

Developing Microservices with Spring Boot

Building Microservices with Spring Boot

- Setting up the latest Spring development environment
- Developing RESTful services using the Spring framework
- Using Spring Boot to build fully qualified microservices
- Useful Spring Boot features to build production-ready microservices

Setting up a Development Environment

- **JDK 1.8:**

<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

- **Spring Tool Suite 3.7.2 (STS):** <https://spring.io/tools/sts/all>
- **Maven 3.3.1:** <https://maven.apache.org/download.cgi>

We are doing this class based on the following versions of Spring libraries:

- Spring Framework 4.2.6.RELEASE
- Spring Boot 1.3.5.RELEASE

Build a Legacy Rest Application with Spring

Please complete LAB 1 : <https://jmp.sh/2xiRgOF>

Legacy to Microservices

- Carefully examining the preceding RESTful service will reveal whether this really constitutes a microservice.
- At first glance, the preceding RESTful service is a fully qualified interoperable REST/JSON service.
- However, it is not fully autonomous in nature.
 - This is primarily because the service relies on an underlying application server or web container.
- This is a traditional approach to developing RESTful services as a web application. However, from the microservices point of view, one needs a mechanism to develop services as executables, self-contained JAR files with an embedded HTTP listener.
 - Spring Boot is a tool that allows easy development of such kinds of services. Dropwizard and WildFly Swarm are alternate server-less RESTful stacks.

Use Spring Boot to Build Microservices

- Spring Boot is a utility framework from the Spring team to bootstrap Spring-based applications and microservices quickly and easily. The framework uses an opinionated approach over configurations for decision making, thereby reducing the effort required in writing a lot of boilerplate code and configurations.
- Using the 80-20 principle, developers should be able to kickstart a variety of Spring applications with many default values. Spring Boot further presents opportunities for the developers to customize applications by overriding the autoconfigured values.

Use Spring Boot to Build Microservices

```
<dependency>
```

```
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-data-jpa</artifactId>
```

```
</dependency>
```

```
<dependency>
```

```
    <groupId>org.hsqldb</groupId>
```

```
    <artifactId>hsqldb</artifactId>
```

```
    <scope>runtime</scope>
```

```
</dependency>
```

Let's Get Started with Spring Boot

- Using the Spring Boot CLI as a command-line tool
- Using IDEs such as STS to provide Spring Boot, which are supported out of the box
- Using the Spring Initializr project at <http://start.spring.io>

CLI

- The easiest way to develop and demonstrate Spring Boot's capabilities is using the Spring Boot CLI, a command-line tool.
- Complete Lab 2: <https://jmp.sh/YW2fGnV>

Lab 3 - Create a Spring Boot Java Microservice with STS

Lab 3 : <https://jmp.sh/PHm3TQX>

POM File

```
<parent>
```

```
  <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-parent</artifactId>
```

```
    <version>1.3.4.RELEASE</version>
```

```
</parent>
```

POM File Properties

`<spring-boot.version>1.3.5.BUILD-SNAPSHOT</spring-boot.version>`

`<hibernate.version>4.3.11.Final</hibernate.version>`

`<jackson.version>2.6.6</jackson.version>`

`<jersey.version>2.22.2</jersey.version>`

`<logback.version>1.1.7</logback.version>`

`<spring.version>4.2.6.RELEASE</spring.version>`

`<spring-data-releasetrain.version>Gosling-SR4</spring-data-releasetrain.version>`

`<tomcat.version>8.0.33</tomcat.version>`

More POM File Review

```
<dependencies>
```

```
  <dependency>
```

```
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-web</artifactId>
```

```
  </dependency>
```

```
  <dependency>
```

```
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-test</artifactId>
```

```
    <scope>test</scope>
```

```
  </dependency>
```

```
</dependencies>
```

Java Version in the POM File

```
<java.version>1.8</java.version>
```

Application.java

Spring Boot, by default, generated a `org.rvslab.session2.Application.java` class under `src/main/java` to bootstrap, as follows:

```
@SpringBootApplication
```

```
public class Application {
```

```
    public static void main(String[] args) {
```

```
        SpringApplication.run(Application.class, args);
```

```
    }
```

```
}
```

More Application.java

@Configuration

@EnableAutoConfiguration

@ComponentScan

public class Application {

application.properties

- A default `application.properties` file is placed under `src/main/resources`.
- It is an important file to configure any required properties for the Spring Boot application.

ApplicationTests.java

- The last file to be examined is `ApplicationTests.java` under `src/test/java`.
 - This is a placeholder to write test cases against the Spring Boot application.

Implement a RESTful Web Service : Lab

Lab 4 : <https://jmp.sh/kQbFFuR>

```
GreetControllerApplication.java  GreetControllerApplicationTests.java  GreetController/pom.xml  application.properties
1 package com.windstream.tutorial;
2
3 import org.junit.Test;
4 import org.junit.runner.RunWith;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.boot.test.context.SpringBootTest;
7 import org.springframework.test.context.junit4.SpringRunner;
8 import org.springframework.web.client.RestTemplate;
9 import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
10 import org.springframework.boot.test.web.client.TestRestTemplate;
11 import org.springframework.boot.test.web.server.LocalServerPort;
12
13 import junit.framework.Assert;
14
15 @RunWith(SpringRunner.class)
16 @SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
17 public class GreetControllerApplicationTests {
18
19     // @Autowired
20     // private TestRestTemplate restTemplate;
21     // @LocalServerPort
22     private int port;
23
24     @SuppressWarnings("deprecation")
25     @Test
26     public void contextLoads() {
27         RestTemplate restTemplate = new RestTemplate();
28         Greet greet = restTemplate.getForObject("http://localhost:" + port + "/", Greet.class);
29         Assert.assertEquals("Hello World!", greet.getMessage());
30     }
31 }
32
33
34
```

```
GreetControllerApplication.java  GreetControllerApplicationTests.java  GreetController/pom.xml  application.properties
1 package com.windstream.tutorial;
2
3 import org.springframework.boot.SpringApplication;
4
5
6
7
8
9 @SpringBootApplication
10 public class GreetControllerApplication {
11
12     public static void main(String[] args) {
13         SpringApplication.run(GreetControllerApplication.class, args);
14     }
15
16
17
18 @RestController
19 class GreetingController {
20     @RequestMapping("/")
21     Greet greet() {
22         return new Greet("Hello World, GreetingController!");
23     }
24 }
25
26 class Greet {
27     private String message;
28
29     public Greet() {}
30
31     public Greet(String message) {
32         this.message = message;
33     }
34
35     public void setMessage(String message) {
36         this.message = message;
37     }
38
39     public String getMessage() {
40         return this.message;
41     }
42 }
43
44
```

HATEOAS

- HATEOAS is a REST service pattern in which navigation links are provided as part of the payload metadata.
- The client application determines the state and follows the transition URLs provided as part of the state.
- This methodology is particularly useful in responsive mobile and web applications in which the client downloads additional data based on user navigation patterns.

HATEOAS : LAB 5

<https://jmp.sh/n8QVINO>

Momentum

- A number of basic Spring Boot examples have been reviewed so far.
- The rest of this section will examine some of the Spring Boot features that are important from a microservices development perspective.
- In the upcoming sections, we will take a look at how to work with dynamically configurable properties, change the default embedded web server, add security to the microservices, and implement cross-origin behavior when dealing with microservices.

Spring Boot Configuration

- In this section, the focus will be on the configuration aspects of Spring Boot.
- The `session2.bootrest` project, already developed, will be modified in this section to showcase configuration capabilities.
- Copy and paste `session2.bootrest` and rename the project as `session2.boot-advanced`.

Spring Boot autoconfiguration

- Spring Boot uses convention over configuration by scanning the dependent libraries available in the class path.
- For each `spring-boot-starter-*` dependency in the POM file, Spring Boot executes a default `AutoConfiguration` class. `AutoConfiguration` classes use the `*AutoConfiguration` lexical pattern, where `*` represents the library.
 - For example, the autoconfiguration of JPA repositories is done through `JpaRepositoriesAutoConfiguration`.
- Run the application with `--debug` to see the autoconfiguration report. The following command shows the autoconfiguration report for the `session2.boot-advanced` project:

```
$java -jar target/bootadvanced-0.0.1-SNAPSHOT.jar --debug
```


Autoconfiguration Classes

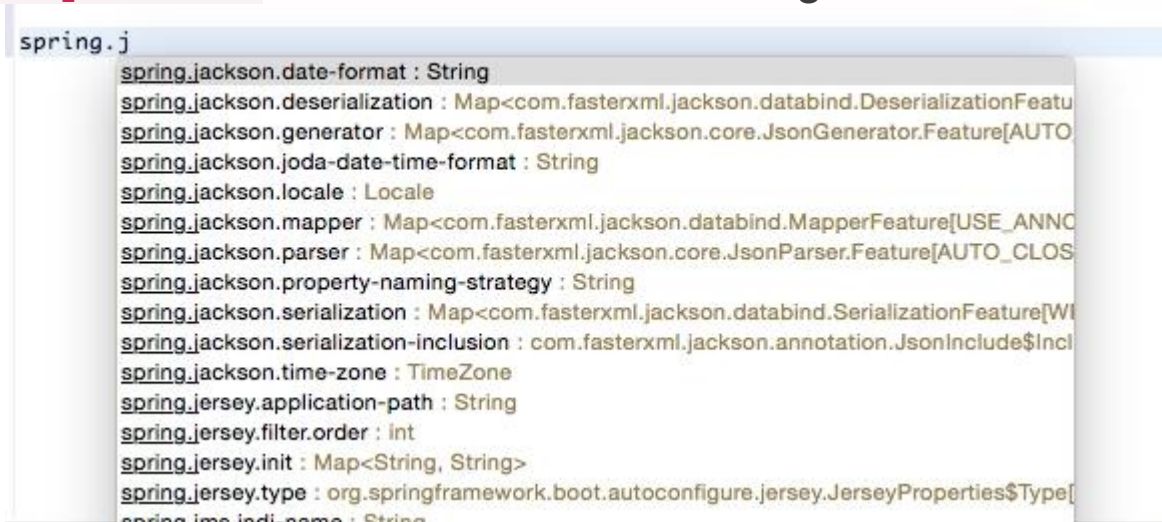
- `ServerPropertiesAutoConfiguration`
- `RepositoryRestMvcAutoConfiguration`
- `JpaRepositoriesAutoConfiguration`
- `JmsAutoConfiguration`

You can exclude the autoconfiguration of a library - here is an example:

```
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
```

It is also possible to override default configuration values using the `application.properties` file.

STS provides an easy-to-autocomplete, contextual help on `application.properties`, as shown in the following screenshot:



```
spring.j
spring.jackson.date-format : String
spring.jackson.deserialization : Map<com.fasterxml.jackson.databind.DeserializationFeatu
spring.jackson.generator : Map<com.fasterxml.jackson.core.JsonGenerator.Feature[AUTO
spring.jackson.joda-date-time-format : String
spring.jackson.locale : Locale
spring.jackson.mapper : Map<com.fasterxml.jackson.databind.MapperFeature[USE_ANNOC
spring.jackson.parser : Map<com.fasterxml.jackson.core.JsonParser.Feature[AUTO_CLOS
spring.jackson.property-naming-strategy : String
spring.jackson.serialization : Map<com.fasterxml.jackson.databind.SerializationFeature[W
spring.jackson.serialization-inclusion : com.fasterxml.jackson.annotation.JsonInclude$Incl
spring.jackson.time-zone : TimeZone
spring.jersey.application-path : String
spring.jersey.filter.order : int
spring.jersey.init : Map<String, String>
spring.jersey.type : org.springframework.boot.autoconfigure.jersey.JerseyProperties$Type[
spring.jersey.type : String
```

Where is the config file?

