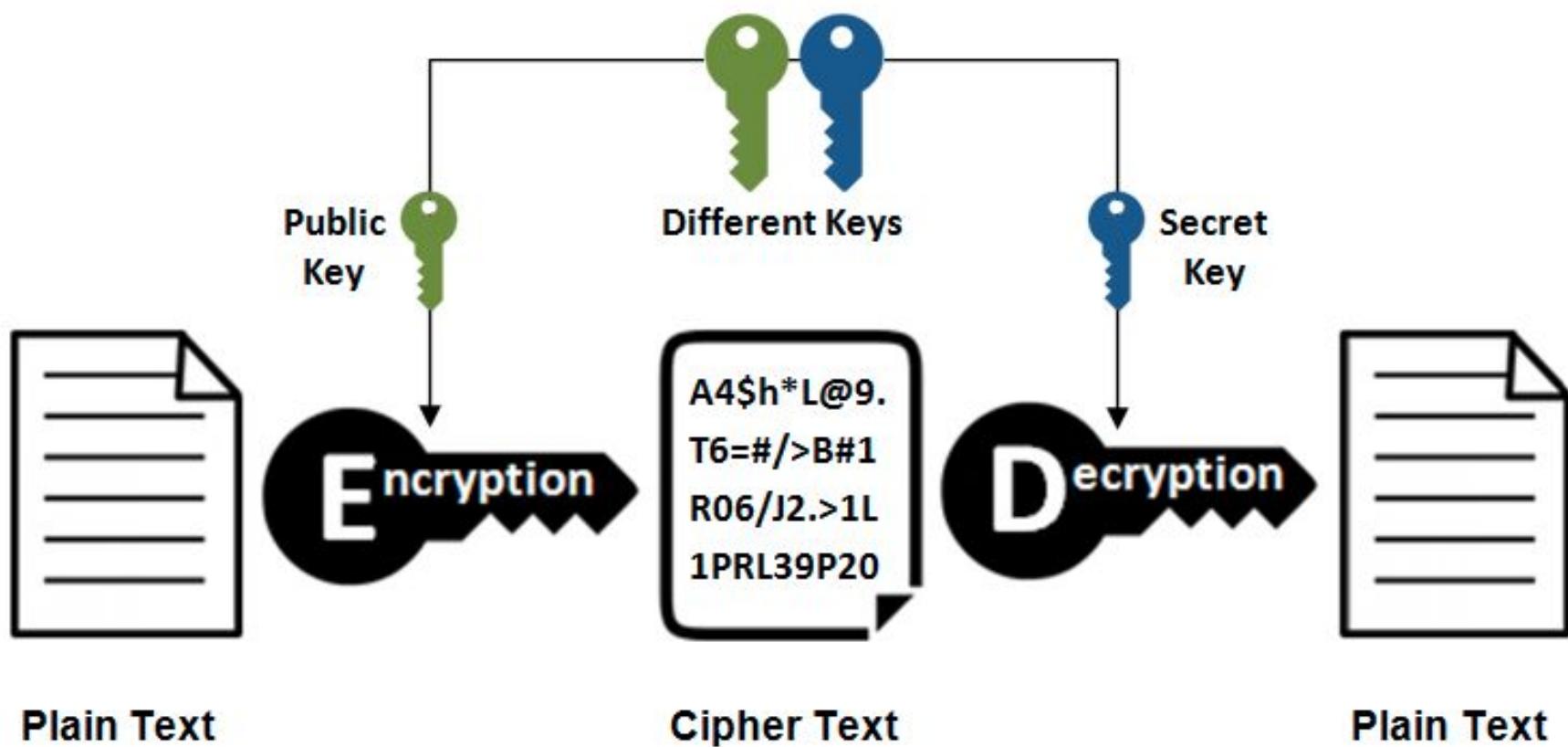
Подтверждение связи (Handshake) (2)

ИТМО ВТ



Асимметричное шифрование

Asymmetric Encryption



Plain Text

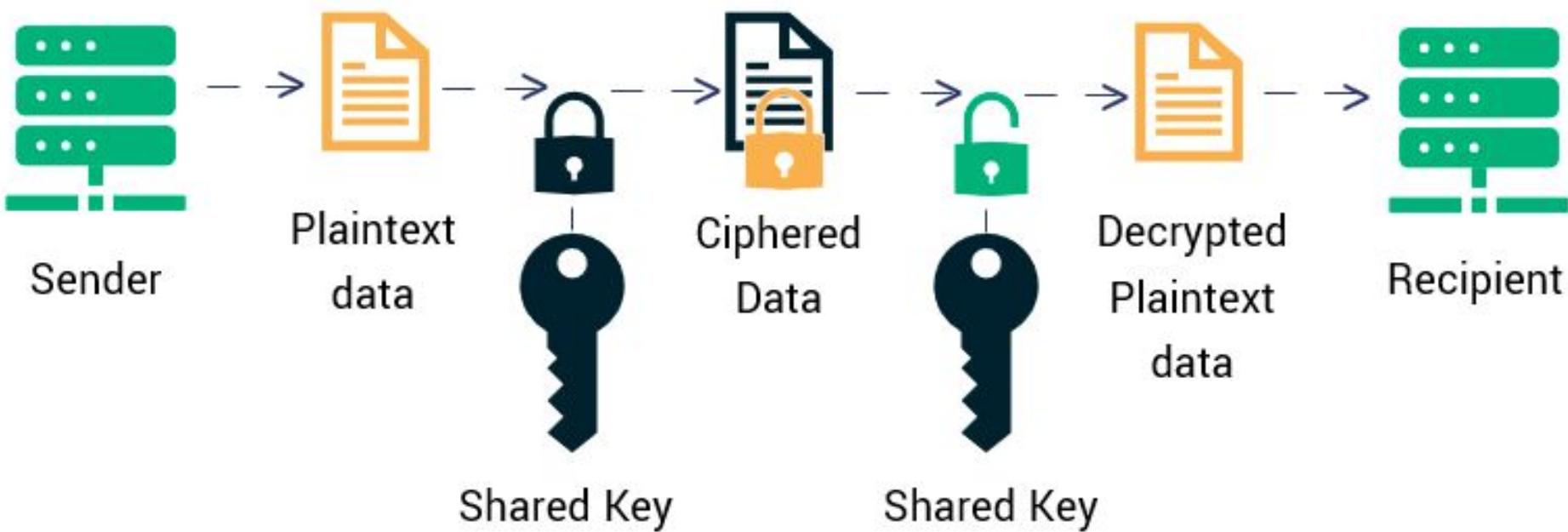
Cipher Text

Plain Text



Симметричное шифрование

Symmetric Encryption



Сертификаты TLS / SSL

- Используются для проверки принадлежности открытого ключа его реальному владельцу.
- Возможные способы получения:
 - Сертификат, выданный СА.
 - Самоподписанный сертификат.
 - “Пустой” сертификат.



Центр сертификации (CA)

- Центр сертификации, удостоверяющий центр (Certification Authority, CA) -- организация, “чья честность неоспорима, а открытый ключ широко известен” (C).
- Подтверждает подлинность ключей шифрования своим сертификатом.
- Обычно сертификаты объединяются в цепочки.



Иерархия сертификатов



Common name: *.aliyun.com
Organization: Taobao(China) Software Co., Ltd
Valid from: May 23, 2016 to July 22, 2017
Issuer: Symantec Class 3 Secure Server CA - G4



Common name: Symantec Class 3 Secure Server CA - G4
Organization: Symantec Corporation
Valid from: October 31, 2013 to October 30, 2023
Issuer: VeriSign Class 3 Public Primary Certification Authority - G5



Common name: VeriSign Class 3 Public Primary Certification Authority - G5
Organization: VeriSign, Inc.
Valid from: November 08, 2006 to July 16, 2036
Issuer: VeriSign Class 3 Public Primary Certification Authority - G5

Самоподписанный сертификат

- Сертификат, выданный самим его субъектом.
- Не может быть отзван.
- Технически, все сертификаты СА являются самоподписанными.

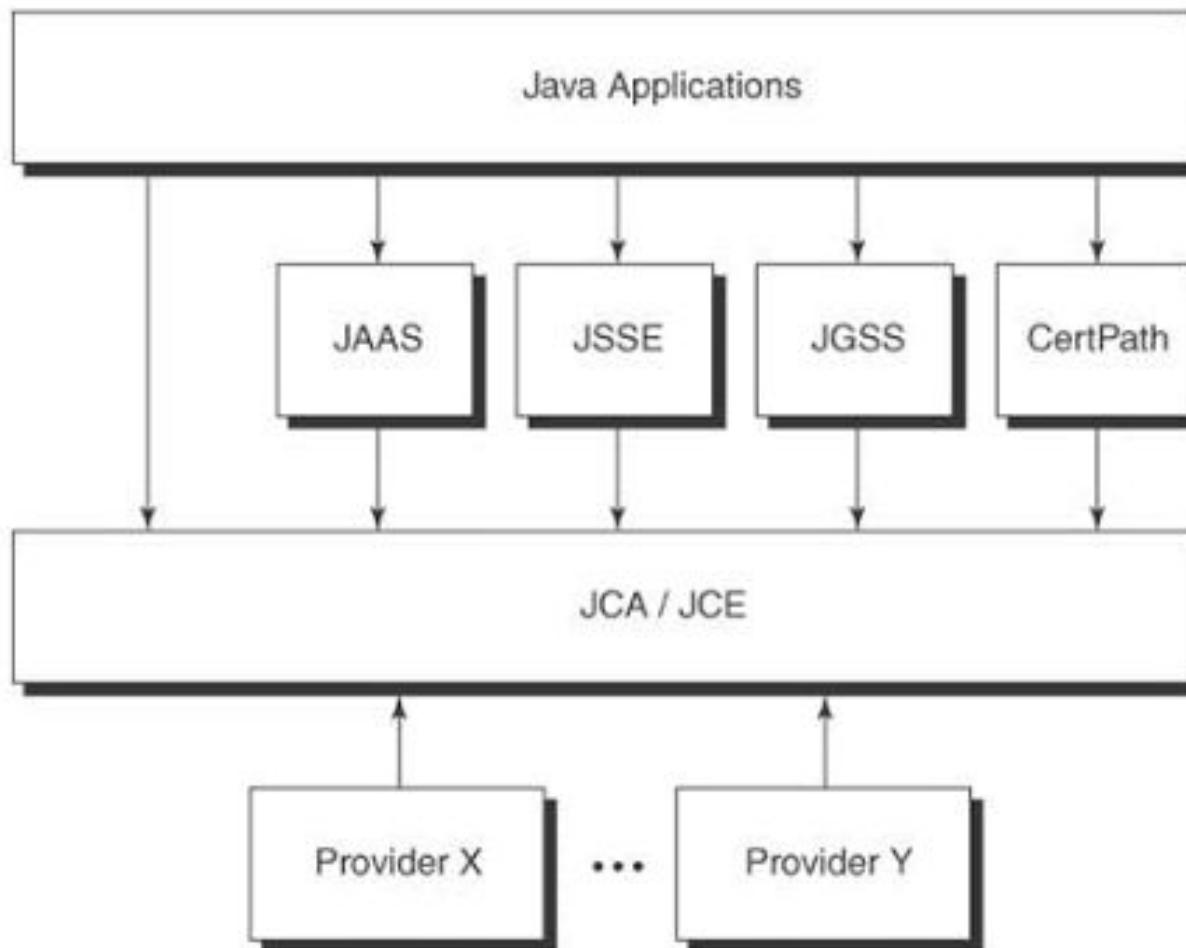


8. Криптография в Java

Введение

- Есть спецификация Java Cryptography Architecture (JCA, не путать с Java Connector Architecture!).
- В случае “корпоративных” приложений обычно реализуется на инфраструктурном уровне.

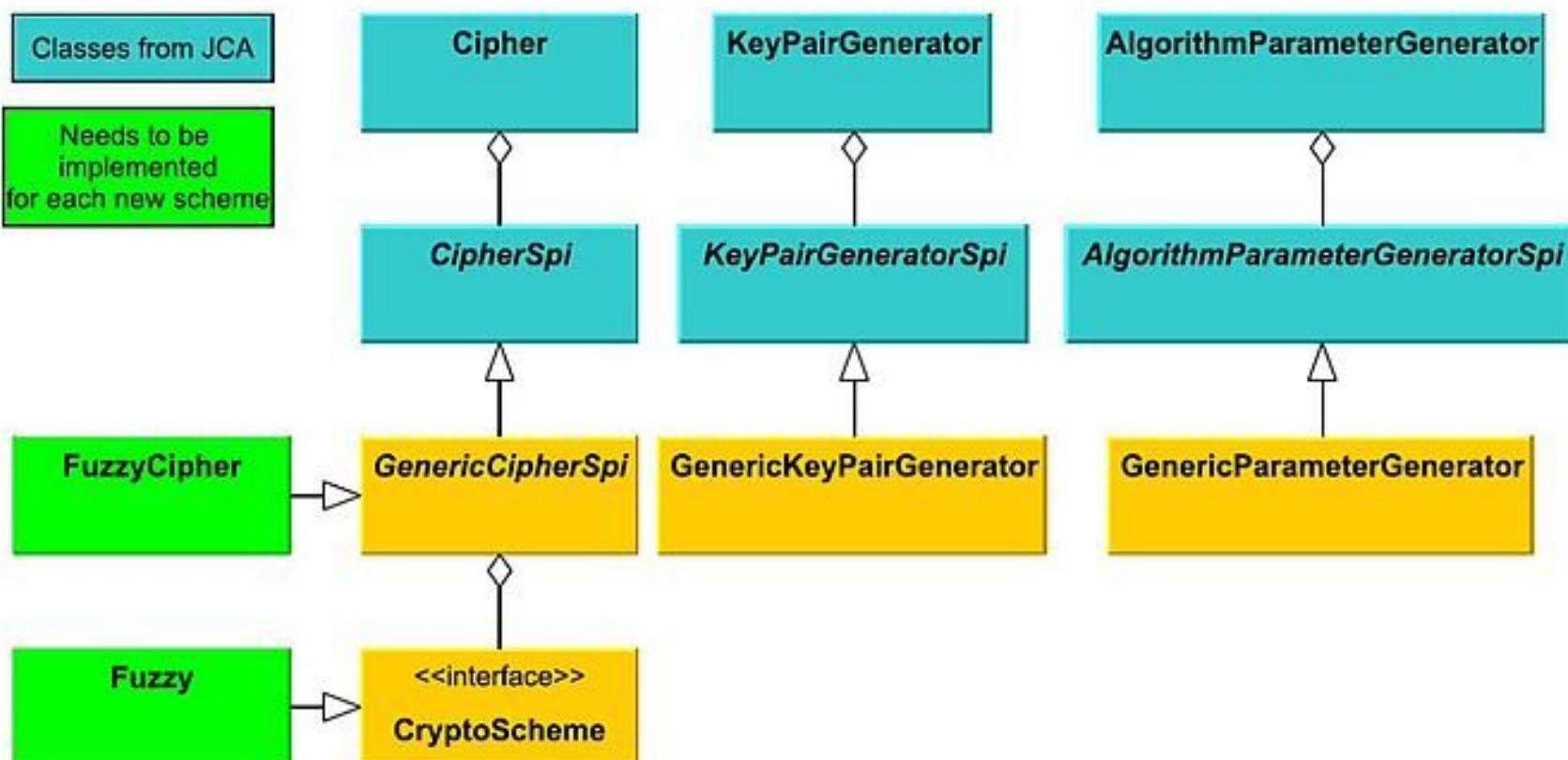
JCA



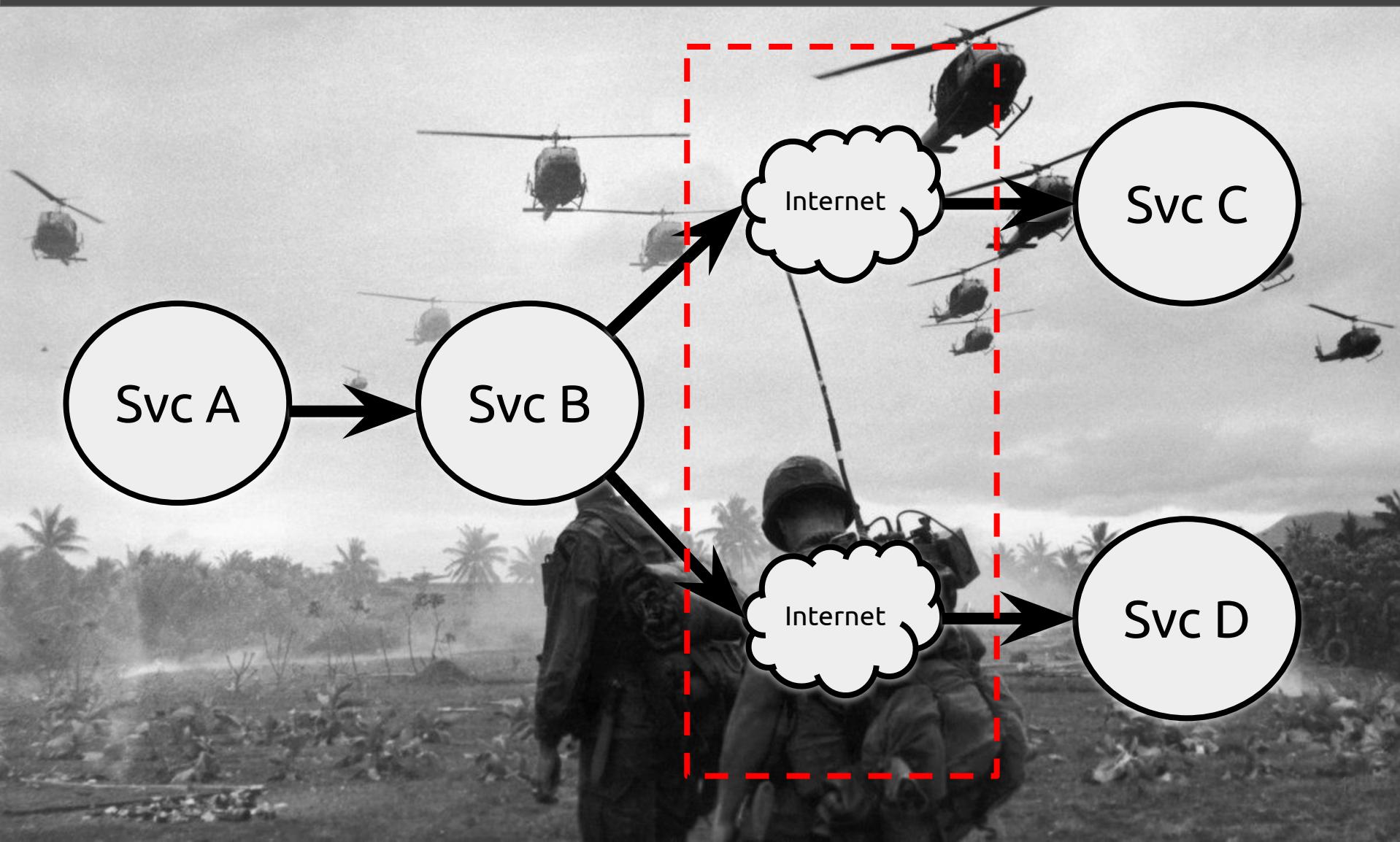


ИТМО BT

JCA API (Out-of-Scope)



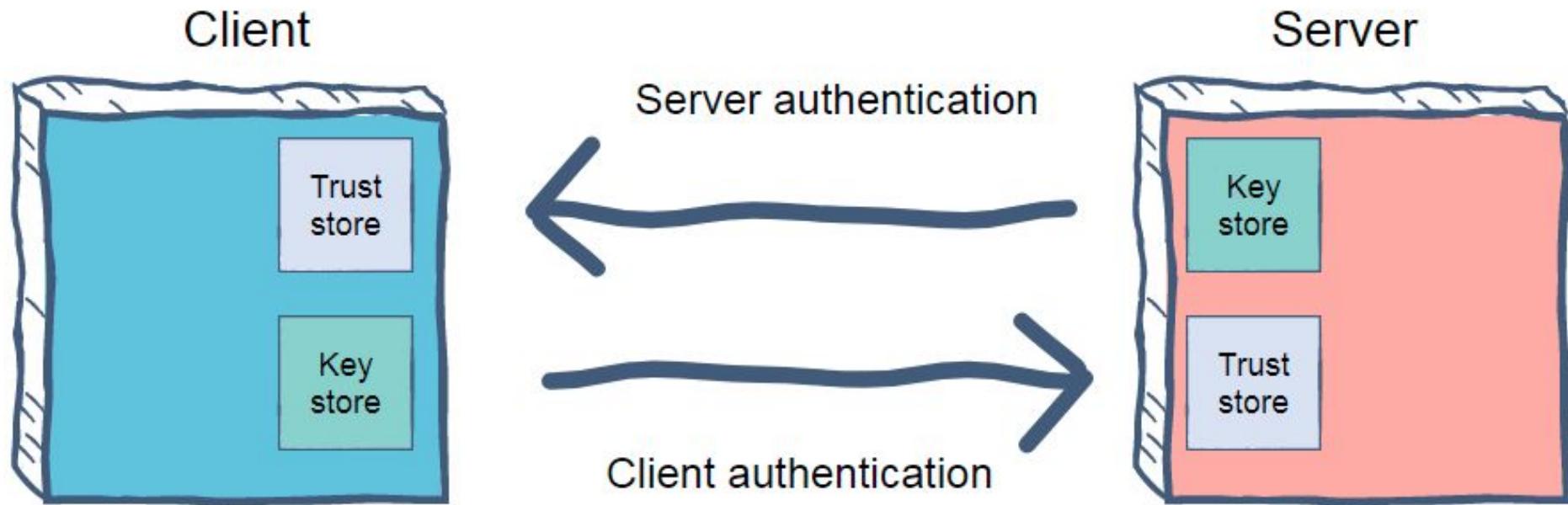
Взаимная аутентификация сервисов





Keystore & Truststore (1)

- В Java обычно используется проприетарный формат JKS.
- Для работы с хранилищами используется утилита keytool.





