SPRING BOOT

[Spring Boot, Spring JPA, Spring DATA, Spring MicroServices, Spring Cloud, Spring Messaging]

K.RAMESH

ASPIRE Technologies

#501, 5th Floor, Mahindra Residency, Maithrivanam Road, Ameerpet, Hyderabad

Ph: 07799 10 8899, 07799 20 8899

E-Mail:ramesh@java2aspire.com website: www.java2aspire.com

1.SPRING BOOT

Without spring boot the problems are:

1) We need to hunt for all the **compatible libraries** for the specific Spring version and add them.
2) Most of the times we configure the **DataSource**, **JdbcTemplate**, **TransactionManager, DispatcherServlet, HandlerMapping, ViewResolver**, etc beans in the same way.
3) We should always deploy in external container.
4) The problem with Spring component-scanning and autowiring is that it's hard to see how all of the components in an application are wired together.

With Spring Boot the advantages are:

1) **Spring Boot Starters** help easy dependency management.
2) **Auto configuration** for most of the commonly used beans such as DataSource, JdbcTemplate, TransactionTemplate, DispatcherServlet, ViewResolver, HandlerMappig, etc using customizable properties. We need to enable auto configuration by adding either @EnableAutoConfiguraiton or @SpringBootApplication.
3) **Embedded Servlet Container**
   Spring-boot-starter-web pulls the spring-boot-starter-tomcat automatically which starts tomcat as a embedded container. So we don't have to deploy our application on any externally installed tomcat server.
4) **Spring Boot Actuators**
   The actuators lets us look inside of our running Spring Boot application. The actuators provide details such as which beans have been configured, autoconfig details, bean dependencies, environment variables, configuration properties, memory usage, garbage collection, web requests, and data source usage.

Note: a) Spring boot increases the speed of development because of Starters and autoconfiguration.
b) One of the great outcomes of Spring Boot is that it almost eliminates the need to have traditional XML configurations.

Spring Boot newly added in Spring 4 and offers following main features:

Spring Boot Starters

Without Spring Boot, we need to configure all the dependencies required in our pom.xml.
Spring Boot starters aggregate common groupings of dependencies into single dependency that can be added to a project's Maven or Gradle build. For example, if web is selected, **spring-boot-starter-web** adds all dependencies required for a Spring MVC project.

#pom.xml

```
<project>
<dependencies>
```

```
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-jdbc</artifactId>
                <version>1.5.3.RELEASE</version>
        </dependency>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-test</artifactId>
                <version>1.5.3.RELEASE</version>
        </dependency>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-web</artifactId>
                <version>1.5.3.RELEASE</version>
        </dependency>
    </dependencies>
</project>
```

Note: The list of starters found in <artifactId>**spring-boot-starters**</artifactId>

```
        <modules>
                <module>spring-boot-starter</module>
                <module>spring-boot-starter-activemq</module>
                <module>spring-boot-starter-amqp</module>
                <module>spring-boot-starter-aop</module>
                <module>spring-boot-starter-jdbc</module>
                <module>spring-boot-starter-test</module>
                <module>spring-boot-starter-data-jpa</module>
                <module>spring-boot-starter-data-web</module>
                …
        </modules>
```

The parent element is one of the interesting aspects in the `pom.xml` file.

```
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>1.5.3.RELEASE</version>
        </parent>
```

The advantage of using the spring-boot-starter-parent POM file is that developers need not worry about finding the right compatible versions of different libraries such as Spring, Jersey, JUnit, Hibernate, Jackson, and so on. The jar versions are defined in <artifactId>**spring-boot-dependencies**</artifactId> .

A snapshot of some of the properties are as shown as follows:

```
        <properties>
            <!-- Dependency versions -->
            <activemq.version>5.14.5</activemq.version>
            <antlr2.version>2.7.7</antlr2.version>
            <appengine-sdk.version>1.9.51</appengine-sdk.version>
            <tomcat.version>8.5.14</tomcat.version>
            <hikaricp.version>2.5.1</hikaricp.version>
            <commons-dbcp.version>1.4</commons-dbcp.version>
            <commons-dbcp2.version>2.1.1</commons-dbcp2.version>
            <hibernate.version>5.0.12.Final</hibernate.version>
            …
```

Below is the modified pom file after adding parent starter:

#pom.xml

```
    <project>
        <parent>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-parent</artifactId>
            <version>1.5.3.RELEASE</version>
        </parent>
        <dependencies>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-jdbc</artifactId>
            </dependency>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-test</artifactId>
            </dependency>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-web</artifactId>
            </dependency>
        </dependencies>
    </project>
```

Auto configuration

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` classes.

Spring Boot provides '**spring-boot-autoconfigure**' module (**spring-boot-autoconfigure-<version>.jar**) which contains many configuration classes to autoconfigure beans. The above jar file contains **META-INF/spring.factories** file which contains list of autoconfigure classes.

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration,\
org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration,\
org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration,\
JpaBaseConfiguration#transactionManager,\
JpaBaseConfiguration#jpaVendorAdapter
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
…
```

Since spring boot provides many autoconfigure classes hence reduces the complexity of configuration.
Spring Boot auto-configuration is a runtime (more accurately, application startup-time) process that considers several factors to decide what Spring configuration should and should not be applied. For example, Is Spring's JdbcTemplate available on the classpath? If so and if there is a DataSource bean, then auto-configure a JdbcTemplate bean.

Example:

```
@Configuration
@ConditionalOnClass({ DataSource.class, JdbcTemplate.class })
public class JdbcTemplateAutoConfiguration {

}
```

The **@EnableAutoConfiguration** enables the magic of auto configuration.

Spring-Boot checks tomcat-jdbc (default), HikariCP, Commons DBCP and Common DBCP2 in this sequence order i.e., Spring Boot checks the availability of the following data source classes and uses the first one that is available in classpath.

1. org.apache.tomcat.jdbc.pool.DataSource
2. com.zaxxer.hikari.HikariDataSource

3. org.apache.commons.dbcp.BasicDataSource
4. org.apache.commons.dbcp2.BasicDataSource

SpringBoot by default pulls in tomcat-jdbc-{version}.jar if we add spring-boot-starter-jdbc dependency.

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

We need to exclude tomcat-jdbc from classpath If we want to use other datasources.

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jdbc</artifactId>
        <exclusions>
                <exclusion>
                        <groupId>org.apache.tomcat</groupId>
                        <artifactId>tomcat-jdbc</artifactId>
                </exclusion>
        </exclusions>
</dependency>
```

We need to add following dependency to use HikariDataSource.

```
<dependency>
        <groupId>com.zaxxer</groupId>
        <artifactId>HikariCP</artifactId>
</dependency>
```

We need to add following dependency to use commons-dbcp.

```
<dependency>
        <groupId>commons-dbcp</groupId>
        <artifactId>commons-dbcp</artifactId>
</dependency>
```

We need to add following dependency to use commons-dbcp2.

```
<dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-dbcp2</artifactId>
</dependency>
```

Spring Boot automatically configures above data sources based on corresponding jar file present in classpath and connection properties should be configured in application.properties file. Various properties can be specified inside our **application.properties/application.yml** file. This section provides a list of common Spring Boot properties and references to the underlying classes that consume them.

```
# src/main/resources/application.properties
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=system
spring.datasource.password=manager

#tomcat-connection settings
#spring.datasource.tomcat.initialSize=20
#spring.datasource.tomcat.max-active=25

# Hikari settings
#spring.datasource.hikari.maximum-pool-size=20

# dbcp settings
#spring.datasource.dbcp.initial-size=20
#spring.datasource.dbcp.max-active=25

# dbcp2 settings
spring.datasource.dbcp2.initial-size=20
spring.datasource.dbcp2.max-total=25
```

If we want to use other than above datasources then we need to configure explicitly.

```
<dependency>
        <groupId>c3p0</groupId>
        <artifactId>c3p0</artifactId>
        <version>0.9.1.2</version>
</dependency>
```

At any point we can start to define our own configuration to replace specific parts of the auto-configuration.

```
#connection.properties
jdbc.driverClass=oracle.jdbc.driver.OracleDriver
jdbc.url=jdbc:oracle:thin:@localhost:1521:xe
jdbc.username=system
jdbc.password=manager
jdbc.initPoolSize=15
jdbc.maxPoolSize=25

@PropertySource(value={"classpath:connection.properties"})
public class SpringJdbcConfig {
        @Autowired
        private Environment env;

        @Bean
        public DataSource dataSource() {
                ComboPooledDataSource ds = new ComboPooledDataSource();
                try{
                        ds.setDriverClass(env.getProperty("jdbc.driverClass"));
                        ds.setJdbcUrl(env.getProperty("jdbc.url"));
                        ds.setUser(env.getProperty("jdbc.username"));
                        ds.setPassword(env.getProperty("jdbc.password"));
                        ds.setInitialPoolSize(env.getProperty("jdbc.initPoolSize", Integer.class));
                        ds.setMaxPoolSize(env.getProperty("jdbc.maxPoolSize", Integer.class));
                }catch(Exception e){
                        e.printStackTrace();
                }
                return ds;
        }
}
```

By default SpringBoot features such as external properties, logging, etc are available in the ApplicationContext only if we use SpringApplication. So, SpringBoot provides @SpringApplicationConfiguration annotation to configure the ApplicationContext for tests which uses SpringApplication behind the scenes.

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

Application.class is passed as a parameter to tell Spring Boot that this is the primary component.

Note: As an alternate to application.properties, we can use a **.yaml** file. YAML provides a JSON-like structured configuration compared to the flat properties file.

#application.yaml

```
Server:
        port: 9080
```

The @**SpringBootApplication** enables Spring component-scanning and Spring Boot auto-configuration. In fact, @SpringBootApplication combines three other useful annotations:

1. **@SpringBootConfiguration**—This annotation hints that the contained class declares one or more @Bean definitions. It can be used as an alternative to the Spring's standard `@Configuration` annotation. The @Configuration is a specialization of @Component hence candidate for component scanning i.e., needs to give configuration class package name in test class. But @SpringBootConfiguration can be found automatically (for example in tests) hence need not to give configuration class package name in test class.

2. **@ComponentScan**—Enables component-scanning so that the web controller classes and other components we write will be automatically discovered and registered as beans in the Spring application context. This annotation is save as <context:component-scan/> element.

3. **@EnableAutoConfiguration**—This enables the magic of Spring Boot auto-configuration.

Embedded Container

Spring-boot-starter-web pulls the spring-boot-starter-tomcat automatically which starts tomcat as a embedded container. So we don't have to deploy our application on any externally installed tomcat server.
That's exactly what Spring Boot's @**WebIntegrationTest** annotation does. By annotating a test class with @WebIntegrationTest, we declare that we want Spring Boot to not only create an application context for our test, but also to start an embedded servlet container. Once the application is running along with the embedded container, we can issue real HTTP requests against it and make assertions against the results.

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

Spring Boot Actuators

The problem with Spring component-scanning and autowiring is that it's hard to see how all of the components in an application are wired together.
The Spring Boot Actuators provide us details such as which beans have been configured, bean dependencies, autoconfig details, environment variables, configuration properties, memory usage, garbage collection, web requests, and data source usage.
The following dependency should be added in pom.xml file:

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

Spring Boot Acturaltors provide following details:

1. What beans have been configured in the Spring application context
2. What decisions were made by Spring Boot's auto-configuration
3. What environment variables, system properties, configuration properties, and command-line arguments are available to our application.
4. A trace of recent HTTP requests handled by our application.
5. Various metrics pertaining to memory usage, garbage collection, web requests, and data source usage

The Actuator provides following REST endpoints:

REST End Point	Description
/beans	Describes all beans in the application context and their relationship to each other.
/autoconfig	Provides an auto-configuration report describing what autoconfiguration conditions passed and failed.
/env	Retrieves all environment properties.
/health	Reports health metrics for the application, as provided by HealthIndicator implementations.
…	

Spring Boot Test

Spring Boot provides a number of utilities and annotations to help when testing our application. Test support is provided by two modules; **spring-boot-test** contains core items, and **spring-boot-test-autoconfigure** supports auto-configuration for tests.

Most developers use the **spring-boot-starter-test** starter which imports both Spring Boot test modules (such as spring-boot-test and spring-boot-test-autoconfigure) as well as JUnit, TestNG, Mockito, AssertJ, Hamcrest, etc.

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

The **SpringApplication** creates an appropriate ApplicationContext (depending on classpath), loads external property files such as application.properties, enables logging and other features of spring boot. At startup, SpringApplication loads all the properties and adds them to the Spring Environment class.

@SpringBootTest annotation can be used as an alternative to the standard spring-test @ContextConfiguration annotation when we need Spring Boot features. This annotation works by creating the ApplicationContext used in our tests via SpringApplication.

Note: Don't forget to add @RunWith(SpringRunner.class) to our test, otherwise the annotations will be ignored.

Example:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes={"SpirngJdbcConfig.class"})
Public class ApplicationTest{
        …
}
```

Spring Boot MVC

The spring-boot-starter-web pulls the spring-boot-starter-tomcat automatically which starts tomcat as a embedded container. So we don't have to deploy our application on any externally installed tomcat server.
Below dependency needed to get web starter:

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

Below dependency needed to process JSP pages:

```
        <dependency>
                <groupId>org.apache.tomcat.embed</groupId>
                <artifactId>tomcat-embed-jasper</artifactId>
                <scope>provided</scope>
        </dependency>
```

Below dependency needed to process html pages:

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

The @EnableWebMvc shouldn't be used i we want to get Spring Boot MVC features.
If we want to take complete control of Spring MVC, we can add @EnableWebMvc.
In traditional web applications, a war file is created and then deployed to a external servlet container, whereas Spring Boot packages all the dependencies, embedded servlet container as a fat JAR file.

http://localhost:9090/NewCustomer.jsp
http://localhost:9090/customers/registration/form

CommandLineRunner

If we need to execute some custom code just before boot application starting up? We can make that happen with a runner i.e., Spring Boot provides CommandLineRunner interface to run specific pieces of code when an application is fully started. When we want to execute some piece of code exactly before the application startup completes, we can use it then.

```
@SpringBootApplication
public class Hello implements CommandLineRunner{
        public static void main(String args){
                SpringApplication.run(Hello.class, args);
        }

        @Override
        public void run(String... arg0) throws Exception {
                ...
        }
}
```

2.Spring JPA

JPA-based applications use an implementation of **EntityManagerFactory** to get an instance of an **EntityManager**.

The JPA specification defines two kinds of entity managers:

1) **Application managed**

   With application managed entity managers, the application is responsible for opening or closing entity manager. This type of entity manager is most appropriate for use in standalone applications that don't run in a Java EE container.

2) **Container managed**

   Entity managers are created and managed by a Java EE container. The application doesn't interact with the entity manager factory at all. Instead, entity managers are obtained directly through injection or from JNDI. This type of entity manager is most appropriate for use by a Java EE container.

Regardless of which variety of EntityManager we want to use, Spring will take responsibility for managing EntityManagers for us.

If we want to use application managed entity manager, Spring plays the role of an application.

In we want to use the container managed entity manager, Spring plays the role of the container.

Hence as a developer we are good enough to configure appropriate factory bean as given below:

- **LocalEntityManagerFactoryBean** produces an application managed EntityManagerFactory.
- **LocalContainerEntityManagerFactoryBean** produces a container managed EntityManagerFactory.

Conclusion: When we're working with Spring JPA, the intricate details of dealing with either form of EntityManagerFactorys are hidden.

The only real difference between application managed and container managed entity manager factories, as far as Spring is concerned, is how each is configured in the Spring application context.

Application managed entity manager factories provide most of the configuration in **persistence.xml**.

#META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
        xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
        instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
        <persistence-unit name="SpringJPAPU" transaction-type="RESOURCE_LOCAL">
                <provider>org.hibernate.ejb.HibernatePersistence</provider>
                <class>edu.aspire.entities.Customer</class>
                <properties>
                    <property name="javax.persistence.jdbc.url" value="jdbc:oracle:thin:@localhost:1521:xe"/>
                    <property name="javax.persistence.jdbc.user" value="system"/>
                    <property name="javax.persistence.jdbc.password" value="manager"/>
                    <property name="javax.persistence.jdbc.driver" value="oracle.jdbc.driver.OracleDriver"/>
                </properties>
        </persistence-unit>
</persistence>
```

Because so much configuration goes into a persistence.xml file, little configuration is required in Spring.

```
@Configuration
Public class SpringJpaConfig{
    @Bean
    public LocalEntityManagerFactoryBean entityManagerFactoryBean() {
        LocalEntityManagerFactoryBean emfb = new LocalEntityManagerFactoryBean();
        emfb.setPersistenceUnitName("SpringJPAPU");
        return emfb;
    }
}
```

In case of Container managed JPA, an EntityManagerFactory can be produced using information provided by the container, which is spring container in this case.
Instead of configuring data-source details in persistence.xml, we should configure this information in the Spring application context.

```
@Configuration
Public class SpringJpaConfig{
    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName("oracle.jdbc.driver.OracleDriver");
        ds.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
        ds.setUsername("system");
        ds.setPassword("manager");
        return ds;
    }

    @Bean
    public JpaVendorAdapter hibJpaVendorAdapter() {
        HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
```

```
                adapter.setDatabase(Database.ORACLE);
                adapter.setShowSql(true);
                adapter.setGenerateDdl(false);
                //adapter.setDatabasePlatform("org.hibernate.dialect.Oracle10gDialect");
                return adapter;
        }

        @Bean
        public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource ds,
        JpaVendorAdapter jpaVendorAdapter) {
                LocalContainerEntityManagerFactoryBean emfb = new LocalContainerEntityManagerFactoryBean();
                emfb.setDataSource(ds);
                emfb.setJpaVendorAdapter(jpaVendorAdapter);
                emfb.setPackagesToScan("edu.aspire.entities");
                return emfb;
        }
}
```

We can use **jpaVendorAdapter** property to provide specifics about the particular JPA implementation (such as Hibernate, Toplink, etc) to use. Spring comes with a handful of JPA vendor adapters to choose from:

- HibernateJpaVendorAdapter
- OpenJpaVendorAdapter
- TopLinkJpaVendorAdapter

In this case, we're using Hibernate as a JPA implementation, so we configure it with a HibernateJpaVendorAdapter.

Just like all of Spring's other persistence integration options, Spring-JPA integration comes in template form with JpaTemplate. Nevertheless, template-based JPA has been set aside in favor of a pure JPA approach.

