

Spring Micro Services

Monolithic architecture overview:

Monolithic architecture allows the development of different components such as presentation, application logic, business logic, and data access objects (DAO), and then you either bundle them together in enterprise archive (EAR)/web archive (WAR), or store them in a single directory hierarchy (for example, Rails, NodeJS, and so on).

Limitation of Monolithic architecture:

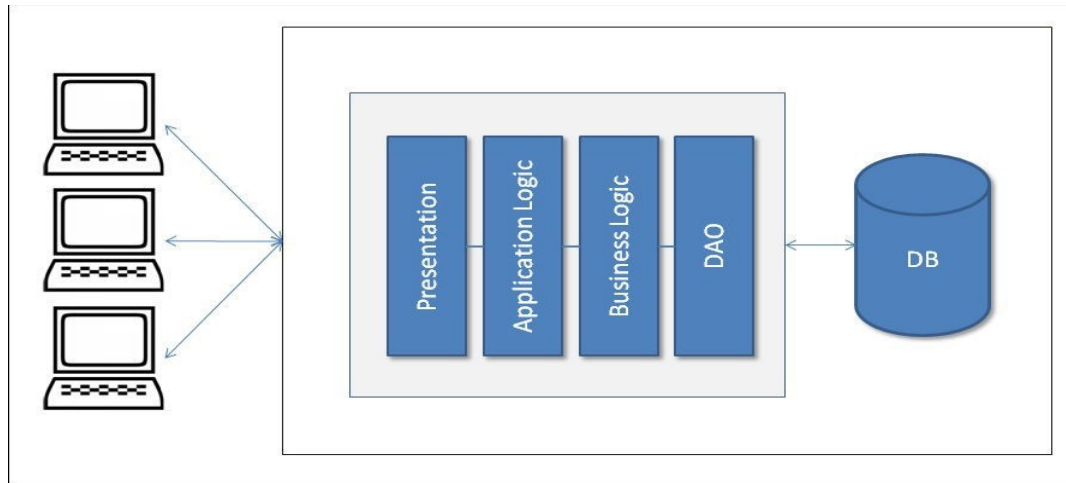
organizations are using agile methodologies to develop applications; it is a fast paced development environment and is also on a much larger scale after the invention of cloud and distributed technologies.

monolithic architecture serve a similar purpose and be aligned with agile methodologies.

To understand the design differences between monolithic and μ Services, let's take an example of a restaurant table-booking application. This app may have many services such as customers, bookings, analytic and so on, as well as regular components such as presentation and database.

We'll explore three different designs here – **traditional monolithic design, monolithic design with services and μ Services design.**

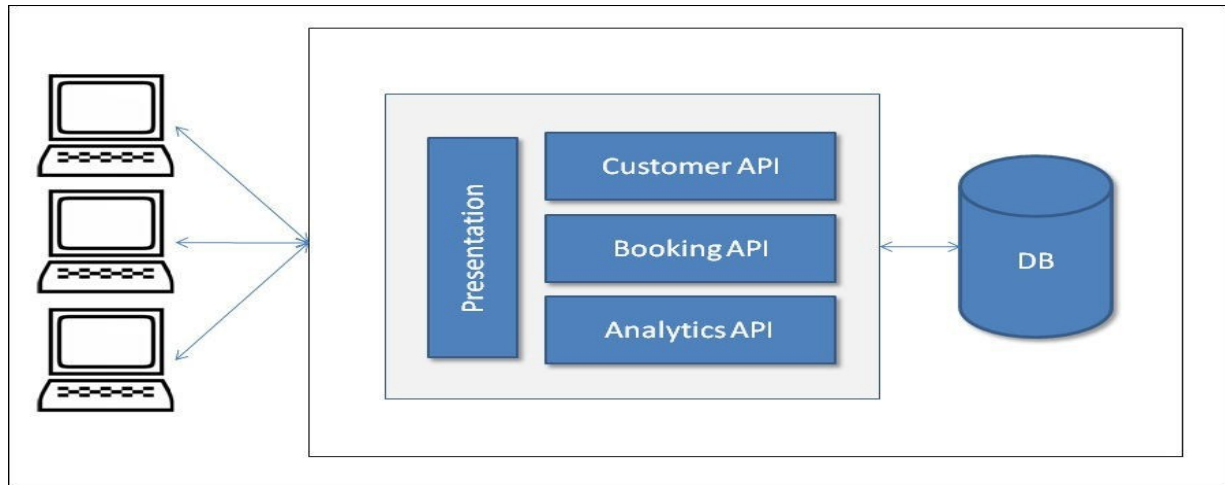
Traditional Monolithic Design



In traditional monolithic design, everything is bundled in the same archive such as presentation code, application logic and business logic code, and DAO and related code that interacts with the database files or another source.