Keystore & Truststore (2)

<i>Keystore</i>	<i>TrustStore</i>
Хранятся приватные ключи и сертификаты (клиентские или серверные)	Хранятся доверенные сертификаты (корневые самоподписанные CA root)
Необходим для настройки SSL на сервере	Необходим для успешного подключения к серверу на клиентской стороне
Клиент будет хранить свой приватный ключ и сертификат в keystore	Сервер будет валидировать клиента при двусторонней аутентификации на основании сертификатов в trustStore
Используется API <code>javax.net.ssl.KeyStore</code>	Используется API <code>javax.net.ssl.TrustStore</code>



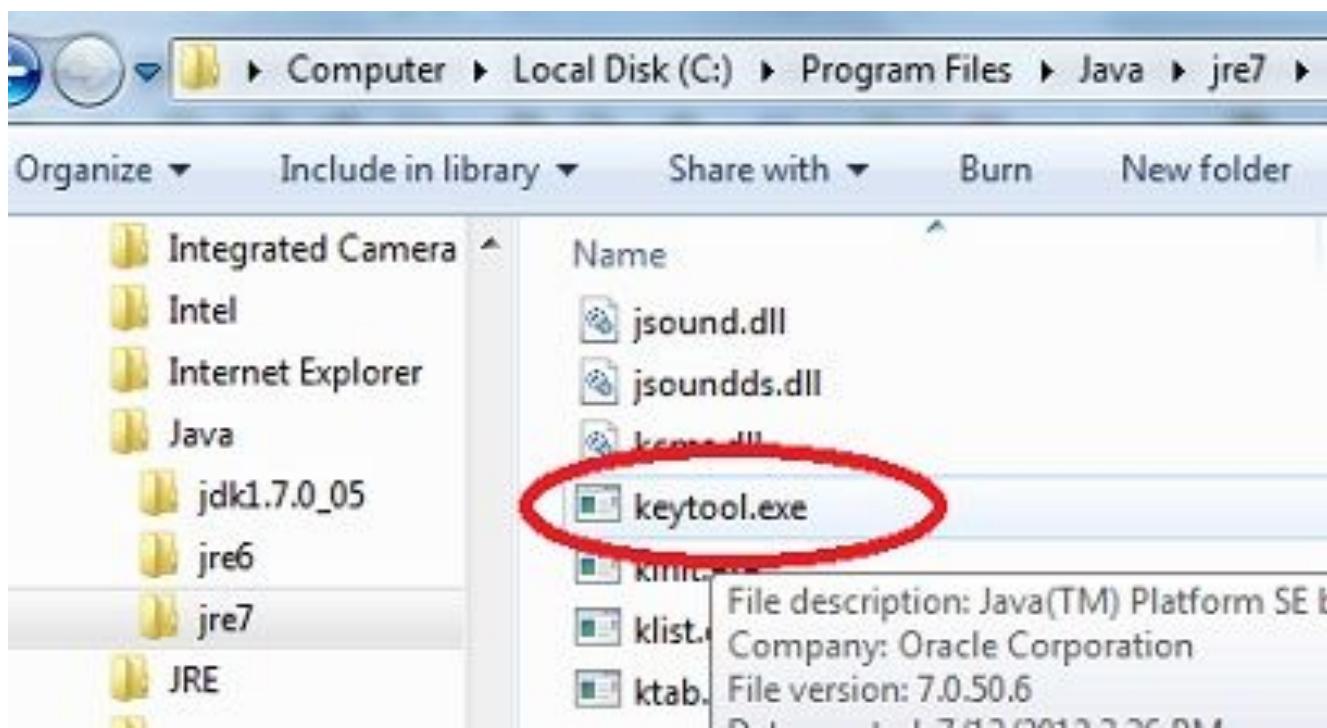
Keystore & Truststore (3)

- В JDK/JRE есть truststore “по умолчанию” --
\$JAVA_HOME/lib/security/cacerts.
- Пароль -- changeit.
- Сервер приложений обычно идёт в комплекте
со своими keystore и truststore.



Утилита keytool

- Предназначена для работы с хранилищами JKS.
- Идёт в комплекте поставки JRE.





Команды keytool (1)

- Создать ключи вместе с keystore:

```
keytool -genkey -alias example.com  
-keyalg RSA -keystore keystore.jks  
-keysize 2048
```

- Создать запрос сертификата (CSR) для существующего Java keystore:

```
keytool -certreq -alias example.com  
-keystore keystore.jks -file  
example.com.csr
```



Команды keytool (2)

- Загрузить корневой или промежуточный CA сертификат:

```
keytool -import -trustcacerts -alias  
root -file Thawte.crt -keystore  
keystore.jks
```

- Импортировать доверенный сертификат:

```
keytool -import -trustcacerts -alias  
example.com -file example.com.crt  
-keystore keystore.jks
```



Команды keytool (3)

- Сгенерировать самоподписанный сертификат и keystore:

```
keytool -genkey -keyalg RSA -alias  
selfsigned -keystore keystore.jks  
-storepass password -validity 360  
-keysize 2048
```

- Посмотреть сертификат:

```
keytool -printcert -v -file  
example.com.crt
```

Команды keytool (4)

- Посмотреть список сертификатов:

```
keytool -list -v -keystore  
keystore.jks
```

- Проверить конкретный сертификат по алиасу:

```
keytool -list -v -keystore  
keystore.jks -alias example.com
```

- Удалить сертификат:

```
keytool -delete -alias example.com  
-keystore keystore.jks
```



Команды keytool (5)

- Изменить пароль keystore:

```
keytool -storepasswd -new  
new_storepass -keystore keystore.jks
```

- Экспортировать сертификат из keystore:

```
keytool -export -alias example.com  
-file example.com.crt -keystore  
keystore.jks
```



Команды keytool (6)

- Посмотреть список доверенных корневых сертификатов:

```
keytool -list -v -keystore  
$JAVA_HOME/jre/lib/security/cacerts
```

- Добавить новый корневой сертификат в truststore:

```
keytool -import -trustcacerts -file  
/path/to/ca/ca.pem -alias CA_ALIAS  
-keystore  
$JAVA_HOME/jre/lib/security/cacerts
```

Пример 1: Подключение SSL в Spring Boot для REST Controller

```
server.ssl.key-store-type=JKS  
server.ssl.key-store=classpath:cert.jks  
server.ssl.key-store-password=changeit  
server.ssl.key-alias=key  
trust.store=classpath:cert.jks  
trust.store.password=changeit
```



Пример 2: Настройка SSL в GlassFish (1)

1. Генерируем приватный ключ:

```
keytool -keysize 2048 -genkey -alias slas  
-keyalg RSA -dname  
"CN=*.mydomain.com, O=Myorganization, L=city, S=  
state, C=country" -keypass changeit -storepass  
changeit -keystore keystore.jks
```

2. Генерируем CSR:

```
keytool -certreq -alias youralias -file  
yourcsrname.csr -keystore  
yourkeystorename.jks
```

Пример 2: Настройка SSL в GlassFish (2)

3. Скачиваем файлы сертификата (3 штуки, их должен выдать CA).
4. Импортируем файлы сертификата:

```
keytool -import -trustcacerts -alias root  
-file (ROOT CERTIFICATE FILE NAME) -keystore  
domain.key  
  
keytool -import -trustcacerts -alias intermed  
-file (INTERMEDIATE CA FILE NAME) -keystore  
domain.key  
  
keytool -import -alias mydomain.com -keystore  
keystore.jks -trustcacerts -file  
mydomain.com.crt
```

Пример 2: Настройка SSL в GlassFish (3)

5. Импортируем файлы в keystore по умолчанию:

```
keytool -importkeystore -srckeystore  
yourkeystorename.jks -destkeystore  
keystore.jks
```

SO MUCH WIN!



Пример 3: Два узла WildFly, самоподписанный сертификат (1)

1. Генерируем серверный сертификат:

```
$ keytool -genkeypair -alias localhost  
-keyalg RSA -keysize 2048 -validity 365  
-keystore server.keystore -dname "cn=Server  
Administrator, o=Acme, c=GB" -keypass secret  
-storepass secret
```

2. Копируем keystore на сервер приложений:

```
$ cp server.keystore  
$JBOSS_HOME/standalone/configuration
```

Пример 3: Два узла WildFly, самоподписанный сертификат (2)

3. Генерируем клиентский сертификат:

```
$ keytool -genkeypair -alias client -keyalg RSA -keysize 2048 -validity 365 -keystore client.keystore -dname "CN=client" -keypass secret -storepass secret
```

4. Экспортируем содержимое клиентского и серверного keystore в файлы сертификатов:

```
$ keytool -exportcert -keystore server.keystore -alias localhost -keypass secret -storepass secret -file server.crt  
$ keytool -exportcert -keystore client.keystore -alias client -keypass secret -storepass secret -file client.crt
```

Пример 3: Два узла WildFly, самоподписанный сертификат (3)

5. Импортируем сертификаты в клиентский и серверный truststore:

```
$ keytool -importcert -keystore  
server.truststore -storepass secret -alias  
client -trustcacerts -file client.crt  
-noprompt  
  
$ keytool -importcert -keystore  
client.truststore -storepass secret -alias  
localhost -trustcacerts -file server.crt  
-noprompt
```

Пример 3: Два узла WildFly, самоподписанный сертификат (4)

6. Копируем клиентский truststore в конфигурацию сервера приложений:

```
$ cp client.truststore  
$JBOSS_HOME/standalone/configuration
```



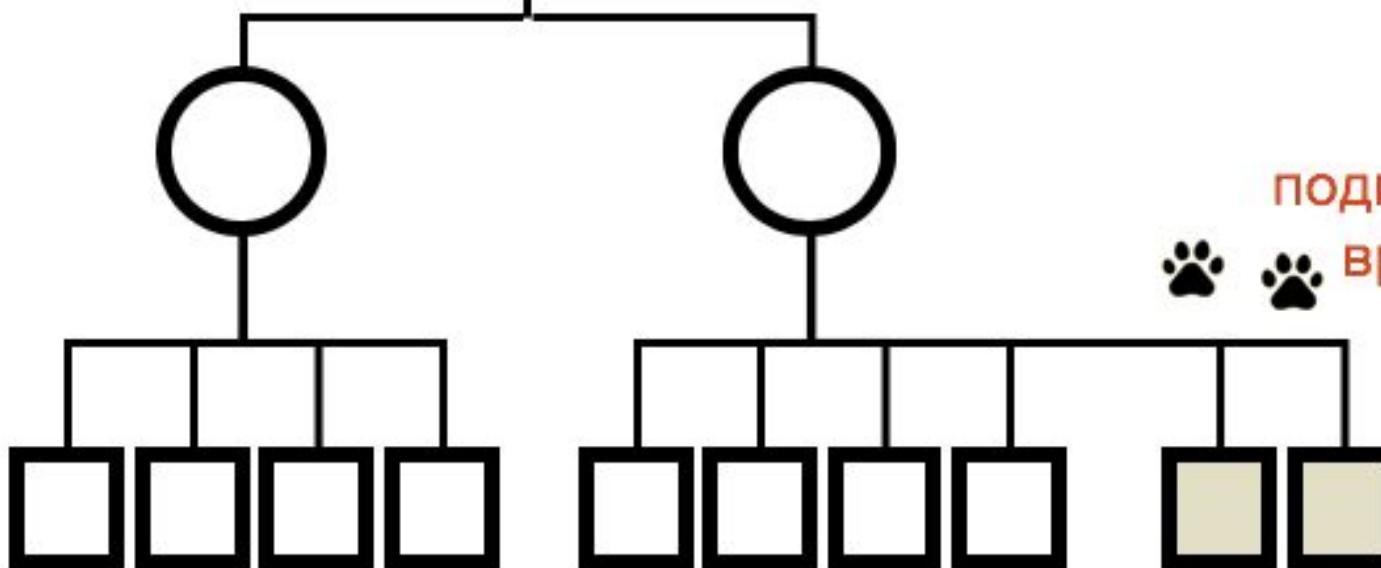
9. Service Discovery



Проблема

балансировщик

балансировщик



5000 серверов
приложений

5000 серверов
приложений

1000 новых
серверов



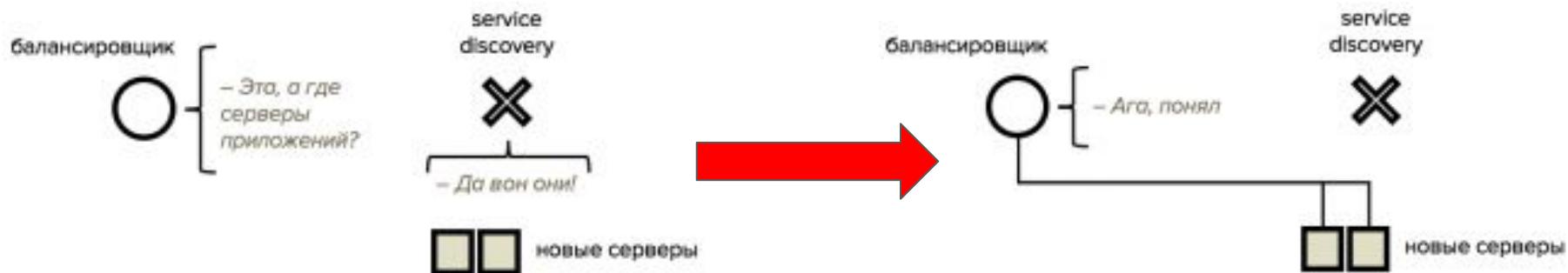
подключение
вручную

Описание подхода

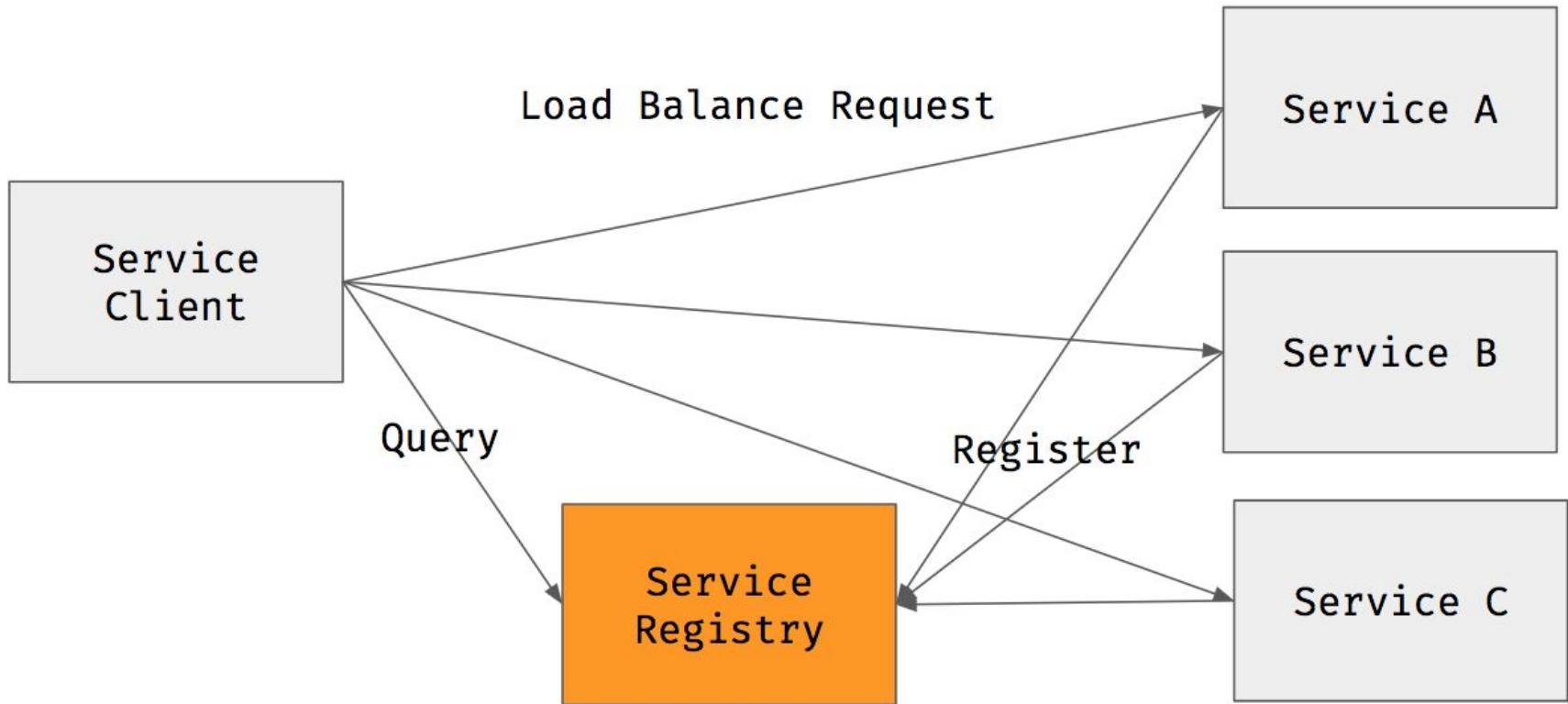
- **Обнаружение сервисов (*Service Discovery*)** — автоматическое определение устройств и сервисов, предоставляемых этими устройствами в компьютерной сети.
- **Протоколы обнаружения сервисов (*Service Discovery Protocol*)** — сетевые протоколы, реализующие SD.
- Таких протоколов много: DNS Service Discovery (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)...

Решаемые задачи

- Реконфигурация системы.
- Упрощение администрирования.
- Горизонтальная масштабируемость.



Общая схема



Особенности и варианты реализации

- 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]).

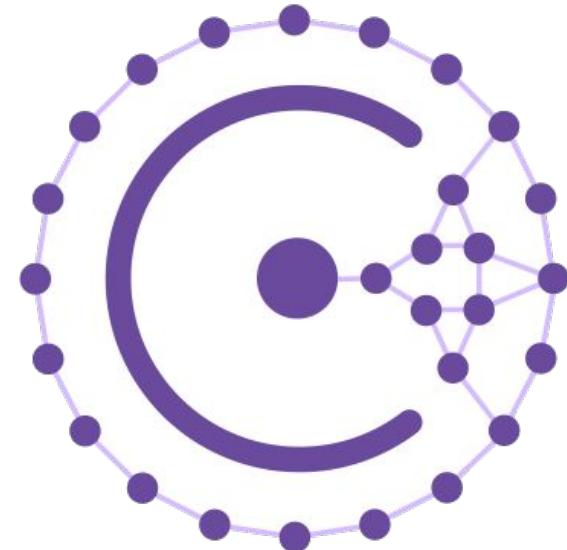
Инфраструктурное ПО

- Конфигурируем SD на уровне управления виртуальными машинами / контейнерами.
- Подробнее – см. модуль 11.

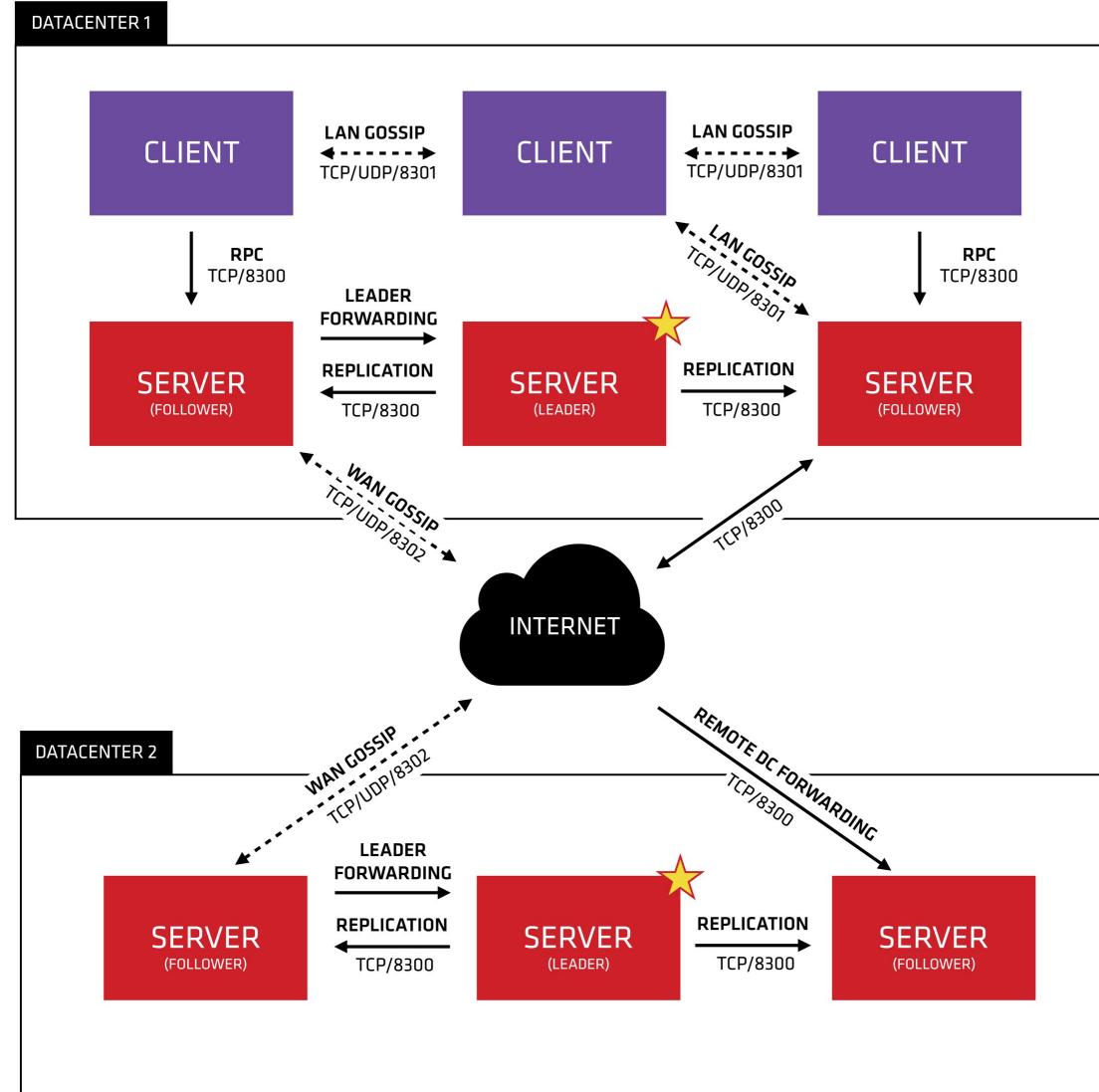


Специальное ПО: Consul

- Система обнаружения сервисов с распределенным хранилищем “ключ-значение”.
- Использует Gossip в качестве протокола обмена данными.
- Распределённая; ПО ставится на каждый узел сети.
- Сервисы регистрируются путём создания файлов в формате json.