Spring Boot externalizes all configurations into `application.properties`

```
spring.config.name= # config file name
```

```
spring.config.location= # location of config file
```

```
$java -jar target/bootadvanced-0.0.1-SNAPSHOT.jar  
--spring.config.name=bootrest.properties
```

Custom Property Files : Lab 6

- At startup, `SpringApplication` loads all the properties and adds them to the Spring `Environment` class.
- Add a custom property to the `application.properties` file.
- In this case, the custom property is named `bootrest.customproperty`.
- Autowire the Spring `Environment` class into the `GreetingController` class.
- Edit the `GreetingController` class to read the custom property from `Environment` and add a log statement to print the custom property to the console.

Lab 6 : <https://jmp.sh/illuGbD>

Default Web Server

Embedded HTTP listeners can easily be customized as follows. By default, Spring Boot supports Tomcat, Jetty, and Undertow. Replace Tomcat is replaced with Undertow (This is Lab 6.5):

```
<dependency>
```

```
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-web</artifactId>
```

```
    <exclusions>
```

```
        <exclusion>
```

```
            <groupId>org.springframework.boot</groupId>
```

```
            <artifactId>spring-boot-starter-tomcat</artifactId>
```

```
        </exclusion>
```

```
    </exclusions>
```

```
</dependency>
```

```
<dependency>
```

Securing Microservices with basic security

- Adding basic authentication to Spring Boot is pretty simple. Add the following dependency to `pom.xml`. This will include the necessary Spring security library files:

```
<dependency>  
  <groupId>org.springframework.boot </groupId>  
  <artifactId>spring-boot-starter-security </artifactId>  
</dependency>
```

Securing Microservices with basic security

Open `Application.java` and add `@EnableGlobalMethodSecurity` to the `Application` class.

This annotation will enable method-level security:

```
@EnableGlobalMethodSecurity
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Securing Microservices with basic security

The default basic authentication assumes the user as being `user`. The default password will be printed in the console at startup. Alternately, the username and password can be added in `application.properties`, as shown here:

```
security.user.name=guest
```

```
security.user.password=guest123
```


Securing Microservices with basic security

```
@Test
public void testSecureService() {
    String plainCreds = "guest:guest123";
    HttpHeaders headers = new HttpHeaders();
    headers.add("Authorization", "Basic " + new
String(Base64.encode(plainCreds.getBytes())));
    HttpEntity<String> request = new HttpEntity<String>(headers);
    RestTemplate restTemplate = new RestTemplate();

    ResponseEntity<Greet> response = restTemplate.exchange("http://localhost:8080",
HttpMethod.GET, request, Greet.class);
    Assert.assertEquals("Hello World!", response.getBody().getMessage());
}
```

Securing Microservices with basic security

As shown in the code, a new `Authorization` request header with Base64 encoding the username-password string is created.

Rerun the application using Maven. Note that the new test case passed, but the old test case failed with an exception. The earlier test case now runs without credentials, and as a result, the server rejected the request with the following message:

```
org.springframework.web.client.HttpClientErrorException: 401 Unauthorized
```

Securing a Microservice with OAuth2

- When a client application requires access to a protected resource, the client sends a request to an authorization server.
- The authorization server validates the request and provides an access token.
- This access token is validated for every client-to-server request.
- The request and response sent back and forth depends on the grant type.

LAB 7 : OATH2 LAB : <https://jmp.sh/C212HLk>

Enabling cross-origin access for Microservices

- Browsers are generally restricted when client-side web applications running from one origin request data from another origin. Enabling cross-origin access is generally termed as **CORS (Cross-Origin Resource Sharing)**.
- With microservices, as each service runs with its own origin, it will easily get into the issue of a client-side web application consuming data from multiple origins. For instance, a scenario where a browser client accessing Customer from the Customer microservice and Order History from the Order microservices is very common in the microservices world.
- Spring Boot provides a simple declarative approach to enabling cross-origin requests.

Enabling cross-origin access for Microservices

- The following example shows how to enable a microservice to enable cross-origin requests:

```
@RestController
class GreetingController{

    @CrossOrigin
    @RequestMapping("/")
    Greet greet(){
        return new Greet("Hello World!");
    }
}
```

Enabling cross-origin access for Microservices

- By default, all the origins and headers are accepted. We can further customize the cross-origin annotations by giving access to specific origins, as follows. The `@CrossOrigin` annotation enables a method or class to accept cross-origin requests:

```
@CrossOrigin("http://mytrustedorigin.com")
```

- Global CORS can be enabled using the `WebMvcConfigurer` bean and customizing the `addCorsMappings(CorsRegistry registry)` method.

Implementing Spring Boot Messaging: Lab 8

Lab 8 : <https://jmp.sh/UGrAhXI>

Developing a comprehensive microservice example:

Lab 9

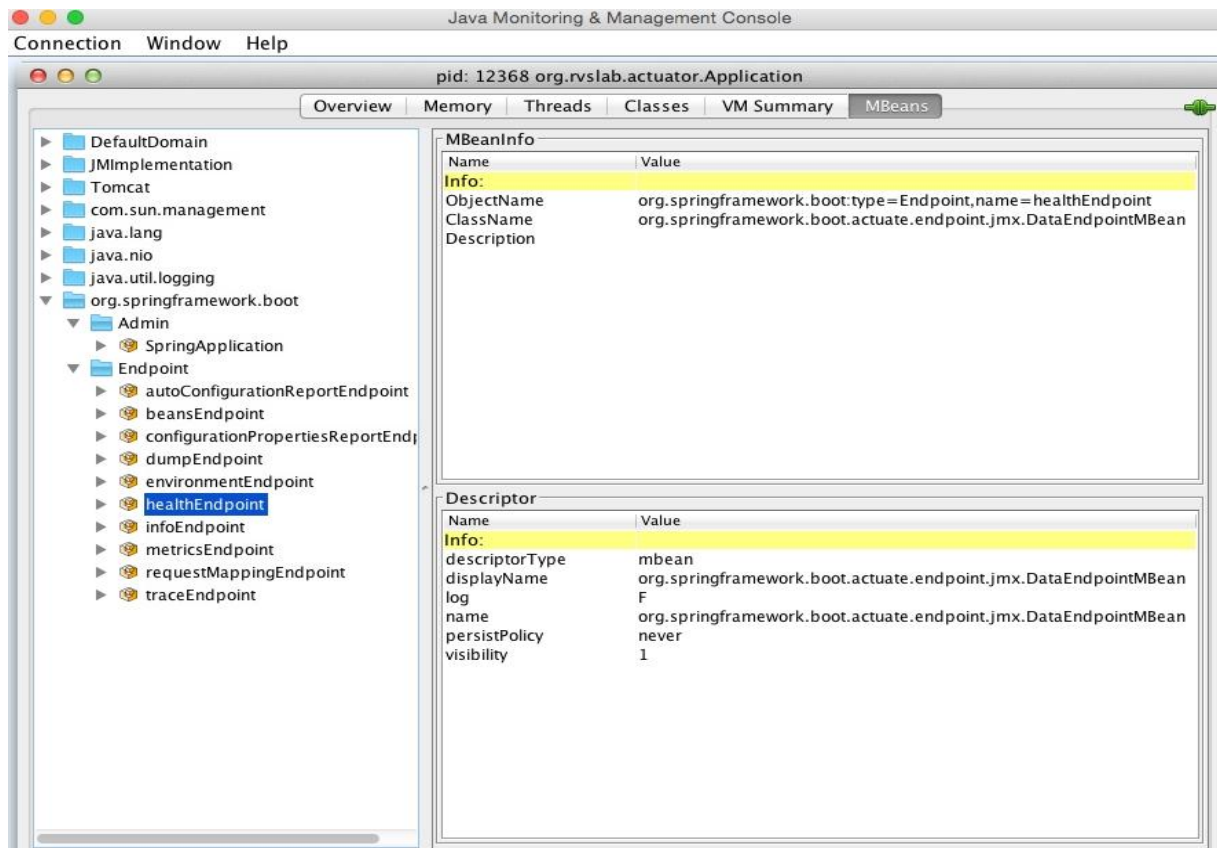
- So far, the examples we have considered are no more than just a simple "Hello world." Putting together what we have learned, this section demonstrates an end-to-end Customer Profile microservice implementation.
- The Customer Profile microservices will demonstrate interaction between different microservices.
- It also demonstrates microservices with business logic and primitive data stores.
- The Customer Profile microservice exposes methods to **create, read, update, and delete(CRUD)** a customer and a registration service to register a customer.
- The registration process applies certain business logic, saves the customer profile, and sends a message to the Customer Notification microservice.
- The Customer Notification microservice accepts the message sent by the registration service and sends an e-mail message to the customer using an SMTP server.
- Asynchronous messaging is used to integrate Customer Profile with the Customer Notification service.

Lab 9 : Complete End to End Microservice : <https://jmp.sh/eQfpQFC>

Spring Boot Actuators

- Spring Boot actuators provide an excellent out-of-the-box mechanism to monitor and manage Spring Boot applications in production
- Lab 10 - <https://jmp.sh/wUXrzcs>

Monitoring Using JConsole



Monitoring Using SSH

Spring Boot provides remote access to the Boot application using SSH. The following command connects to the Spring Boot application from a terminal window:

```
$ ssh -p 2000 user@localhost
```

The password can be customized by adding the `shell.auth.simple.user.password` property in the `application.properties` file. The updated `application.properties` file will look similar to the following:

```
shell.auth.simple.user.password=admin
```

Configuring Application Information

```
management.endpoints.web.exposure.include=*  
info.app.name=Boot Actuator  
info.app.description=My Greetings Service  
info.app.version=1.0.0  
#endpoints.app.name=Boot Actuator
```

```
class TPSCounter {
    LongAdder count;
    int threshold = 2;
    Calendar expiry = null;

    TPSCounter(){
        this.count = new LongAdder();
        this.expiry = Calendar.getInstance();
        this.expiry.add(Calendar.MINUTE, 1);
    }

    boolean isExpired(){
        return Calendar.getInstance().after(expiry);
    }

    boolean isWeak(){
        return (count.intValue() > threshold);
    }

    void increment(){
        count.increment();
    }
}
```

Documenting Microservices

- The traditional approach of API documentation is either by writing service specification documents or using static service registries.
- With a large number of microservices, it would be hard to keep the documentation of APIs in sync.