Monolithic design with services:

The following diagram depicts the monolithic application with different services; here services are being used with a presentation component. All services, the presentation component, or any other components are bundled together:

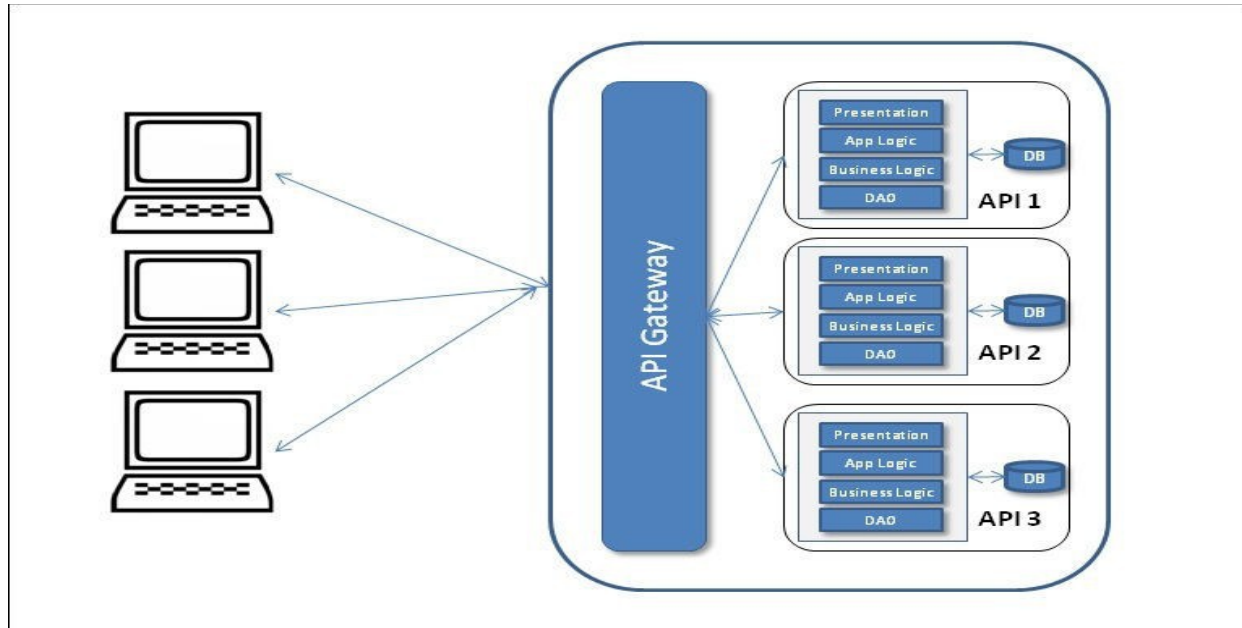


MicroServices Design:

The following third design depicts the μ Services. Here, each component represents autonomy. Each component could be developed, built, tested, and deployed independently. Here, even the application UI component could also be a client and consume the μ Services. For the purpose of our example, the layer designed is used within μ Service.

The API gateway provides the interface where different clients can access the individual services and solve the following problems:

- ✓ What to do when you want to send different responses to different clients for the same service. For example, a booking service could send different responses to a mobile client (minimal information) and a desktop client (detailed information) providing different details and something different again to a third-party client.
- ✓ A response may require fetching information from two or more services



After observing all the sample design diagrams, which are very high-level designs, we might find out that in monolithic design, the components are bundled together and tightly coupled and all the services are part of the same bundle.

Similarly, in the second design figure, we can see a variant of the first figure where all services could have their own layers and form different APIs, but, as shown in the figure, these are also all bundled together.

But In μ Services, design components are not bundled together and have loose coupling. Each service has its own layers and DB and is bundled in a separate archive. All these deployed services provide their specific API such as Customers, Bookings, or Customer which are ready to consume. Even the UI is also deployed separately and designed using μ Service. For this reason, it provides various advantages over its monolithic design. But It would also be noted that there are some exceptional cases where monolithic app development is highly successful, like peer-to-peer e-commerce web applications.

MicroServices:

Martin Fowler explains:

The term "microservice" was discussed at a workshop of software architects near Venice in May, 2011 to describe what the participants saw as a common architectural style that many of them had been recently exploring. In May 2012, the same group decided on " μ Services" as the most appropriate name."

What are MicroServices?

MicroServices are an architecture style used by many organizations to develop more physically separated modular applications to achieve a high degree of agility, speed of delivery, and scale.

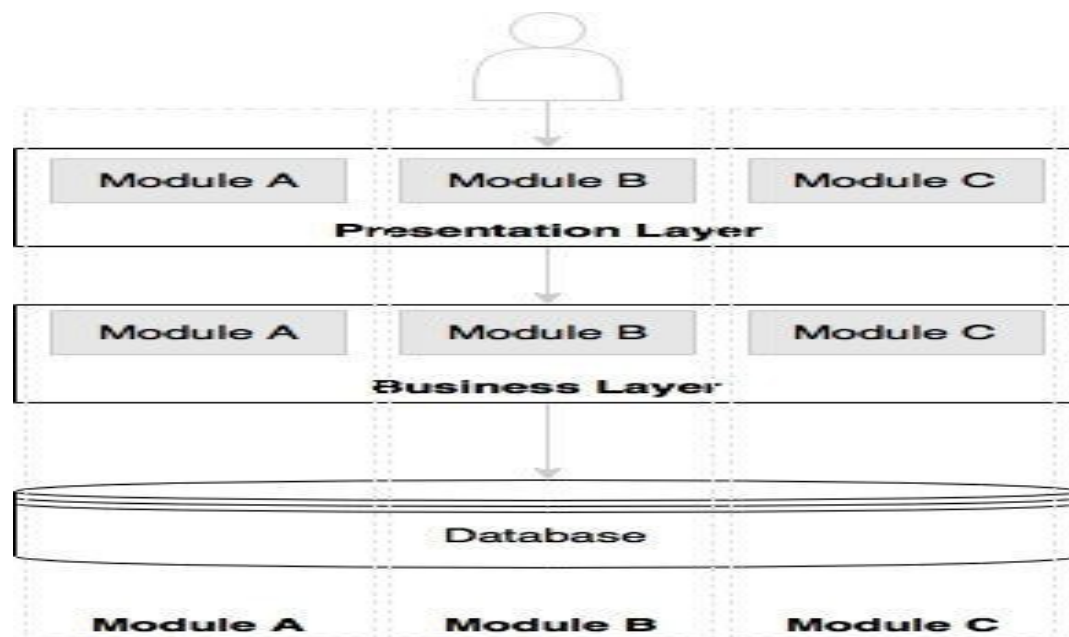
MicroServices are not invented rather evolved.