Архитектура Consul





Регистрация сервиса в Consul

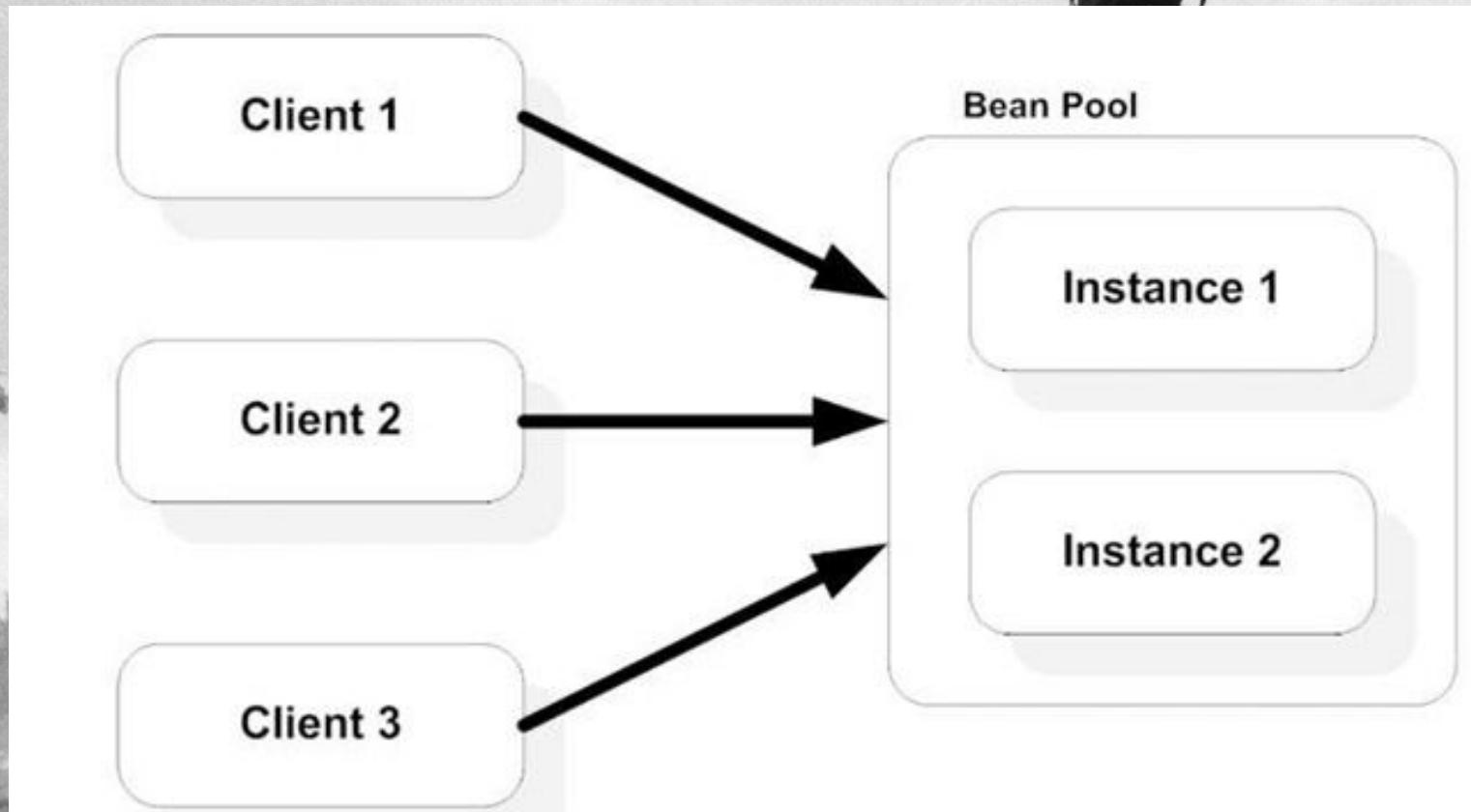
/etc/consul.d/mongodb.json:

```
{  
  "service": {  
    "name": "mongo-db",  
    "tags": ["mongo"],  
    "address": "123.23.34.56",  
    "port": 27017,  
    "checks": [  
      {  
        "name": "Checking MongoDB"  
        "script": "/usr/bin/check_mongo.py --host 123.23.34.56 --port 27017",  
        "interval": "5s"  
      }  
    ]  
  }  
}
```

- 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.



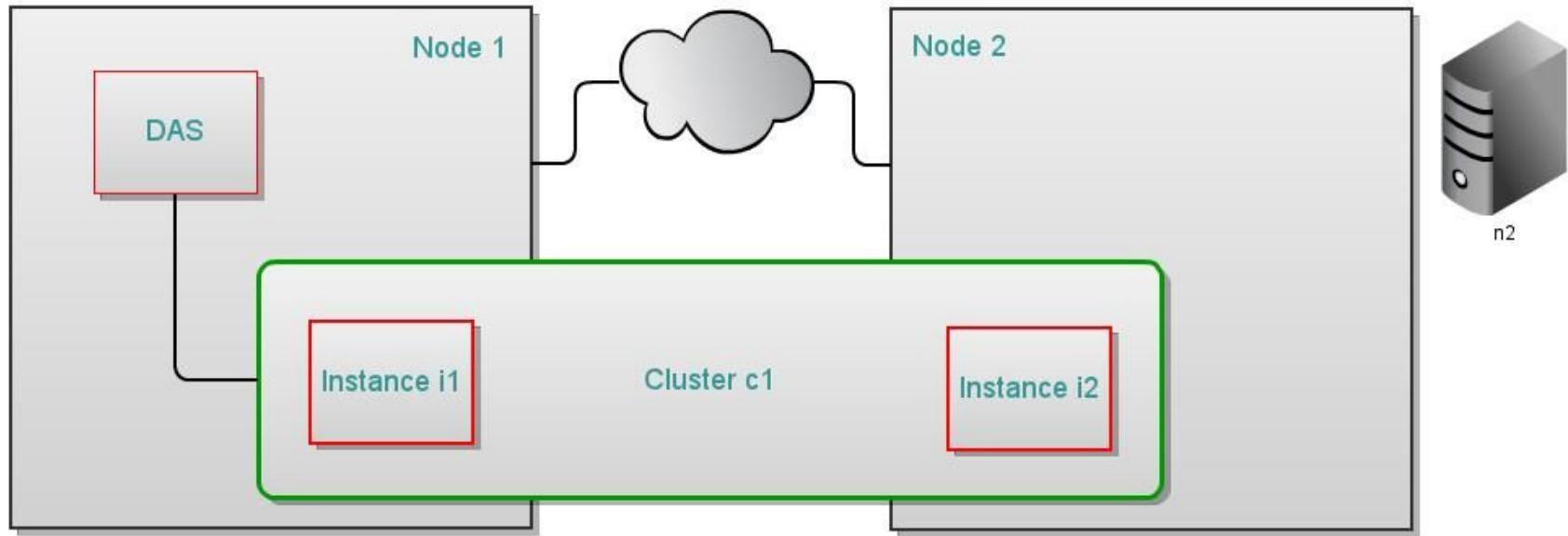
Java EE: EJB Pool @ GlassFish Cluster (1)





Java EE: EJB Pool @ GlassFish Cluster (2)

ИТМО BT





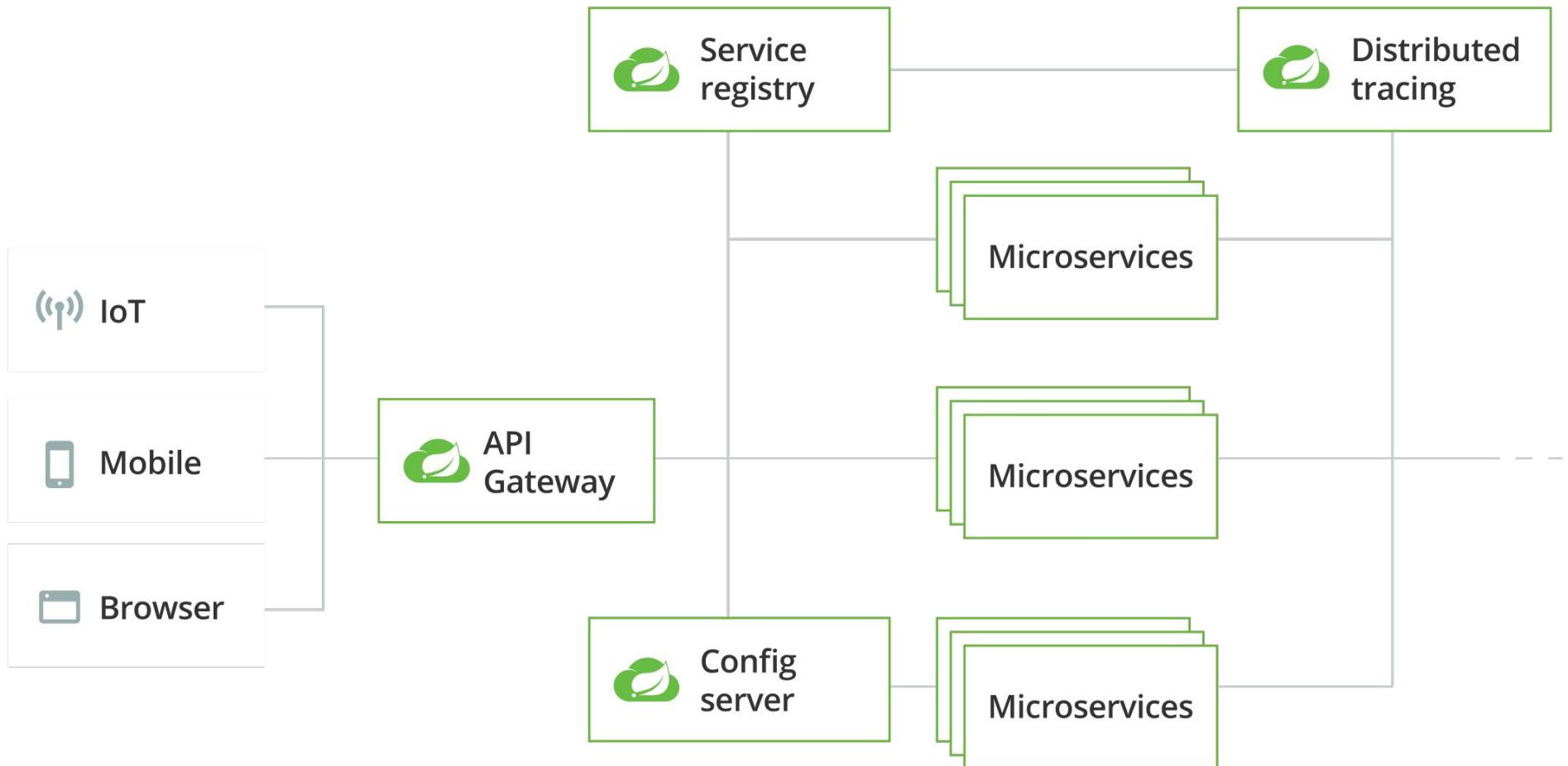
ИТМО BT

Spring Cloud

- Инструментарий на базе Spring для построения распределённых 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.
 - Балансировка нагрузки.



Архитектура Spring Cloud





Service Registry

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

```
@Autowired  
private DiscoveryClient discoveryClient;  
  
 @RequestMapping("/service-instances/{applicationName}")  
 public List<ServiceInstance> serviceInstancesByApplicationName(  
     @PathVariable String applicationName) {  
     return this.discoveryClient.getInstances(applicationName);  
 }
```



Конфигурация Service Registry

- “Снаружи” -- Maven / Gradle.
- “Изнутри” -- аннотации:

```
package com.example.serviceregistrationanddiscoveryservice;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;  
  
@EnableEurekaServer  
@SpringBootApplication  
public class ServiceRegistrationAndDiscoveryServiceApplication {  
  
    public static void main(String[] args) {  
  
        SpringApplication.run(ServiceRegistrationAndDiscoveryServiceApplication.class,  
        args);  
    }  
}
```

COPY



Использование Service Registry

ИТМО ВТ

```
@EnableDiscoveryClient
@SpringBootApplication
public class ServiceRegistrationAndDiscoveryClientApplication {

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

@RestController
class ServiceInstanceRestController {

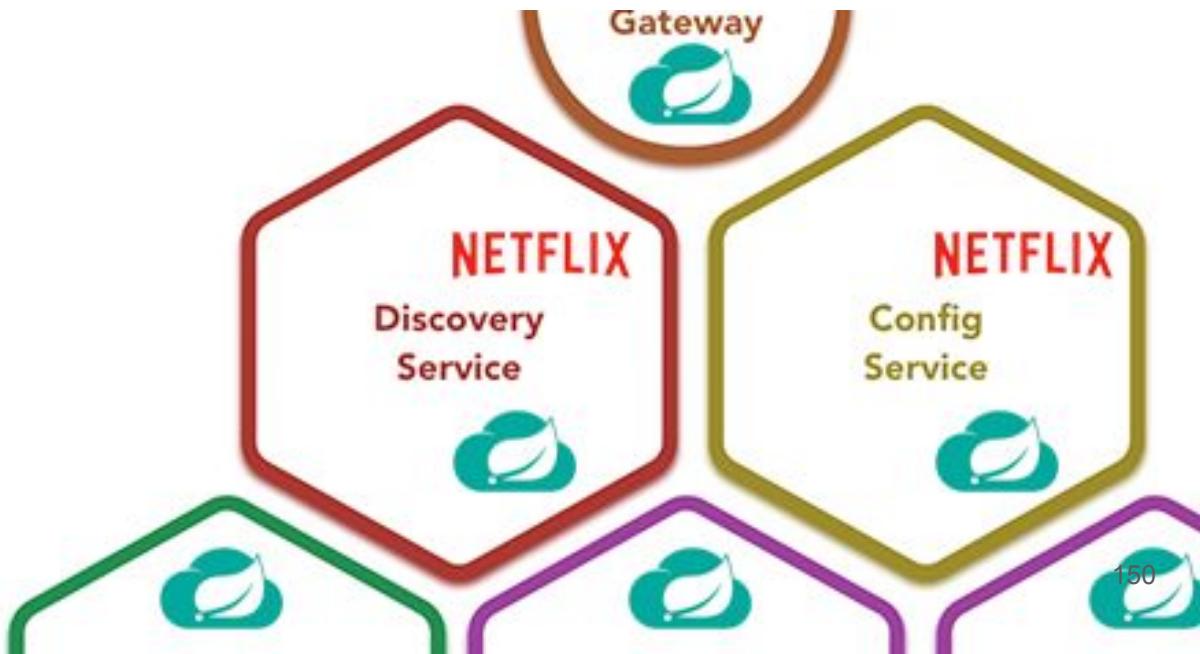
    @Autowired
    private DiscoveryClient discoveryClient;

    @RequestMapping("/service-instances/{applicationName}")
    public List<ServiceInstance> serviceInstancesByApplicationName(
            @PathVariable String applicationName) {
        return this.discoveryClient.getInstances(applicationName);
    }
}
```



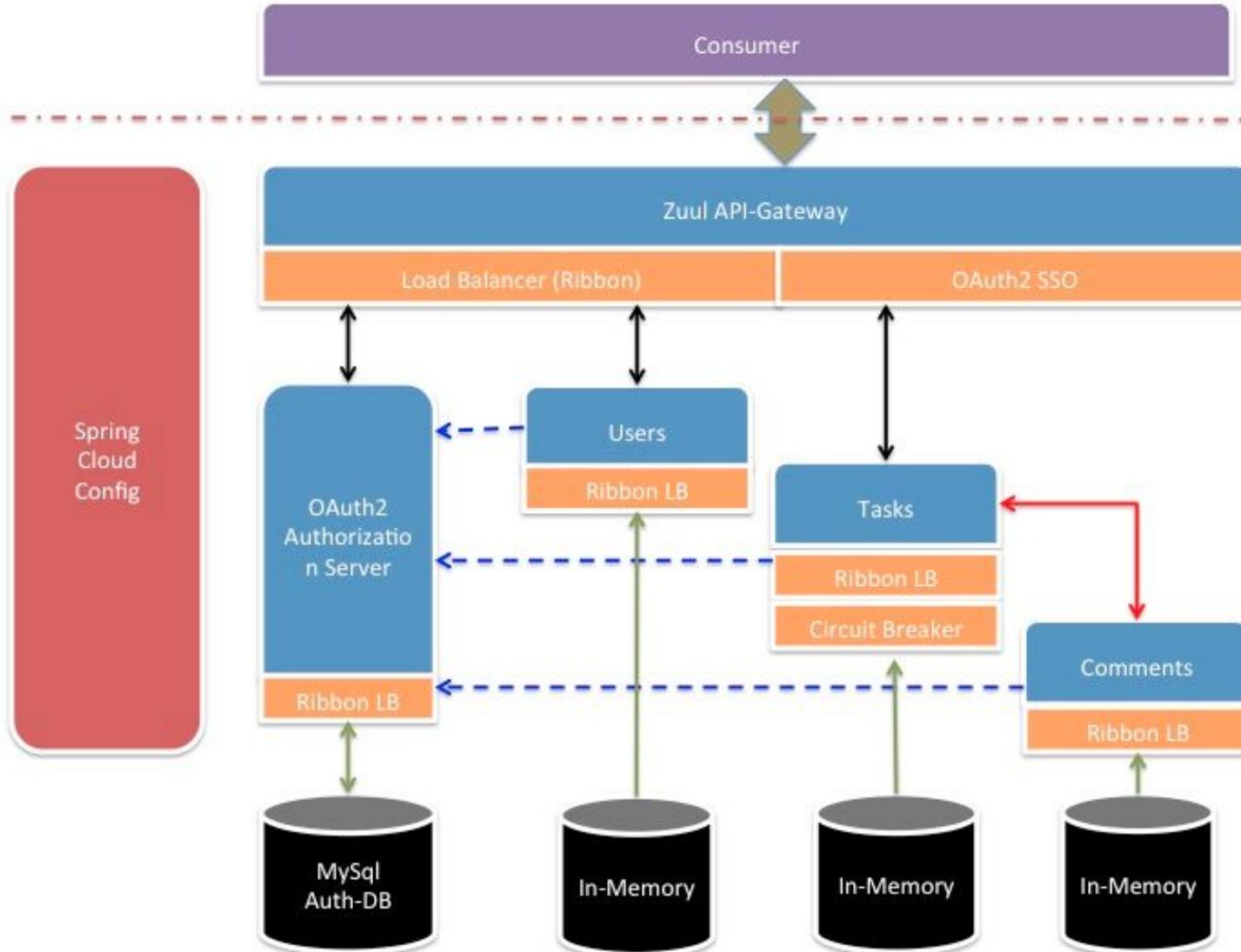
Spring Cloud Netflix

- Бандл для построения SOA-систем на базе Spring Cloud и стека Netflix OSS:
 - Eureka Server для SD.
 - Балансировщик нагрузки Ribbon.
 - Hystrix Circuit Breaker.





Архитектура Spring Cloud Netflix



10. Микросервисная архитектура

Общие понятия

- **Микросервисная архитектура** — вариант СОА, в котором приложение строится из [насколько это возможно] небольших, слабо связанных и легко изменяемых модулей — **микросервисов**.
- Идеология, а не стандарт или спецификация.
- Получила распространение с середины 2010-х гг.
- Хорошо “ложится” на методики гибкой разработки (agile-like) и практики DevOps.

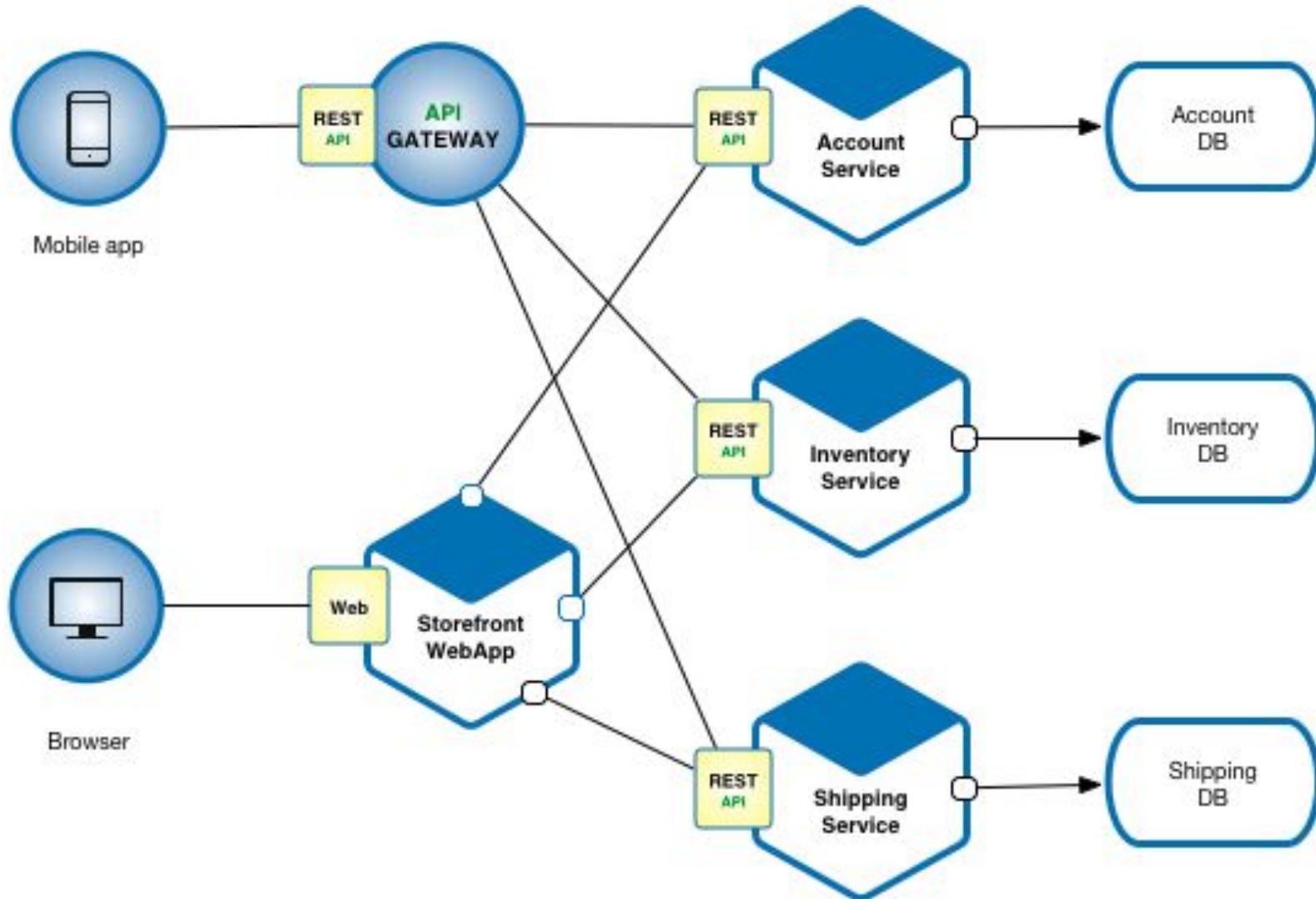


Особенности MSA

- 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.



