

Developing Microservices

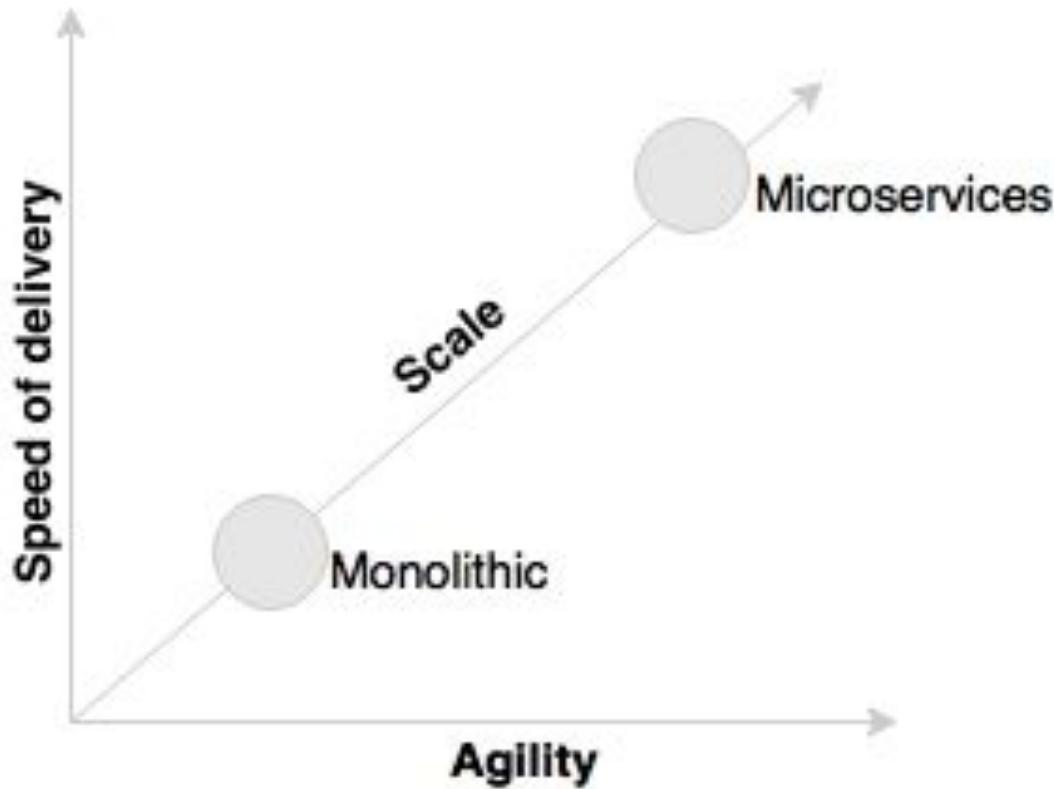
Demystifying Microservices

- The progression of microservices
- The definition of the microservices architecture with examples
- Concepts and characteristics of the microservices architecture
- Typical use cases of the microservices architecture
- The relationship of microservices with SOA and Twelve-Factor Apps

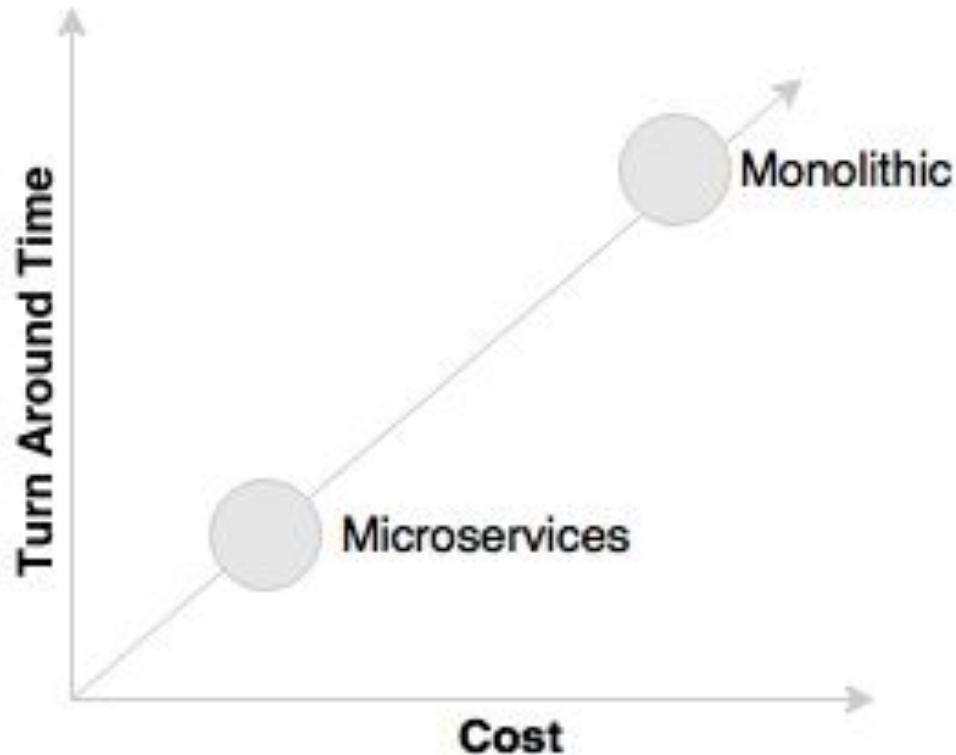
Progression of Microservices

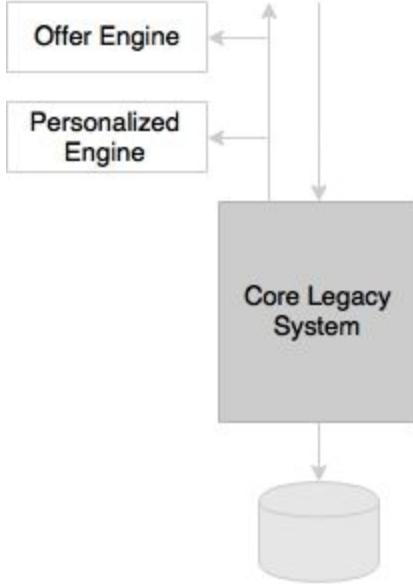
- Business demand as a catalyst for microservices evolution
- Technology as a catalyst for the microservices evolution

Business demand as a catalyst for microservices evolution

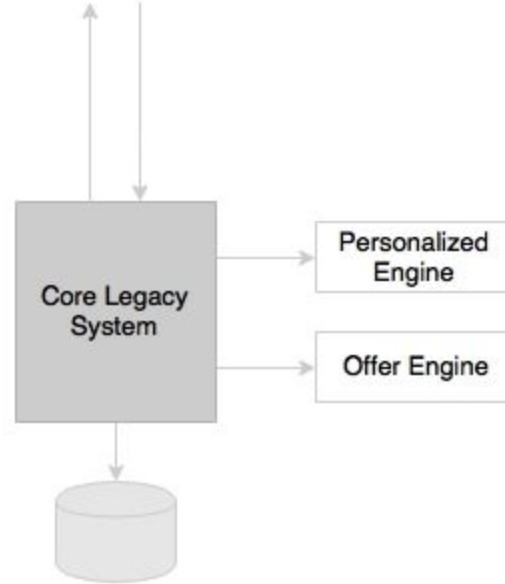


Business demand as a catalyst for microservices evolution





A) Response is intercepted to include new functions

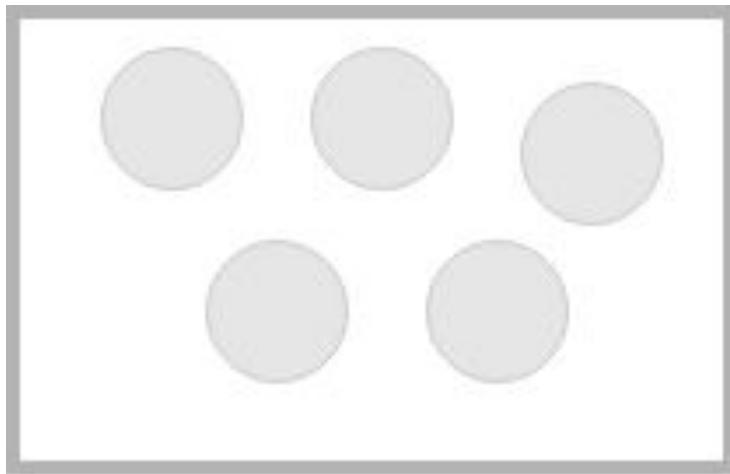


B) Core logic is rewritten to callout new functions

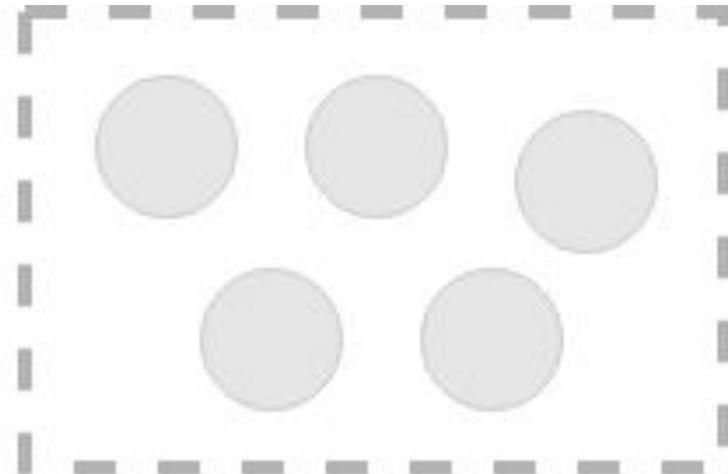
Technology as a catalyst for the microservices evolution

- Platform as a Services (PaaS)
- Integration Platform as a Service (iPaaS)

Progression of Architecture



**Monolithic
Architecture**

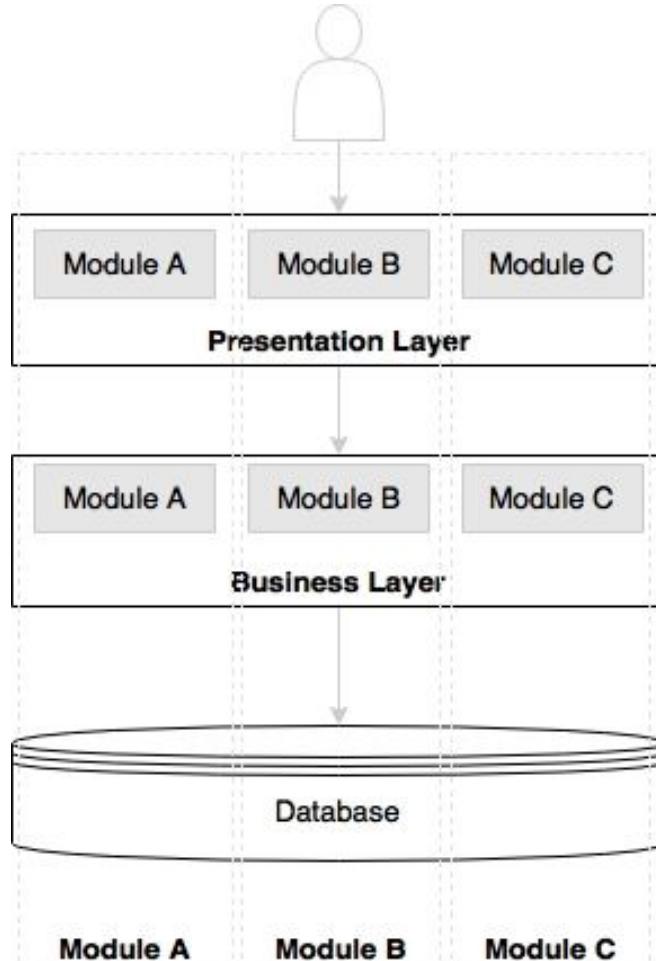


**Microservices
Architecture**

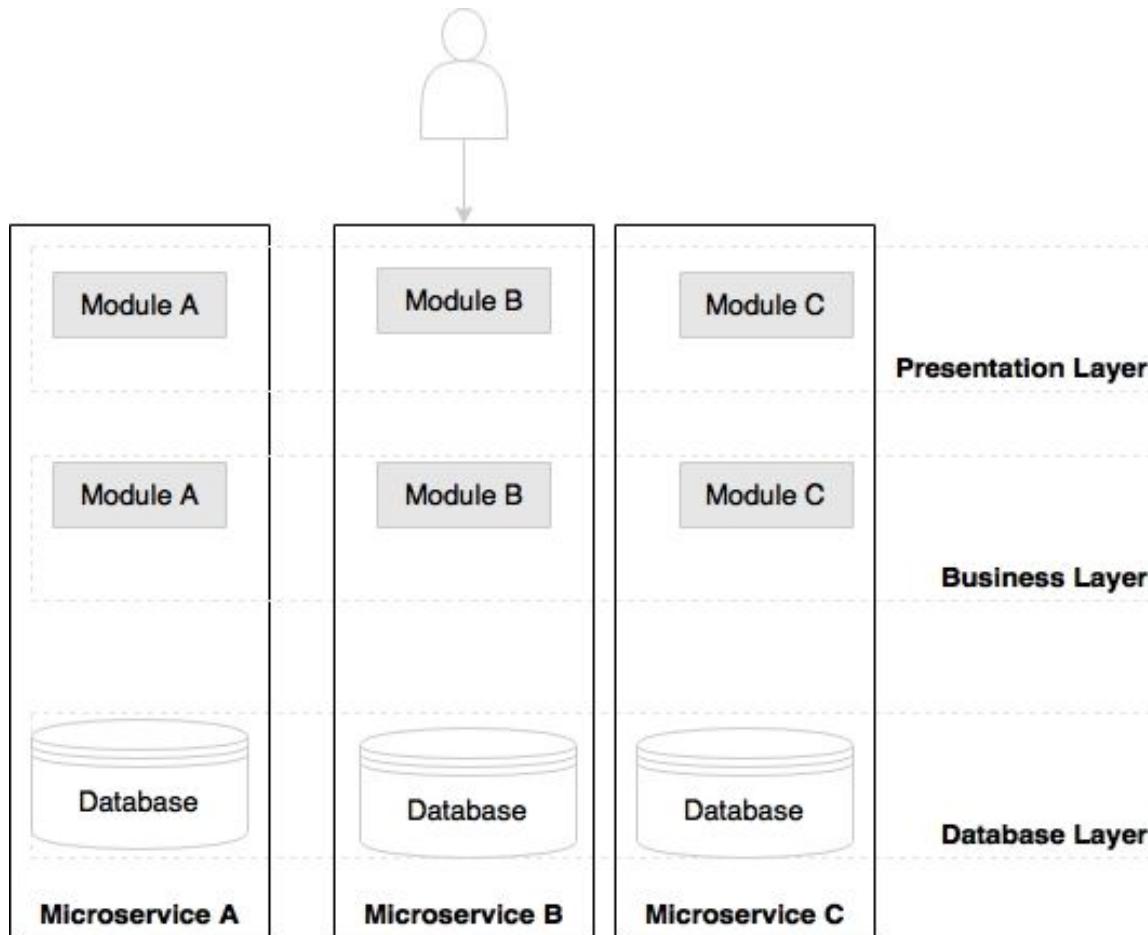
Microservices - What is it?

Architecture style

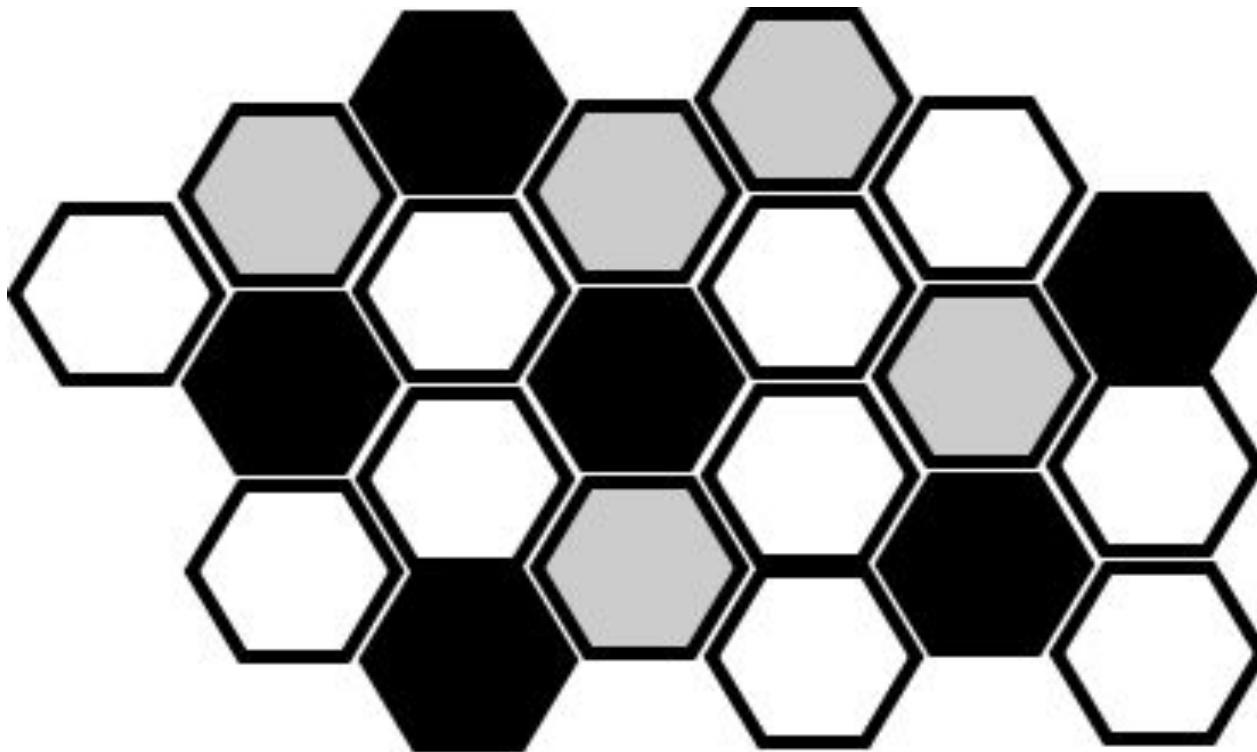
Microservices Architecture



Microservices Architecture



Microservices – the honeycomb analogy



Principles of Microservices

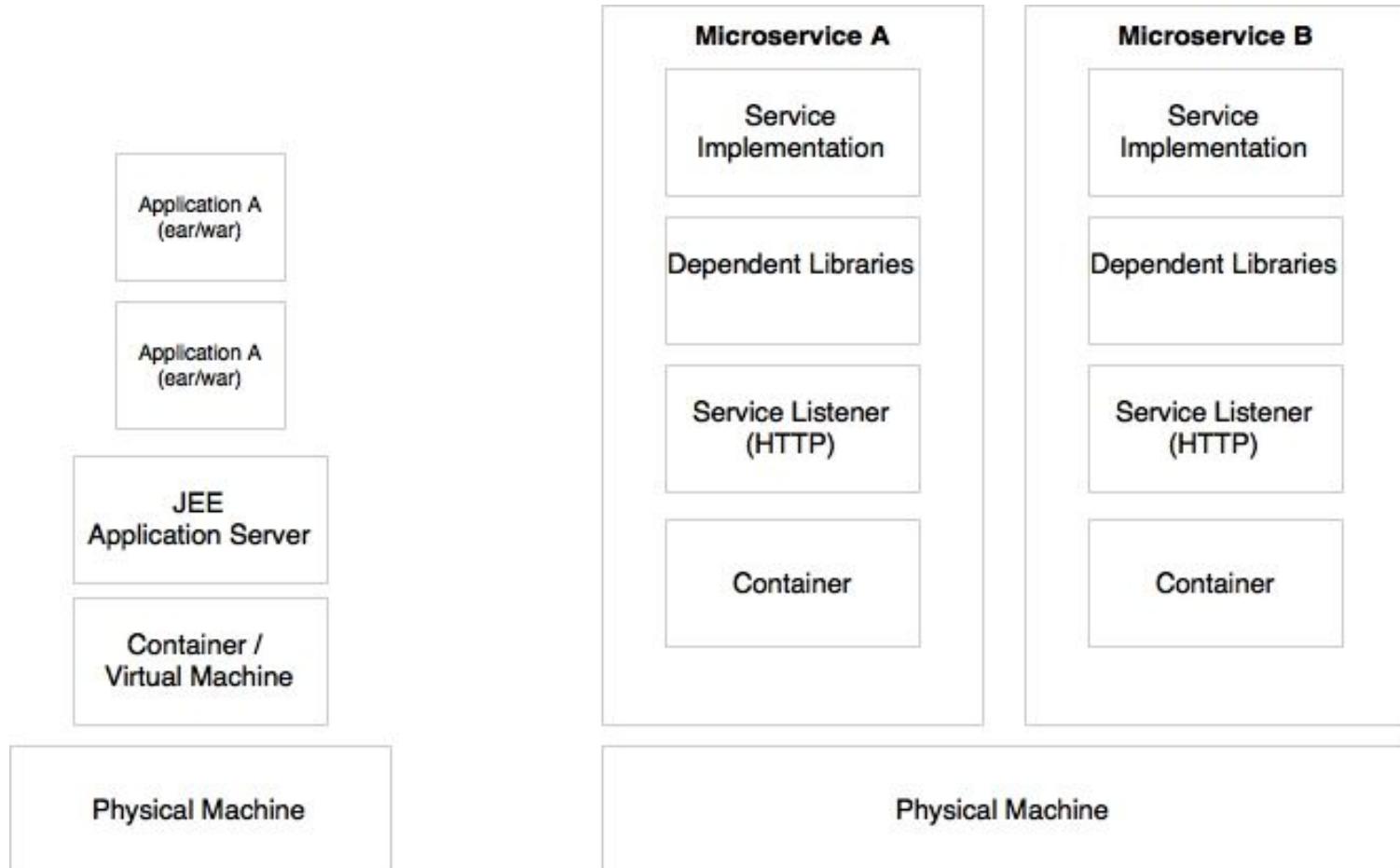
- Single responsibility per service
- Microservices are autonomous

Single responsibility per service



**Single Responsibility
Microservices**

Microservices are autonomous



Characteristics of microservices

- Services are first-class citizens
- Microservices are lightweight
- Microservices with polyglot architecture
- Automation in a microservices environment
- Microservices with a supporting ecosystem
- Microservices are distributed and dynamic
- Antifragility, fail fast, and self-healing

Services are first-class citizens

- Characteristics of services in a microservice
 - Service contract - JSON Schema, WADL, Swagger, and RAML are a few examples.
 - Loose coupling
 - Service abstraction
 - Service reuse
 - Statelessness
 - Services are discoverable
 - Service interoperability
 - Service composeability

Microservices are lightweight

Physical Machine

Virtual Machine

Application Server



All in one App EAR

Physical Machine

Docker Container



Microservice A

Docker Container



Microservice B

Docker Container



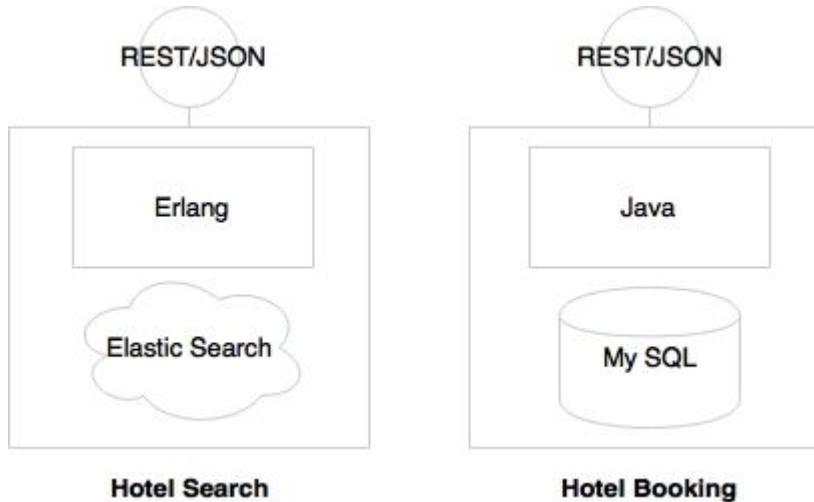
Microservice C

Traditional Deployment

Microservices Deployment

Microservices with polyglot architecture

- Different services use different versions of the same technologies. One microservice may be written on Java 1.7, and another one could be on Java 1.8.
- Different languages are used to develop different microservices, such as one microservice is developed in Java and another one in Scala.
- Different architectures are used, such as one microservice using the Redis cache to serve data, while another microservice could use MySQL as a persistent data store.

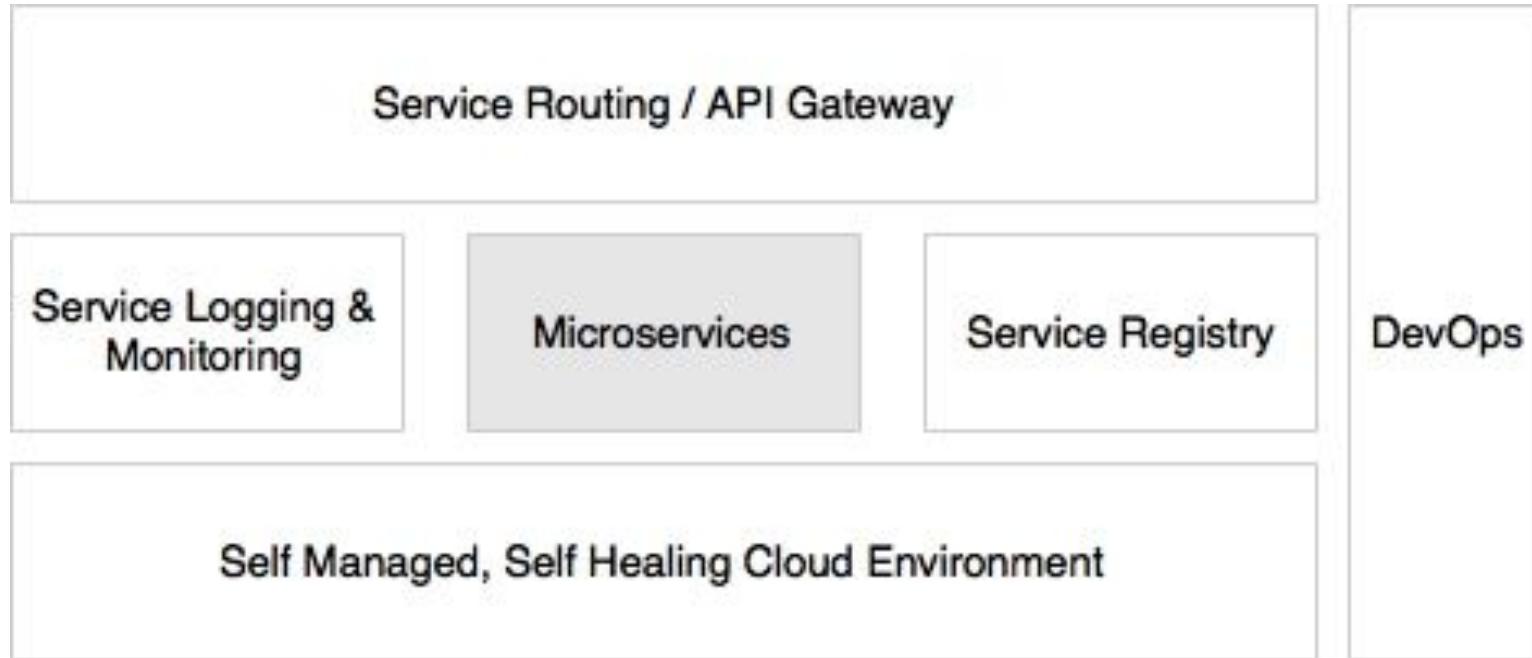


Automation in a microservices environment

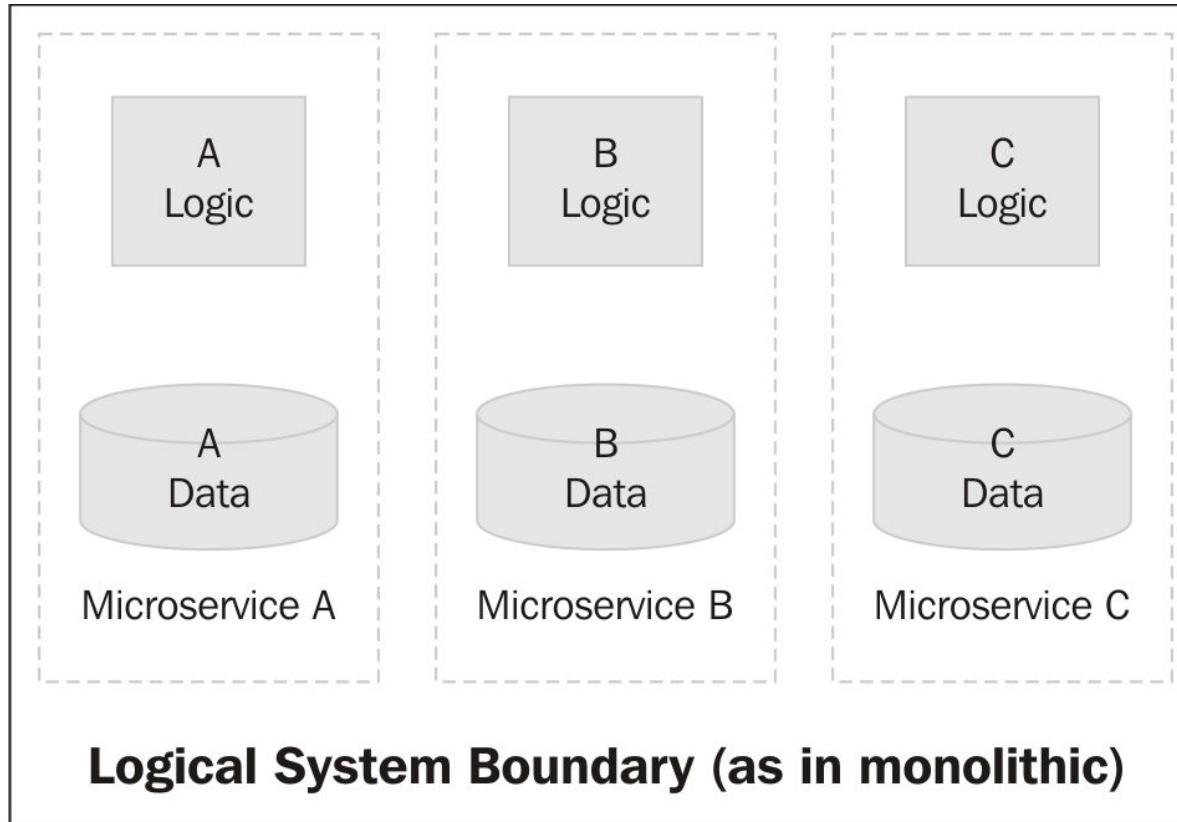


- The development phase
- The testing phase
- Infrastructure provisioning

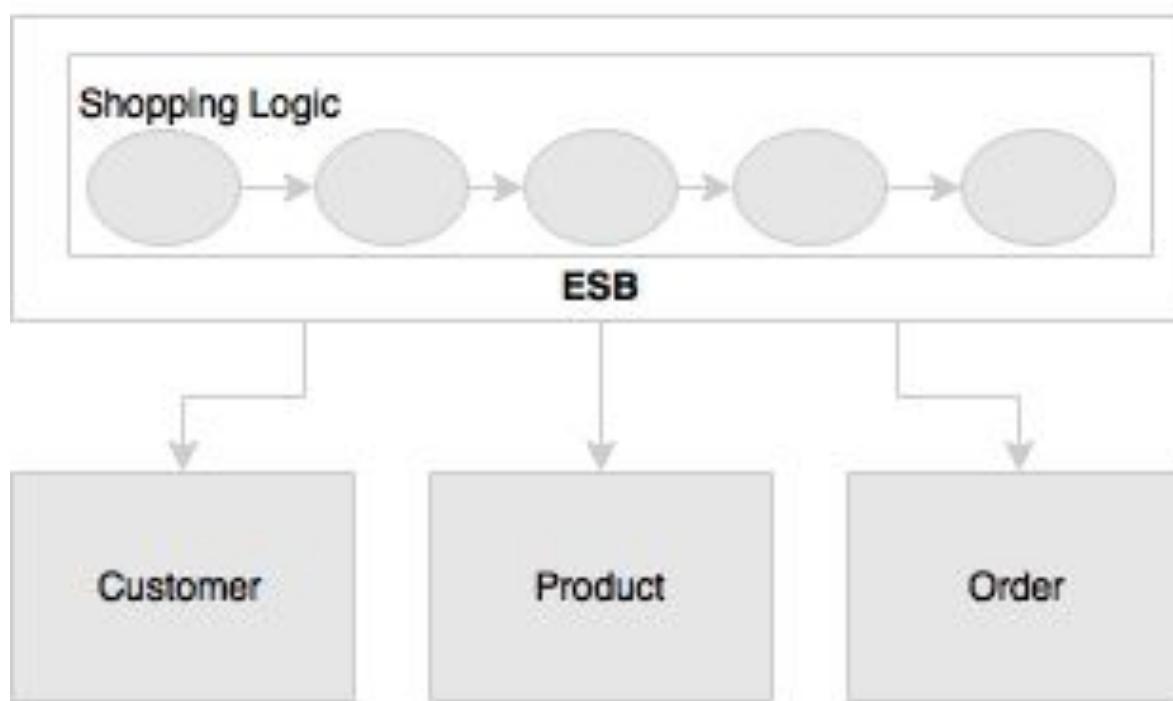
Microservices with a supporting ecosystem



Microservices are distributed and dynamic



Microservices are distributed and dynamic



Antifragility, fail fast, and self-healing

- Software systems are consistently challenged
- Fail fast is another concept used to build fault-tolerant, resilient systems

Microservices Examples

- Holiday Portal
- A microservice-based order management system
- A travel agent portal

Holiday Portal

The screenshot shows a web browser window for the 'flybypoints' website at <https://www.flybypoints.com>. The page title is 'Fly By Points'. A user profile for 'Jeo' is visible. The main content area is titled 'Welcome to Fly By Points Services' and features three cards: 'Points' (21123), 'Offers' (4), and 'Trips' (2). A navigation bar at the bottom says 'Destinations and More...'. The browser interface includes standard navigation buttons and a search bar.

flybypoints

https://www.flybypoints.com

Fly By Points

Jeo

Welcome to Fly By Points Services

21123

4

2

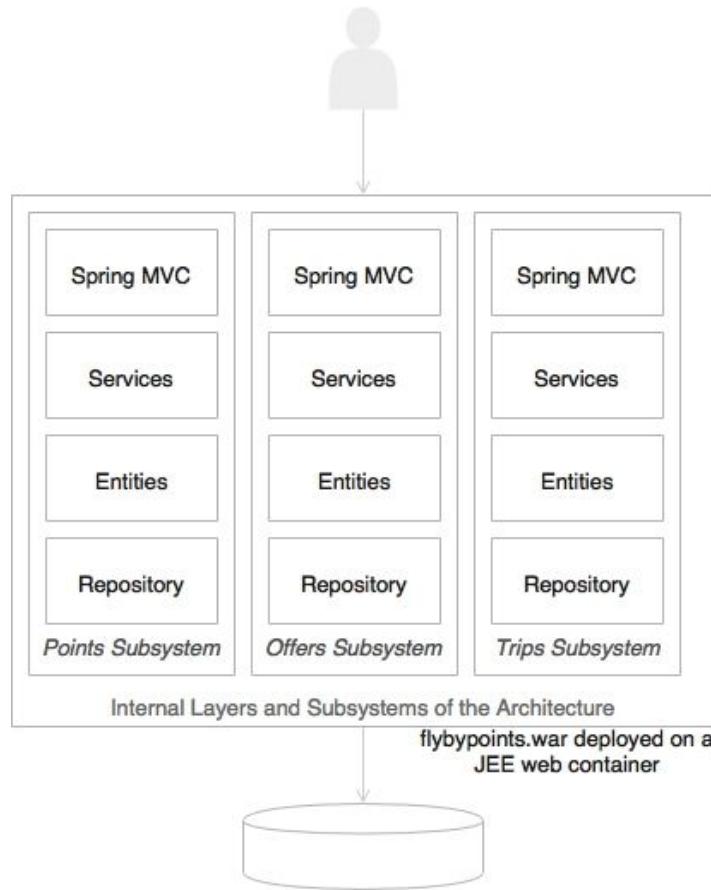
Points

Offers

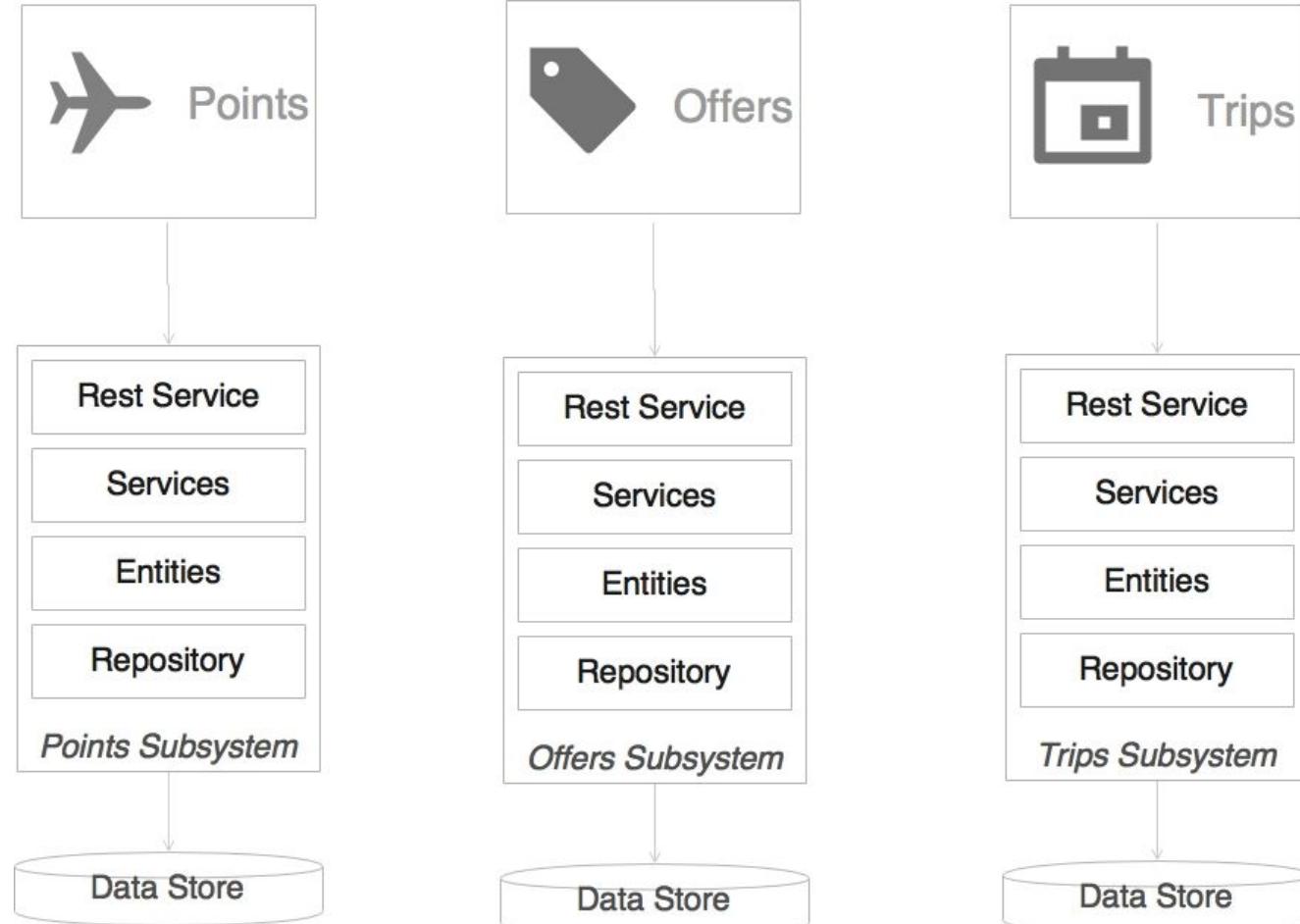
Trips

Destinations and More...

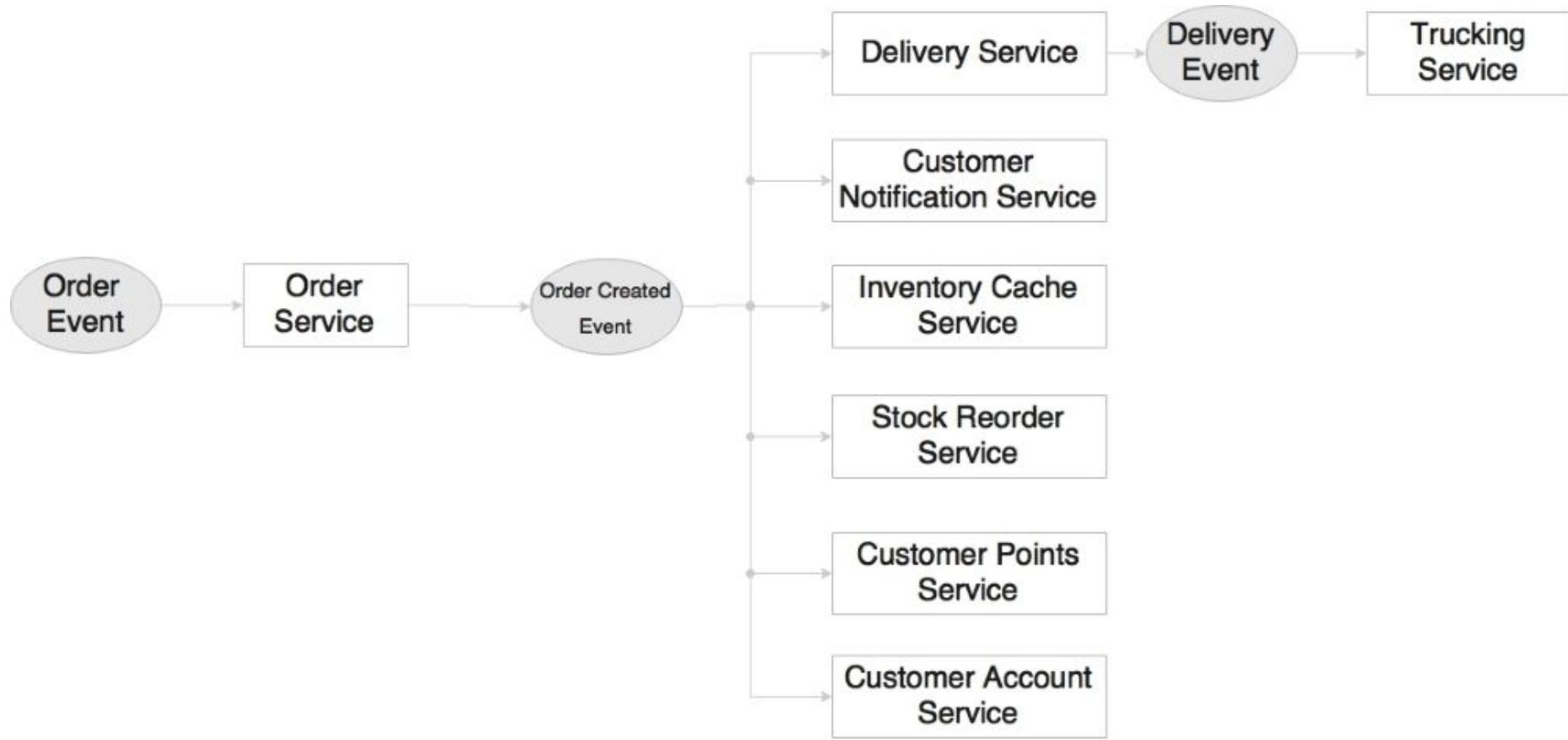
Holiday Portal



Holiday Portal



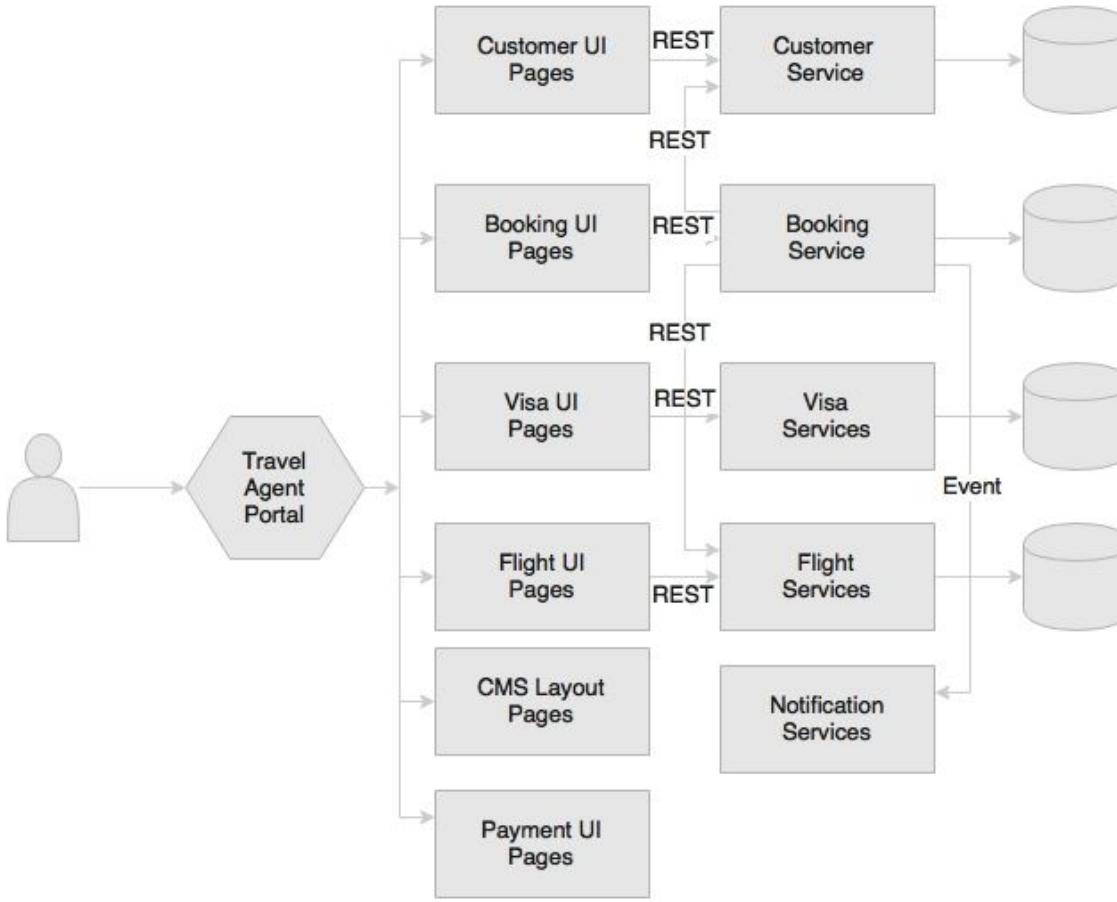
A microservice-based order management system



A microservice-based order management system

1. Order Service kicks off when Order Event is received. Order Service creates an order and saves the details to its own database.
2. If the order is successfully saved, Order Successful Event is created by Order Service and published.
3. A series of actions take place when Order Successful Event arrives.
4. Delivery Service accepts the event and places Delivery Record to deliver the order to the customer. This, in turn, generates Delivery Event and publishes the event.
5. Trucking Service picks up Delivery Event and processes it. For instance, Trucking Service creates a trucking plan.
6. Customer Notification Service sends a notification to the customer informing the customer that an order is placed.
7. Inventory Cache Service updates the inventory cache with the available product count.
8. Stock Reorder Service checks whether the stock limits are adequate and generates Replenish Event if required.
9. Customer Points Service recalculates the customer's loyalty points based on this purchase.
10. **Customer Account Service** updates the order history in the customer's account.