If the property is annotated with **@PersistenceContext**, then Spring can inject the EntityManager into the repository.

```
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
@Repository("custdao")
@Transactional
public class CustomerDaoImpl implements ICustomerDao {
        @PersistenceContext
        private EntityManager em;
        …
}
```

Example:

```
/*
CREATE TABLE CUSTOMER(CNO NUMBER(5)PRIMARY KEY, CNAME VARCHAR2(20), ADDRESS VARCHAR2(100),
PHONE NUMBER(15));
CREATE SEQUENCE CUSTOMER_SEQ;
*/
package edu.aspire.entities;
import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;

@Entity
@Table(name = "CUSTOMER")
@NamedQueries({
	@NamedQuery(name = "cust.findAll", query = "select c from Customer c"),
	@NamedQuery(name = "cust.findByName", query = "select c from Customer c where c.cname=?") })
public class Customer implements Serializable {
	@Id
	@Column(name = "CNO")
	@SequenceGenerator(name="CUSTOMER_CNO_GENERATOR", sequenceName="CUSTOMER_SEQ", allocationSize=1)
	@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="CUSTOMER_CNO_GENERATOR")
	private int cno;

	@Column(name = "CNAME")
	private String cname;

	@Column(name = "ADDRESS")
	private String address;

	@Column(name = "PHONE")
	private long phone;
```

```
        public Customer() {  }
        public int getCno() { return cno; }
        public void setCno(int cno) { this.cno = cno; }
        public String getCname() { return cname; }
        public void setCname(String cname) { this.cname = cname; }
        public String getAddress() { return address; }
        public void setAddress(String address) { this.address = address; }
        public long getPhone() { return phone; }
        public void setPhone(long phone) { this.phone = phone;  }
}

package edu.aspire.daos;
import java.util.List;
import edu.aspire.entities.Customer;
public interface ICustomerDao {
        public void create(Customer c);
        public Customer read(int cno);
        public void update(Customer c);
        public void delete(Customer c);

        //finder methods
        public List<Customer> findAll();
        public List<Customer> findByName(String cname);
}

package edu.aspire.daos;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
import edu.aspire.entities.Customer;

@Repository("custdao")
@Transactional
public class CustomerDaoImpl implements ICustomerDao {
        @PersistenceContext
        private EntityManager em;
```

```
	@Override
	public void create(Customer c) {
		em.persist(c);
		System.out.println("Customer details successfully inserted");
	}
	@Override
	public Customer read(int cno) {
		return em.find(Customer.class, cno);
	}
	@Override
	public void update(Customer c) {
		em.merge(c);
		System.out.println("Customer details successfully modified");
	}
	@Override
	public void delete(Customer c) {
		em.remove(em.merge(c));
		System.out.println("Customer details successfully deleted");
	}
	@Override
	public List<Customer> findAll(){
		Query q = em.createNamedQuery("cust.findAll");
		return q.getResultList();
	}
	@Override
	public List<Customer> findByName(String cname){
		Query q = em.createNamedQuery("cust.findByName");
		q.setParameter(1, cname);
		return q.getResultList();
	}
}
```

```
package edu.aspire.config;
import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;
import org.springframework.boot.SpringBootConfiguration;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.domain.EntityScan;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.Database;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@SpringBootConfiguration
@ComponentScan(basePackages = {"edu.aspire.daos"})
@EntityScan(basePackages  = {"edu.aspire.entities"})
@EnableAutoConfiguration
@EnableTransactionManagement
public class SpringJpaConfig {
        //Not required because of DataSourceConfiguration.Tomcat matched:
        /*@Bean
        public DataSource dataSource() {
                DriverManagerDataSource ds = new DriverManagerDataSource();
                ds.setDriverClassName("oracle.jdbc.driver.OracleDriver");
                ds.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
                ds.setUsername("system");
                ds.setPassword("manager");
                return ds;
        }*/

        //Not required because of JpaBaseConfiguration#jpaVendorAdapter matched:
        /*@Bean
        public JpaVendorAdapter hibJpaVendorAdapter() {
                HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
                adapter.setDatabase(Database.ORACLE);
                adapter.setShowSql(true);
                adapter.setGenerateDdl(false);
                // adapter.setDatabasePlatform("org.hibernate.dialect.Oracle10gDialect");
                return adapter;
        }*/

        //Not required because of HibernateJpaAutoConfiguration matched:
        /*@Bean
        public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource ds,
                            JpaVendorAdapter jpaVendorAdapter) {
```

```
			LocalContainerEntityManagerFactoryBean emfb = new LocalContainerEntityManagerFactoryBean();
			emfb.setDataSource(ds);
			emfb.setJpaVendorAdapter(jpaVendorAdapter);
			emfb.setPackagesToScan("edu.aspire.entities");
			return emfb;
	}*/

	//Not required because of JpaBaseConfiguration#transactionManager matched:
	/*@Bean
	public PlatformTransactionManager transactionManager(LocalContainerEntityManagerFactoryBean entityManagerFactory) {
			EntityManagerFactory factory = entityManagerFactory.getObject();
			return new JpaTransactionManager(factory);
	}*/
}
```

#src/main/resources/application.properties

```
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=system
spring.datasource.password=manager

#tomcat-connection settings
spring.datasource.tomcat.initialSize=20
spring.datasource.tomcat.max-active=25

spring.jpa.show-sql=true
#spring.jpa.hibernate.ddl-auto=create

debug=true
```

```
package edu.aspire.test;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import edu.aspire.config.SpringJpaConfig;
import edu.aspire.daos.ICustomerDao;
import edu.aspire.entities.Customer;
```

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes={SpringJpaConfig.class})
public class SpringJpaTest {
        @Autowired
        ApplicationContext context;

        @Test
        public void testInsertJpa() {
                ICustomerDao custDao = (ICustomerDao) context.getBean("custdao");
                Customer c = new Customer();
                c.setCname("ramesh");
                c.setAddress("Ameerpet");
                c.setPhone(7799108899L);
                custDao.create(c);
        }

        /*@Test
        public void testReadJpa() {
                ICustomerDao custDao = (ICustomerDao) context.getBean("custdao");
                Customer c = custDao.read(1);
                System.out.println(c.getCno() + "  " + c.getCname() +" " + c.getAddress() +"  " + c.getPhone());
        }

        @Test
        public void testUpdateJpa(){
                ICustomerDao custDao = (ICustomerDao) context.getBean("custdao");
                Customer c = custDao.read(1);
                c.setPhone(7799208899L);
                custDao.update(c);
        }*/

        /*@Test
        public void testDeleteJpa(){
                ICustomerDao custDao = (ICustomerDao) context.getBean("custdao");
                Customer c = custDao.read(1);
                custDao.delete(c);
        }*/

        /*@Test
        public void testFindAllJpa(){
                ICustomerDao custDao = (ICustomerDao) context.getBean("custdao");
                List<Customer> custs = custDao.findAll();
                System.out.println("***FindAll***:" + custs.size());
        }*/

        /*@Test
        public void testFindByNameJpa(){
                ICustomerDao custDao = (ICustomerDao) context.getBean("custdao");
                List<Customer> custs = custDao.findByName("ramesh");
                System.out.println("***FindByName***:" + custs.size());
        }*/
}
```

#pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>edu.aspire</groupId>
        <artifactId>SpringJPA </artifactId>
        <version>1</version>
        <packaging>jar</packaging>
        <name>Spring JPA Project</name>
        <url>http://www.java2aspire.com</url>

        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>1.5.3.RELEASE</version>
        </parent>

        <dependencies>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-data-jpa</artifactId>
                </dependency>

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

            <dependency>
                <groupId>oracle</groupId>
                <artifactId>oracle-jdbc</artifactId>
                <version>11</version>
            </dependency>
        </dependencies>

        <build>
            <plugins>
                <!-- using Java 8 -->
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-compiler-plugin</artifactId>
                    <configuration>
                        <source>1.8</source>
                        <target>1.8</target>
                    </configuration>
                </plugin>
            </plugins>
        </build>
    </project>
```

3.Spring DATA JPA

The create(), read(), update() and delete() methods are fairly common across all DAOs. The only difference is domain types.
Spring DATA JPA internally provides implementation for all such common methods (as part of org.springframework.data.repository.**CrudRepository**). Hence developer need not to write implementation for these common methods.
Example:

```
Package edu.aspire.daos;
import org.springframework.data.jpa.repository.JpaRepository;
public interface CustomerDao extends JpaRepository<Customer, Integer>{
}
```

The JpaRepository is parameterized such that it takes domain class and type of ID. It also inherits 18 methods for performing common persistence operations, such as saving, deleting, finding, etc from CrudRepository class.

Spring DATA JPA will automatically provide implementation for all these common methods if and only if we add **@EnableJpaRepositories(basePackages="edu.aspire.daos")** in configuration class.
Example:

```
@SpringBootConfiguration
@EnableJpaRepositories(basePackages = "edu.aspire.daos")
public class SpringDataConfig {
        …
}
```

The @EnableJpaRepositories scans its base package for any interfaces that extends JpaRepository interface. When it finds any interface extends JpaRepository, it automatically generates an implementation of that dao interface.

*Spring DATA JPA not only provides implementation for commonly used methods but also provides a way to add **custom methods**. The method signature tells Spring Data JPA everything it needs to know in order to create an implementation for the method. Spring Data defines a sort of **domain-specific language** (**DSL**) where persistence details are expressed in **method signatures**.*
Example:

```
public interface CustomerDao extends JpaRepository<Customer, Integer>{
        //finder methods
        public List<Customer> findByCname(String cname);
}
```

Repository methods are composed of a **verb**, an optional **subject**, the word **By**, and a **predicate**.

In case of findByCname(), the verb is **find** and the predicate is **Cname**; the subject isn't specified and is implied to be a Customer.
Spring DATA JPA allows **four verbs** in the method name: **get, read, find, and count**. The get, read, and find verbs are synonymous; all three result in repository methods that query for data and return objects. The count verb, on the other hand, returns a count of matching objects, rather than the objects themselves.

The method name, **readCustomerByFirstNameOrLastName**(String first, String last), the verb is **read** and the predicate is **FirstNameOrLastName**.
The predicate is the most interesting part of the method name.
We can use any of the following **comparion operator** between property and parameter:

1) IsAfter, After, IsGreaterThan, GreaterThan
2) IsGreaterThanEqual, GreaterThanEqual
3) IsBefore, Before, IsLessThan, LessThan
4) IsLessThanEqual, LessThanEqual
5) IsBetween, Between
6) IsNull, Null
7) IsNotNull, NotNull
8) IsIn, In
9) IsNotIn, NotIn
10) IsStartingWith, StartingWith, StartsWith
11) IsEndingWith, EndingWith, EndsWith
12) IsContaining, Containing, Contains
13) IsLike, Like
14) IsNotLike, NotLike
15) IsTrue, True
16) IsFalse, False
17) Is, Equals
18) IsNot, Not

The property value will be compared against the parameter. The full method signature looks like this:
public List<Customer> readByFirstnameOrLastname(String first, String last);
In above method signature, the comparison operator is left off, it's implied to be an **equals** operation.

In method signature public List<Customer> readByFirstname**IgnoringCase**OrLastname**IgnoresCase**(String first, String last), the conditions are IgnoringCase or IgnoresCase to ignore case on the firstname and lastname properties.
Note: The IgnoringCase and IgnoresCase are synonymous.
As an alternative to IgnoringCase/IgnoresCase, we may also use either **AllIgnoringCase** or **AllIgnoresCase**.

    List<Customer> readByFirstnameOrLastname**AllIgnoresCase**(String first, String last);

We can sort the results by adding **OrderBy** at the end of the method name.

To sort the results in ascending order by the lastname property, the method signature is:

Public List<Customer> readByFirstnameOrLastname**OrderBy**Lastname**Asc**(String first, String last);

To sort results in ascending order by the firstname property and decending order by the lastname property, the method signature is:
List<Customer> readByFirstnameOrLastnameOrderByFirstnameAscLastnameDesc(String first, String last);

Althouth Spring DATA JPA generates an implementation method to query for almost anything we can imagine, nevertheless, Spring Data's mini-DSL has its limits, and sometimes it isn't convenient or even possible to express the desired query in a method name. When that happens, Spring DATA JPA provides ***@Query*** *annotation to write query explicitly*.

Suppose we want to create a repository method to find all customers whose email address is a Gmail address. One way to do this is to define a findByEmailLike() method and pass in %gmail.com to find Gmail users. But it would be nice to define a more convenient findAllGmailCustomers() method that doesn't require the partial email address to be passed in: List<Customer> findAllGmailCustomers(). Unfortunately, this method name doesn't adhere to Spring Data's method-naming conventions (DSL). In situations where the desired data can't be adequately expressed in the method name, we can use the @Query annotation to provide Spring Data with the query that should be performed. For the findAllGmailCustomers() method, we might use
@Query like this:

@Query("select c from Customer c where c.email like '%gmail.com'")
Public List<Customer> findAllGmailCustomers();

We still don't write the implementation of the findAllGmailCustomrs() method. We only give the query, hinting to Spring Data JPA about how it should implement the method.
Also, @Query can also be useful if we followed the naming convention, the method name would be incredibly long. In such situation, we'd probably rather come up with a shorter method name and use @Query to specify how the method should query the database.
The @Query annotation is handy for adding custom query methods to a Spring Data JPA-enabled interface.

Sometimes we cannot describe functionality with Spring Data's method-naming conventions or even with a query given in the @Query annotation. Such specific scenarios can be implemented using EntityManager (using Spring JPA) and remaining functionalities can be worked with Spring DATA JPA i.e., we can ***mix*** *EntitityManager (to work at lower level) in Spring JPA with Spring DATA JPA (grunt work for the stuff it knows how to do).*

The Spring Data JPA generates the implementation for a repository interface whose name is same as interface's name and **postfixed with impl**.
When Spring Data JPA generates the implementation for a repository interface, it also looks for a class whose name is the same as the interface's name **postfixed with Impl**. If the class exists, Spring Data JPA merges its methods with those generated by Spring Data JPA. For the CustomerDao interface, the class it looks for is named CustomerDao**Impl**.
Example:

```
Public interface IFindCustomerMobile{
        Public Customer getCustomer(long phone);
}

Package edu.aspire.daos;
public class CustomerDaoImpl implements IFindCustomerMobile {
        @PersistenceContext
        private EntityManager em;

        public int getCustomer(long phone) {
        …
        }
}
```

Notice that CustomerDaoImpl doesn't implement the CustomerDao interface. Spring Data JPA is still responsible for implementing that interface. Instead CustomerDaoImpl implements IFindCustomerMobile interface.
We should also make sure the getCustomer() method is declared in the CustomerDao interface. The easy way to do that and avoid duplicating code is to change CustomerDao so that it extends IFindCustomerMobile.

```
import org.springframework.data.jpa.repository.JpaRepository;
public interface CustomerDao extends JpaRepository<Customer, Integer>, IFindCustomerMobile{
        public List<Customer> findAll();
        public List<Customer> findByCname(String name);
}
```

Example:

```
/*
CREATE TABLE CUSTOMER(CNO NUMBER(5)PRIMARY KEY, CNAME VARCHAR2(20), ADDRESS VARCHAR2(100),
PHONE NUMBER(15));
CREATE SEQUENCE CUSTOMER_SEQ;
*/
package edu.aspire.entities;
import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;

@Entity
@Table(name = "CUSTOMER")
public class Customer implements Serializable {
        @Id
```

```
	@Column(name = "CNO")
	@SequenceGenerator(name="CUSTOMER_DNO_GENERATOR", sequenceName="CUSTOMER_SEQ", allocationSize=1)
	@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="CUSTOMER_DNO_GENERATOR")
	private int cno;

	@Column(name = "CNAME")
	private String cname;

	@Column(name = "ADDRESS")
	private String address;

	@Column(name = "PHONE")
	private long phone;

	public Customer() { }
	public int getCno() { return cno; }
	public void setCno(int cno) { this.cno = cno; }
	public String getCname() { return cname; }
	public void setCname(String cname) { this.cname = cname; }
	public String getAddress() { return address; }
	public void setAddress(String address) { this.address = address; }
	public long getPhone() { return phone; }
	public void setPhone(long phone) { this.phone = phone; }
}
```

```
package edu.aspire.daos;
import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import edu.aspire.entities.Customer;
//@Transactional
public interface ICustomerDao extends JpaRepository<Customer, Integer>{
	public List<Customer> findAll();
	public List<Customer> findByCname(String name);
}
```

#src/main/config/application.properties

```
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=system
spring.datasource.password=manager

#tomcat-connection settings
spring.datasource.tomcat.initialSize=20
spring.datasource.tomcat.max-active=25

spring.jpa.show-sql=true
#spring.jpa.hibernate.ddl-auto=create

debug=true
```

```
package edu.aspire.config;
import org.springframework.boot.SpringBootConfiguration;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.domain.EntityScan;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@SpringBootConfiguration
@EnableJpaRepositories(basePackages = "edu.aspire.daos")
@EntityScan(basePackages  = {"edu.aspire.entities"})
@EnableAutoConfiguration
@EnableTransactionManagement
public class SpringDataJpaConfig {
	//Not required because of DataSourceConfiguration.Tomcat matched:
	/*@Bean
	public DataSource dataSource() {
		DriverManagerDataSource ds = new DriverManagerDataSource();
		ds.setDriverClassName("oracle.jdbc.driver.OracleDriver");
		ds.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
		ds.setUsername("system");
		ds.setPassword("manager");
		return ds;
	}*/

	//Not required because of JpaBaseConfiguration#jpaVendorAdapter matched
	/*@Bean
	public JpaVendorAdapter hibJpaVendorAdapter() {
		HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
		adapter.setDatabase(Database.ORACLE);
		adapter.setShowSql(true);
		adapter.setGenerateDdl(false);
		// adapter.setDatabasePlatform("org.hibernate.dialect.Oracle10gDialect");
		return adapter;
	}*/

	//Not required because of HibernateJpaAutoConfiguration matched
	//Method name must be entitiyManagerFactory because Spring Data Jpa by default looks for an
						EntityManagerFactory named 'entityManagerFactory'
	/*@Bean
	public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource ds,
				JpaVendorAdapter jpaVendorAdapter) {
		LocalContainerEntityManagerFactoryBean emfb = new LocalContainerEntityManagerFactoryBean();
		emfb.setDataSource(ds);
		emfb.setJpaVendorAdapter(jpaVendorAdapter);
		emfb.setPackagesToScan("edu.aspire.entities");
		return emfb;
	}*/

	//Not required because of JpaBaseConfiguration#transactionManager matched
	/*@Bean
	public PlatformTransactionManager transactionManager(LocalContainerEntityManagerFactoryBean entityManagerFactory) {
		EntityManagerFactory factory = entityManagerFactory.getObject();
		return new JpaTransactionManager(factory);
	}*/
}

package edu.aspire.test;
import java.util.List;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import edu.aspire.config.SpringDataJpaConfig;
import edu.aspire.daos.ICustomerDao;
import edu.aspire.entities.Customer;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes={SpringDataJpaConfig.class})
public class SpringDataJpaTest {
	@Autowired
	ApplicationContext context;
```

```
@Autowired
ICustomerDao custDao;

@Test
public void testInsertJpa() {
            Customer c = new Customer();
            c.setCname("ramesh");
            c.setAddress("Ameerpet");
            c.setPhone(7799108899L);
            custDao.save(c);
}

/*@Test
public void testReadJpa() {
            Customer c = custDao.findOne(1);
            System.out.println(c.getCno() + "  " + c.getCname() +" " + c.getAddress() +"  " + c.getPhone());
}

@Test
public void testUpdateJpa(){
            Customer c = custDao.findOne(1);
            c.setPhone(7799208899L);
            custDao.save(c); //In Spring Data JPA the save() is either persist() or merge() based on primary key present or not.
}

@Test
public void testDeleteJpa(){
            Customer c = custDao.findOne(1);
            custDao.delete(c);
}

@Test
public void testFindAllJpa(){
            List<Customer> custs = custDao.findAll();
            System.out.println("***FindAll***:" + custs.size());
}

@Test
public void testFindByNameJpa(){
			List<Customer> custs = custDao.findByCname("ramesh");
			System.out.println("***FindByCname***:" + custs.size());
		}*/
}
```

#pom.xml
Same as previous as Spring JPA example.