Пример системы на базе MSA



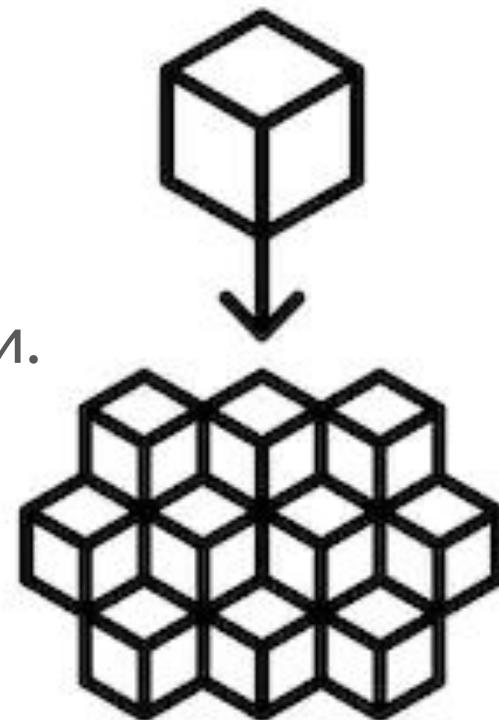
Свойства микросервиса (1)

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.



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.



Что значит “небольшой”?

- Один сервис может развивать одна команда не более чем из дюжины человек.
- Команда из полудюжины человек может развивать полдюжины сервисов.
- Контекст (не только бизнеса, но и разработки) одного сервиса помещается в голове одного человека.
- Один сервис может быть полностью переписан одной командой за одну Agile-итерацию.

Что значит “независимый”?

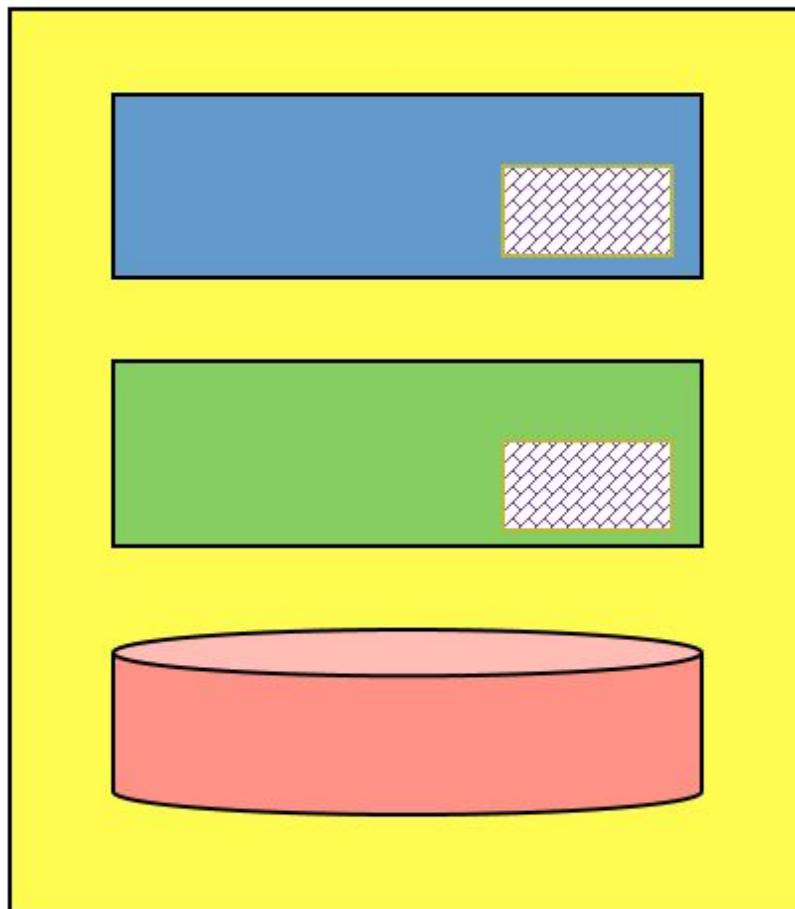
- Использует свой контекст.
- Не использует [общие] библиотеки.
- Использует своё хранилище данных (!).
- Следствие независимости -- при большом числе сервисов требуются продвинутые инструменты CI и CD.



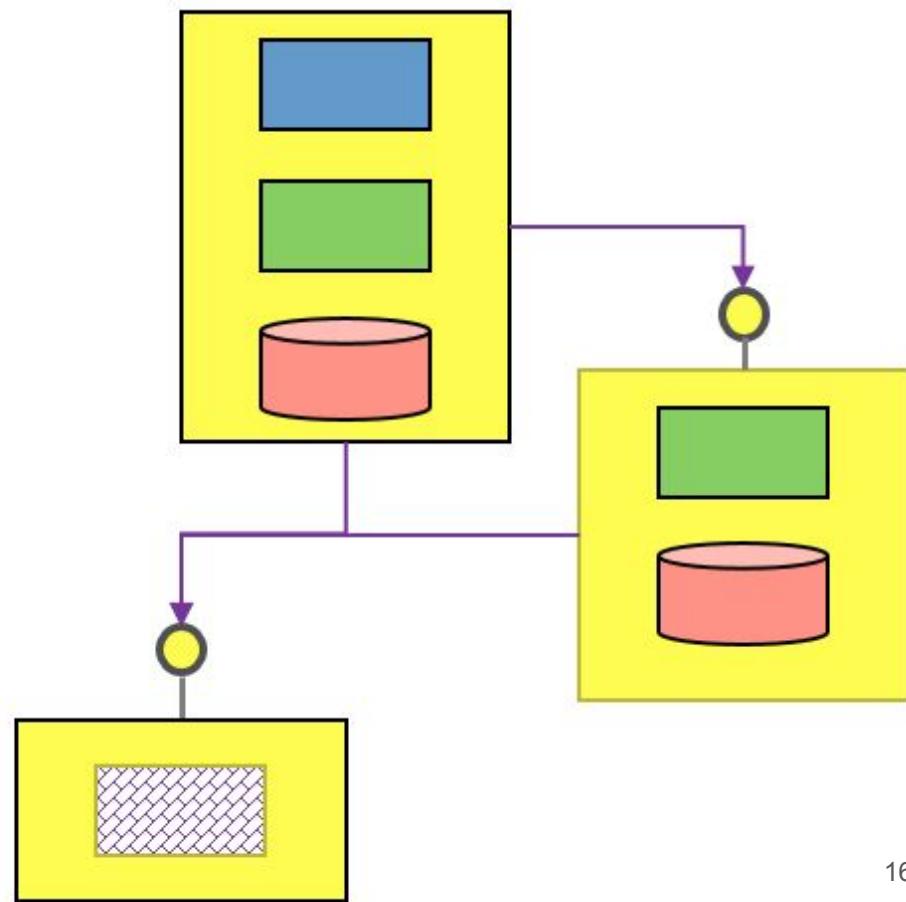


Замена библиотек на API

Монолит с Библиотекой



Микросервисы



Smart Endpoints & Dumb Pipes

- Для интеграции используются простые текстовые протоколы (обычно REST на HTTP).
- Допустимо использование бинарных протоколов (например, RMI).
- Использование синхронных вызовов не приветствуется.
- Практически все вызовы -- удалённые, вследствие чего частые взаимные вызовы сервисов вызывают "просадку" быстродействия.



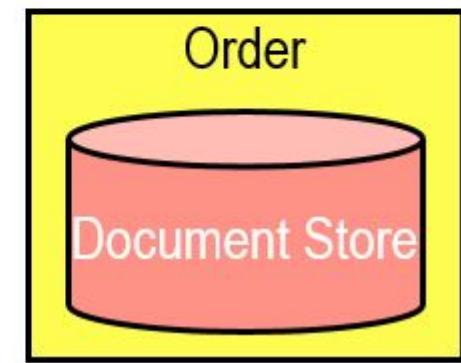
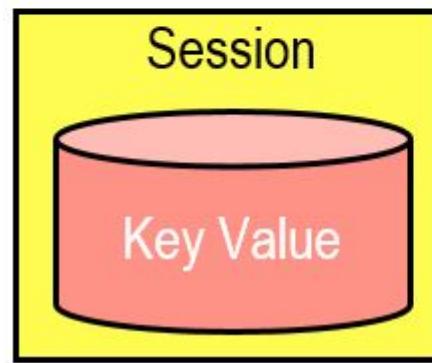
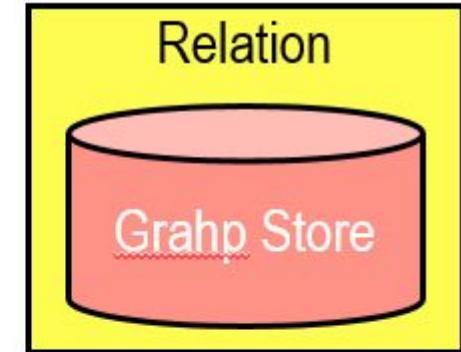
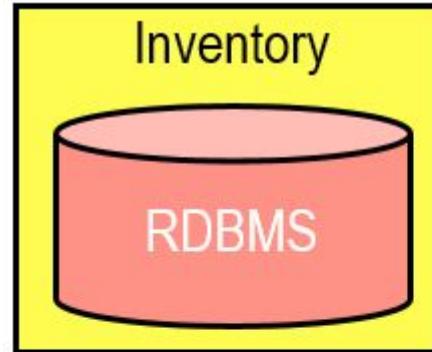
Design For Failure

- Есть ряд проблемных мест:
 - Ненадёжность сети (а практически все вызовы -- удалённые!).
 - Сложность реализации событийной архитектуры.
- Их очень сложно и дорого устраниТЬ, вследствие чего:
 - Наличие некритичных ошибок при работе -- нормальная ситуация.
 - Повышаются требования к инфраструктуре, позволяющей оперативно выявлять и исправлять проблемы.



Децентрализация данных

- Каждый сервис использует своё хранилище.
- Возникает дополнительная задача синхронизации данных.
- Используется модель Eventual Consistency.
- Часто приходится усложнять архитектуру.



Недостатки подхода (1)

- Информационные барьеры между сервисами.
- Сложности с организацией взаимных вызовов сервисов из-за задержек вызовов по сети.
- Сложные развёртывание и тестирование.
- Не всегда возможно эффективно разделить функциональность между микросервисами.
- Большое число простых сервисов могут решать задачу менее эффективно, чем небольшое число сложных.

Недостатки подхода (2)

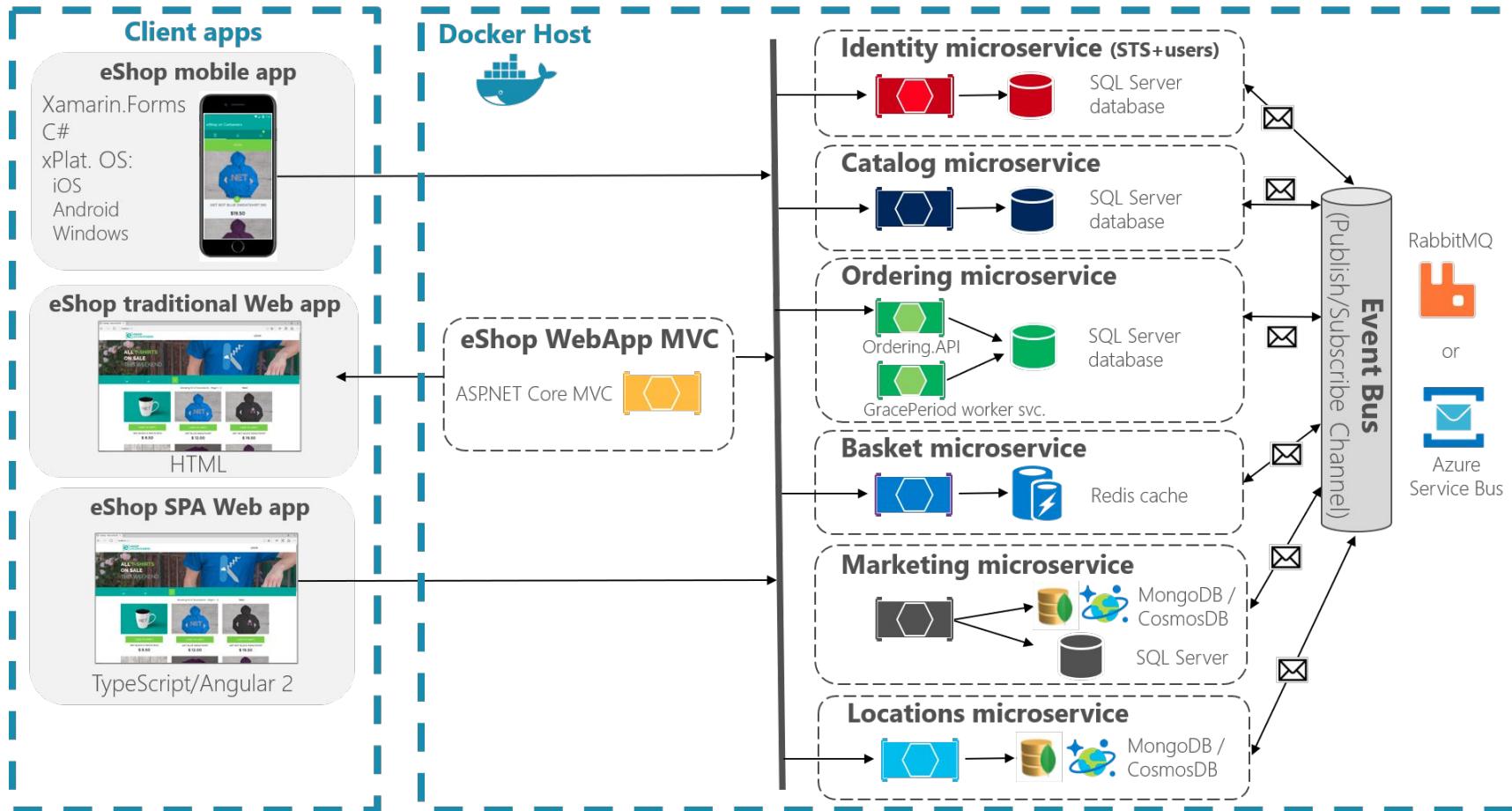
- Сложности с обеспечением целостности данных и транзакционных механизмов.
- Из-за “зоопарка” технологий могут предъявляться повышенные требования к квалификации администраторов / девопсов.
- Протокол HTTP не очень эффективен для организации взаимодействия между сервисами (т.к. он для этого не предназначен).
- Размытость понятия микросервиса, повышенные требования к квалификации программиста.

11. Инфраструктура для микросервисов

Контейнеризация

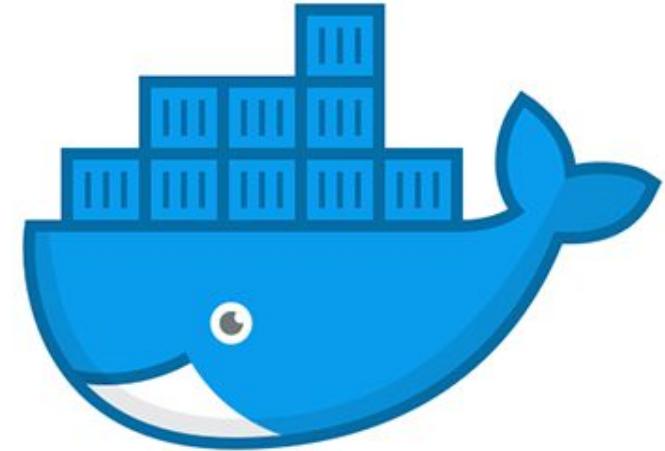
eShopOnContainers reference application

(Development environment architecture)

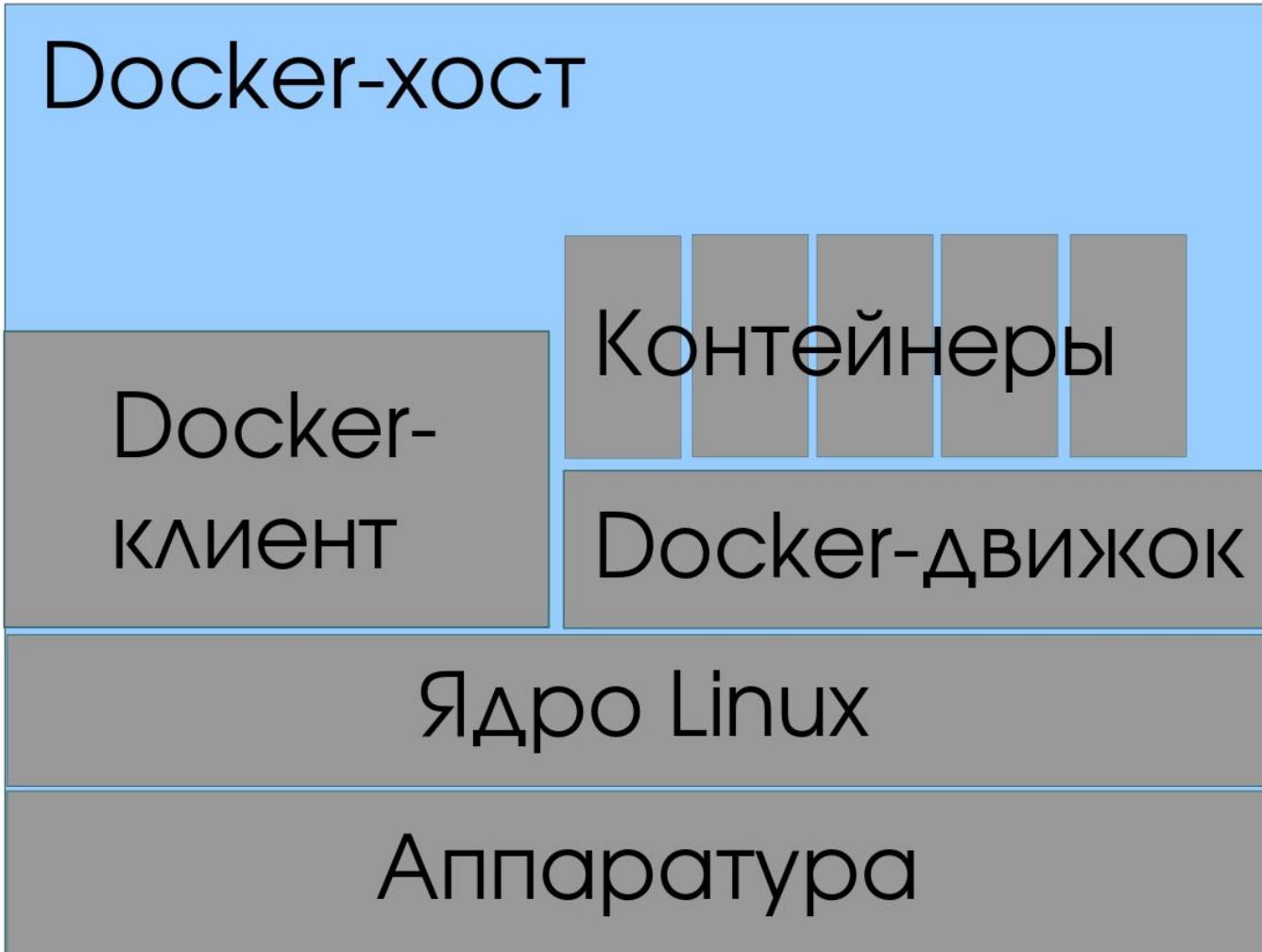


Docker

- ПО для автоматизации развёртывания и управления приложениями в средах с поддержкой контейнеризации.
- Открытое ПО (Apache 2.0), написано на Go.
- Использует возможности контейнеризации, предоставляемые ядром Linux (libcontainer).
- Работает на Linux, Windows, macOS.
- Есть репозитории готовых контейнеров (Docker Hub).