Evolution Of MicroServices

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. These organizations solved a number of issues they were experiencing with their monolithic applications. Following the success of these organizations, many other organizations started adopting this as a common pattern to re factor their monolithic applications. Later, evangelists termed this pattern as the **MicroServices** architecture.

MicroServices are an architectural style to building the system/application as a set of smaller business components that are autonomous, self-contained, and loosely coupled.

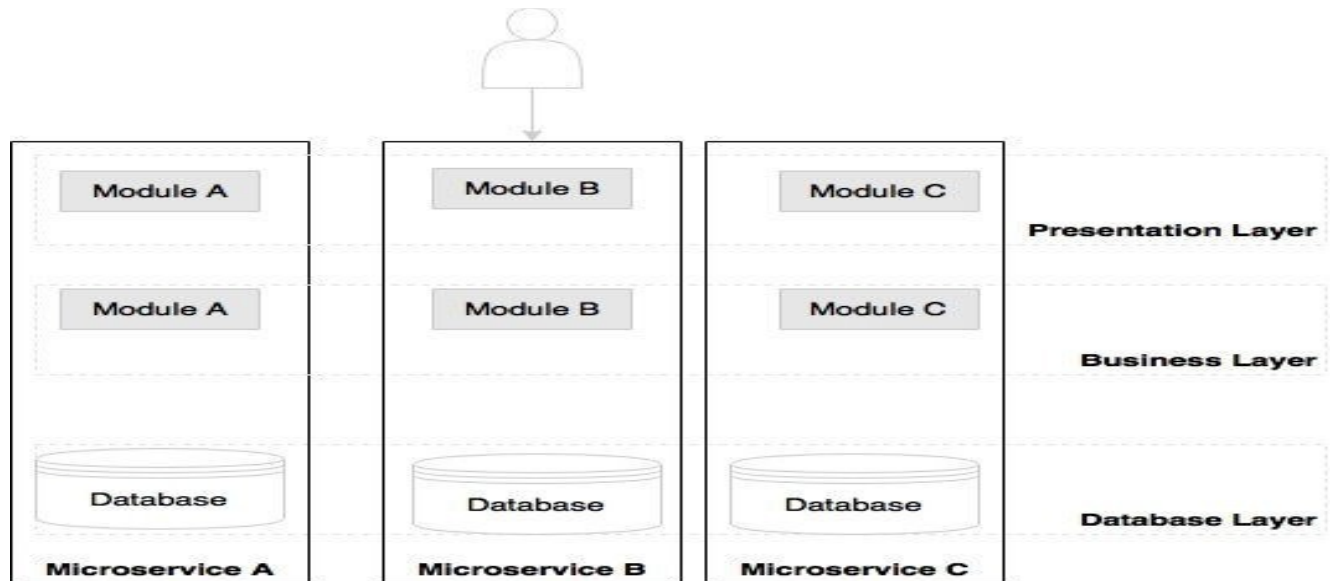
Lets Consider the below diagram:



The preceding diagram depicts a traditional N-tier application architecture having a presentation layer, business layer, and database layer. The modules A, B, and C represent three different business capabilities. The layers in the diagram represent a separation of architecture concerns. Each layer holds all three business capabilities pertaining to this 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.

Lets Consider the following Diagram:



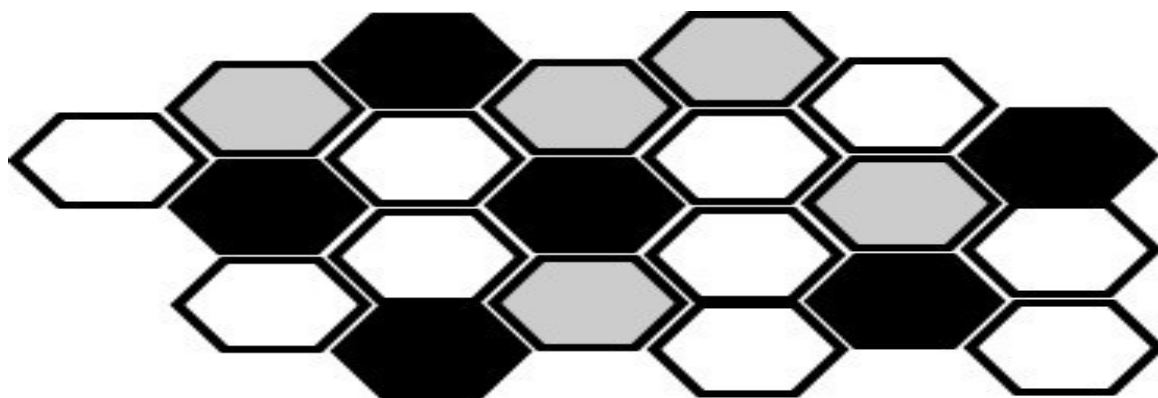
As we can note in the preceding diagram, the boundaries are inversed in the MicroServices architecture. Each vertical slice represents a micro service.