```
<dependency>
```

```
<groupId>io.springfox<...
```

Summary

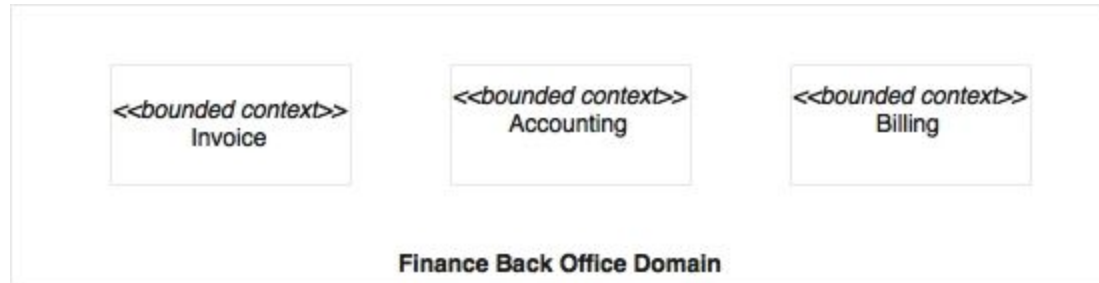
3 - Microservices Applied

- Trade-offs between different design choices and patterns to be considered when developing microservices
- Challenges and anti-patterns in developing enterprise grade microservices
- A capability model for a microservices ecosystem

Boundaries

- One of the most common questions relating to microservices is regarding the size of the service.
- How big (mini-monolithic) or how small (nano service) can a microservice be, or is there anything like right-sized services?
- Does size really matter?

Domains



Autonomous functions

- If the function under review is autonomous by nature, then it can be taken as a microservices boundary.
- Autonomous services typically would have fewer dependencies on external functions.
- They accept input, use its internal logic and data for computation, and return a result.
- All utility functions such as an encryption engine or a notification engine are straightforward candidates.

Size of a deployable unit

- Most of the microservices ecosystems will take advantage of automation, such as automatic integration, delivery, deployment, and scaling.
- Microservices covering broader functions result in larger deployment units.
- Large deployment units pose challenges in automatic file copy, file download, deployment, and start up times.
 - For instance, the size of a service increases with the density of the functions that it implements.

Most appropriate function or subdomain

- It is important to analyze what would be the most useful component to detach from the monolithic application.
- This is particularly applicable when breaking monolithic applications into microservices.
- This could be based on parameters such as resource-intensiveness, cost of ownership, business benefits, or flexibility.

Polyglot architecture

- One of the key characteristics of microservices is its support for polyglot architecture.
- In order to meet different non-functional and functional requirements, components may require different treatments.
- It could be different architectures, different technologies, different deployment topologies, and so on.
- When components are identified, review them against the requirement for polyglot architectures.

Selective Scaling

- Selective scaling is related to the previously discussed polyglot architecture.
- In this context, all functional modules may not require the same level of scalability.
- Sometimes, it may be appropriate to determine boundaries based on scalability requirements.

Small, agile teams

- Microservices enable Agile development with small, focused teams developing different parts of the pie.
- There could be scenarios where parts of the systems are built by different organizations, or even across different geographies, or by teams with varying skill sets.
- This approach is a common practice, for example, in manufacturing industries.

Single Responsibility

- In theory, the single responsibility principle could be applied at a method, at a class, or at a service.
 - However, in the context of microservices, it does not necessarily map to a single service or endpoint.
- A more practical approach could be to translate single responsibility into single business capability or a single technical capability.
 - As per the single responsibility principle, one responsibility cannot be shared by multiple microservices.