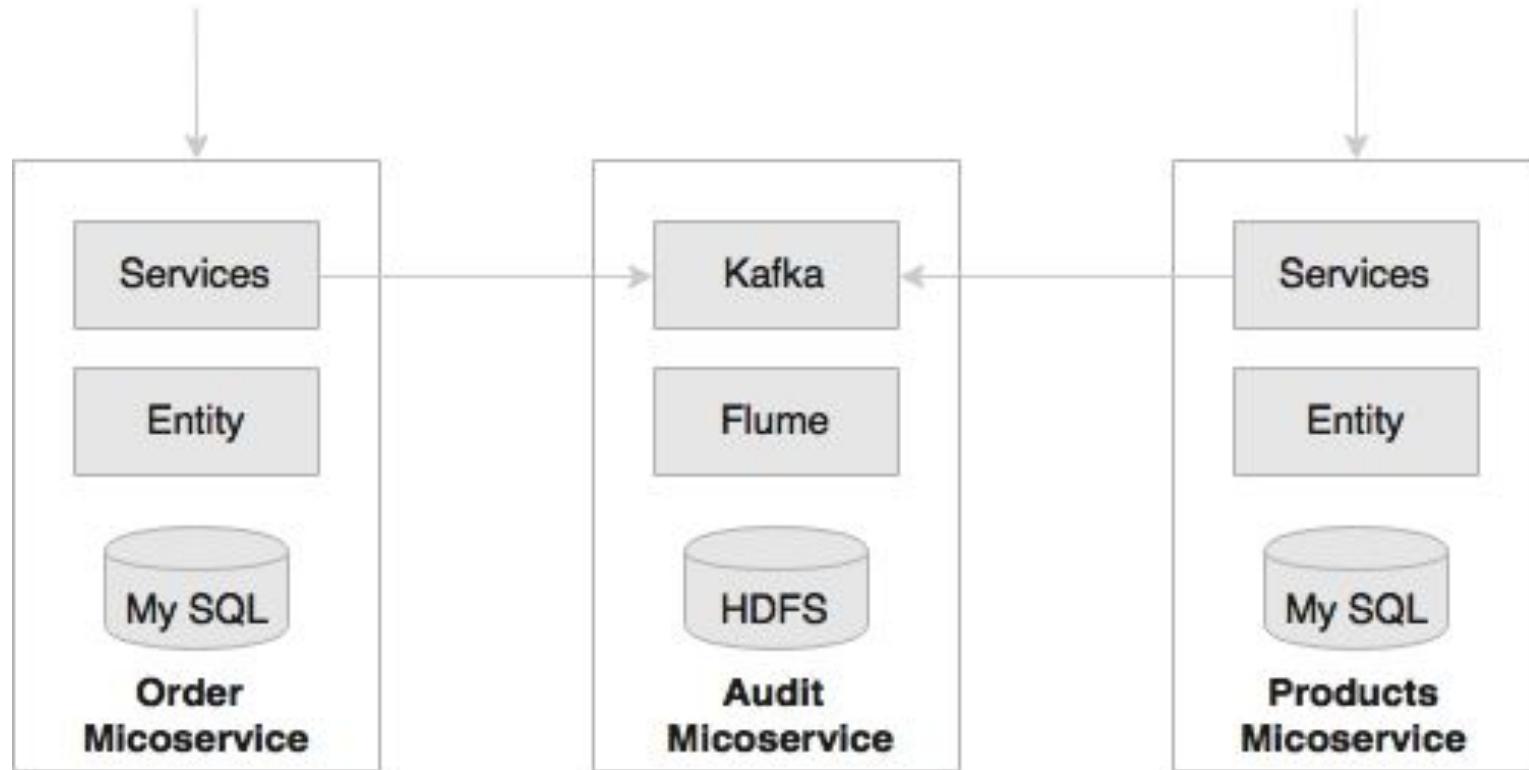
Travel agent portal



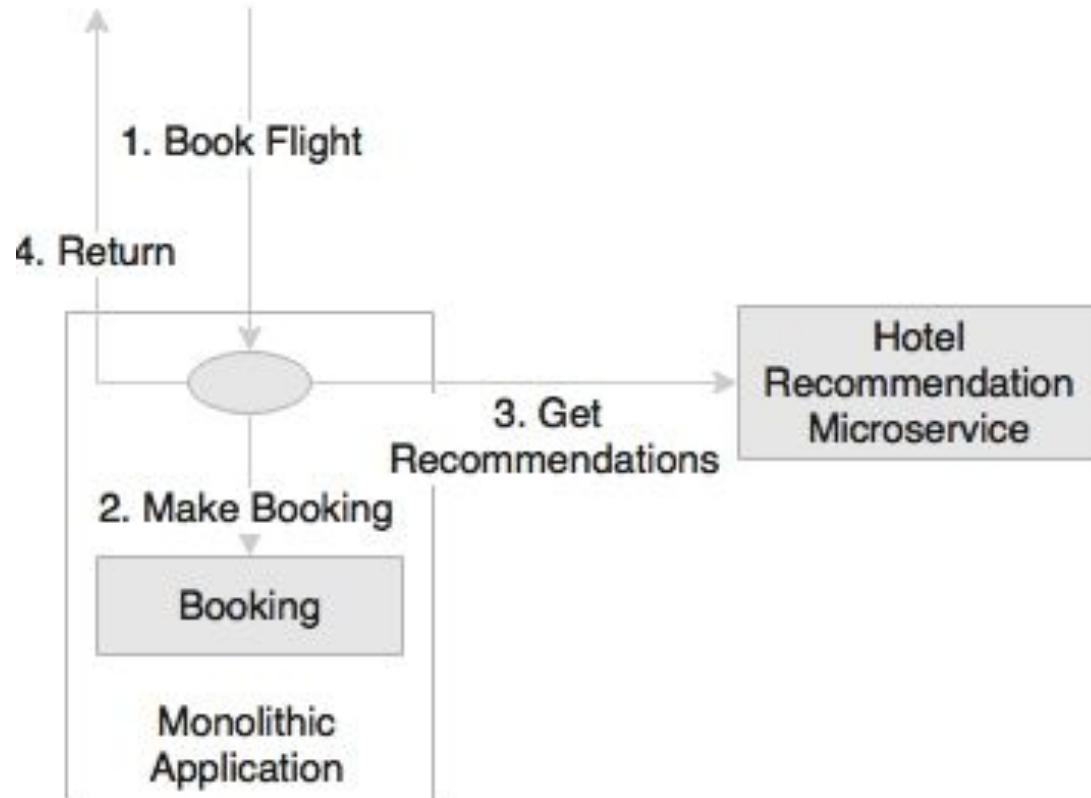
Microservices benefits

- Supports polyglot architecture
- Enabling experimentation and innovation
- Elastically and selectively scalable
- Allowing substitution
- Enabling to build organic systems
- Helping reducing technology debt
- Allowing the coexistence of different versions
- Supporting the building of self-organizing systems
- Supporting event-driven architecture
- Enabling DevOps

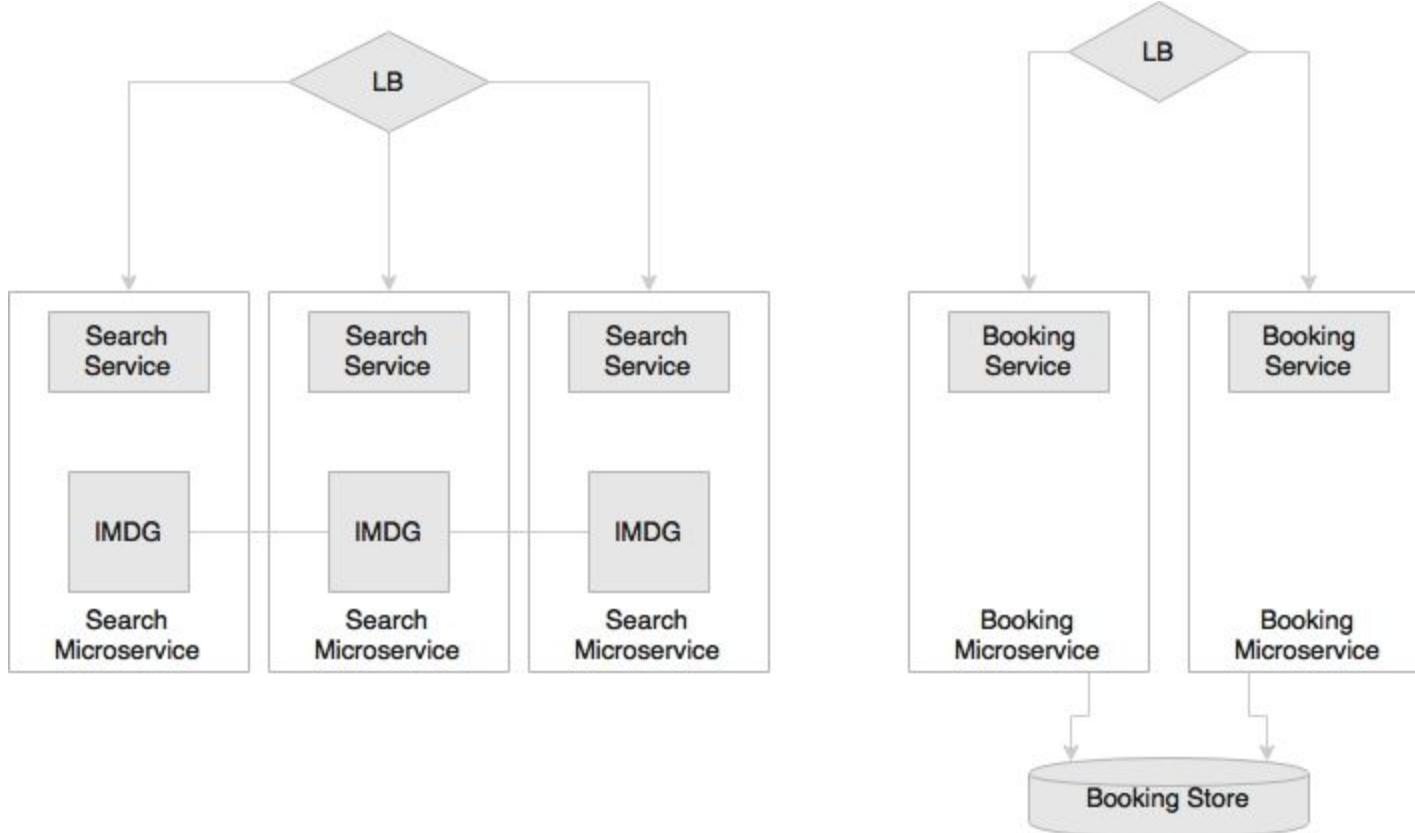
Supports polyglot architecture



Enabling experimentation and innovation



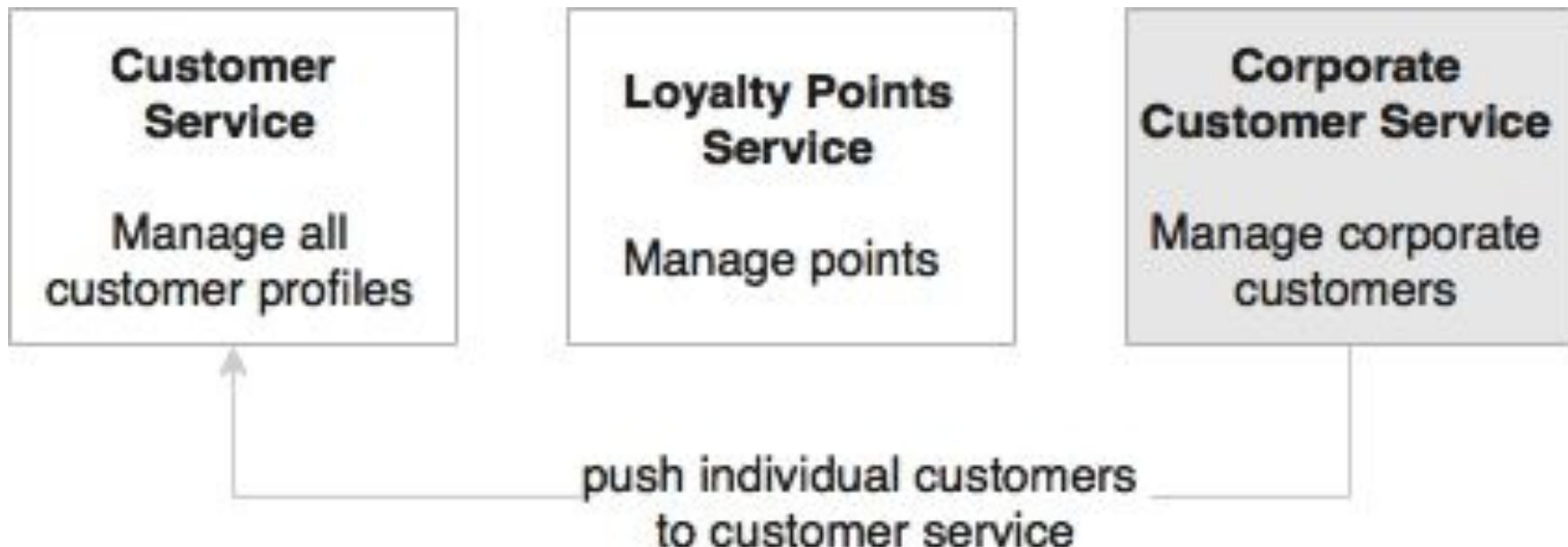
Elastically and selectively scalable



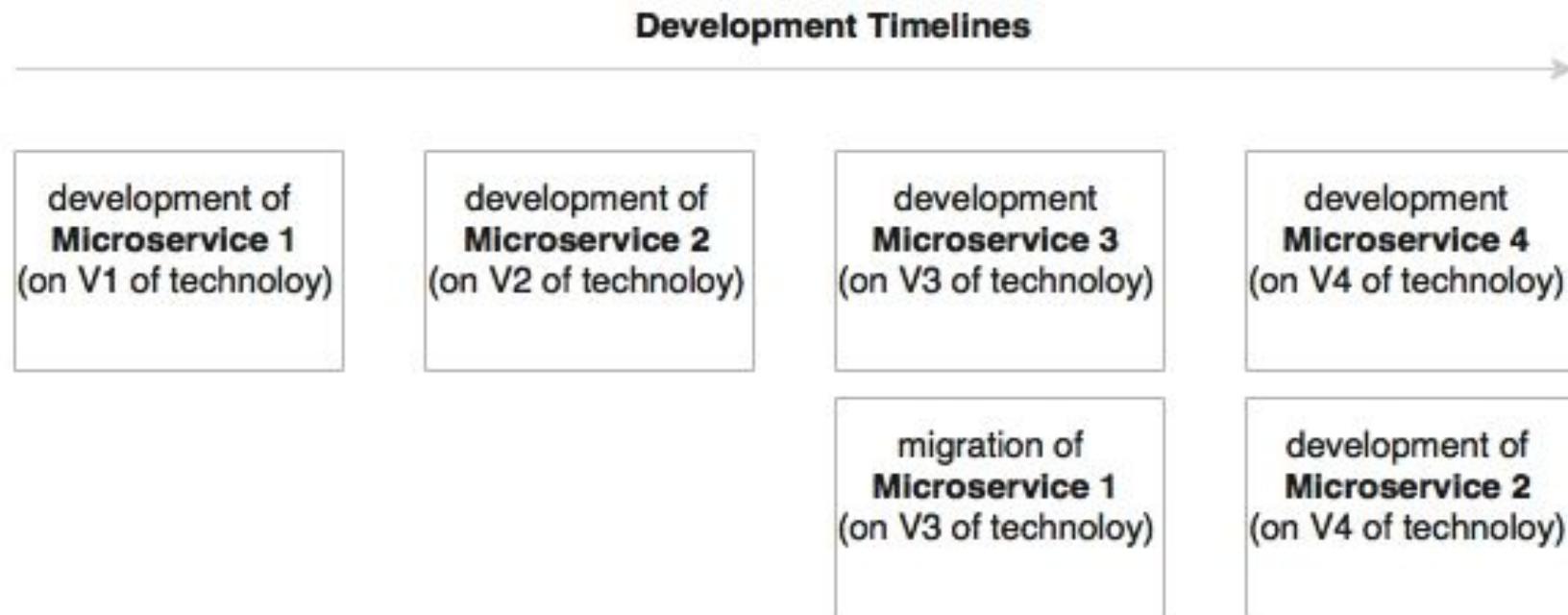
Allowing substitution



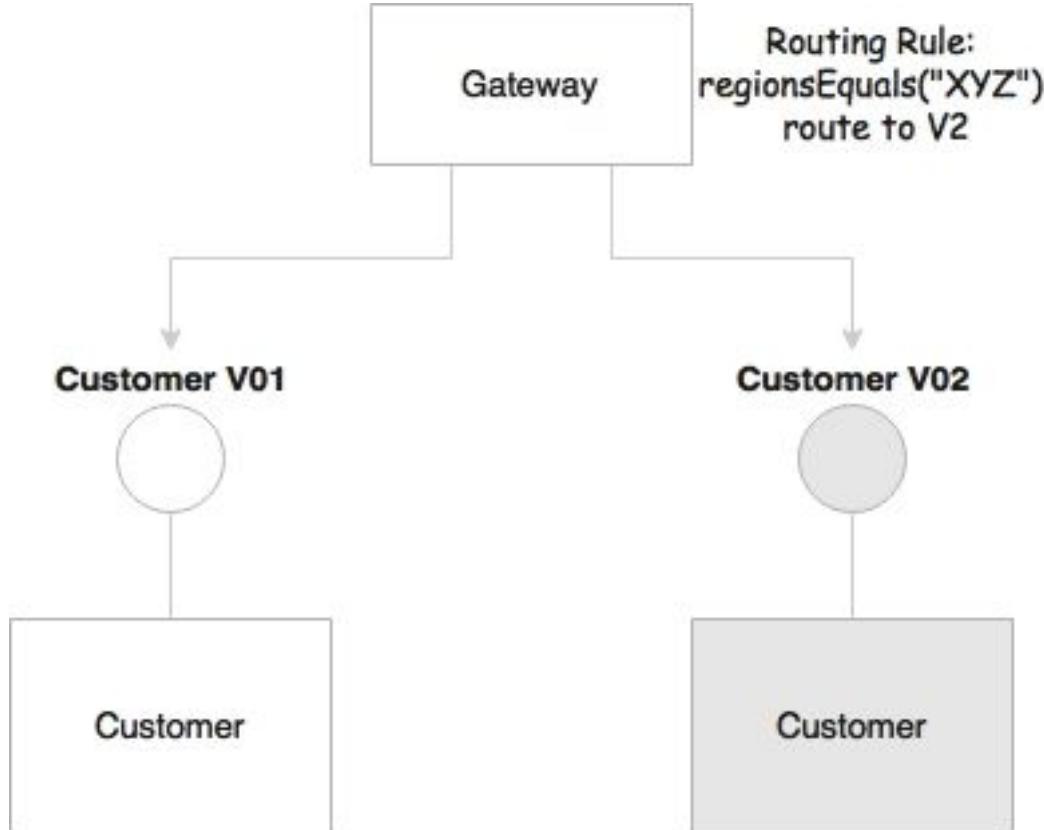
Enabling to build organic systems



Helping reduce technology debt



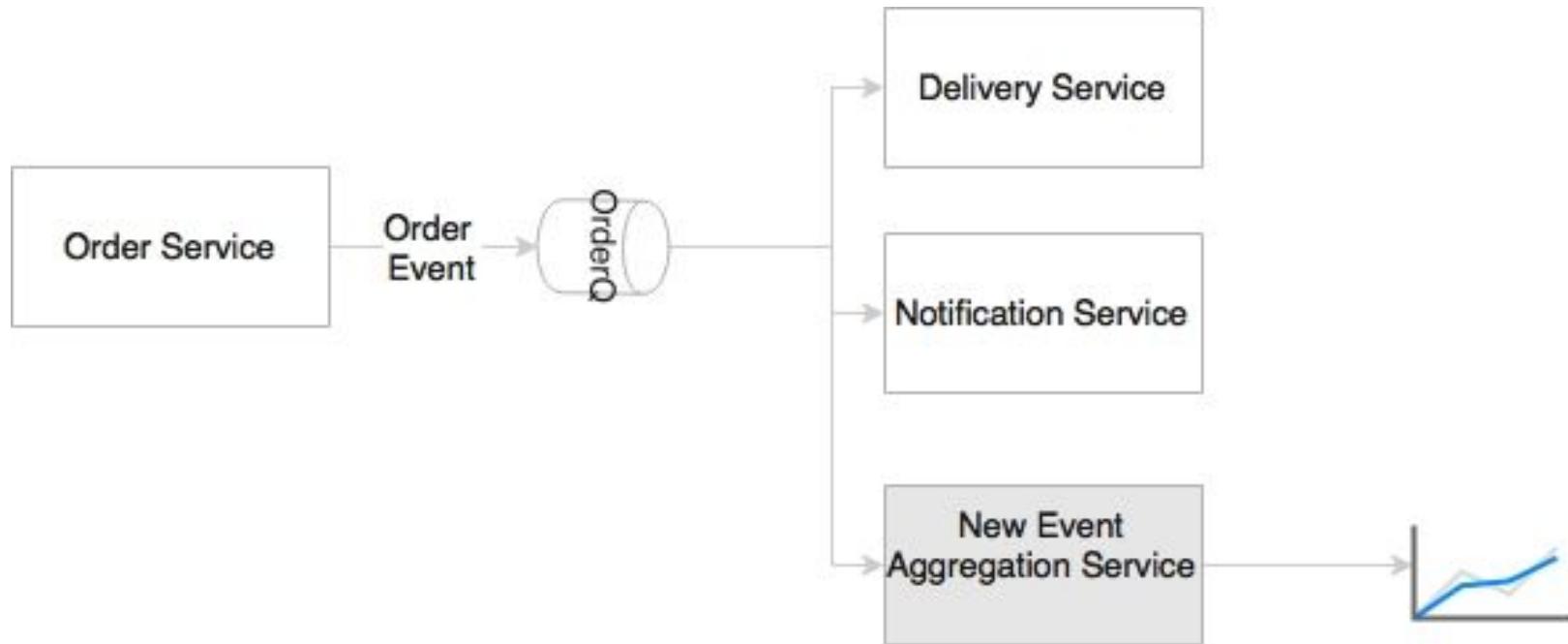
Allowing the coexistence of different versions



Supporting the building of self-organizing systems



Supporting event-driven architecture



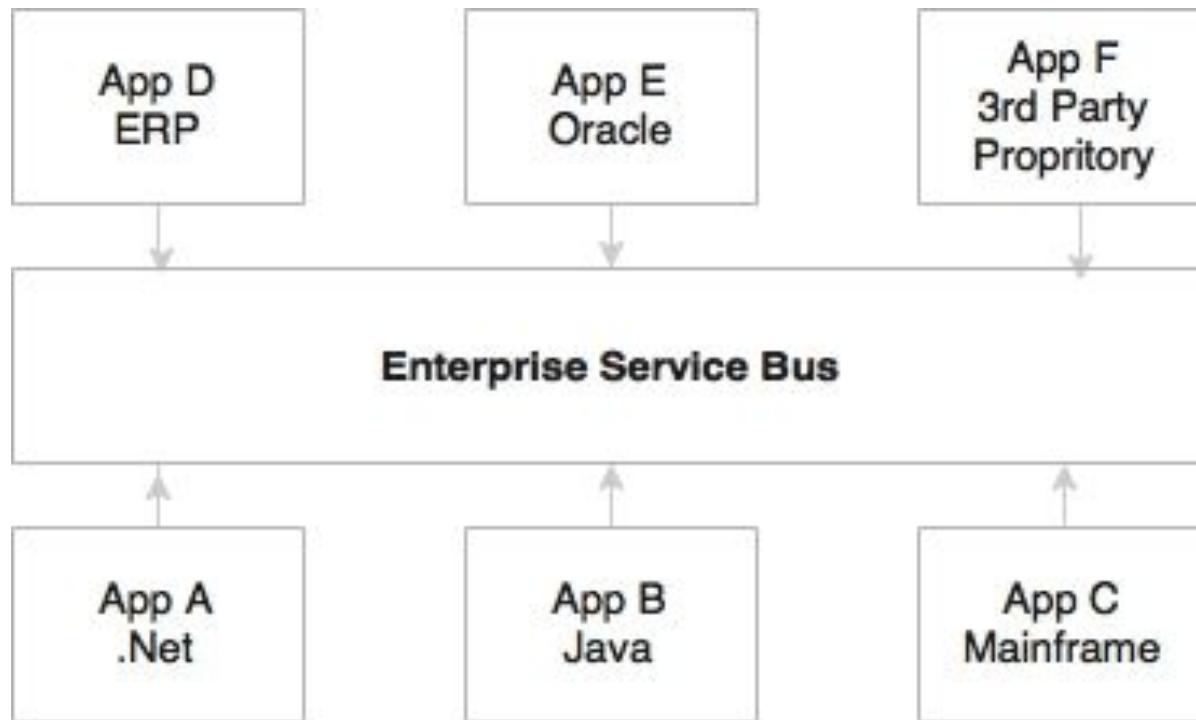
Enabling DevOps

- Microservices is a key component of DevOps
- Microservices is at the center of many DevOps implementation

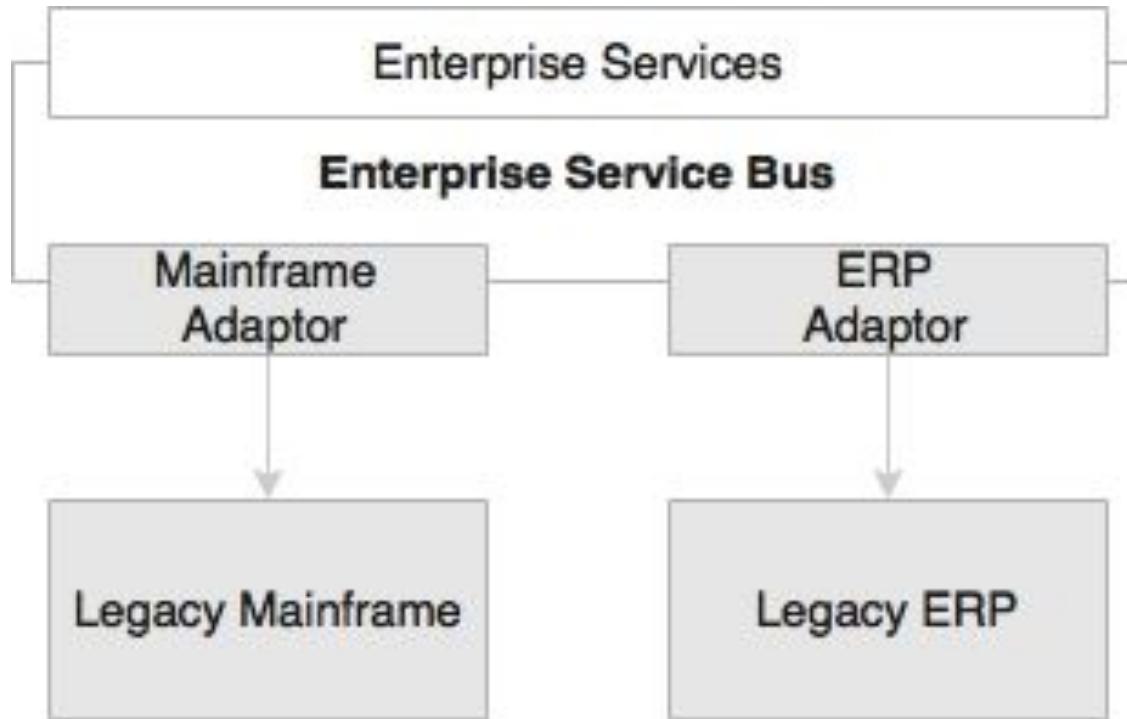
Microservices Relationship with SOA

- Service-oriented Integration
- Legacy modernization
- Service-oriented Application
- Monolithic migration using SOA

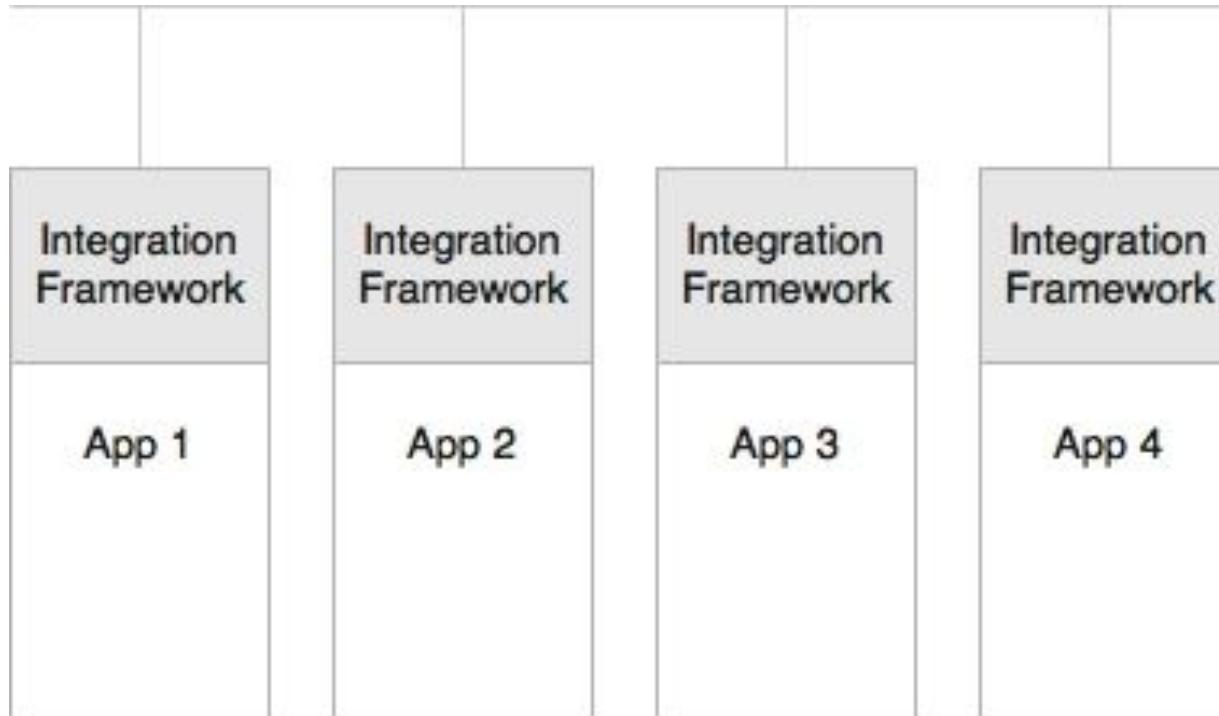
Service-oriented Integration



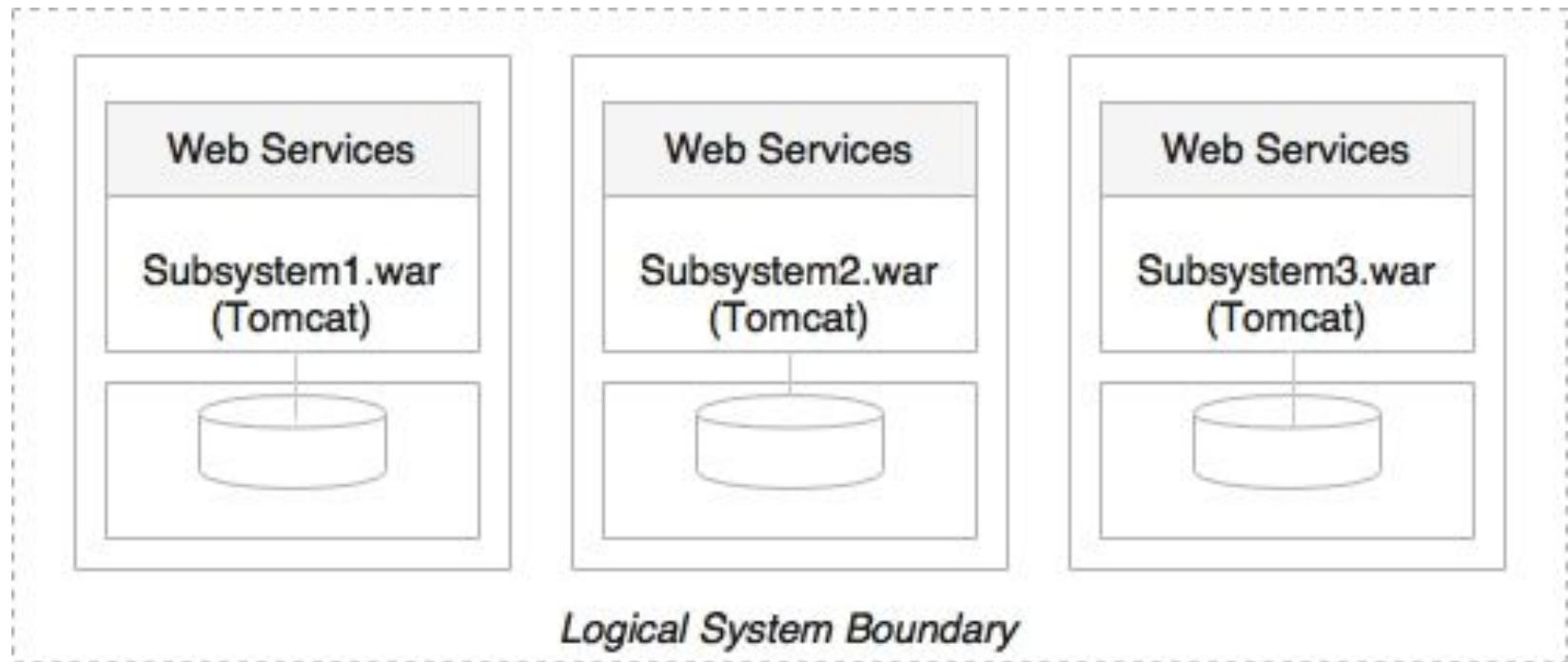
Legacy modernization



Service-oriented Application



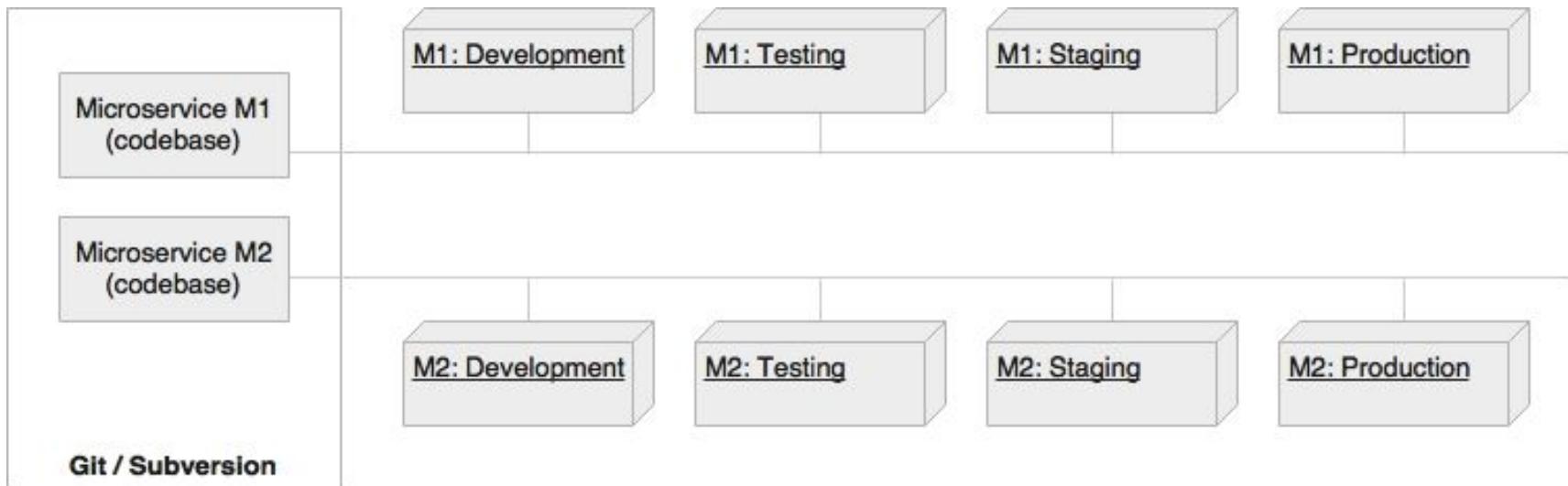
Monolithic migration using SOA



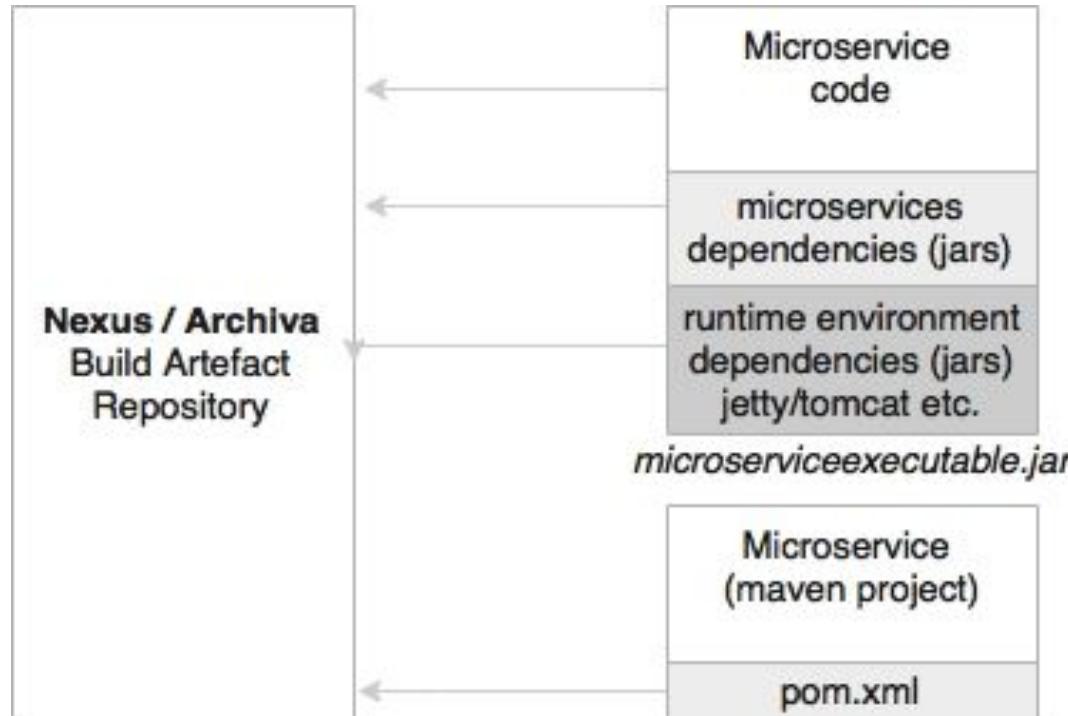
Microservices Relationship with Twelve Factor apps

- Single Code Base
- Bundling Dependencies
- Externalizing Configurations
- Backing Services are Addressable
- Isolation Between Build, Release and Run
- Stateless Shared Nothing Processes
- Exposing Services Through Port Bindings
- Concurrency to Scale Out
- Disposability with Minimal Overhead
- Development and Production Parity
- Externalizing Logs
- Package Admin Processes

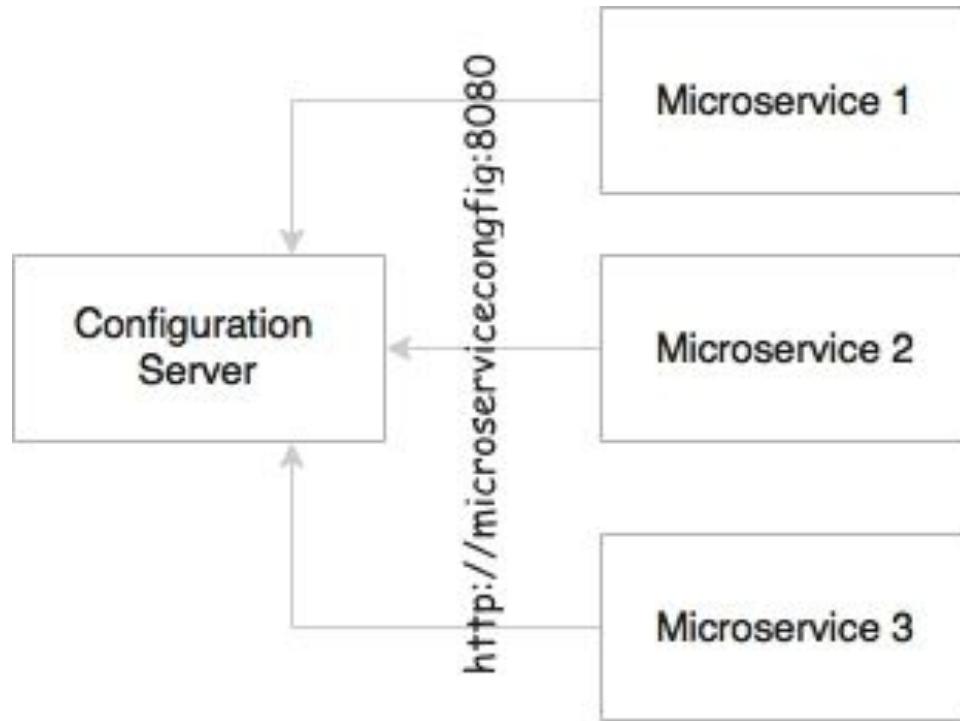
Single Code Base



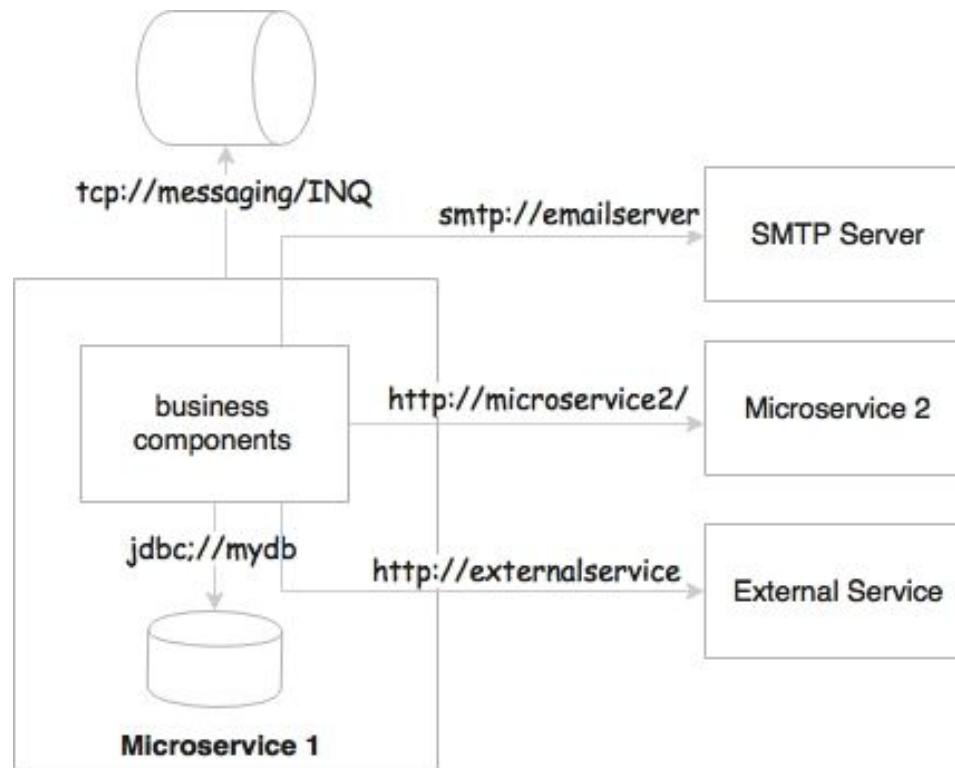
Bundling Dependencies



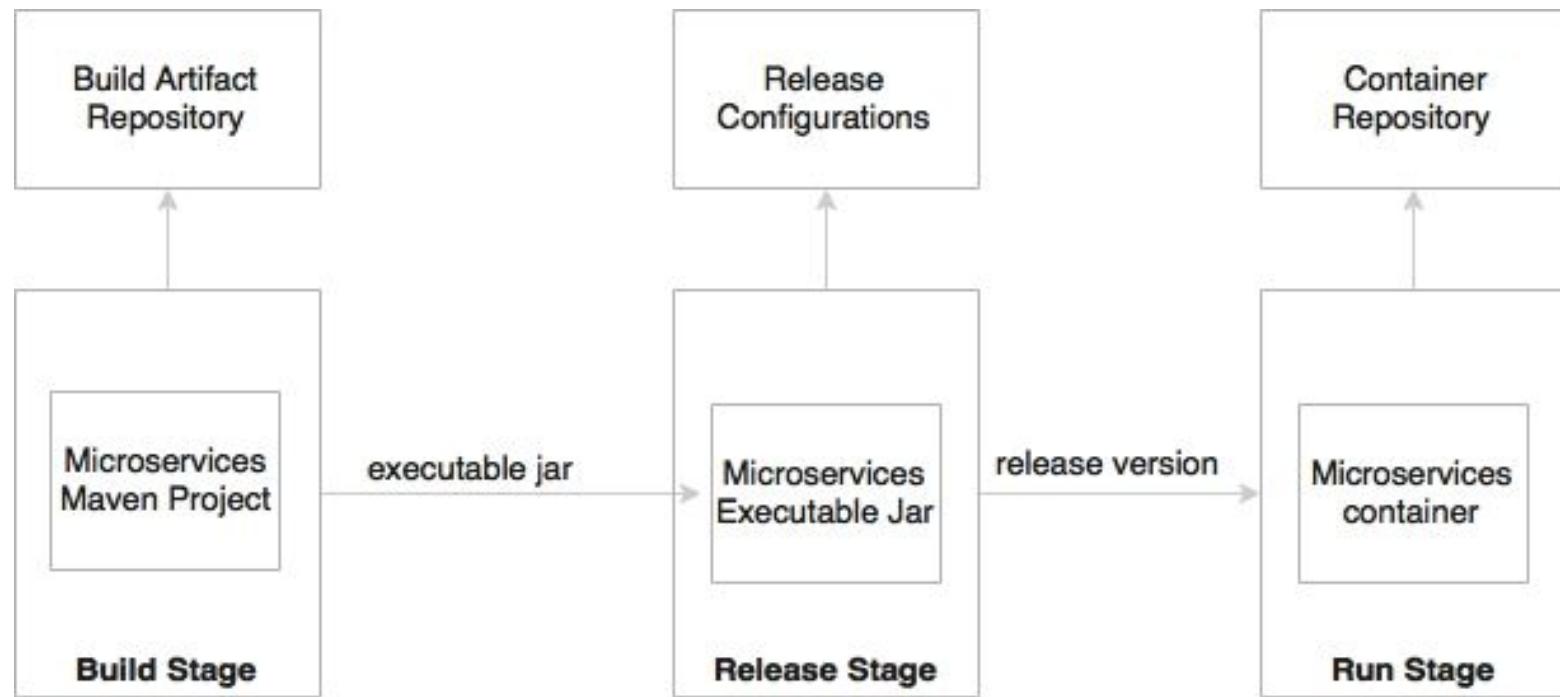
Externalizing Configurations



Backing Services are Addressable



Isolation Between Build, Release, and Run



Stateless, Shared Nothing Processes

- Fault tolerant
- Easily scaled out

Exposing Services Through Port Bindings

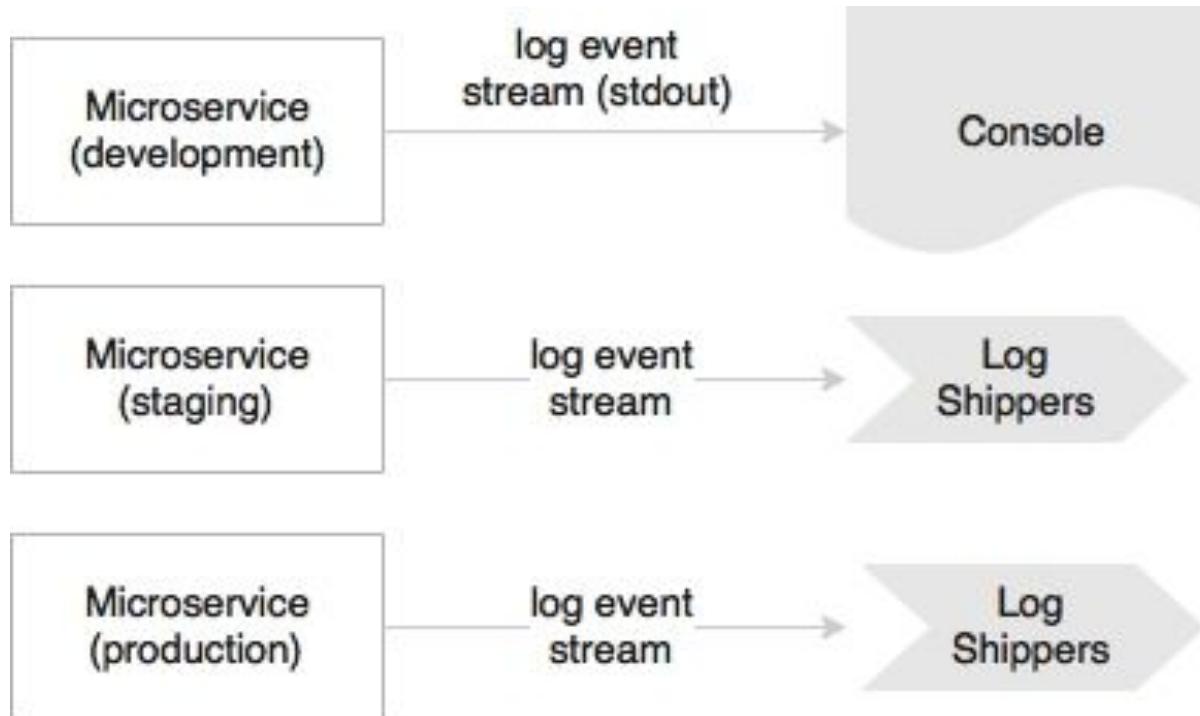
Concurrency to Scale Out

-

Disposability with Minimal Overhead

Development and Production Parity

Externalizing Logs



Package Admin Processes

Microservices Use Cases

1. Migrating a monolithic application due to improvements required in scalability, manageability, agility, or speed of delivery.
2. Utility computing scenarios such as integrating an optimization service, forecasting service, price calculation service, prediction service, offer service, recommendation service, and so on
3. Headless business applications or services that are autonomous in nature—for instance, the payment service, login service, flight search service, customer profile service, notification service, and so on.
4. Micro or macro applications that serve a single purpose and performing a single responsibility.
5. Backend services of a well-architected, responsive client-side MVC web application (the **Backend as a Service (BaaS)** scenario) load data on demand in response to the user navigation.
6. Highly agile applications, applications demanding speed of delivery or time to market, innovation pilots, applications selected for DevOps, applications of the System of Innovation type, and so on
7. Applications that we could anticipate getting benefits from microservices such as polyglot requirements, applications that require **Command Query Responsibility segregations (CQRS)**, and so on

When to consider avoiding using Microservices

1. If the organization's policies are forced to use centrally managed heavyweight components such as ESB to host a business logic or if the organization has any other policies that hinder the fundamental principles of microservices
2. If the organization's culture, processes, and so on are based on the traditional waterfall delivery model, lengthy release cycles, matrix teams, manual deployments and cumbersome release processes, no infrastructure provisioning, and so on

Microservices Early Adopters

1. **Netflix** (www.netflix.com)
2. **Uber** (www.uber.com)
3. **Airbnb** (www.airbnb.com)
4. **Orbitz** (www.orbitz.com)
5. **eBay** (www.ebay.com)
6. **Amazon** (www.amazon.com)
7. **Gilt** (www.gilt.com)
8. **Twitter** (www.twitter.com)
9. **Nike** (www.nike.com)

Common Theme

Monolithic Migrations!