Each microservice has its own presentation layer, business layer, and database layer. By doing so, changes to one micro service do not impact others.

There is no standard for communication or transport mechanisms for microservices. In general, microservices communicate with each other using widely adopted lightweight protocols, such as HTTP and REST, or messaging protocols, such as JMS.

MicroServices – the honeycomb analogy

The honeycomb is an ideal analogy for representing the evolutionary microservices architecture.



In the real world, bees build a honeycomb by aligning hexagonal wax cells. They start small, using different materials to build the cells. Construction is based on what is available at the time of building. Repetitive cells form a pattern and result in a strong fabric structure. Each cell in the honeycomb is independent but also integrated with other cells.

By adding new cells, the honeycomb grows organically to a big, solid structure. **The content inside each cell is abstracted and not visible outside.** Damage to one cell does not damage other cells, and bees can reconstruct these cells without impacting the overall honeycomb.

Principles of MicroServices

we will examine some of the principles of the MicroServices architecture. These principles are **"must have"** when designing and developing MicroServices.

Principle-1: Single Responsibility Per Service

The **"Single Responsibility Per Service"** principle is one of the principles defined as part of the **SOLID** design pattern. **It states that a unit should only have one responsibility.** There are five principles to be followed while designing a class, these five principles are said to be SOLID:

(SRP) The SingleResponsibilityPrinciple

(OCP) The OpenClosedPrinciple

(LSP) The LiskovSubstitutionPrinciple

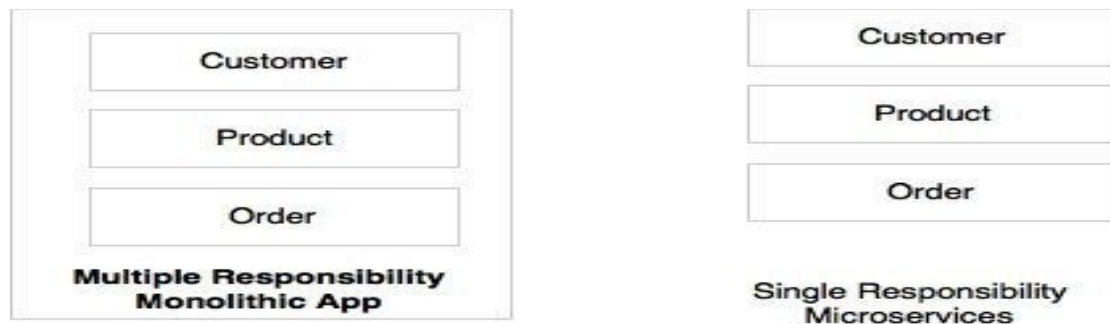
(ISP) The InterfaceSegregationPrinciple

(DIP) The DependencyInversionPrinciple

Source:

<http://c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign>

This implies that a unit, either a class, a function, or a service, should have only one responsibility. At no point should two units share one responsibility or one unit have more than one responsibility. A unit with more than one responsibility indicates tight coupling.



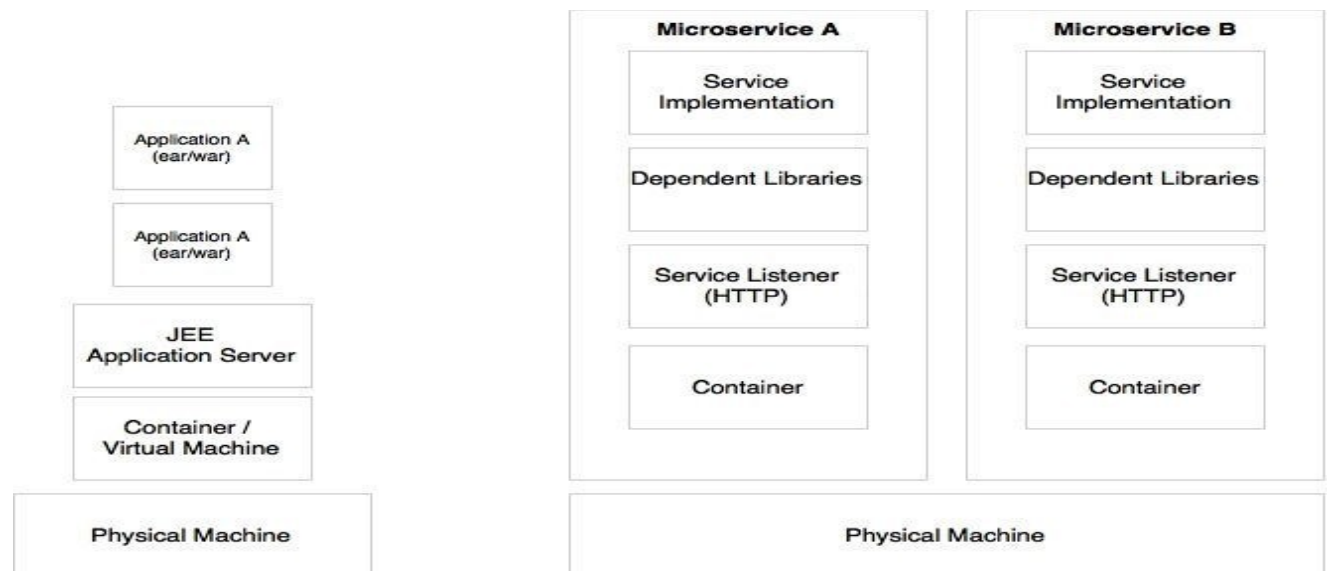
As shown in the preceding diagram, Customer, Product, and Order are different functions of an e-commerce application. Rather than building all of them into one application, **it is better to have three different services, each responsible for exactly one business function, so that changes to one responsibility will not impair others.**