 - Similarly, one microservice should not perform multiple responsibilities.

Replicate and Change

- Innovation and speed are of the utmost importance in IT delivery.
- Microservices boundaries should be identified in such a way that each microservice is easily detachable from the overall system, with minimal cost of re-writing.
- If part of the system is just an experiment, it should ideally be isolated as a microservice.

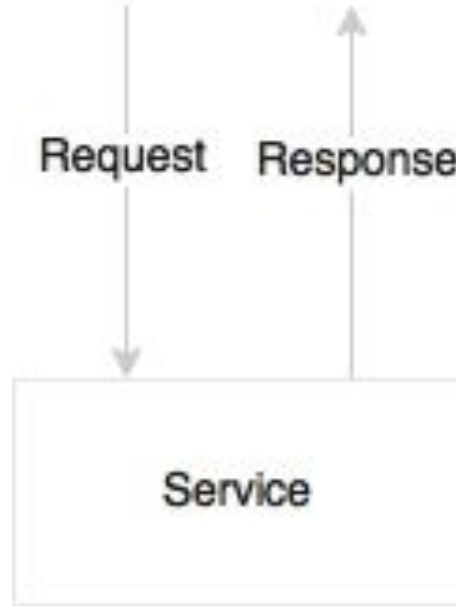
Coupling and Cohesion

- Coupling and cohesion are two of the most important parameters for deciding service boundaries.
- Dependencies between microservices have to be evaluated carefully to avoid highly coupled interfaces.
- A functional decomposition, together with a modeled dependency tree, could help in establishing a microservices boundary.
- Avoiding too chatty services, too many synchronous request-response calls, and cyclic synchronous dependencies are three key points, as these could easily break the system.

Microservice as a Product

- DDD also recommends mapping a bounded context to a product.
- As per DDD, each bounded context is an ideal candidate for a product.
- Think about a microservice as a product by itself.
- When microservice boundaries are established, assess them from a product's point of view to see whether they really stack up as product.
- It is much easier for business users to think boundaries from a product point of view.
- A product boundary may have many parameters, such as a targeted community, flexibility in deployment, sell-ability, reusability, and so on.

Synchronous Communication



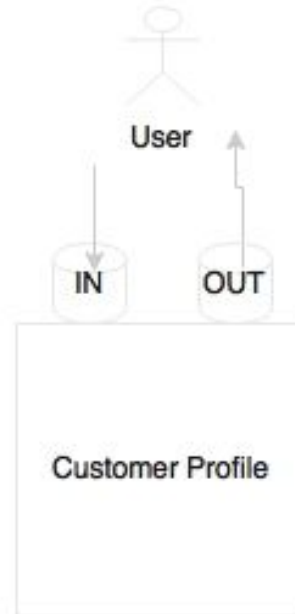
Asynchronous Communication



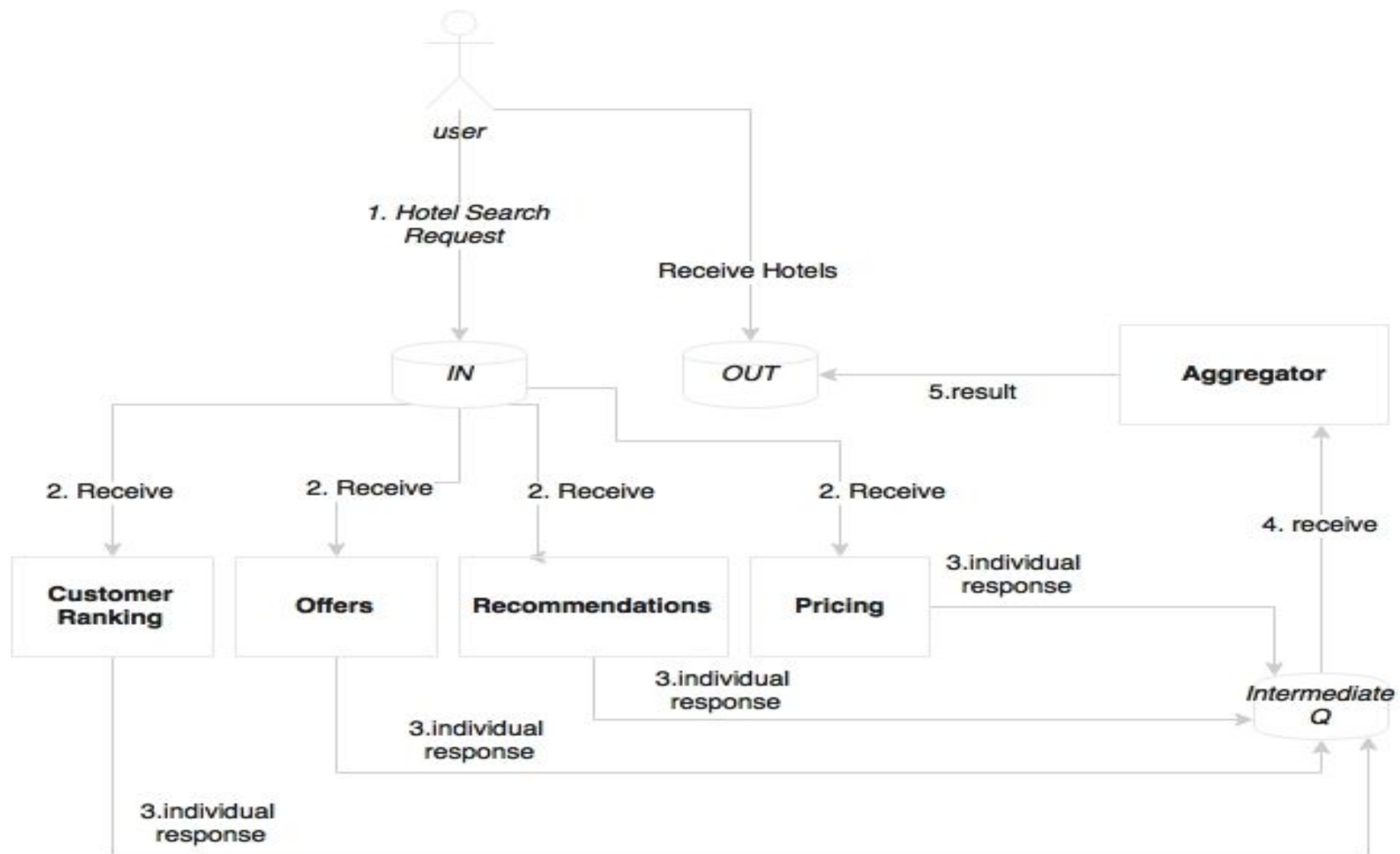
Which Style?

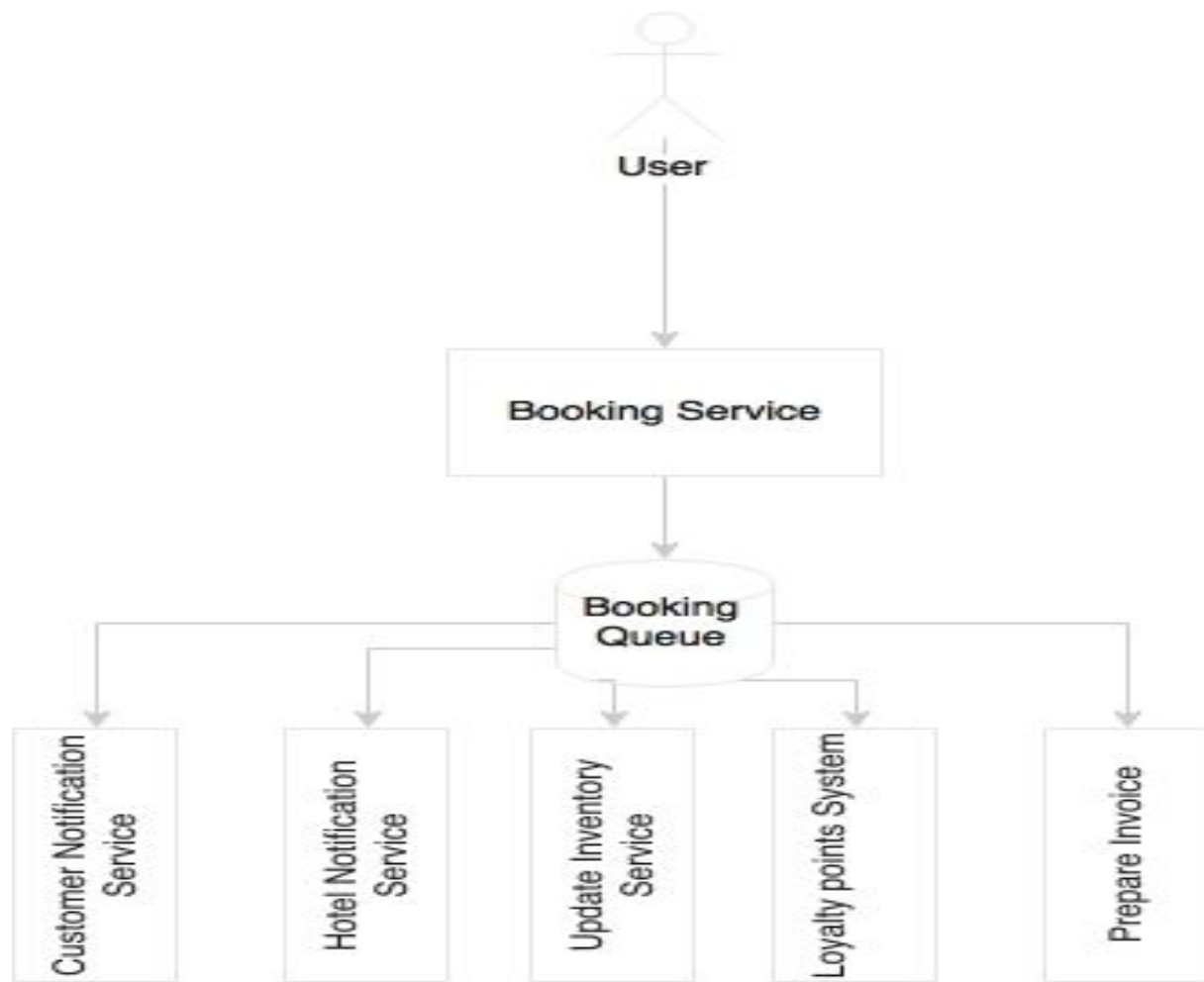


A) synchronous
request - response



B) asynchronous
request - response

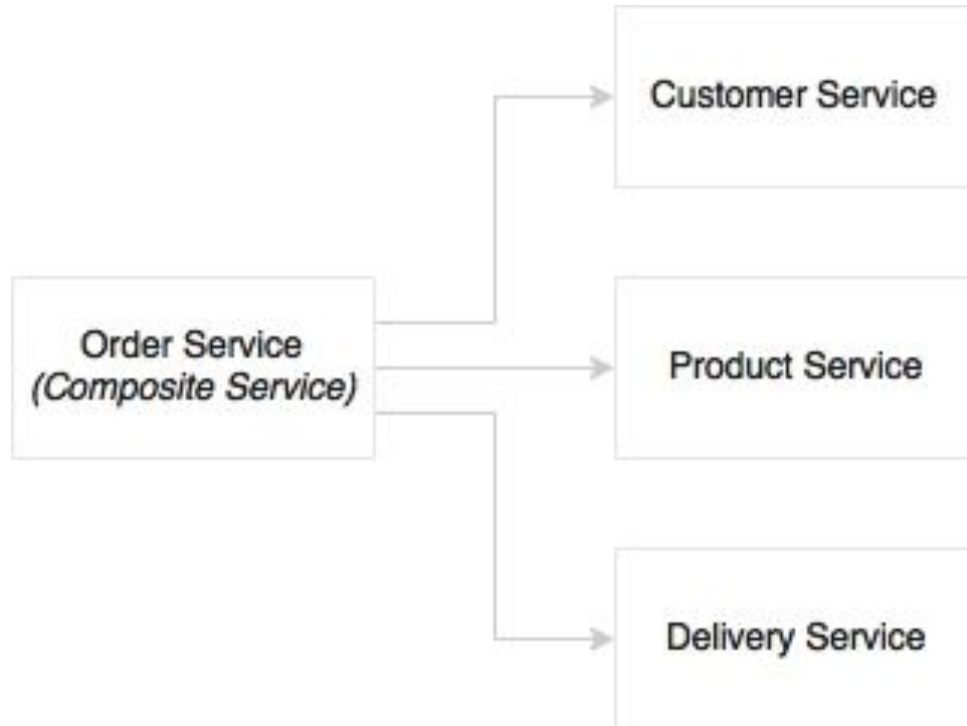




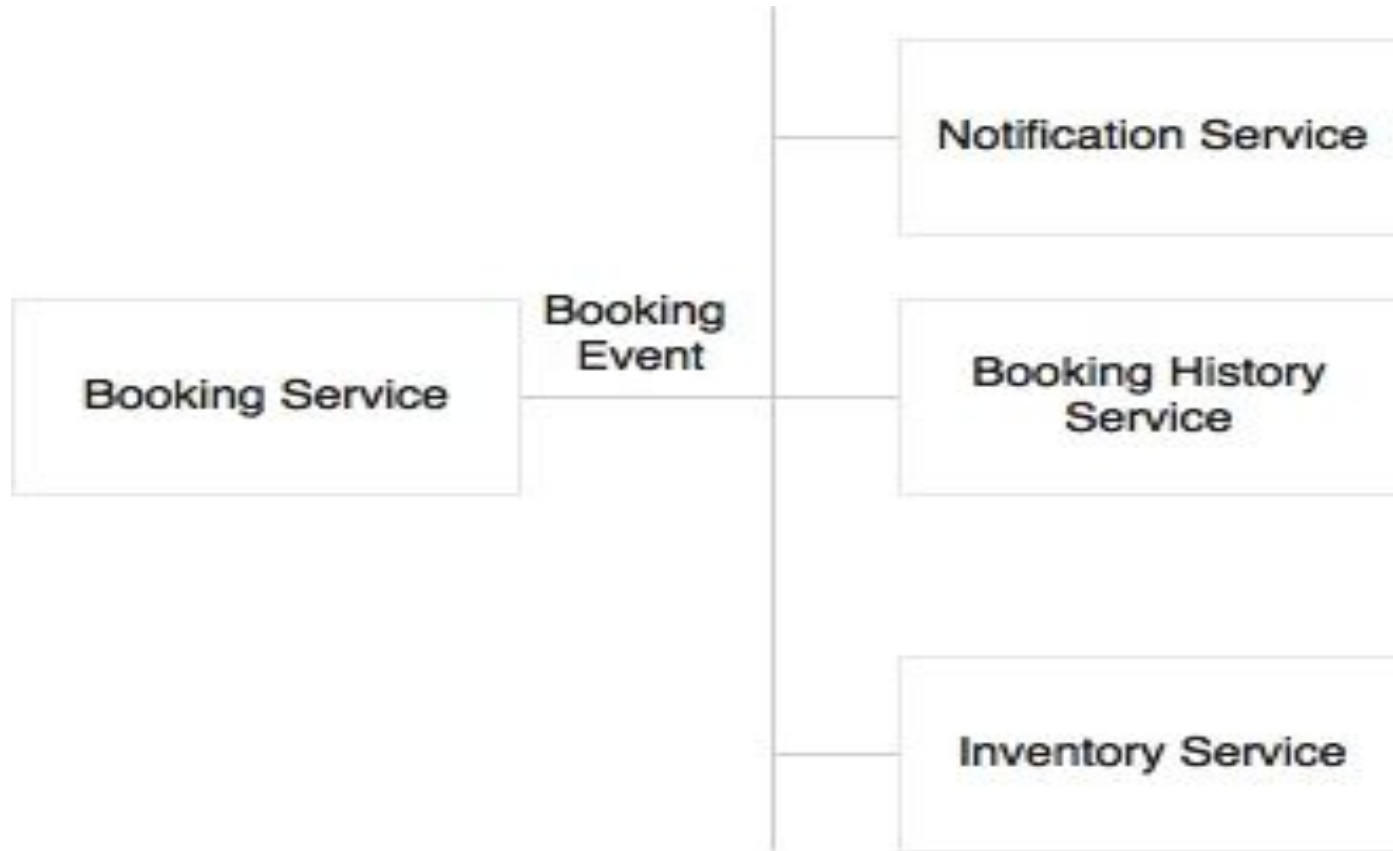
Orchestration of Microservices

- Composability is one of the service design principles.
- This leads to confusion around who is responsible for the composing services.
- In the SOA world, ESBs are responsible for composing a set of finely-grained services.
- In some organizations, ESBs play the role of a proxy, and service providers themselves compose and expose coarse-grained services. In the SOA world, there are two approaches for handling such situations.

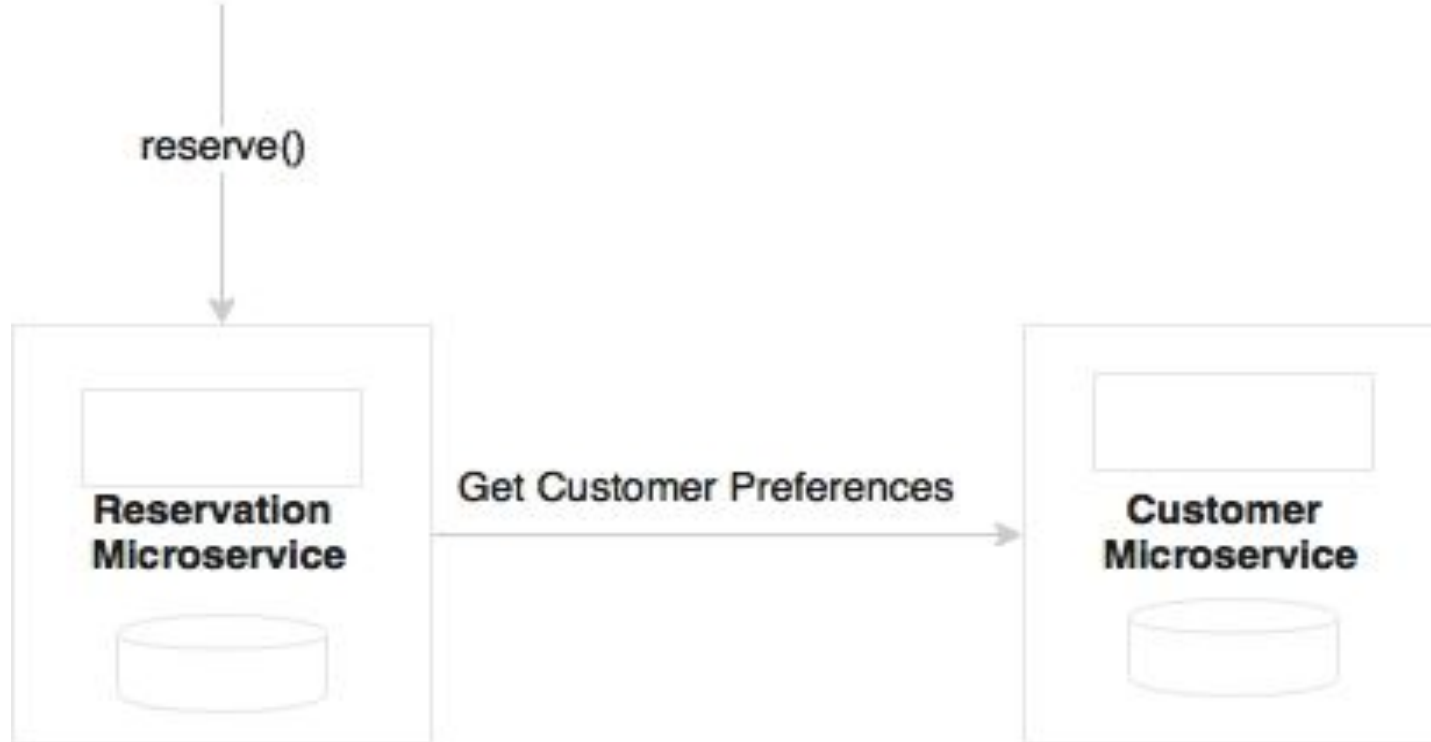
Orchestration



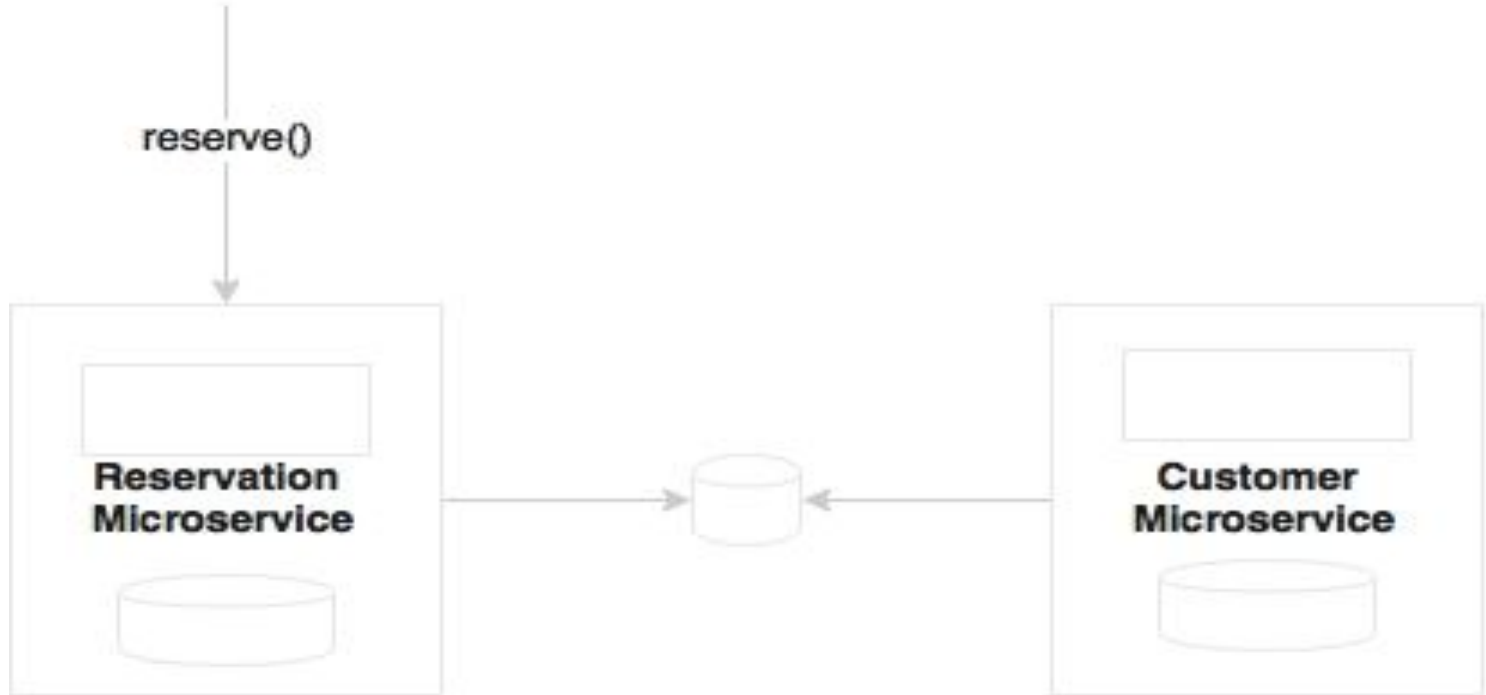
Choreography

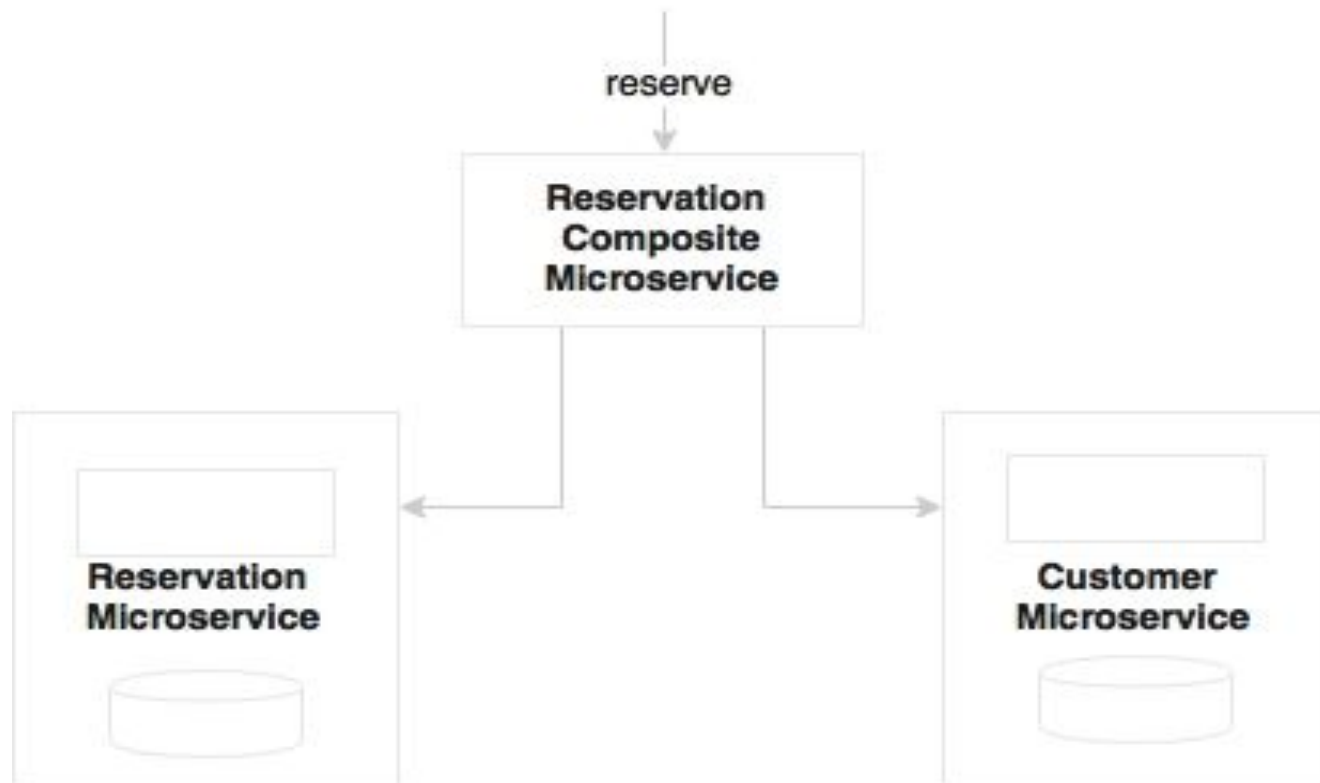


Does this work with Choreagraphy?

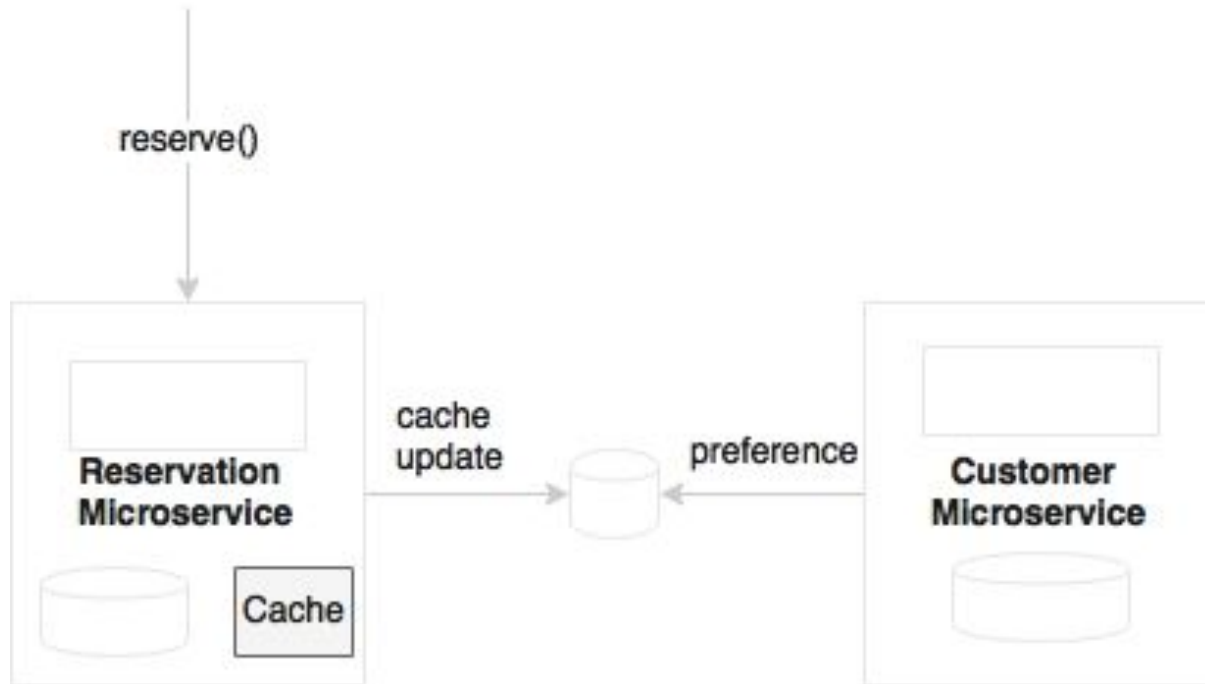


Can we make the reservation to Customer call Asynch

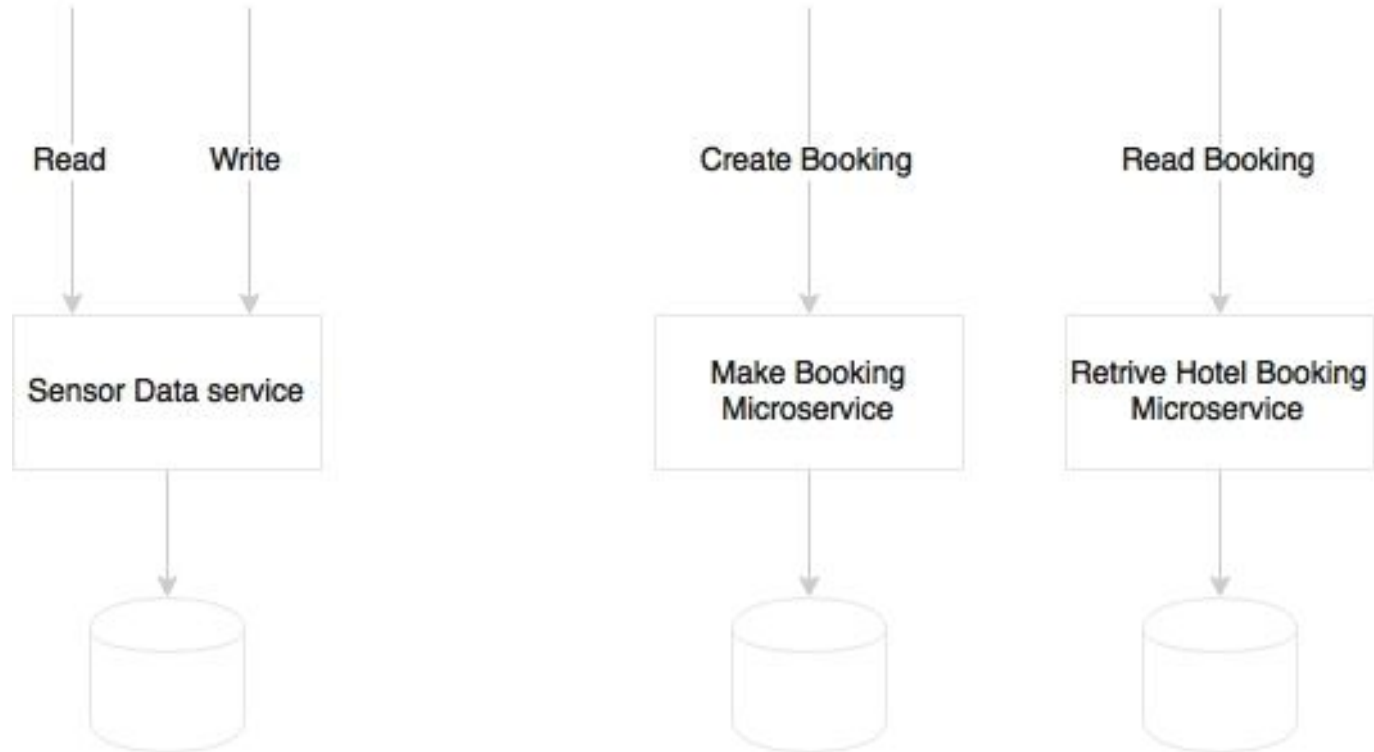




Can we duplicate customer preference & keep a copy of the preference in the Reservation?



How Many Endpoints in a microservices?



One microservice per VM or Several?

- One microservice could be deployed in multiple **Virtual Machines (VMs)** by replicating the deployment for scalability and availability.
 - This is a no brainer.
- The question is whether multiple microservices could be deployed in one virtual machine?
- There are pros and cons for this approach.
 - This question typically arises when the services are simple, and the traffic volume is less.

One VM or Many?

- Does the VM have enough capacity to run both services under peak usage?