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 4 (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 `2.1.6.RELEASE`

Build a Legacy Rest Application with Spring (optional)

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

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>2.1.6.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>

```
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.boot.test.context.SpringBootTest.WebEnvironment;
9 import org.springframework.boot.test.web.client.TestRestTemplate;
10 import org.springframework.boot.test.web.client.TestRestTemplate;
11 import org.springframework.boot.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 }
```

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

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})
```

Overriding default config values

It is also possible to override default

configuration values using the

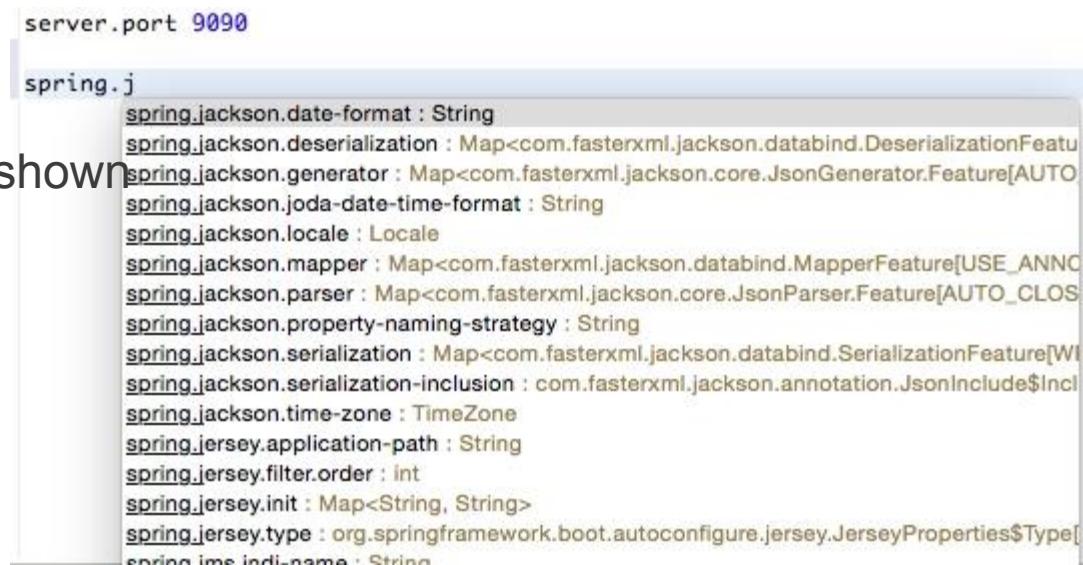
`application.properties` file.

STS provides an easy-to-autocomplete,

contextual help on

`application.properties`, as shown

in the following screenshot:



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

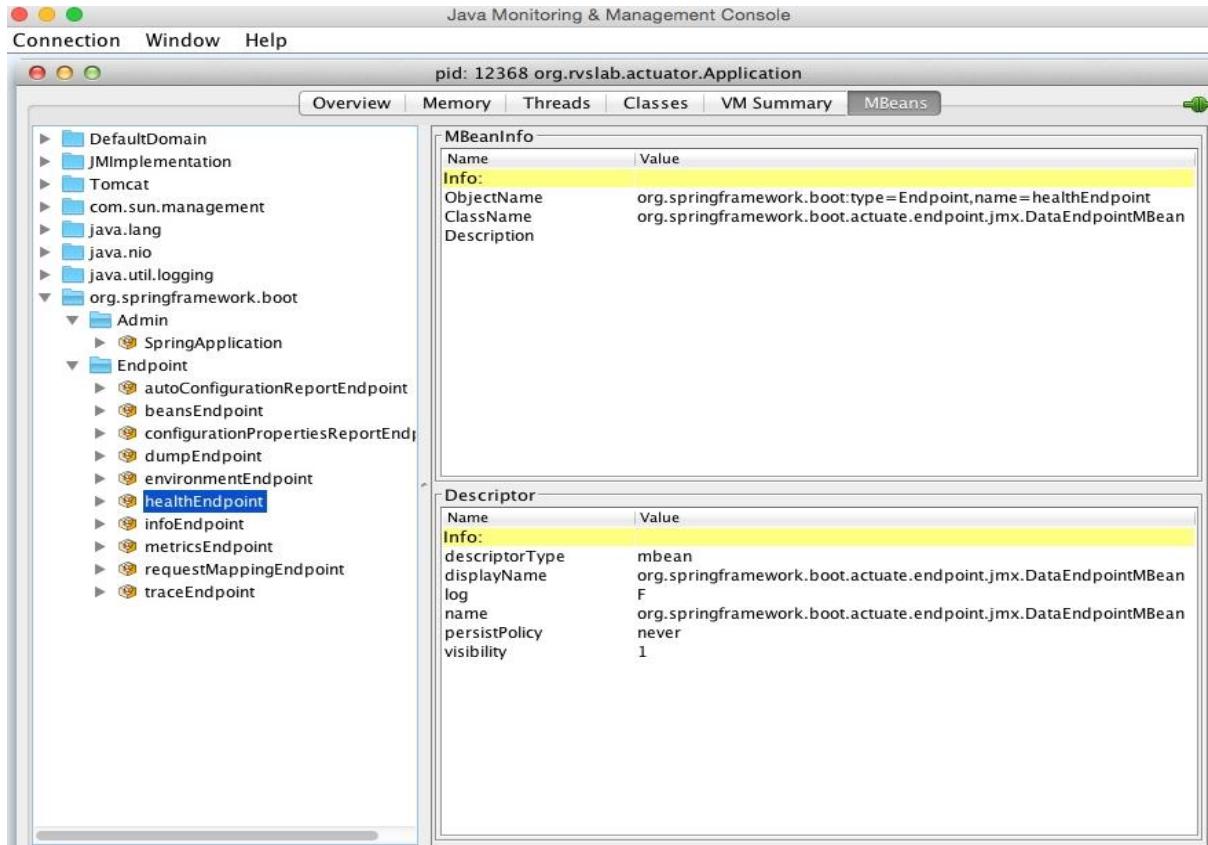
- So far, the examples we have considered are no more than just a simple "Hello world." Putting together what we have learned, this section demonstrates an end-to-end Customer Profile microservice implementation.
- The Customer Profile microservices will demonstrate interaction between different microservices.
- 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.

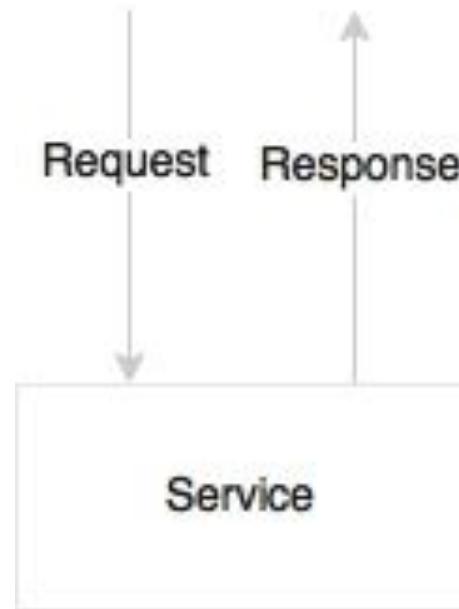
Coupling and Cohesion

- 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



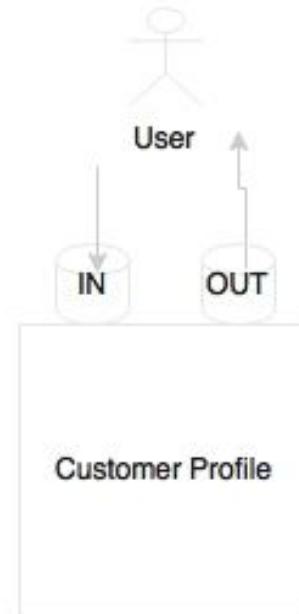
Asynchronous Communication



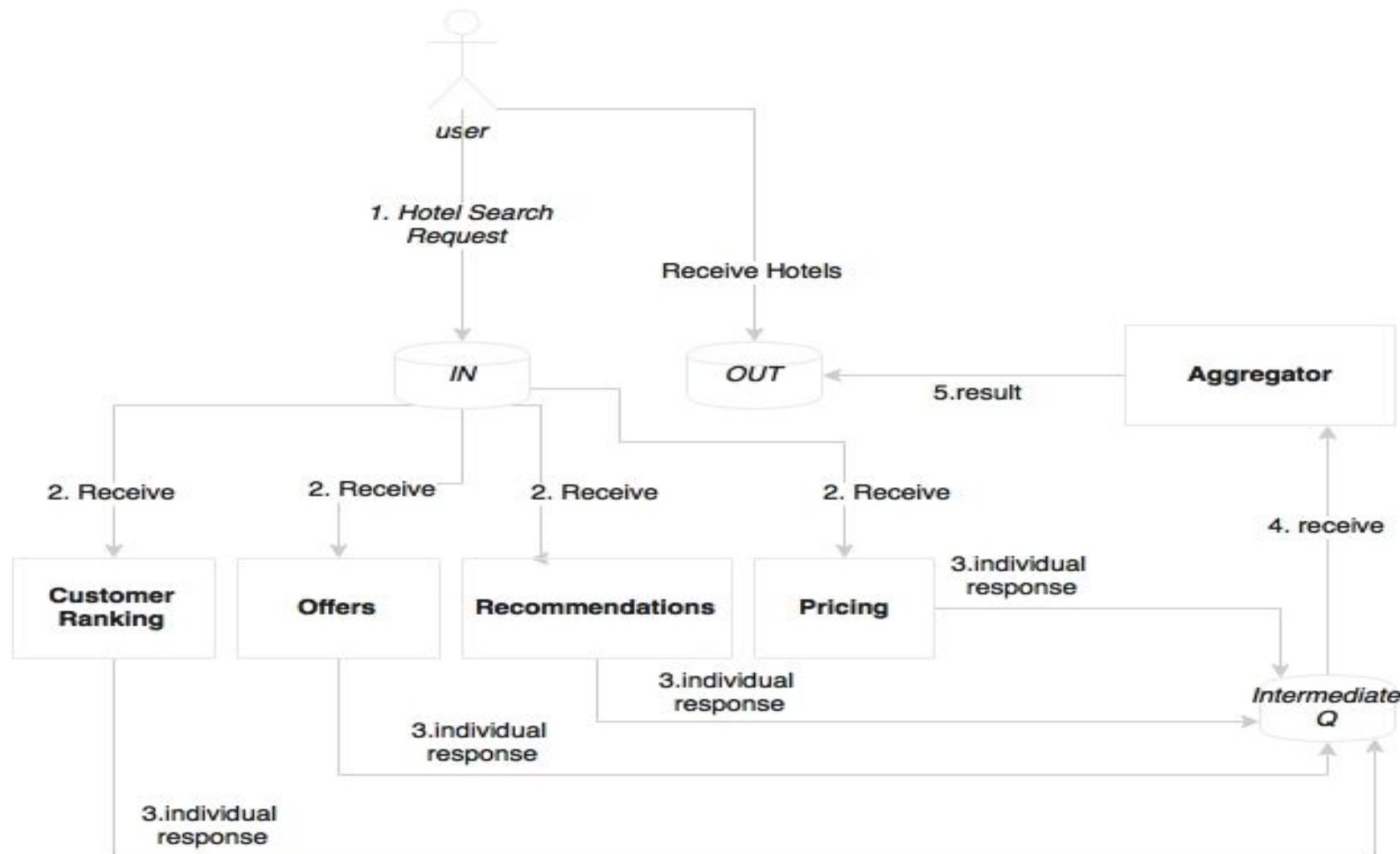
Which Style?

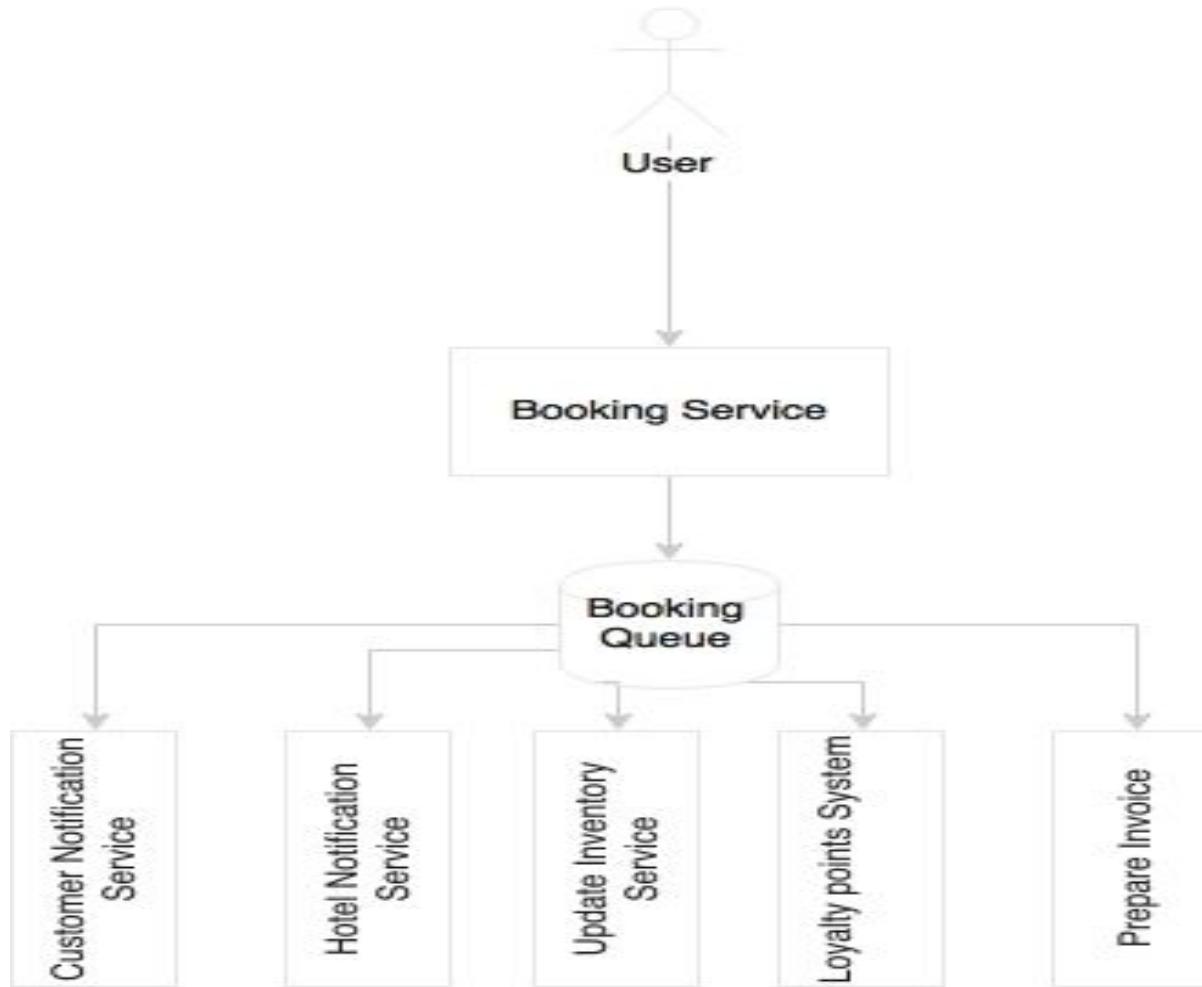


A) synchronous
request - response



B) asynchronous
request - response

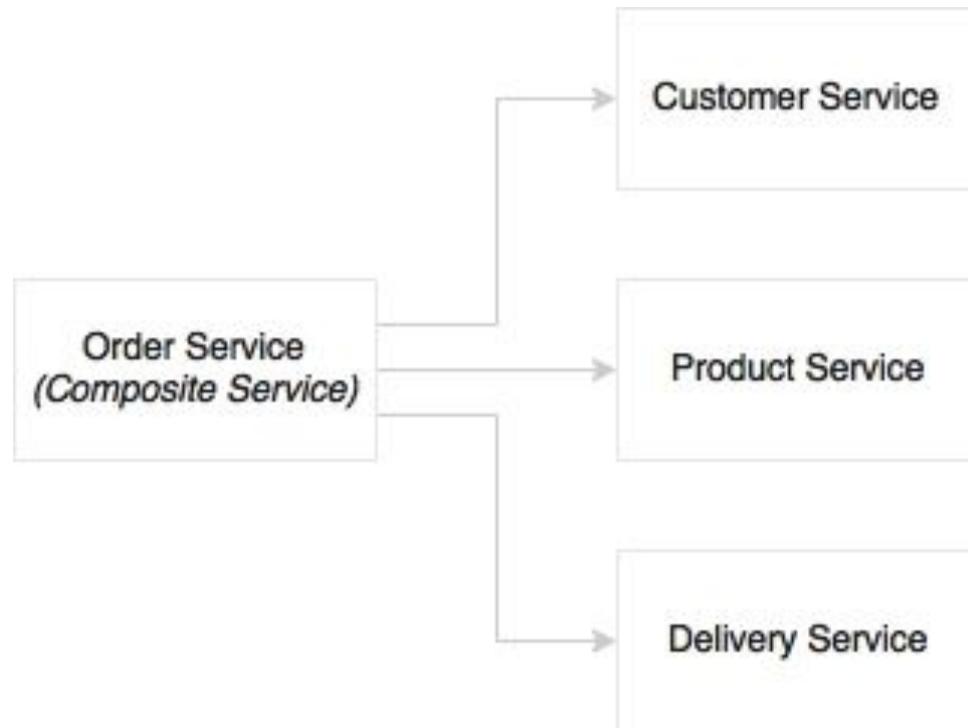




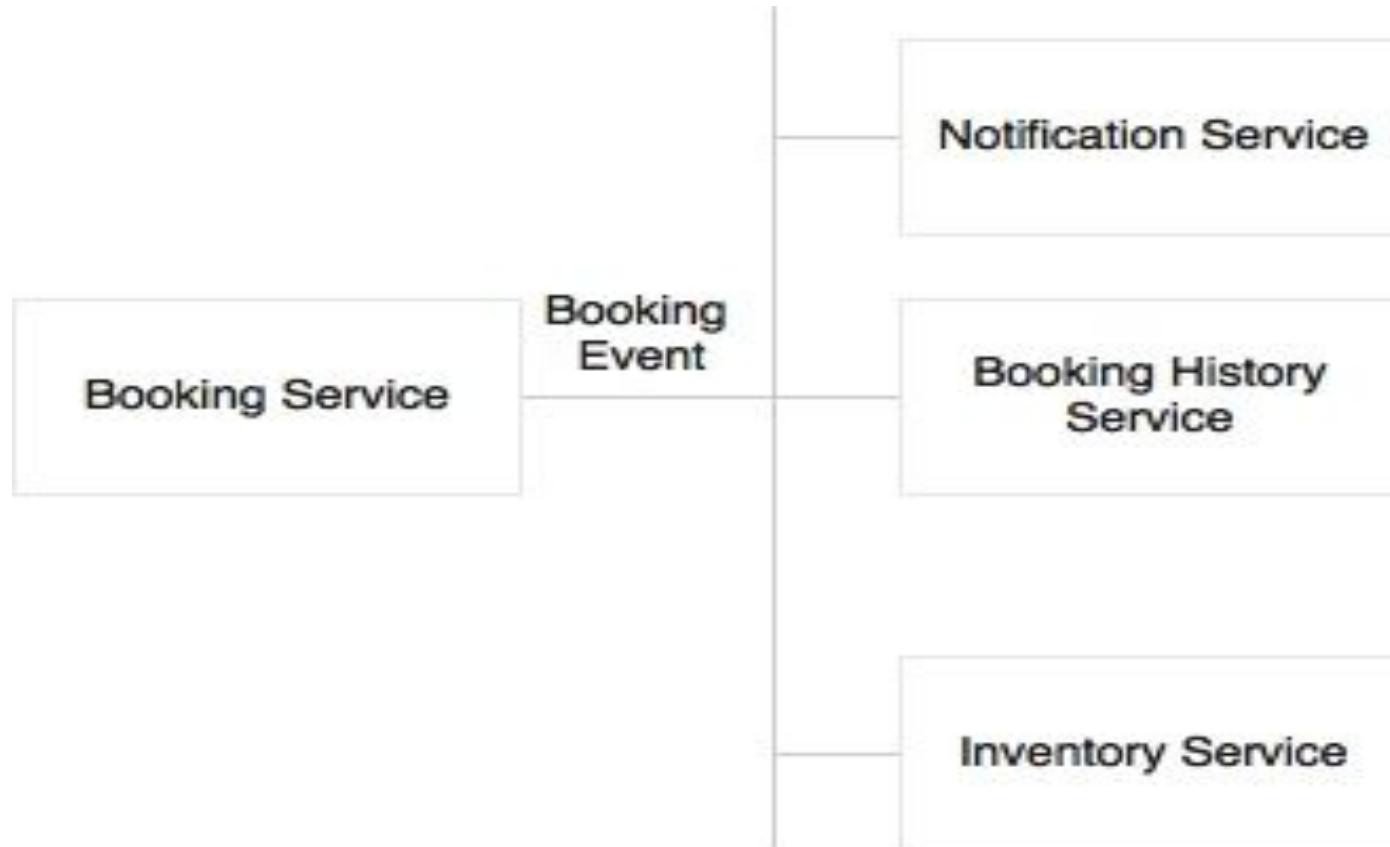
Orchestration of Microservices

- 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.

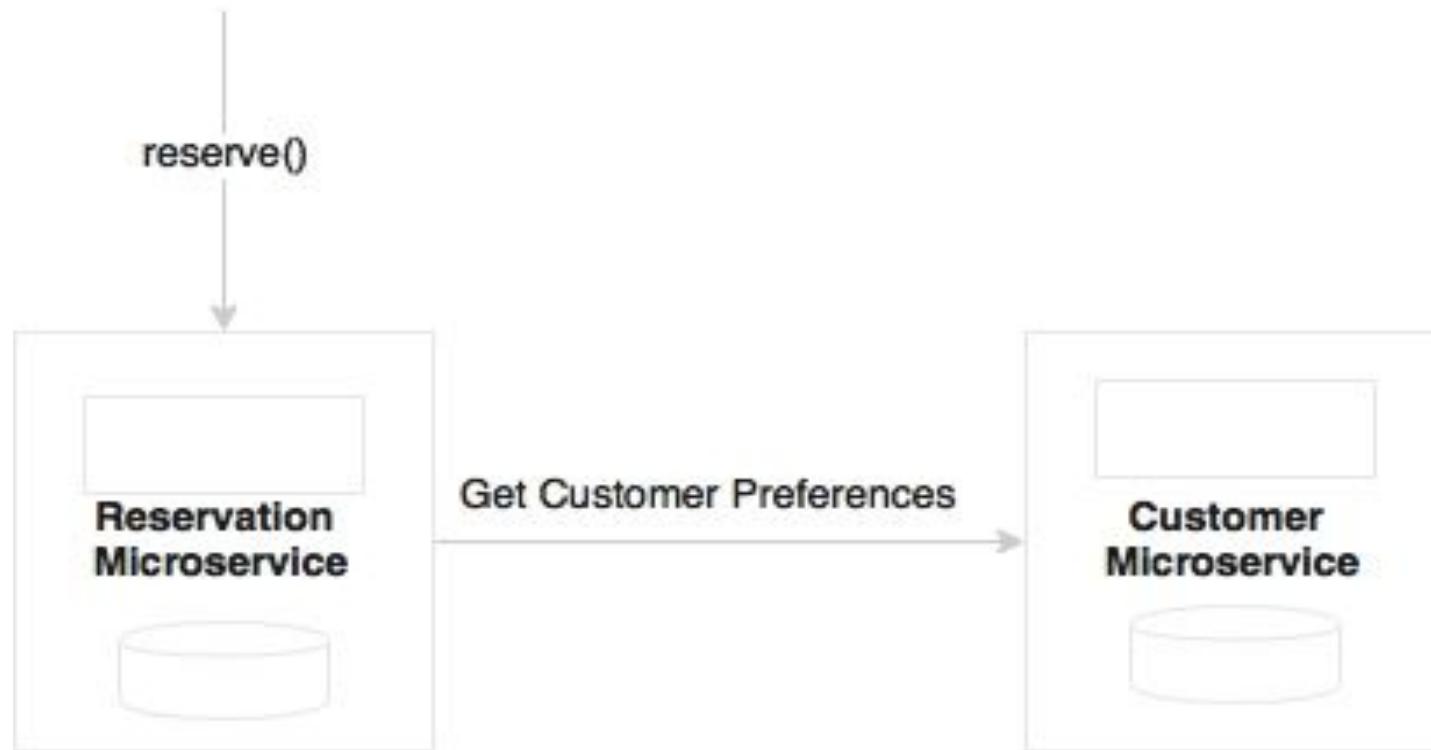
Orchestration



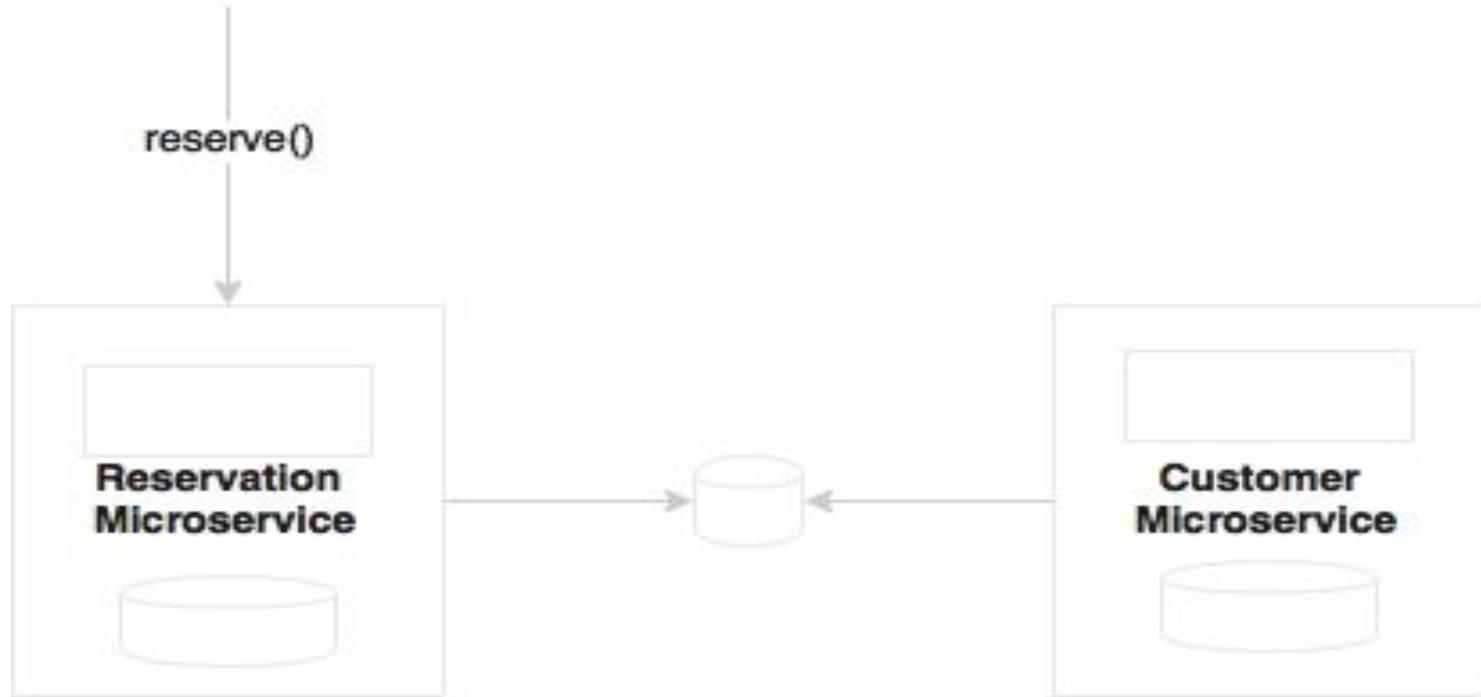
Choreography

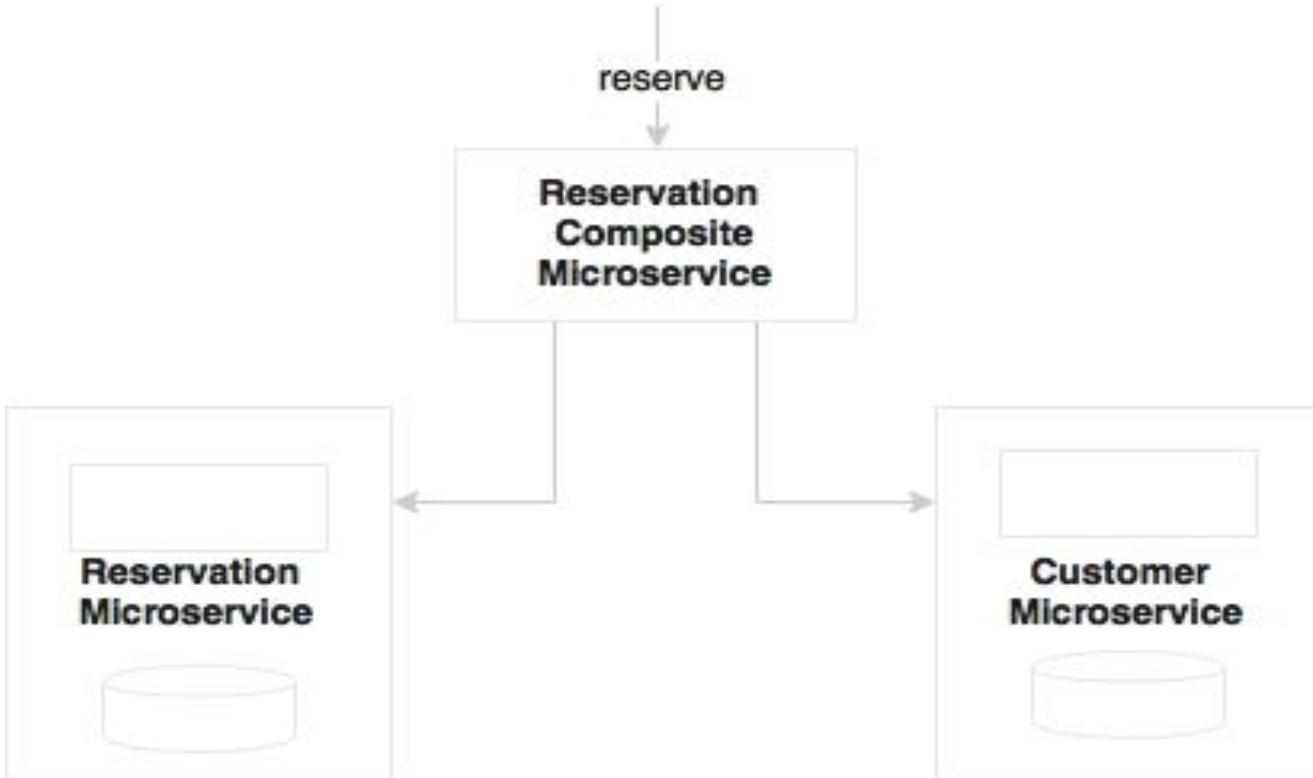


Does this work with Choreography?

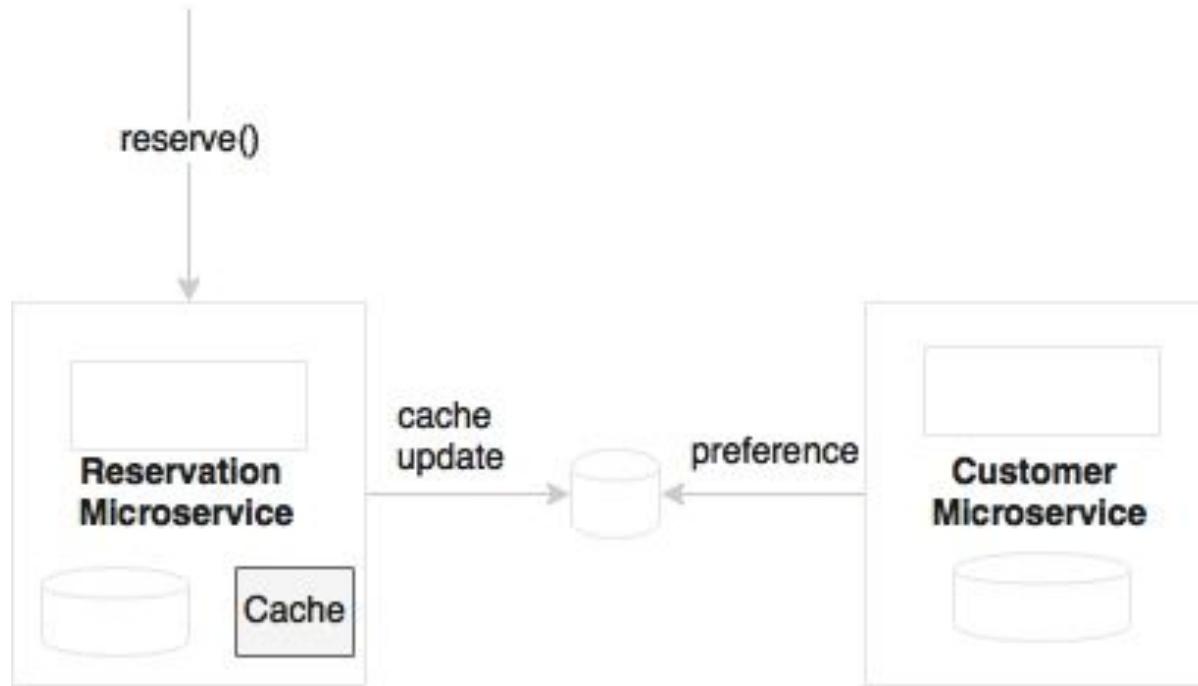


Can we make the reservation to Customer call Asynch

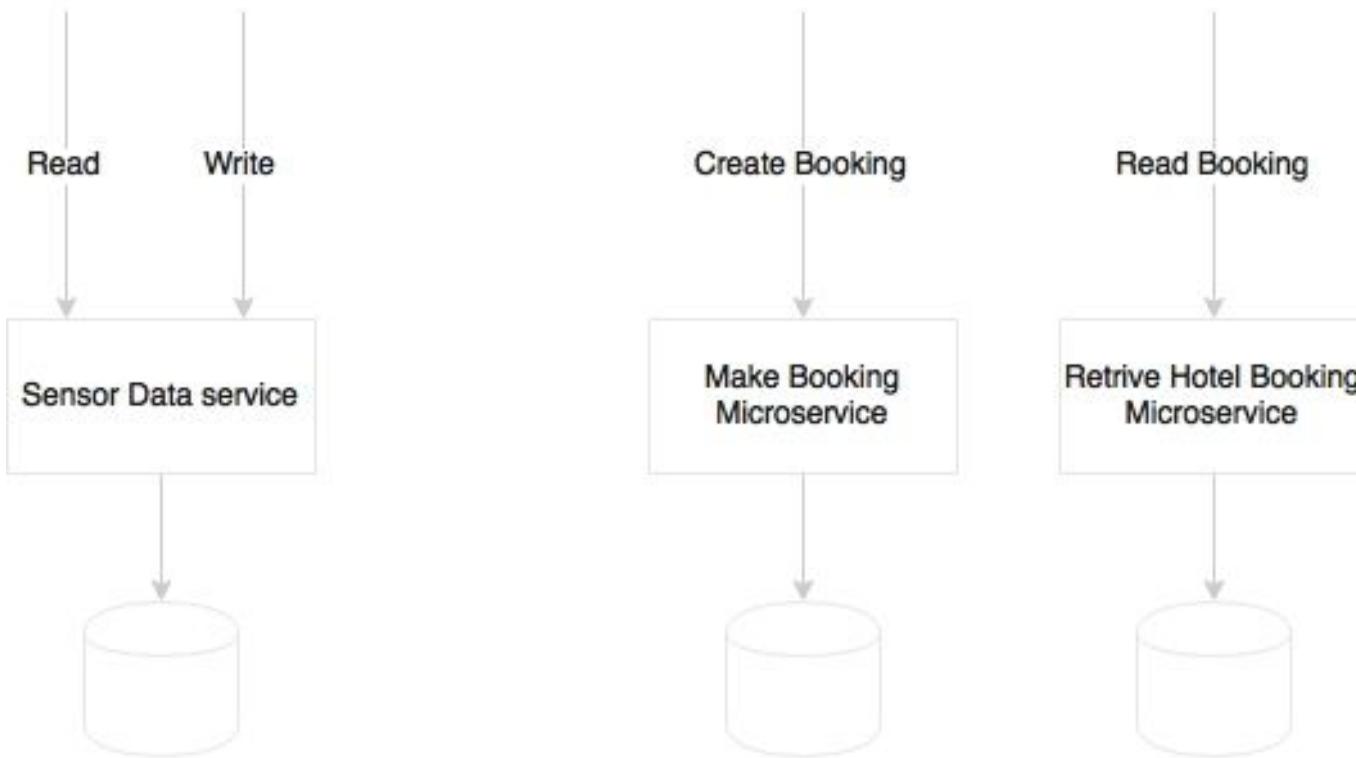




Can we duplicate customer preference & keep a copy of the preference in the Reservation?



How Many Endpoints in a microservice?



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

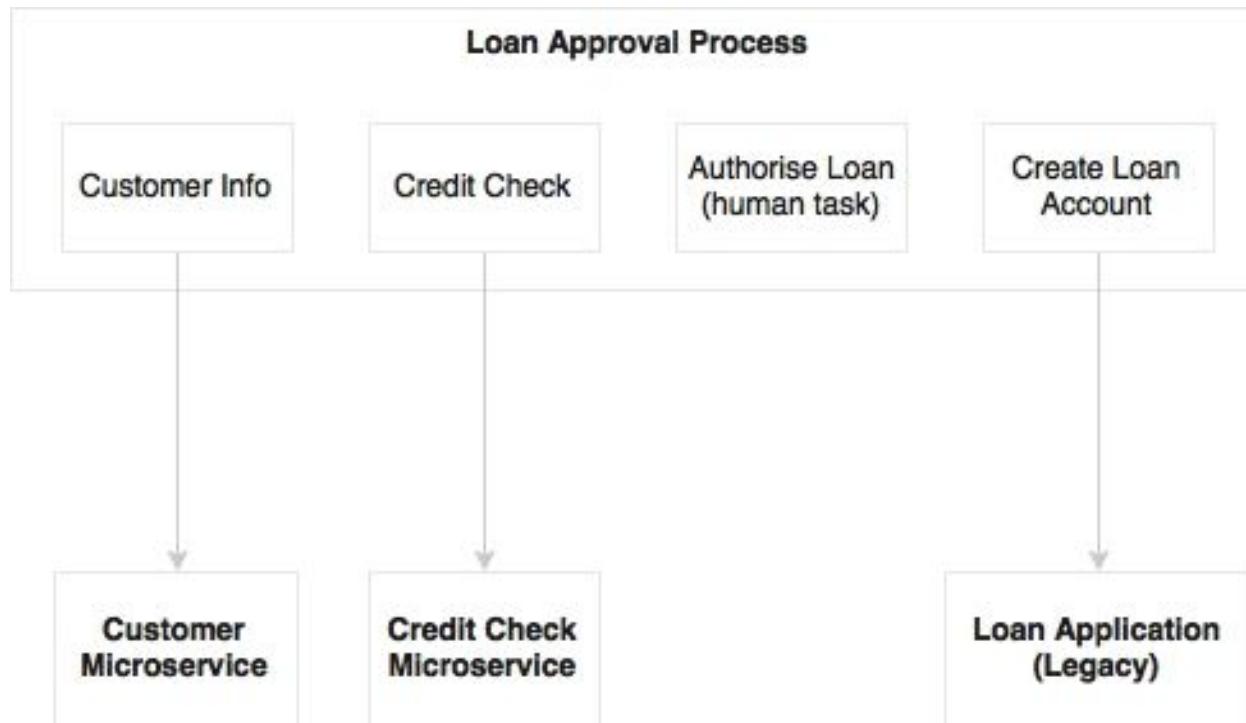
*Embedded
Rules Engine*



BPM and Workflows with Microservices

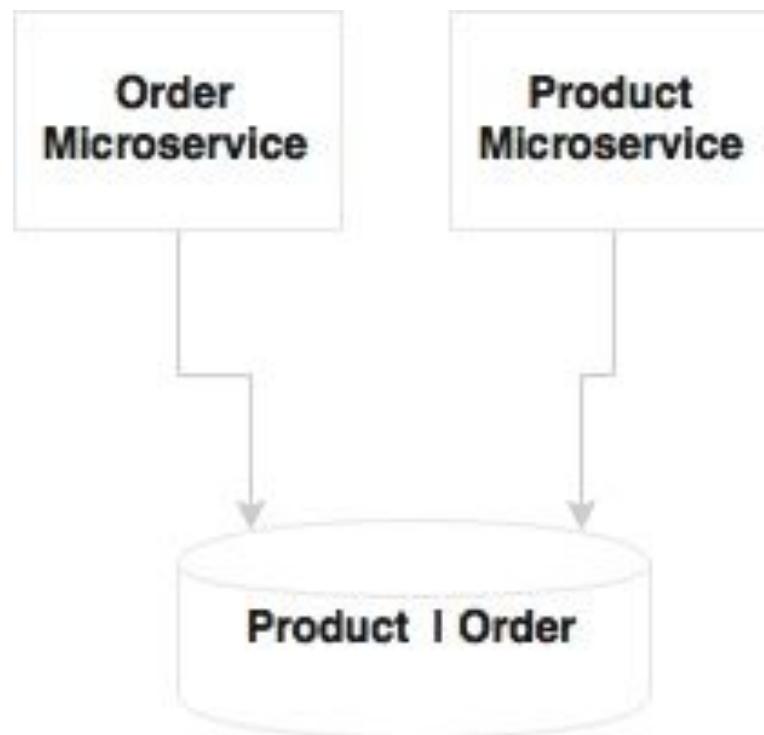
- Coordinating a long-running business process, where some processes are realized by existing assets, whereas some other areas may be niche, and there is no concrete implementation of the processes being in place. BPM allows composing both types, and provides an end-to-end automated process. This often 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

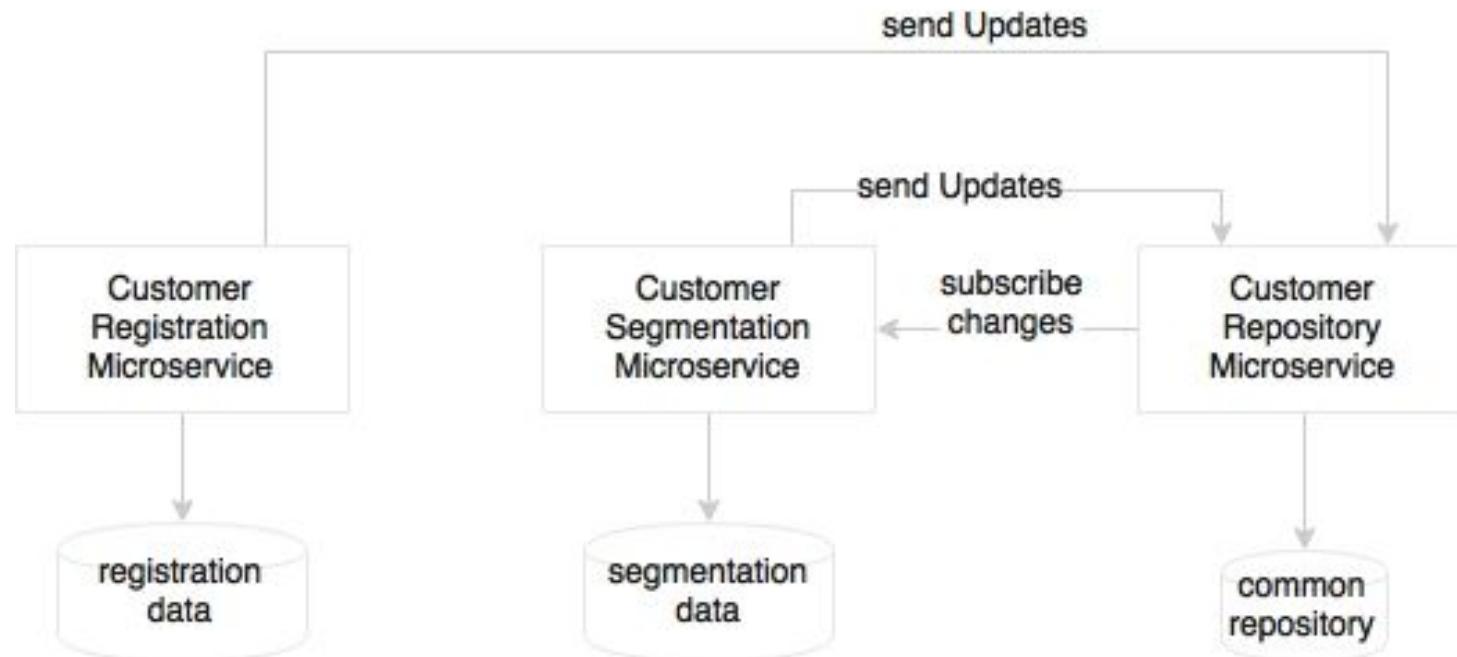




Share Data Stores with Microservices







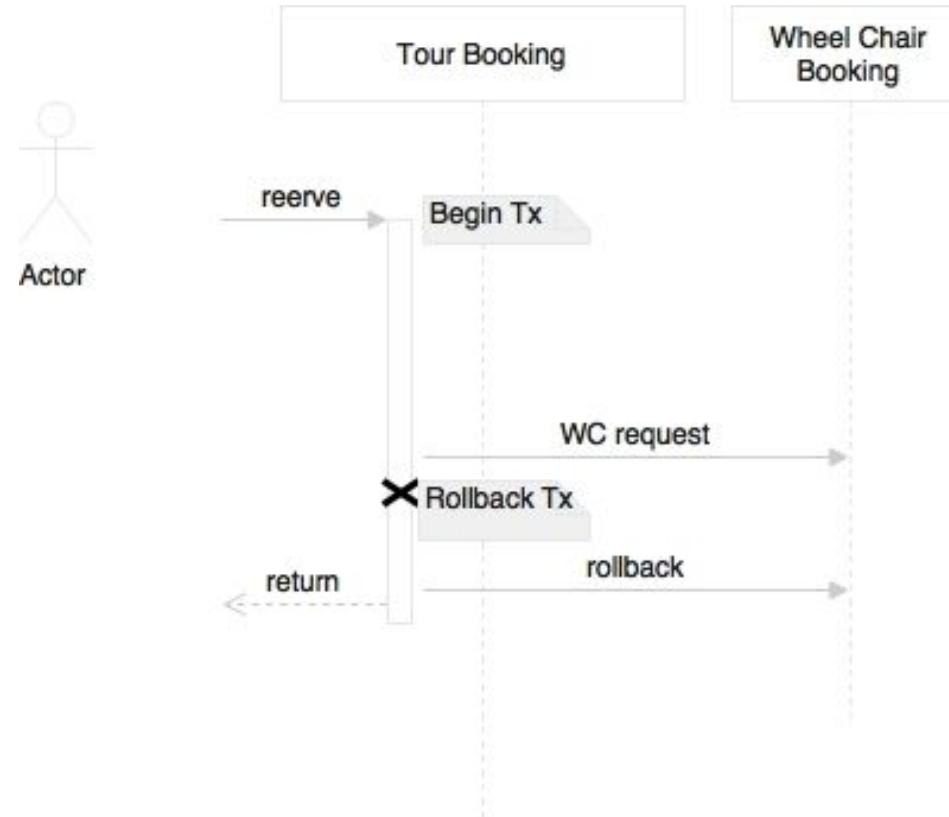
Database Transaction Boundaries

- 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.

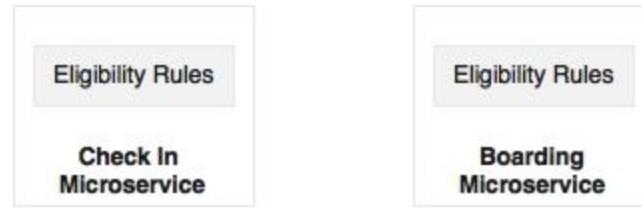
HTTP and REST endpoints

- 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



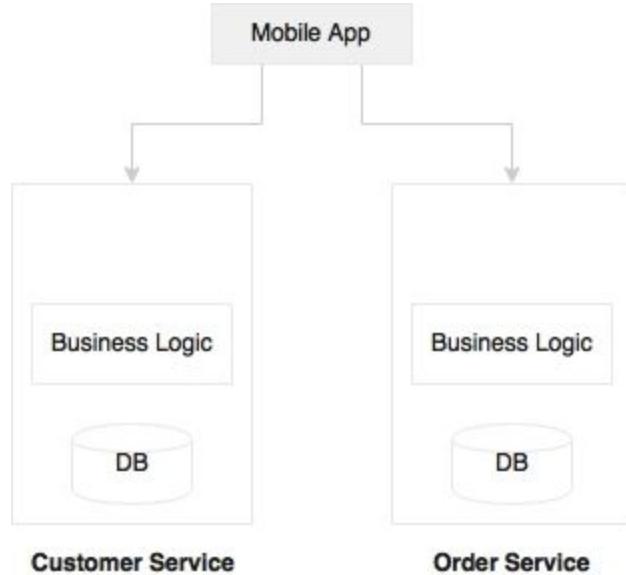
**Check In
Microservice**

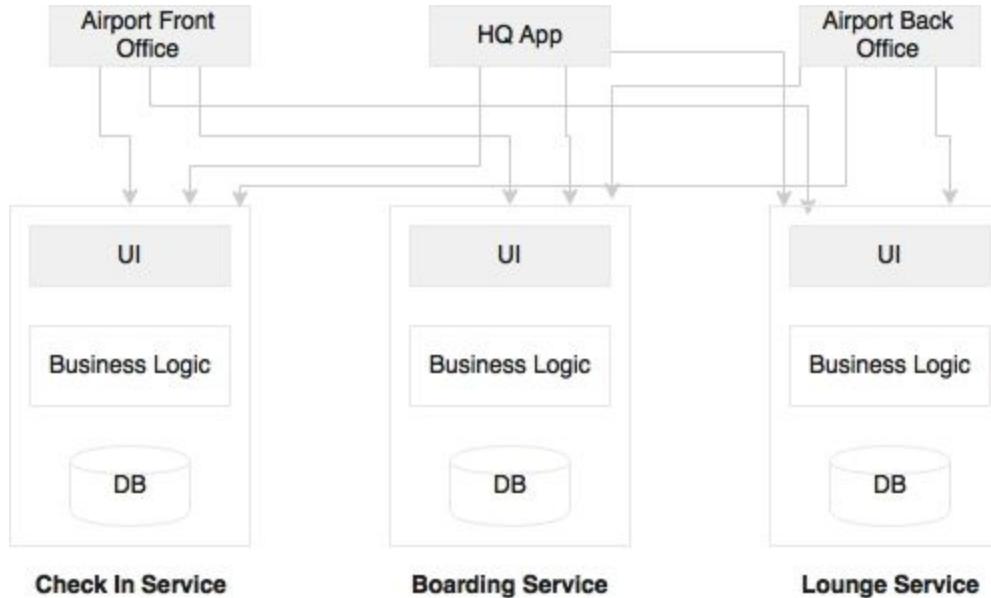
**Boarding
Microservice**

**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 {

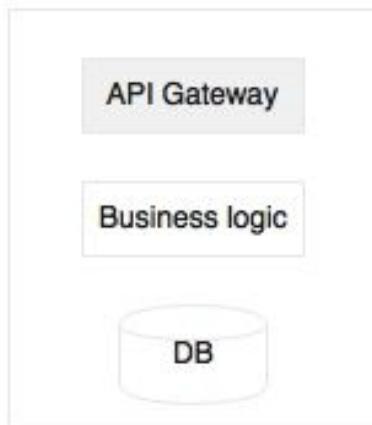
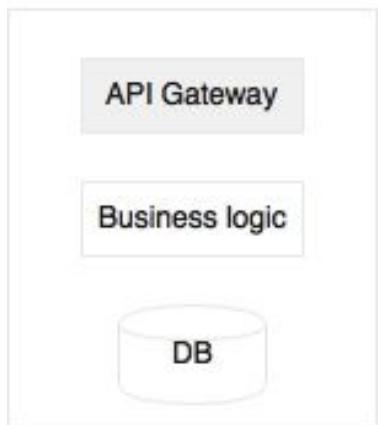
 Id: 1

 Name: /customer/name/1

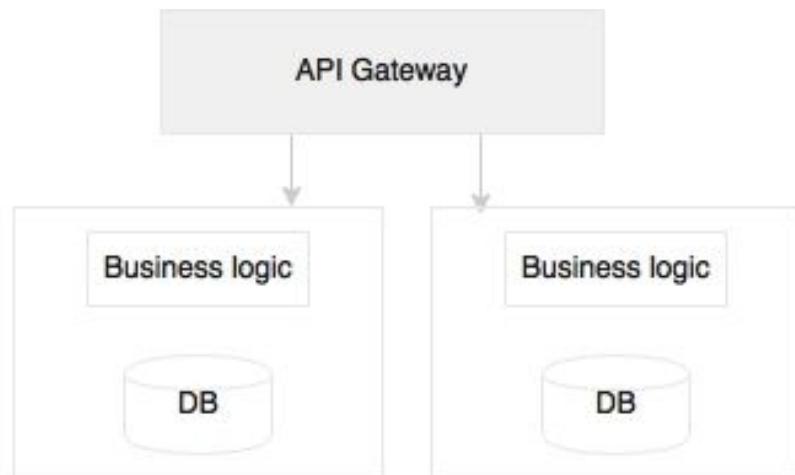
 Address: /customer/address/1

 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

3 Ways to Version Services

- URI versioning
- Media type versioning
- Custom header

URI Versioning

/api/v3/customer/1234

/api/customer/1234 - aliased to v3.