CREATING FARE SCHEMA

Step 1: Connect to database
C:\>sqlplus system/manager@xe

Step2: Create tablespace

```
CREATE TABLESPACE tbs_fareuser DATAFILE 'tbs_fareuser.dat' SIZE
10M AUTOEXTEND ON;
```

Note: alter session set "_ORACLE_SCRIPT"=true; This is required in Oracle 12c

Step3: Create a new user in Oracle

```
CREATE USER fareuser IDENTIFIED BY aspire123 DEFAULT TABLESPACE
tbs_fareuser QUOTA unlimited on tbs_fareuser;
```

Note: In oracle a schema is created when a user is created.

Step4: Grant permissions

```
GRANT create session TO fareuser;
GRANT create table TO fareuser;
GRANT create sequence TO fareuser;
```

Step5: Disconnect from system account and connect to fareuser

```
Sql>exit
C:\>sqlplus fareuser/aspire123@xe
```

Step6: Create tables and sequences

```
drop table fare cascade constraints;
drop sequence fare_seq;

create table fare (id number(19) primary key, fare
varchar2(255), flight_date varchar2(255), flight_number
varchar2(255));

create sequence fare_seq start with 1 increment by 1;
```

Step7: Insert records

```
insert into fare(id, fare, flight_date, flight_number) values
(fare_seq.nextVal, '100', '22-JAN-16', 'BF100');
insert into fare(id, fare, flight_date, flight_number) values
(fare_seq.nextVal, '101', '22-JAN-16', 'BF101');
insert into fare(id, fare, flight_date, flight_number) values
(fare_seq.nextVal, '102', '22-JAN-16', 'BF102');
```

```
insert into fare(id, fare, flight_date, flight_number) values
(fare_seq.nextVal, '103', '22-JAN-16', 'BF103');
insert into fare(id, fare, flight_date, flight_number) values
(fare_seq.nextVal, '104', '22-JAN-16', 'BF104');
insert into fare(id, fare, flight_date, flight_number) values
(fare_seq.nextVal, '105', '22-JAN-16', 'BF105');
insert into fare values (fare_seq.nextVal, '106', '22-JAN-16',
'BF106');

commit;
```

Step8: Read data from FAREUSER schema

```
SELECT * FROM "FAREUSER"."FARE";
```

ID	FLIGHT_NUMBER	FLIGHT_DATE	FARE
1	BF100	22-JAN-16	100
2	BF101	22-JAN-16	101
3	BF102	22-JAN-16	102
4	BF103	22-JAN-16	103
5	BF104	22-JAN-16	104
6	BF105	22-JAN-16	105
7	BF106	22-JAN-16	106

CREATING SEARCH SCHEMA

Step 1: Connect to database (ignore if already connected)
C:\>sqlplus system/manager@xe

Step2: Create tablespace

```
CREATE TABLESPACE tbs_searchuser DATAFILE 'tbs_searchuser.dat'
SIZE 10M AUTOEXTEND ON;
```

Note: alter session set "_ORACLE_SCRIPT"=true; This is required in Oracle 12c

Step3: Create a new user in Oracle

```
CREATE USER searchuser IDENTIFIED BY aspire123 DEFAULT
TABLESPACE tbs_searchuser QUOTA unlimited on tbs_searchuser;
```

Note: In oracle a schema is created when a user is created.

Step4: Grant permissions

```
GRANT create session TO searchuser;
GRANT create table TO searchuser;
GRANT create sequence TO searchuser;
```

Step5: Disconnect from system account and connect to searchuser

```
Sql>exit
C:\>sqlplus searchuser/aspire123@xe
```

Step6: Create tables and sequences

```
drop table fare cascade constraints;
drop table flight cascade constraints;
drop table inventory cascade constraints;

drop sequence fare_seq;
drop sequence flight_seq;
drop sequence inventory_seq;

create sequence fare_seq start with 1 increment by 1;
create sequence flight_seq start with 1 increment by 1;
create sequence inventory_seq start with 1 increment by 1;

create table fare (fare_id number(19) primary key, currency
varchar2(255), fare varchar2(255));

create table inventory (inv_id number(19) primary key, count
number(10) not null);

create table flight (id number(19) primary key, destination
varchar2(255), flight_date varchar2(255),
flight_number varchar2(255), origin varchar2(255), fare_id
number(19) references fare(fare_id), inv_id number(19)
references inventory(inv_id));

Step7: Insert records
insert into fare (currency, fare, fare_id) values ('USD', 100,
fare_seq.nextVal);
insert into fares (currency, fare, fare_id) values ('USD', 101,
fare_seq.nextVal);
insert into fare (currency, fare, fare_id) values ('USD', 102,
fare_seq.nextVal);
insert into fare (currency, fare, fare_id) values ('USD', 103,
fare_seq.nextVal);
insert into fare (currency, fare, fare_id) values ('USD', 104,
fare_seq.nextVal);
insert into fare (currency, fare, fare_id) values ('USD', 105,
fare_seq.nextVal);
```

```
insert into fare (currency, fare, fare_id) values ('USD', 106,
fare_seq.nextVal);

insert into inventory (count, inv_id) values (100,
inventory_seq.nextVal);
insert into inventory (count, inv_id) values (100,
inventory_seq.nextVal);
insert into inventory (count, inv_id) values (100,
inventory_seq.nextVal);
insert into inventory (count, inv_id) values (100,
inventory_seq.nextVal);
insert into inventory (count, inv_id) values (100,
inventory_seq.nextVal);
insert into inventory (count, inv_id) values (100,
inventory_seq.nextVal);
insert into inventory (count, inv_id) values (100,
inventory_seq.nextVal);

insert into flight (id, flight_number, origin, destination,
flight_date, fare_id, inv_id) values (flight_seq.nextVal,
'BF100', 'SEA', 'SFO', '22-JAN-16', 1, 1);
insert into flight (id, flight_number, origin, destination,
flight_date, fare_id, inv_id) values (flight_seq.nextVal,
'BF101', 'NYC', 'SFO', '22-JAN-16', 2, 2);
insert into flight (id, flight_number, origin, destination,
flight_date, fare_id, inv_id) values (flight_seq.nextVal,
'BF102', 'CHI', 'SFO', '22-JAN-16', 3, 3);
insert into flight (id, flight_number, origin, destination,
flight_date, fare_id, inv_id) values (flight_seq.nextVal,
'BF103', 'HOU', 'SFO', '22-JAN-16', 4, 4);
insert into flight (id, flight_number, origin, destination,
flight_date, fare_id, inv_id) values (flight_seq.nextVal,
'BF104', 'LAX', 'SFO', '22-JAN-16', 5, 5);
insert into flight (id, flight_number, origin, destination,
flight_date, fare_id, inv_id) values (flight_seq.nextVal,
'BF105', 'NYC', 'SFO', '22-JAN-16', 6, 6);
insert into flight (id, flight_number, origin, destination,
flight_date, fare_id, inv_id) values (flight_seq.nextVal,
'BF106', 'NYC', 'SFO', '22-JAN-16', 7, 7);

commit;
```

Step8: Read data from SEARCHUSER schema

```
SELECT * FROM "SEARCHUSER"."FARE";
```

FARE_ID	FARE	CURRENCY
1	100	USD
2	101	USD
3	102	USD
4	103	USD
5	104	USD
6	105	USD
7	106	USD

```
SELECT * FROM "SEARCHUSER"."INVENTORY";
```

INV_ID	COUNT
1	100
2	100
3	100
4	100
5	100
6	100
7	100

```
SELECT * FROM "SEARCHUSER"."FLIGHT";
```

ID	FLIGHT_NUMBER	FLIGHT_DATE	ORIGIN	DESTINATION	FARE_ID	INV_ID
1	BF100	22-JAN-16	SEA	SFO	1	1
2	BF101	22-JAN-16	NYC	SFO	2	2
3	BF102	22-JAN-16	CHI	SFO	3	3
4	BF103	22-JAN-16	HOU	SFO	4	4
5	BF104	22-JAN-16	LAX	SFO	5	5
6	BF105	22-JAN-16	NYC	SFO	6	6
7	BF106	22-JAN-16	NYC	SFO	7	7

CREATING BOOKING SCHEMA

Step 1: Connect to database (ignore if already connected)
C:\>sqlplus system/manager@xe

Step2: Create tablespace

```
CREATE TABLESPACE tbs_bookinguser DATAFILE 'tbs_bookinguser.dat'
SIZE 10M AUTOEXTEND ON;
```

Note: alter session set "_ORACLE_SCRIPT"=true; This is required in Oracle 12c

Step3: Create a new user in Oracle

```
CREATE USER bookinguser IDENTIFIED BY aspire123 DEFAULT
TABLESPACE tbs_bookinguser QUOTA unlimited on tbs_bookinguser;
```

Note: In oracle a schema is created when a user is created.

Step4: Grant permissions

```
GRANT create session TO bookinguser;
GRANT create table TO bookinguser;
GRANT create sequence TO bookinguser;
```

Step5: Disconnect from system account and connect to bookinguser

```
Sql>exit
C:\>sqlplus bookinguser/aspire123@xe
```

Step6: Create tables and sequences

```
drop table booking_record cascade constraints;
drop table inventory cascade constraints;
drop table passenger cascade constraints;

drop sequence booking_seq;
drop sequence inventory_seq;
drop sequence passenger_seq;

create sequence booking_seq start with 1 increment by 1;
create sequence inventory_seq start with 1 increment by 1;
create sequence passenger_seq start with 1 increment by 1;

create table booking_record (id number(19) primary key,
booking_date timestamp, destination varchar2(255), fare
varchar2(255), flight_date varchar2(255), flight_number
varchar2(255), origin varchar2(255), status varchar2(255));

create table inventory (id number(19) primary key, available
number(10) not null, flight_date varchar2(255), flight_number
varchar2(255));

create table passenger (id number(19) primary key, first_name
varchar2(255), gender varchar2(255), last_name varchar2(255),
booking_id number(19) references booking_record(id));
```

```
Step7: Insert records
insert into inventory (flight_number, flight_date, available, id)
values ('BF100', '22-JAN-16', 100, inventory_seq.nextVal);
insert into inventory (flight_number, flight_date, available, id)
values ('BF101', '22-JAN-16', 100, inventory_seq.nextVal);
insert into inventory (flight_number, flight_date, available, id)
values ('BF102', '22-JAN-16', 100, inventory_seq.nextVal);
```

```
insert into inventory (flight_number, flight_date, available, id)
values ('BF103', '22-JAN-16', 100, inventory_seq.nextVal);
insert into inventory (flight_number, flight_date, available, id)
values ('BF104', '22-JAN-16', 100, inventory_seq.nextVal);
insert into inventory (flight_number, flight_date, available, id)
values ('BF105', '22-JAN-16', 100, inventory_seq.nextVal);
insert into inventory (flight_number, flight_date, available, id)
values ('BF106', '22-JAN-16', 100, inventory_seq.nextVal);

commit;
```

Step8: Read data from BOOKINGUSER schema

```
SELECT * FROM "BOOKINGUSER"."INVENTORY";
```

ID	FLIGHT_NUMBER	FLIGHT_DATE	AVAILABLE
1	BF100	22-JAN-16	100
2	BF101	22-JAN-16	**99**
3	BF102	22-JAN-16	100
4	BF103	22-JAN-16	100
5	BF104	22-JAN-16	100
6	BF105	22-JAN-16	100
7	BF106	22-JAN-16	100

```
SELECT * FROM "BOOKINGUSER"."BOOKING_RECORD";
```

ID	BOOKING_DATE	ORIGIN	DESTINATION	FARE	FLIGHT_DATE	FLIGHT_NUMBER	STATUS
1	2017-06-06 20:46:01	NYC	SFO	101	22-JAN-16	BF101	**BOOKING_CONFIRMED**

```
SELECT * FROM "BOOKINGUSER"."PASSENGER";
```

ID	FIRST_NAME	LAST_NAME	GENDER	BOOKING_ID
1	Gean	Franc	Male	1

CREATING CHECKIN SCHEMA

Step 1: Connect to database (ignore if already connected)
C:\>sqlplus system/manager@xe

Step2: Create tablespace

```
CREATE TABLESPACE tbs_checkinuser DATAFILE 'tbs_checkinuser.dat'
SIZE 10M AUTOEXTEND ON;
```

Note: alter session set "_ORACLE_SCRIPT"=true; This is required in Oracle 12c

Step3: Create a new user in Oracle

```
CREATE USER checkinuser IDENTIFIED BY aspire123 DEFAULT
TABLESPACE tbs_checkinuser QUOTA unlimited on tbs_checkinuser;
```

Note: In oracle a schema is created when a user is created.

Step4: Grant permissions

```
GRANT create session TO checkinuser;
GRANT create table TO checkinuser;
GRANT create sequence TO checkinuser;
```

Step5: Disconnect from system account and connect to checkinuser

```
Sql>exit
C:\>sqlplus checkinuser/aspire123@xe
```

Step6: Create tables and sequences

```
drop table check_in_record cascade constraints;
drop sequence checkin_seq;

create sequence checkin_seq start with 1 increment by 1;

create table check_in_record (id number(19)primary key,
booking_id number(19) not null, check_in_time timestamp,
first_name varchar2(255), flight_date varchar2(255),
flight_number varchar2(255), last_name varchar2(255),
seat_number varchar2(255));
```

```
Step7: Insert records
No need to insert data manually
```

Step8: Read data from CHECKINUSER schema

```
SELECT * FROM "CHECKINUSER"."CHECK_IN_RECORD";
```

ID	BOOKING_ID	CHECK_IN_TIME	FIRST_ NAME	LAST_NAME	FLIGHT_DATE	FLIGHT_NUMBER	SEAT_NUMBER
1	1	2017-06-06 21:18:46	Gean	Franc	22-JAN-16	BF101	**28A**

Other useful commands

DROP TABLESPACE tbs_testuser INCLUDING CONTENTS AND DATAFILES;

```
DROP USER testuser;
```

SPRING MICROSERVICES

K.RAMESH

ASPIRE Technologies

#501, 5th Floor, Mahindra Residency, Maithrivanam Road, Ameerpet, Hyderabad

Ph: 07799 10 8899, 07799 20 8899

E-Mail:ramesh@java2aspire.com website: www.java2aspire.com

1. INTRODUCTION

Microservice is an architecture style in which complex applications are **decomposed** into smaller services that work together to form larger business services. Microservices are autonomous, self-contained, and independently deployable.

Softwares

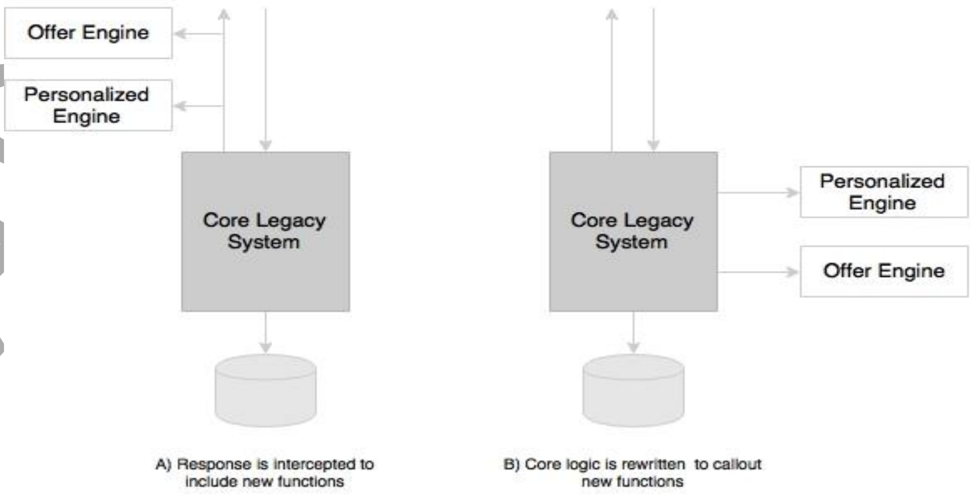
1. JDK 1.8
2. Spring Tool Suite 3.7.2 (**STS**)
3. Maven 3.x
4. Spring Framework 4.2.6.RELEASE or above
5. Spring Boot 1.3.5.RELEASE or above
6. RabbitMQ 3.5.6
7. Git
8. Spring cloud
9. Oracle DB

Microservices are not invented; rather they are evoluated from the previous architecture styles.

The evolution of microservices

The microservices evolution is greatly influenced by the **Quick business demands, Evolution of technologies** and **Evolution of Architectures** in the last few years.

Enterprisers want to quickly develop personalization engine (based on the customer's past shopping) or offers and plug them into their legacy application.



Offer Engine
Personalized Engine
Core Legacy System
A) Response is intercepted to include new functions
Core Legacy System
Personalized Engine
Offer Engine
B) Core logic is rewritten to callout new functions

Modern architectures are expected to maximize the ability to replace their parts and minimize the cost of replacing their parts. The microservices approach is a means to achieving this.
Enterprises are no longer interested in developing consolidated applications to manage their end-to-end business functions as they did a few years ago.
Microservices promise more agility, speed of delivery, and scale compared to traditional monolithic applications resulting in less overall cost.
Emerging technologies such as Cloud, NoSQL, Hadoop, Social media, Internet of Things (IoT), AWS, Docker, Angular JS, etc have also made us rethink the way we build software systems.
Application architecture has always been evolving alongside demanding business requirements and the evolution of technologies.
Different architecture approaches and styles such as mainframes, client server, MVC, and service-oriented were popular at different timeframes. Irrespective of the choice of architecture styles, we always used to build **monolithic architectures**.
The microservices architecture evolved as a result of modern business demands such as agility and speed of delivery, emerging technologies, and learning from previous generations of architectures.

What are microservices?

Microservices are not invented rather many organizations such as Netflix, Amazon, and eBay successfully used the **divide-and-conquer** technique to functionally partition their **monolithic** applications into smaller **atomic** units, each performing a single function.

In below diagram, each layer holds all three business capabilities pertaining to that layer. The presentation layer has web components of all the three modules, the business layer has business components of all the three modules, and the database hosts tables of all the three modules. In most cases, **layers are physically spreadable, whereas modules within a layer are hardwired**.

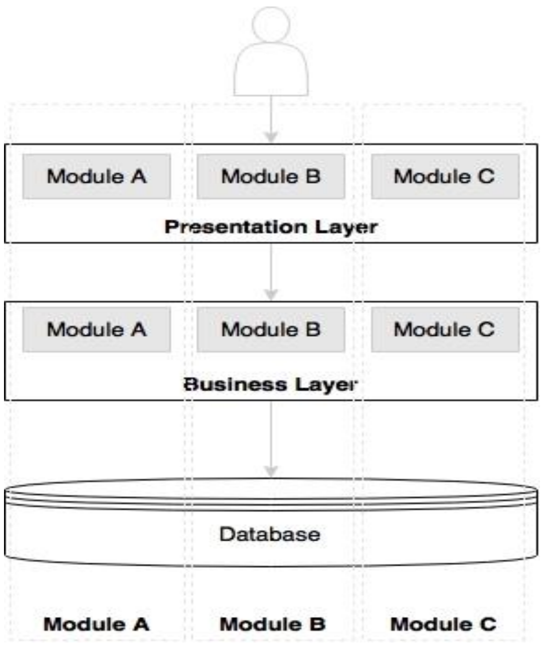


Figure: Monolithic approach

Let's now examine a microservices-based architecture. Each microservice has its own presentation layer, business layer, and database layer. Microservices are aligned towards business capabilities. By doing so, changes to one microservice doesn't impact others.