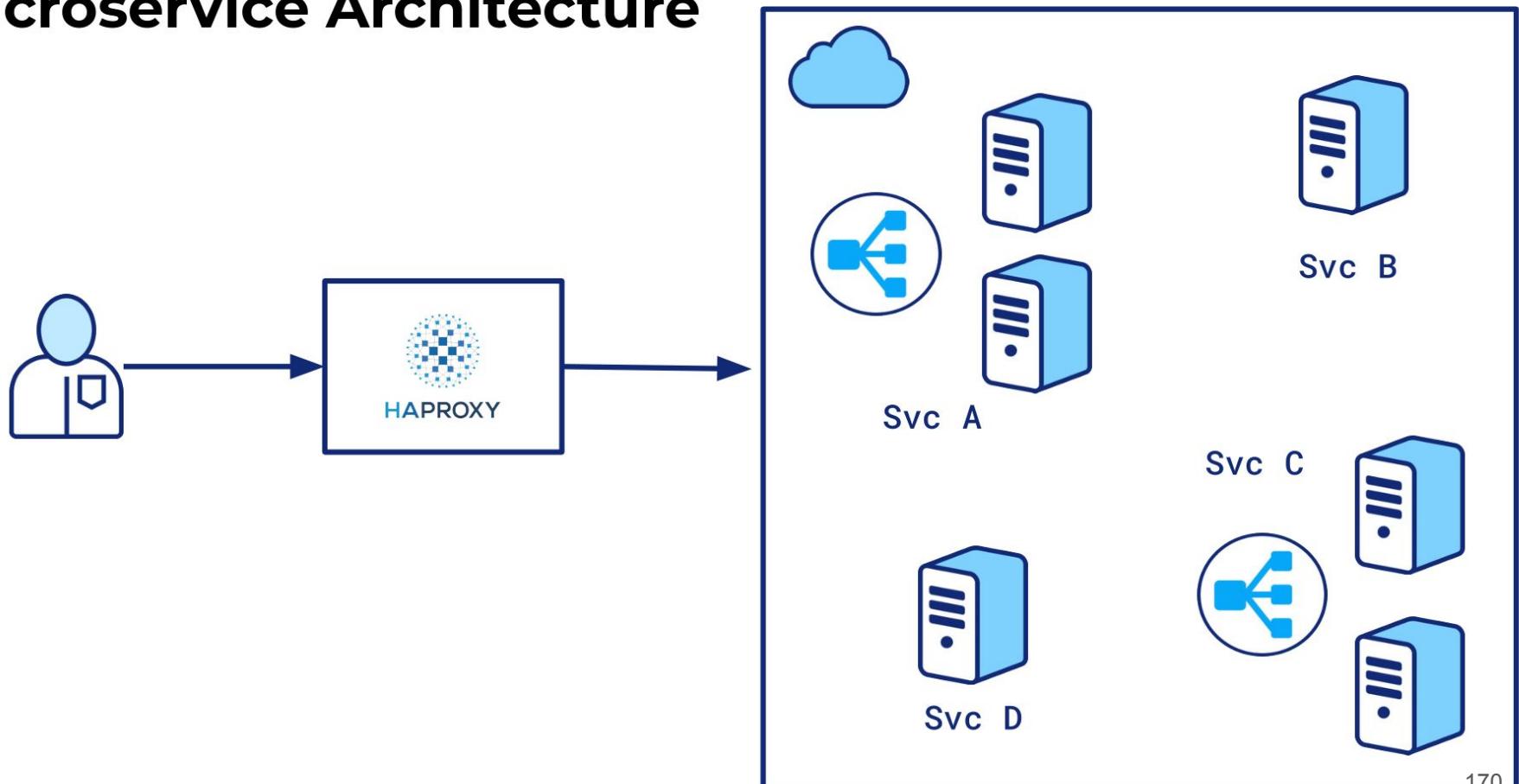


Архитектура Docker



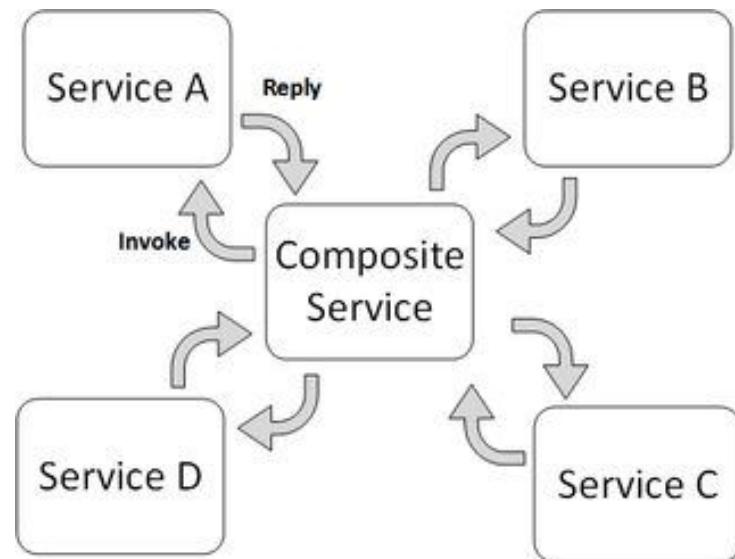
Балансировка нагрузки

Microservice Architecture



Оркестровка

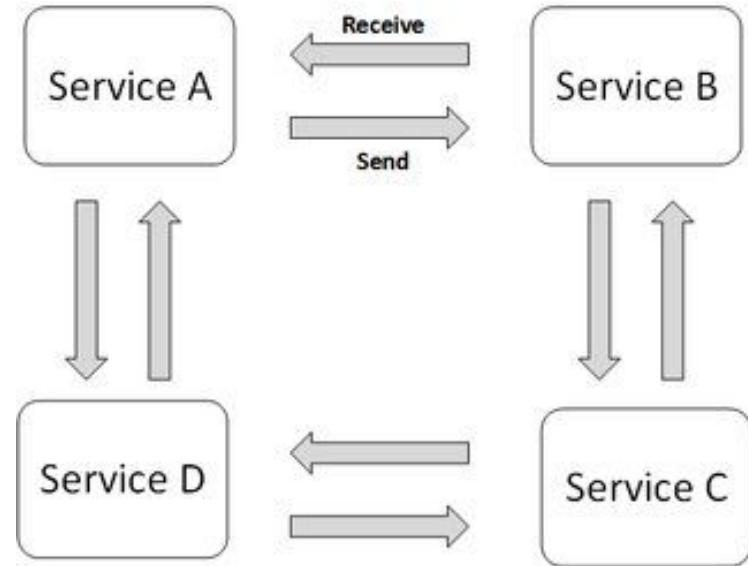
- Добавляется подсистема, которая координирует взаимодействие между сервисами ("дирижёр" -- *Orchestrator*).
- Есть готовые решения для оркестровки микросервисов:
 - Kubernetes;
 - Azure Kubernetes Service (ACS);
 - Apache Mesos;
 - Amazon Elastic Container Service (ECS).





Хореография

- Описывает взаимодействие между несколькими сервисами.
- В отличие от оркестровки, “центрального” сервиса нет.
- Административная логика “размазана” по сервисам.
- Более “честно” ложится на МСА, но сложнее в реализации.
- Могут использоваться брокеры событий (Event Broker).



Kubernetes

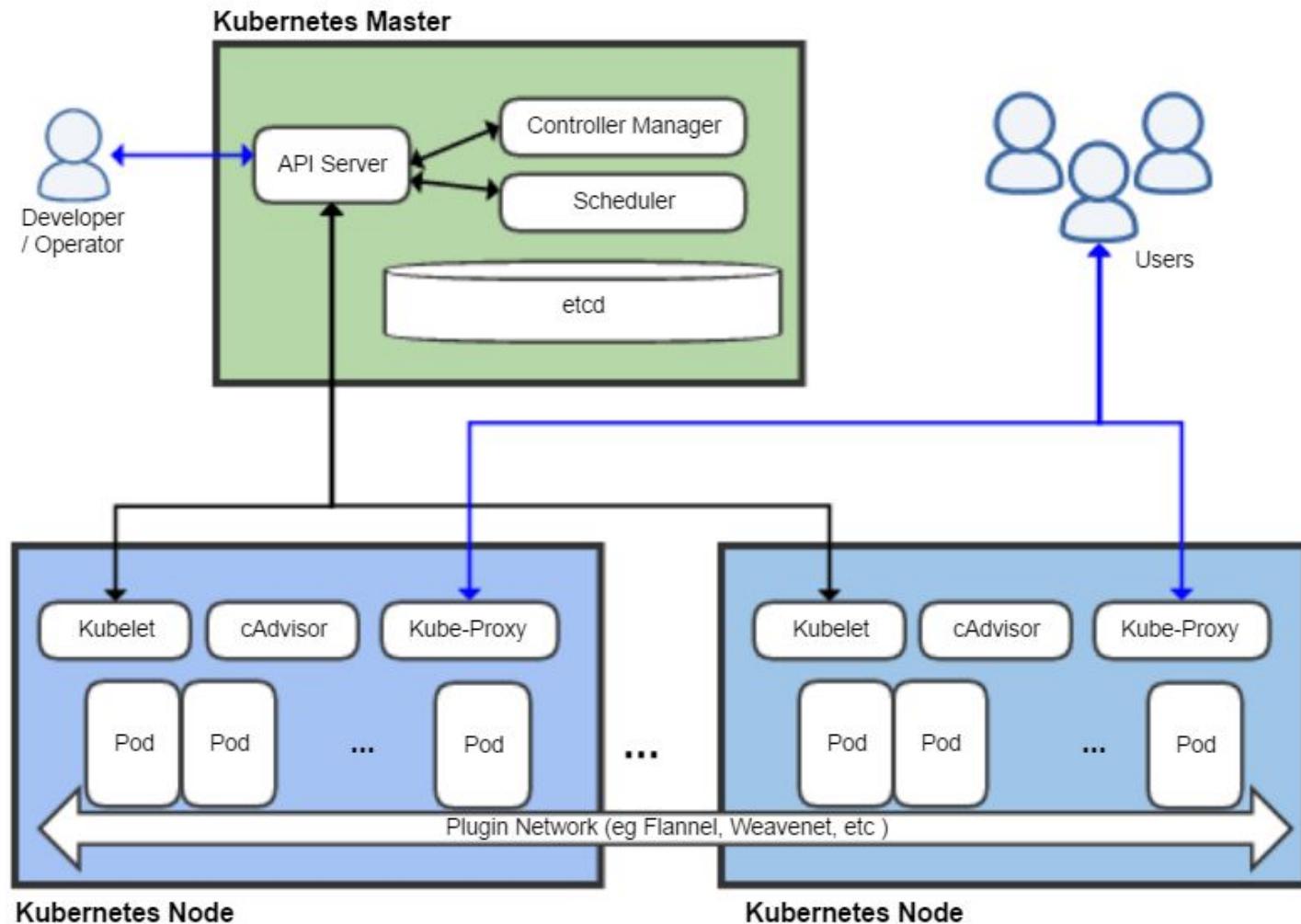
- Открытое ПО для автоматизации развёртывания, масштабирования и управления контейнеризованными приложениями.
- Разработан Google, написан на языке Go, первая версия вышла в 2014 г.
- Поддерживает все основные технологии контейнеризации, в т.ч. и Docker.
- Умеет в Service Discovery.



kubernetes



Архитектура Kubernetes (1)



Архитектура Kubernetes (2)

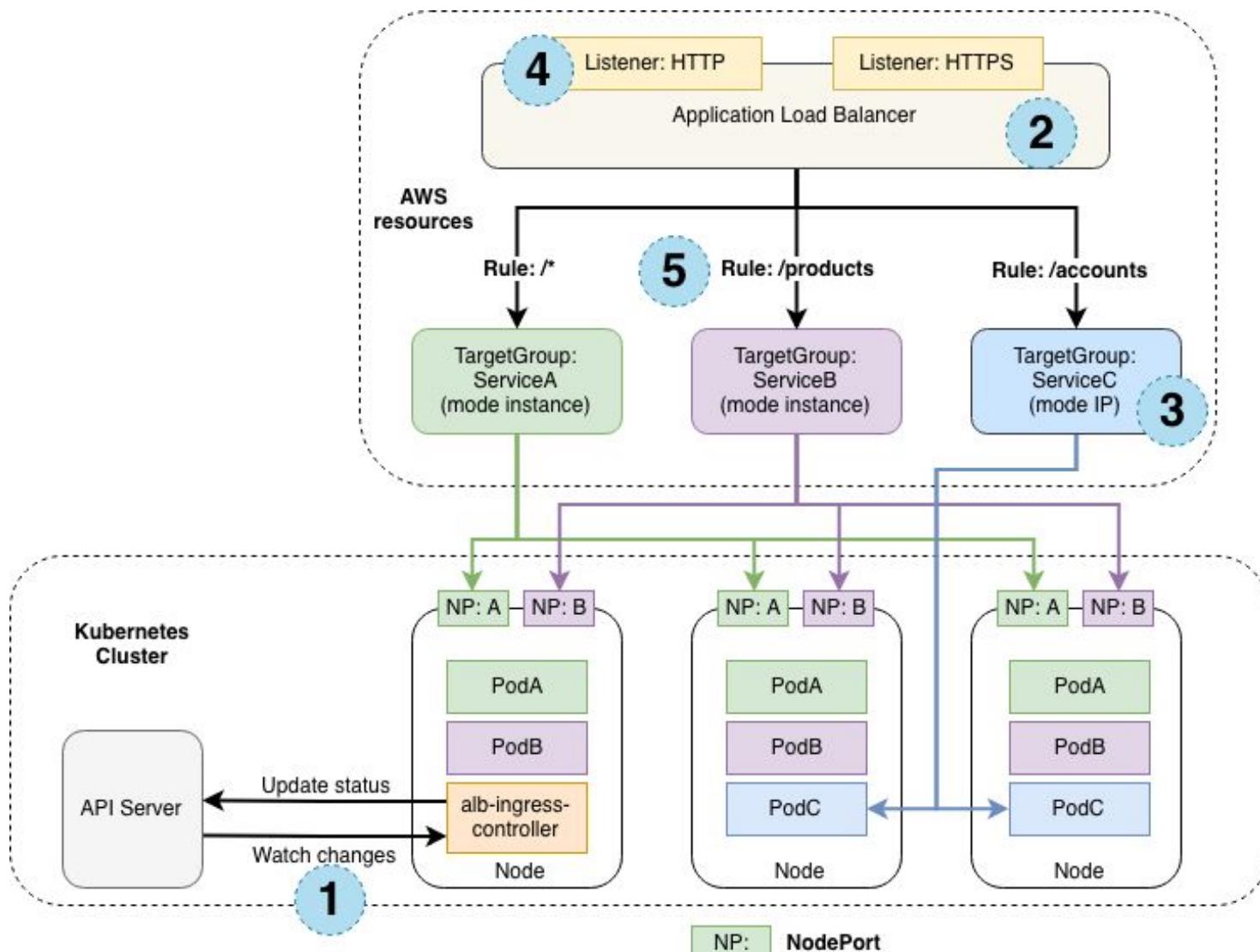
- **Узел (*node*)** — отдельная физическая или виртуальная машина, на которой развернуты и выполняются контейнеры приложений.
- Каждый узел в кластере содержит сервисы для запуска приложений в контейнерах, а также компоненты, предназначенные для управления узлом.
- **Под (*pod* — «стручок, кокон»)** — один или несколько контейнеров, которым гарантирован запуск на одном узле, обеспечивается разделение ресурсов, межпроцессное взаимодействие и предоставляется уникальный IP-адрес.

Архитектура Kubernetes (3)

- **Том (volume)** — общий ресурс хранения для совместного использования из контейнеров, развёрнутых в пределах одного пода.
- Все объекты управления (узлы, поды, контейнеры) в Kubernetes помечаются **метками (label)**.
- **Селекторы меток (label selector)** — это запросы, которые позволяют получить ссылку на объекты, соответствующие какой-то из меток
- **Сервисом** в Kubernetes называют совокупность логически связанных наборов подов и политик доступа к ним.



Kubernetes: пример кластера



12. Микросервисы на Java

Общие сведения

- Т.к. микросервисы -- те же веб-сервисы (только маленькие!), можно использовать те же самые стеки технологий.
- Специальные библиотеки и фреймворки существуют, но используются редко.
- Независимо от выбранного стека, есть общие подходы и паттерны для решения типовых задач.
- Можно применять решения для оркестровки / хореографии и контейнеризации.

Проблема коммуникации

- Т.к. сервисы маленькие, возникает проблема эффективной коммуникации между ними.
- Проблема актуальна, когда сложная функция реализуется несколькими простыми сервисами.
- Решение делится на две подзадачи:
 - Синхронная коммуникация -- когда ответ нужен “сразу же”.
 - Асинхронная коммуникация -- когда ответ формируется “в фоновом режиме”.

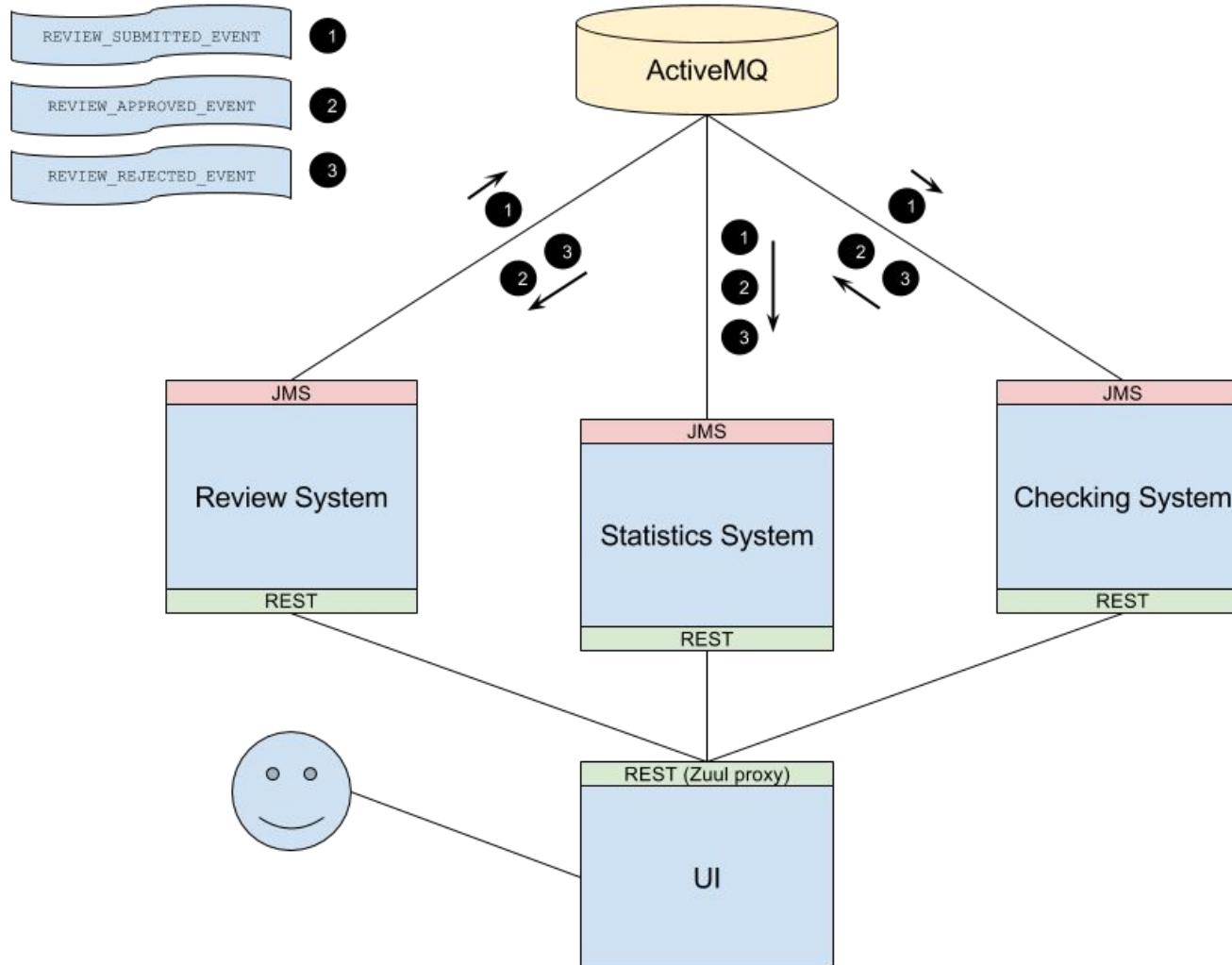
Синхронная коммуникация

- Снижает масштабируемость, без крайней необходимости использовать не рекомендуется.
- Обычно реализуется через REST@HTTP.
- Можно использовать любую библиотеку, способную формировать запросы HTTP, например:
 - `java.net.http.HttpClient`.
 - Apache HttpClient.
 - OkHttp.
 - JAX-RS.

Асинхронная коммуникация

- Реализуется с помощью обмена сообщениями между сервисами.
- Стандартные варианты -- JMS и AMQP.
- Для JMS нужно использовать брокер сообщений:
 - ActiveMQ.
 - RabbitMQ.
 - Kafka.

Пример: обмен сообщениями через ActiveMQ



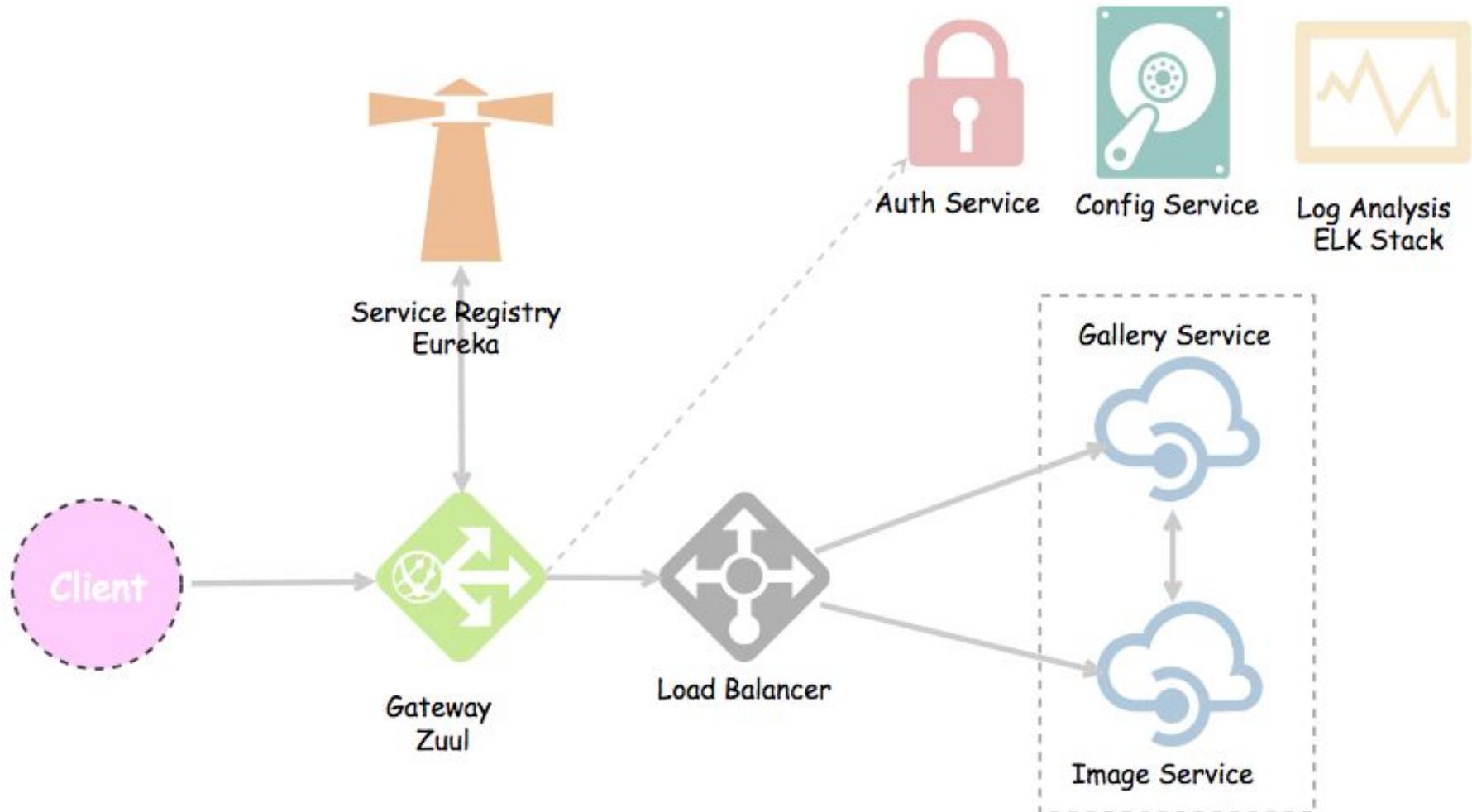
Микросервисы на Java EE

- 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).