```
@RestController("CustomerControllerV3")
```

```
@RequestMapping("api/v3/customer")
```

```
public class CustomerController {
```

```
}
```

Version in the URL Parameter

A slightly different approach is to use the version number as part of the URL parameter:

api/customer/100?v=1.5

In case of media type versioning, the version is set by the client on the HTTP `Accept` header as follows:

```
Accept: application/vnd.company.customer-v3+json
```

A less effective approach for versioning is to set the version in the custom header:

```
@RequestMapping(value = "/{id}", method = RequestMethod.GET, headers = {"version=3"})  
  
public Customer getCustomer(@PathVariable("id") long id) {  
  
    //other code goes here.  
}
```

Design for Cross Origin

- With microservices, there is no guarantee that the services will run from the same host or same domain.
 - Composite UI web applications may call multiple microservices for accomplishing a task, and these could come from different domains and hosts.
- CORS allows browser clients to send requests to services hosted on different domains.
 - This is essential in a microservices-based architecture.
- One approach is to enable all microservices to allow cross origin requests from other trusted domains.
 - The second approach is to use an API gateway as a single trusted domain for the clients.

Handling Shared Reference Data

- When breaking large applications, one of the common issues which we see is the management of master data or reference data.
- Reference data is more like shared data required between different microservices.
 - City master, country master, and so on will be used in many services such as flight schedules, reservations, and others.

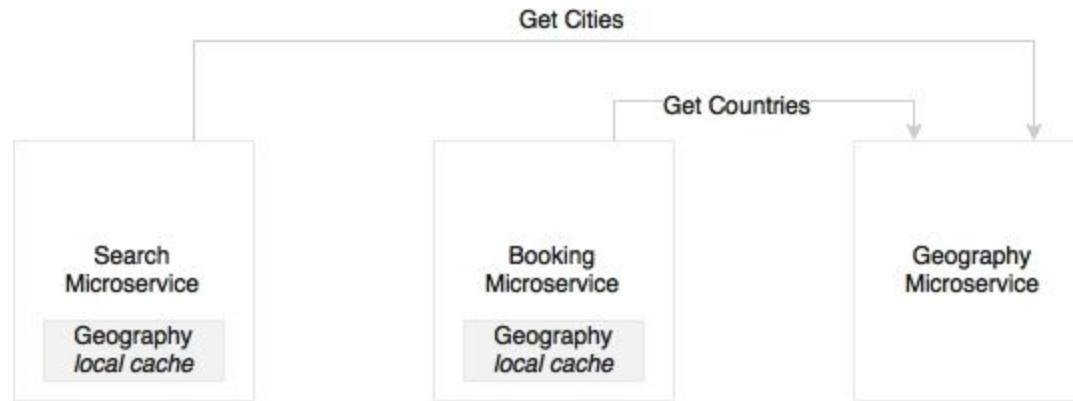
Handling Shared Reference Data Continued



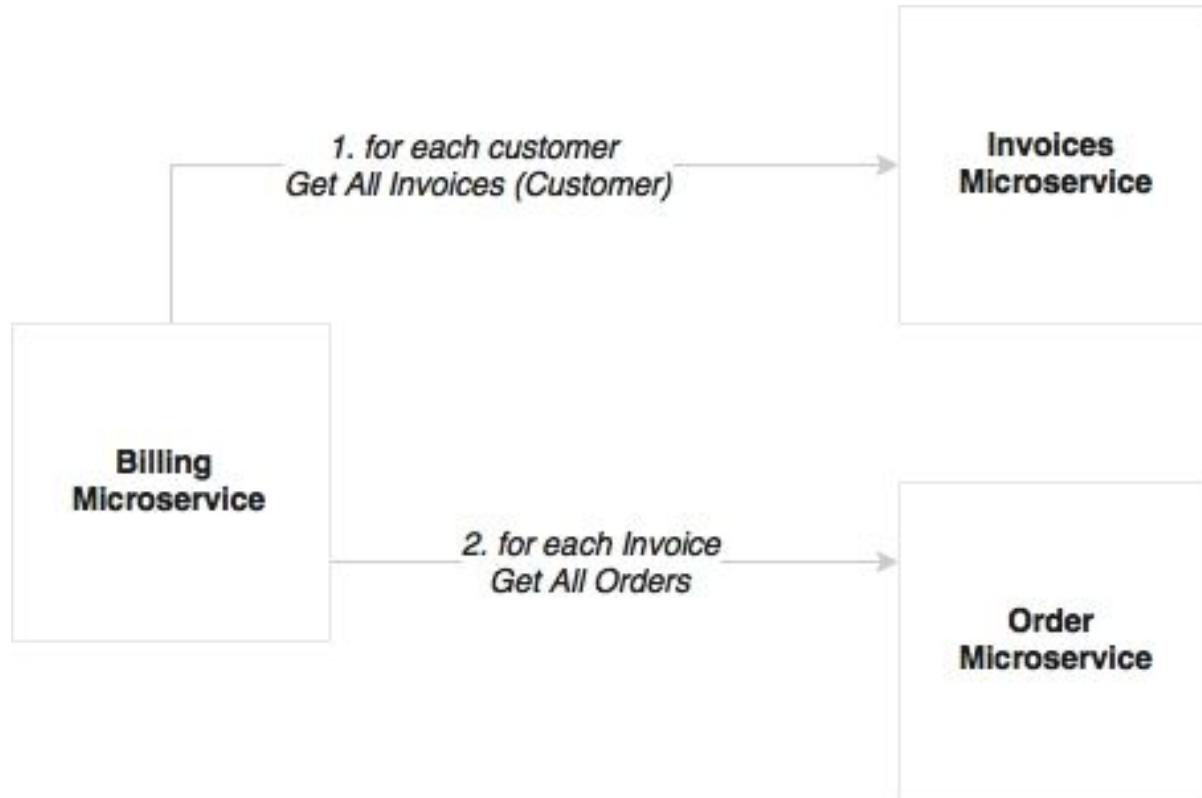
Handling Shared Reference Data Continued



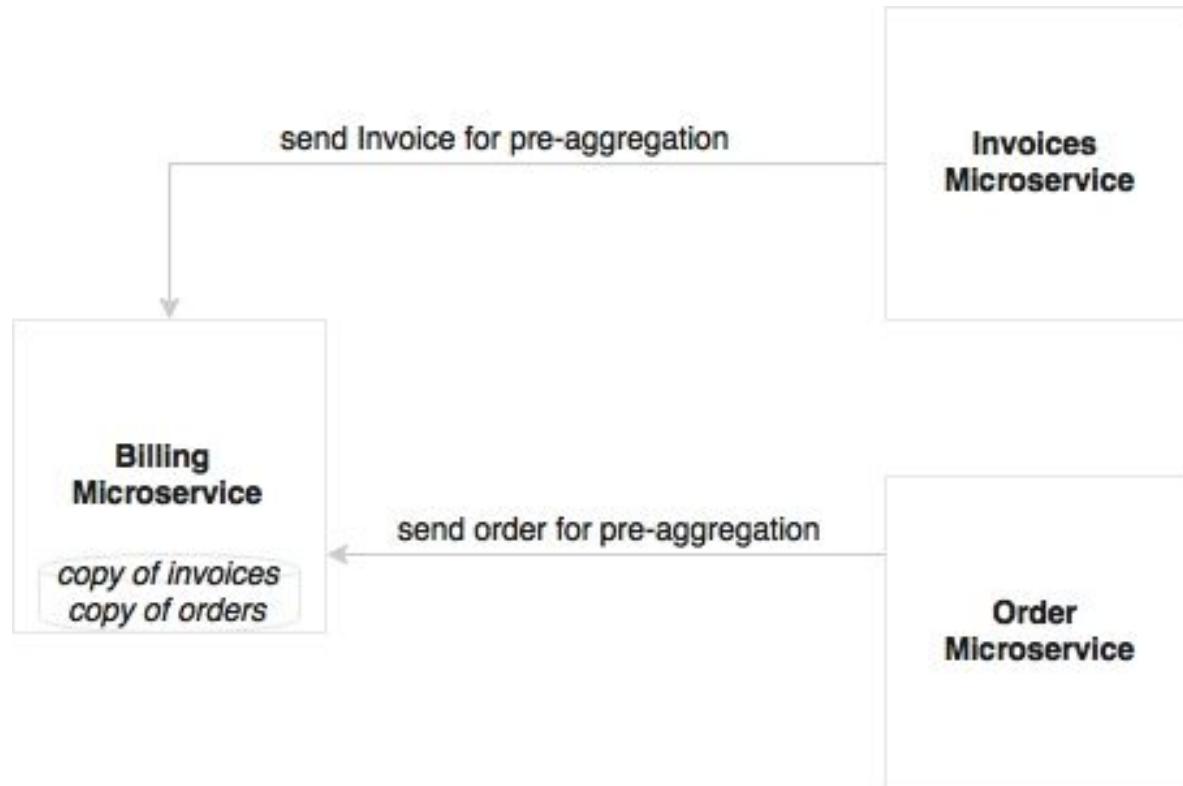
Handling Shared Reference Data Continued



Bulk Operations and Microservices



Bulk Operations and Microservices Continued



Bulk Operations and Microservices Continued

There are two ways we can think about for solving Batch.

- The first approach is to pre-aggregate data as and when it is created.
 - When an order is created, an event is sent out.
 - Upon receiving the event, the **Billing** microservice keeps aggregating data internally for monthly processing.
 - In this case, there is no need for the **Billing** microservice to go out for processing.
 - The downside of this approach is that there is duplication of data.
- A second approach, when pre-aggregation is not possible, is to use batch APIs.
 - In such cases, we call `GetAllInvoices`, then we use multiple batches, and each batch further uses parallel threads to get orders.
 - Spring Batch is useful in these situations.

Microservices Challenges

- Data Islands
- Logging
- Monitoring
- Dependency Management
- Organizational Culture
- Governance Challenges
- Operation Overheads
- Testing
- Infrastructure

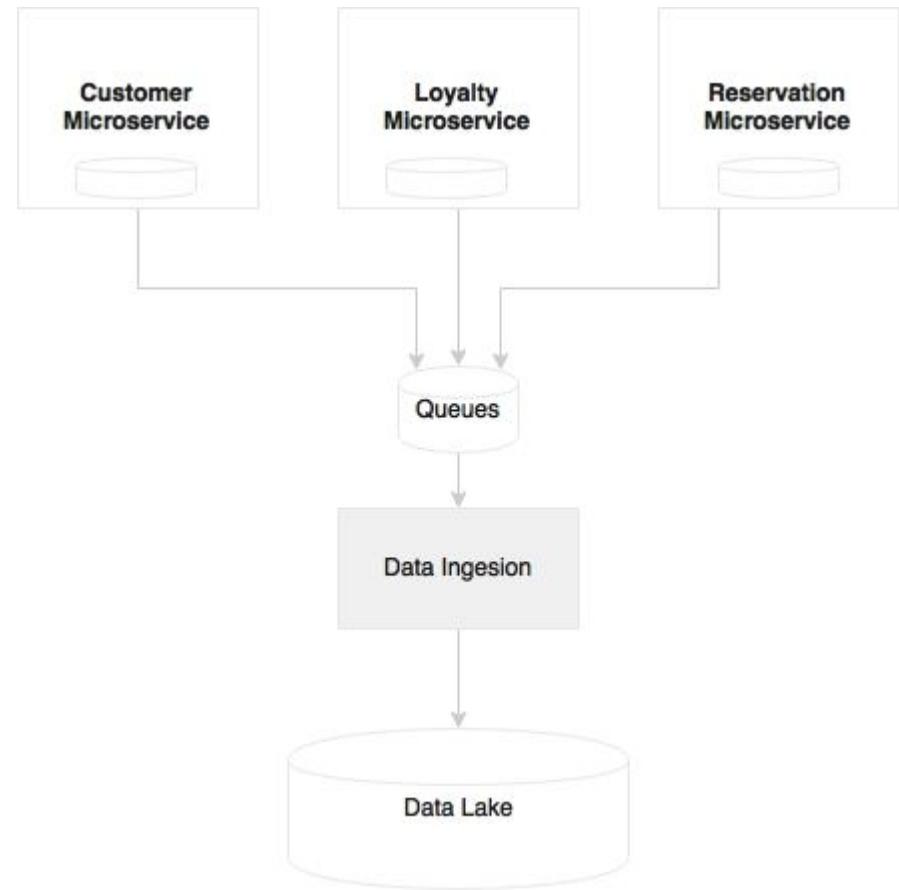
Data Islands

- Microservices abstract their own local transactional store, which is used for their own transactional purposes.
 - The type of store and the data structure will be optimized for the services offered by the microservice.
- For instance, if we want to develop a customer relationship graph, we may use a graph database like Neo4j, OrientDB, and the like.
 - A predictive text search to find out a customer based on any related information such as passport number, address, e-mail, phone, and so on could be best realized using an indexed search database like Elasticsearch or Solr.

Data Islands Continued

- This will place us into a unique situation of fragmenting data into heterogeneous data islands.
 - For example, Customer, Loyalty Points, Reservations, and others are different microservices, and hence, use different databases.
 - What if we want to do a near real-time analysis of all high value customers by combining data from all three data stores?
 - *This was easy with a monolithic application, because all the data was present in a single database*

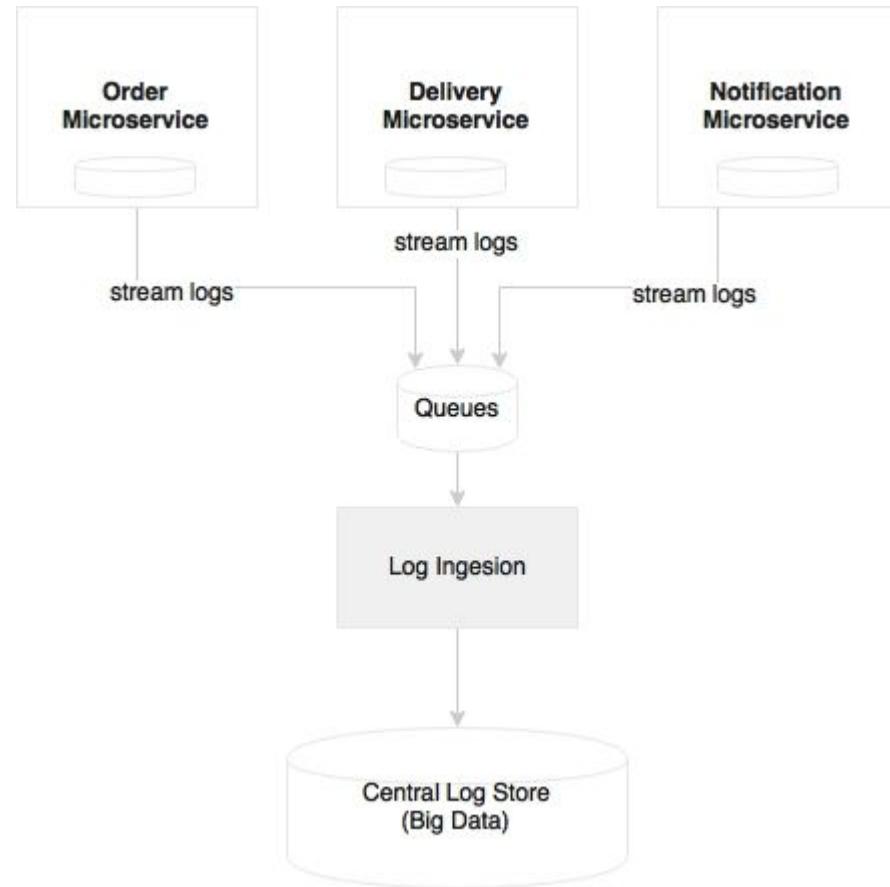
Data Islands Solved



Logging and Monitoring

- Log files are a good piece of information for analysis and debugging.
- Since each microservice is deployed independently, they emit separate logs, maybe to a local disk.
- This results in fragmented logs.
 - When we scale services across multiple machines, each service instance could produce separate log files.
 - This makes it extremely difficult to debug and understand the behavior of the services through log mining.

Logging and Monitoring



Logging and Monitoring

- When implementing microservices, we need a capability to ship logs from each service to a centrally managed log repository.
- With this approach, services do not have to rely on the local disk or local I/Os. A second advantage is that the log files are centrally managed, and are available for all sorts of analysis such as historical, real time, and trending.
- By introducing a correlation ID, end-to-end transactions can be easily tracked.

Dependency Management

Too many dependencies could raise challenges in microservices. Four important design aspects are stated as follows:

- Reducing dependencies by properly designing service boundaries.
- Reducing impacts by designing dependencies as loosely coupled as possible. Also, designing service interactions through asynchronous communication styles.
- Tackling dependency issues using patterns such as circuit breakers.
- Monitoring dependencies using visual dependency graphs.

Corporate Culture

- One of the biggest challenges in microservices implementation is the organization culture.
- To harness the speed of delivery of microservices, the organization should adopt Agile development processes, continuous integration, automated QA checks, automated delivery pipelines, automated deployments, and automatic infrastructure provisioning.
- Organizations following a waterfall development or heavyweight release management processes with infrequent release cycles are a challenge for microservices development.
- Insufficient automation is also a challenge for microservices deployment.

Governance

- Microservices impose decentralized governance, and this is quite in contrast with the traditional SOA governance.
 - Organizations may find it hard to come up with this change, and that could negatively impact the microservices development.
- There are number of challenges that comes with a decentralized governance model.
 - How do we understand who is consuming a service?
 - How do we ensure service reuse?
 - How do we define which services are available in the organization?
 - How do we ensure enforcement of enterprise policies?

CMDB

- With many microservices, the number of **configurable items (CIs)** becomes too high, and the number of servers in which these CIs are deployed might also be unpredictable.
- This makes it extremely difficult to manage data in a traditional **Configuration Management Database (CMDB)**.
- *In most cases, it is more useful to dynamically discover the current running topology than a statically configured CMDB-style deployment topology.*

Testing

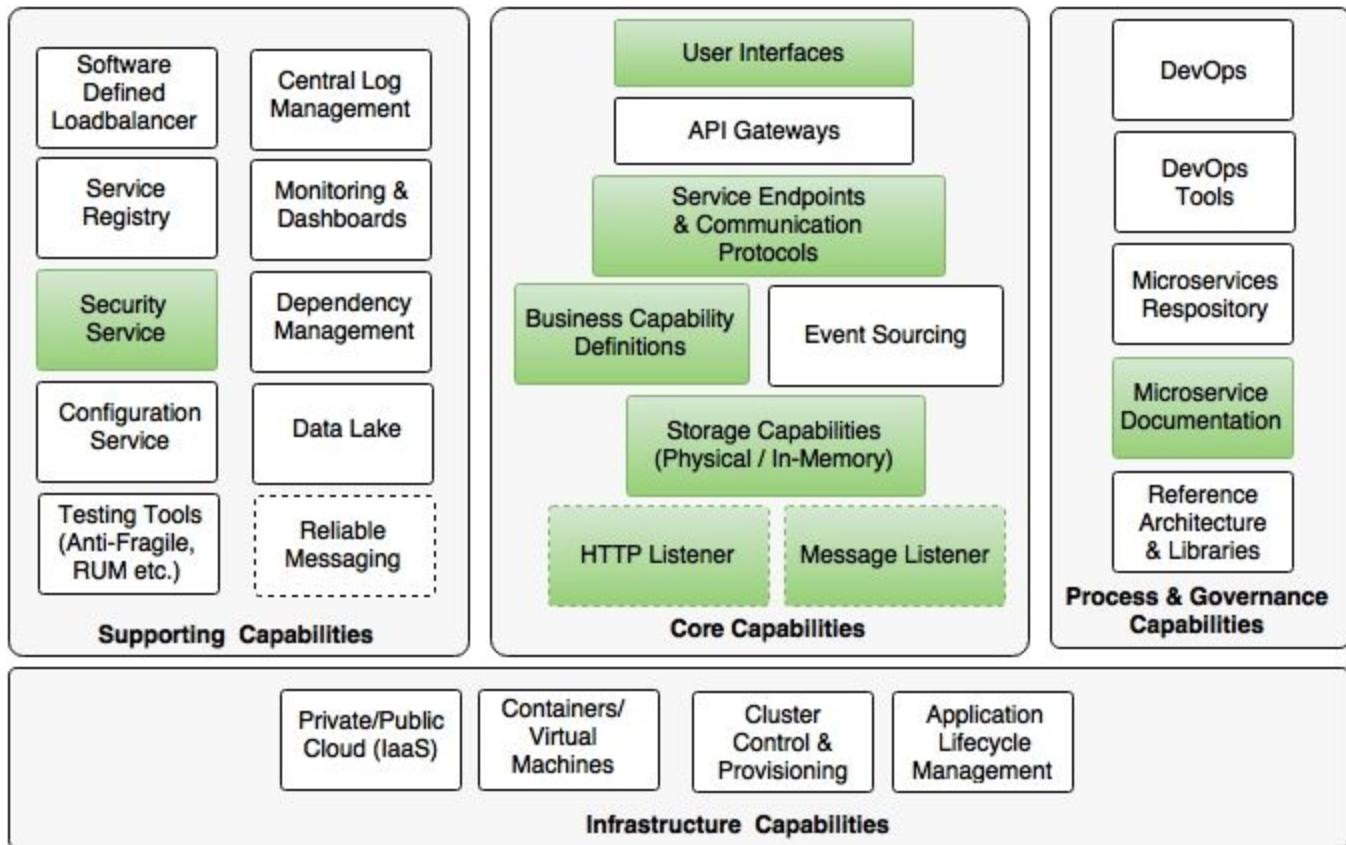
- Microservices also pose a challenge for the testability of services.
 - In order to achieve a full-service functionality, one service may rely on another service, and that, in turn, on another service—either synchronously or asynchronously.
 - The issue is how do we test an end-to-end service to evaluate its behavior? T
 - The dependent services may or may not be available at the time of testing.
- Service virtualization or service mocking is one of the techniques used for testing services without actual dependencies.
 - In testing environments, when the services are not available, mock services can simulate the behavior of the actual service.
 - The microservices ecosystem needs service virtualization capabilities.
 - However, this may not give full confidence, as there may be many corner cases that mock services do not simulate, especially when there are deep dependencies.

Provisioning Infrastructure

- Manual deployment could severely challenge the microservices rollouts.
- If a deployment has manual elements, the deployer or operational administrators should know the running topology, manually reroute traffic, and then deploy the application one by one till all services are upgraded.
- With many server instances running, this could lead to significant operational overheads. Moreover, the chances of errors are high in this manual approach.

4 - Microservices se Use Case / Case Study

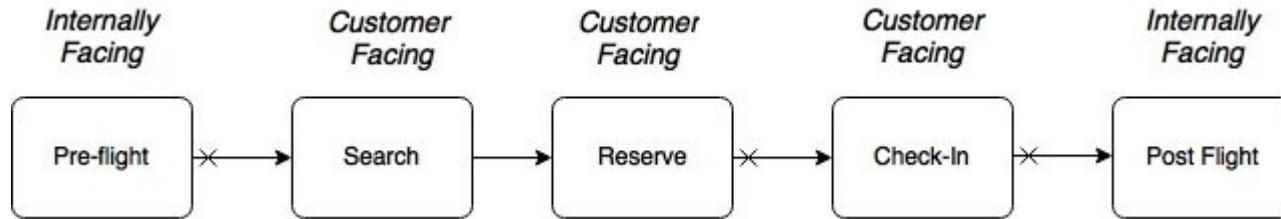
- This chapter will introduce BrownField Airline (BF), a fictitious budget airline, and their journey from a monolithic **Passenger Sales and Service (PSS)** application to a next generation microservices architecture.
- This section examines the PSS application in detail, and explains the challenges, approach, and transformation steps of a monolithic system to a microservices-based architecture, adhering to the principles and practices that were explained in the previous section.
- The intention of this case study is to get us as close as possible to a live scenario so that the architecture concepts can be set in stone.



Understanding the Application

- Business Process View
- Functional View
- Architectural View
- Design View
- Implementation View
- Deployment View
-

Business Process View

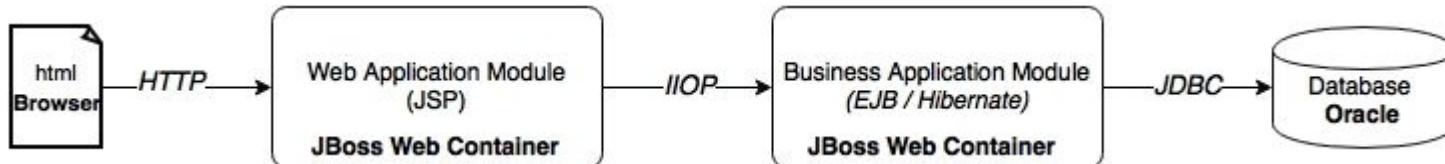


- The current solution is automating certain customer-facing functions as well as certain internally facing functions.
- There are two internally facing functions, **Pre-flight** and **Post-flight**. **Pre-flight** functions include the planning phase, used for preparing flight schedules, plans, aircrafts, and so on.
- **Post-flight** functions are used by the back office for revenue management, accounting, and so on.
- The **Search** and **Reserve** functions are part of the online seat reservation process, and the **Check-in** function is the process of accepting passengers at the airport.
- The **Check-in** function is also accessible to the end users over the Internet for online check-in.

Functional View

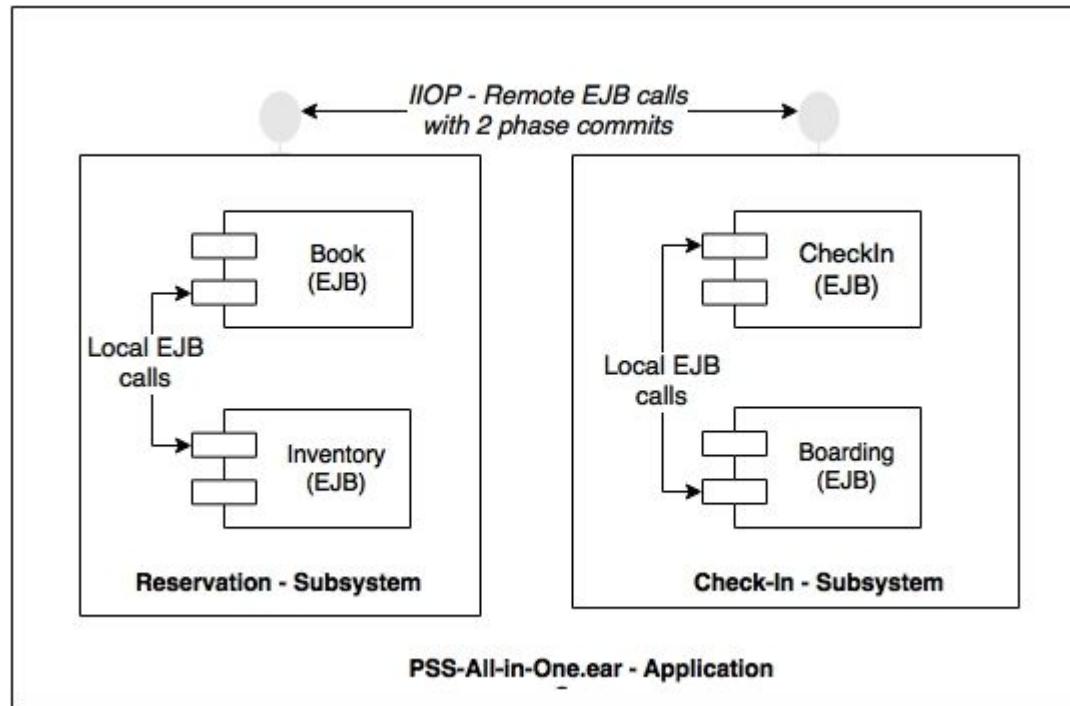
Search Functions	Search Flight availability between cities for a given date	Flight Flight routes, aircraft type and schedules	Fare Fares between cities for each flight & date	
Reservation Functions	Book Book passengers on a selected flight & date	Inventory Number of seats available on a flight & date	Payment Payment gateway for online payments	
Check In Functions	Check In Accept a passenger on a flight on the day of travel	Boarding Mark passenger as boarded on the airplane	Seating Allocate passenger a seat based on rules	Baggage Accept passenger baggage and print bag tag
Back Office Functions	CRM Customer relationship management	Data Analysis Business intelligence analysis and reporting	Revenue Management Fare calculations based on forecasts	Accounting Invoicing and Billing
Data Management Functions	Reference Data Country, City, Aircrafts, Currency etc.	Customer Manage customers		
Cross Cutting Functions	User Management Manage user, roles, privileges	Notification Send SMS and e-mails to customers		

Architectural View

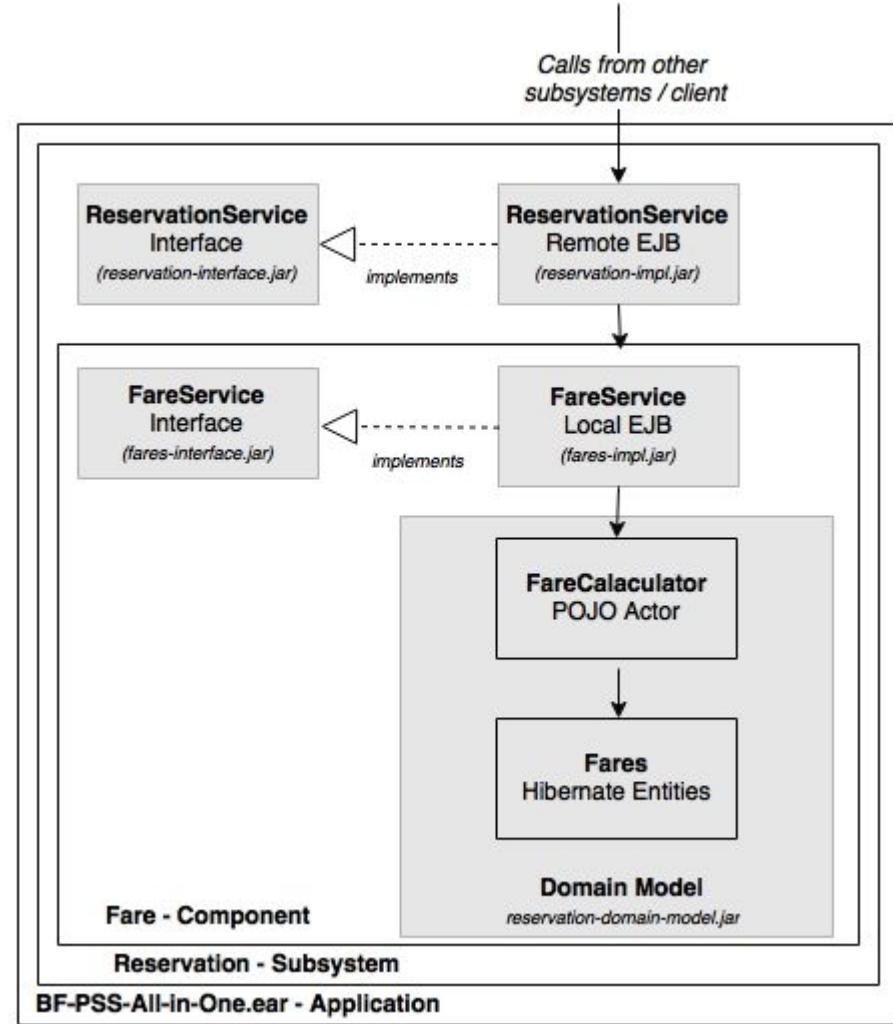


- In order to effectively manage the end-to-end passenger operations, BrownField had developed an in-house PSS application, almost ten years back.
- This well-architected application was developed using Java and JEE technologies combined with the best-of-the-breed open source technologies available at the time.
- The architecture has well-defined boundaries.
- Also, different concerns are separated into different layers.
- The web application was developed as an N-tier, component-based modular system.
- The functions interact with each other through well-defined service contracts defined in the form of EJB endpoints.

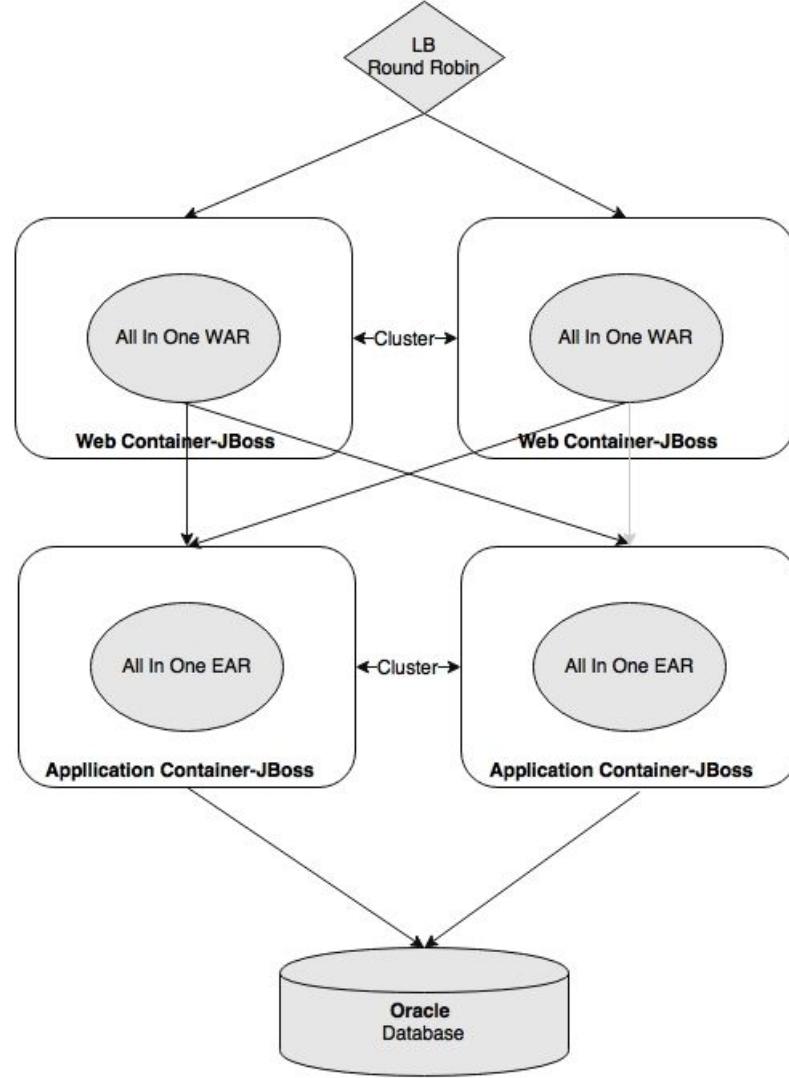
Design View



Implementation View



Deployment View



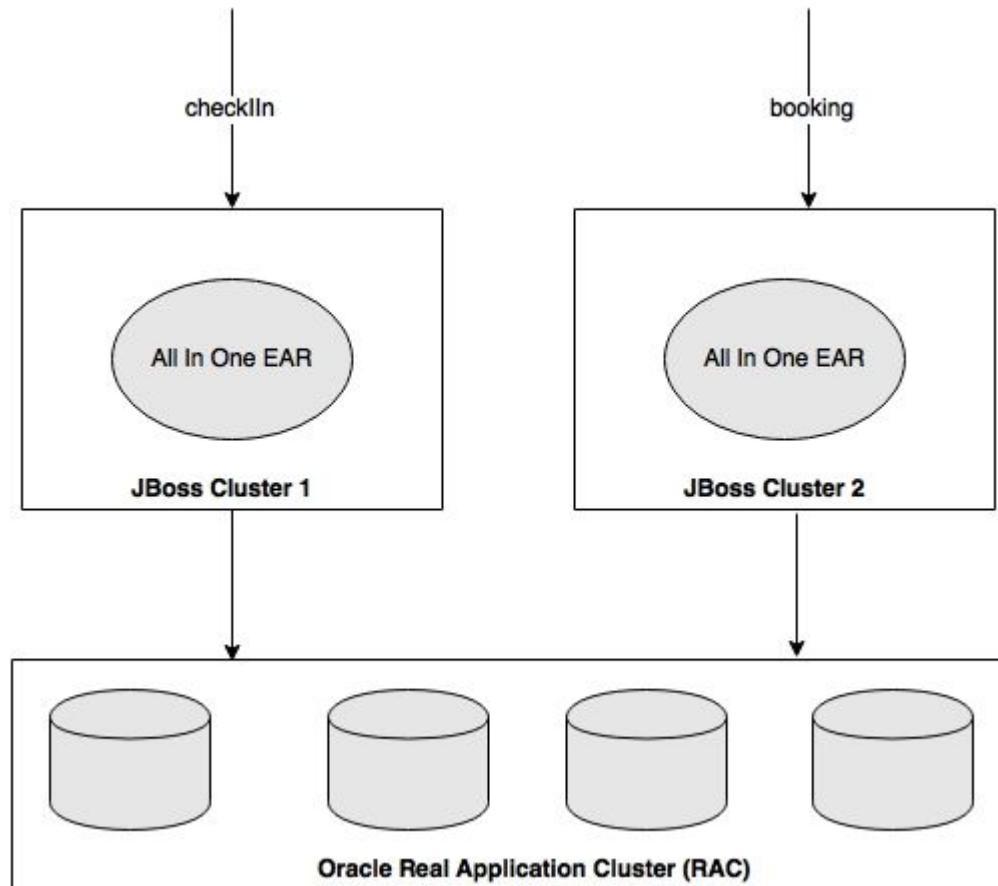
Why Microservices? (Death of the Monolith)

- The PSS application was performing well, successfully supporting all business requirements as well as the expected service levels.
 - The system had no issues in scaling with the organic growth of the business in the initial years.
- The business has seen tremendous growth over a period of time.
 - The fleet size increased significantly, and new destinations got added to the network.
 - As a result of this rapid growth, the number of bookings has gone up, resulting in a steep increase in transaction volumes, up to 200 - to 500 - fold of what was originally estimated.

Why Microservices? (Death of the Monolith)

- Pain Points:
 - Stability
 - Outages
 - Agility

Stop Gap



Retrospection

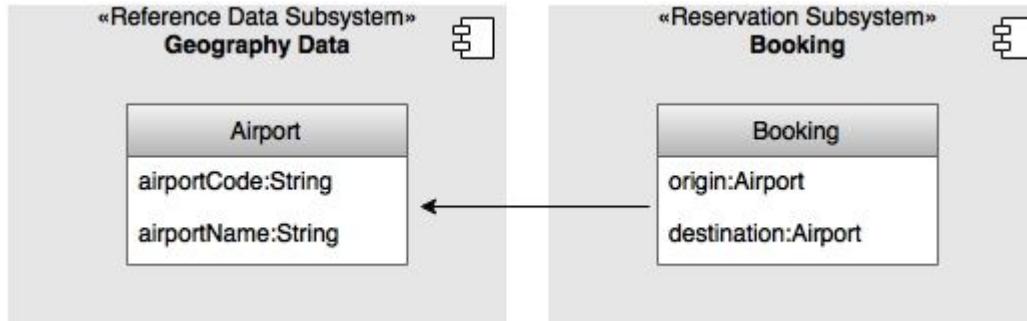
- Although the application was well-architected, there was a clear segregation between the functional components.
 - They were loosely coupled, programmed to interfaces, with access through standards-based interfaces, and had a rich domain model.