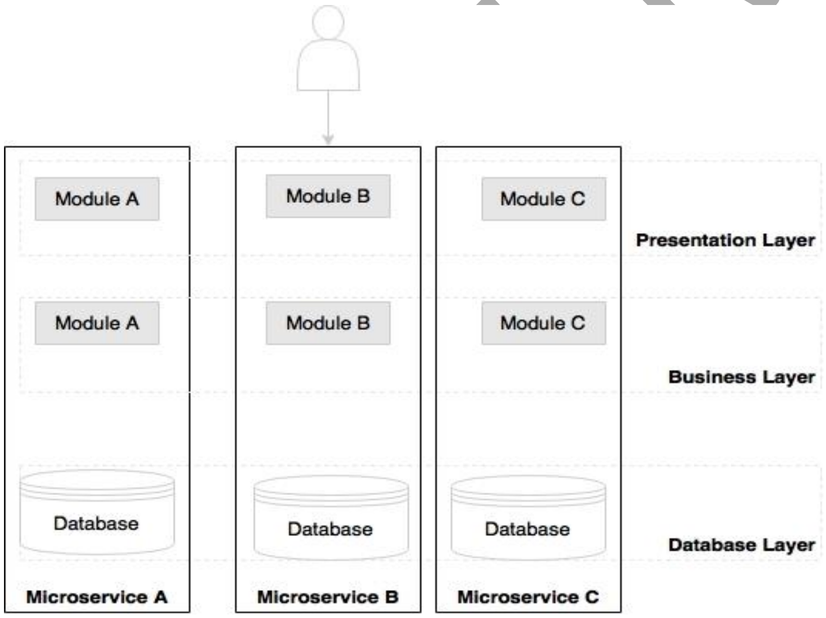


Figure: Microservices approach

What are micorservices?

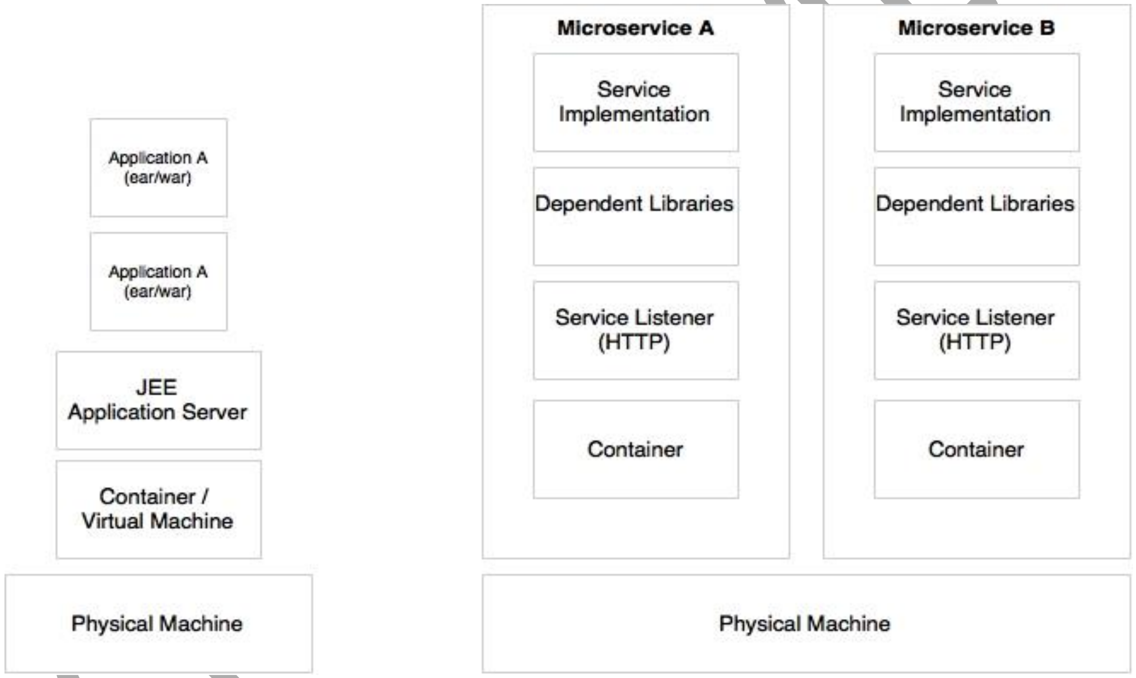
Ans) Microservices are autonomous, self-contained, loosely coupled, independently deployable and contains its own presentation layer, business layer, and database layer.

Principles of microservices

The below principles are a "**must have**" when designing and developing microservices:

1) Single microservice per single business responsibility.
2) Microservices are independently deployable. Hence they bundle all dependencies, including library dependencies, and execution environments such as web servers and containers, virtual machines and databases.

One of the major differences between Microservices and SOA is in their level of decomposition. While most SOA implementations provide service-level decomposition, microservices go further and decompose till execution environment.



In monolithic developments, we build a WAR or an EAR, then deploy it into a JEE application server, such as with JBoss, WebLogic, WebSphere, and so on. We may deploy multiple applications into the same JEE Server. In microservices approach, each microservice will be built as a **Fat Jar** using boot, which contains all dependencies including web servers and run as a standalone Java process.

Characteristics of microservices

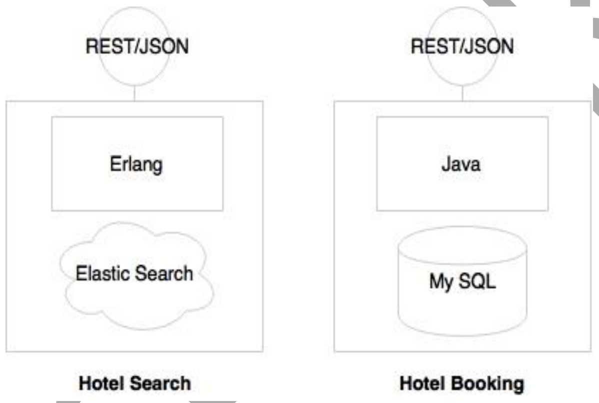
1) **Services are first class citizens**

   In the microservices architecture, there is **no more application development rather service development**. Microservices expose service endpoints as APIs and abstract all their realization details

i.e., the internal implementation logic, architecture, and technologies are completely hidden behind the service API.
Messaging (JMS / AMQP), HTTP, and REST are commonly used for interaction between microservices.
Microservices are reusable business services.
Well-designed microservices are stateless and share nothing with no shared state or conversational state maintained by the services.
Microservices are discoverable.

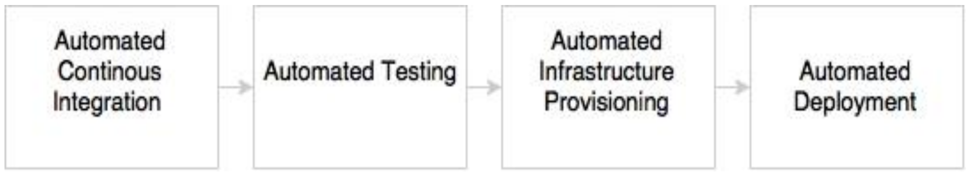
2) **Microservices are lightweight**
The microservices are aligned to a **single business capability**, so they perform only one function.
When selecting supporting technologies, such as web servers, we will have to ensure that they are also lightweight. For example, Jetty or Tomcat are better choices as servers for microservices compared to more complex traditional application servers such as WebLogic or WebSphere.
Preferred to use Docker containers instead of VMs to help us keep the infrastructure footprint as minimal as possible.

3) **Microservices with polyglot architecture**
Since Microservices are autonomous hence different services may use different technologies such as one service may be developed using java and another service may be developed using Scala, etc.



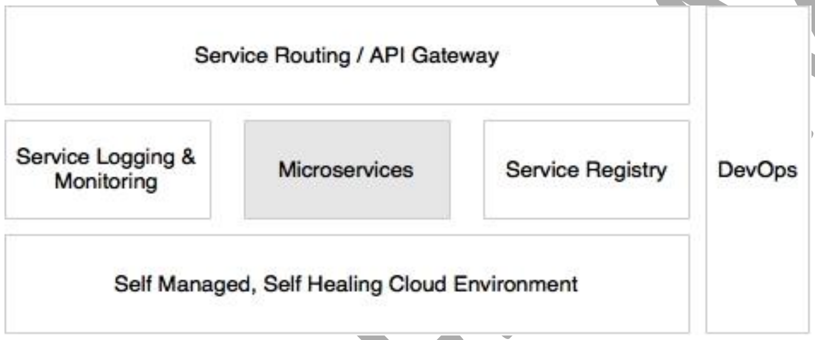
Each microservice has its own database, http container such as tomcat or jetty.

4) **Automation in a microservices environment**
As microservices break monolithic application into a number of smaller services, large enterprises may have many microservices. A large number of microservices are hard to manage until and unless automation is in place. Hence microservices should be automated from development to production: For example, automated builds, automated testing, automated deployment, and automated Infrasturcture provisioning.


Automated Continous Integration
Automated Testing
Automated Infrastructure Provisioning
Automated Deployment

The development phase is automated using version control tools such as Git together with **Continuous Integration** (**CI**) tools such as Jenkins.
The testing phase will be automated using testing tools such as Selenium.
Automated deployments are handled using DevOps.
Infrastructure provisioning is done through Cloud.

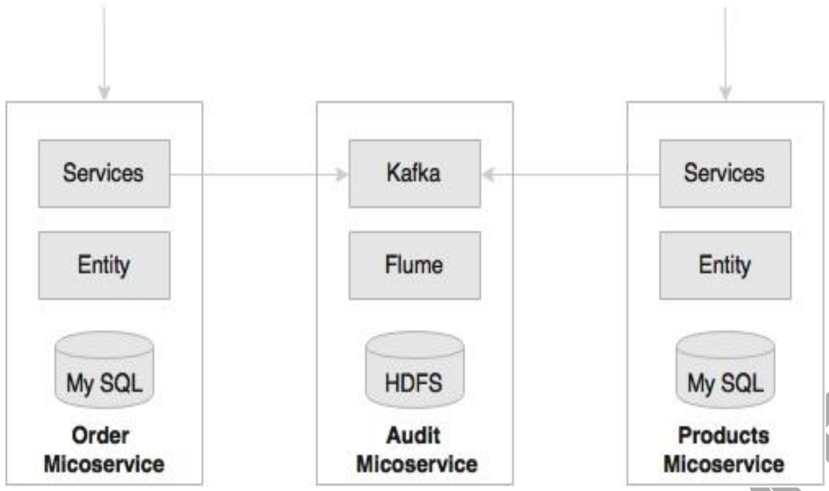
5) **Microservices with supporting ecosystem**
Microservices implementations have a supporting ecosystem including DevOps, Centralized log management, Service registry, API gateways, Service routing, Flow control mechanisms.


Service Routing / API Gateway
Service Logging & Monitoring
Microservices
Service Registry
DevOps
Self Managed, Self Healing Cloud Environment

6) Microservices are distributed and dynamic

Microservices benefits

1) **Supports polyglot architecture**
With microservices, architects and developers can choose fit for purpose architectures and technologies for each microservice i.e., each service can run with its own architecture or technology or different versions of technologies.



2) **Enabling experimentation and innovation**
With large monolithic applications, experimentation was not easy. With microservices, it is possible to write a small microservice to achieve the targeted functionality and plug it into the system in a reactive style.

3) **Selective scaling**
A monolithic application, packaged as a single WAR or an EAR, can only be scaled as a whole. In Microservices, **each service could be independently scaled up or down depending on scalability requirement**. As scalability can be selectively applied at each service, the cost of scaling is comparatively less with the microservices approach.

4) **Allowing substitution**
Microservices are self-contained, independent deployment modules enabling the substitution of one microservice with another similar microservice. Many large enterprises follow **buy-versus-build** policies to implement software systems. A common scenario is to build most of the functions in house and buy certain niche capabilities from specialists outside.

5) **Supporting Cloud**

6) **Enabling DevOps**
Microservices are one of the key enablers of DevOps. DevOps is widely adopted as a practice in many enterprises, primarily to increase the speed of delivery and agility. DevOps advocates having agile development, high-velocity release cycles, automatic testing, and automated deployment.

Relationship with SOA

SOA and Microservices follow similar concepts i.e., many service characteristics are common in both approaches.

Service-Oriented Architecture (SOA) is an architectural style that supports service orientation. Service orientation is a way of thinking in terms of services and service-based development and the outcomes of services.

A service: Is a logical representation of a repeatable business activity that has a specified outcome (e.g., check customer credit, provide weather data, and consolidate drilling reports)

1) It is self-contained.
2) It may be composed of other services.
3) It is a "black box" to consumers of the service."

We observed similar aspects in microservices as well. Then how microservices differ from SOA?

One of the major differences between Microservices and SOA is in their level of abstraction. While most SOA implementations provide service-level abstraction, Microservices go further and abstract the realization and execution environment i.e., in SOA development, we may deploy multiple services into the same JEE container. In the microservices approach, each microservice will be built as a **fat Jar**, embedding all dependencies including web containers and run as a standalone Java process.

In case of Legacy modernization, the services are built and deployed in the ESB layer connecting to backend systems using ESB adapters. In these cases, microservices are different from SOA.

Microservice use cases

A microservice will not solve all the architectural challenges of today's world. There is no hard-and-fast rule or rigid guideline on when to use microservices.

The first and the foremost activity is to do a test of the use case against the microservices benefits.

Let's discuss some commonly used scenarios that are suitable candidates for a microservices architecture:

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 are good candidates for microservices because these are independent stateless computing units that accept certain data, apply algorithms, and return the results.
3) Independent technical services such as the communication service, the encryption service, authentication services, and so on are also good candidates for microservices.
4) In many cases, we can build 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. These are normally reused across multiple channels and, hence, are good candidates for building them as microservices.

5) There could be micro or macro applications that serve a single purpose and performing a single responsibility.
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 could also be considered as potential candidates for the microservices architecture.
7) Applications that we could anticipate getting benefits from microservices such as polyglot requirements.

There are few scenarios in which we should consider avoiding microservices:

1) If the organization's policies are forced to use centrally managed heavyweight components such as ESB to host a business logic.
2) If the organization's culture, processes, and so on are based on the traditional waterfall delivery model, lengthy release cycles, manual deployments and cumbersome release processes, no infrastructure provisioning, and so on, then microservices may not be the right fit.

Microservices early adopters

Many organizations have already successfully embarked on their journey to the microservices world.

1) **Netflix**
   Netflix, an international on-demand media streaming company, is a pioneer in the microservices space. Netflix transformed their large pool of developers developing traditional monolithic code to smaller development teams producing microservices. At Netflix, engineers started with monolithic, went through the pain, and then broke the application into smaller units that are loosely coupled and aligned to the business capability.
2) **Amazon**
   The well-architected monolithic application was based on a tiered architecture with many modular components. However, all these components were tightly coupled. As a result, Amazon was not able to speed up their development cycle. Amazon then separated out the code as independent functional services, wrapped with web services, and eventually advanced to microservices.
3) **Twitter**
   When Twitter experienced growth in its user base, they went through an architecture-refactoring cycle. With this refactoring, Twitter moved away from a typical web application to an API-based event driven code. Twitter uses Scala and Java to develop microservices with polyglot persistence.
4) **Uber**
   When Uber expanded their business from one city to multiple cities, the challenges started. Uber then moved to microservice based architecture by breaking the system into smaller independent units. Each module was given to different teams and empowered them to choose their language, framework, and database. Uber has many microservices deployed in their ecosystem using REST.
   …

Building microservices with boot

Traditionally a war was explicitly created and deployed on a Tomcat server. But microservices need to develop services as executables, self-contained JAR files with an embedded HTTP listener (such as tomcat or jetty). Spring boot is a tool to develop such kinds of services i.e., Spring Boot enables microservices development by packaging all the required runtime dependencies in a **fat executable JAR file**.

2. DESIGNING MICROSERVICES

Microservices have gained enormous popularity in recent years. They have evolved as the preferred choice of architects, putting SOA into the backyards. While acknowledging the fact that microservices are a vehicle for developing scalable cloud native systems, **successful microservices need to be carefully designed** to avoid catastrophes. Hence number of factors are to be considered when designing microservices, as detailed in the following sections.

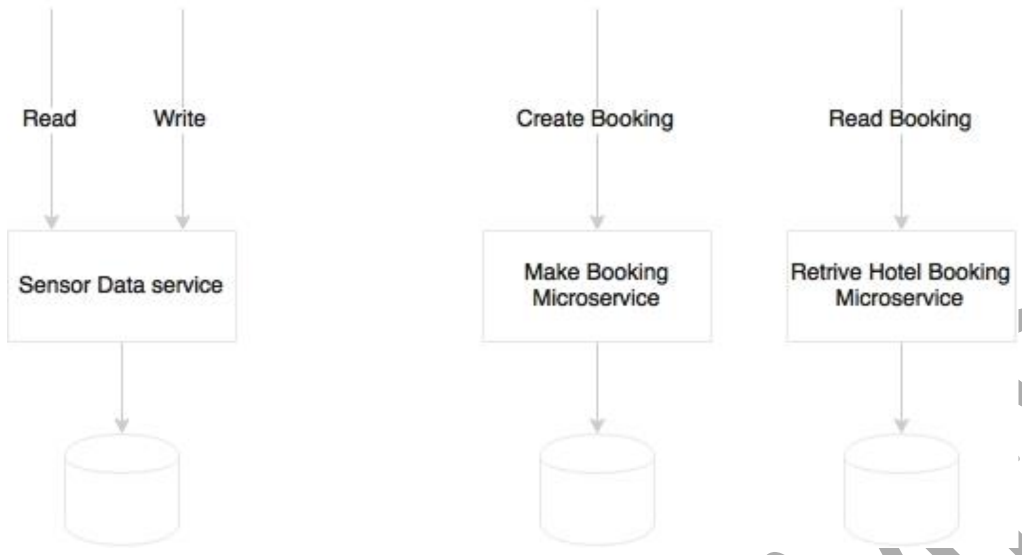
Identifying microservice boundaries

The following scenarios could help in defining microservice boundaries:

1) Autonomous functions : If the function under review is autonomous by nature, then it can be taken as a microservices boundary.
2) Size of deployable unit: A good microservice ensures that the size of its deployable units remains manageable.
3) Polyglot Architecture: If different requirements need different architectures, different technologies, etc then split them as separate Microservices.
4) Selective Scaling: All functions may not require the same level of scalability. Sometimes it may be appropriate to determine boundaries based on scalability requirements. For example, in the flight booking, the Search microservice has to scale considerably more than Booking microservice.
5) Small, Agile teams
6) Single Responsibility: One microservice per one business capability.
7) Replicability or Changeability: Microservice boundaries should be identified in such a way that each microservice is easily detachable from the overall system.
8) Coupling and Cohesion

Number Of Endpoints for a Microservice

The number of endpoints is not really a decision point. In some cases, there may be only one endpoint, whereas in some other cases, there could be more than one endpoint in a microservice.