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”.

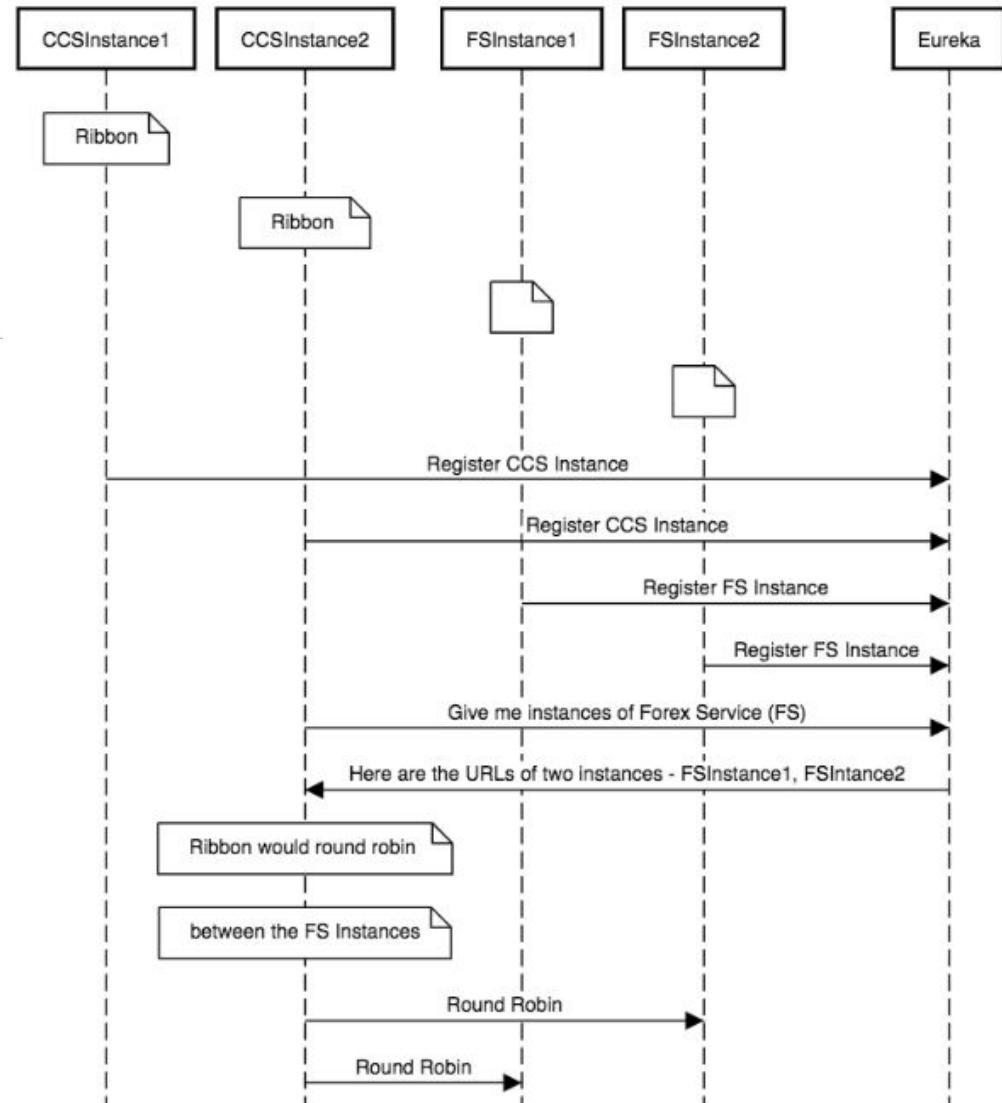
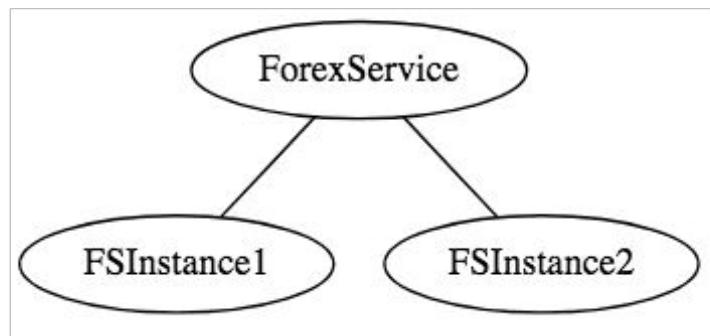
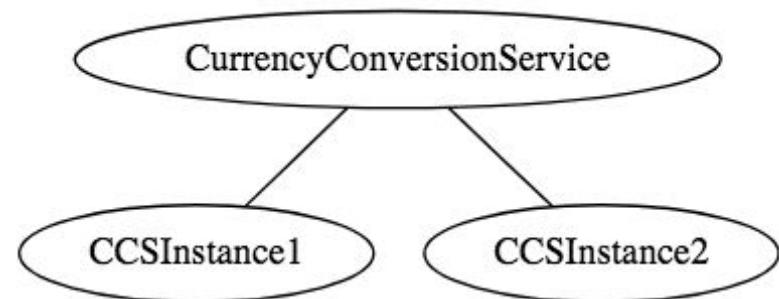
Пример архитектуры



Балансировка нагрузки и использование индекса



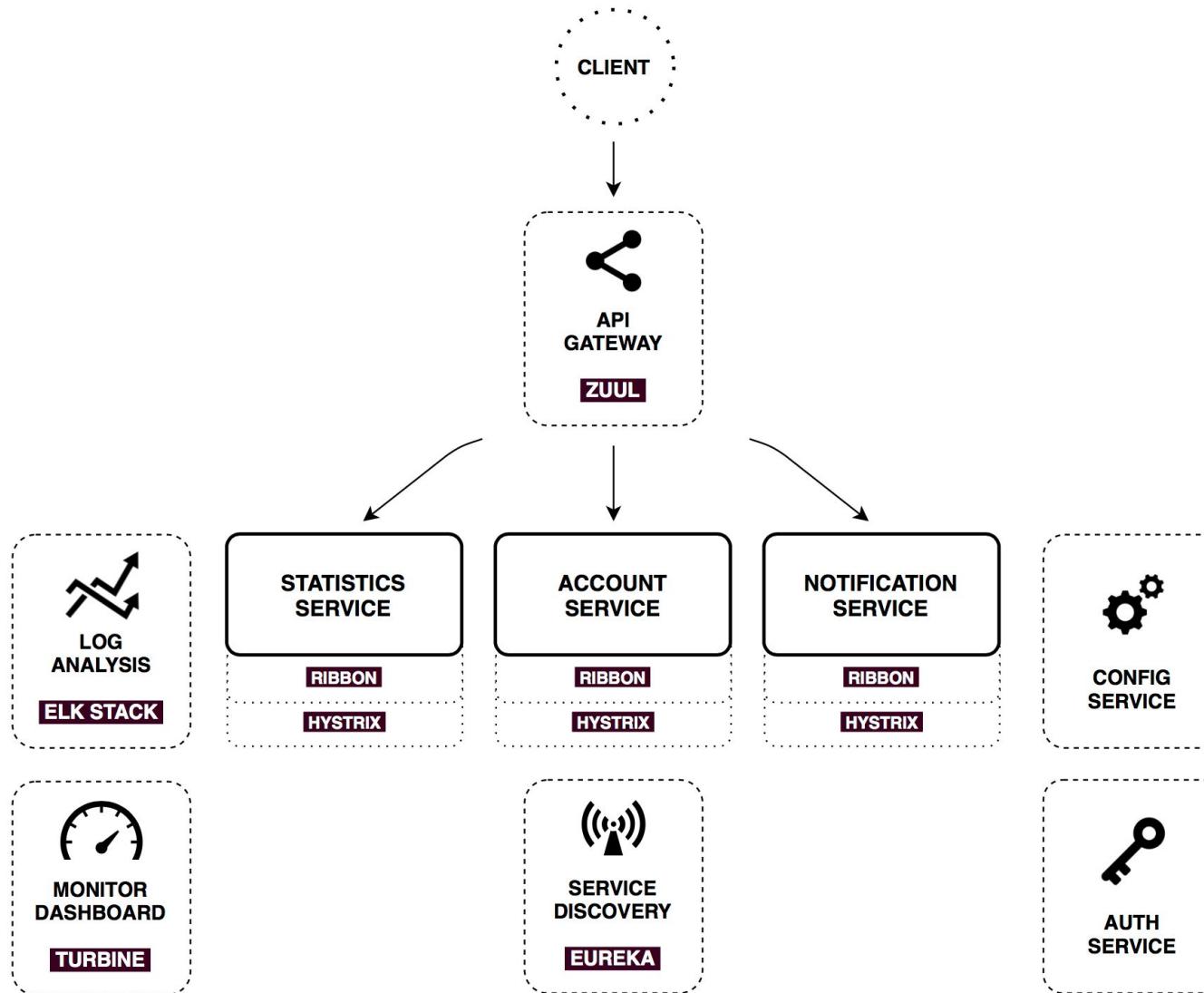
ИТМО ВТ



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.

Пример архитектуры

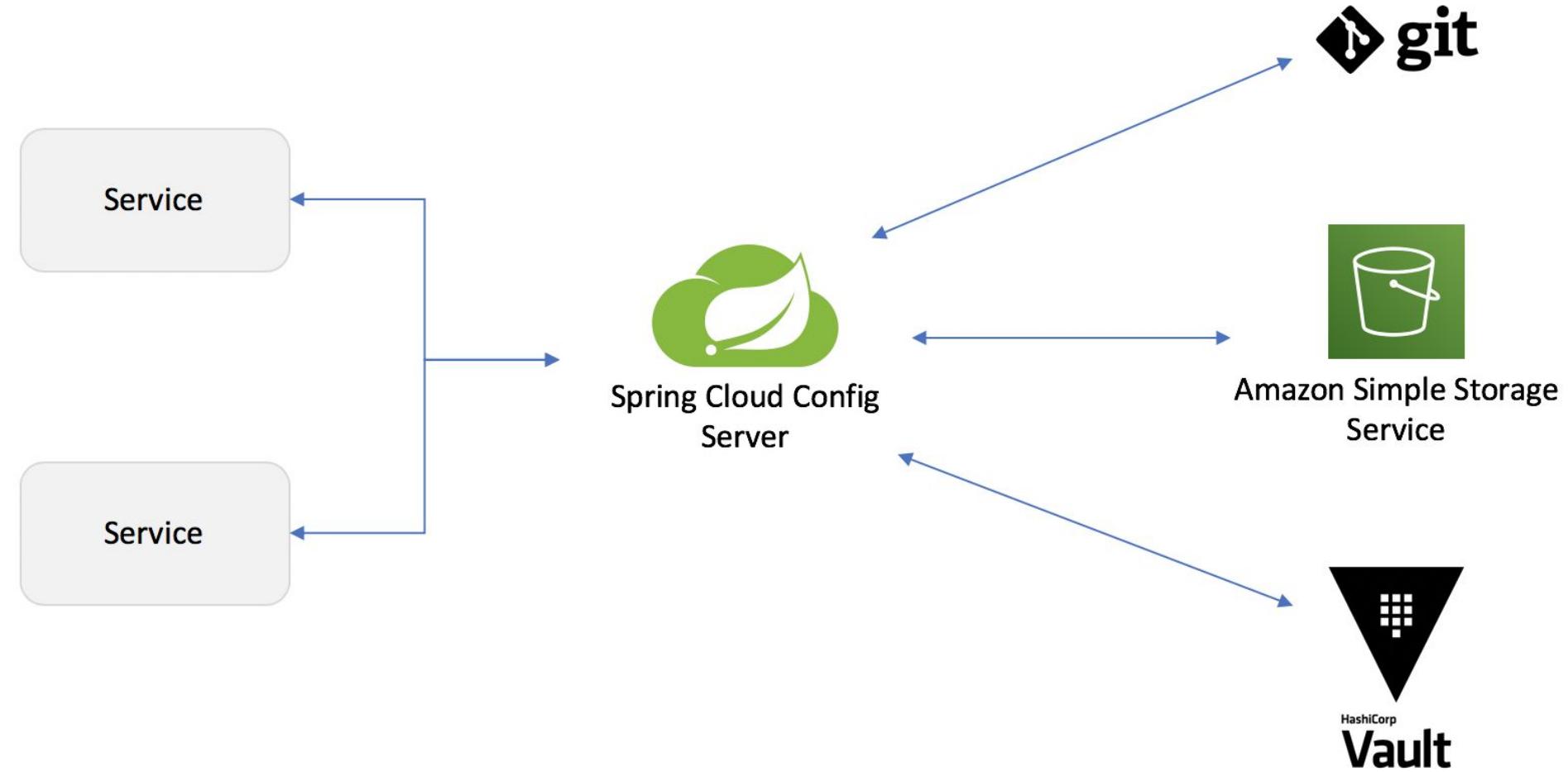




Spring Cloud Config Server

- Сервер для хранения конфигурации распределённого окружения.
- Конфигурация хранится в формате “ключ-значение”.
- Конфигурацию можно использовать в сервисах с помощью аннотаций.
- Сам сервер -- приложение на базе Spring Boot. Включается аннотацией `@EnableConfigServer`.
- В клиентах подключается аннотацией
`@EnableConfigClient`.
- Физически может хранить конфигурацию много где --
например, в git-репозитории.

Spring Cloud Config Server: пример использования

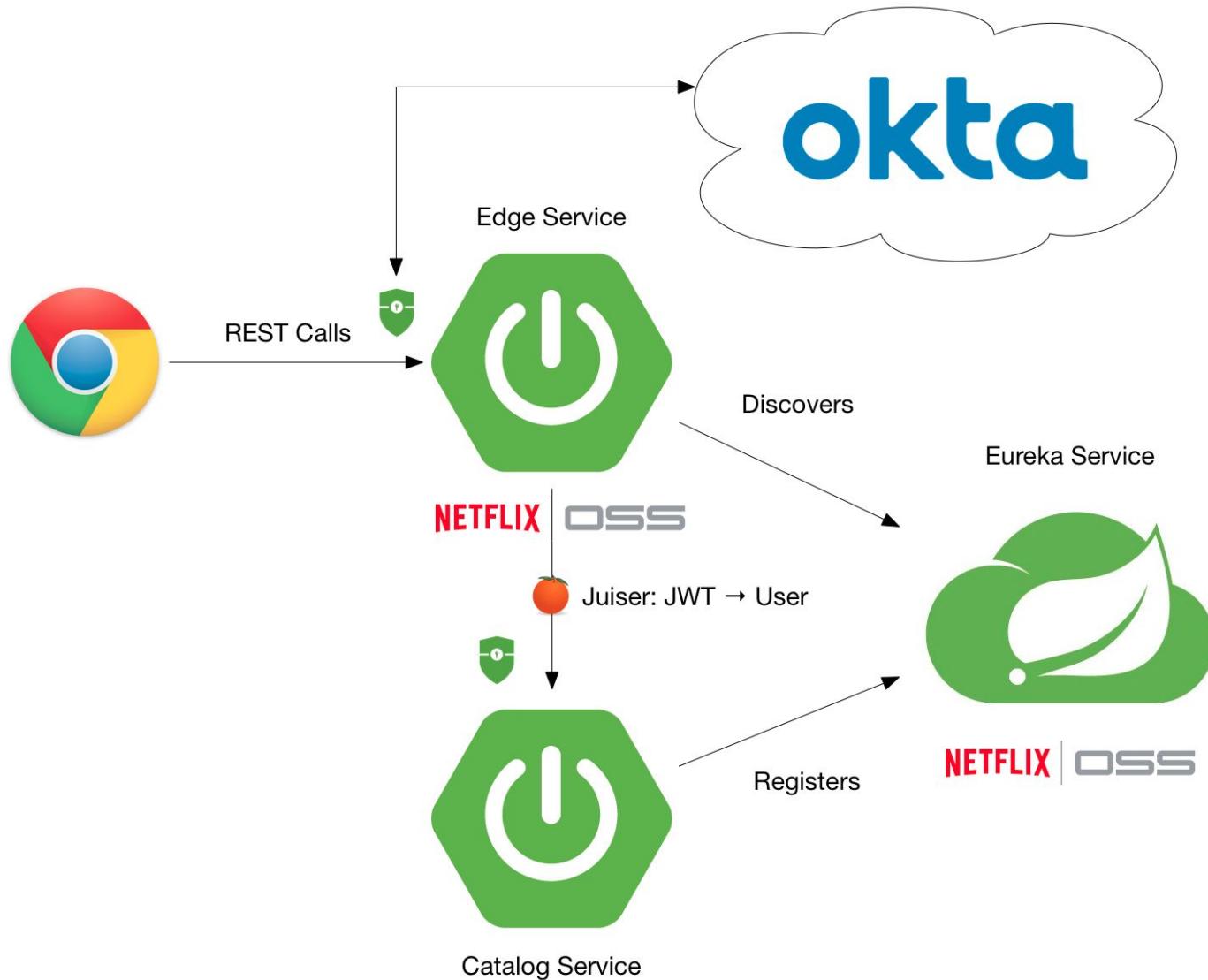




Auth Server

- “SSO” для микросервисов.
- Предоставляет возможности централизованной аутентификации и авторизации для всех сервисов приложения.
- Интегрирован в инфраструктуру Spring Security.
- Клиенты взаимодействуют с сервером по протоколу OAuth 2.0.
- Сам сервер -- приложение на базе Spring Boot. Включается аннотацией `@EnableAuthorizationServer`.

Auth Server: пример использования

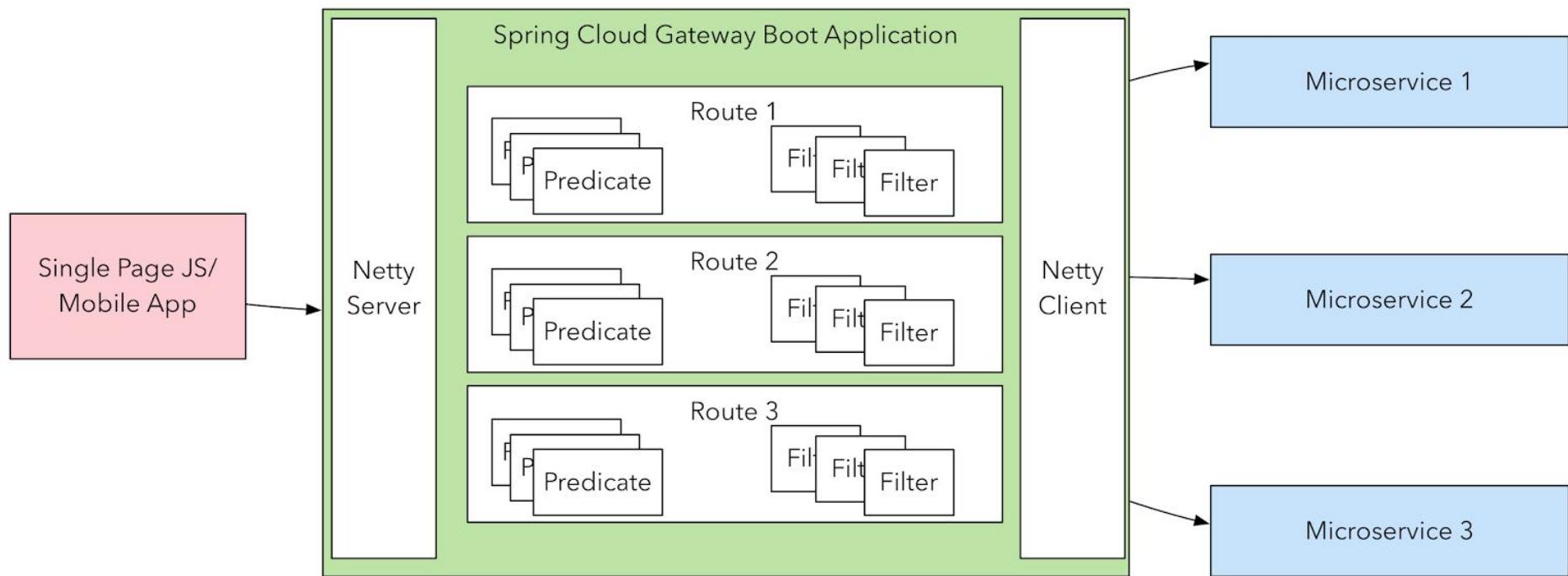


API Gateway

- Библиотека для построения “шлюзов” для доступа внешних клиентов к микросервисам.
- Построена на базе Spring Framework 5, Project Reactor и Spring Boot 2.0.
- Позволяет маршрутизировать запросы, используя любые их атрибуты.
- Интеграция “из коробки” с другими сервисами Spring Cloud – Circuit Breaker, DiscoveryClient.



API Gateway: пример использования





Service Discovery

...см. модуль 9...