- The obvious question is, how come such a well-architected application failed to live up to the expectations?
 - What else could the architects have done?
- It is important to understand what went wrong over a period of time.
 - *In the context of this class, it is also important to understand how microservices can avoid the recurrence of these scenarios.*

Shared Data

- Almost all functional modules require reference data such as the airline's details, airplane details, a list of airports and cities, countries, currencies, and so on.
 - For example, fare is calculated based on the point of origin (city), a flight is between an origin and a destination (airports), check-in is at the origin airport (airport), and so on.
 - In some functions, the reference data is a part of the information model, whereas in some other functions, it is used for validation purposes.
- Much of this reference data is neither fully static nor fully dynamic.
 - Addition of a country, city, airport, or the like could happen when the airline introduces new routes.
 - Aircraft reference data could change when the airline purchases a new aircraft, or changes an existing airplane's seat configuration.

Shared Data continued

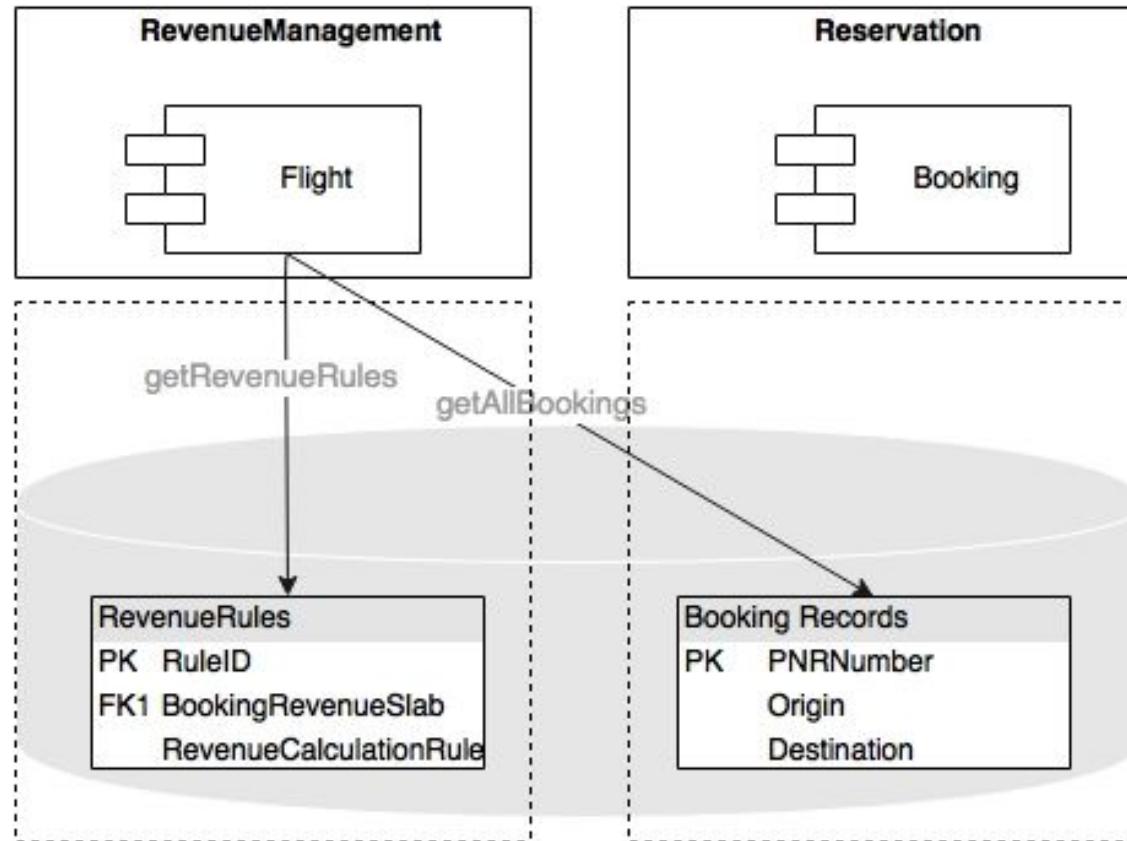
- The architects considered multiple approaches when designing the system.
- Separating the reference data as an independent subsystem like other subsystems was one of the options considered, but this could lead to performance issues.
- The team took the decision to follow an exception approach for handling reference data as compared to other transactions.
- Considering the nature of the query patterns discussed earlier, the approach was to use the reference data as a shared library.



Single Database

Native Queries

- The Hibernate framework provides a good abstraction over the underlying databases.
 - It generates efficient SQL statements, in most of the cases targeting the database using specific dialects.
 - However, sometimes, writing native JDBC SQLs offers better performance and resource efficiency.
 - In some cases, using native database functions gives an even better performance.
- The single database approach worked well at the beginning.
 - But over a period of time, it opened up a loophole for the developers by connecting database tables owned by different subsystems.
 - Native JDBC SQL was a good vehicle for doing this.



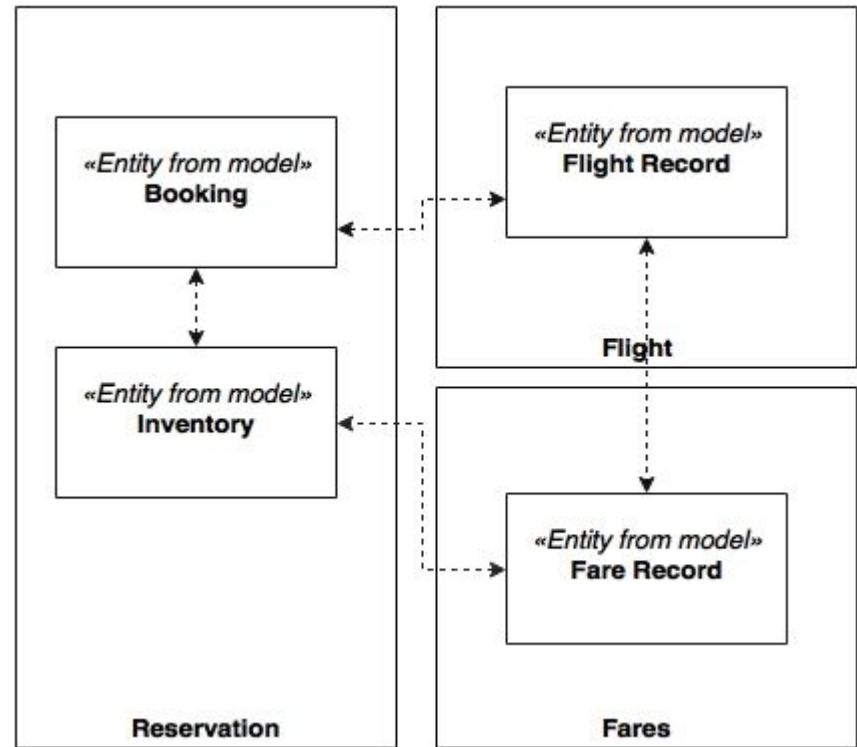
Single Database continued

- Another issue that surfaced as a result of the use of a single database was the use of complex stored procedures.
 - Some of the complex data-centric logic written at the middle layer was not performing well, causing slow response, memory issues, and thread-blocking issues.
- In order to address this problem, the developers took the decision to move some of the complex business logic from the middle tier to the database tier by implementing the logic directly within the stored procedures.
 - This decision resulted in better performance of some of the transactions, and removed some of the stability issues. More and more procedures were added over a period of time.
 - However, this eventually broke the application's modularity.

Domain Boundaries

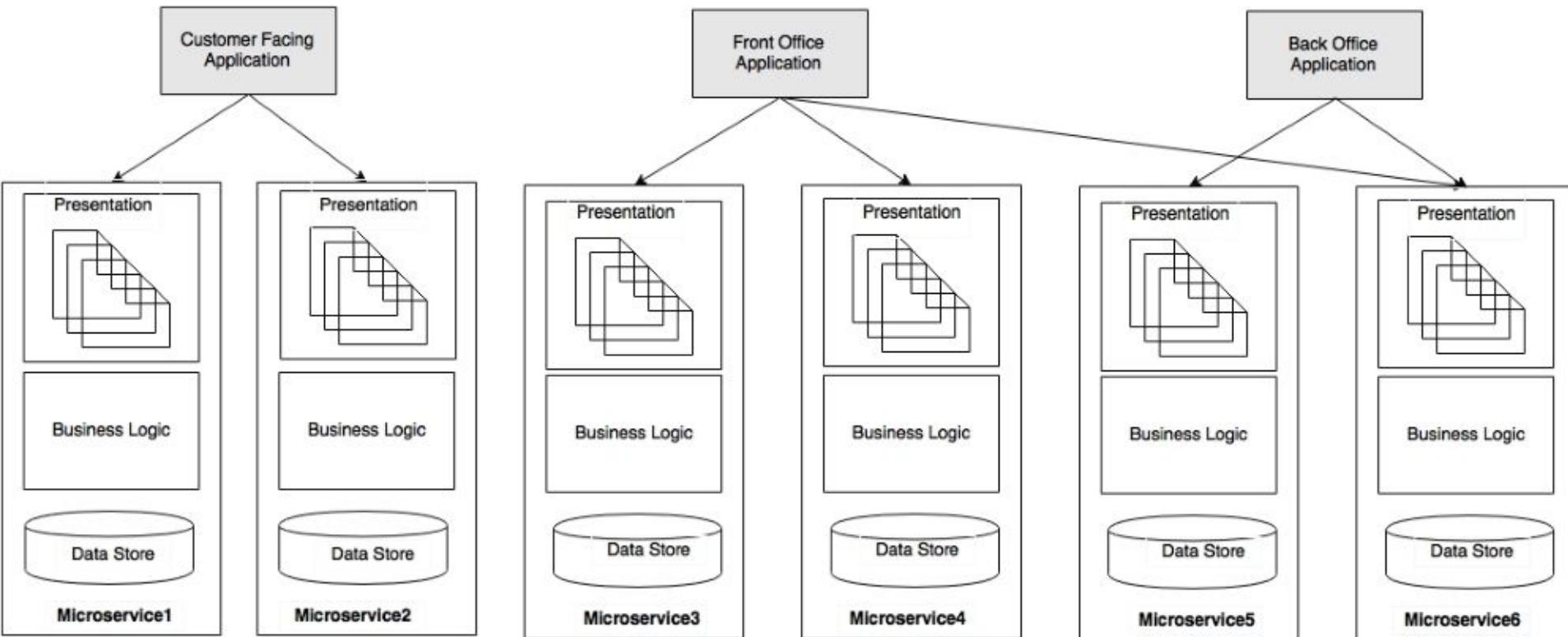
- Though the domain boundaries were well established, all the components were packaged as a single EAR file.
- Since all the components were set to run on a single container, there was no stopping the developers referencing objects across these boundaries.
- Over a period of time, the project teams changed, delivery pressure increased, and the complexity grew tremendously. The developers started looking for quick solutions rather than the right ones.
- Slowly, but steadily, the modular nature of the application went away.

Single Database continued



How can microservices help?

- There are not many improvement opportunities left to support the growing demand of BrownField Airline's business.
 - BrownField Airline was looking to re-platform the system with an evolutionary approach rather than a revolutionary model.
- Microservices is an ideal choice in these situations—for transforming a legacy monolithic application with minimal disruption to the business.



Is there a business case for microservices?

- Service Dependencies
- Physical Boundaries
- Selective Scaling
- Technology Obsolescence

Business Case

- **Service dependencies**
 - While migrating from monolithic applications to microservices, the dependencies are better known, and therefore the architects and developers are much better placed to avoid breaking dependencies and to future-proof dependency issues.
 - Lessons from the monolithic application helps architects and developers to design a better system.

Business Case

- **Physical boundaries**
 - Microservices enforce physical boundaries in all areas including the data store, the business logic, and the presentation layer.
 - Access across subsystems or microservices are truly restricted due to their physical isolation.
 - Beyond the physical boundaries, they could even run on different technologies.

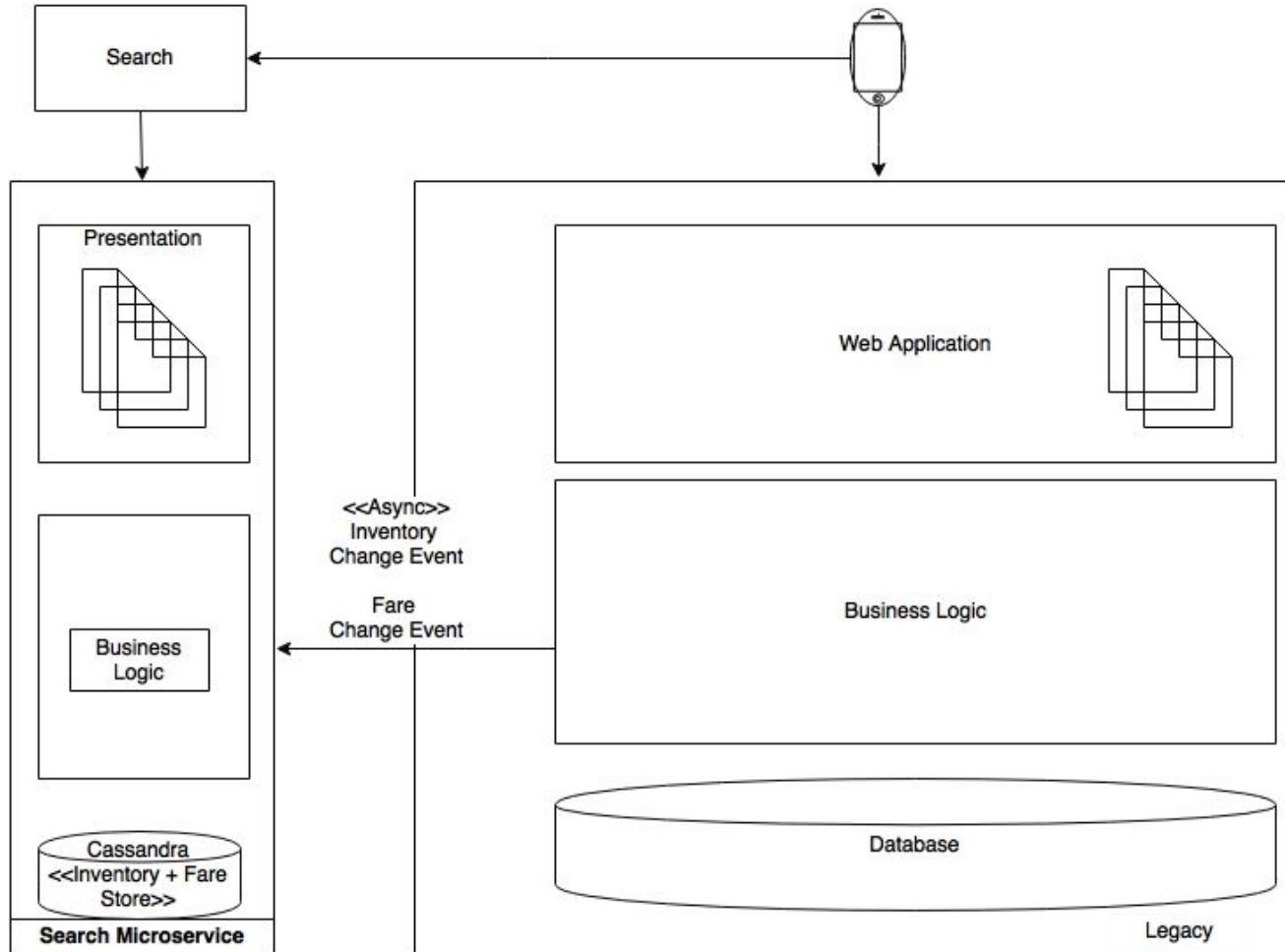
Business Case

- **Selective scaling**
 - Selective scale out is possible in microservices architecture.
 - This provides a much more cost-effective scaling mechanism compared to the Y-scale approach used in the monolithic scenario.

Business Case

- **Technology obsolescence**
 - Technology migrations could be applied at a microservices level rather than at the overall application level.
 - Therefore, it does not require a humongous investment.

Plan the microservices migration



Key Questions to be Answered

- Identification of microservices' boundaries
- Prioritizing microservices for migration
- Handling data synchronization during the transition phase
- Handling user interface integration, working with old and new user interfaces
- Handling of reference data in the new system
- Testing strategy to ensure the business capabilities are intact and correctly reproduced
- Identification of any prerequisites for microservice development such as microservices capabilities, frameworks, processes, and so on

Identify Service Boundaries

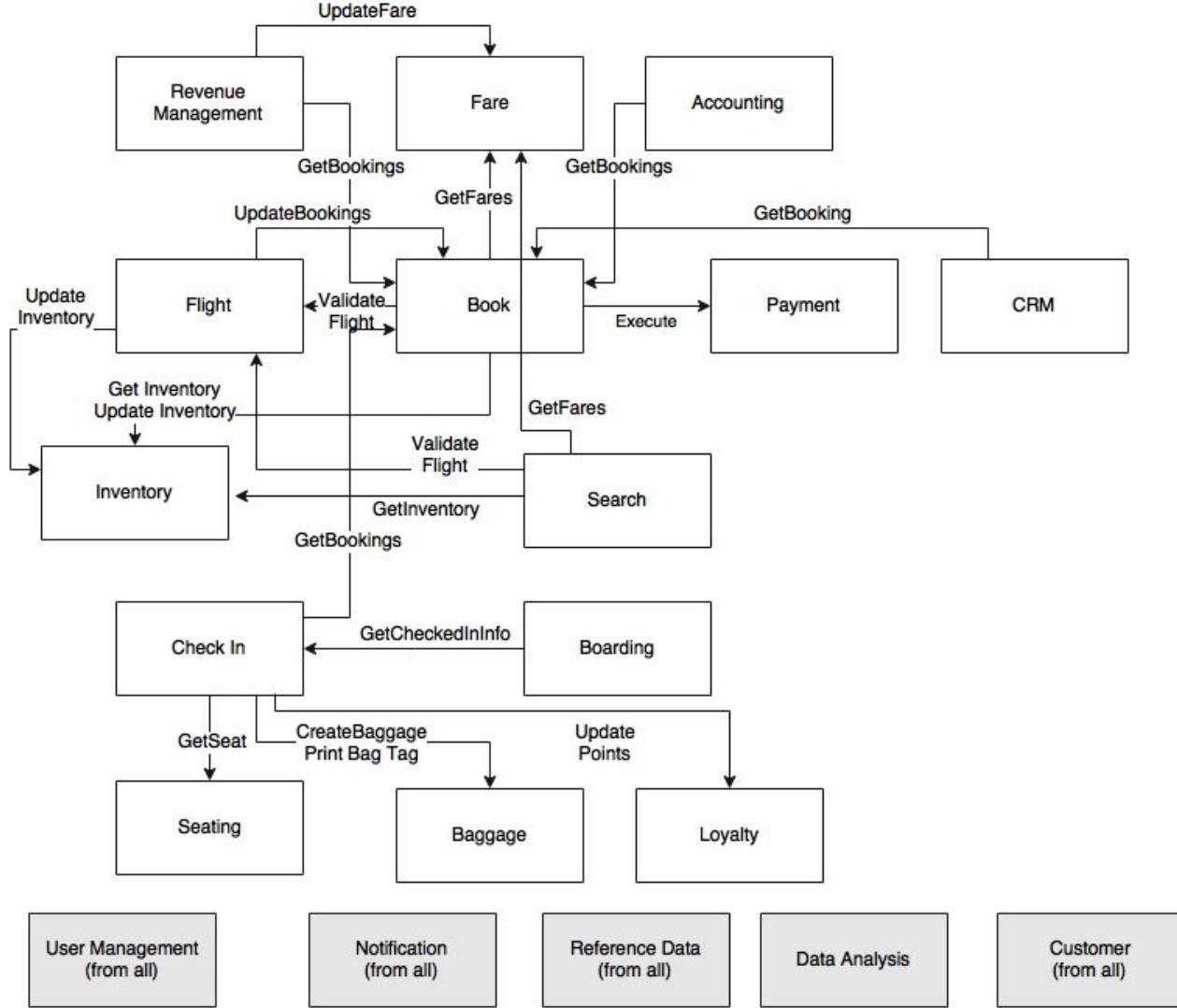
- The first and foremost activity is to identify the microservices' boundaries.
 - This is the most interesting part of the problem, and the most difficult part as well.
 - If identification of the boundaries is not done properly, the migration could lead to more complex manageability issues.
- Like in SOA, a service decomposition is the best way to identify services.
 - However, it is important to note that decomposition stops at a business capability or bounded context.
 - In SOA, service decomposition goes further into an atomic, granular service level.

Identify Service Boundaries

- A top-down approach is typically used for domain decomposition.
- The bottom-up approach is also useful in the case of breaking an existing system, as it can utilize a lot of practical knowledge, functions, and behaviors of the existing monolithic application.

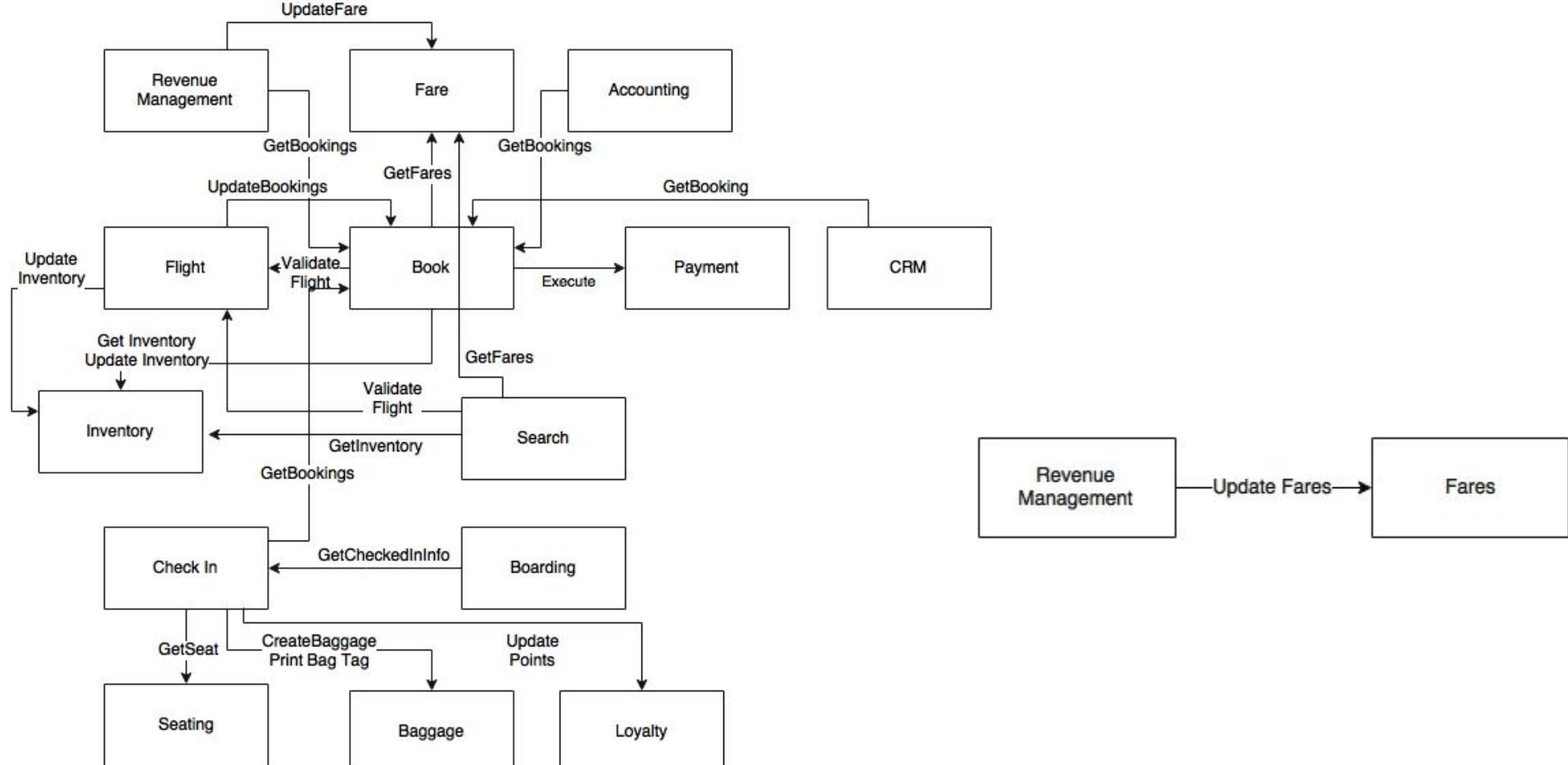
Analyze Dependencies

- Analyzing the manual code and regenerating dependencies.
- Using the experience of the development team to regenerate dependencies.
- Using a Maven dependency graph. There are a number of tools we could use to regenerate the dependency graph, such as PomExplorer, PomParser, and so on.
- Using performance engineering tools such as AppDynamics to identify the call stack and roll up dependencies.



Look for Events (Asynch) as Opposed to Queries (Sync)

- Dependencies could be query-based or event-based.
- Event-based is better for scalable systems. Sometimes, it is possible to convert query-based communications to event-based ones.
- In many cases, these dependencies exist because either the business organizations are managed like that, or by virtue of the way the old system handled the business scenario.



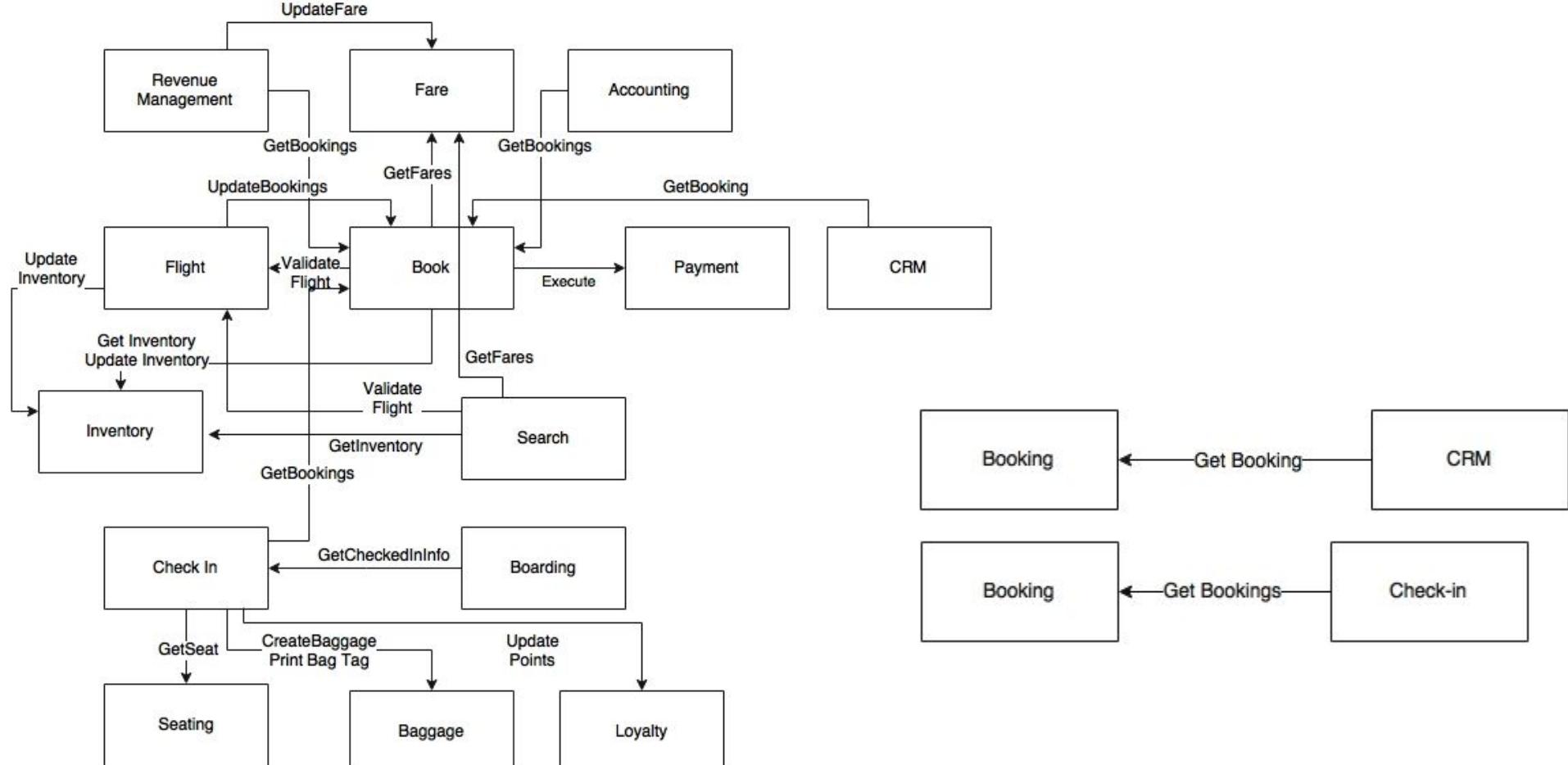
User Management
(from all)

Notification
(from all)

Reference Data
(from all)

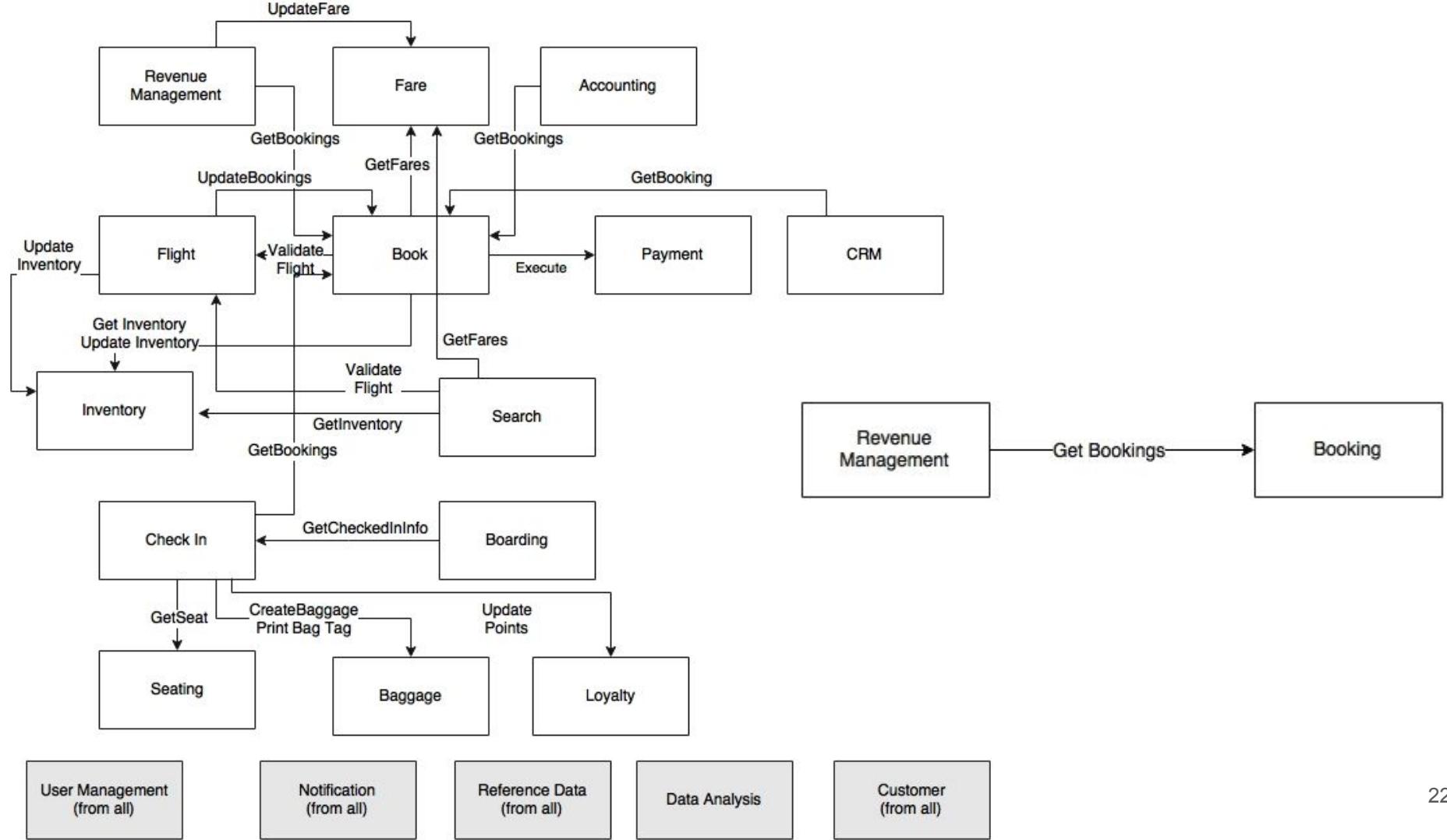
Data Analysis

Customer
(from all)

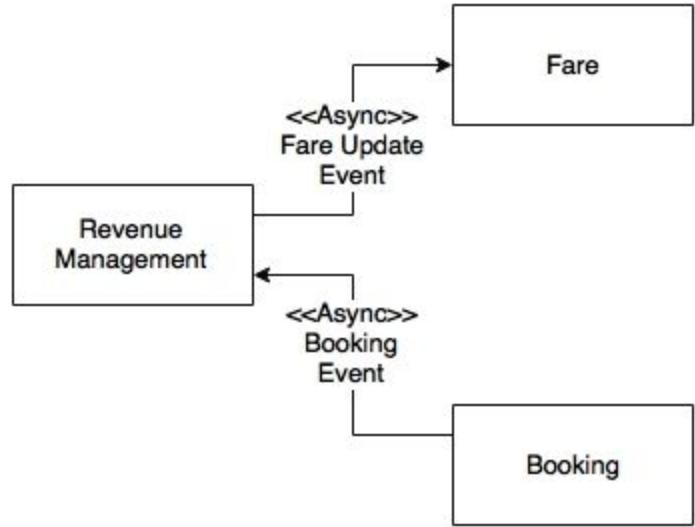


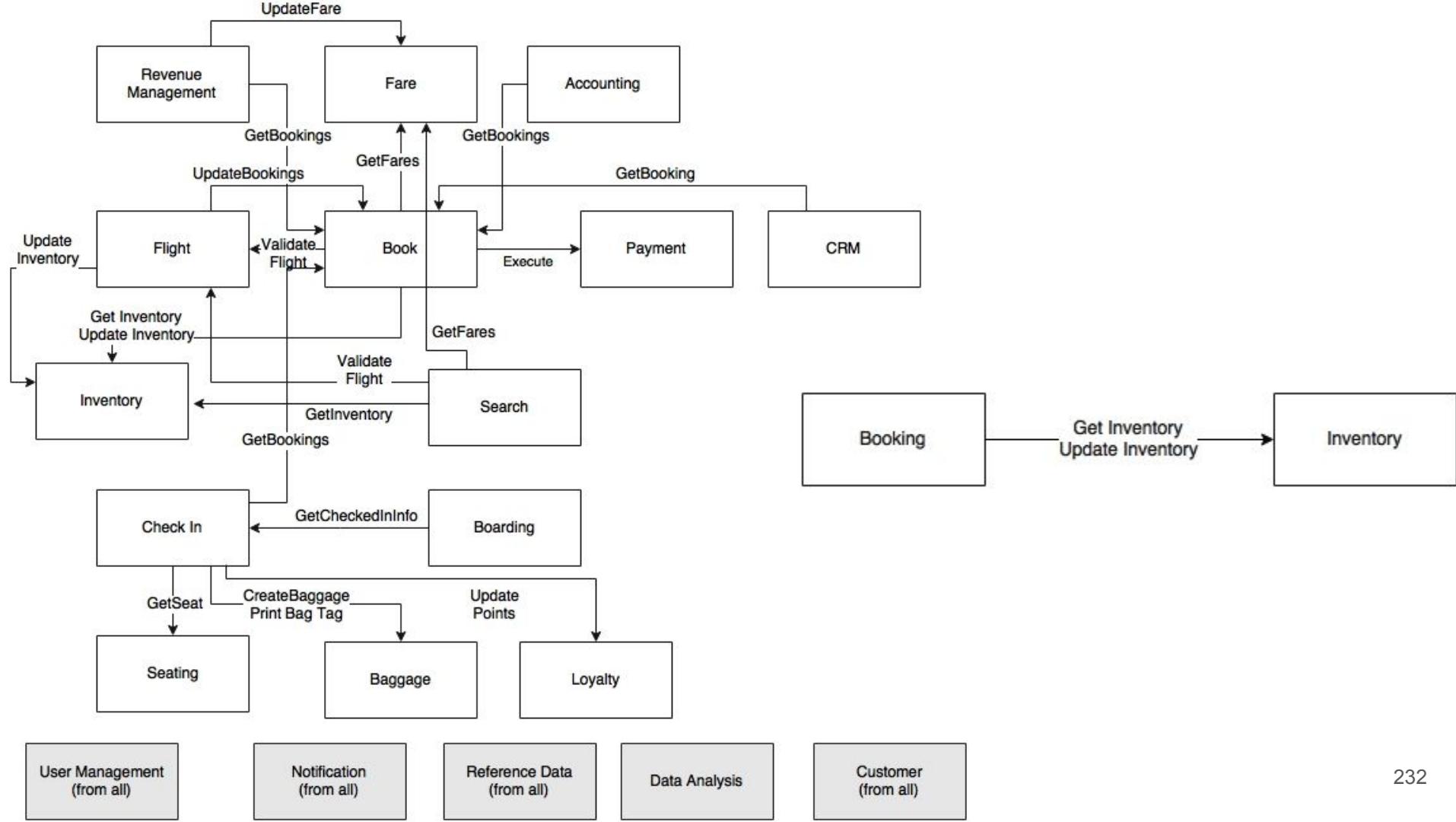
Events as opposed to subscription updates

- Apart from the query model, a dependency could be an update transaction as well. Consider the scenario between Revenue Management and Booking

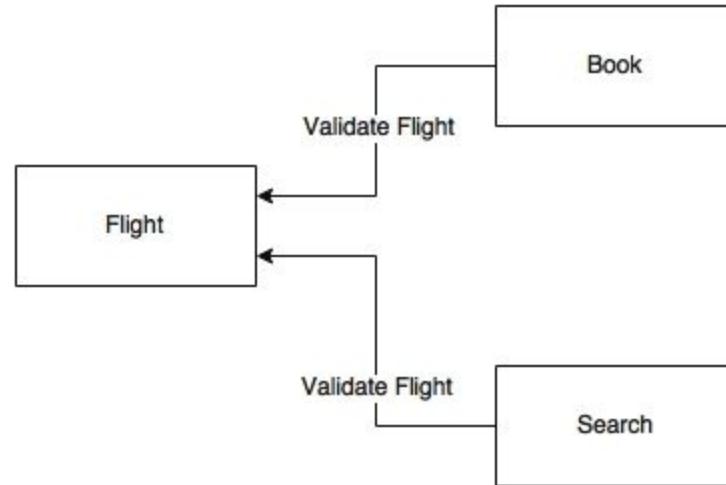


- An alternative approach is to send new bookings and the changes in bookings as soon as they take place in the Booking module as an asynchronous push.
- The same pattern could be applied in many other scenarios such as from Booking to Accounting, from Flight to Inventory, and also from Flight to Booking.
- In this approach, the source service publishes all state-change events to a topic.
- All interested parties could subscribe to this event stream and store locally.
- This approach removes many hard wirings, and keeps the systems loosely coupled.

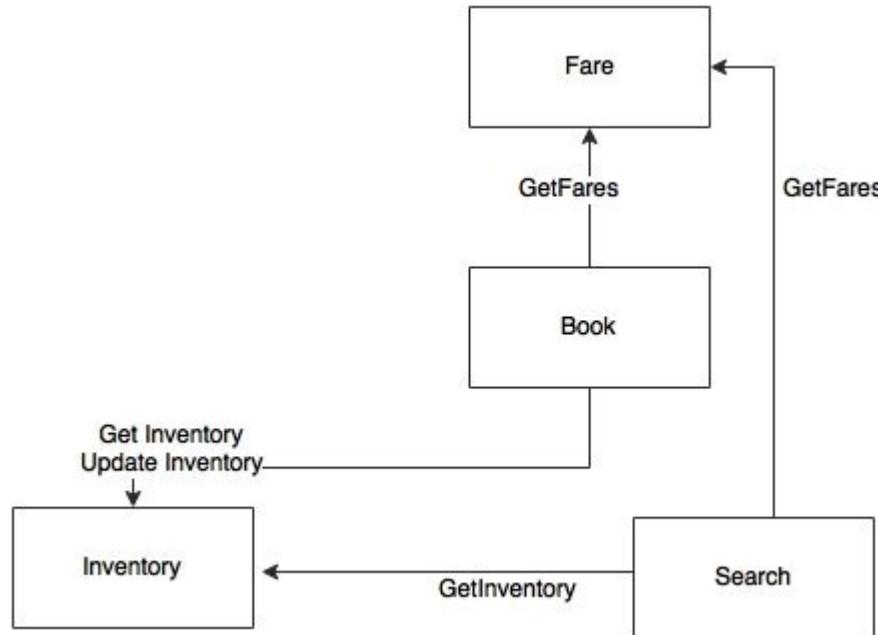


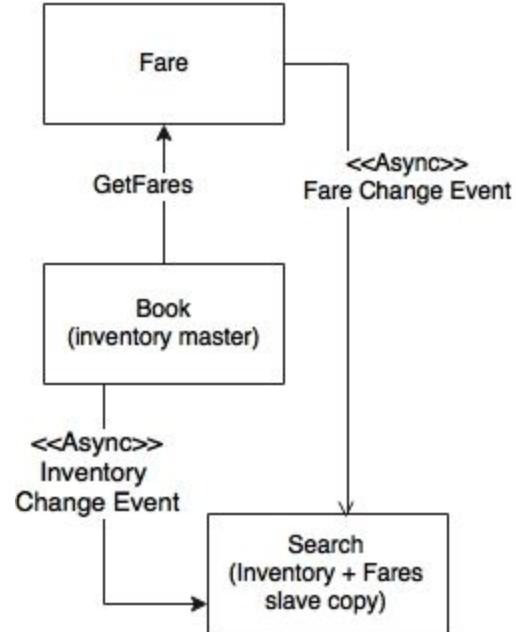


Challenge the requirements

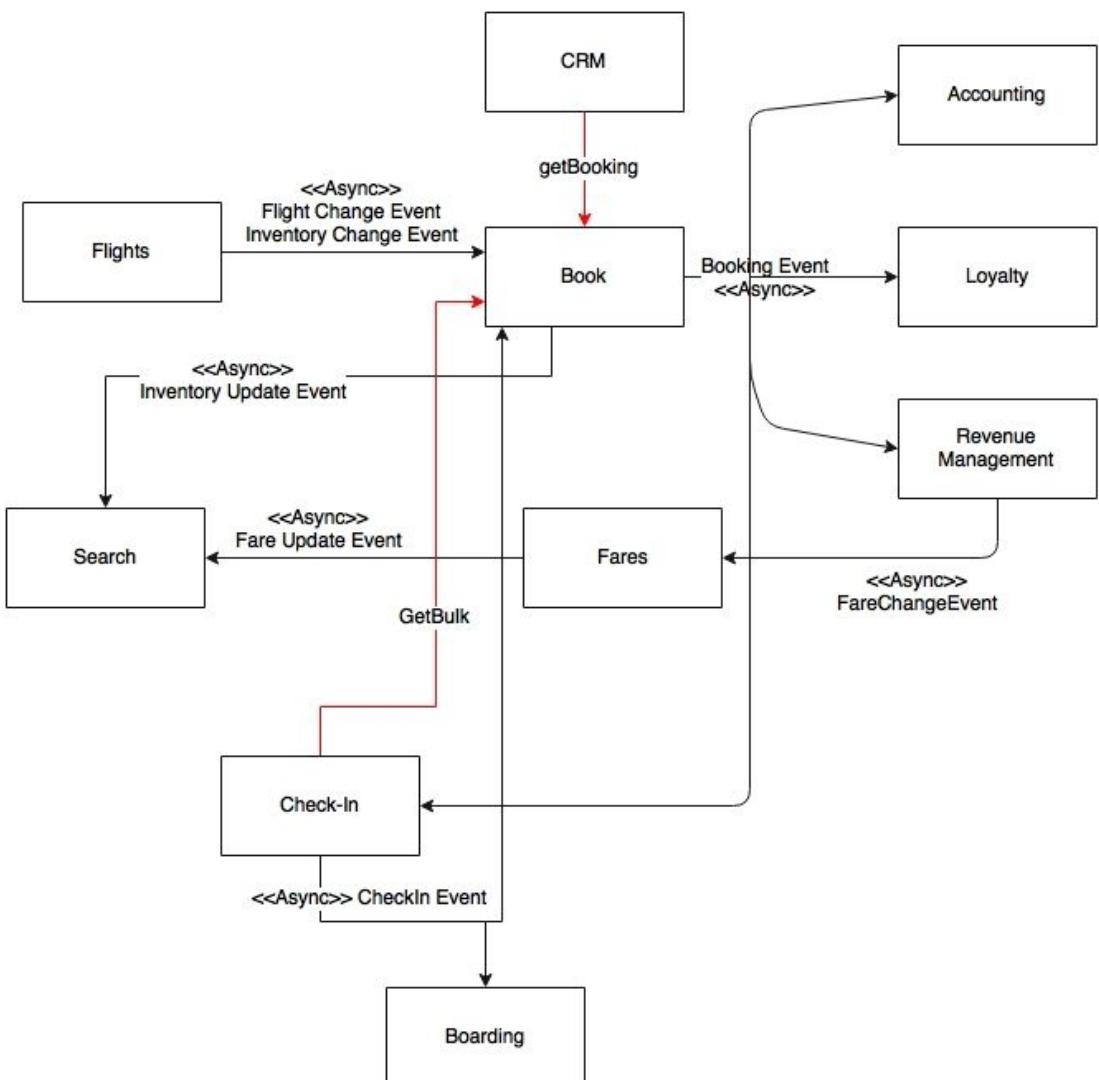


Challenge the service boundaries





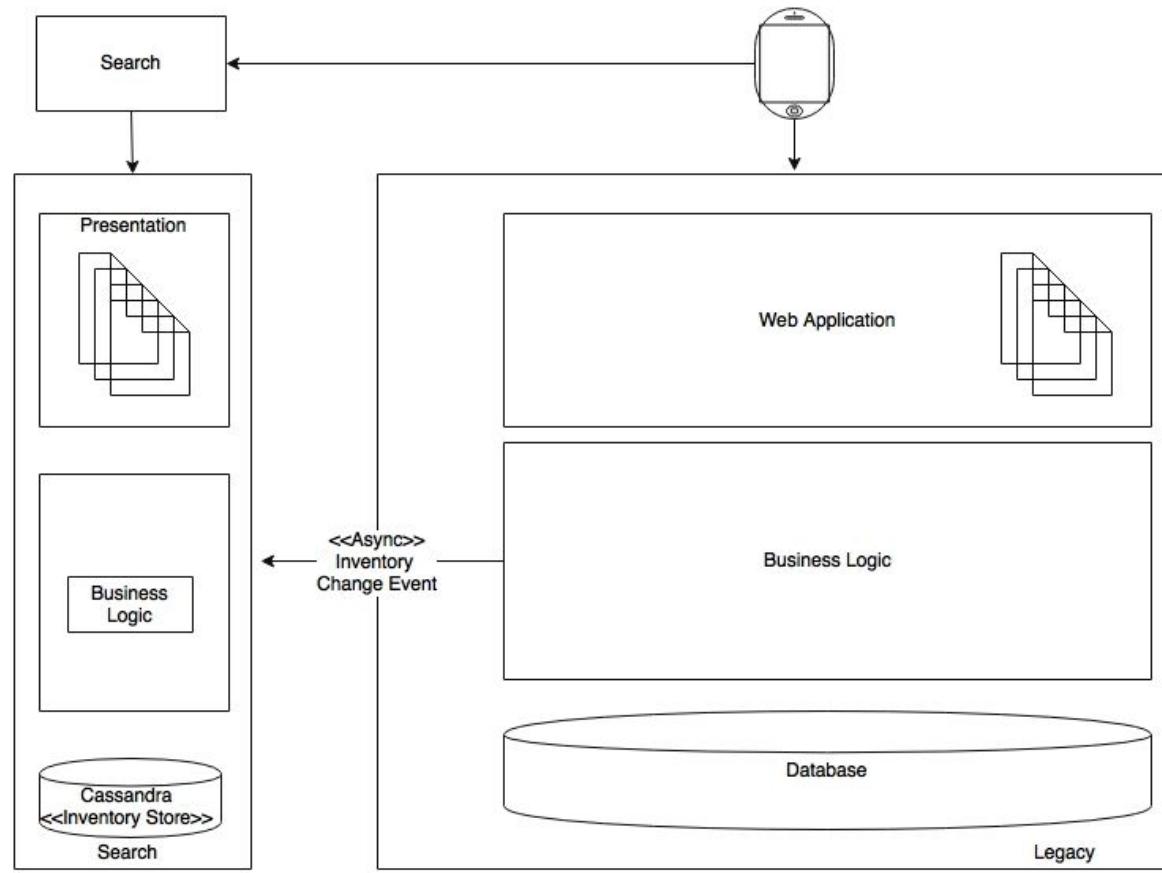
Final Dependency Graph

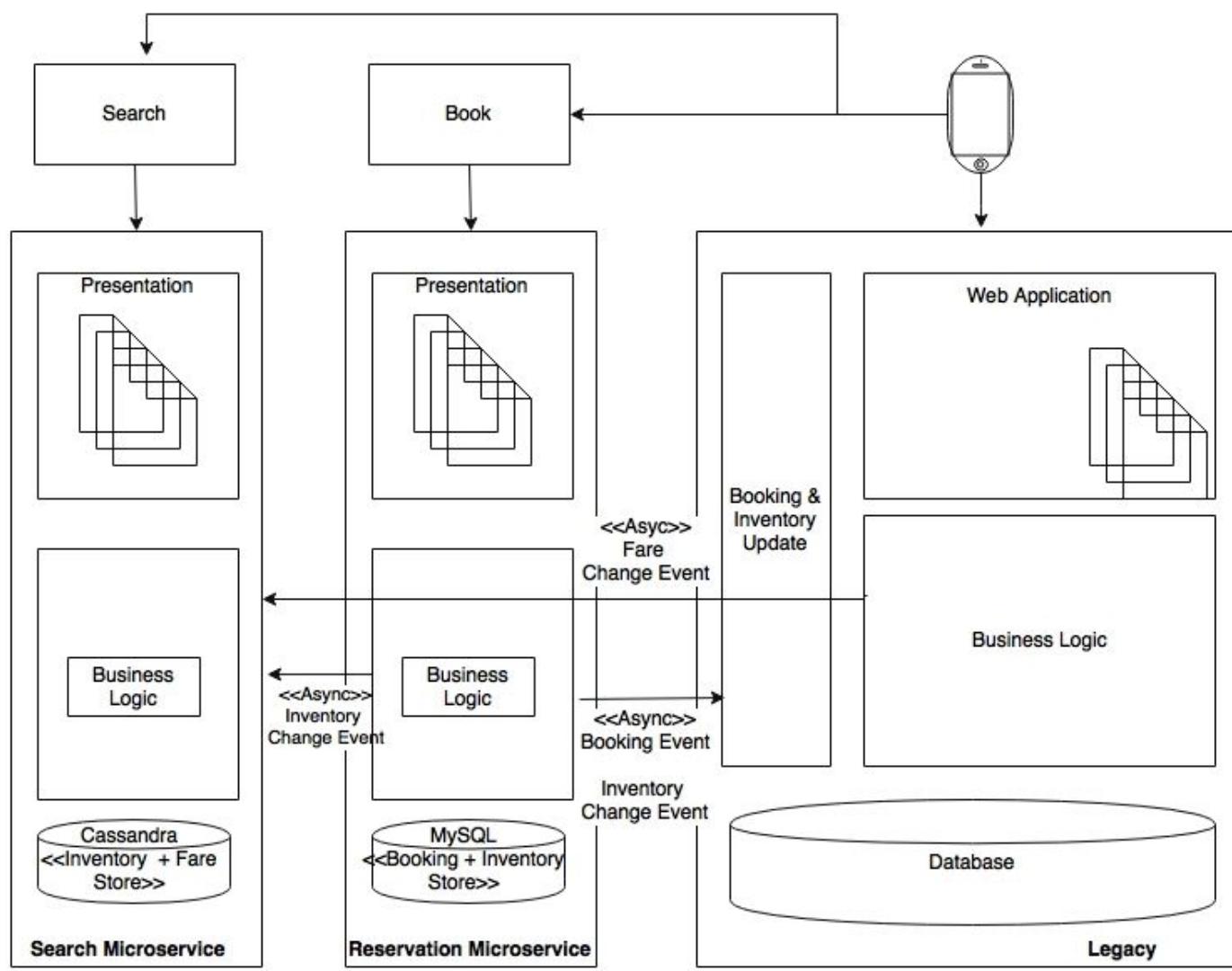


Prioritizing Microservices

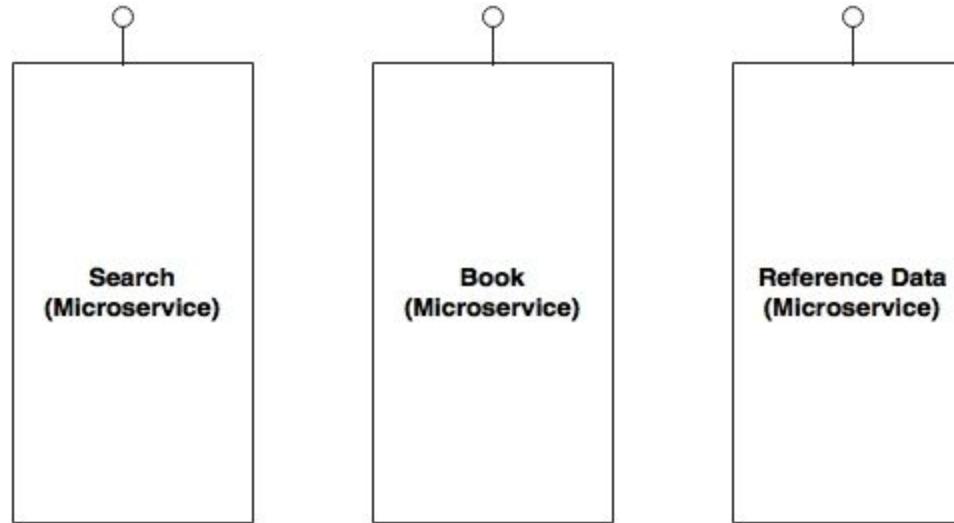
- Dependency
- Transaction volume
- Resource utilization
- Complexity
- Business criticality
- Velocity of changes
- Innovation

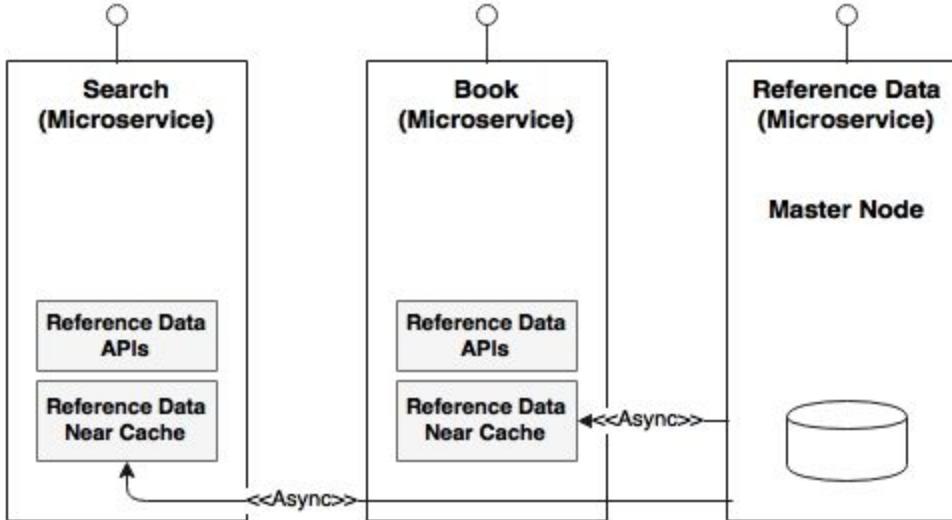
Data Sync During Migration

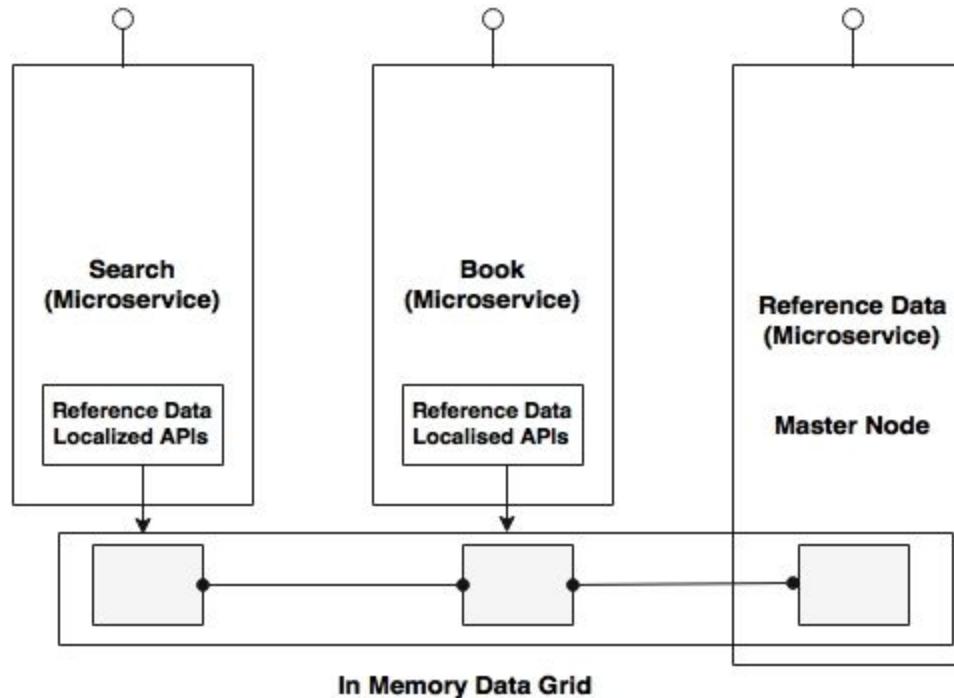




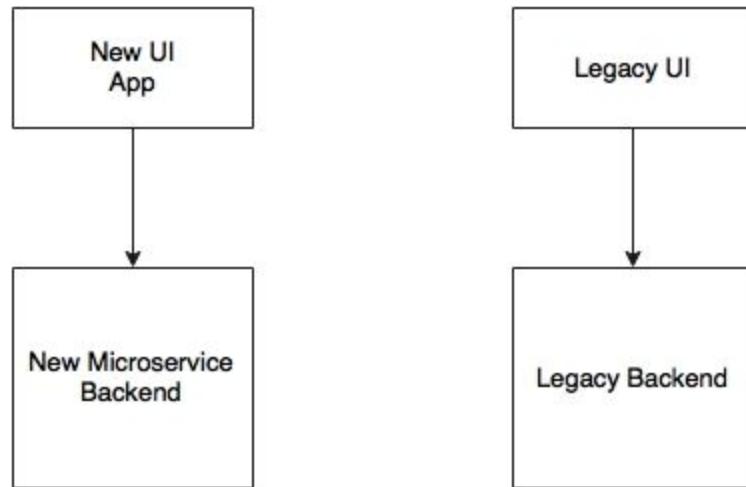
Managing Reference Data

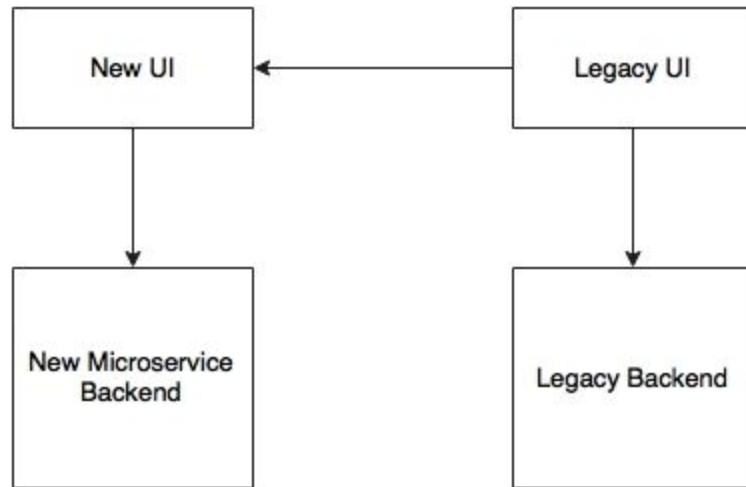


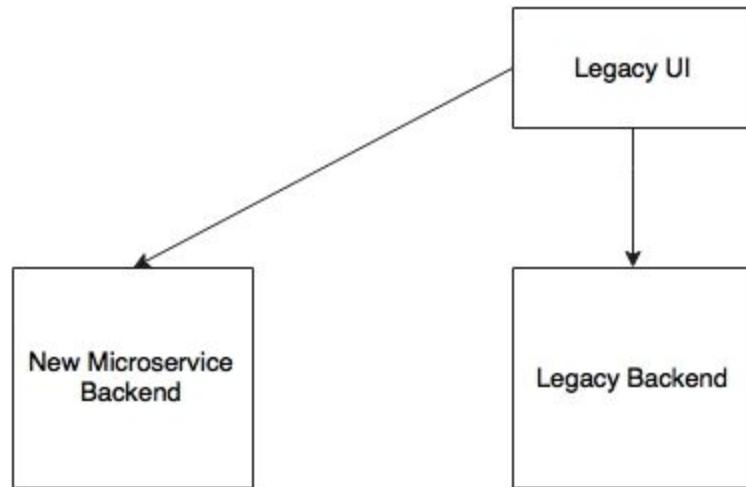




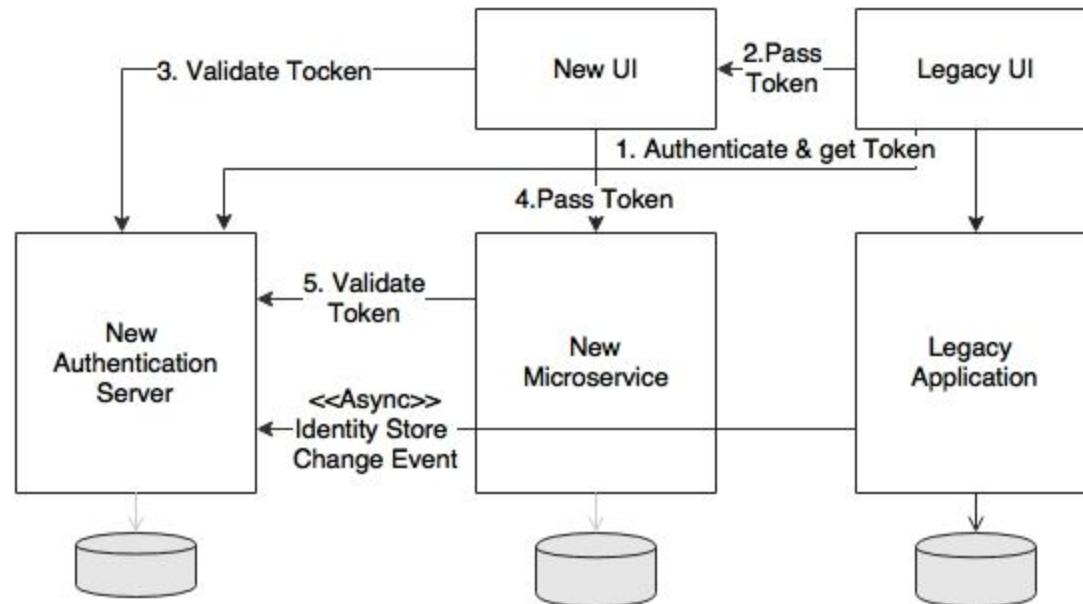
UI's and Web Apps



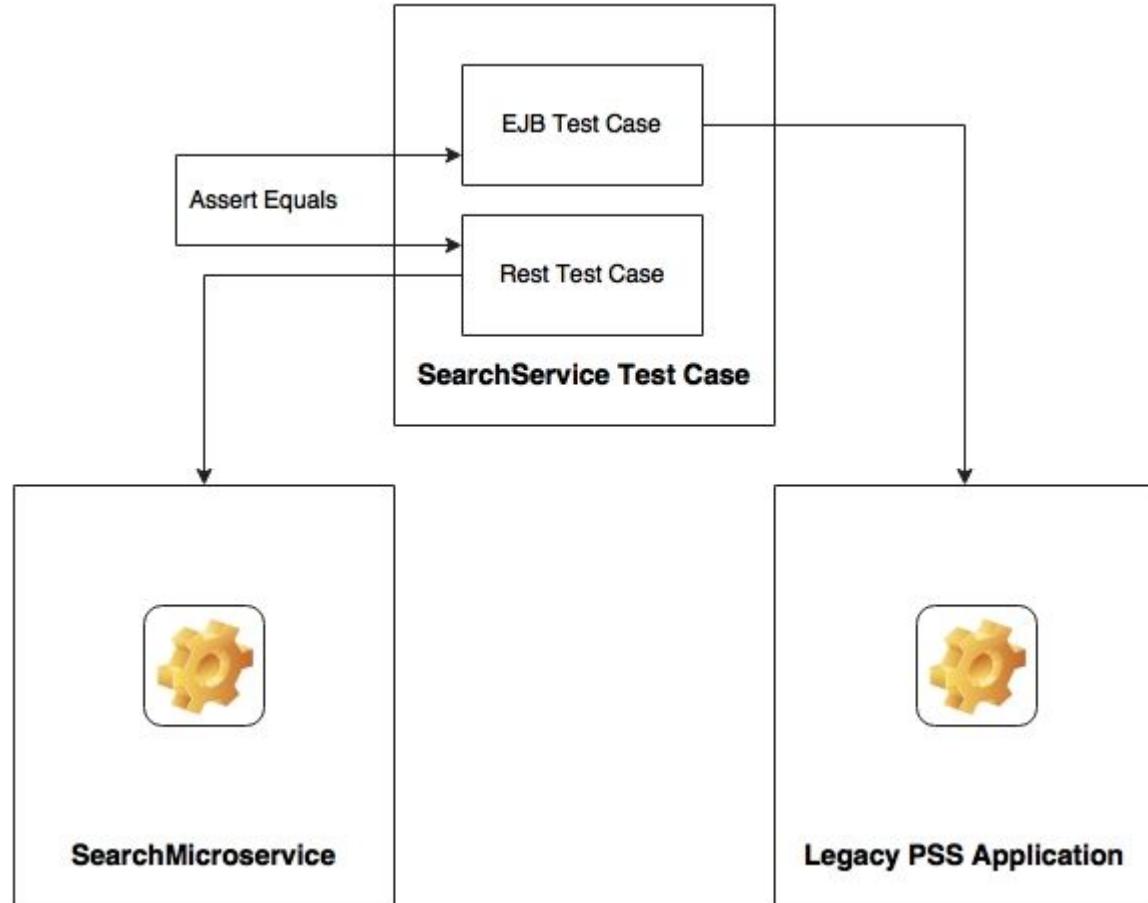




Session Handling and Security



Test Strategy



Migrate only what is necessary

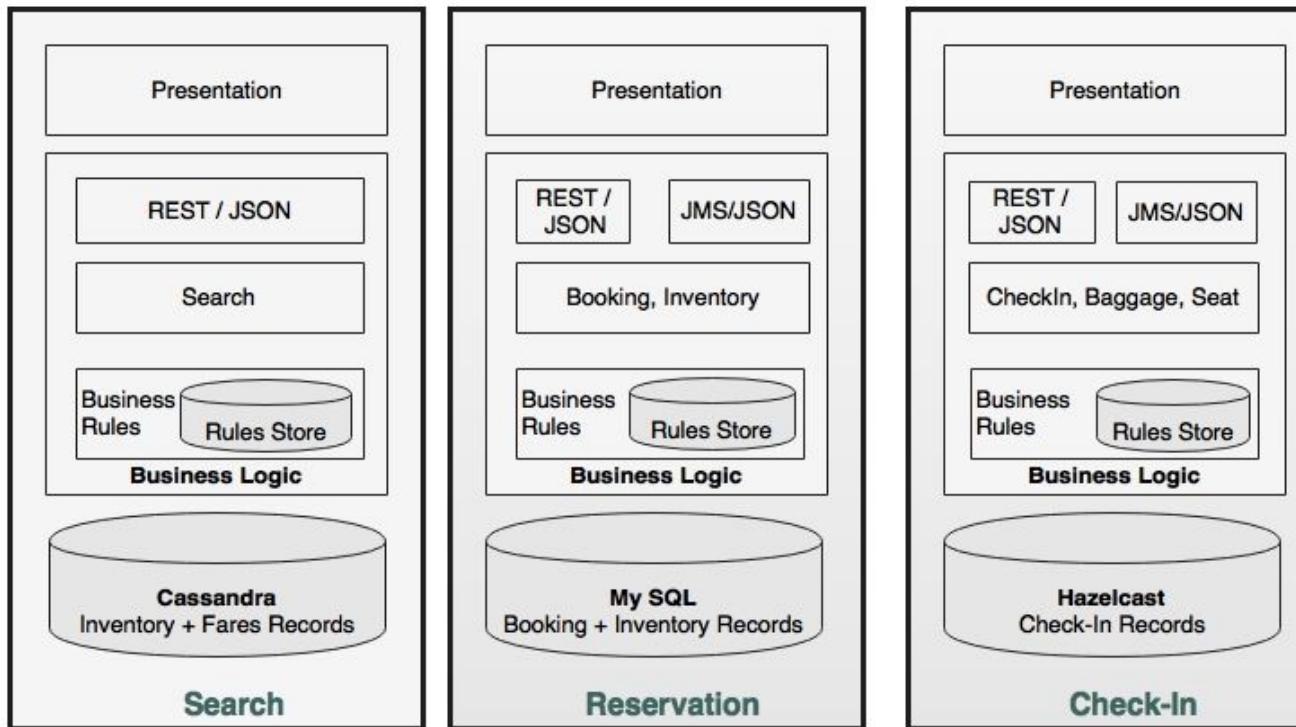
- In the previous sections, we have examined approaches and steps for transforming from a monolithic application to microservices.
 - It is important to understand that it is not necessary to migrate all modules to the new microservices architecture, unless it is really required.
 - A major reason is that these migrations incur cost.

Migrate only what is necessary

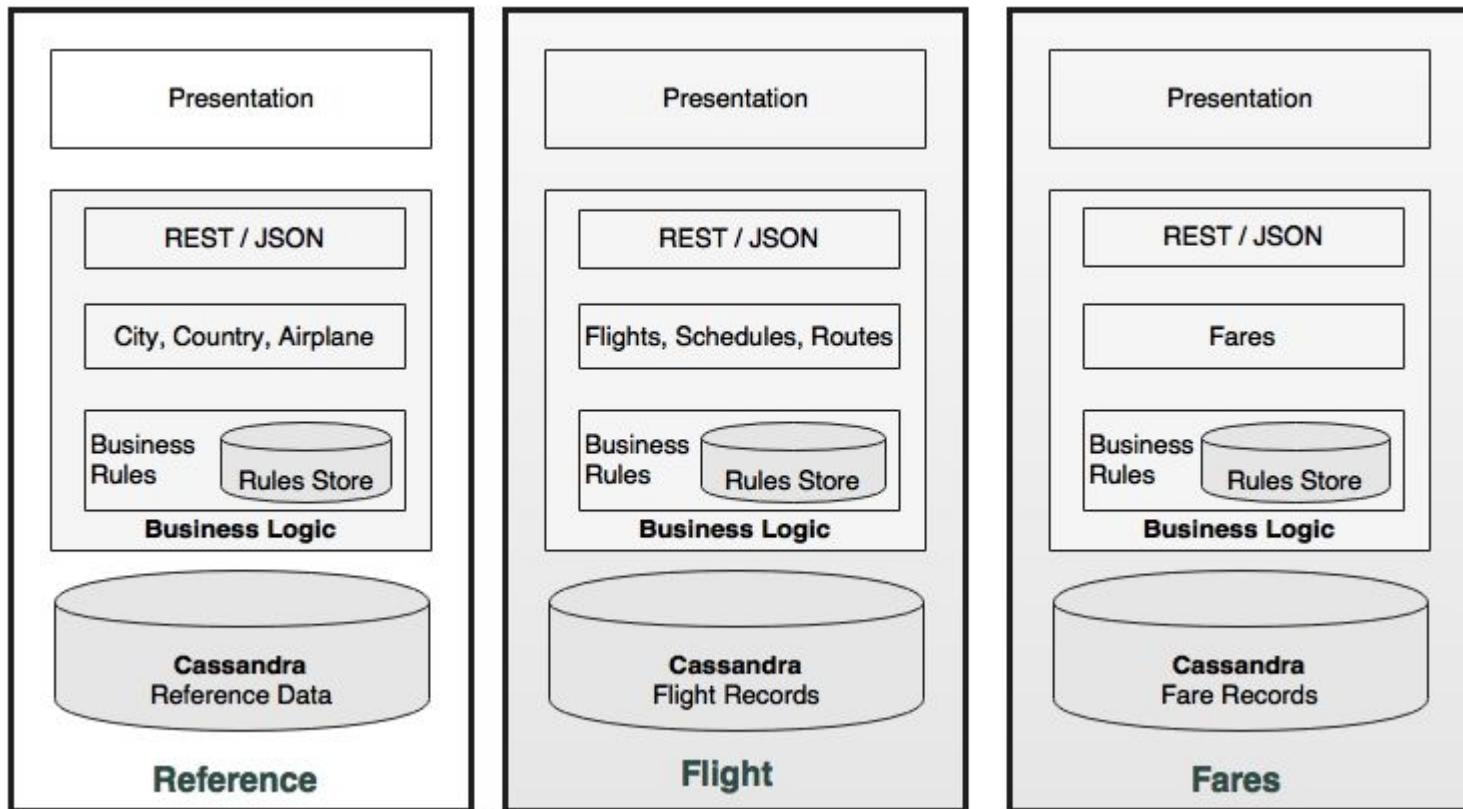
- We will review a few such scenarios here.
 - BrownField has already taken a decision to use an external revenue management system in place of the PSS revenue management function.
 - BrownField is also in the process of centralizing their accounting functions, and therefore, need not migrate the accounting function from the legacy system.
 - Migration of CRM does not add much value at this point to the business.
 - Therefore, it is decided to keep the CRM in the legacy system itself.
 - The business has plans to move to a SaaS-based CRM solution as part of their cloud strategy.
 - Also note that stalling the migration halfway through could seriously impact the complexity of the system.

-

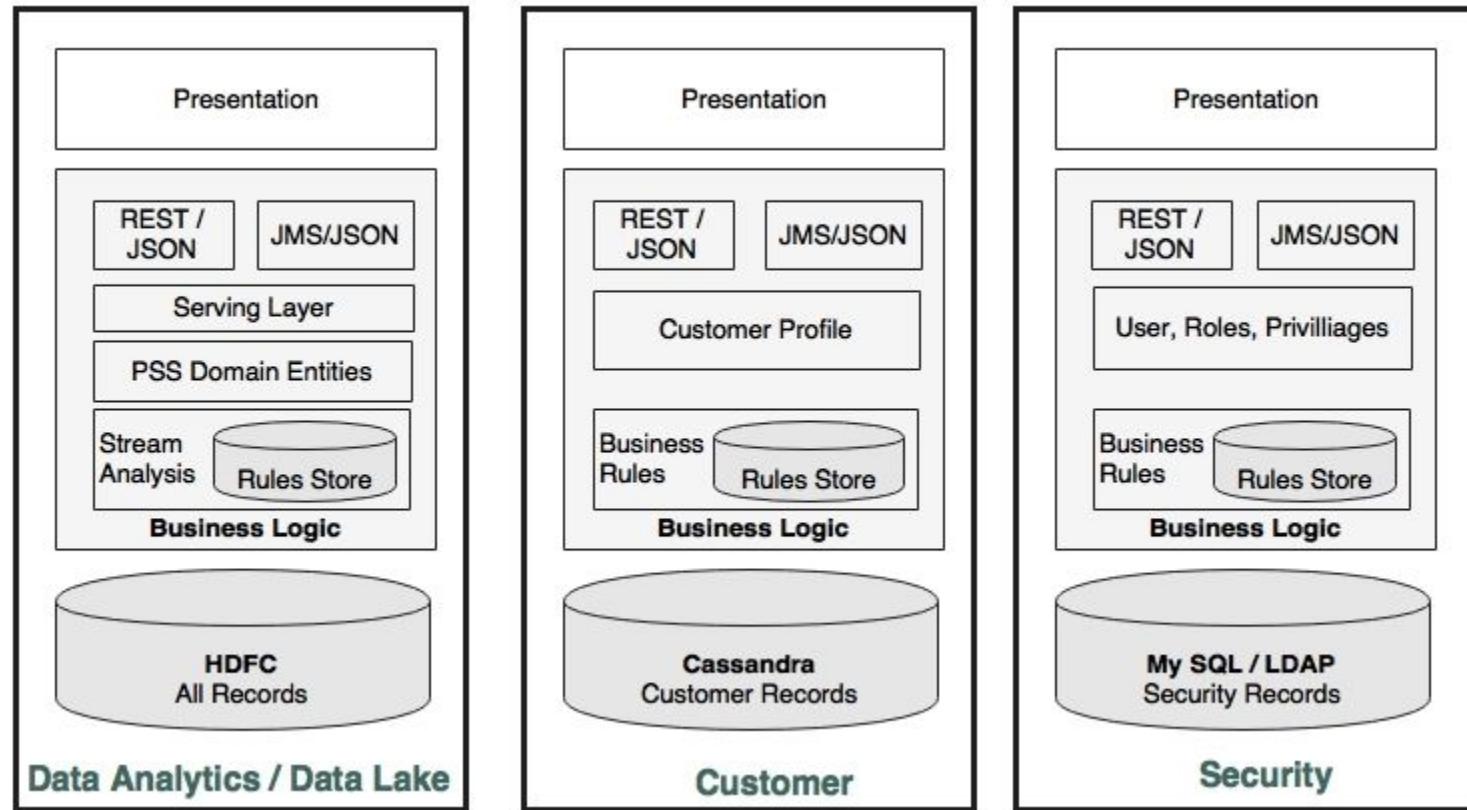
Target Implementation Architecture



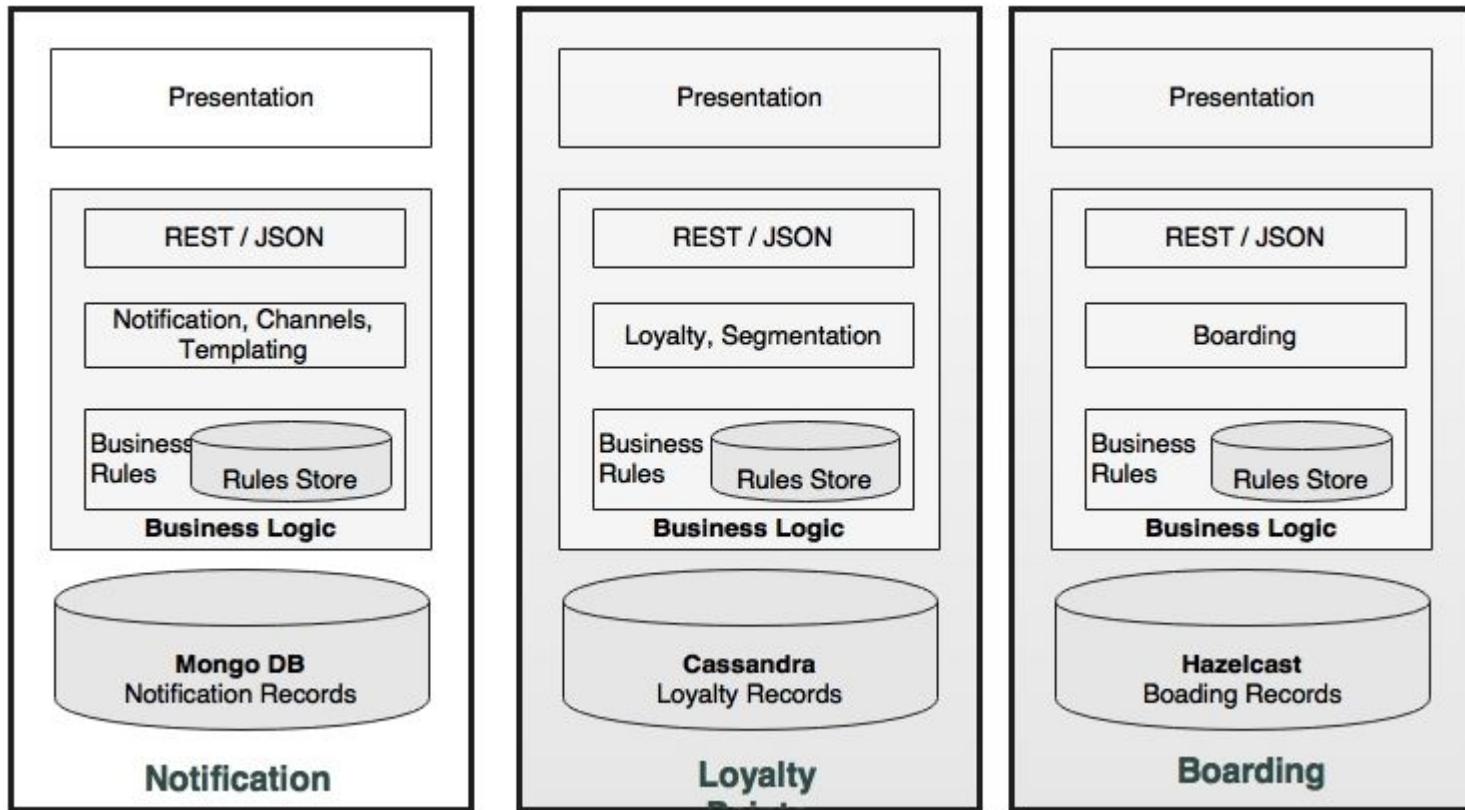
Target Implementation Architecture



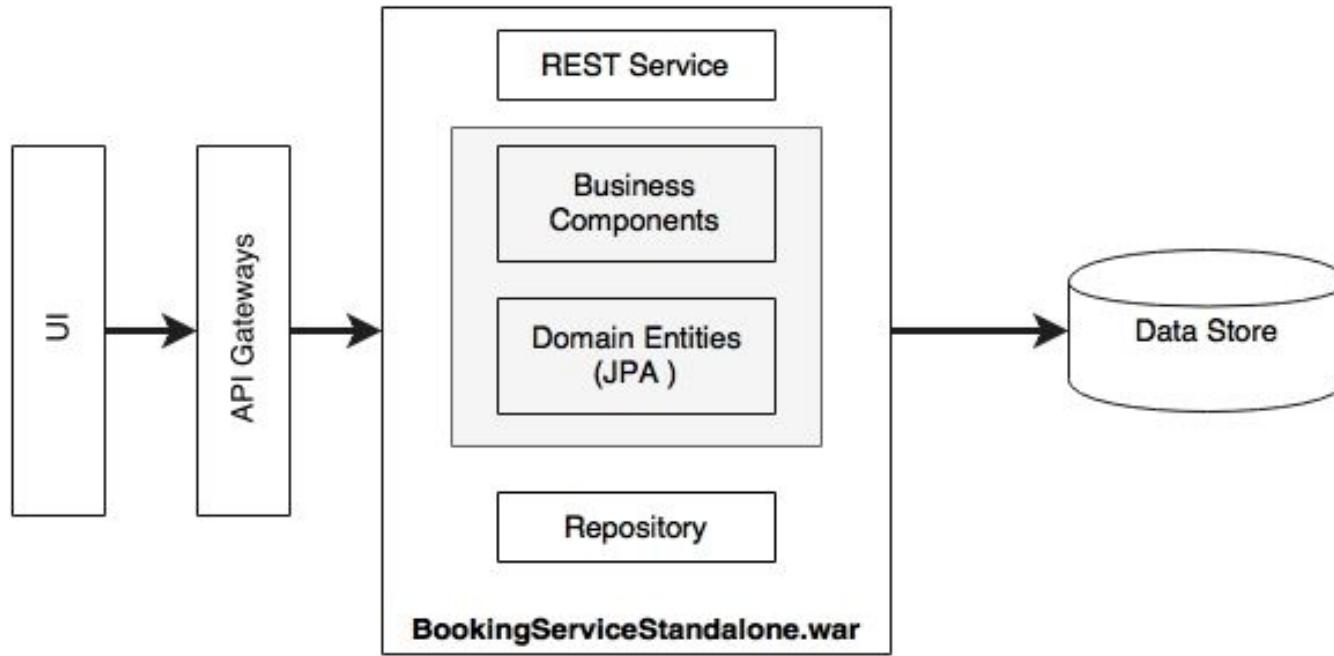
Target Implementation Architecture



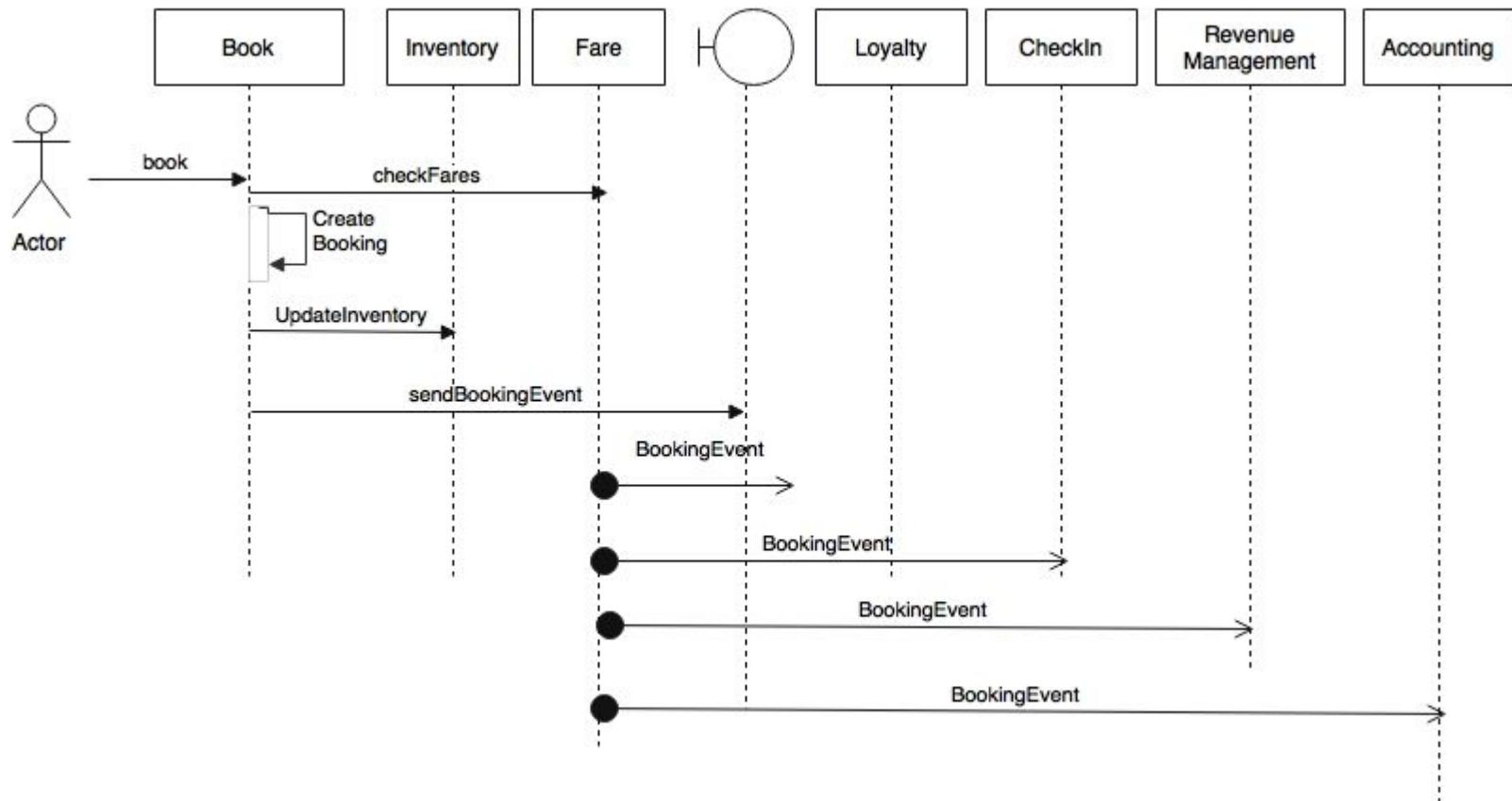
Target Implementation Architecture



Internal layering of microservices



Orchestrating Microservices



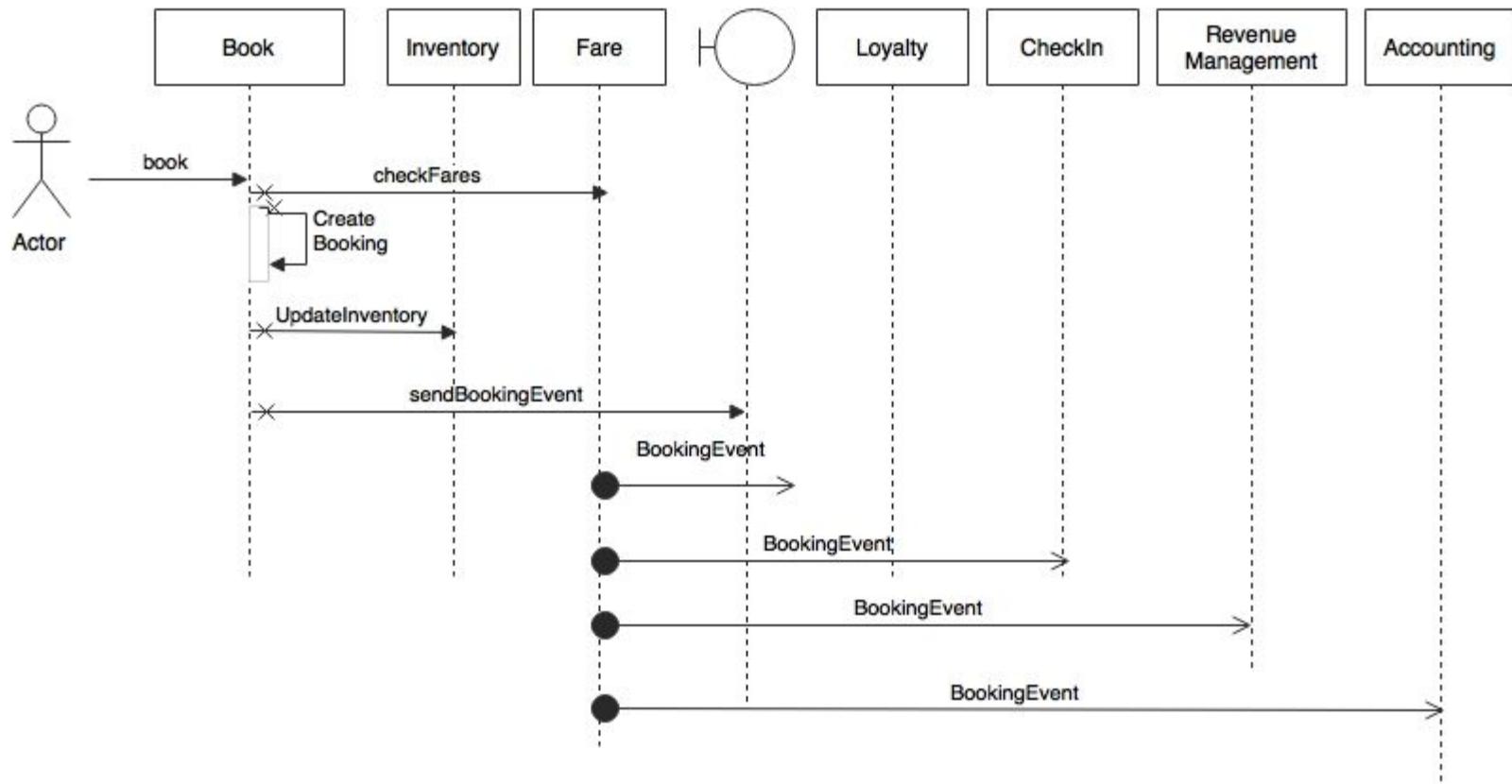
Integration with other systems

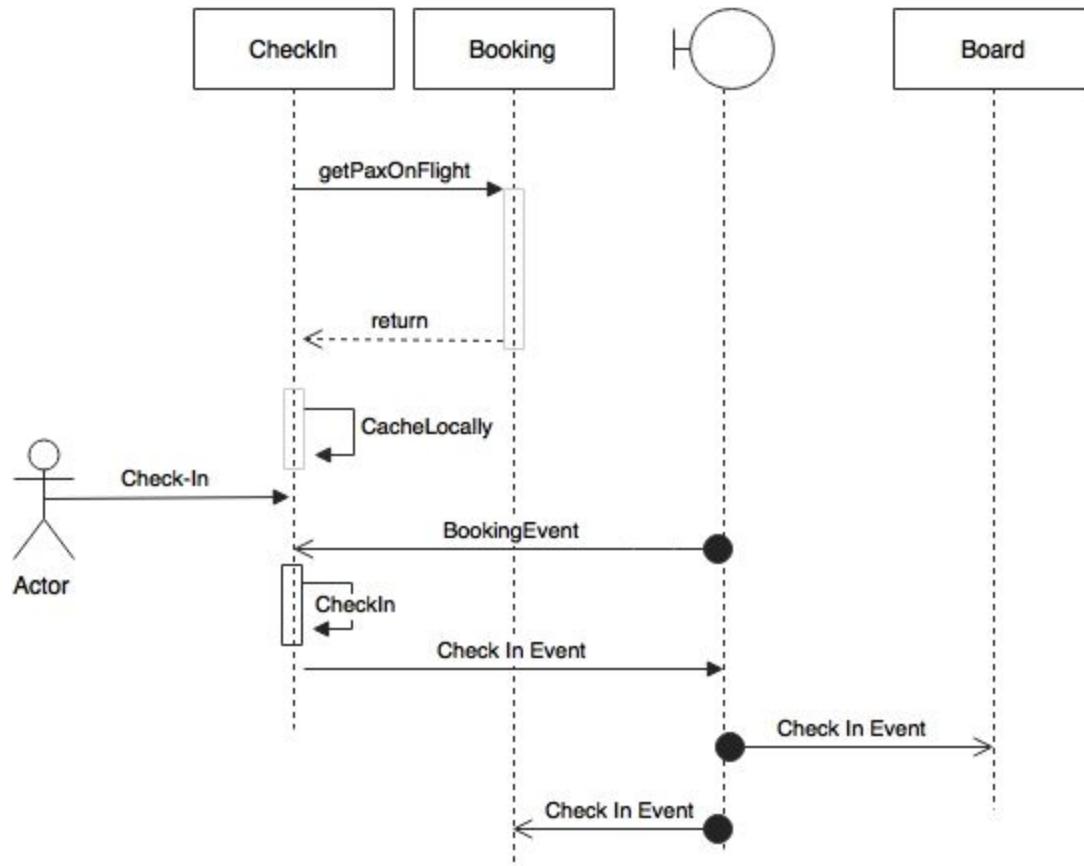
- In the microservices world, we use an API gateway or a reliable message bus for integrating with other non-microservices.
- Let us assume that there is another system in BrownField that needs booking data.
 - Unfortunately, the system is not capable of subscribing to the booking events that the Booking microservice publishes.
 - In such cases, an **Enterprise Application integration (EAI)** solution could be employed, which listens to our booking events, and then uses a native adaptor to update the database.

Managing Shared Libraries

- Certain business logic is used in more than one microservice.
- Search and Reservation, in this case, use inventory rules.
- In such cases, these shared libraries will be duplicated in both the microservices.

Exception Handling





Target View (Lab)

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

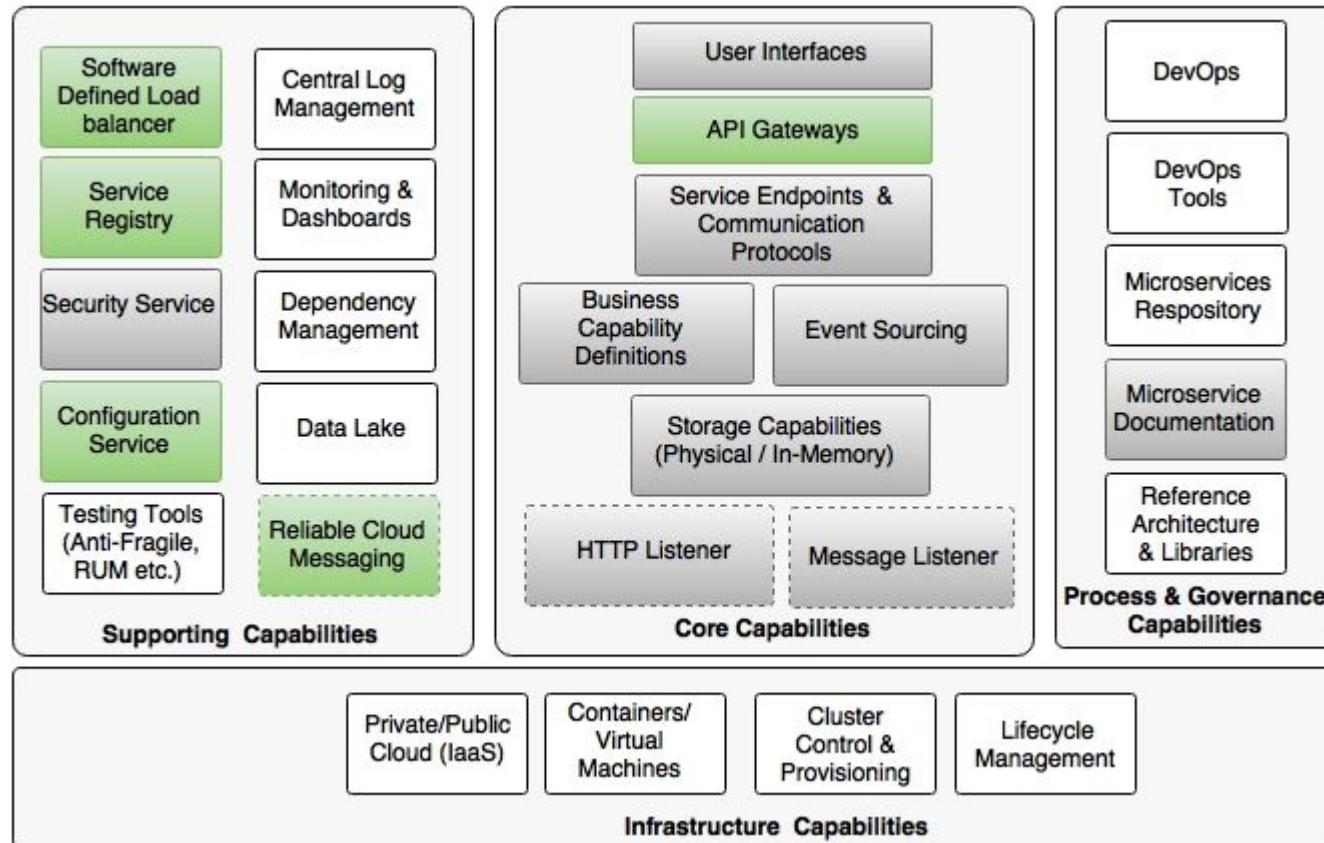
Setting Up Environment for BrownField PSS

Decision Point: Spring Cloud or Kubernetes?

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

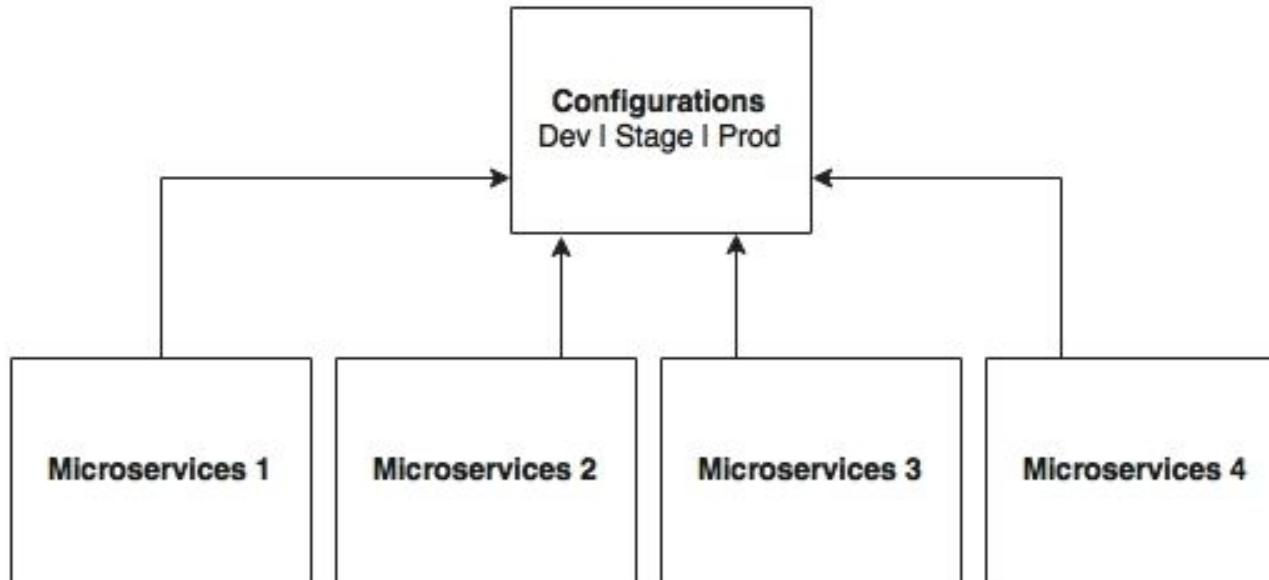
```
@PropertySource("file:${CONF_DIR}/application.properties")  
public class ApplicationConfig {  
}
```

- 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

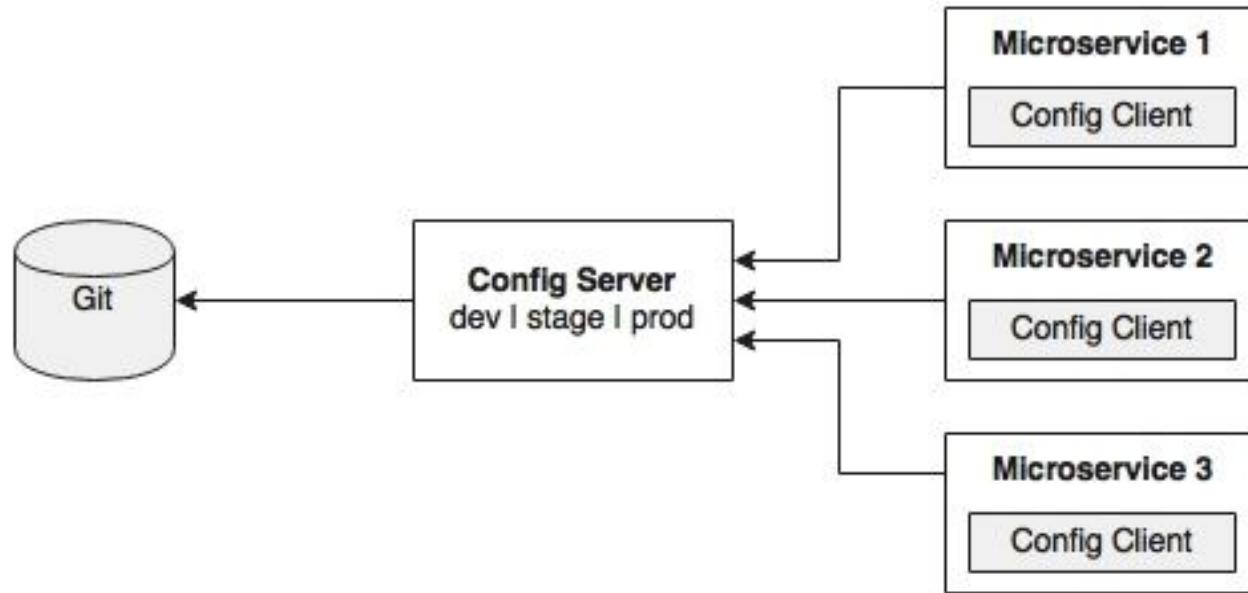
- Zookeeper Configuration**

- 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 - <https://jmp.sh/4LfCxpD>

Understanding the Config Server URL

- In the previous section, we used `http://localhost:8888/application/default/master` to explore the properties.
- How do we interpret this URL?

Understanding the Config Server URL