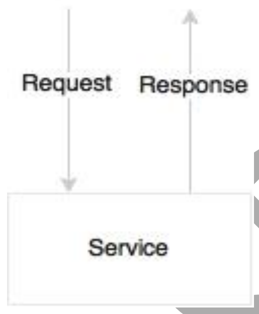
The Sensor data service has two logical end points: read and write

Communication styles

Communication between microservices can be designed either in **synchronous** or **asynchronous** styles.

Synchronous style

The following diagram shows an example of synchronous (request/response) style service:



In synchronous communication, there is no shared state or queue. When a caller requests a service, it passes the required information and **waits** for a response.

Advantages:

1) No messaging server overhead.
2) The error will be propagated back to the caller immediately.

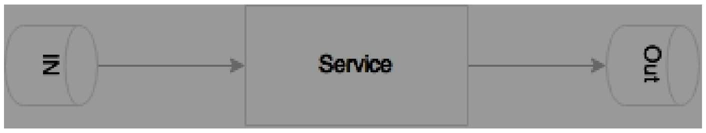
Dis advantages:

1) The caller has to wait until the requested process gets completed.

2) Adds hard dependencies between Microservices i.e., If one service in the chain fails, then the entire service chain will fail.

Asynchronous style

The following diagram is a service designed to accept an asynchronous message as input, and send the response asynchronously for others to consume:



The asynchronous style is based on **reactive** event loop semantics which **decouple** microservices.

Advantages:

1) Decouple Microservices
2) Higher level of scalability because of services are independent. Hence if there is a slowdown in one of the services, it will not impact the entire chain.

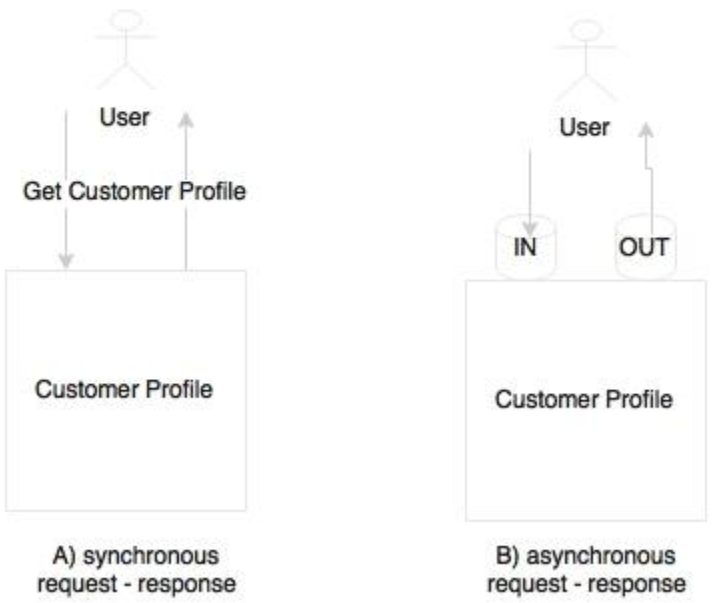
Dis advantages:

1) It has a dependency to an external messaging server.
2) It is complex to handle the fault tolerance of a messaging server.

How to decide which style to choose?

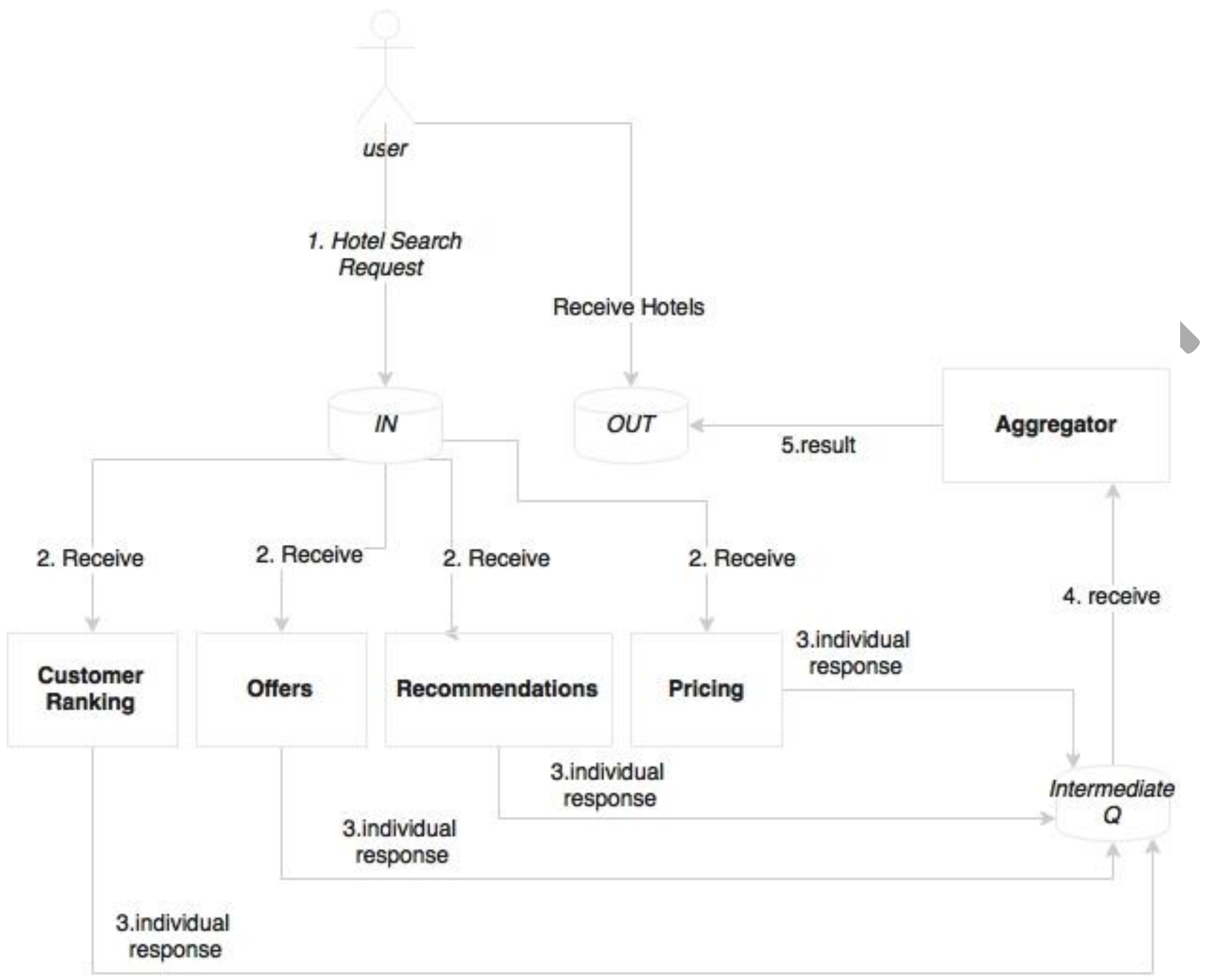
It is not possible to develop a system with just one approach. A combination of both approaches are required based on the use cases. In principle, the asynchronous approach is great for microservices. However, attempting to model everything as asynchronous leads to complex system designs.

How does the following example look in the context where an end user clicks on a UI to get profile details?


User
Get Customer Profile
Customer Profile
A) synchronous request - response
User
IN
OUT
Customer Profile
B) asynchronous request - response

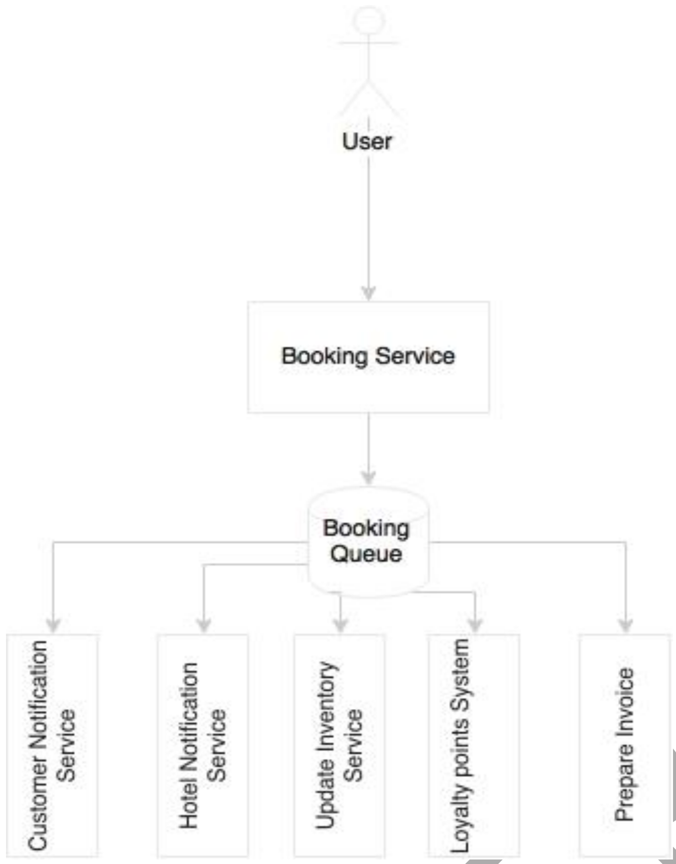
This is perfect scenario for synchronous communication. This can also be modeled in an asynchronous style by pushing a message to an input queue, wait and read response from the output queue. However, though we use asynchronous messaging, the user is still blocked for the entire duration of the query. Hence no advantage of using asynchronous style.

Another use case is user clicking on a UI to search hotels, which is depicted in the following diagram:



When the system receives this request, it calculates the customer ranking, gets offers based on the destination, gets recommendations based on customer preferences, and optimizes the prices based on customer values and revenue factors, and so on. In this case, we have an opportunity to do many of these activities in parallel so that we can aggregate all these results before presenting them to the customer. As shown in the preceding diagram, virtually any computational logic could be plugged in to the search pipeline listening to the **IN** queue. An effective approach in this case is to start with a synchronous request response, and refactor later to introduce an asynchronous style when there is value in doing that.

The following example shows a fully asynchronous style of service interactions:



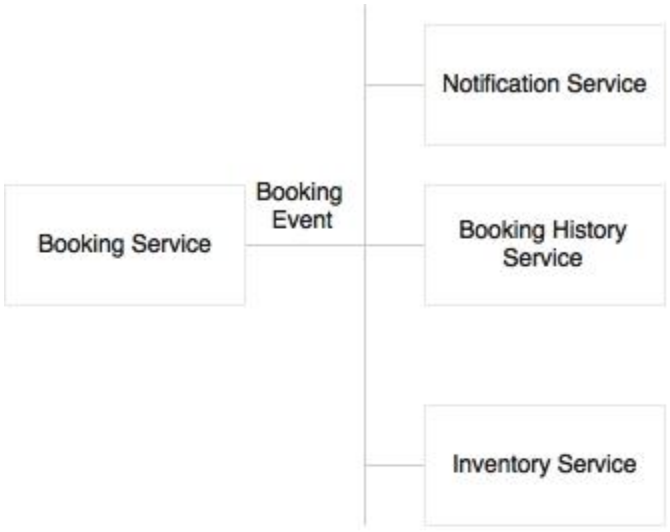
When booking is successful, it sends a message to the customer's e-mail address, sends a message to the hotel's booking system, updates the cached inventory, updates the loyalty points system, prepares an invoice, and perhaps more. Instead of pushing the user into a long wait state, a better approach is to break the service into pieces. Let the user wait till a booking record is created by the Booking Service. On successful completion, a booking event will be published, and return a confirmation message back to the user. Subsequently, all other activities will happen in parallel, asynchronously.

Conclusion:
In general, an asynchronous style is always better in the microservices world, but identifying the right pattern should be purely based on merits. If there are no merits in modeling a communication in an asynchronous style, then use the synchronous style till we find an appealing case.

Orchestration of Microservices

Composability (means controlling) is one of the service design principles. In the SOA world, ESBs are responsible for composing a set of fine-grained services i.e., In the SOA world, ESBs play the role of orchestration.

Microservices are autonomous. This means that all required components to complete their function should be within the service. This includes the database, orchestration of its internal services, state management, and so on. But in reality, microservices may need to talk with other microservices to fulfil their function.
The following approach is preferred to connect multiple microservices together:



Number Of VMs per MicroService

The one microservice can be deployed in one or multiple Virtual Machines (VMs) by replicating the deployment for scalability and availability.
Multiple Microservices can be deployed in one VM if the service is simple and the traffic volume is less.

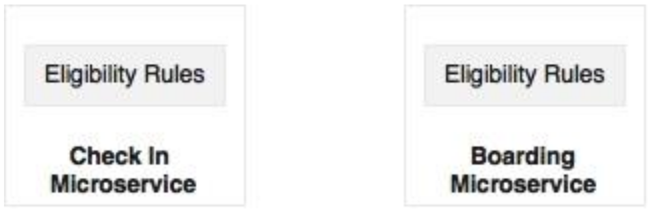
In case of cloud infrastructure, the developers need not to worry about where the services are running. Developers may not even think about capacity planning. Services will be deployed in a compute cloud. Based on the infrastructure availability and the nature of the service, the infrastructure self-manages deployments.

Can microservices share data stores?

In principle, microservices should abstract presentation, business logic, and data stores i.e., each microservice logically could use an independent database.
Shared data models, Shared schema, and shared tables are disasters when developing microservices.
If the services have only a few tables, it may not be worth investing a full instance of a database like Oracle instance. In such cases, schema level segregation is good enough to start with.

Shared Libraries

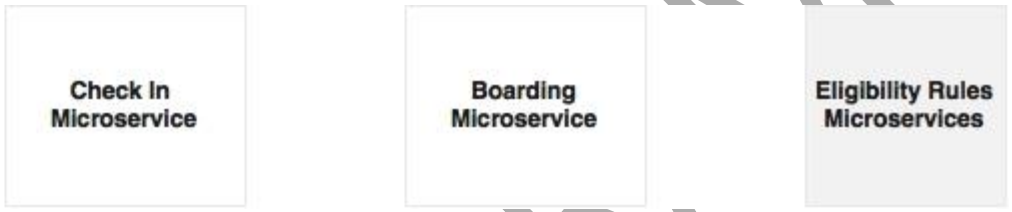
Sometimes code and libraries may be duplicated in order to adhere to autonomous and self-contained principle.



The eligibility for a flight upgrade will be checked at the time of check-in as well as when boarding. This was the trade-off between overheads in communication versus duplicating libraries in multiple services:

1) It may be easy to duplicate code or shared library but downside of this approach is that in case of a bug or an enhancement on the shared library, it has to be upgraded in more than one place.

2) An alternative option of developing the shared library as another microservice itself needs careful analysis. If it is not qualified as a microservice from the business capability point of view, then it may add more complexity than its usefulness.



3. MICROSERVICES CHALLENGES

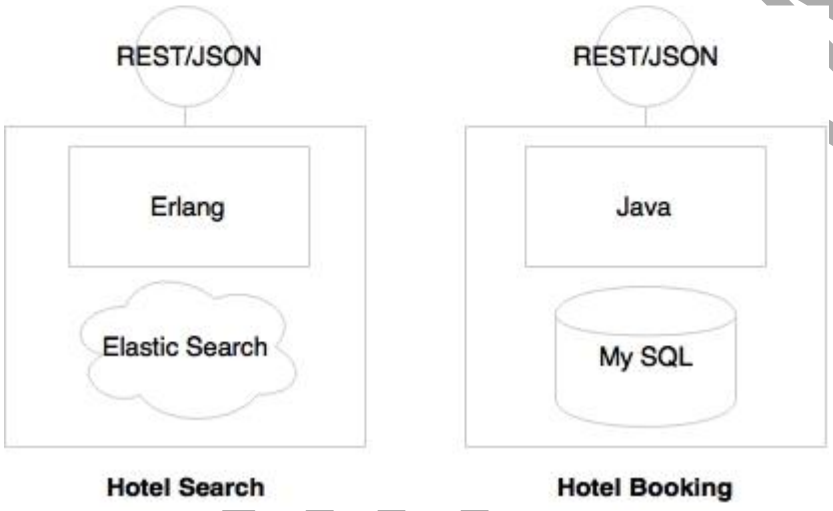
In this chapter, we will review some of the challenges with microservices, and how to address them for a successful microservice development.

Infrastructure provisioning

With many Microservices running, manual deployment could lead to significant operational overheads and the chances of errors are high.
To address this challenge, Microservices should use elastic cloud-like infrastructure which can automatically provision VMs or containers, automatically deploy applications, adjust traffic flows, replicate new version to all instances, and gracefully phase out older versions. The automation also takes care of scaling up elastically by adding containers or VMs on demand, and scaling down when the load falls below threshold.

Data Islands

Microservices use their own local transactional store, which is used for their own transactional purposes.



In the preceding diagram, Hotel search is expected to have high transaction volume hence preferred to use Elasticsearch. The Hotel booking needs more ACID transactions hence preferred to use MySQL. That means different Microservices may use different types of databases which leads data islands.

What if we want to do an analysis by combining data from two data stores?
In order to satisfy this requirement, a data warehouse (traditional) or a **data lake** is required. The tools like Spring Cloud Data Flow, Kafka, Flume, etc are useful.

Logging and monitoring

Since each microservice is deployed independently, they emit separate log files. This makes it extremely difficult to debug and understand the behavior of the services through logs. Hence we need **centralized logging** mechanism which can be achieved using Graylog, Splunk, ELK stack, AWS CloudTrail, Google Cloud Logging, Spring Boot's Loggly, Logstash, Kibana, Elastic search, SaaS, etc.

Organization culture

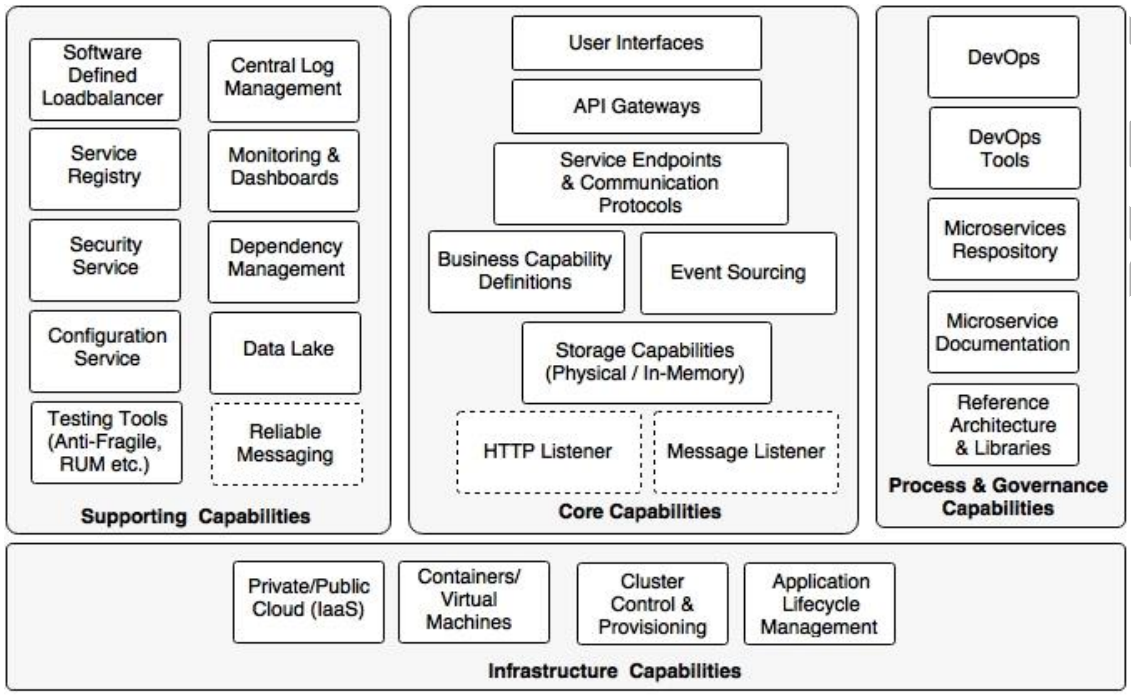
One of the biggest challenges in microservices implementation is the organization culture.