- Do we want to treat these services differently to achieve SLAs (selective scaling)?
 - i. For example, for scalability, if we have an all-in-one VM, we will have to replicate VMs which replicate all services.
- Are there any conflicting resource requirements? For example, different OS versions, JDK versions, and others.

Shared or Embedded Rules Engine?

- Rules are an essential part of any system.
- For example, an offer eligibility service may execute a number of rules before making a yes or no decision.
- Either we hand code rules, or we may use a rules engine.
- Many enterprises manage rules centrally in a rules repository as well as execute them centrally.
- These enterprise rule engines are primarily used for providing the business an opportunity to author and manage rules as well as reuse rules from the central repository.
- **Drools** is one of the popular open source rules engines.
- IBM, FICO, and Bosch are some of the pioneers in the commercial space.
- These rule engines improve productivity, enable reuse of rules, facts, vocabularies, and provide faster rule execution using the rete algorithm.

**Eligibility
Microservice**

*custom
rule engine*

**Eligibility
Microservice**

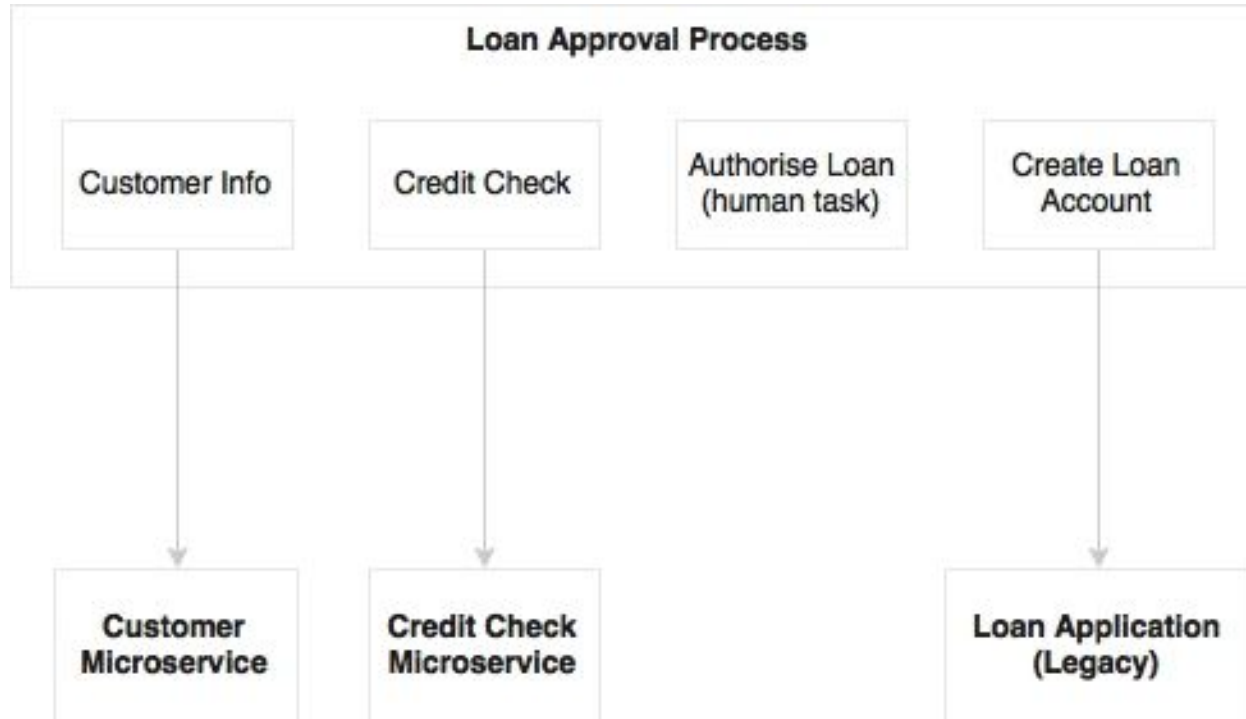
*Embedded
Rules Engine*



BPM and Workflows with Microservices

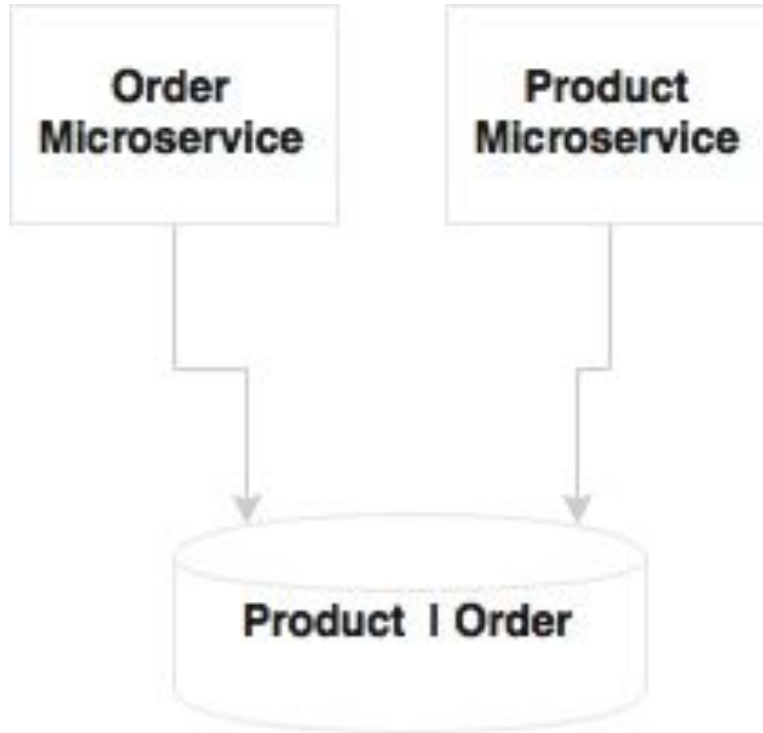
- Coordinating a long-running business process, where some processes are realized by existing assets, whereas some other areas may be niche, and there is no concrete implementation of the processes being in place. BPM allows composing both types, and provides an end-to-end automated process. This often involves systems and human interactions.
- Process-centric organizations, such as those that have implemented Six Sigma, want to monitor their processes for continuous improvement on efficiency.
- Process re-engineering with a top-down approach by redefining the business process of an organization.

BPM and Workflows

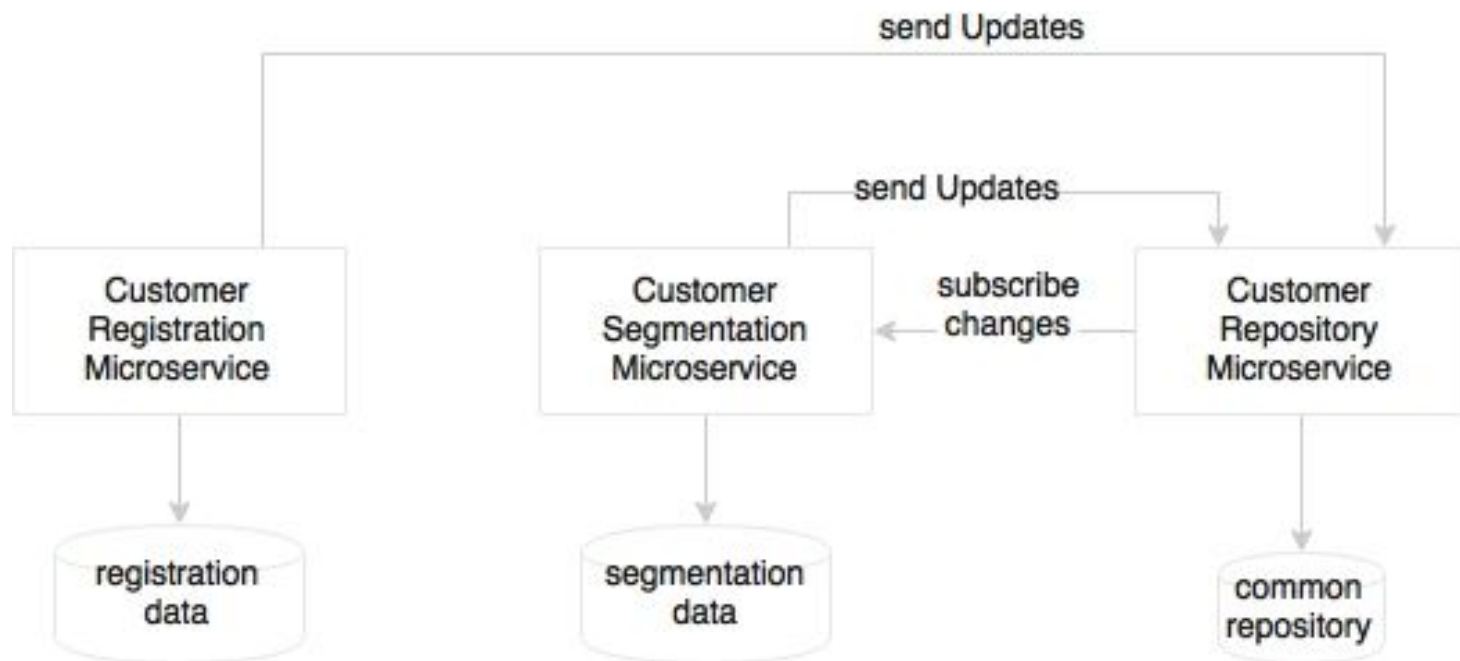




Share Data Stores with Microservices







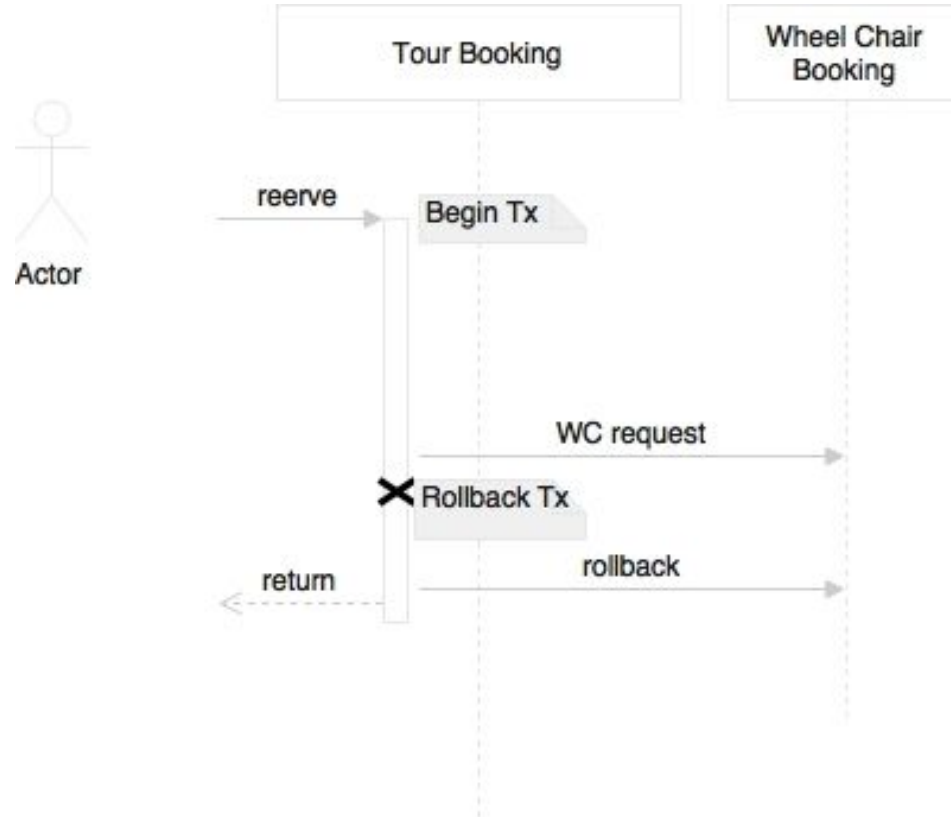
Transaction Boundaries

- Transactions in operational systems are used to maintain the consistency of data stored in an RDBMS by grouping a number of operations together into one atomic block.
- They either commit or rollback the entire operation.
- Distributed systems follow the concept of distributed transactions with a two-phase commit.
- This is particularly required if heterogeneous components such as an RPC service, JMS, and so on participate in a transaction.

Change the Use Case for Tx's with Microservices

- Eventual consistency is a better option than distributed transactions that span across multiple microservices.
- Eventual consistency reduces a lot of overheads, but application developers may need to re-think the way they write application code.
- This could include remodeling functions, sequencing operations to minimize failures, batching insert and modify operations, remodeling data structure, and finally, compensating operations that negate the effect.

Distributed Tx's



Service Endpoints

One of the important aspects of microservices is service design.

Service design has two key elements:

1. contract design
2. protocol selection.

Contract Design

- The first and foremost principle of service design is simplicity.
- The services should be designed for consumers to consume.
- A complex service contract reduces the usability of the service.