```
application-development.properties  
application-production.properties
```

These are accessible using the following URLs respectively:

- `http://localhost:8888/application/development`
- `http://localhost:8888/application/production`

In the preceding example, all the following three URLs point to the same configuration:

- `http://localhost:8888/application/default`
- `http://localhost:8888/application/master`
- `http://localhost:8888/application/default/master`

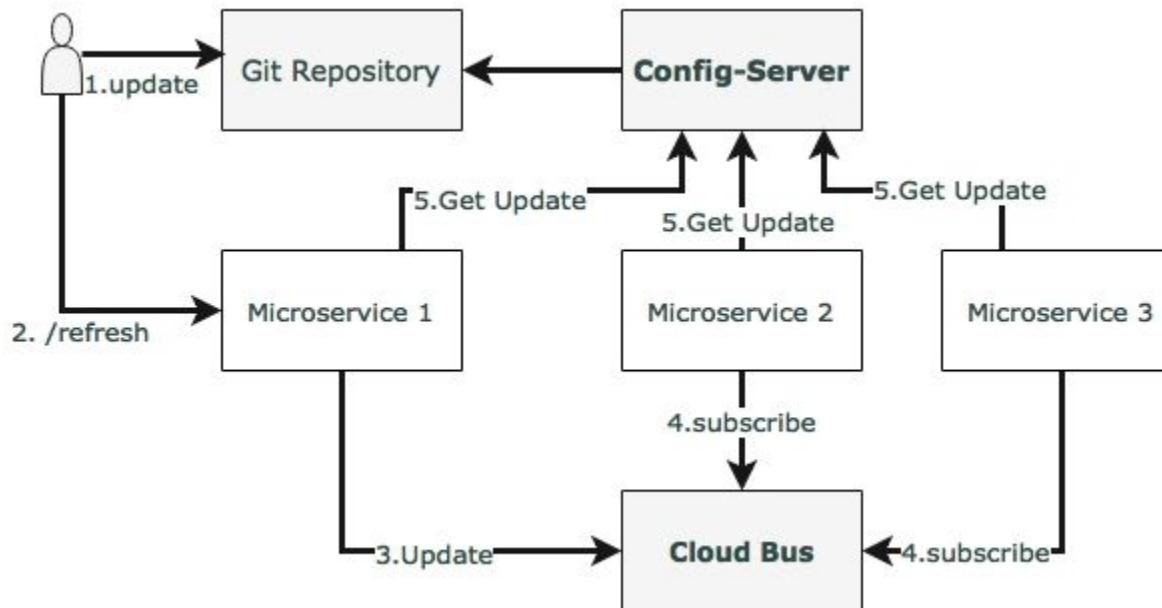
Accessing the Config Server from Clients

Lab 12 -

Handling configuration changes

Lab 13 -

Spring Cloud Bus for propagating configuration changes

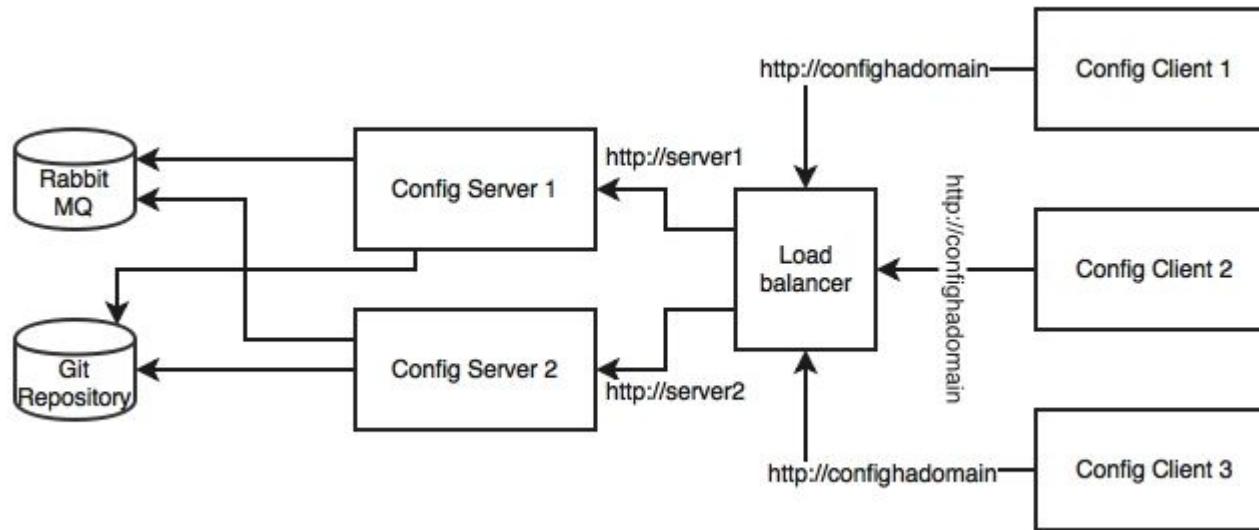


Spring Cloud Bus for propagating configuration changes

Lab 14 -

Setting Up High Availability for the Config server

The following diagram shows a high availability architecture for the Config server:



Setting Up the High Availability for the Config Server

- In order to make the Config server highly available, we need multiple instances of the Config servers.
- Since the Config server is a stateless HTTP service, multiple instances of configuration servers can be run in parallel.
- Based on the load on the configuration server, a number of instances have to be adjusted.

Monitoring the Config Server Health

- The health of the server can be monitored using the following actuator URL:
`http://localhost:8888/health`.

Config Server for configuration files

- The name, profile, and label have the same meanings as explained earlier. The path indicates the file name such as `logback.xml`.
- This is achievable by using the URL format as follows: `/{{name}}/{{profile}}/{{label}}/{{path}}` .

Completing Changes to use the Config Server

```
private static final String FareURL = "/fares";  
@Value("${fares-service.url}")  
  
private String fareServiceUrl;  
  
Fare = restTemplate.getForObject(fareServiceUrl+FareURL  
+"/get?flightNumber="+record.getFlightNumber()+"&flightDate="+record.g  
etFlightDate(), Fare.class);
```

- As shown in the preceding code snippet, the Fare service URL is fetched through a new property: `fares-service.url`.
- We are not externalizing the queue names used in the Search, Booking, and Check-in services at the moment. Later in this session, these will be changed to use Spring Cloud Streams.

Feign as a declarative REST client

- In the Booking microservice, there is a synchronous call to Fare.
- `RestTemplate` is used for making the synchronous call.
- When using `RestTemplate`, the URL parameter is constructed programmatically, and data is sent across to the other service.
- In more complex scenarios, we will have to get to the details of the HTTP APIs provided by `RestTemplate` or even to APIs at a much lower level.

Feign as a declarative REST client

```
Fare fare = restTemplate.getForObject(FareURL  
+ "/get?flightNumber="+record.getFlightNumber()+"&flightDate="+record.getFlightDate(), Fare.class);
```

```
<dependency>  
    <groupId>org.springframework.cloud </groupId>  
    <artifactId>spring-cloud-starter-feign </artifactId>  
</dependency>
```

Feign as a declarative REST client



Cloud Routing

- Zuul

Intelligent and programmable routing with spring-cloud-netflix Zuul

- Ribbon

Client side load balancing with spring-cloud-netflix and Ribbon

- Feign

Declarative REST clients with spring-cloud-netflix Feign

Feign as a declarative REST client

```
@FeignClient(name="fares-proxy", url="localhost:8080/fares")  
public interface FareServiceProxy {  
    @RequestMapping(value = "/get", method=RequestMethod.GET)  
    Fare getFare(@RequestParam(value="flightNumber") String flightNumber,  
    @RequestParam(value="flightDate") String flightDate);  
}
```

Feign as a declarative REST client