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.

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.

4. THE MICROSERVICES CAPABILITY MODEL

We will review a capability model for microservices based on the design guidelines, challenges, common patterns and solutions described so far.


Software Defined Loadbalancer
Central Log Management
Service Registry
Monitoring & Dashboards
Security Service
Dependency Management
Configuration Service
Data Lake
Testing Tools (Anti-Fragile, RUM etc.)
Reliable Messaging
Supporting Capabilities
User Interfaces
API Gateways
Service Endpoints & Communication Protocols
Business Capability Definitions
Event Sourcing
Storage Capabilities (Physical / In-Memory)
HTTP Listener
Message Listener
Core Capabilities
DevOps
DevOps Tools
Microservices Respository
Microservice Documentation
Reference Architecture & Libraries
Process & Governance Capabilities
Private/Public Cloud (IaaS)
Containers/ Virtual Machines
Cluster Control & Provisioning
Application Lifecycle Management
Infrastructure Capabilities

The capability model is broadly classified into four areas:

1. **Core capabilities**: These are part of the microservices themselves
2. **Supporting capabilities**: These are software solutions supporting core microservice implementations
3. **Infrastructure capabilities**: These are infrastructure level expectations for a successful microservices implementation
4. **Governance capabilities**: These are more of process, people, and reference information

Core capabilities

The core capabilities are explained as follows:

- **Service listeners** (HTTP/Message): If microservices are enabled for a HTTP-based service endpoint, then the **HTTP listener is embedded within the microservices**, thereby eliminating the need to have any external application server requirement.
  If the microservice is based on asynchronous communication, then instead of an HTTP listener, a **message listener** is started. Spring Boot and Spring Cloud Streams provide this capability.

- **Storage capability**: The microservices have some kind of storage mechanisms to store state or transactional data pertaining to the business capability. The storage could be either a physical storage

(RDBMS such as MySQL; NoSQL such as Hadoop, Cassandra, Neo 4J, Elasticsearch, and so on), or it could be an in-memory store (cache like Ehcache, Redis, data grids like Hazelcast, Infinispan, and so on)

- **Business capability definition:** This is the core of microservices, where the business logic is implemented. This could be implemented in any applicable language such as Java, Scala, Conjure, Erlang, and so on. All required business logic to fulfill the function will be embedded within the microservices themselves.
- **Event sourcing:** Microservices send out state changes to the external world without really worrying about the targeted consumers of these events. These events could be consumed by other microservices, audit services, replication services, or external applications, and the like. This allows other microservices and applications to respond to state changes.
- **Service endpoints and communication protocols:** These define the APIs for external consumers to consume. These could be synchronous endpoints or asynchronous endpoints.
- **API gateway:** The API gateway provides a level of indirection by either proxying service endpoints or composing multiple service endpoints. There are many API gateways available in the market. Spring Cloud Zuul, Mashery, Apigee, and 3scale are some examples of the API gateway providers.
- **User interfaces:** Generally, user interfaces are also part of microservices for users to interact with the business capabilities realized by the microservices. These could be implemented in any technology.

Infrastructure capabilities

Certain infrastructure capabilities are required for a successful deployment, and managing large scale microservices. When deploying microservices at scale, not having proper infrastructure capabilities can be challenging, and can lead to failures:

- **Cloud:** Microservices implementation is difficult in a traditional data center environment with long lead times to provision infrastructures. Even a large number of infrastructures dedicated per microservice may not be very cost effective. Managing them internally in a data center may increase the cost of ownership and cost of operations. A cloud-like infrastructure is better for microservices deployment.
- **Containers or virtual machines:** Managing large physical machines is not cost effective, and they are also hard to manage. Virtualization is adopted by many organizations because of its ability to provide optimal use of physical resources. It also provides resource isolation. It also reduces the overheads in managing large physical infrastructure components. Containers are the next generation of virtual machines. VMWare, Citrix, and so on provide virtual machine technologies. Docker, Drawbridge, Rocket, and LXD are some of the containerizer technologies.
- **Cluster control and provisioning**: Once we have a large number of containers or virtual machines, it is hard to manage and maintain them automatically. Cluster control tools provide a uniform operating environment on top of the containers, and share the available capacity across multiple services. Apache Mesos and Kubernetes are examples of cluster control systems.
- **Application lifecycle management**: Application lifecycle management tools help to invoke applications when a new container is launched, or kill the application when the container shuts down. Application life cycle management allows for script application deployments and releases. It automatically detects

failure scenario, and responds to those failures thereby ensuring the availability of the application. This works in conjunction with the cluster control software. Marathon partially addresses this capability.

Supporting capabilities

Supporting capabilities are not directly linked to microservices, but they are essential for large scale microservices development:

- **Software defined load balancer**: The load balancer should be smart enough to understand the changes in the deployment topology, and respond accordingly. This moves away from the traditional approach of configuring static IP addresses, domain aliases, or cluster addresses in the load balancer. When new servers are added to the environment, it should automatically detect this, and include them in the logical cluster by avoiding any manual interactions. Similarly, if a service instance is unavailable, it should take it out from the load balancer. A combination of *Ribbon*, Eureka, and Zuul provide this capability in Spring Cloud Netflix.

- **Central log management**: A capability is required to centralize all logs emitted by service instances with the correlation IDs. This helps in debugging, identifying performance bottlenecks, and predictive analysis. The result of this is fed back into the life cycle manager to take corrective actions.

- **Service registry**: A service registry provides a runtime environment for services to automatically publish their availability at runtime. A registry will be a good source of information to understand the services topology at any point. *Eureka* from Spring Cloud, Zookeeper, and Etcd are some of the service registry tools available.

- **Security service**: A distributed microservices ecosystem requires a central server for managing service security. This includes service authentication and token services. OAuth2-based services are widely used for microservices security. Spring Security and *Spring Security OAuth* are good candidates for building this capability.

- **Service configuration**: All service configurations should be externalized as discussed in the Twelve-Factor application principles. A central service for all configurations is a good choice. *Spring Cloud Config server*, and Archaius are out-of-the-box configuration servers.

- **Testing tools (anti-fragile, RUM, and so on)**: Netflix uses Simian Army for anti-fragile testing. Matured services need consistent challenges to see the reliability of the services, and how good fallback mechanisms are. Simian Army components create various error scenarios to explore the behavior of the system under failure scenarios.

- **Monitoring and dashboards**: Microservices also require a strong monitoring mechanism. This is not just at the infrastructure-level monitoring but also at the service level. Spring Cloud Netflix Turbine, Hysterix Dashboard, and the like provide service level information. End-to-end monitoring tools like AppDynamic, New Relic, Dynatrace, and other tools like statd, Sensu, and Spigo could add value to microservices monitoring.

- **Dependency and CI management**: We also need tools to discover runtime topologies, service dependencies, and to manage configurable items. A graph-based CMDB is the most obvious tool to manage these scenarios.

- **Data lake**: We need a mechanism to combine data stored in different microservices, and perform near real-time analytics. A data lake is a good choice for achieving this. Data ingestion tools like Spring Cloud Data Flow, Flume, and Kafka are used to consume data. HDFS, Cassandra, and the like are used for storing data.

- **Reliable messaging**: If the communication is asynchronous, we may need a reliable messaging infrastructure service such as RabbitMQ or any other reliable messaging service. Cloud messaging or messaging as a service is a popular choice in Internet scale message-based service endpoints.

Process and governance capabilities

The last piece in the puzzle is the process and governance capabilities that are required for microservices:

- **DevOps**: The key to successful implementation of microservices is to adopt DevOps. DevOps compliment microservices development by supporting Agile development, high velocity delivery, automation, and better change management.

- **DevOps tools**: DevOps tools for Agile development, continuous integration, continuous delivery, and continuous deployment are essential for successful delivery of microservices. A lot of emphasis is required on automated functioning, real user testing, synthetic testing, integration, release, and performance testing.

- **Microservices repository**: A microservices repository is where the versioned binaries of microservices are placed. These could be a simple Nexus repository or a container repository such as a Docker registry.

- **Microservice documentation**: It is important to have all microservices properly documented. Swagger or API Blueprint are helpful in achieving good microservices documentation.

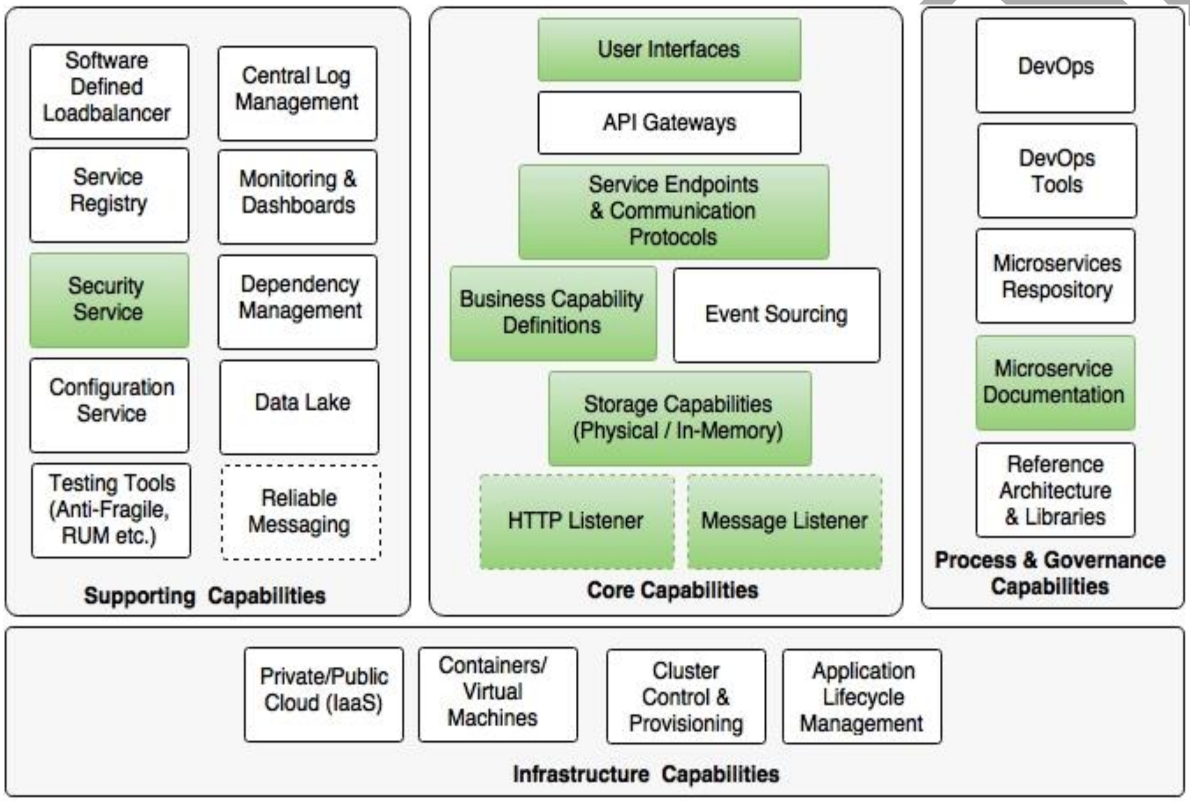
- **Reference architecture and libraries**: The reference architecture provides a blueprint at the organization level to ensure that the services are developed according to certain standards and guidelines in a consistent manner. Many of these could then be translated to a number of reusable libraries that enforce service development philosophies.

5. MICROSERVICES EVOLUTION – A CASE STUDY

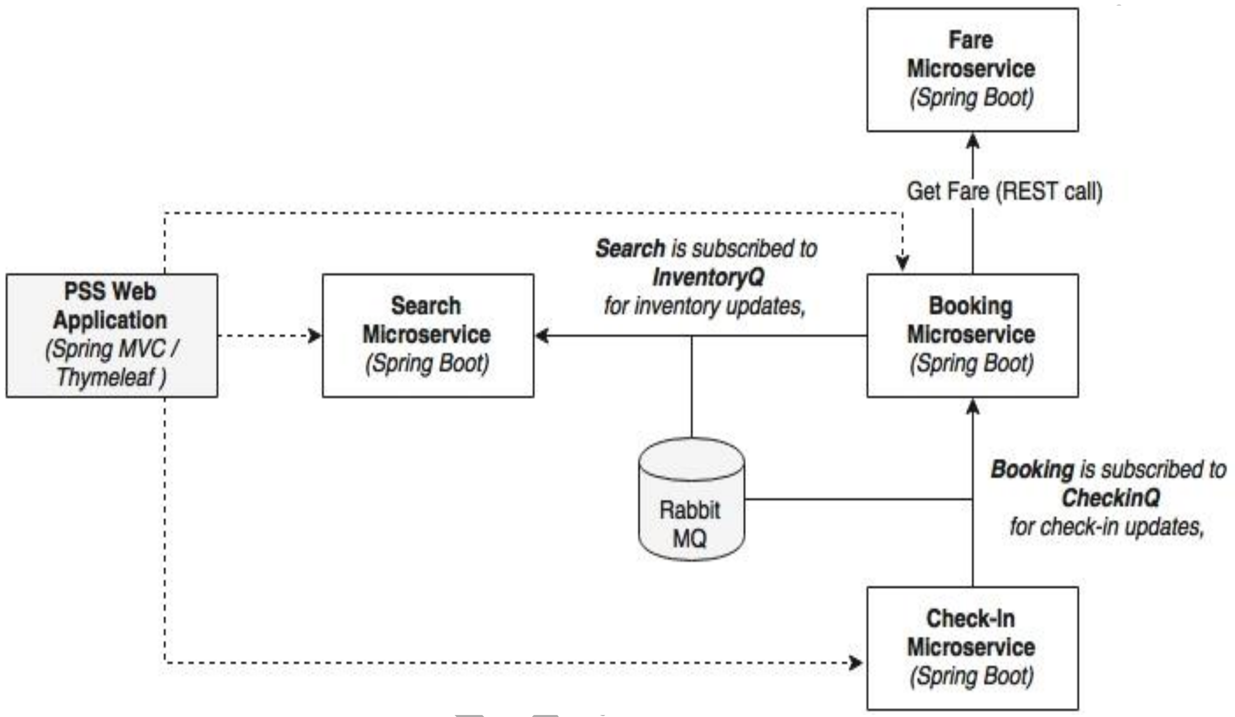
We will discuss BrownField Airline and their journey from a monolithic **Passenger Sales and Service** (PSS) application to a next generation microservices architecture by adhering to the principles and practices that were discussed before.

Reviewing the microservices capability model

In this chapter, we will explore the following microservices capabilities highlighted in green color from the microservices capability model discussed before.


Software Defined Loadbalancer
Central Log Management
Service Registry
Monitoring & Dashboards
Security Service
Dependency Management
Configuration Service
Data Lake
Testing Tools (Anti-Fragile, RUM etc.)
Reliable Messaging
Supporting Capabilities
User Interfaces
API Gateways
Service Endpoints & Communication Protocols
Business Capability Definitions
Event Sourcing
Storage Capabilities (Physical / In-Memory)
HTTP Listener
Message Listener
Core Capabilities
DevOps
DevOps Tools
Microservices Respository
Microservice Documentation
Reference Architecture & Libraries
Process & Governance Capabilities
Private/Public Cloud (IaaS)
Containers/ Virtual Machines
Cluster Control & Provisioning
Application Lifecycle Management
Infrastructure Capabilities

We are about to implement four microservices such as Fare, Search, Booking, and Check-in. In order to test the application, there is a website application developed using Spring MVC with Thymeleaf templates (needed for html pages). The asynchronous messaging is implemented with the help of RabbitMQ. In this implementation, the **Oracle database is used with separate schema for each Microservice**.
The code in this section demonstrates all the capabilities highlighted in green color above.


Fare Microservice (Spring Boot)
Get Fare (REST call)
Search is subscribed to InventoryQ for inventory updates,
PSS Web Application (Spring MVC / Thymeleaf)
Search Microservice (Spring Boot)
Booking Microservice (Spring Boot)
Rabbit MQ
Booking is subscribed to CheckinQ for check-in updates,
Check-In Microservice (Spring Boot)

The following steps are used to setup PSS Microservices project:

1) Create tablespaces, schemas, tables, sequences and insert data by referring 'Documents/Misc/ Airline_PSS_Schema.doc' file.
2) Start STS (Spring Tool Suite) and select '**MicroservicesWorkspace**' from the backup.
   Note: Download STS from https://spring.io/tools/sts/all
3) Start FaresFlightTickets by right click and Run as **Spring Boot App**.
4) Install **RabbitMQ** Server from Softwares folder. After installation check service status in start-> run -> services.msc
   Observation: Status: Running, Startup type: Automatic
   Note: The pre-requisite for RabbitMQ is **Erlang**. (Install OTP_win64_19.3.exe from Softwares folder)
5) Start SearchFlightTickets by right click and Run as **Spring Boot App**.
6) Start BookingFlightTickets by right click and Run as Spring Boot App.
7) Start CheckInCustomers by right click and Run as Spring Boot App.
8) Start FlightWebSite by right click and Run as Spring Boot App.

Each service has multiple packages and their purposes are explained as follows:

1. The entity package contains the JPA entity classes for mapping to the database tables.
2. The repository package contains repository classes, which are based on Spring Data JPA.
3. The component package hosts all the service components where the business logic is implemented.
4. The controller package hosts the **REST endpoints** and the **Messaging endpoints**. Controller classes internally utilize the component classes for execution.
5. The root package (com.brownfield.pss.fares) contains the default Spring Boot application.

The below table contains service endpoints and communication styles:

Microservice Name	REST endpoints (synchronous)	Messaging Endpoints (asynchronous)	Used By
FareFlightTickets	http://localhost:8081/fares/get		Booking microservice
SearchFlightTickets	http://localhost:8090/search/get		Website
SearchFlightTickets		@RabbitListener(queues = "**SearchQ**")	Search microservice itself subscribed to **SearchQ** for inventory updates.
BookingFlightTickets	http://localhost:8060/booking/create		Website
BookingFlightTickets	http://localhost:8060/booking/get/{id}		Checkin, website
BookingFlightTickets		template.convertAndSend("**SearchQ**", message);	Search Microservice
BookingFlightTickets		@RabbitListener(queues = "**CheckINQ**")	Booking service subscribed to **CheckINQ** for check-in updates.
CheckInCustomers	http://localhost:8070/checkin/create		Website
CheckInCustomers	http://localhost:8070/checkin/get/{id}		Not used
CheckInCustomers		template.convertAndSend("**CheckINQ**", message);	Booking Microservice

We have accomplished the following items in our microservice implementation so far:

1. Each microservice exposes a set of REST/JSON endpoints for accessing business capabilities
2. Each microservice implements certain business functions using the Spring framework.
3. Each microservice has its own schema in Oracle database.
4. Microservices are built with Spring Boot, which has an embedded Tomcat server as the HTTP listener.
5. RabbitMQ is used as an external messaging service. Search, Booking, and Check-in interact with each other through asynchronous messaging.