In the preceding scenario, Customer, Product, and Order will be treated as three independent MicroServices.

Principle-2: Autonomous

MicroServices are **self-contained, independently deployable, and autonomous services** that takes full responsibility of a business capability and its execution. They bundle all dependencies, including library dependencies, and execution environments such as web servers and containers or virtual machines that abstract physical resources.

In traditional application 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 container. **In the MicroServices approach, each microservice will be built as a fat Jar, embedding all dependencies and run as a standalone Java process.**



As shown in the preceding diagram, MicroServices get their own containers for execution. Containers are portable, independently manageable, lightweight run time environments. Container technologies, such as Docker, are an ideal choice for MicroServices deployment.

Characteristics Of MicroServices:

There is no single, concrete, and universally accepted definition for MicroServices. However, all successful MicroServices implementations exhibit a number of common characteristics. Therefore, it is important to understand these characteristics rather than sticking to theoretical definitions.

Some of the common characteristics are:

1)MicroServices are first-class citizens:

In the MicroServices world, services are first-class citizens. Micro services expose service endpoints as APIs and abstract all their realization details. The internal implementation logic, architecture, and

technologies (including programming language, database, quality of services mechanisms, and so on) are completely hidden behind the service API.

Moreover, in the MicroServices architecture, there is no more application development; instead, organizations focus on service development.

2) MicroServices are lightweight:

Well-designed MicroServices are aligned to a single business capability, so they perform only one function. As a result, one of the common characteristics we see in most of the implementations of the MicroServices are with smaller system components.

Hence in selecting supporting technologies, such as web containers, we will have to ensure that they are also lightweight so that the overall System components remains manageable.

3) MicroServices with polyglot architecture:

As MicroServices are autonomous and abstract everything behind service APIs, it is possible to have different architectures for different MicroServices. A few common characteristics that we see in MicroServices implementations are:

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

Lets consider the below diagram:



As Hotel Search is expected to have high transaction volumes with demanded performance requirements, it is implemented using Erlang. In order to support predictive searching, Elasticsearch is used as the data store.

At the same time, Hotel Booking needs more ACID transactional characteristics. Therefore, it is implemented using MySQL and Java.

In these two micro services, the internal implementations are hidden behind service endpoints defined as REST/JSON over HTTP as MicroServices First Class Citizens.

4) MicroServices are Automated:

As MicroServices break monolithic applications into a number of smaller services, A large number of MicroServices is hard to manage until and unless automation is in place. MicroServices are automated end to end—for example, automated builds, automated testing, automated deployment, and elastic scaling.



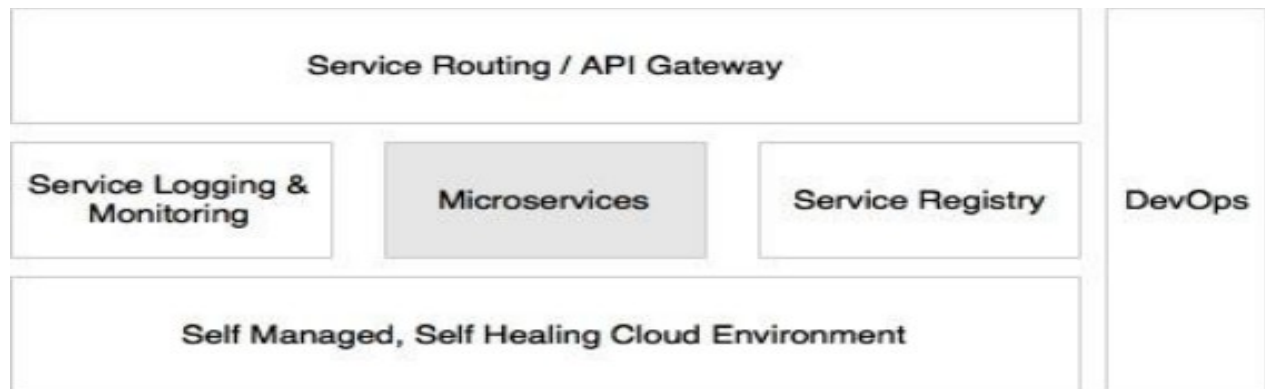
As indicated in the preceding diagram, automations are typically applied during the development, test, release, and deployment phases:

- ❖ **The development phase** is automated using version control tools such as Git together with Continuous Integration (CI) tools such as Jenkins, Travis CI, and so on. Automation of a full build on every code check-in is also achievable with microservices.
- ❖ **The testing phase** will be automated using testing tools such as Selenium, Cucumber, and other AB testing strategies. As microservices are aligned to business capabilities, the number of test cases to automate is fewer compared to monolithic applications, hence regression testing on every build also becomes possible. As microservices are aligned to business capabilities, the number of test cases to automate is fewer compared to monolithic applications, hence regression testing on every build also becomes possible.
- ❖ **Infrastructure provisioning** is done through container technologies such as Docker, together with release management tools such as Chef or Puppet, and configuration management tools such as Ansible.
- ❖ **Automated deployments** are handled using tools such as Spring Cloud, Kubernetes, Mesos, and Marathon.

5) MicroService Supports Eco-System:

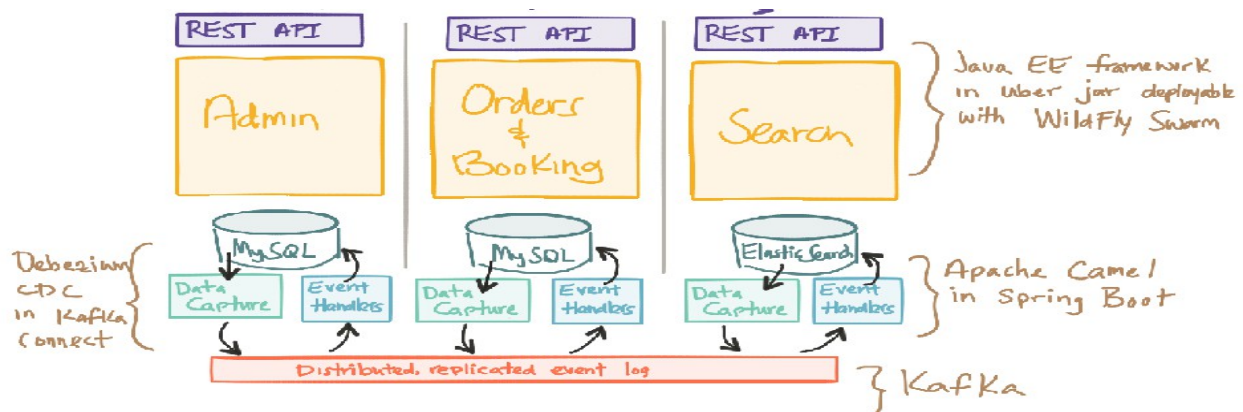
The ecosystem capabilities include: DevOps processes, API gateways, service registry, service routing, centralized log management, extensive monitoring and flow control mechanisms.

MicroServices supports well when the ecosystem capabilities are in place.



6) MicroServices are distributed and dynamic:

Unfortunately, data access becomes much more complex when we move to a MicroServices architecture. That is because the data owned by each microservice is private to that microservice and can only be accessed via its API. **Encapsulating the data ensures that the MicroServices are loosely coupled and can evolve independently of one another.** If multiple services access the same data, schema updates require time-consuming, coordinated updates to all of the services. The solution is to use an eventdriven architecture. In this architecture, a microservice publishes an event when something notable happens, such as when it updates a business entity. Other MicroServices subscribe to those events. When a microservice receives an event it can update its own business entities, which might lead to more events being published.



Micro Services relationship with SOA

As MicroServices are evolved from SOA, many characteristics of MicroServices are similar to SOA. Let's first examine the definition of SOA.

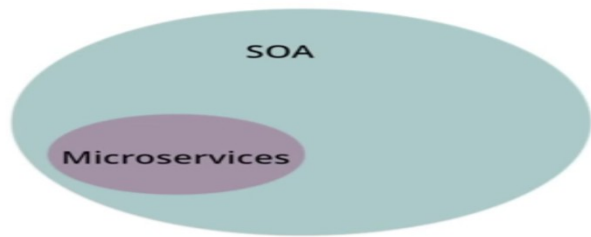
The definition of SOA from The Open Group consortium is as follows:

"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, consolidate drilling reports). **It is self-contained. It may be composed of other services. It is a "black box" to consumers of the service.**

The core difference between SOA and MicroServices lies in the size and scope. As the word "micro" suggests, it has to be significantly smaller than what SOA tends to be. Microservice is a small(er) independently deployable unit. A SOA can be either a monolith or it can be comprised of multiple MicroServices. Martin Fowler says he likes to think SOA to be a superset of MicroServices.