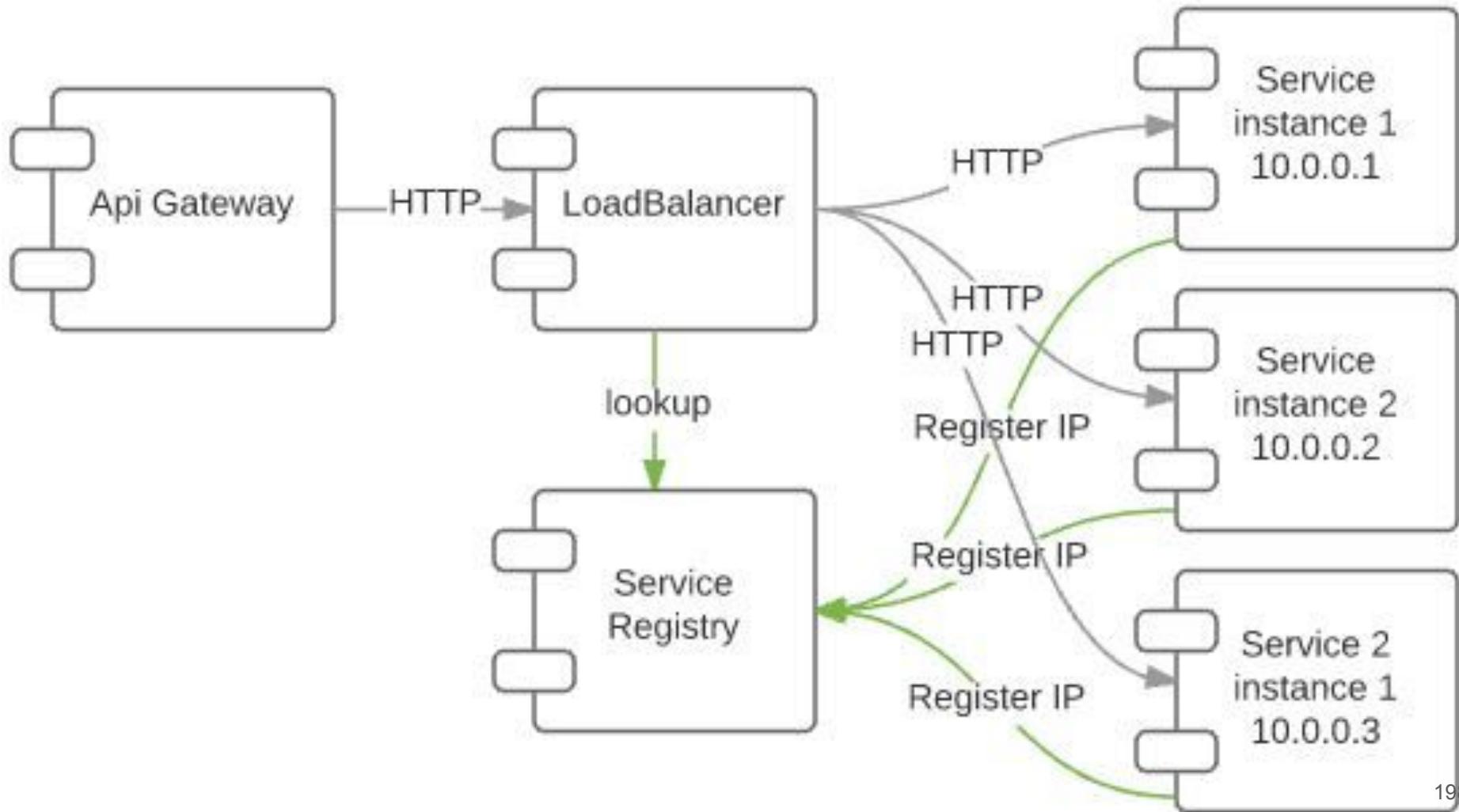
Load Balancer

- Балансирует нагрузку между процессами сервисов (ваш кэп!).
- Работает в “связке” с Service Registry.
- Можно использовать разные, “каноничный” вариант -- Ribbon из стека Netflix.
- Два алгоритма балансировки нагрузки:
 - На стороне клиента.
 - На стороне сервера.



ИТМО ВТ

Load Balancer: пример использования





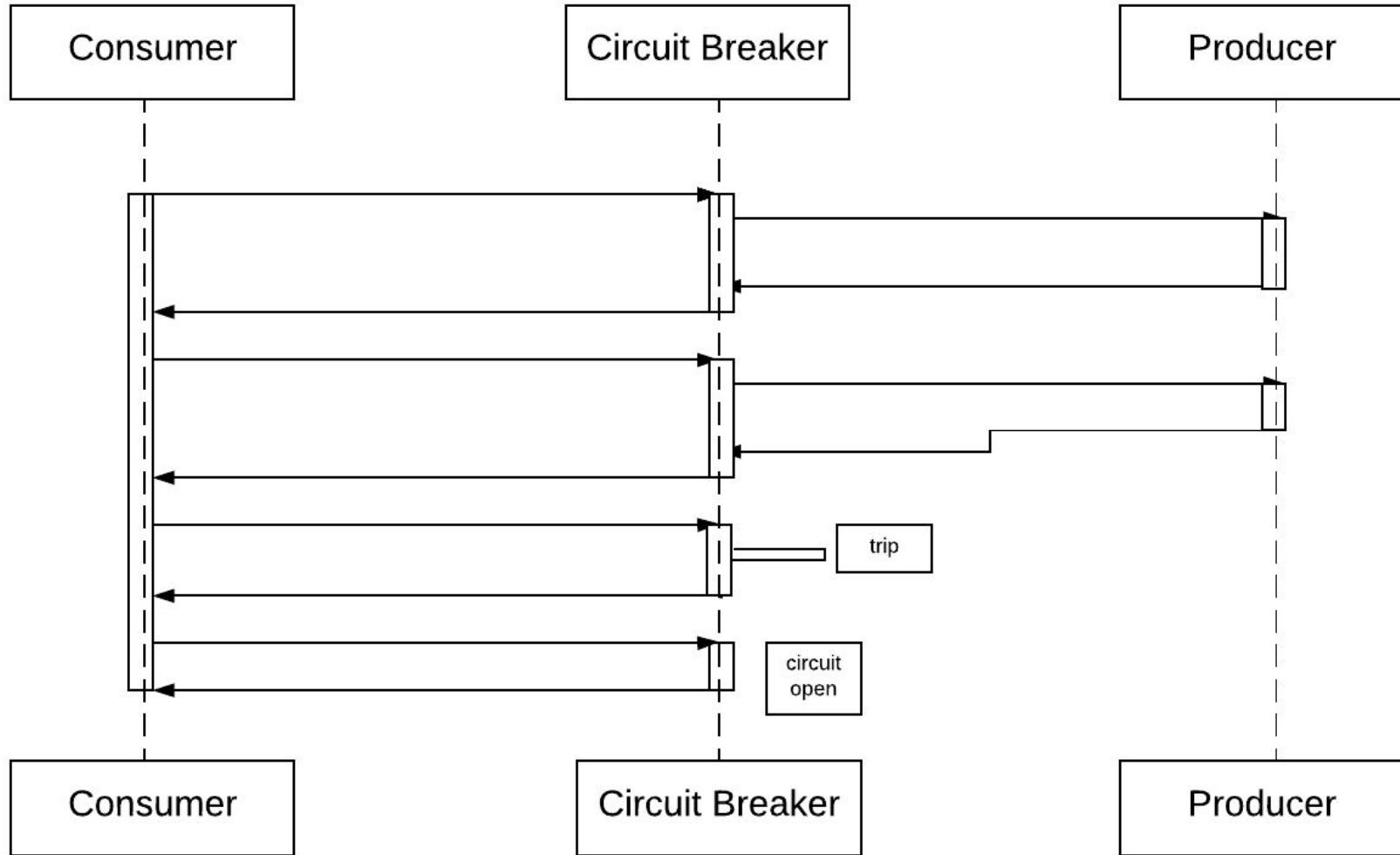
Circuit Breaker

- ПО, предотвращающее “заведомо обречённые” запросы к сервисам.
- “Каноничный” вариант -- **Hystrix** из стека **Netflix**.
- Конфигурируется аннотациями (пример для **Hystrix**):
 - `@EnableCircuitBreaker`.
 - `@EnableHystrixDashboard`.
 - `@HystrixCommand(fallbackMethod = "myFallbackMethod")`.



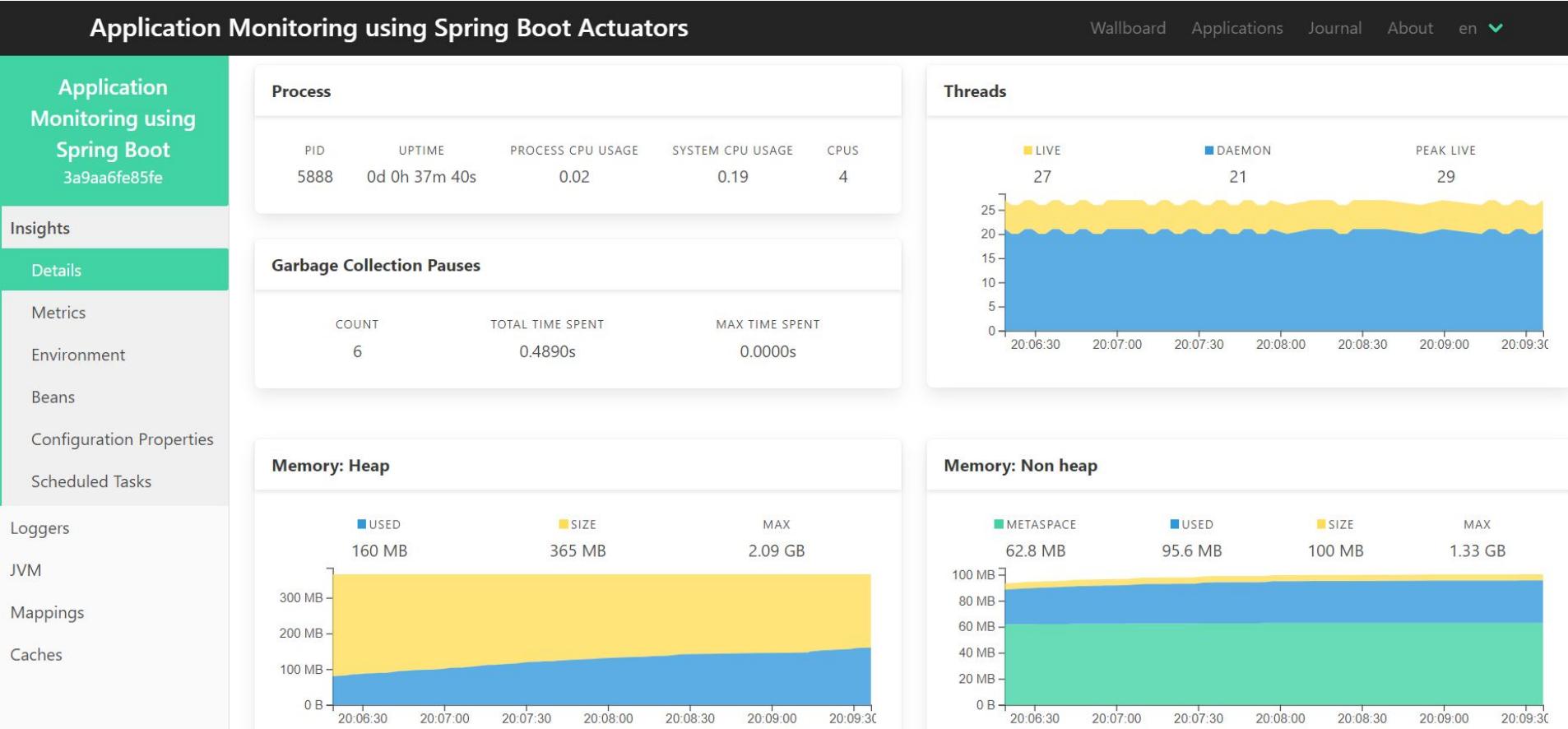
ИТМО ВТ

Circuit Breaker: пример использования





Панель мониторинга



Специальные решения для микросервисов

- Существуют.
- Используются достаточно редко.
- Примеры:
 - Quarcus.
 - Micronaut.
 - Eclipse Vert.x.
 - Helidon.

13. Веб-сервисы на базе SOAP

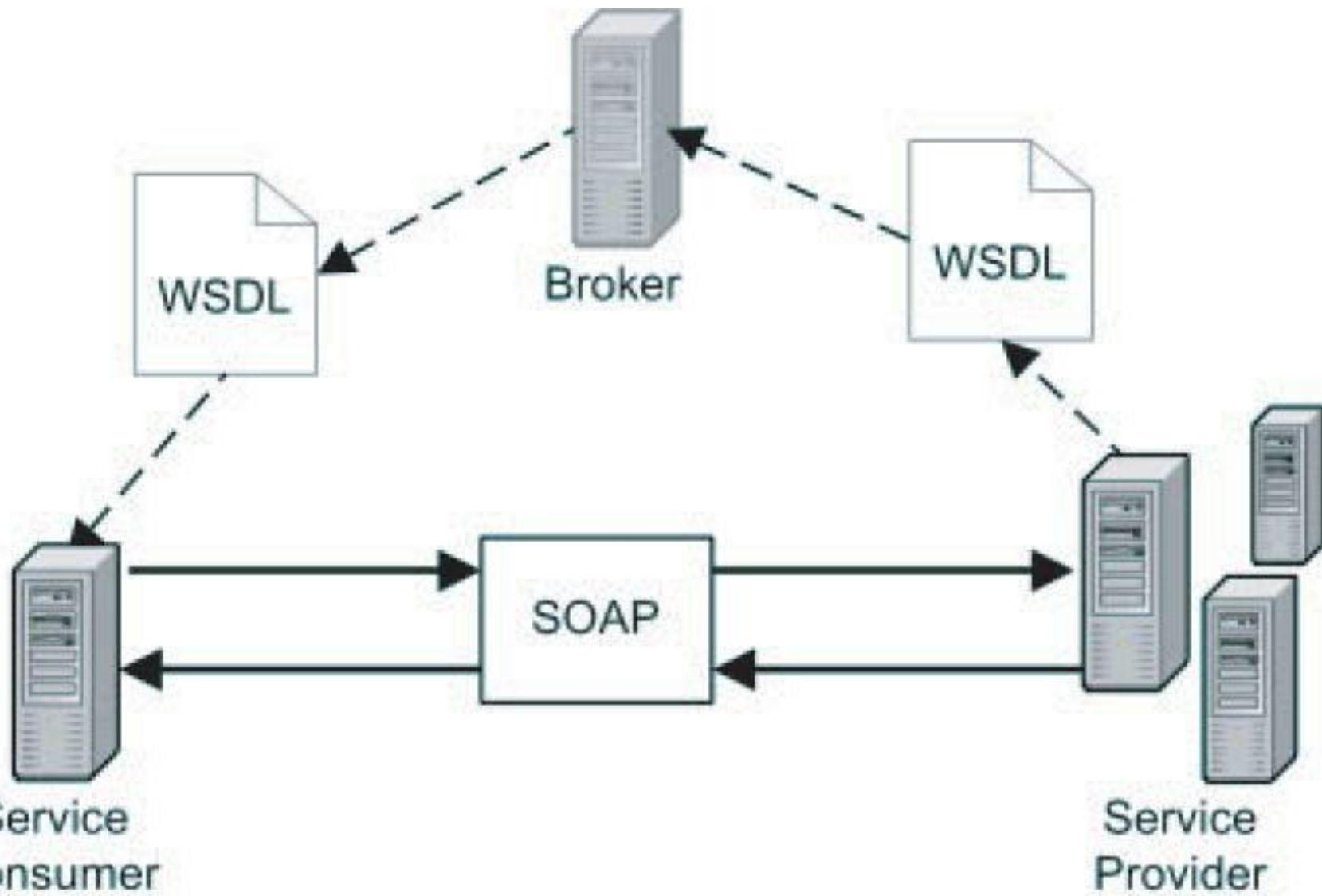
- Протокол для разработки веб-сервисов.
- Базируется на идеологии RPC.
- Стандартизирован W3C.
- Есть реализации “по умолчанию” для различных платформ.
- Предполагает использование инфраструктурного ПО – реестров и сервисных шин.





ИТМО ВТ

Инфраструктура SOAP



WSDL: Web Services Description Language

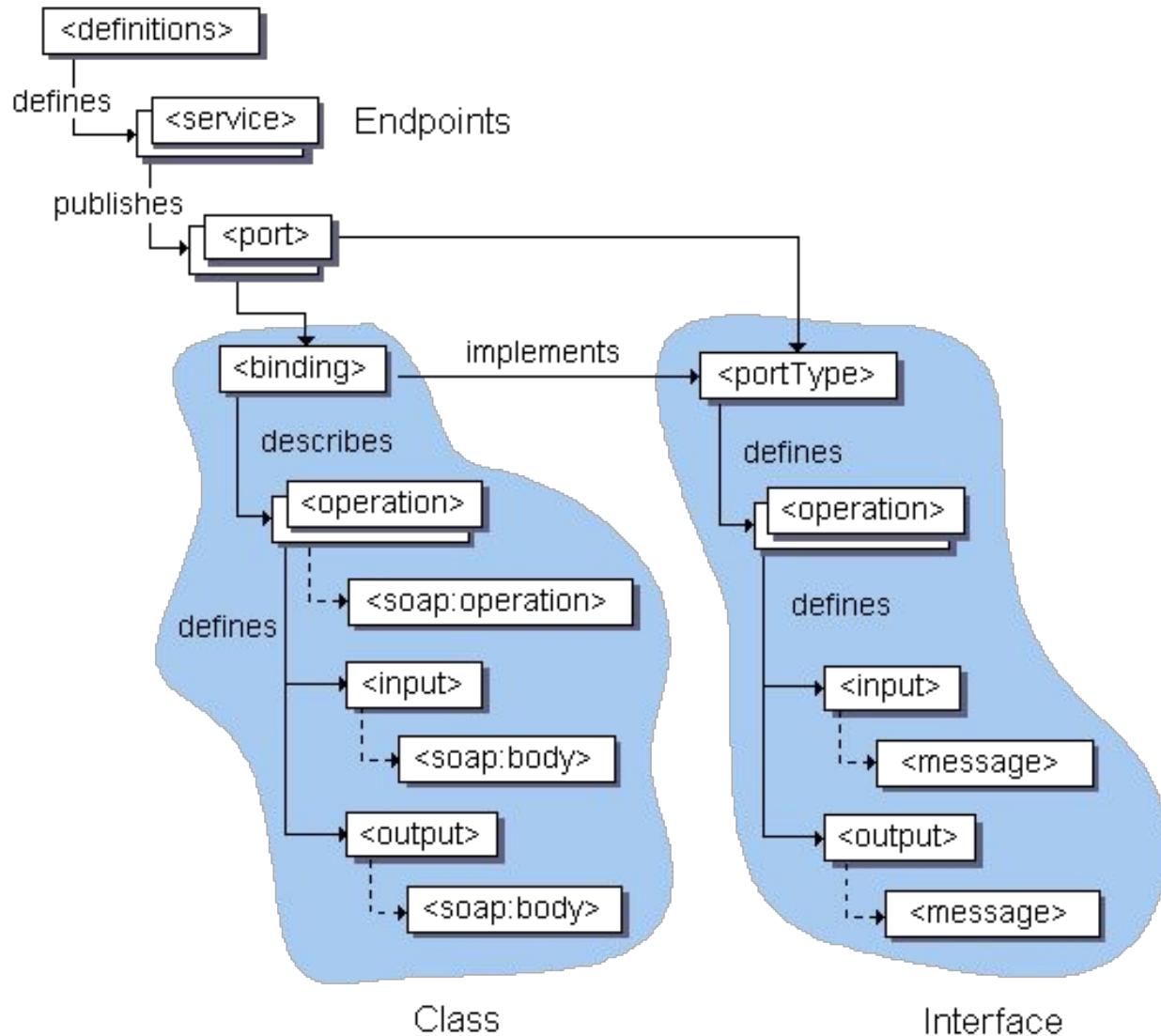
- Язык спецификации SOAP веб-сервисов.
- Базируется на XML.
- Описывает весь интерфейс сервиса:
 - функции;
 - аргументы;
 - возвращаемые значения.
- Может автогенерироваться по API сервиса, или, наоборот -- API сервиса может автогенерироваться по WSDL.

Структура документа WSDL 1.1 (1)

- *Types* (определение типов данных) — определение вида отправляемых и получаемых сервисом XML-сообщений.
- *Message* (элементы данных) — сообщения, используемые web-сервисом
- *PortType* (абстрактные операции) — список операций, которые могут быть выполнены с сообщениями.
- *Binding* (связывание сервисов) — способ, которым сообщение будет доставлено.