- The **KISS (Keep It Simple Stupid)** principle helps us to build better quality services faster, and reduces the cost of maintenance and replacement.
- The **YAGNI (You Ain't Gonna Need It)** is another principle supporting this idea. Predicting future requirements and building systems are, in reality, not future-proofed.
- This results in large upfront investment as well as higher cost of maintenance.

Consumer Driven Contracts

- CDC is a great idea that supports evolutionary design.
- when the service contract gets changed, all consuming applications have to undergo testing.
- This makes change difficult.
 - CDC helps in building confidence in consumer applications.
- CDC advocates each consumer to provide their expectation to the provider in the form of test cases so that the provider uses them as integration tests whenever the service contract is changed.

Protocol Selection

- In the SOA world, HTTP/SOAP, and messaging were kinds of default service protocols for service interactions.
- Microservices follow the same design principles for service interaction.
- Loose coupling is one of the core principles in the microservices world too.

Message Oriented Services

- If we choose an asynchronous style of communication, the user is disconnected, and therefore, response times are not directly impacted.
- We may use standard JMS or AMQP protocols for communication with JSON as payload. Messaging over HTTP is also popular, as it reduces complexity.
- Many new entrants in messaging services support HTTP-based communication.
- Asynchronous REST is also possible, and is handy when calling long-running services.

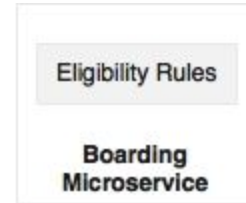
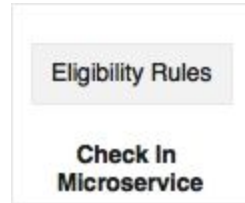
HTTP and REST endpoints

- Communication over HTTP is always better for interoperability, protocol handling, traffic routing, load balancing, security systems, and the like.
- Since HTTP is stateless, it is more compatible for handling stateless services with no affinity.
- Most of the development frameworks, testing tools, runtime containers, security systems, and so on are friendlier towards HTTP.

Optimized Communication Protocols

- If the service response times are stringent, then we need to pay special attention to the communication aspects.
- In such cases, we may choose alternate protocols such as Avro, Protocol Buffers, or Thrift for communicating between services.
- But this limits the interoperability of services.
- The trade-off is between performance and interoperability requirements.
- Custom binary protocols need careful evaluation as they bind native objects on both sides—consumer and producer.
- This could run into release management issues such as class version mismatch in Java-based RPC style communications.

Shared Libraries



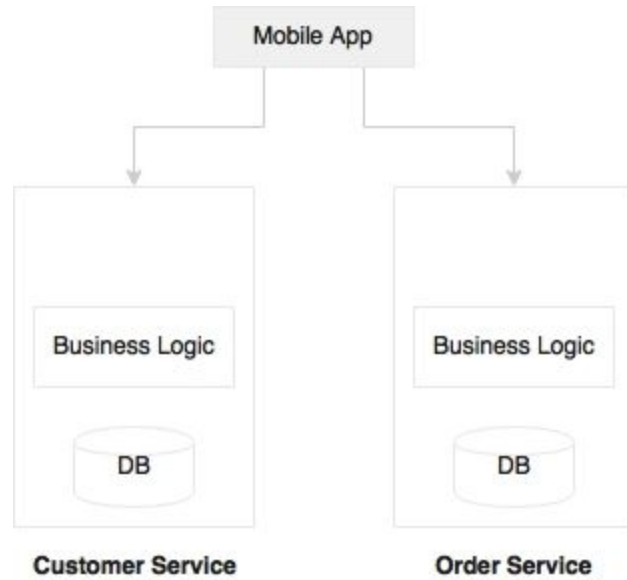
**Check In
Microservice**

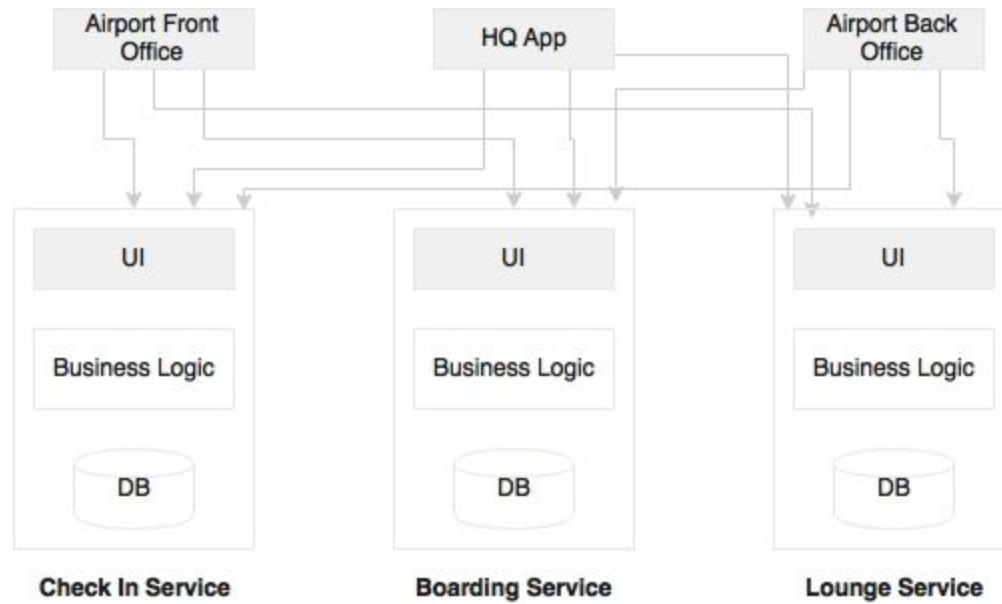
**Boarding
Microservice**

**Eligibility Rules
Microservices**

User Interfaces in Microservices







API Gateways in Microservices

- With the advancement of client-side JavaScript frameworks like AngularJS, the server is expected to expose RESTful services.
- This could lead to two issues.
 - The first issue is the mismatch in contract expectations.
 - The second issue is multiple calls to the server to render a page.
 - We start with the contract mismatch case. For example, `GetCustomer` may return a JSON with many fields:

```
Customer {  
  Name:  
  Address:  
  Contact:  
}
```

Customer {

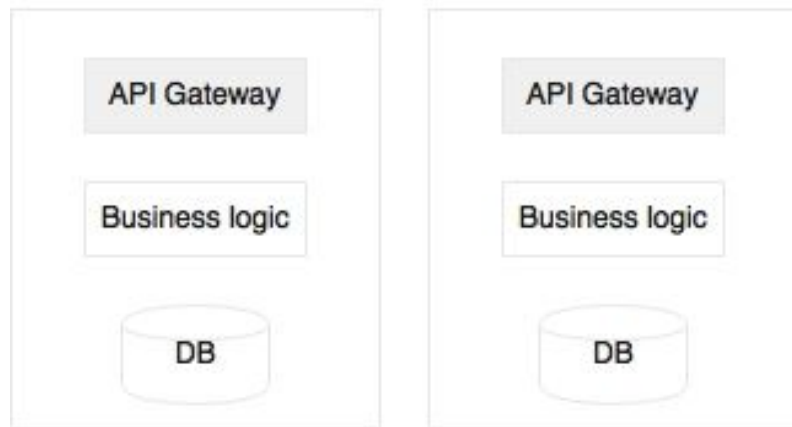
Id: 1

Name: /customer/name/1

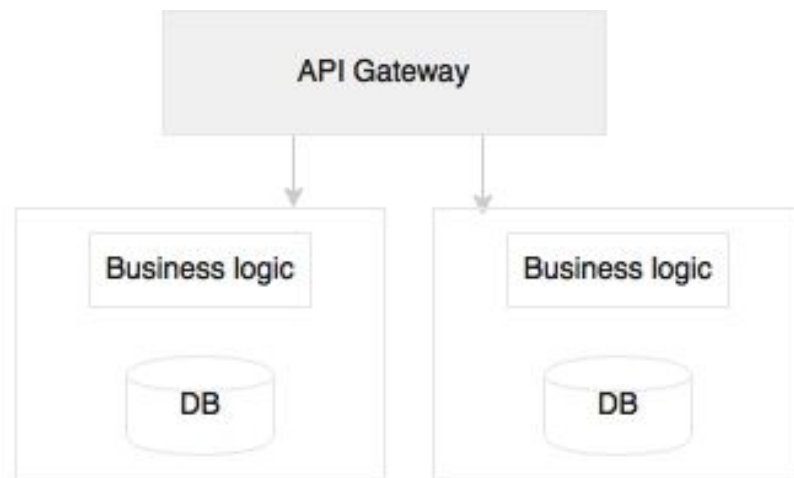
Address: /customer/address/1

Contact: /customer/contact/1

}



A) API gateway is part of the micro service



B) Common API gateway

Service Version Considerations

There are three different ways in which we can version REST services:

- URI versioning
- Media type versioning
- Custom header

5 - Reviewing BrownField's PSS Implementation

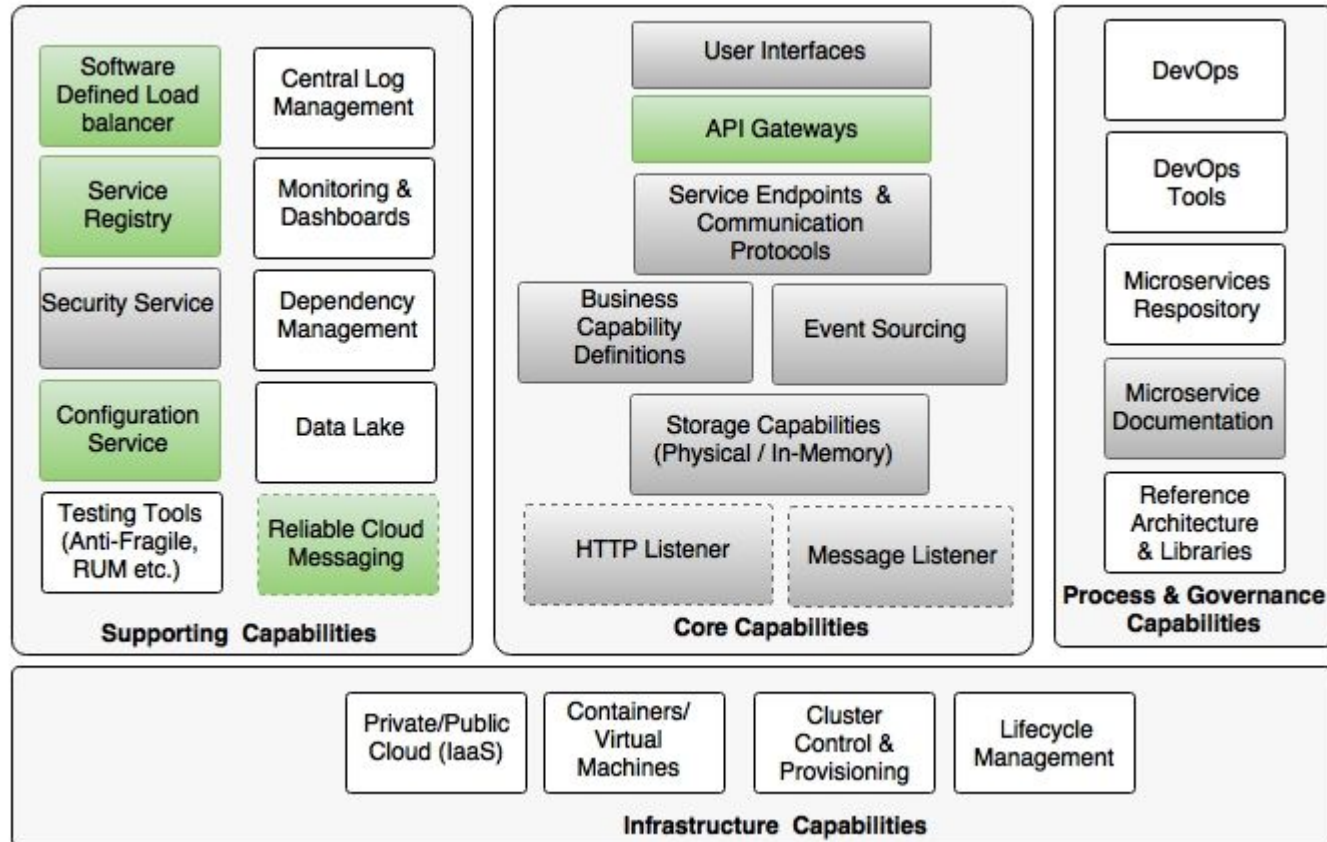
- Each microservice exposes a set of REST/JSON endpoints for accessing business capabilities
- Each microservice implements certain business functions using the Spring framework.
- Each microservice stores its own persistent data using H2, an in-memory database
- Microservices are built with Spring Boot, which has an embedded Tomcat server as the HTTP listener
- RabbitMQ is used as an external messaging service. Search, Booking, and Check-in interact with each other through asynchronous messaging
- Swagger is integrated with all microservices for documenting the REST APIs.
- An OAuth2-based security mechanism is developed to protect the microservices

Setting Up Environment for BrownField PSS

Scaling Microservices with Spring Cloud

- The Spring Config server for externalizing configuration
- The Eureka server for service registration and discovery
- The relevance of Zuul as a service proxy and gateway
- The implementation of automatic microservice registration...

Reviewing Microservices Capabilities



Reviewing BrownField's PSS Implementation

- Each microservice exposes a set of REST/JSON endpoints for accessing business capabilities
- Each microservice implements certain business functions using the Spring framework.
- Each microservice stores its own persistent data using H2, an in-memory database
- Microservices...

What is Spring Cloud?

- Implements a set of common patterns
- Not a cloud solution

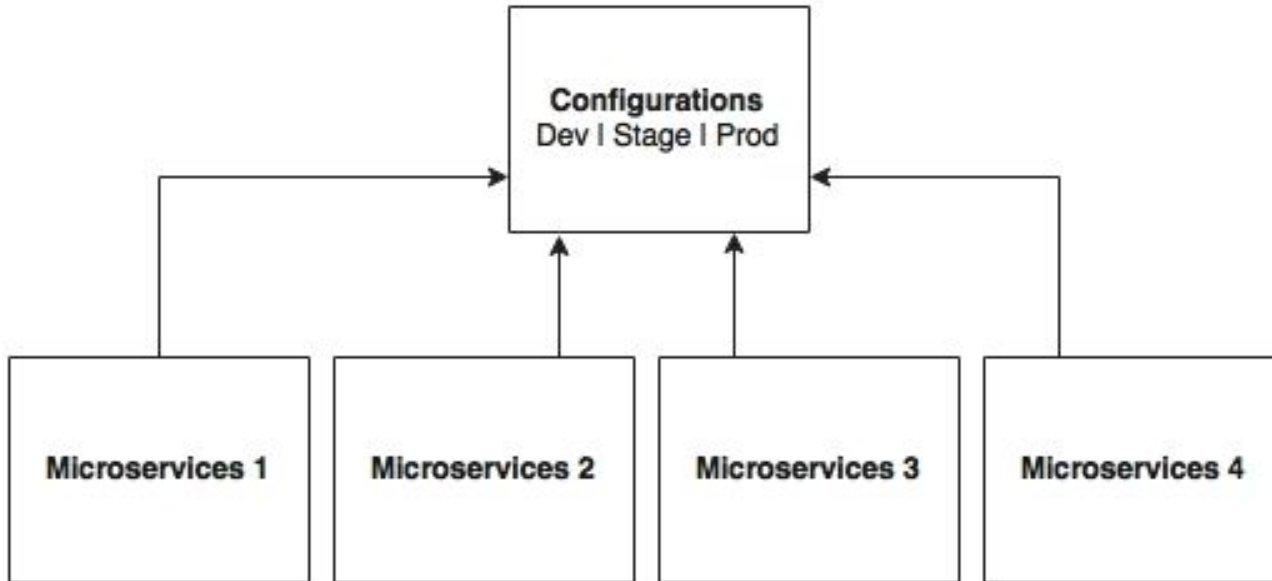
Setting up the environment for BrownField PSS

Spring Cloud Config

- From an external JNDI server using JNDI namespace (`java:comp/env`)
- Using the Java system properties (`System.getProperties()`) or using the `-D` command line option
- Using the `PropertySource` configuration:
- Copy
- ```
@PropertySource("file:${CONF_DIR}/application.properties")