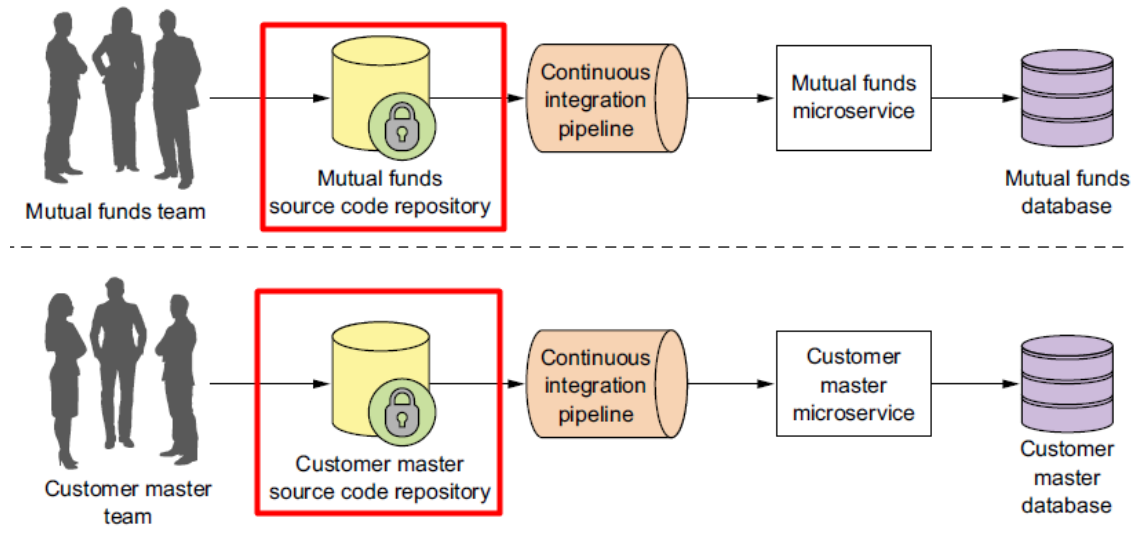
SOA	Micro Services
SOA Services are often implemented in a monolith application and hence multiple services are deployed in to the same J2EE container as part of monolith application	MicroServices must be independently deployable. Meaning each micro service will be built as fat jar /war, including all dependencies and can be able to run as stand alone jar file with the embedded Http Server Listener
A service in SOA, may have many responsibilities	A Micro Service is always a single responsibility component
SOA initially started with SOAP, XML/WSDL	Micro Service architecture uses REST API.
In SOA, services share the data storage	Each service can have an independent data storage

Micro Service Relations with Twelve-Factor apps

Twelve-Factor App, forwarded by Heroku, is a methodology describing the characteristics expected from modern cloud-ready applications. Twelve-Factor App is equally applicable for MicroServices as well. Hence, it is important to understand Twelve-Factor App.

1) Single Code Base

The code base principle advises that each application has a single code base. There can be multiple instances of deployment of the same code base, such as development, testing, and production. Code is typically managed in a source control system such as Git, Subversion, and so on.



As per this each microservice should have its own code base, and this code base is not shared with any other microservice. It also means that one microservice has exactly one code base.