6. An OAuth2-based security mechanism is developed to protect the Microservices.

Kindly refer http://blog.arungupta.me/microservice-design-patterns/ for Microservices Design Patterns.

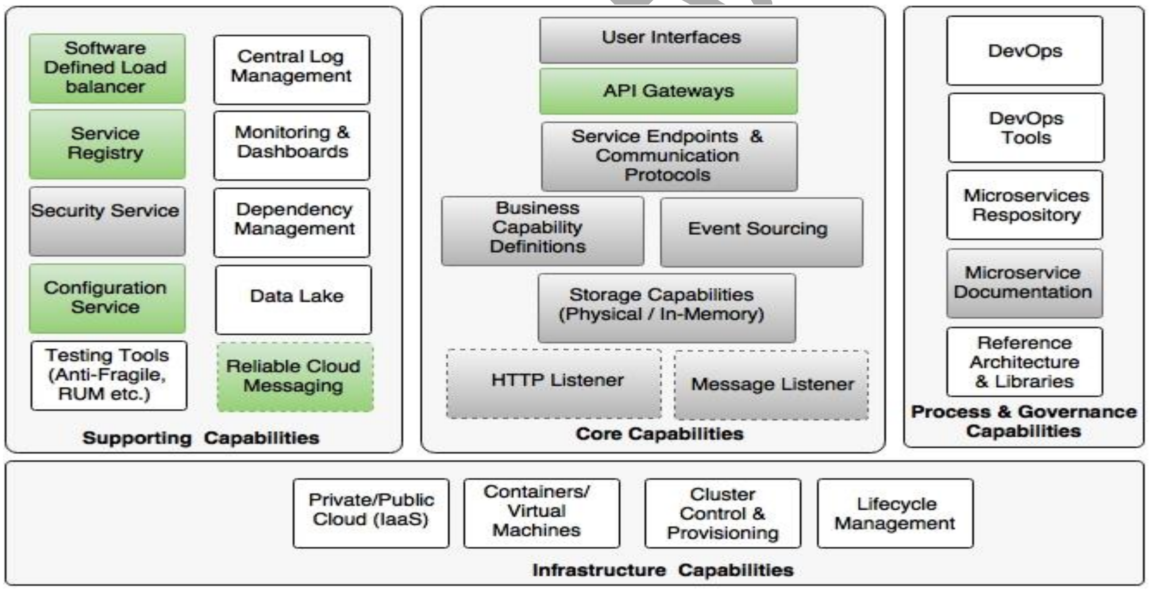
SPRING CLOUD

Spring Cloud provides capabilities such as **centralized configuration**, **load balancing**, **service registry**, monitoring, service gateway, and so on.

Cloud computing promises many benefits, such as cost advantage, speed, agility, flexibility, and elasticity. There are many cloud providers such as Amazon (AWS), Microsoft (Azure), Google (Google cloud), Pivotal (Cloud Foundry), etc.
Manual deployment could severely challenge the microservices rollouts. With many server instances running, this could lead to significant operational overheads. Moreover, the chances of errors are high in this manual approach.

Microservices require a supporting elastic cloud-like infrastructure which can automatically provision VMs or containers, automatically deploy applications, adjust traffic flows, replicate new version to all instances, and gracefully phase out older versions. The automation also takes care of scaling up elastically by adding containers or VMs on demand, and scaling down when the load falls below threshold.

The spring cloud related capabilities such as Load balancer, Service Registry, Configuration service, Cloud messaging, API Gateways, etc are highlighted in green color.


Software Defined Load balancer
Service Registry
Security Service
Configuration Service
Testing Tools (Anti-Fragile, RUM etc.)
Central Log Management
Monitoring & Dashboards
Dependency Management
Data Lake
Reliable Cloud Messaging
Supporting Capabilities
User Interfaces
API Gateways
Service Endpoints & Communication Protocols
Business Capability Definitions
Event Sourcing
Storage Capabilities (Physical / In-Memory)
HTTP Listener
Message Listener
Core Capabilities
DevOps
DevOps Tools
Microservices Respository
Microservice Documentation
Reference Architecture & Libraries
Process & Governance Capabilities
Private/Public Cloud (IaaS)
Containers/ Virtual Machines
Cluster Control & Provisioning
Lifecycle Management
Infrastructure Capabilities

Spring Cloud by itself is not a cloud solution. Rather, it provides a number of capabilities that are essential when developing applications targeting cloud deployments that adhere to the Twelve-Factor application principles. The cloud-ready solutions that are developed using Spring Cloud are also portable across many cloud providers such as **Cloud Foundry**, **AWS**, and so on.

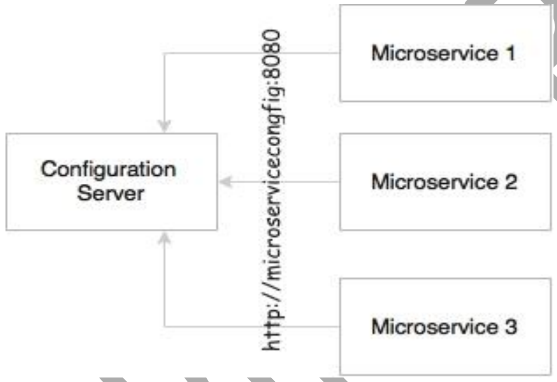
Twelve-Factor apps

Twelve-Factor Apps defines a set of principles of developing applications targeting the cloud.
Many organizations prefer to lift and shift their applications to the cloud. **The application (in our case microservice) has to follow and check Twelve-factor rules before moving to cloud** i.e., in order to run microservices seamlessly across multiple cloud providers, it is important to follow Twelve-factor rules while developing cloud native microservices.

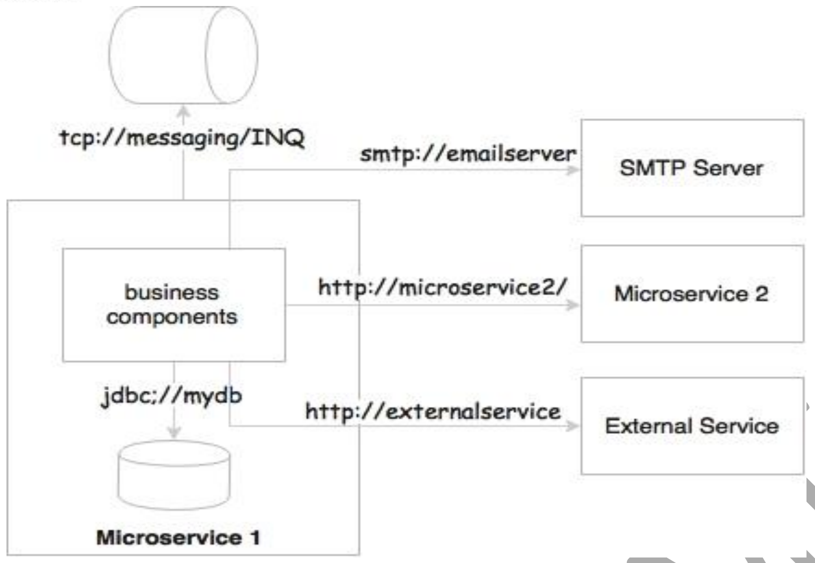
1) **Single code base**
   Each microservice has its own code base. Code is typically managed in a source control system such as Git, Subversion, etc.

2) **Bundling Dependencies**
   Each microservice should bundle all the required dependencies and execution libraries such as the HTTP listener and so on in the final executable bundle.

3) **Externalizing Configurations**
   This principle advises the externalization of all configuration parameters from the code. The microservices configuration parameters should be loaded from an external server.



4) **Backing Services are addressable**
   All backing services should be accessible through an addressable URL. All services need to talk to some external resources during the life cycle of their execution. For example, they could be listening or sending messages to a messaging system, sending an e-mail, persisting data to database, and so on. All these services should be reachable through a URL.



Microservices either talk to a messaging system to send or receive messages OR they could accept or send messages to other service APIs using REST/JSON over HTTP.

5) **Isolation between build, release, and run**
This principle advocates a strong isolation among the build, release, and run stages.
In microservices, the **build** will create executable fat JAR files, including the service runtime such as an HTTP listener.
During the **release phase**, these executables will be combined with release configurations such as production URLs and so on and create a release version, most probably as a container similar to Docker. In the **run stage**, these containers will be deployed on production via a container scheduler.

6) **Stateless, shared nothing processes**
This principle suggests that processes should be stateless and share nothing. If the application is stateless, then it can be scaled out easily.
All microservices should be designed as stateless functions. If there is any requirement to store a state, it should be done with a backing database or in an in-memory cache.

7) **Exposing services through port bindings**
A Twelve-Factor application is expected to be self-contained. Traditionally, applications are deployed to a server: a web server or an application server such as Apache Tomcat or JBoss. **A Twelve-Factor application does not rely on an external web server**. HTTP listeners such as Tomcat or Jetty have to be embedded in the service itself.

Port binding is one of the fundamental requirements for microservices to be autonomous and self-contained. Microservices embed service listeners as a part of the service itself.

8) **Concurrency to scale out**
In the microservices, services are designed to scale out. The services can be elastically scaled or shrunk based on the traffic flow. Further to this, microservices may make use of parallel processing and concurrency frameworks to further speed up or scale up the transaction processing.

9) **Disposability with minimal overhead**
This principle advocates building applications with minimal startup and shutdown times with graceful shutdown support.
In the microservices, in order to achieve full automation, it is extremely important to keep the size of the application as thin as possible, with minimal startup and shutdown time.

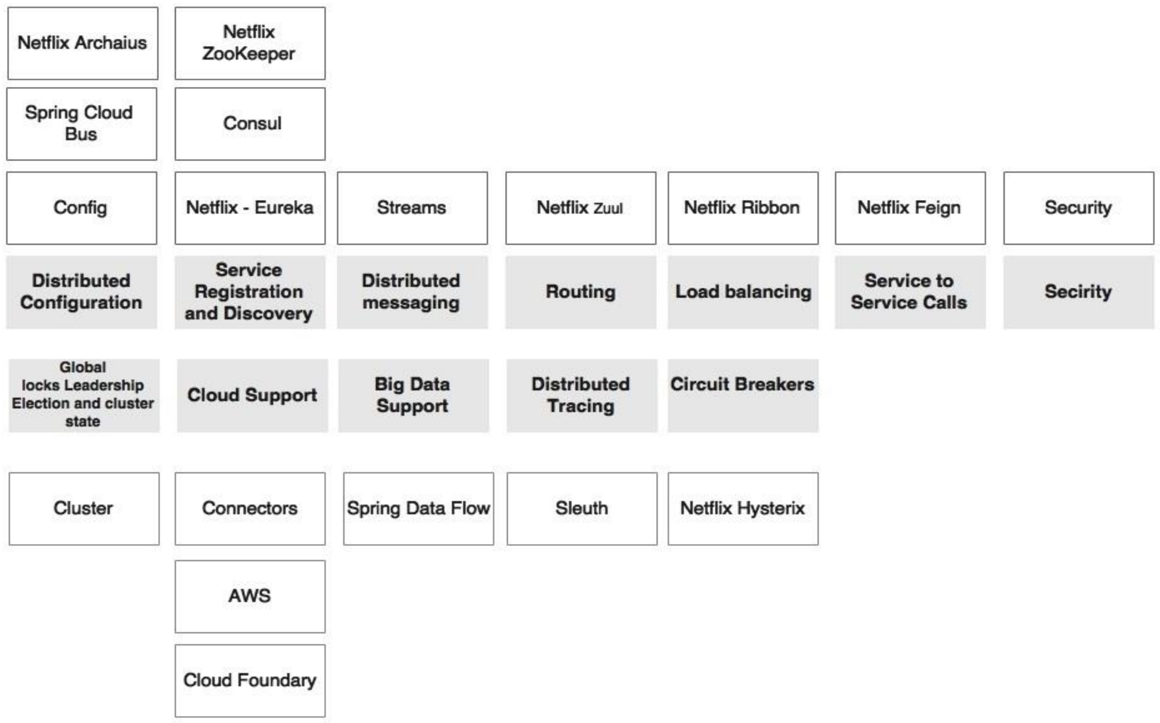
10) **Development and production parity**
This principle states the importance of keeping development and production environments as identical as possible. In a development environment, we tend to run all of them on a single machine, whereas in production, we will facilitate independent machines to run each of these processes. This is primarily to manage the cost of infrastructure. The downside is that if production fails, there is no identical environment to re-produce and fix the issues.
Not only is this principle valid for microservices, but it is also applicable to any application development.

11) **Externalizing logs**
A Twelve-Factor applicaion never attempts to store or ship log files. In a cloud, it is better to avoid local I/Os. If the I/Os are not fast enough in a given infrastructure, it could create a bottleneck. The solution to this is to use a centralized logging framework. Splunk, Greylog, Logstash, Logplex, and Loggly are some examples of log shipping and analysis tools.
In a microservices ecosystem, this is very important as we are breaking a system into a number of smaller services, which could result in decentralized logging. If they store logs in a local storage, it would be extremely difficult to correlate logs between services.

12) **Package admin processes**
Apart from application services, most applications provide admin tasks as well. This principle advises to use the same release bundle as well as an identical environment for both application services and admin tasks. Admin code should also be packaged along with the application code.

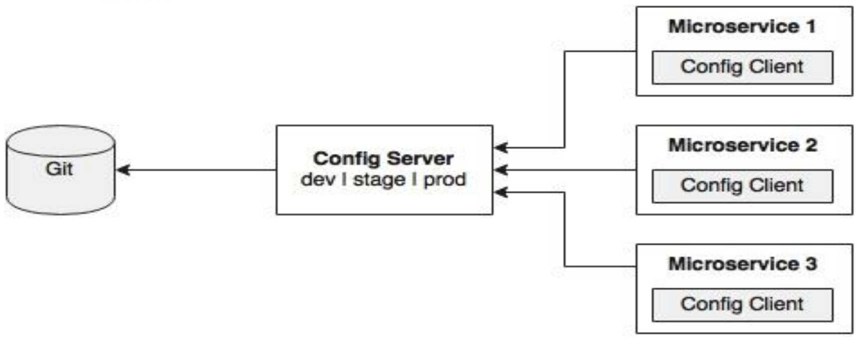
Components of Spring Cloud

Each Spring Cloud component specifically addresses certain distributed system capabilities. The grayed-out boxes show the capabilities, and the boxes placed on top of these capabilities showcase the Spring Cloud subprojects addressing these capabilities:



Spring Cloud Config

In Spring Boot applications, all configuration parameters were read from a property file packaged inside the project, either application.properties or application.yaml. This approach is good, since all properties are moved out of code to a property file. However, when microservices are moved from one environment to another, these properties need to undergo changes, which require an application re-build. This is violation of one of the Twelve-Factor application principles, which advocate one-time build and moving of the binaries across environments.
Hence it is always recommended to **externalize and centralize** microservice configuration parameters using **Spring Cloud Config server**.
The Spring Config server stores properties in a version-controlled repository such as Git or SVN.
The Spring Cloud Config server architecture is shown in the following diagram:



As shown in the preceding diagram, the **Config client** embedded in the Spring Boot microservices does a configuration lookup from a central configuration server. The configuration properties may be application related (such as trade limit per day) or infrastructure related properties (urls, credentials, etc).

Unlike Spring Boot, **Spring Cloud uses a bootstrap context**, which is a parent context of the main application. **Bootstrap context is responsible for loading configuration properties from the Config server**. The bootstrap context looks for **bootstrap.properties** or bootstrap.yaml for loading initial configuration properties. Hence rename application.properties as bootstrap.properties.

Setting up the Config server

The following steps are used to setting up config server:

1) Create a new **Spring Starter Project** named **'ConfigServer'**, and select **Config Server** and **Actuator**.

2) Download and install **Git**. Make sure that 'D:\Program Files\Git\cmd' added to path.
   Create 'config-repo' folder in windows home [${user.home}]. Navigate to ${user.home}/config-repo
   ```
   C:\Users\Ramesh-PC\config-repo>git init .
   C:\Users\Ramesh-PC\config-repo>echo message : helloworld > application.properties
   git add -A .
   git commit -m "Adding sample application.properties"
   ```
   Note: This code snippet creates a new Git repository on the local filesystem. A property file named application.properties with a message property and value helloworld is also created.

3) Expand ConfigServer project, goto src/main/resources folder, rename application.properties file to **bootstrap.properties**. Change the configuration in the config server to use the Git repository created in the previous step. For this, add following properties in bootstrap.properties file:
   ```
   server.port=8888
   spring.cloud.config.server.git.uri: file://${user.home}/config-repo
   ```

Note: Port 8888 is the default port for the Config server. Even without configuring server.port, the Config server will bind to 8888.

4) Add @**EnableConfigServer** in Application.java file.

5) Run Config Server by right-click on the project and Run as **Spring Boot App**.

6) Visit **`http://localhost:8888/env`** to see whether the server is running. If everything is fine, this will list all environment configurations. **Note that `/env` is an actuator endpoint**.
Note: Add **management.security.enabled=false** in bootstrap.properties file to avoid (type=Unauthorized, status=401)
Note: Use 'http://www.jsoneditoronline.org/ for formatting JSON message.

7) Also check Config server URL "http://localhost:8888/application/default/master" to see the properties specific to `application.properties`, which were added in the earlier step.
The first element in the URL is the application name. It is a logical name given to the application, using the **`spring.application.name`** property in `bootstrap.properties` of the Spring Boot application. The Config server will use the name to resolve and pick up appropriate properties from the Config server repository. The application name is also sometimes referred to as service ID. If there is an application with the name **`search-service`**, then there should be a **`search-service.properties`** in the configuration repository to store all the properties related to that application.
The second part of the URL represents the profile. The default profile is named `default`.
The last part of the URL is the label, and is named master by default. The label is an optional Git label that can be used, if required.

Accessing the Config Server from clients

In this section, the Search microservice will be modified to use the Config server. The Search microservice will act as a Config client.

1) Add the **Spring Cloud Config dependency** and the actuator (if the actuator is not already in place) to the `pom.xml` file.

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

Since we are modifying the Spring Boot Search microservice from the earlier project, we will have to add the following to include the Spring Cloud dependencies.

```
<dependencyManagement>
        <dependencies>
                <dependency>
                        <groupId>org.springframework.cloud</groupId>
                        <artifactId>spring-cloud-dependencies</artifactId>
                        <version>Dalston.SR1</version>
                        <type>pom</type>
                        <scope>import</scope>
                </dependency>
        </dependencies>
</dependencyManagement>
```

2) Rename the application.properties to **bootstrap.properties** in src/main/resources folder and add an application name and a configuration server URL. Also comment out configuration properties.

```
#src/main/resources/bootstrap.properties
spring.application.name=search-service
spring.cloud.config.uri=http://localhost:8888

server.port=8090

#Below properties are moved to seach-service.properties file in Git
#spring.rabbitmq.host=localhost
#spring.rabbitmq.port=5672
#spring.rabbitmq.username=guest
#spring.rabbitmq.password=guest

management.security.enabled=false
```

Note: The 'search-service' is a logical name given to the search microservice. This will be treated as service ID. The Config server will look for **`search-service.properties`** in the GIT repository to resolve the properties. Hence search-service.properties file should be created in ${user.home}/config-repo which is explained in next step.