Структура документа WSDL 1.1 (2)

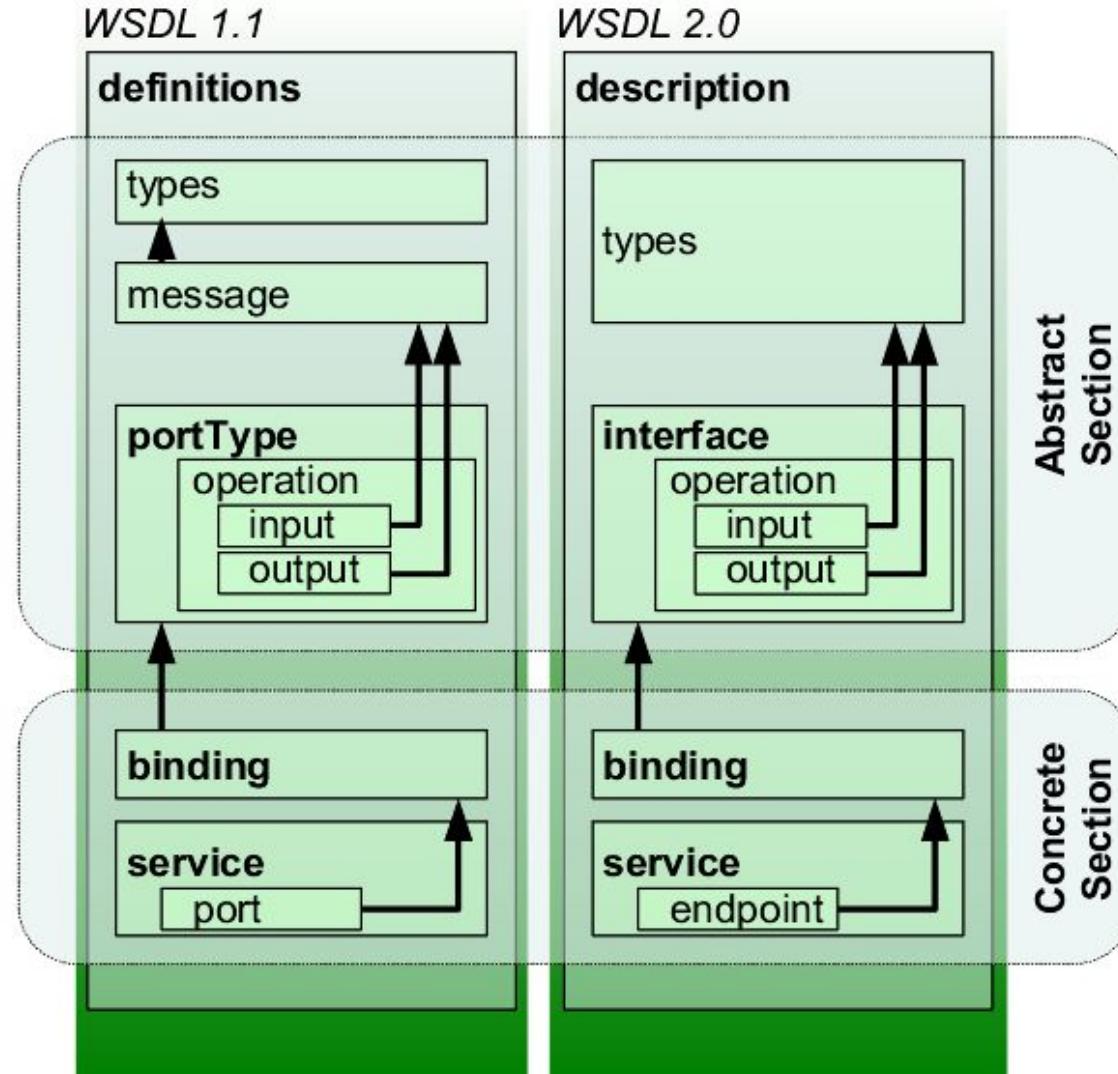


Пример описания сервиса на WSDL 1.1

```
<message name="getTermRequest">
    <part name="term" type="xs:string"/>
</message>
<message name="getTermResponse">
    <part name="value" type="xs:string"/>
</message>
<portType name="glossaryTerms">
    <operation name="getTerm">
        <input message="getTermRequest"/>
        <output message="getTermResponse"/>
    </operation>
</portType>
```



WSDL 1.1 vs WSDL 2.0



JAX-WS

- “Каноничный” вариант реализации SOAP веб-сервисов на Java.
- Входит в спецификацию Java EE, поддерживается любым full profile сервером приложений.
- Основан на метaprogramмировании -- разметке класса аннотациями.
- Хорошо совместим с EJB.
- WSDL можно генерировать автоматически по коду (или наоборот -- код по WSDL).

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.



Пример сервиса на JAX-WS: интерфейс

@WebService

```
public interface EmployeeService {
```

@WebMethod

```
Employee getEmployee(int id);
```

@WebMethod

```
Employee updateEmployee(int id, String name);
```

@WebMethod

```
boolean deleteEmployee(int id);
```

@WebMethod

```
Employee addEmployee(int id, String name);
```

```
// ...
```

```
}
```

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);
    }

    @WebMethod
    public Employee updateEmployee(int id, String name) {
        return employeeRepositoryImpl.updateEmployee(id, name);
    }

    // ...
}
```

Пример сервиса на JAX-WS: публикация

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

SOAP-клиент на Java

- Обычный класс на Java.
- Использует аннотацию `javax.xml.ws.WebServiceRef` для инъекции сервиса:

```
@WebServiceRef(wsdlLocation="http://localhost:  
:8080/employeeservice?wsdl")  
static EmployeeService service;
```

- Взаимодействие с сервисом реализуется через прокси-класс (порт):

```
EmployeeService port =  
    service.getEmployeeServicePort();
```

SOAP-клиент на Java: wsimport

- Утилита, входила в состав JDK до версии 1.8.
- Сейчас доступна для загрузки как независимое OpenSource-приложение, либо как плагин Maven.
- Позволяет сгенерировать стаб клиента по дескриптору WSDL:

```
> wsimport -d -keep -verbose  
http://localhost:8080/soap/mywebservice?wsdl
```



SOAP-клиент на Java: JAX-WS

```
// Стаб клиента сгенерирован с помощью wsimport
public class EmployeeServiceClient {
    public static void main(String[] args) throws Exception {
        URL url =
            new URL("http://localhost:8080/employeeservice?wsdl");

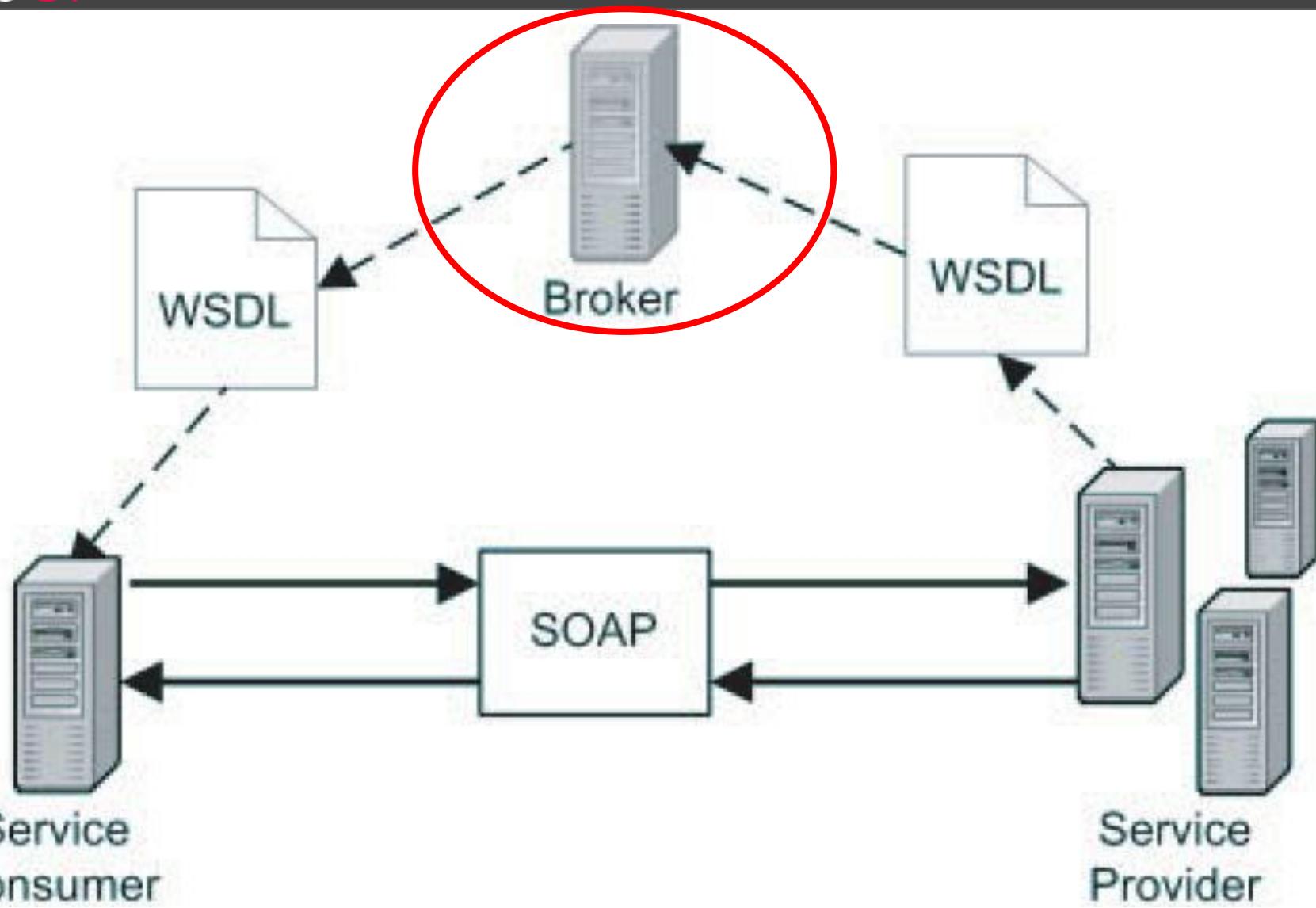
        EmployeeService_Service employeeService_Service
            = new EmployeeService_Service(url);
        EmployeeService employeeServiceProxy
            = employeeService_Service.getEmployeeServiceImplPort();

        List<Employee> allEmployees
            = employeeServiceProxy.getAllEmployees();
    }
}
```

14. Реестры сервисов и сервисные шины



Реестры сервисов

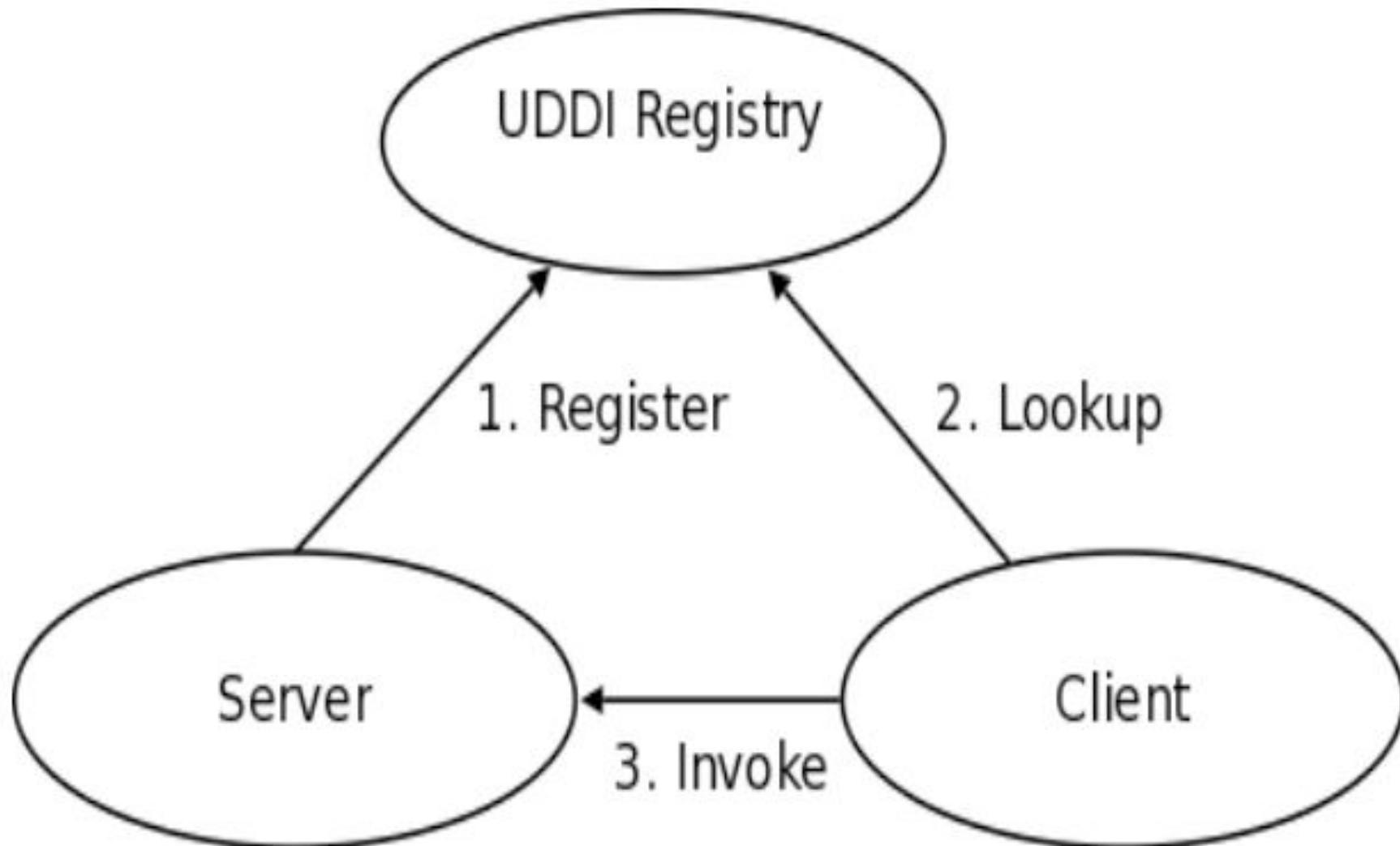


UDDI: Universal Description, Discovery and Integration

- Централизованное хранилище дескрипторов WSDL со стандартизованным API.
- Предложен в 2000 г., специфицирован OASIS.
- Реализует функциональность Service Discovery для SOAP.
- Есть разные реализации от разных вендоров.
- Разработан для публичных сервисов, но в настоящее время используется, в основном, во внутренней инфраструктуре.



Принцип работы UDDI



Структура реестра UDDI

Белые страницы	Жёлтые страницы	Зелёные страницы
<ul style="list-style-type: none">• Наименование компании.• Контактная информация.• Краткое описание.• Идентификаторы (ИНН / КПП и тд и тп).	<ul style="list-style-type: none">• Список предоставляемых сервисов.• Индустриальные коды (а-ля ОКВЭД).• Почтовые и географические индексы.	<ul style="list-style-type: none">• Техническая информация о сервисах.• Спецификация API.• Связывающие шаблоны.



UDDI: пример записи в реестре

```
<businessEntity businessKey = "uuid:C0E6D5A8-C446-4f01-99DA-70E212685A40"
    operator = "http://www.ibm.com" authorizedName = "John Doe">
    <name>Acme Company</name>
    <description>
        We create cool Web services
    </description>
    <contacts>
        <contact useType = "general info">
            <description>General Information</description>
            <personName>John Doe</personName>
            <phone>(123) 123-1234</phone>
            <email>jdoe@acme.com</email>
        </contact>
    </contacts>
    <businessServices>
        ...
    </businessServices>
    <identifierBag>
        <keyedReference tModelKey = "UUID:8609C81E-EE1F-4D5A-B202-3EB13AD01823"
            name = "D-U-N-S" value = "123456789" />
    </identifierBag>
    <categoryBag>
        <keyedReference tModelKey = "UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"
            name = "NAICS" value = "111336" />
    </categoryBag>
</businessEntity>
```

UDDI: описание сервиса

```
<businessService serviceKey =
    "uuid:D6F1B765-BDB3-4837-828D-8284301E5A2A"
businessKey =
    "uuid:C0E6D5A8-C446-4f01-99DA-70E212685A40">
<name>Hello World Web Service</name>
<description>A friendly Web service</description>
<bindingTemplates>
    ...
</bindingTemplates>
<categoryBag />
</businessService>
```

UDDI: связывающий шаблон

```
<bindingTemplate serviceKey = "uuid:D6F1B765-BDB3-4837-828D-8284301E5A2A"
    bindingKey = "uuid:C0E6D5A8-C446-4f01-99DA-70E212685A40" >
    <description>Hello World SOAP Binding </description>
    <accessPoint URLType = "http">http://localhost:8080 </accessPoint>
    <tModelInstanceDetails>
        <tModelInstanceInfo tModelKey =
            "uuid:EB1B645F-CF2F-491f-811A-4868705F5904" >
            <instanceDetails>
                <overviewDoc>
                    <description>
                        references the description of the WSDL service definition
                    </description>
                    <overviewURL>
                        http://localhost/helloworld.wsdl
                    </overviewURL>
                </overviewDoc>
            </instanceDetails>
        </tModelInstanceInfo>
    </tModelInstanceDetails>
</bindingTemplate>
```

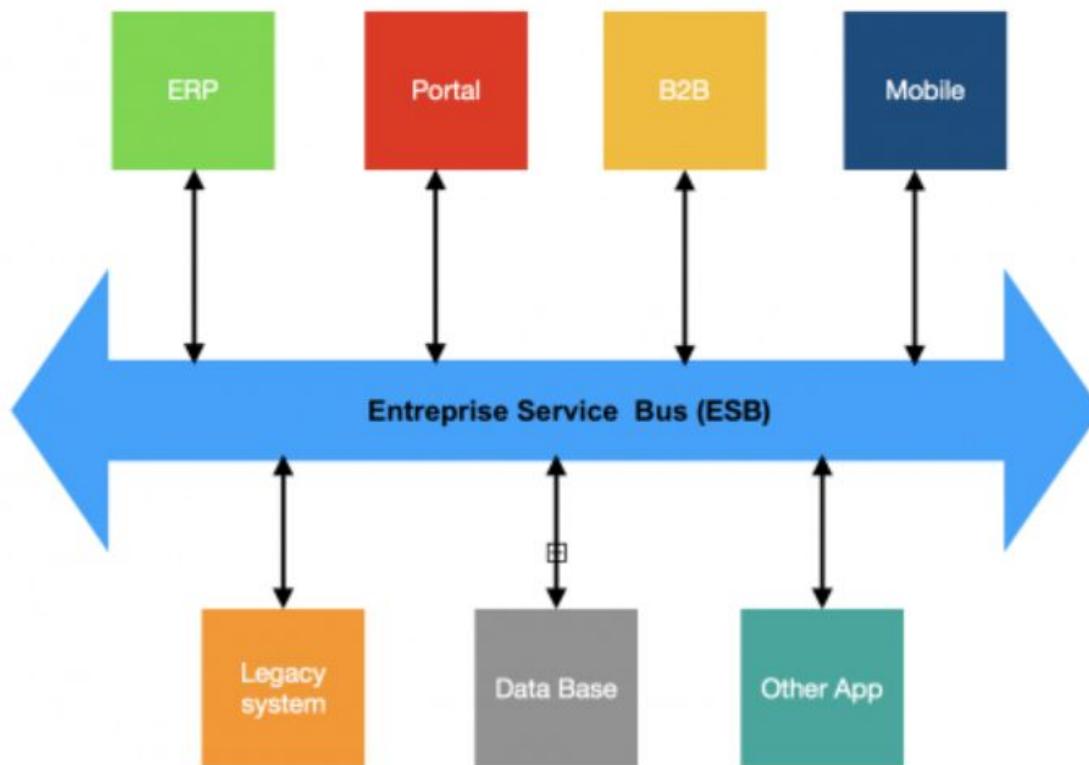


UDDI API



Сервисные шины

- Сервисная шина предприятия (Enterprise Service Bus, ESB) — ПО, обеспечивающее обмен сообщениями между различными ИС на принципах СОА.



Возможности ESB

- 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).



Примеры решений ESB

Коммерческие	Открытые
<ul style="list-style-type: none">• IBM App Connect.• InterSystems Ensemble.• Information_Builders iWay Service Manager.• Microsoft Azure Service Bus.• Microsoft BizTalk Server.• Mule ESB.• Oracle ESB.• Progress Software Sonic ESB.• SAP Process Integration.• TIBCO Software ActiveMatrix BusinessWorks.	<ul style="list-style-type: none">• Apache Camel.• Apache ServiceMix.• Apache Synapse.• Fuse ESB from Red Hat.• JBoss ESB.• NetKernel.• Open ESB.• Petals ESB.• Spring Integration.• UltraESB.• WSO2 ESB.

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.

Mule: проекты

- Проект -- обычное Java-приложение.
- Может собираться Anypoint Studio, может -- с помощью Maven.
- Для сборки с помощью Maven нужно подключить плагин:

```
<pluginGroups>
    <pluginGroup>org.mule.tools</pluginGroup>
</pluginGroups>
```

Взаимодействие с объектом сообщения

```
public Object onCall(  
    MuleEventContext eventContext)  
throws Exception {  
  
    MuleMessage message = eventContext  
        .getMessage();  
  
    message.setPayload("Message payload" +  
        " is changed here.");  
  
    return message;  
}
```



Конфигурация потока с помощью XML

ИТМО ВТ

```
<flow name="Flow">
    <http:listener
        config-ref="HTTP_Listener_Configuration"
        path="/" doc:name="HTTP"
        allowedMethods="POST"/>
    <logger message="Original
        paylaod: #[payload]"
        level="INFO" doc:name="Logger"/>
    <custom-transformer
        class="com.baeldung.transformer.InitializationTransformer"
        doc:name="Java"/>
    <logger message="Payload After Initialization: #[payload]"
        level="INFO" doc:name="Logger"/>
    <set-variable variableName="f1"
        value="#['Flow Variable 1']" doc:name="F1"/>
    <set-session-variable variableName="s1"
        value="#['Session variable 1']" doc:name="S1"/>
    <vm:outbound-endpoint exchange-pattern="request-response"
        path="test" doc:name="VM"/>
</flow>
```



ИТМО ВТ

Конфигурация потока с помощью Anypoint Studio

The screenshot shows the Anypoint Studio interface with the following components:

- Mule Palette**: A sidebar on the right containing a list of Mule components with icons:
 - Logger (Core)
 - Listener (HTTP)
 - Request (HTTP)
 - Transform Message (Core)
 - Set Payload (Core)
 - Flow Reference (Core)
 - Choice (Core)
 - Set Variable (Core)
 - Create (Salesforce)
 - SAP Send IDoc (SAP)
 - Integrations (Workday)
 - Invoke (ServiceNow)
 - Query (Salesforce)
- Message Flows**: Three message flows are visible in the main workspace:
 - accountsFlow1**: Starts with a Scheduler component, followed by a Retrieve component, then a Select component. This is followed by a Choice component with two branches. The first branch contains a Store component with the condition `#{sizeOf(payload as Ar...}`, a Transform Message component, and a Logger component. The second branch is labeled "Default" and contains a single Logger component.
 - getCSVAccountsFlow**: Starts with a Scheduler component, followed by a Transform Message component, and a final Logger component.
 - accountsFlow**: Starts with a Global Elements component, followed by a Transform Message component, and a final Logger component.
- Toolbars and Panels**: The top bar includes standard file operations like New, Open, Save, and Exit. The left side has a Package Explorer panel showing project structure and an MUnit panel showing test results. The bottom includes Mule Properties and Problems panels.