```
Fare = fareServiceProxy.getFare(record.getFlightNumber(),  
record.getFlightDate());
```

Ribbon for Load Balancing

- In the previous setup, we were always running with a single instance of the microservice.
- The URL is hardcoded both in client as well as in the service-to-service calls.

Eureka for registration and discovery

- If there is a large number of microservices, and if we want to optimize infrastructure utilization, we will have to dynamically change the number of service instances and the associated servers. It is not easy to predict and preconfigure the server URLs in a configuration file.
- When targeting cloud deployments for highly scalable microservices, static registration and discovery is not a good solution considering the elastic nature of the cloud environment.
- In the cloud deployment scenarios, IP addresses are not predictable, and will be difficult to statically configure in a file. We will have to update the configuration file every time there is a change in address.

Understanding dynamic service registration and discovery

- Dynamic registration is primarily from the service provider's point of view.
- With dynamic registration, when a new service is started, it automatically enlists its availability in a central service registry.
- Similarly, when a service goes out of service, it is automatically delisted from the service registry.
- The registry always keeps up-to-date information of the services available, as well as their metadata.

Understanding dynamic service and registration and discovery

Cloud Discovery

- Eureka Discovery

Service discovery using spring-cloud-netflix and Eureka

- Eureka Server

spring-cloud-netflix Eureka Server

- Zookeeper Discovery

Service discovery with Zookeeper and spring-cloud-zookeeper-discovery

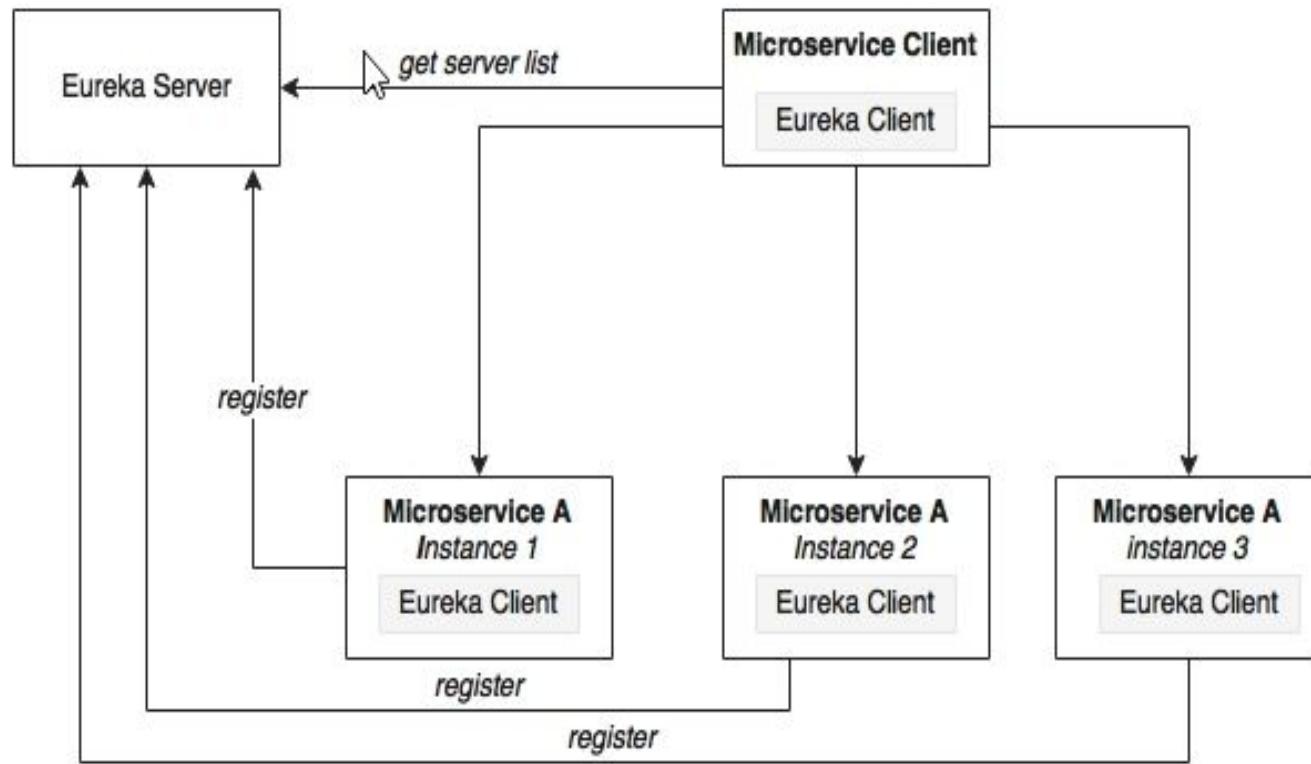
- Cloud Foundry Discovery

Service discovery with Cloud Foundry

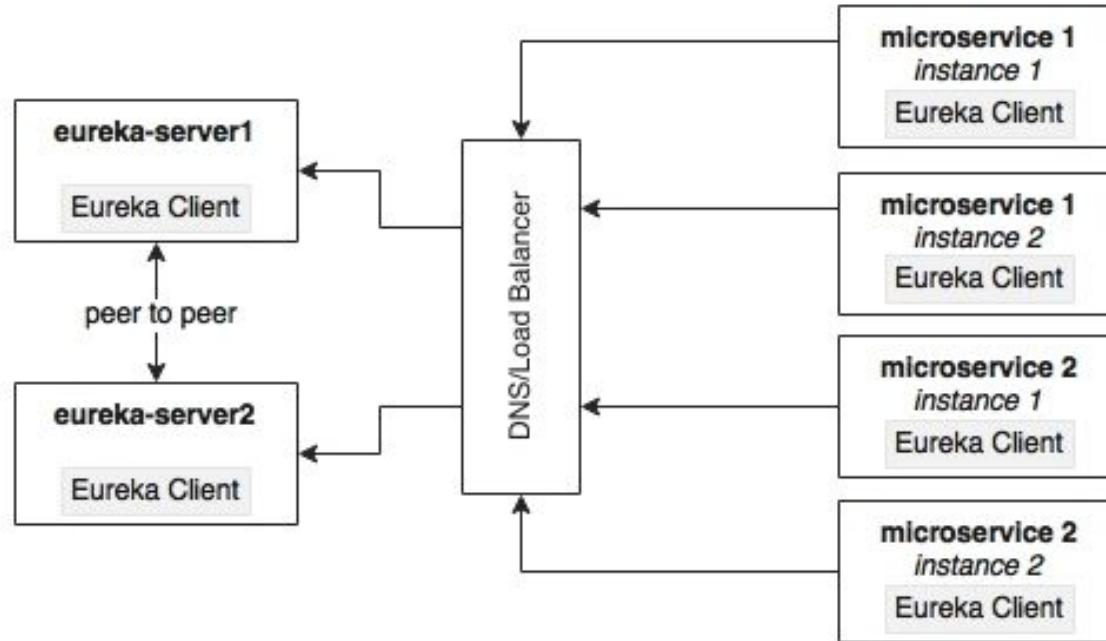
- Consul Discovery

Service discovery with Hashicorp Consul

Understanding Eureka



High availability for Eureka



High availability for Eureka

eureka-server1.properties

```
eureka.client.serviceUrl.defaultZone:http://localhost:8762/eureka/  
eureka.client.registerWithEureka:false  
eureka.client.fetchRegistry:false
```

eureka-server2.properties

```
eureka.client.serviceUrl.defaultZone:http://localhost:8761/eureka/  
eureka.client.registerWithEureka:false  
eureka.client.fetchRegistry:false
```

High availability for Eureka

- `spring.application.name=eureka
spring.cloud.config.uri=http://localhost:8888`
- `java -jar -Dserver.port=8761 -Dspring.profiles.active=server1
demo-0.0.1-SNAPSHOT.jar
java -jar -Dserver.port=8762 -Dspring.profiles.active=server2
demo-0.0.1-SNAPSHOT.jar`

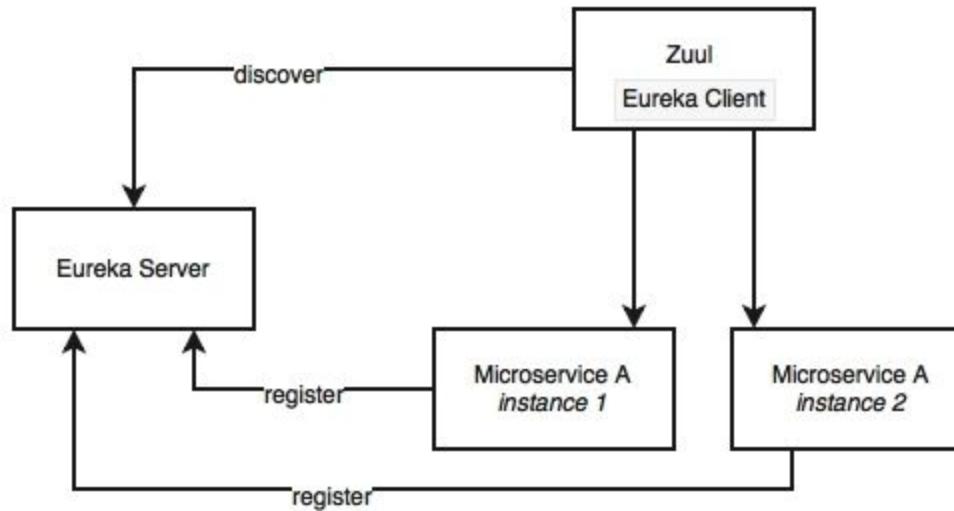
Setting up the Eureka server

Lab 15 -

Zuul Proxy as the API Gateway

- Only a selected set of microservices are required by the clients.
- If there are client-specific policies to be applied, it is easy to apply them in a single place rather than in multiple places. An example of such a scenario is the cross-origin access policy.
- It is hard to implement client-specific transformations at the service endpoint.
- If there is data aggregation required, especially to avoid multiple client calls in a bandwidth-restricted environment, then a gateway is required in the middle.

Zuul proxy as the API Gateway



Zuul

- Enforcing authentication and other security policies at the gateway instead of doing that on every microservice endpoint.
- Business insights and monitoring can be implemented at the gateway level. Collect real-time statistical data, and push it to an external system for analysis.
- API gateways are useful in scenarios where dynamic routing is required based on fine-grained controls.
- Handling the load shredding and throttling requirements is another scenario where API gateways are useful.
- The Zuul gateway is useful for fine-grained load balancing scenarios.
- The Zuul gateway is also useful in scenarios where data aggregation requirements are in place.

Zuul

- ```
public class CustomZuulFilter extends ZuulFilter{
 public Object run() {}
 public boolean shouldFilter() {}
 public int filterOrder() {}
 public String filterType() {}
}

● @Bean
public CustomZuulFilter customFilter() {
 return new CustomZuulFilter();
}
```

# Zuul

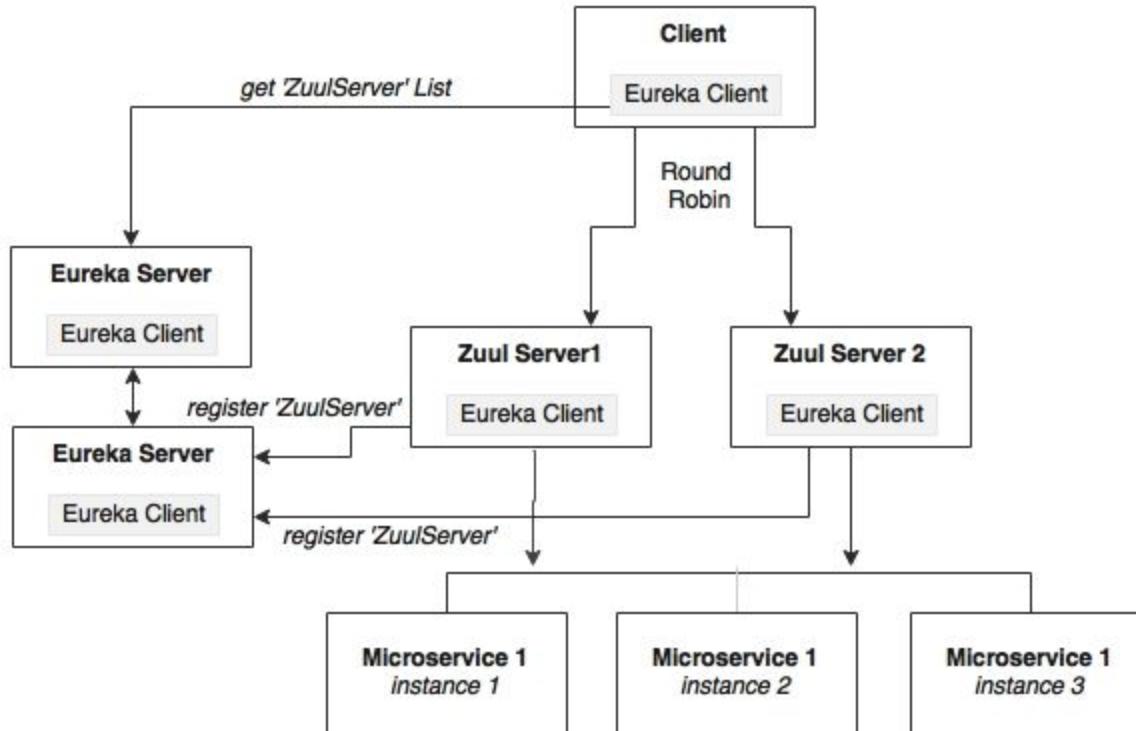
```
@RestController
class SearchAPIGatewayController {

 @RequestMapping("/")
 String greet(HttpServletRequest req) {
 return "<H1>Search Gateway Powered By Zuul </H1>";
 }
}
```

# High Availability of Zuul

- When a client-side JavaScript MVC such as AngularJS accesses Zuul services from a remote browser.
- Another microservice or non-microservice accesses services via Zuul

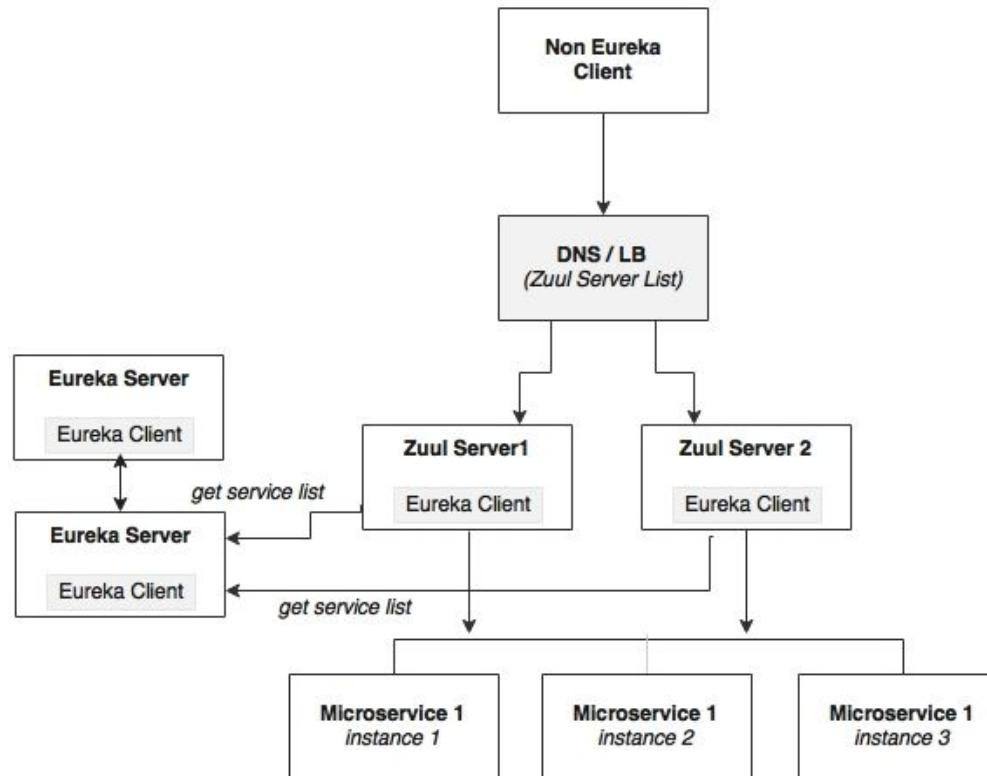
# High availability of Zuul when the client is also a Eureka client



# High Availability of Zuul when the client is also a Eureka client

- Flight[] flights =  
  searchClient.postForObject("http://search-apigateway/api/search/get",  
  searchQuery, Flight[].class);

# High availability when the client is not a Eureka client



# Setting up Zuul

Lab 16 -

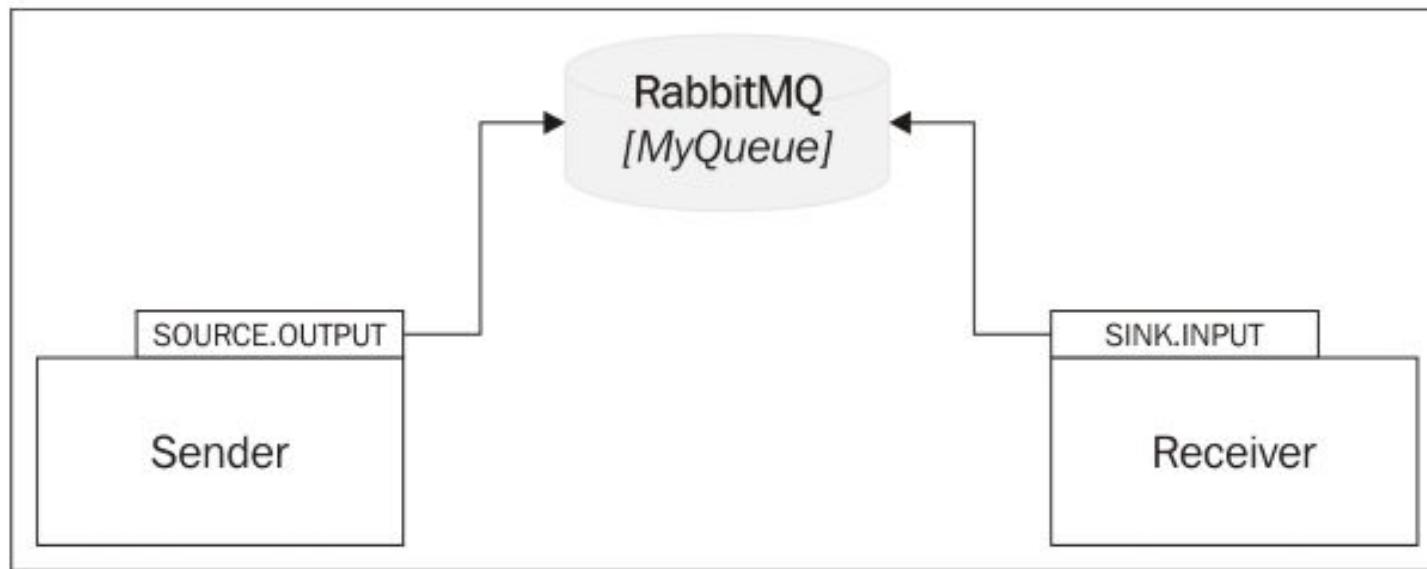
# Completing Zuul for all other services

## Lab 17 -

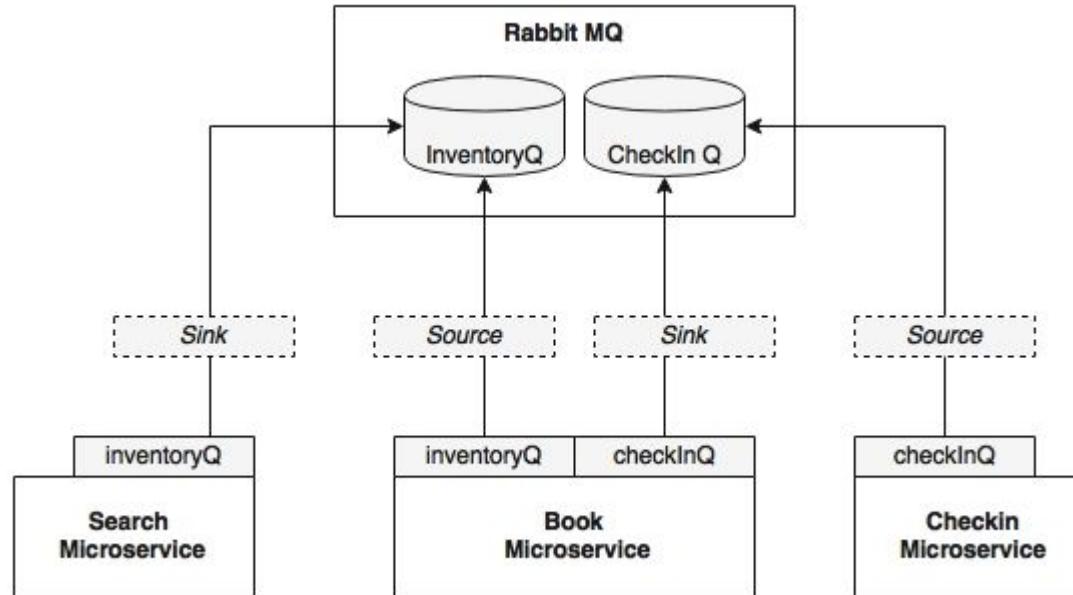
In the end, we will have the following API gateway projects:

- session5.fares-apigateway
- session5.search-apigateway
- session5.checkin-apigateway
- session5.book-apigateway

# Streams for reactive microservices



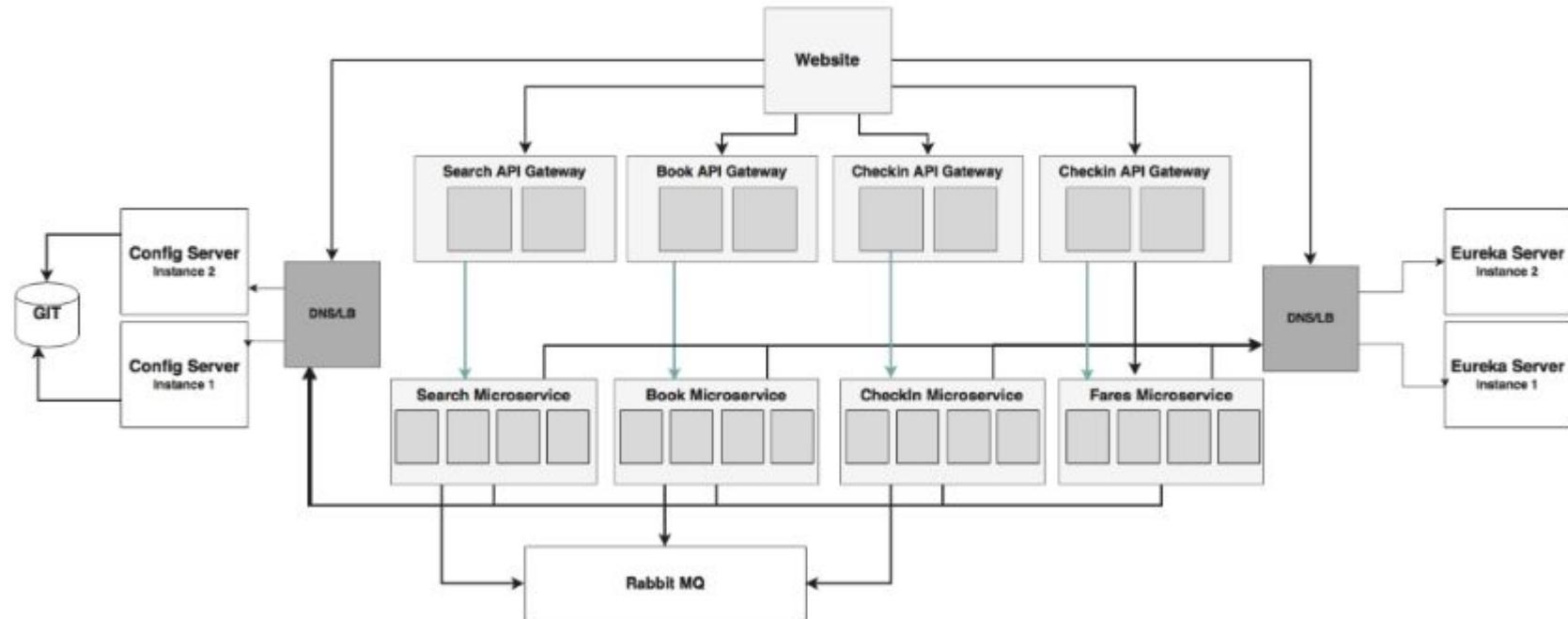
# Streams for reactive microservices



# Streams for Reactive Microservices

Lab 18 -

# Summarizing the Brownfield PSS Architect



| Microservice               | Projects                    | Port         |
|----------------------------|-----------------------------|--------------|
| Book microservice          | chapter5.book               | 8060 to 8064 |
| Check-in microservice      | chapter5.checkin            | 8070 to 8074 |
| Fare microservice          | chapter5.fares              | 8080 to 8084 |
| Search microservice        | chapter5.search             | 8090 to 8094 |
| Website client             | chapter5.website            | 8001         |
| Spring Cloud Config server | chapter5.configserver       | 8888 / 8889  |
| Spring Cloud Eureka server | chapter5.eurekaserver       | 8761 / 8762  |
| Book API gateway           | chapter5.book-apigateway    | 8095 to 8099 |
| Check-in API gateway       | chapter5.checkin-apigateway | 8075 to 8079 |
| Fares API gateway          | chapter5...                 |              |

# Summarizing the BrownField PSS Architecture

Lab 19 -

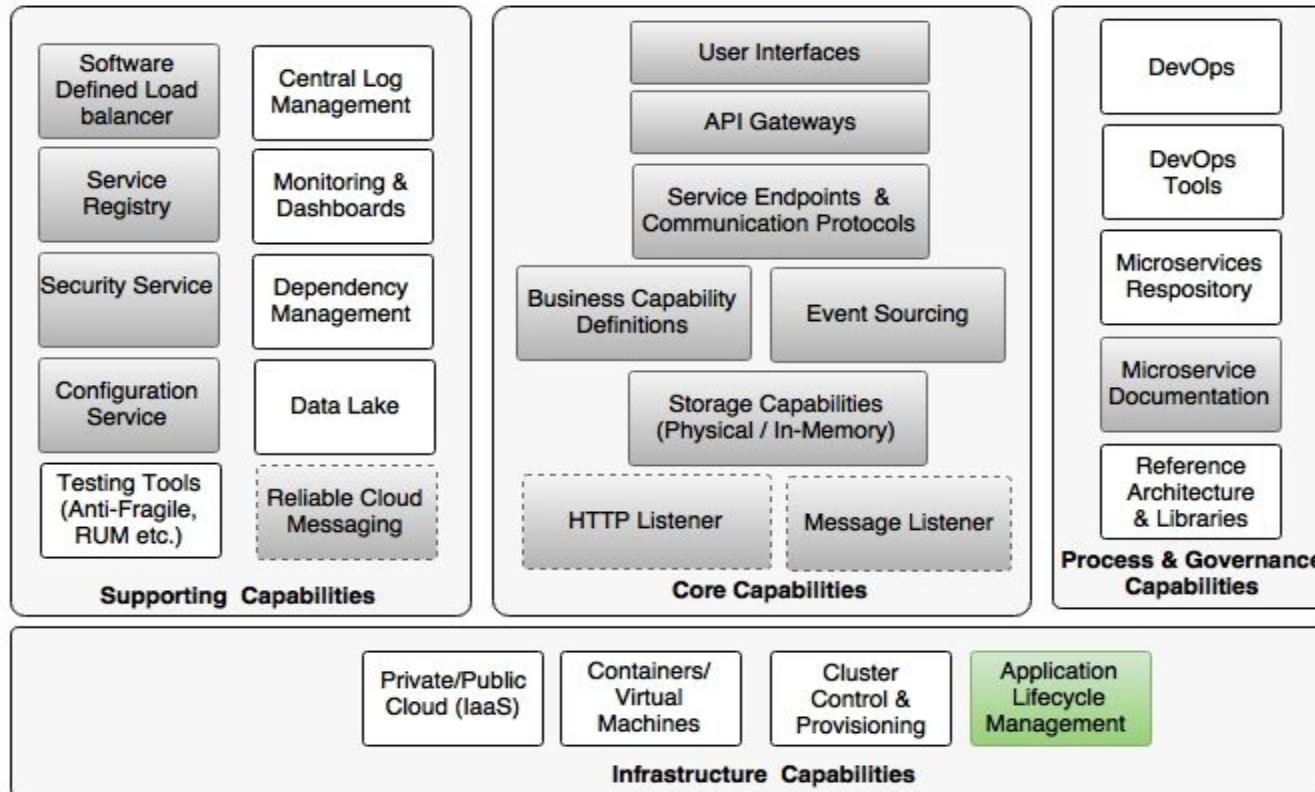
# Summary

- How to scale a Twelve-Factor Spring Boot microservice using the Spring Cloud project.
- Applied to the BrownField Airline's PSS microservice that we developed in the previous session.

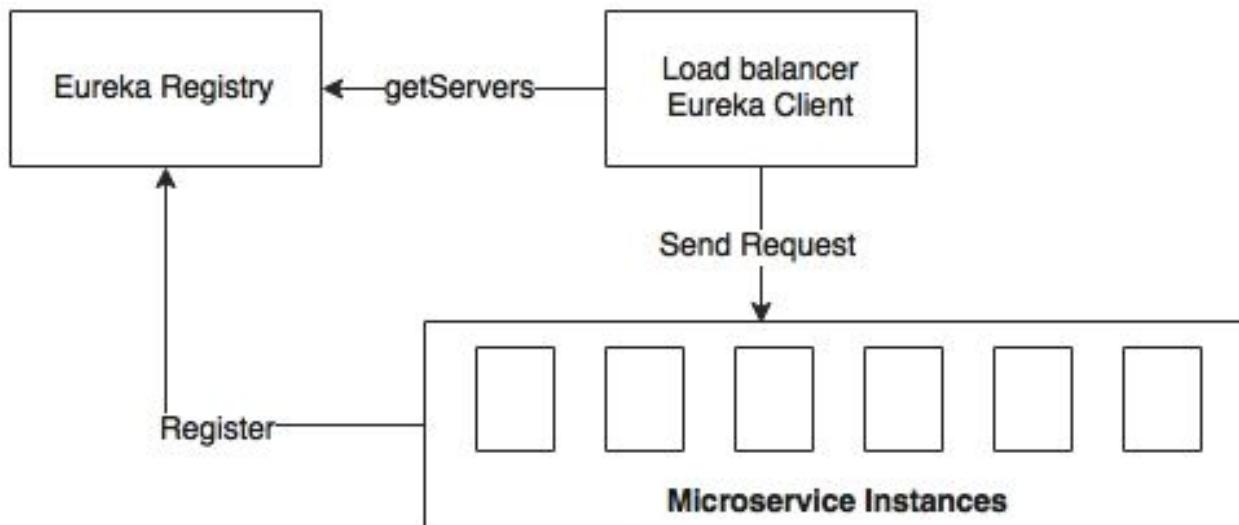
# 6 - Autoscaling Microservices

- The basic concept of autoscaling and different approaches for autoscaling
- The importance and capabilities of a life cycle manager in the context of microservices
- Examining the custom life cycle manager to achieve autoscaling
- Programmatically collecting statistics from the Spring Boot actuator and using it to control and shape incoming traffic

# Reviewing the microservice capability model



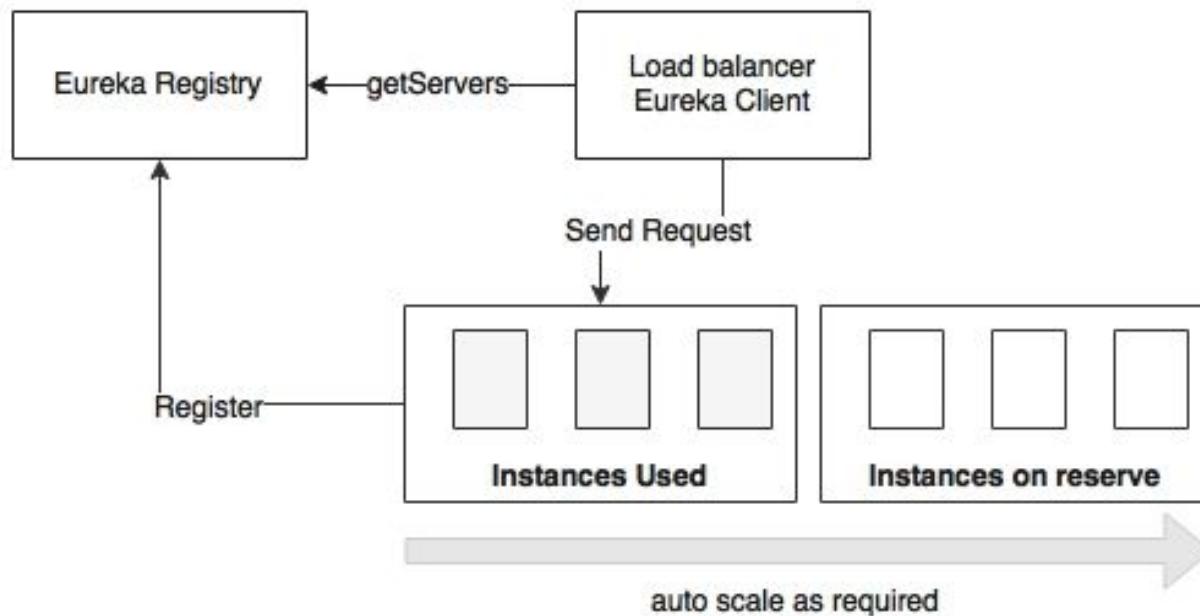
# Scaling microservices with Spring Cloud



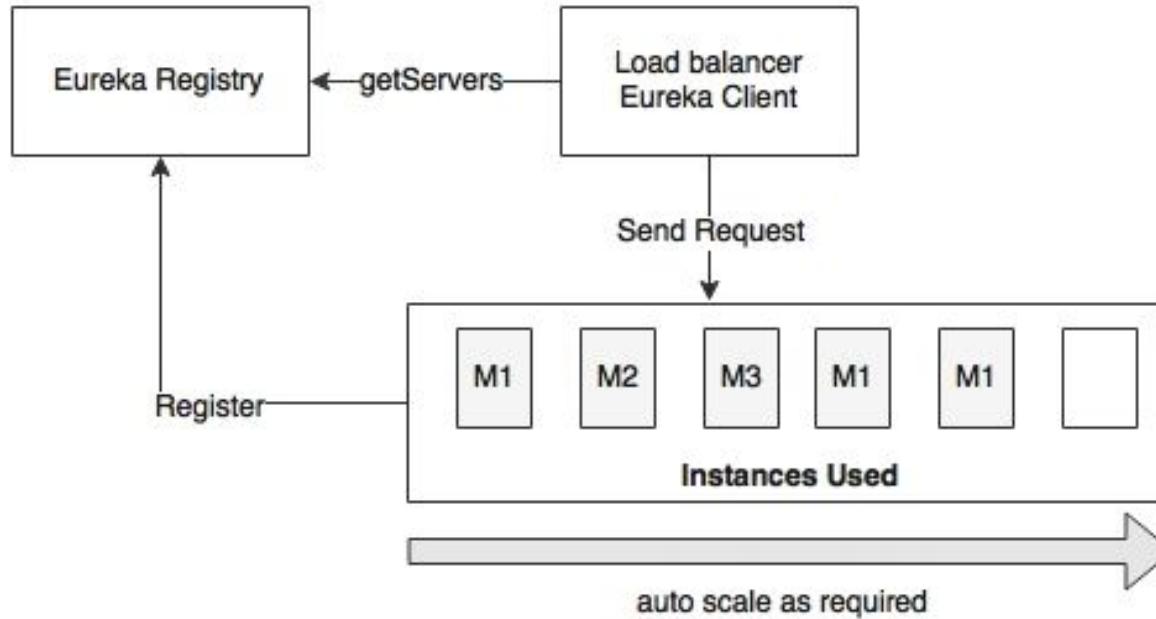
# Scaling microservices with Spring Cloud

- **Eureka** is the central registry component for microservice registration and discovery.
- The **Eureka** client, together with the **Ribbon** client, provide client-side dynamic load balancing.
- The third component is the **microservices** instances developed using Spring Boot with the actuator endpoints enabled.

# Understanding the concept of autoscaling



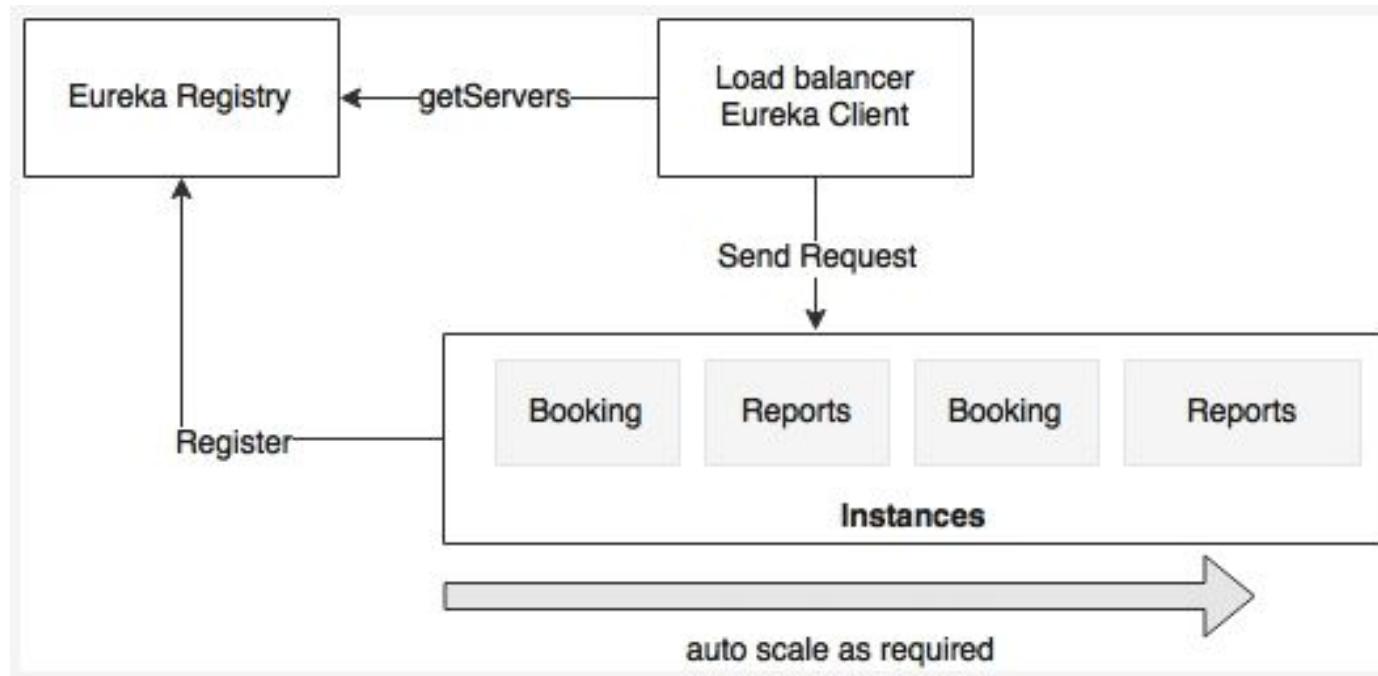
# The benefits of autoscaling



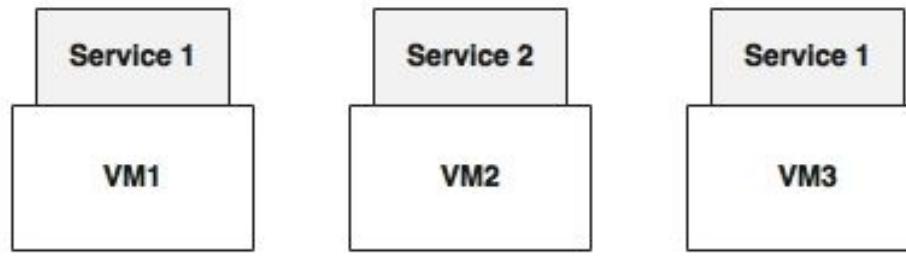
# Key benefits of autoscaling

- It has high availability and is fault tolerant:
- It increases scalability:
- It has optimal usage and is cost saving:
- It gives priority to certain services or group of services:

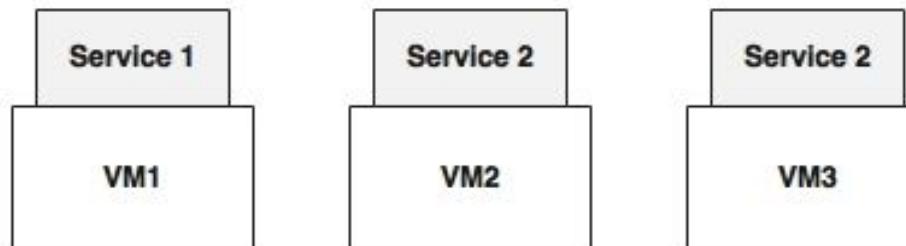
# The benefits of autoscaling



# Different autoscaling models

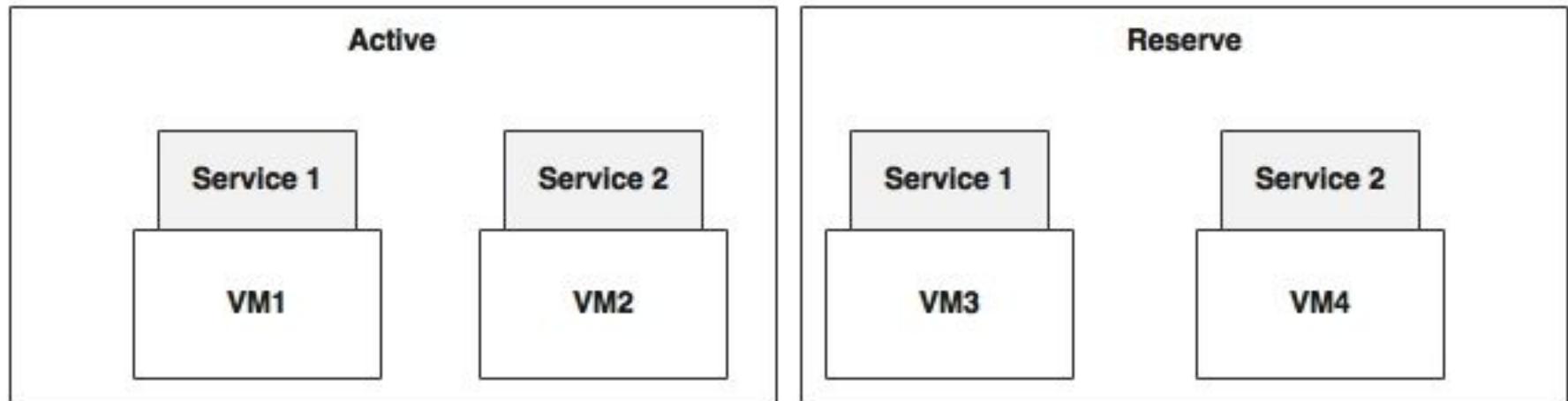


A) VM3 is used for service 1



B) VM3 is used for service 2

# Autoscaling the infrastructure

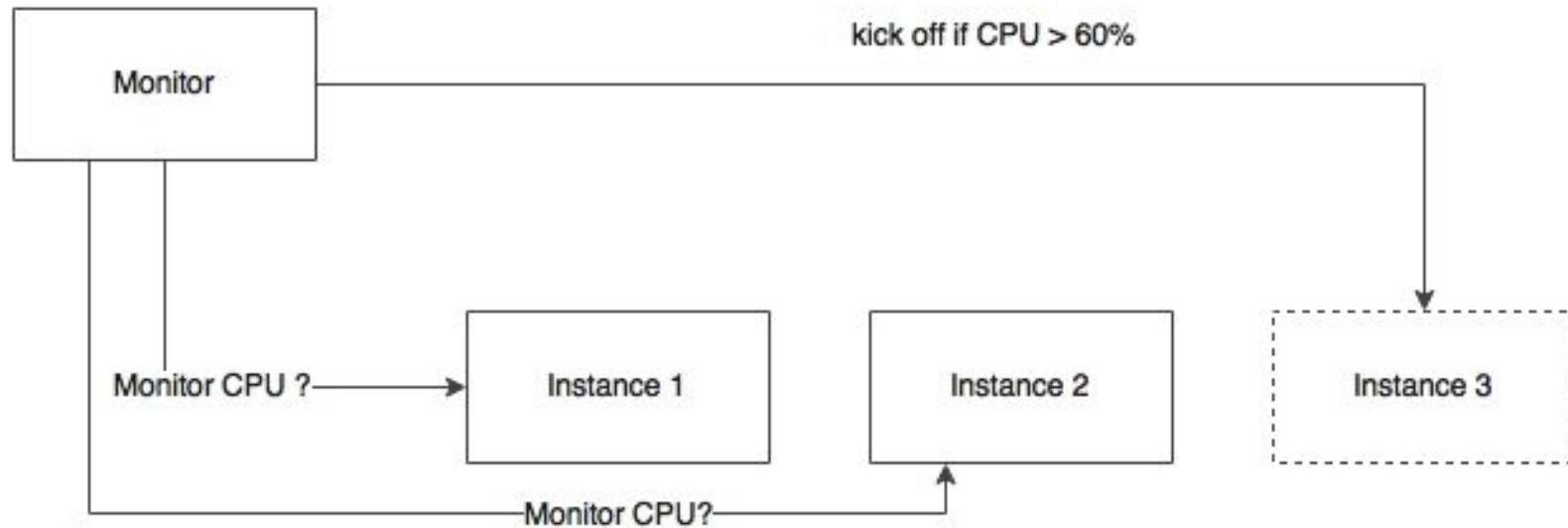


# Autoscaling in the cloud

- Elasticity or autoscaling is one of the fundamental features of most cloud providers.
- Cloud providers use infrastructure scaling patterns, as discussed in the previous section.
- These are typically based on a set of pooled machines.

# Approaches to Autoscaling

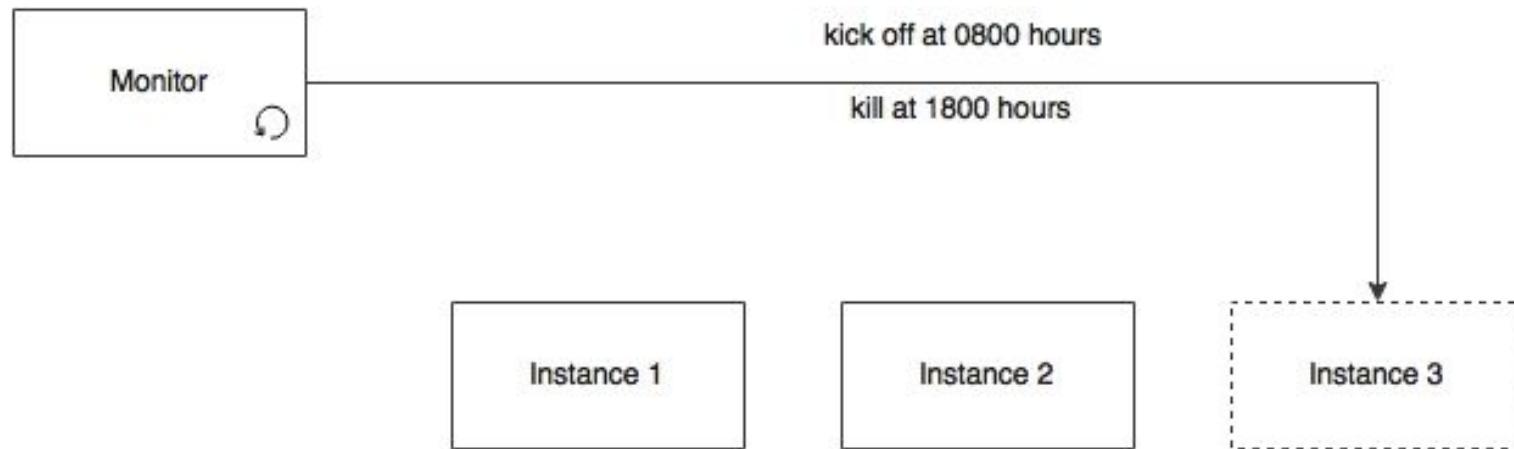
- This approach is based on real-time service metrics collected through monitoring mechanisms.
- Generally, the resource-scaling approach takes decisions based on the CPU, memory, or the disk of machines.
- This can also be done by looking at the statistics collected on the service instances themselves, such as heap memory usage.



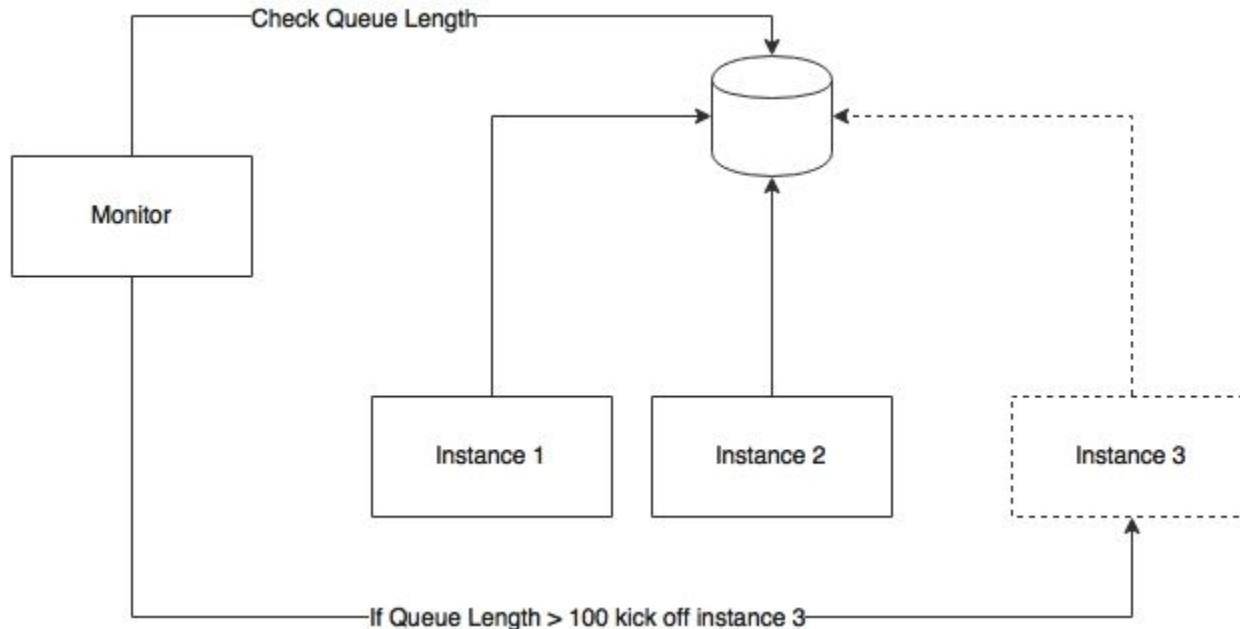
# Approaches to Scaling

- An example of a **response sliding window** is if 60% of the response time of a particular transaction is consistently more than the set threshold value in a 60-second sampling window, increase service instances
- In a **CPU sliding window**, if the CPU utilization is consistently beyond 70% in a 5 minutes sliding window, then a new instance is created
- An example of the **exception sliding window** is if 80% of the transactions in a sliding window of 60 seconds or 10 consecutive executions result in a particular system exception, such as a connection timeout due to exhausting the thread pool, then a new service instance is created

# Scaling during time frames



# Scale based on queue length



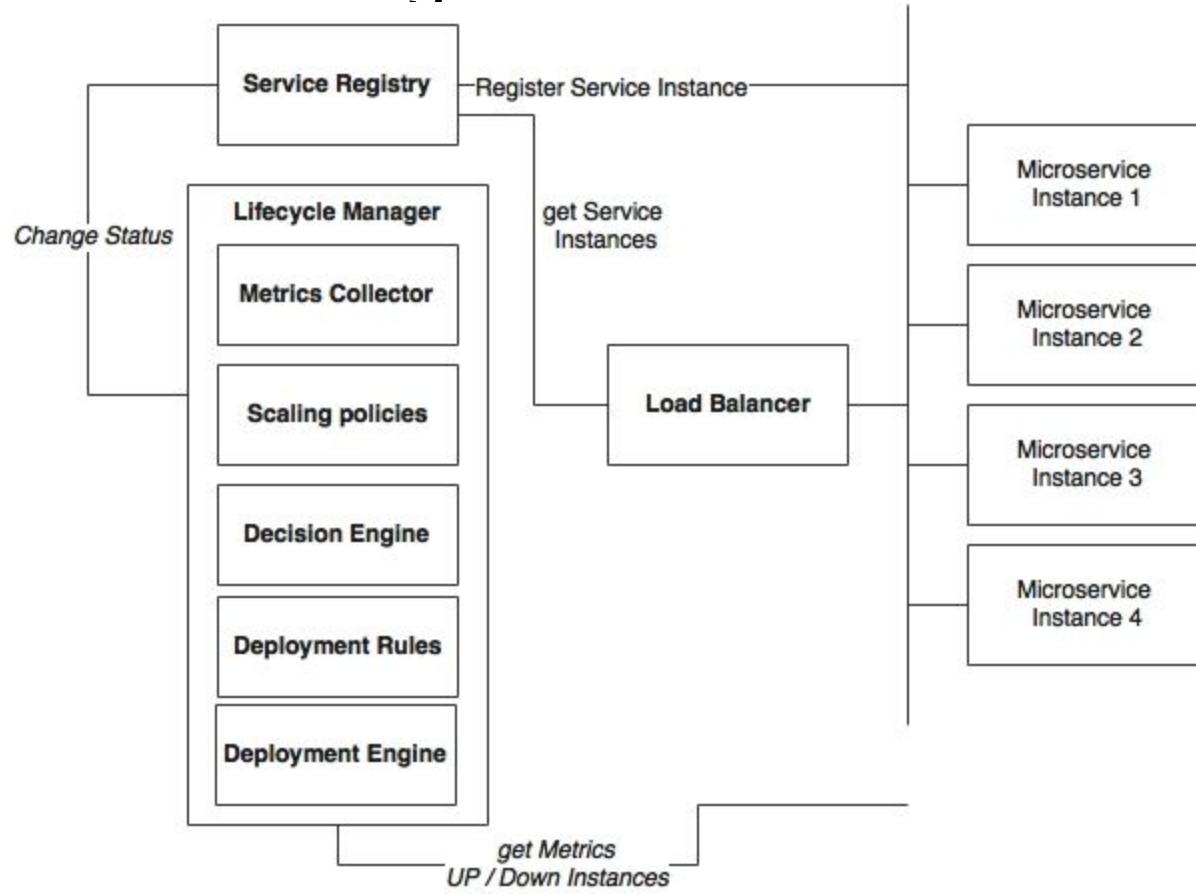
# Scale based on business parameters



# Predictive Autoscaling

- Predictive scaling is a new paradigm of autoscaling that is different from the traditional real-time metrics-based autoscaling.
- A prediction engine will take multiple inputs, such as historical information, current trends, and so on, to predict possible traffic patterns.
- Autoscaling is done based on these predictions.
- Predictive autoscaling helps avoid hardcoded rules and time windows. Instead, the system can automatically predict such time windows.
- In more sophisticated deployments, predictive analysis may use cognitive computing mechanisms to predict autoscaling.

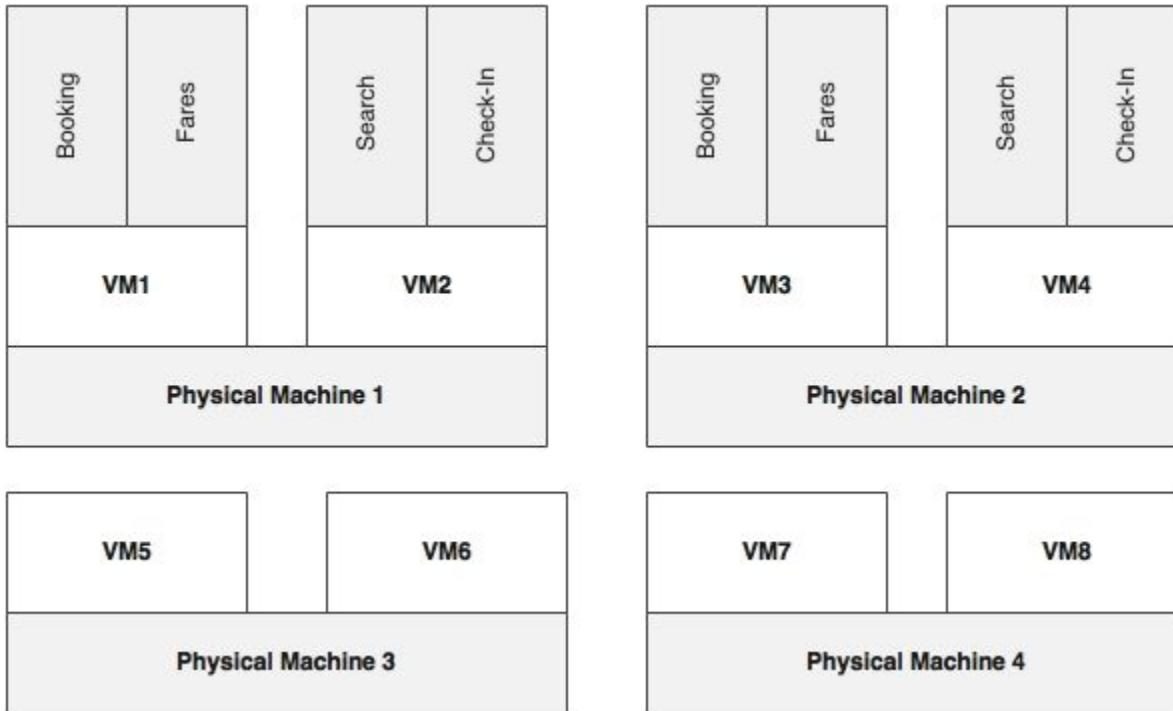
# Capabilities for Scaling

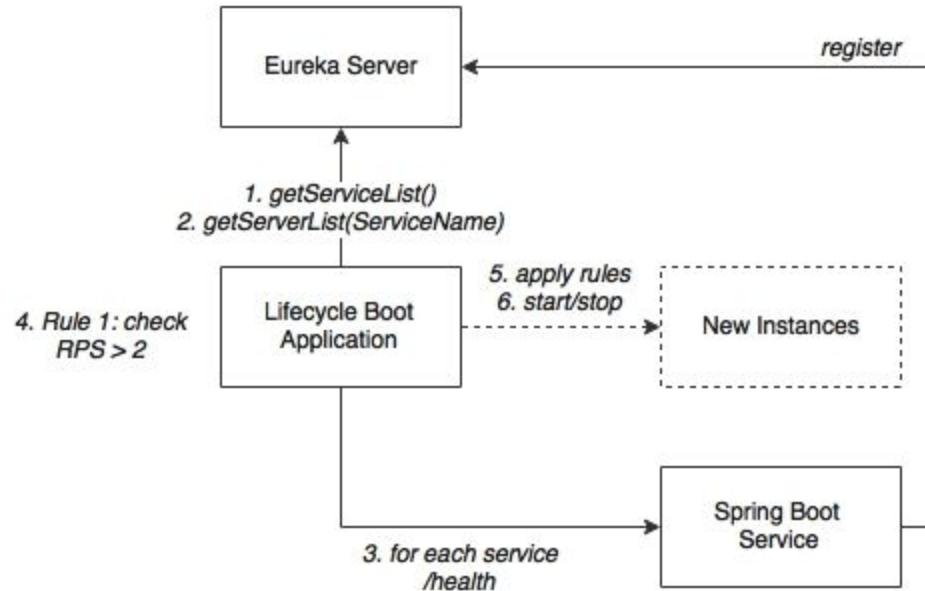


# Components

- **Microservices:** These are sets of the up-and-running microservice instances that keep sending health and metrics information. Alternately, these services expose actuator endpoints for metrics collection. In the preceding diagram, these are represented as **Microservice 1** through **Microservice 4**.
- **Service Registry:** A service registry keeps track of all the services, their health states, their metadata, and their endpoint URI.
- **Load Balancer:** This is a client-side load balancer that looks up the service registry to get up-to-date information about the available service instances.
- **Lifecycle Manager:** The life cycle manager is responsible for autoscaling, which has the following subcomponents:

# Custom Lifecycle Manager

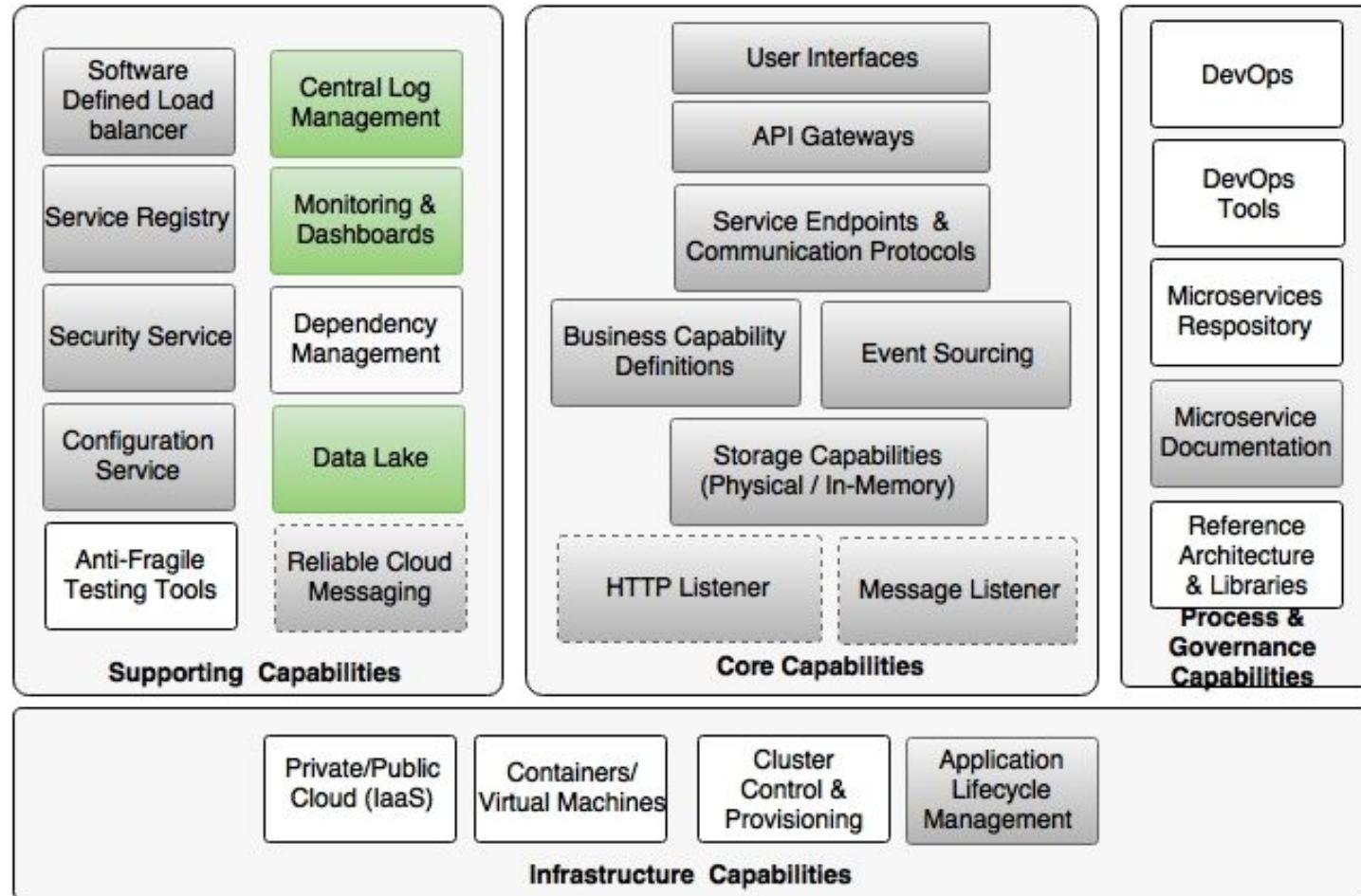


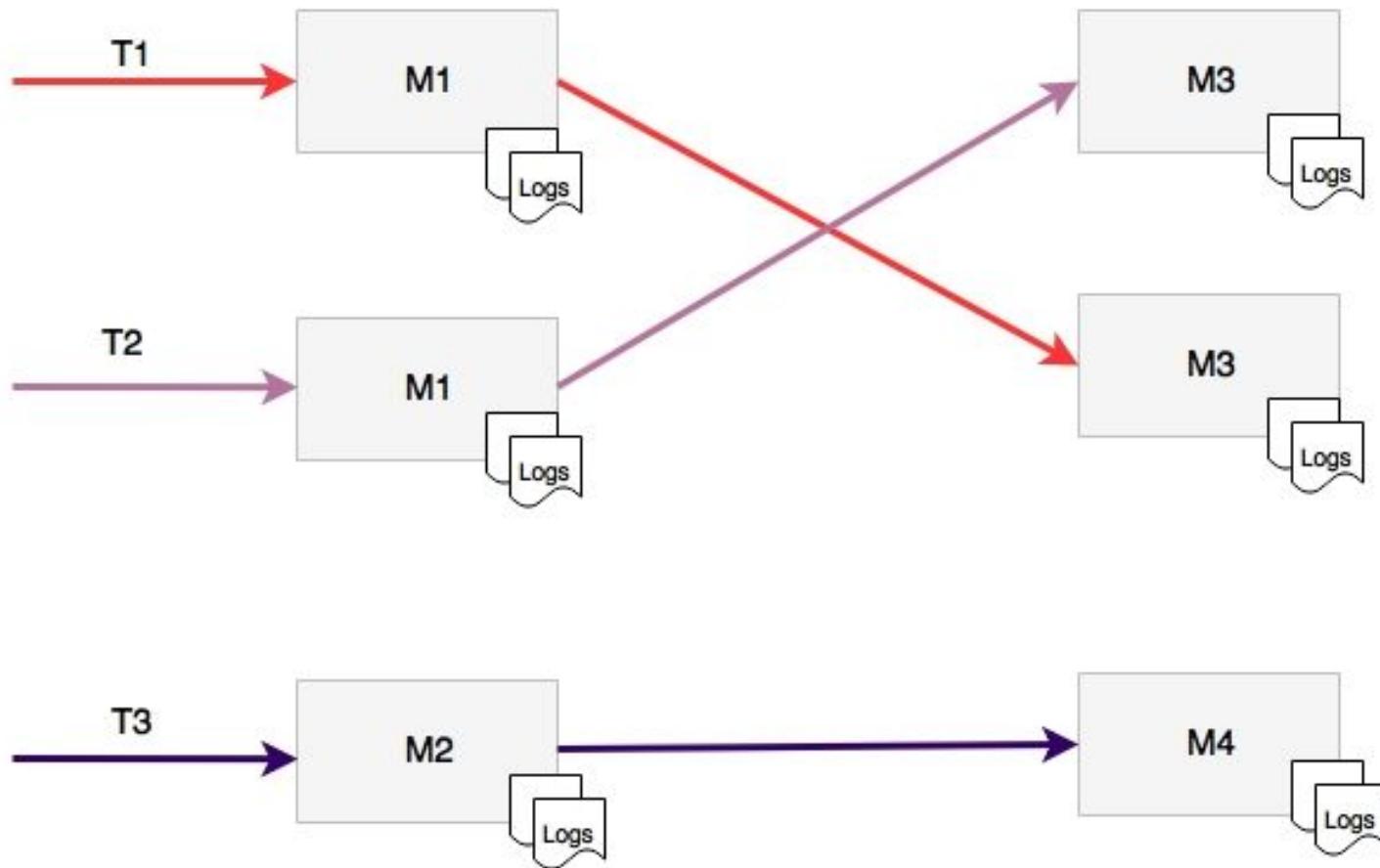


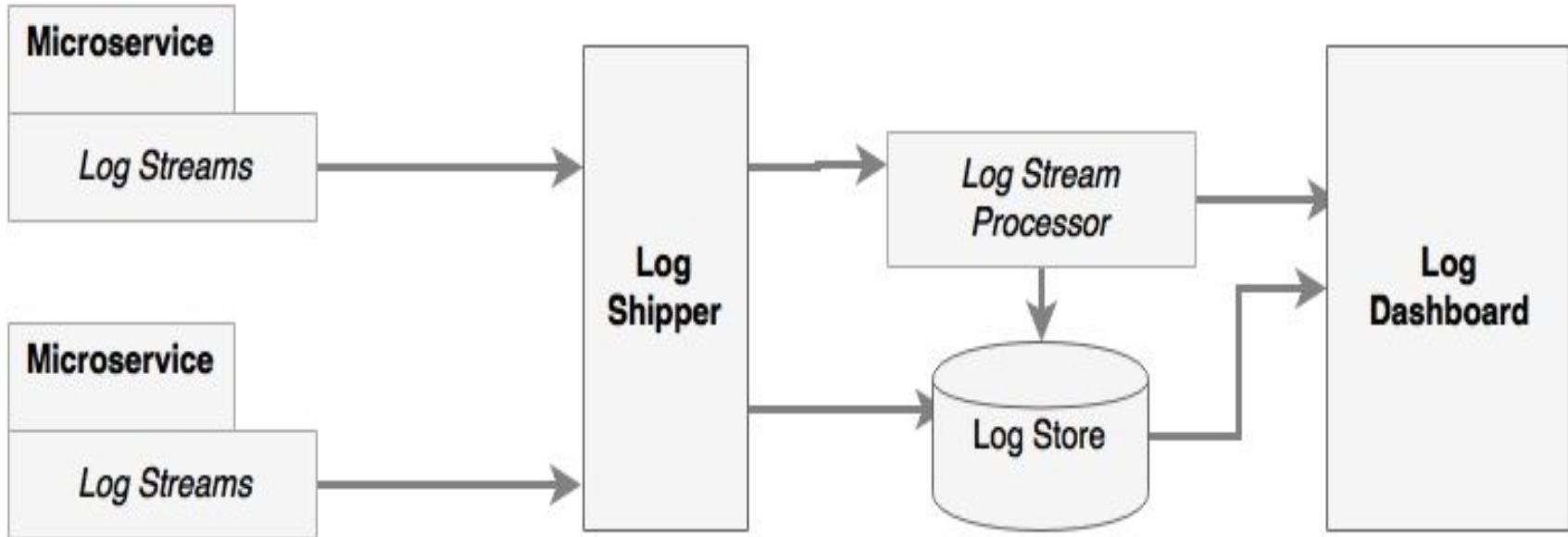
# Lab 20

# 7 - Logging and Monitoring

- The different options, tools, and technologies for log management
- The use of Spring Cloud Sleuth in tracing microservices
- The different tools for end-to-end monitoring of microservices
- The use of Spring Cloud Hystrix and Turbine for circuit monitoring
- The use of data lakes in enabling business data analysis

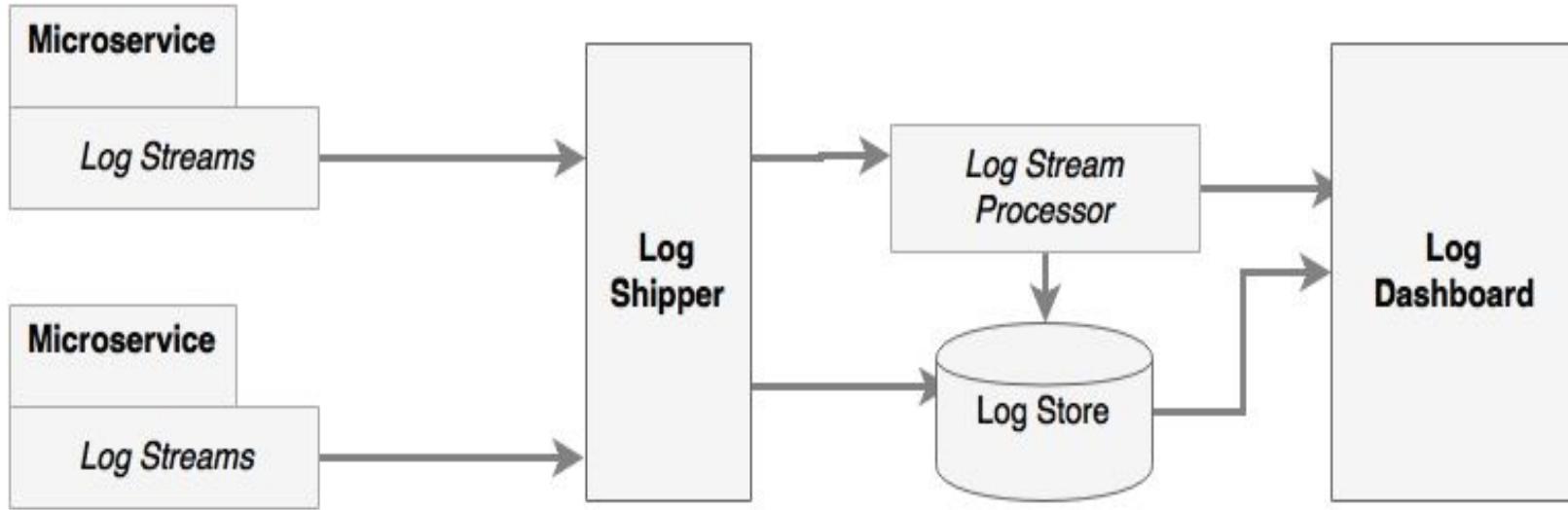


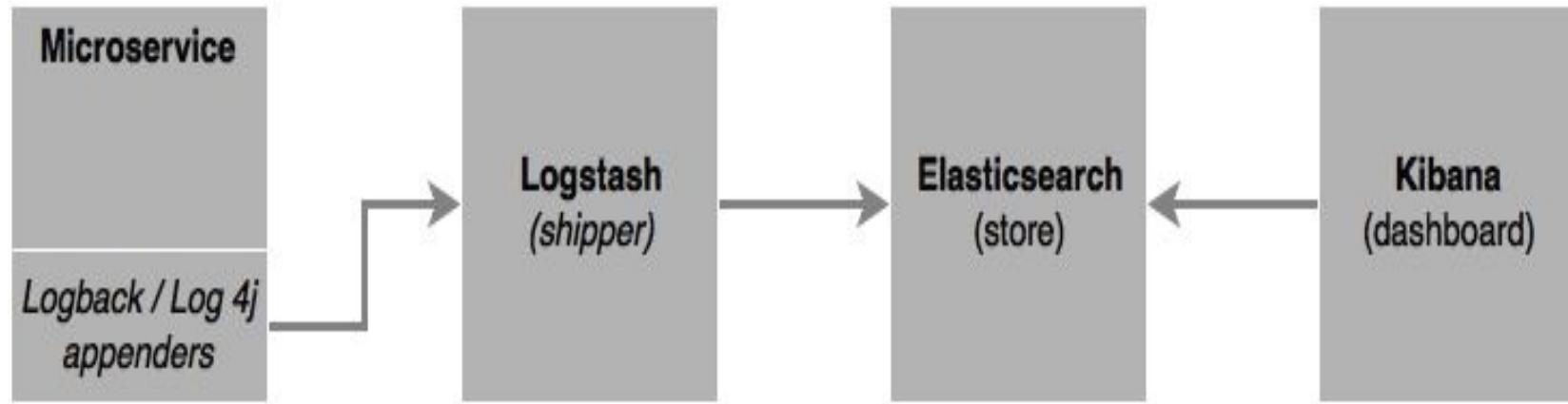




# The Logging Problem

- The ability to collect all log messages and run analytics on top of the log messages
- The ability to correlate and track transactions end to end
- The ability to keep log information for longer time periods for trending and forecasting
- The ability to eliminate dependency on the local disk system
- The ability to aggregate log information coming from multiple sources such as network devices, operating system, microservices, and so on





# Lab 21

Lab 21 - ELK and Kibana :

# Tracing

- With the central logging solution, we can have all the logs in a central storage.  
However, it is still almost impossible to trace end-to-end transactions.
- In order to do end-to-end tracking, transactions spanning microservices need to have a correlation ID.
- Twitter's Zipkin, Cloudera's HTrace, and Google's Dapper systems are examples of distributed tracing systems.
- Spring Cloud provides a wrapper component on top of these using the Spring Cloud Sleuth library.



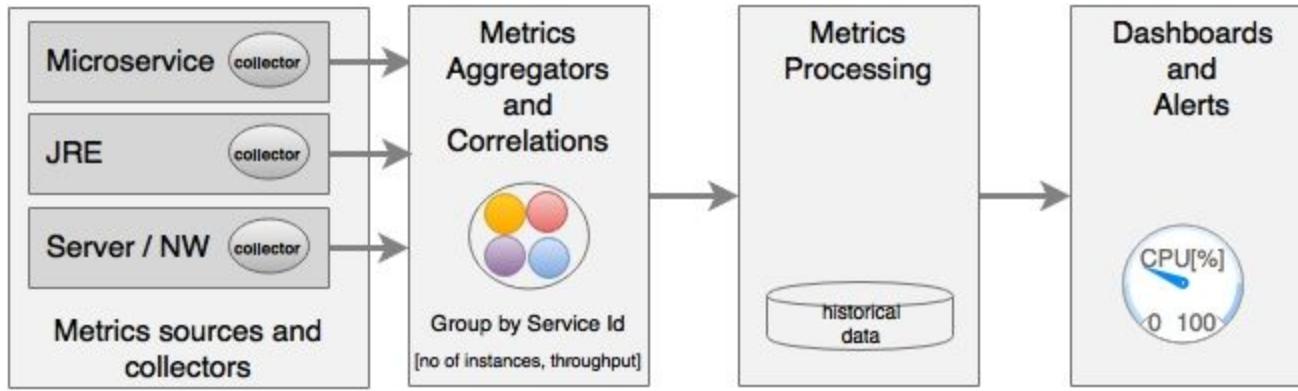
# Lab 22 : Sleuth

# Monitoring

- Microservices are truly distributed systems with a fluid deployment topology.
- Without sophisticated monitoring in place, operations teams may run into trouble managing large-scale microservices.
- Traditional monolithic application deployments are limited to a number of known services, instances, machines, and so on.
- This is easier to manage compared to the large number of microservices instances potentially running across different machines.
- To add more complication, these services dynamically change their topologies. A centralized logging capability only addresses part of the issue.
- It is important for operations teams to understand the runtime deployment topology and also the behavior of the systems.
- This demands more than a centralized logging can offer.

# Why is Monitoring Hard?

- The statistics and metrics are fragmented across many services, instances, and machines.
- Heterogeneous technologies may be used to implement microservices, which makes things even more complex. A single monitoring tool may not give all the required monitoring options.
- Microservices deployment topologies are dynamic, making it impossible to preconfigure servers, instances, and monitoring parameters.
- If the agents require deep integration with the services or operating systems, then this will be hard to manage in a dynamic environment
- If these tools impose overheads when monitoring or instrumenting the application, it may lead to performance issues



# Monitoring Tools

- There are many tools available to monitor microservices.
- There are also overlaps between many of these tools.
- The selection of monitoring tools really depends upon the ecosystem that needs to be monitored. In most cases, more than one tool is required to monitor the overall microservice ecosystem.

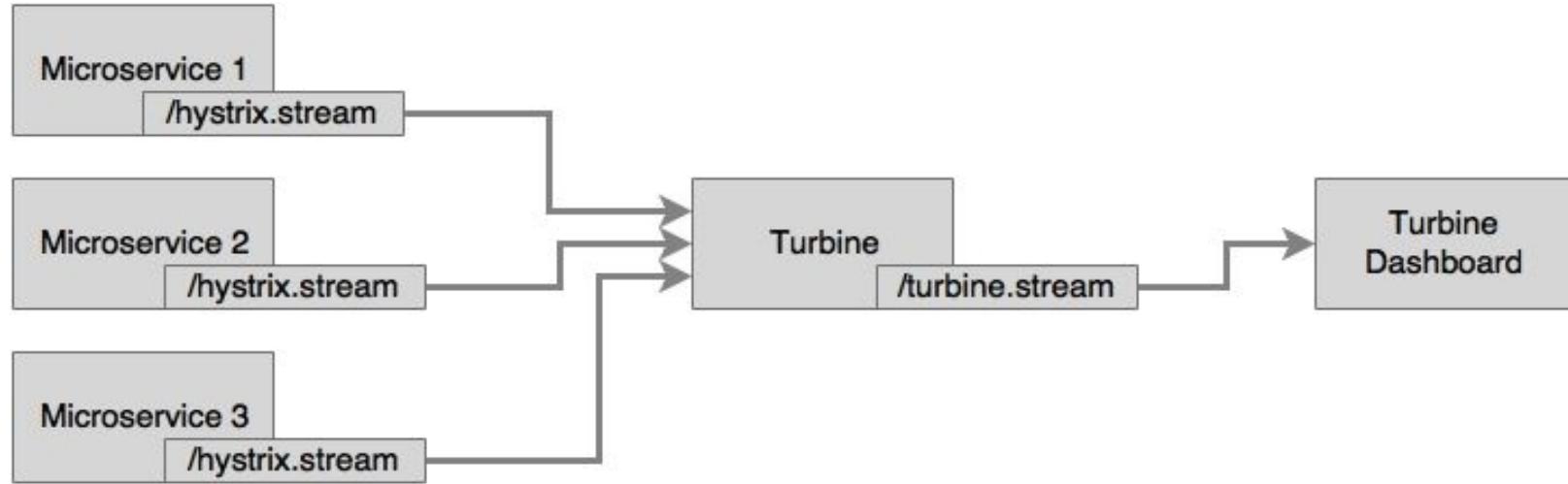
# Monitoring Microservice Dependencies

- Mentoring tools such as AppDynamics, Dynatrace, and New Relic can draw dependencies among microservices. End-to-end transaction monitoring can also trace transaction dependencies. Other monitoring tools, such as Spigo, are also useful for microservices dependency management.
- CMDB tools such as Device42 or purpose-built tools such as Accordance are useful in managing the dependency of microservices. **Veritas Risk Advisor (VRA)** is also useful for infrastructure discovery.
- A custom implementation with a Graph database, such as Neo4j, is also useful. In this case, a microservice has to preconfigure its direct and indirect dependencies. At the startup of the service, it publishes and cross-checks its dependencies with a Neo4j database.

# Lab 23

# Turbine

- In the previous example, the `/hystrix.stream` endpoint of our microservice was given in the Hystrix Dashboard.
- The Hystrix Dashboard can only monitor one microservice at a time.
- If there are many microservices, then the Hystrix Dashboard pointing to the service has to be changed every time we switch the microservices to monitor.
- Looking into one instance at a time is tedious, especially when there are many instances of a microservice or multiple microservices.



# Turbine Lab

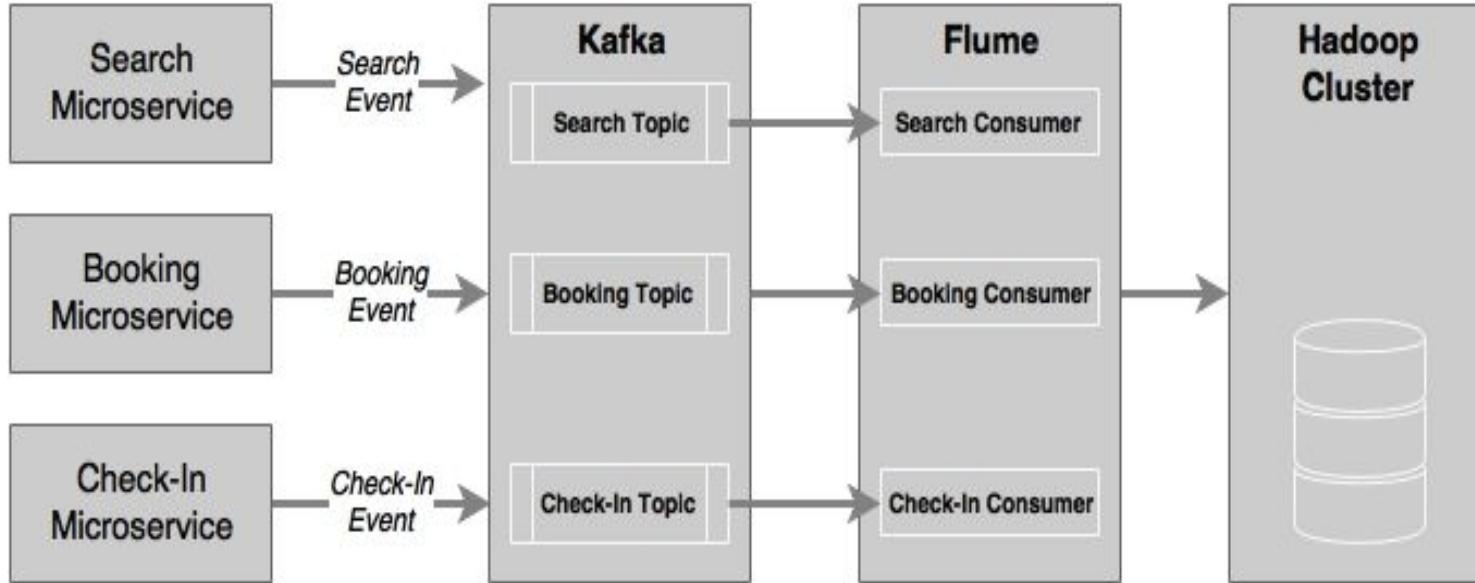
Lab 23 and a Half :

Turbine currently works only with different hostnames. Each instance has to be run on separate hosts. If you are testing multiple services locally on the same host, then update the host file (`/etc/hosts`) to simulate multiple hosts. Once done, `bootstrap.properties` has to be configured as follows:

```
eureka.instance.hostname: localdomain2
```

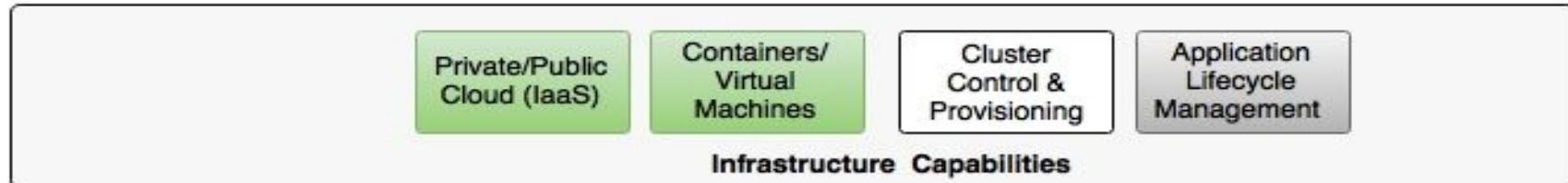
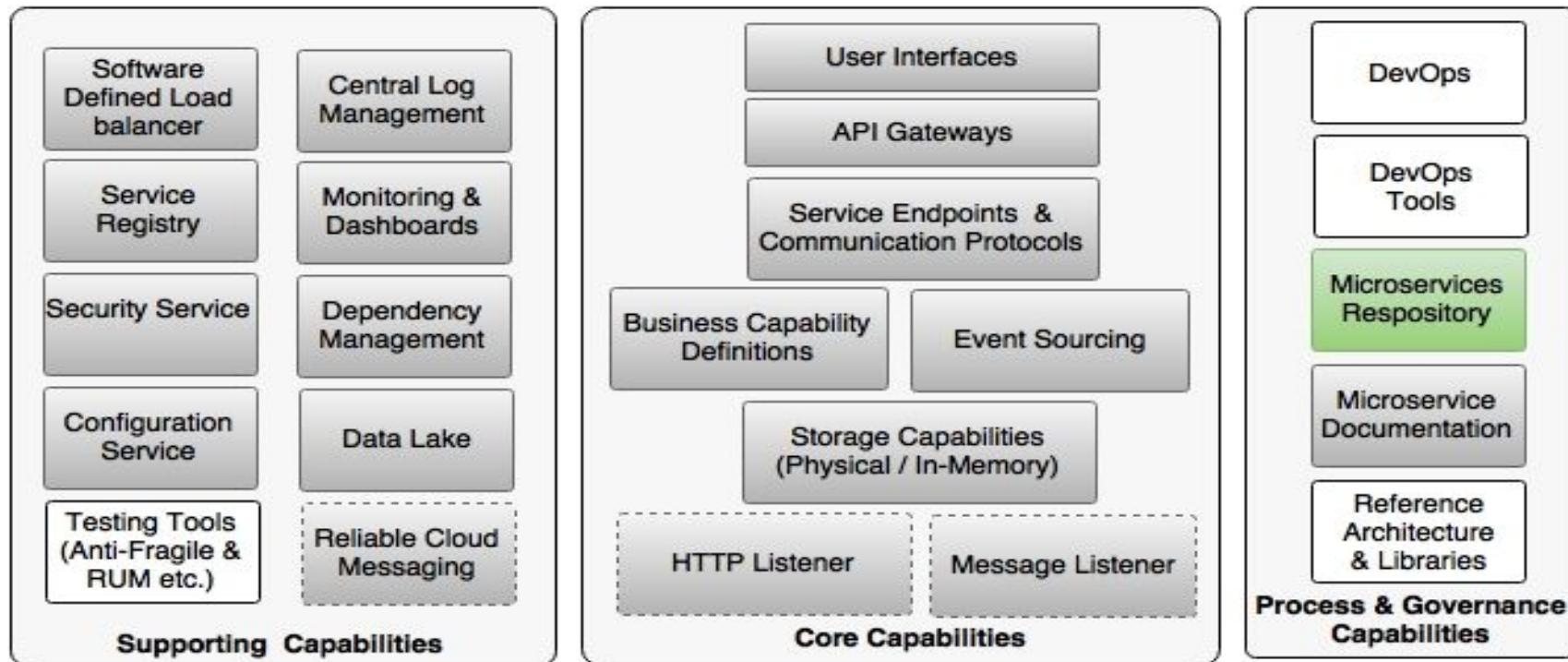
# Big Data and Microservices

- A data lake or data hub is an ideal solution to handling such scenarios.
- An event-sourced architecture pattern is generally used to share the state and state changes as events with an external data store.
- When there is a state change, microservices publish the state change as events. Interested parties may subscribe to these events and process them based on their requirements.
- A central event store may also subscribe to these events and store them in a big data store for further analysis.



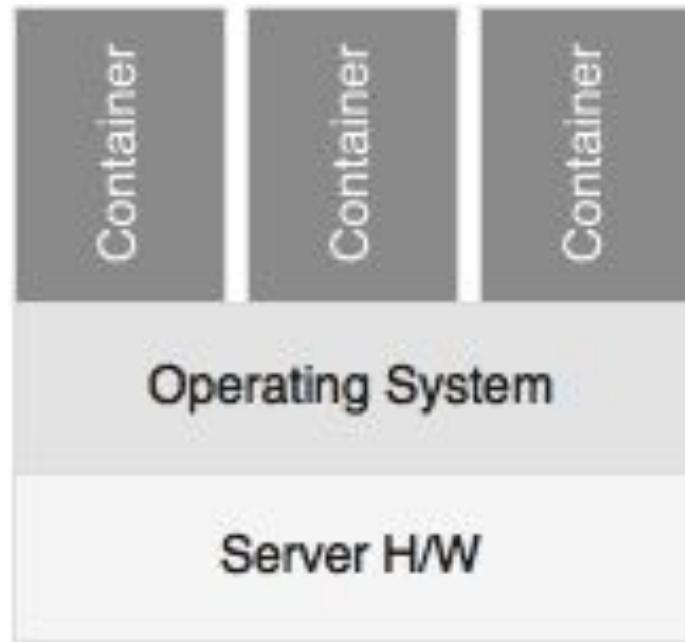
# 8 - Containerizing your microservice

- Containers and virtual machines
- The private/public cloud
- The microservices repository



# Containers

- Containers are not revolutionary, ground-breaking concepts.
- They have been in action for quite a while.
- However, the world is witnessing the re-entry of containers, mainly due to the wide adoption of cloud computing.
- The shortcomings of traditional virtual machines in the cloud computing space also accelerated the use of containers.
- Container providers such as **Docker** simplified container technologies to a great extent, which also enabled a large adoption of container technologies in today's world.
- The recent popularity of DevOps and microservices also acted as a catalyst for the rebirth of container technologies.

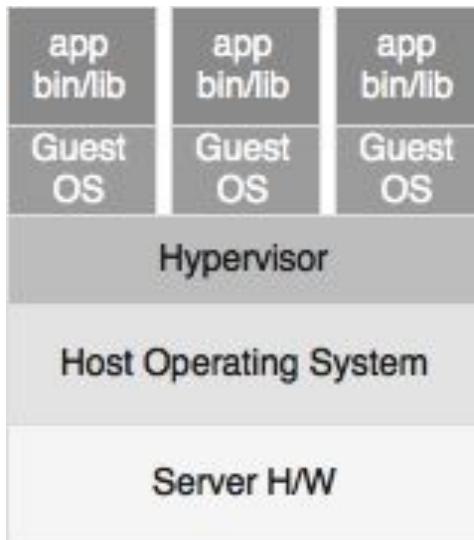


# VM versus Container

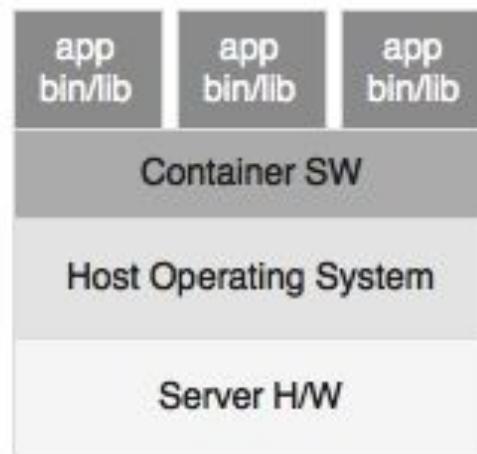
- Both virtualization and containerization exhibit exactly the same characteristics.
- However, in a nutshell, containers and virtual machines are not the same.
- Therefore, it is unfair to make an apple-to-apple comparison between VMs and containers.
- Virtual machines and containers are two different techniques and address different problems of virtualization.

# VM versus Container

- 



A) Virtual Machine Stack



B) Container Stack

# VM versus Container

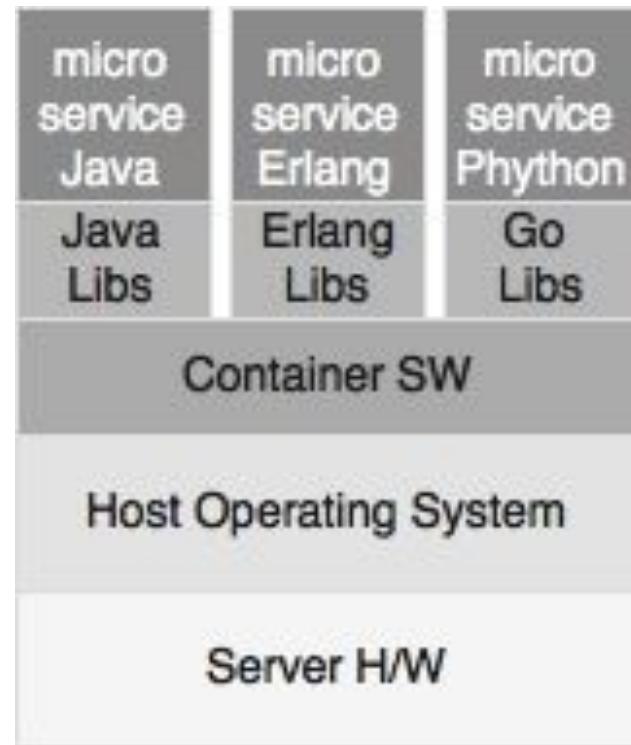
- In the container world, containers do not emulate the entire hardware or operating system.
- Unlike VMs, containers share certain parts of the host kernel and operating system.
- There is no concept of guest OS in the case of containers.
- Containers provide an isolated execution environment directly on top of the host operating system.
- This is its advantage as well as disadvantage.
- The advantage is that it is lighter as well as faster.
- As containers on the same machine share the host operating system, the overall resource utilization of containers is fairly small.
- As a result, many smaller containers can be run on the same machine, as compared to heavyweight VMs.
- As containers on the same host share the host operating system, there are limitations as well.
- For example, it is not possible to set iptables firewall rules inside a container.
- Processes inside the container are completely independent from the processes on different containers running on the same host.

# Container Benefits

- Self contained
- Lightweight
- Scalable
- Portable
- Lower License Cost
- DevOps
- Version Controlled
- Reusable
- Immutable Containers

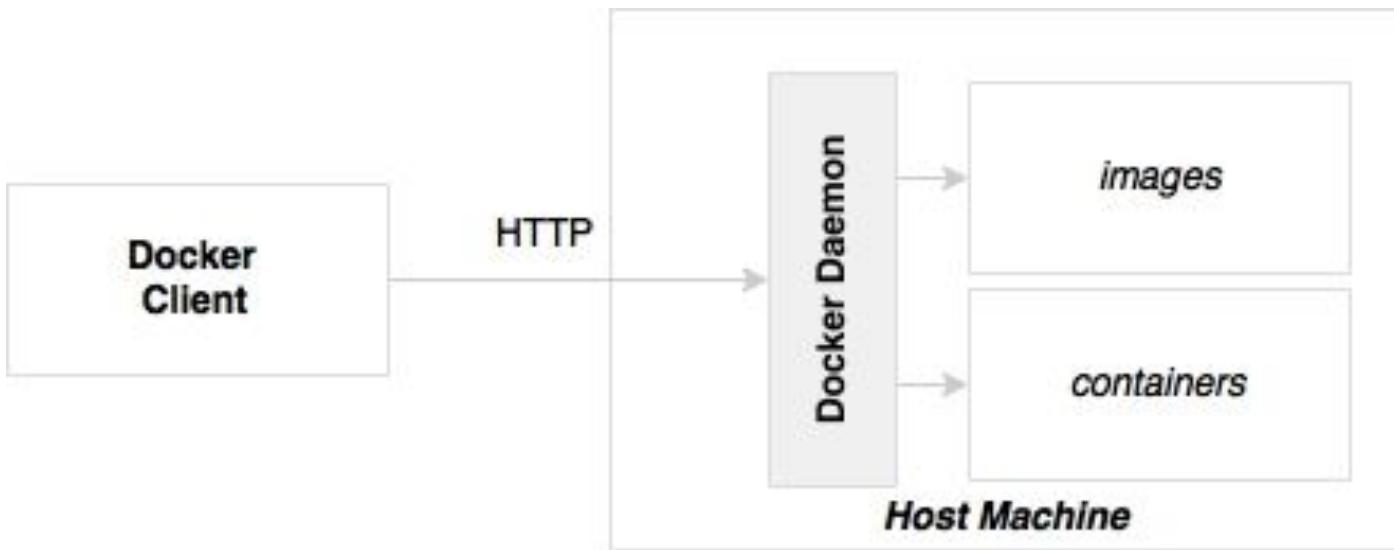
# Microservices and Containers

- There is no direct relationship between microservices and containers.
- Microservices can run without containers, and containers can run monolithic applications.
  - However, there is a sweet spot between microservices and containers.



# Docker

- Docker is a platform to build, ship, and run lightweight containers based on Linux kernels.
- Docker has default support for Linux platforms. It also has support for Mac and Windows using **Boot2Docker**, which runs on top of Virtual Box.
- Amazon **EC2 Container Service (ECS)** has out-of-the-box support for Docker on AWS EC2 instances.
- Docker can be installed on bare metals and also on traditional virtual machines such as VMWare or Hyper-V.



# Docker Daemon

- The Docker daemon is a server-side component that runs on the host machine responsible for building, running, and distributing Docker containers.
- The Docker daemon exposes APIs for the Docker client to interact with the daemon.
- These APIs are primarily REST-based endpoints.
- One can imagine that the Docker daemon as a controller service running on the host machine.
- Developers can programmatically use these APIs to build custom clients as well.

# Docker Client

- The Docker client is a remote command-line program that interacts with the Docker daemon through either a socket or REST APIs.
- The CLI can run on the same host as the daemon is running on or it can run on a completely different host and connect to the daemon remotely.
- Docker users use the CLI to build, ship, and run Docker containers.

# Docker Images

- One of the key concepts of Docker is the image.
- A Docker image is the read-only copy of the operating system libraries, the application, and its libraries.
- Once an image is created, it is guaranteed to run on any Docker platform without alterations.
- In Spring Boot microservices, a Docker image packages operating systems such as Ubuntu, Alpine, JRE, and the Spring Boot fat application JAR file.
- It also includes instructions to run the application and expose the services

Spring Boot  
Application Jar

Java Runtime

Operating System

# More Docker

- Every time we rebuild the application, only the changed layer gets rebuilt, and the remaining layers are kept intact.
- All the intermediate layers are cached, and hence, if there is no change, Docker uses the previously cached layer and builds it on top.
- Multiple containers running on the same machine with the same type of base images would reuse the base image, thus reducing the size of the deployment.
- For instance, in a host, if there are multiple containers running with Ubuntu as the base image, they all reuse the same base image.
- This is applicable when publishing or downloading images as well

Write - FS

Spring Boot

JRE

rootfs  
(Ubuntu / Alpine)

bootfs  
(Linux kernel)

# Docker Containers

- Docker containers are the running instances of a Docker image.
- Containers use the kernel of the host operating system when running.
- Hence, they share the host kernel with other containers running on the same host.
- The Docker runtime ensures that the container processes are allocated with their own isolated process space using kernel features such as **cgroups** and the kernel **namespace** of the operating system.
- In addition to the resource fencing, containers get their own filesystem and network configurations as well.

# Docker Registry

- The Docker registry is a central place where Docker images are published and downloaded from.
- The URL <https://hub.docker.com> is the central registry provided by Docker. The Docker registry has public images that one can download and use as the base registry.
- Docker also has private images that are specific to the accounts created in the Docker registry.

# Docker File

- A Dockerfile is a build or scripting file that contains instructions to build a Docker image.
- There can be multiple steps documented in the Dockerfile, starting from getting a base image.
- A Dockerfile is a text file that is generally named Dockerfile.
- The `docker build` command looks up Dockerfile for instructions to build.
- One can compare a Dockerfile to a `pom.xml` file used in a Maven build.

As our example also uses RabbitMQ, let's explore how to set up RabbitMQ as a Docker container. The following command pulls the RabbitMQ image from Docker Hub and starts RabbitMQ:

```
docker run -net host rabbitmq3
```

Ensure that the URL in `*-service.properties` is changed to the Docker host's IP address. Apply the earlier rule to find out the IP address in the case of Mac or Windows.

# Lab - Containerization

Lab - Containerization

# Docker Registry Lab

# 9 - Deploying your microservice with Kubernetes

- Containers make services portable by allowing you to package code, dependencies, and the runtime environment together in one artifact.
- Deploying containers is generally easier than deploying applications that do not run in containers.
- The host does not need to have any special configuration or state; it just needs to be able to execute the container runtime.
- The ability to deploy one or more containers on a single host gave rise to another challenge when managing production environments—scheduling and orchestrating containers to run on specific hosts and manage scaling.

- Kubernetes was started by Google as an open source project in 2014 and has enjoyed widespread adoption by organizations deploying code in containers.
- Installing and managing a Kubernetes cluster is beyond the scope of this class.
- **Minikube Allows** you to easily run a single-node Kubernetes cluster on your development machine.
  - Even though the cluster only has one node, the way you interface with Kubernetes when deploying your service is generally the same, so the steps here can be followed for any Kubernetes cluster.

# Lab 28 - Kubernetes

# Canary Deployments

- All the techniques discussed in this session help prevent predictable mistakes, but they do nothing to prevent actual software bugs from negatively impacting users of the software we write.
- Canary deployment is a technique for reducing this risk and increasing confidence in new code that is being deployed to production.

- Canary deployments used to be very difficult to get right.
- Teams shipping software in this way usually had to come up with some kind of feature-toggling solution that would gate requests to certain versions of the application being deployed.
- Thankfully, containers have made this much easier, and Kubernetes has made it even more so.

# Lab 29 - Kubernetes and Containers