3) Create a new **search-service.properties** under the config-repo folder where the Git repository is created. Move service-specific properties from bootstrap.properties to the new search-service.properties file.

# **C:/Users/Ramesh-PC/config-repo/search-services.properties**

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

Commit the changes in the Git repository.

```
git add -A .
git commit -m "Adding search-service.properties"
```

4) Type Config server URL "http://localhost:8888/search-service/default/master" to see the properties specific to `search-service.properties`, which were added in the earlier step.

5) Repeat all above changes in booking and check-in microservices.

6) **Restart** all services and check them. Also perform booking and check-in through web at http://localhost:8001
Username: guest
Password: guest123

7) In order to demonstrate the centralized configuration of properties and propagation of changes, add a new application-specific property to the search-service.properties file. We will add **originairports.shutdown** to temporarily take out an airport from the search. Users will not get any flights when searching for an airport mentioned in the shutdown list.
Add below property in search-service.properties file in Git and commit it in Git repository.

```
# C:/Users/Ramesh-PC/config-repo/search-services.properties
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest

originairports.shutdown=SEA
```

Commit the changes in the Git repository.

```
git add –A .
git commit –m "adding origin airports shutdown property"
```

8) Stop search microservice. Modify the Search microservice code to use the configured parameter, originairports.shutdown. A **@RefreshScope** annotation has to be added at the class

level to allow properties to be refreshed when there is a change. In this case, we are adding a refresh scope to the SearchRestController class.

```
//A RefreshScope annotation has to be added at the class level to allow properties to be
refreshed when there is a change in search-service.properties when /refresh actuator is used.
@RefreshScope
Public class SearchRestController {

}
```

Also, add the following instance variable as a place holder for the new property that is just added in the Config server. The property name in the search-service.properties file must match.

```
@RefreshScope
Public class SearchRestController {
        @Value("${originairports.shutdown}")
        private String originAirportShutdownList;
        …
}
```

Change the application code to use this property. This is done by modifying the search method as follows:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.cloud.context.config.annotation.RefreshScope;
@RefreshScope
public class SearchRestController {
        private static final Logger logger = LoggerFactory.getLogger(SearchComponent.class);

        @Value("${originairports.shutdown}")
        private String originAirportShutdownList;
        …

        @RequestMapping(value="/get", method = RequestMethod.POST)
        Public List<Flight> search(@RequestBody SearchQuery query){
              logger.info("Input : "+ query);
              if(Arrays.asList(originAirportShutdownList.split(",")).contains(query.getOrigin())){
                      logger.info("The origin airport is in shutdown state");
                      return new ArrayList<Flight>();
               }
               /*tpm.increment();
```

```
                    gaugeService.submit("tpm", tpm.count.intValue());*/

                    return searchComponent.search(query);
            }
        }
```

The search method is modified to read the parameter originAirportShutdownList and see whether the requested origin is in the shutdown list. If there is a match, then instead of proceeding with the actual search, rather the search method will return an empty flight list.

9) Restart search microservice.
10) Goto website (http://localhost:8001) and search travelling from SEA and going to SFO.
Observation: 1) Empty flights list should be displayed in web page.
2) 'The origin airport is in shutdown state' should be printed on search console.

Handling configuration changes

This section will demonstrate how to propagate configuration changes automatically when there is a change:

1) Change the property in the search-service.properties file to the following:

   ```
   originairports.shutdown:NYC
   ```

   Commit the changes in the Git repository.
   git add –A .
   git commit –m "modifying origin airports shutdown value"

   Refresh the Config server URL (`http://localhost:8888/search-service/default/master`) for this service and see whether the property change is reflected.

2) Check in website project now without restarting search microservice. We can observe that the change is not reflected in the search service, and the service is still working with an old copy of the configuration properties.

3) In order to **force reloading of the configuration properties**, call the **/refresh** endpoint of the Search microservice. This is actually the actuator's refresh endpoint. The following command will send an empty POST to the /refresh endpoint:
   curl –d {} localhost:8090/refresh

   Note: If we install Git for Windows we get Curl automatically too. The installation comes with GNU bash (**git-bash.exe**), a really powerful shell.
   Double click on "D:\Program Files\Git\git-bash.exe"

$curl –d {} localhost:8090/refresh

4) Now check in website without restarting search microservice. We can observe that the change is reflected in the search service, and the service is refreshing with new copy of the configuration properties.
   Observation: With this approach, configuration parameters can be changed without restarting the microservices.

Spring Cloud Bus

The above approach is good in case of few instances. In case of many instances, hitting /refresh for every instance is not good.
The **Spring Cloud Bus** provides a mechanism to refresh configurations across multiple instances without knowing how many instances there are, or their locations.
The following steps are used to configure cloud bus:

1) Stop search microservice and add below dependency:
```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

2) Copy search microservice project. Set port number as 8091 and **start both instances**. The two instances of the Search service are running now, one on port 8090 and another one on 8091.

3) Now, change origin airports shutdown value in search-service.properties.
```
originairports.shutdown:SEA
```
   Commit the changes in the Git repository.
```
git add –A .
git commit –m "changing origin airports shutdown value"
```

4) Run the following command with **/bus/refresh**. Note that we are running a new bus endpoint against one of the instances, 8090 in this case:
```
$ curl –d {} localhost:8090/bus/refresh
```

5) Immediately, we will see the following message for both instances on search console:
   Observation: Received remote refresh request. Keys refreshed [originairports.shutdown]

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**. 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.

Example:

```
@Component
public class BookingComponent {
        private static final String FareURL = "http://localhost:8081/fares";
        public long book(BookingRecord record) {
                //call fares to get fare
                Fare fare = restTemplate.getForObject(FareURL +"/get?flightNumber=" +
                                record.getFlightNumber()+"&flightDate="+record.getFlightDate(),Fare.class);

}
```

Feign is a Spring Cloud Netflix library for providing a **higher level of abstraction** over REST-based service calls. Spring Cloud Feign works on a declarative principle. When using Feign, we write declarative REST service interfaces at the client, and use those interfaces to program the client. The developer need not to worry about the implementation of this interface. This will be dynamically provisioned by Spring at runtime. With this declarative approach, developers need not get into the details of the HTTP level APIs provided by RestTemplate.

The following steps are required to use Feign:

1) Stop **both** booking and checkin Microservices. In order to use Feign, add below dependency to the pom.xml file in booking microservice:

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

2) Create a new FareServiceProxy interface. This will act as a proxy interface of the actual Fare service.

```
package com.brownfield.pss.book.component;
//Feign makes writing web service (REST)clients easier
//This annotation tells Spring to create a REST client based on the interface provided.
//The "fares-proxy" is an arbitrary client name, which is used by Ribbon load balancer
@FeignClient(name = "fares-proxy", url = "localhost:8081/fares")
```

```
public interface FareServiceProxy {
            @RequestMapping(value = "/get", method = RequestMethod.GET)
            Fare getFare(@RequestParam(value = "flightNumber") String flightNumber,
                                                  @RequestParam(value = "flightDate") String flightDate);
}
```

3) Use this fare proxy to call fare microservice. In the Booking microservice, we have to tell Spring that Feign clients exist in the Spring Boot application, which are to be scanned and discovered. This will be done by adding `@EnableFeignClients` at the class level of `BookingComponent`. Optionally, we can also give the package names to scan.

```
package com.brownfield.pss.book.component;
@EnableFeignClients
public class BookingComponent {

}
```

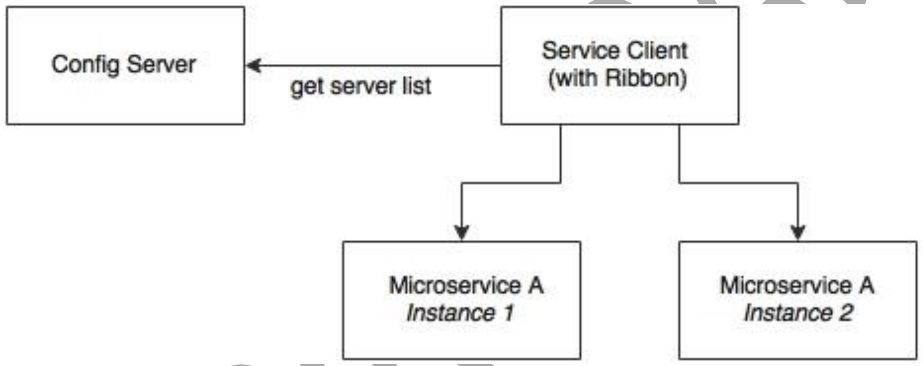
4) In BookingComponent, make changes to the calling to part. This is as simple as calling other java interface.

```
@EnableFeignClients
public class BookingComponent {
            //private static final String FareURL = "http://localhost:8081/fares"; //Not required if we use Feign

            //@Autowired
            //private RestTemplate restTemplate; //Not required if we use Feign

            @Autowired
            FareServiceProxy fareServiceProxy;

            public long book(BookingRecord record) {
            Fare fare = null;
            try{
             // call fares to get fare
             //fare = restTemplate.getForObject(FareURL + "/get?flightNumber=" +
                  record.getFlightNumber() + "&flightDate=" + record.getFlightDate(), Fare.class);
              fare = fareServiceProxy.getFare(record.getFlightNumber(),
              record.getFlightDate());logger.info("calling fares to get fare " + fare);
            }catch(Exception e){
                        logger.error("FARE SERVICE IS NOT AVAILABLE");
            }
```

```
        }
}
```
5) Start **both** booking and checkin services.
   Observation: 'Looking for fares flightNumber BF101 flightDate 22-JAN-16' in fare microservice console

Load Balancing Using Ribbon

So far we were running with a single instance of the microservice. The URL is hardcoded both in client as well as in the service-to-service calls. In the real world, this is not a recommended approach, since there could be more than one service instance. If there are multiple instances, then ideally, we should use a load balancer to abstract the actual instance locations, and configure an alias name or the load balancer address in the clients. The load balancer then receives the alias name, and resolves it with one of the available instances. With this approach, we can configure as many instances behind a load balancer. This is achievable with Spring Cloud Netflix **Ribbon**. Ribbon is a client-side load balancer which can do **round-robin load balancing** across a set of servers.


Config Server
get server list
Service Client (with Ribbon)
Microservice A Instance 1
Microservice A Instance 2

As shown in the preceding diagram, the Ribbon client looks for the Config server to get the list of available microservice instances, and, by default, applies a round-robin load balancing algorithm.

The following steps are used to configure load balancing:

1) In order to use Ribbon, stop booking and checkin microservices and add the following dependency to the `pom.xml` file in booking microservice:

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

2) Copy fare microservice project named as FareFlightTickets2. Set port number as 8082 and start this project. The two instances of the fare service are running now, one on port 8081 and another one on 8082.

3) Modify FareServiceProxy class to use Ribbon client.
//The "fares-proxy" is an arbitrary client name, which is used by Ribbon load balancer
//@FeignClient(name = "fares-proxy", url = "localhost:8081/fares") //without ribbon
@FeignClient(**name = "fares-proxy"**)
@RibbonClient
public interface FareServiceProxy {
    //@RequestMapping(value = "/get", method = RequestMethod.GET) //without ribbon
    @RequestMapping(value = "**/fares/get**", method = RequestMethod.GET)
    Fare getFare(@RequestParam(value = "flightNumber") String flightNumber,
@RequestParam(value = "flightDate") String flightDate);
}

4) Update the Booking microservice configuration file, **booking-service.properties**, to include a new property to keep the list of the Fare microservices:
#booking-service.properties
fares-proxy.ribbon.listOfServers=localhost:8081,localhost:8082

Commit the changes in the Git repository.
git add –A .
git commit –m "adding new configuration"

5) Start booking and checkin microservices.
Observation: The following text will be printed on booking microservice console:
DynamicServerListLoadBalancer:{NFLoadBalancer:name=**fares-proxy**,current list of Servers=[**localhost:8081, localhost:8082**], Load balancer stats=Zone stats: {unknown=[Zone:unknown; Instance count:2; Active connections count: 0; Circuit breaker tripped count: 0; Active connections per server: 0.0;]

When booking microservice is bootstrapped, the CommandLineRunner automatically inserts one booking record. This will go to the first server.
Observation: The following text will be printed on fare service1 console:
c.b.pss.fares.component.FaresComponent: Looking for fares flightNumber BF101 flightDate 22-JAN-16

6) When we book ticket through website project, it calls the booking service. This request will go to the second server.
   Observation: The following text will be printed on fare service2 console:
   c.b.pss.fares.component.FaresComponent: Looking for fares flightNumber BF101 flightDate 22-JAN-16

Eureka for Registration and Discovery

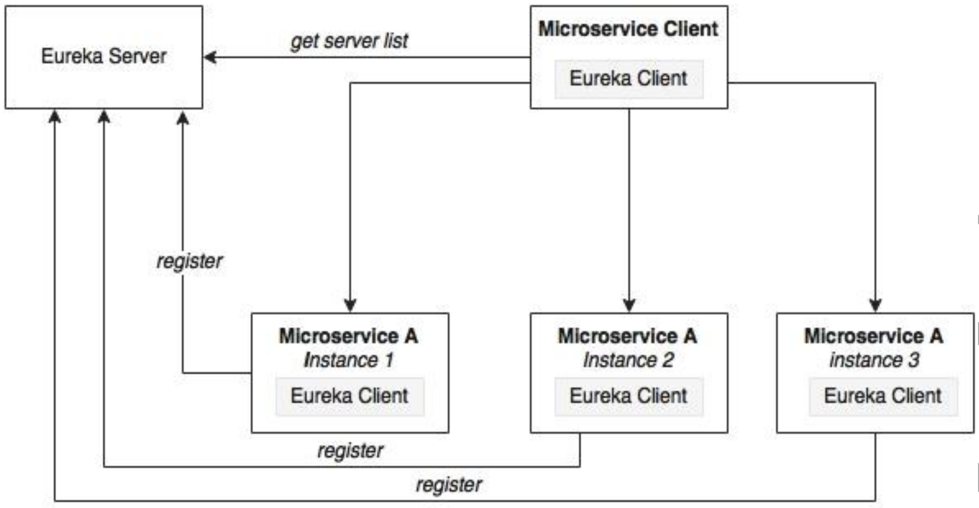
So far, we have achieved externalizing configuration parameters as well as load balancing across many service instances.
Ribbon-based load balancing is sufficient for most of the microservices requirements. However, this approach falls short in a couple of scenarios:

a) 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.
b) 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.
c) 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.

The Ribbon approach partially addresses this issue. With Ribbon, we can dynamically change the service instances, but whenever we add new service instances or shut down instances, we will have to manually update the Config server. Though the configuration changes will be automatically propagated to all required instances, the manual configuration changes will not work with large scale deployments. When managing large deployments, automation, wherever possible, is paramount.
To fix this gap, the microservices should self-manage their life cycle by **dynamically registering service availability**, and **provision automated discovery for consumers**.
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.
Dynamic discovery is where clients look for the service registry to get the current state of the services topology, and then invoke the services accordingly. In this approach, instead of statically configuring the service URLs, the URLs are picked up from the service registry.

There are a number of options available for dynamic service registration and discovery. Netflix **Eureka**, ZooKeeper, and Consul are available as part of Spring Cloud. In this chapter, we will focus on the Eureka implementation.

Eureka is primarily used for self-registration, dynamic discovery, and load balancing. Eureka uses Ribbon for load balancing internally.



As shown in the preceding diagram, Eureka consists of a server component and a client-side component. The server component is the registry in which all microservices register their availability. The registration typically includes service identity and its URLs. The microservices use the Eureka client for registering their availability. The consuming components will also use the Eureka client for discovering the service instances. By default, the Eureka server itself is another Eureka client. This is particularly useful when there are multiple Eureka servers running for high availability.

Setting up the Eureka server

The following steps required for setting up the Eureka server:

1) Create a new Spring Starter project named 'EurekaServer', and select **Eureka Server**, **Config Client**, and **Actuator**.

2) Rename application.properties to **bootstrap.properties** since this is using the config server.

```
spring.application.name=eureka-server1
server.port:8761
spring.cloud.config.uri=http://localhost:8888
```

3) Create a **eureka-server1.properties** file

```
spring.application.name=eureka-server1
#back to the same standalone instance
eureka.client.serviceUrl.defaultZone:http://localhost:8761/eureka/
eureka.client.registerWithEureka:false
eureka.client.fetchRegistry:false
```

Commit changes to the Git repository.

```
git add –A .
git commit –m "adding new configuration"
```

4) Rename package name to 'edu.aspire.eurekaserver' and add **@EnableEurekaServer** in EurekaServerApplication.java file.

5) Start Eureka server. Ensure that config server should be started before starting eureka server. Once the server is started, type `http://localhost:8761` in a browser to see the Eureka console.
Observation: No instances available

6) Stop all Microservices and add the following additional dependency in all microservices in their pom.xml files to enable dynamic registration and discovery using the Eureka service.

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Note: The Config Client, Actuator, Web dependencies should be in pom.xml file.

7) Rename application.properties file to bootstrap.properties in both fares microservices. Add config server properties.

```
spring.application.name=fares-service
spring.cloud.config.uri=http://localhost:8888
```

Note: Ensure that give same spring.application.name in both fares microservices.

8) Create **fares-service.properties** in Git repository
eureka.client.serviceUrl.defaultZone: http://localhost:8761/eureka/

Note: The above property should be added to all microservices in their respective configuration files under `config-repo`. This will help the microservices to connect to the Eureka server.

Commit to Git
git add –A .
git commit –m "adding new configuration"

9) Add **@EnableDiscoveryClient** in all microservices in their respective Spring Boot main classes. This asks Spring Boot to register these services at start up to advertise their availability.

10) Start fare microservice
Observation:
On fare microservice console:
DiscoveryClient_FARES-SERVICE/Ramesh:fares-service:8081 - registration status: 204

Eureka server console:
Registered instance FARES-SERVICE/Ramesh:fares-service:8081 with status UP

Web: Refresh http://localhost:8761
Instances currently registered with Eureka
FARES-SERVICE n/a (1) (1) UP (1) - Ramesh:fares-service:8081

11) Eureka internally uses Ribbon for load balancing hence remove ribbon dependency from booking microservice pom file. Ensure that eureka, config, auctuator and web starters should be in pom.xml file.

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

12) Also remove the `@RibbonClient` annotation from the `FareServiceProxy` class.

13) Update `@FeignClient(name="fares-service")` to match the actual Fare microservices' service ID.

14) Also remove the list of servers from the `booking-service.properties` file. Ensure that eureka.client.serviceUrl.defaultZone property should be added in booking-service.properties file.

```
#fares-proxy.ribbon.listOfServers=localhost:8081,localhost:8082
eureka.client.serviceUrl.defaultZone: http://localhost:8761/eureka/
```

Commit to Git

```
git add –A .
git commit –m "adding new configuration"
```

15) Start booking and check-in microservices. Also start website.

Observation:

On booking console:

```
DynamicServerListLoadBalancer:{NFLoadBalancer:name=fares-service,current list of Servers=[Ramesh:8081, Ramesh:8082],Load balancer stats=Zone stats: …]
```