public class ApplicationConfig {
 }
}
```
- Using a command-line parameter pointing a file to an external location:
- Copy
- ```
java -jar myproject.jar --spring.config.location=
```

Spring Cloud Config



Spring Cloud Config

Cloud Config

- ☐ Config Client

spring-cloud-config Client

- ☐ Config Server

Central management for configuration via a git or svn backend

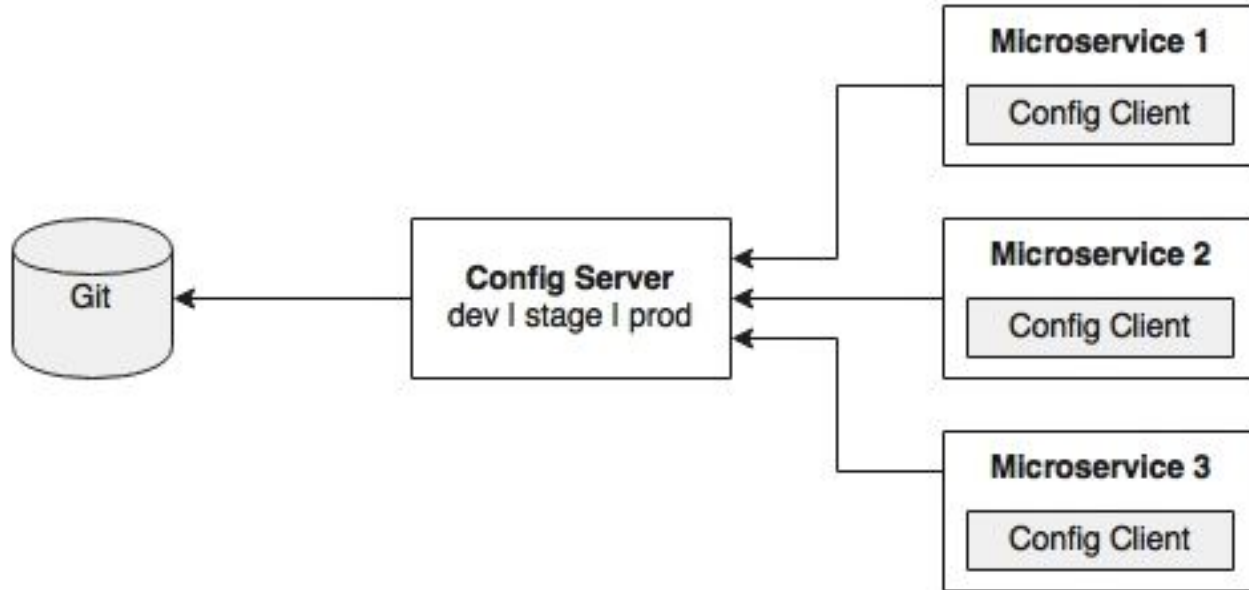
- ☐ Zookeeper Configuration

Configuration management with Zookeeper and spring-cloud-zookeeper-config

- ☐ Consul Configuration

Configuration management with Hashicorp Consul

Spring Cloud Config



Setting Up the Config Server

Lab 11 -

Accessing the Config Server from Clients

Lab 12 -

Handling configuration changes

Lab 13 -

Spring Cloud Bus for propagating configuration changes

Lab 14 -

Setting up the Eureka server

Lab 15 -

Setting up Zuul

Lab 16 -

Streams for Reactive Microservices

Lab 17 -

Summarizing the BrownField PSS Architecture

Lab 18 -