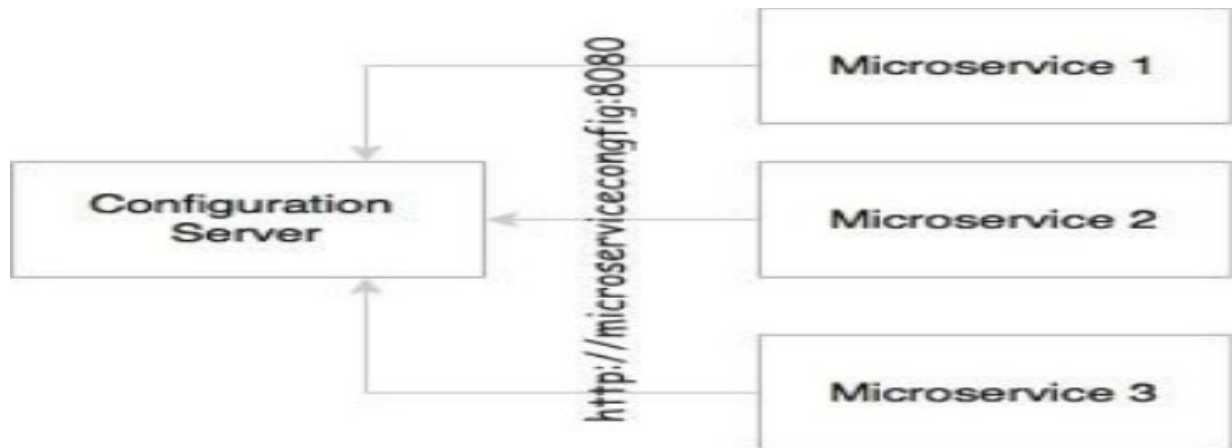
2) Bundling dependencies

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

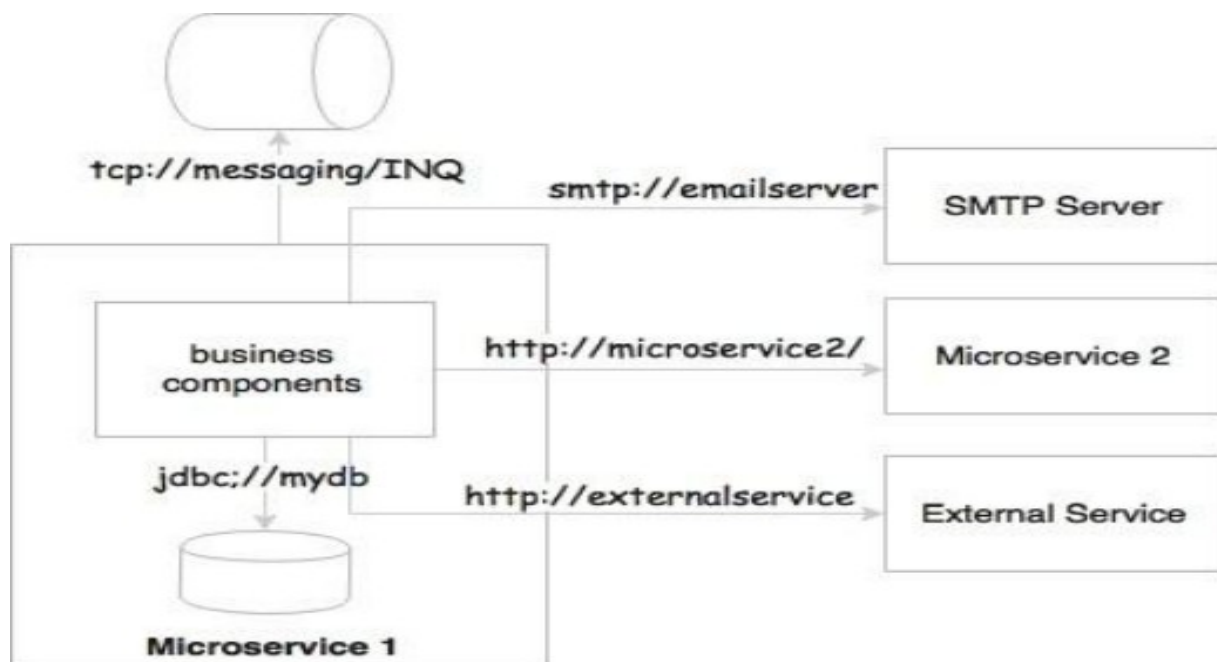
All service configurations should be externalized. An application's configuration parameters vary between environments, such as support to the e-mail IDs or URL of an external system, username, passwords, queue name, and so on. These will be different for development, testing, and production.



The same principle is obvious for microservices as well. The microservices configuration parameters should be loaded from an external source. This will also help to automate the release and deployment process as the only difference between these environments is the configuration parameters.

4) Backing services are addressable

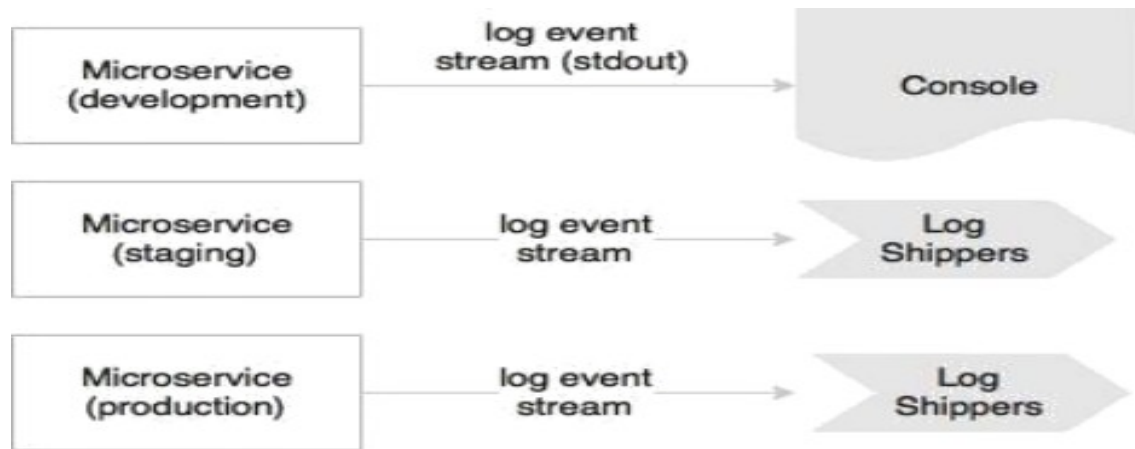
All services need to talk to some external resources during the life cycle of their execution. For example, they could be listening or sending messages to a messaging system, sending an e-mail, persisting data to database, and so on. All these services should be reachable through a URL without complex communication requirements.



In the MicroServices world, MicroServices either talk to a messaging system to send or receive messages, or they could accept or send messages to other service APIs. In a regular case, these are either HTTP endpoints using REST and JSON or TCP- or HTTP-based messaging endpoints.

5) Externalizing logs

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.



In development, the microservice may direct the log stream to stdout, whereas in production, these streams will be captured by the log shippers and sent to a central log service for storage and analysis.

6) 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.

7) Stateless, shared nothing processes

This principle suggests that processes should be stateless and share nothing. 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.

8) Disposability with minimal overhead

In an automated deployment environment, we should be able bring up or bring down instances as quick as possible. In a cloud environment targeting auto-scaling, we should be able to spin up new instance quickly. This is also applicable when promoting new versions of services.

In the MicroServices context, in order to achieve full automation, it is extremely important to keep the size of the application as thin as possible, with minimal start-up and shutdown time. MicroServices also should consider a lazy loading of objects and data.

9) Development and production parity

This principle states the importance of keeping development and production environments as identical as possible. 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.

10) 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.

Not only is this principle valid for microservices, but also it is applicable to any application development.

11) Concurrency to scale out

MicroServices may make use of parallel processing and concurrency frameworks to further speed up or scale up the transaction processing.

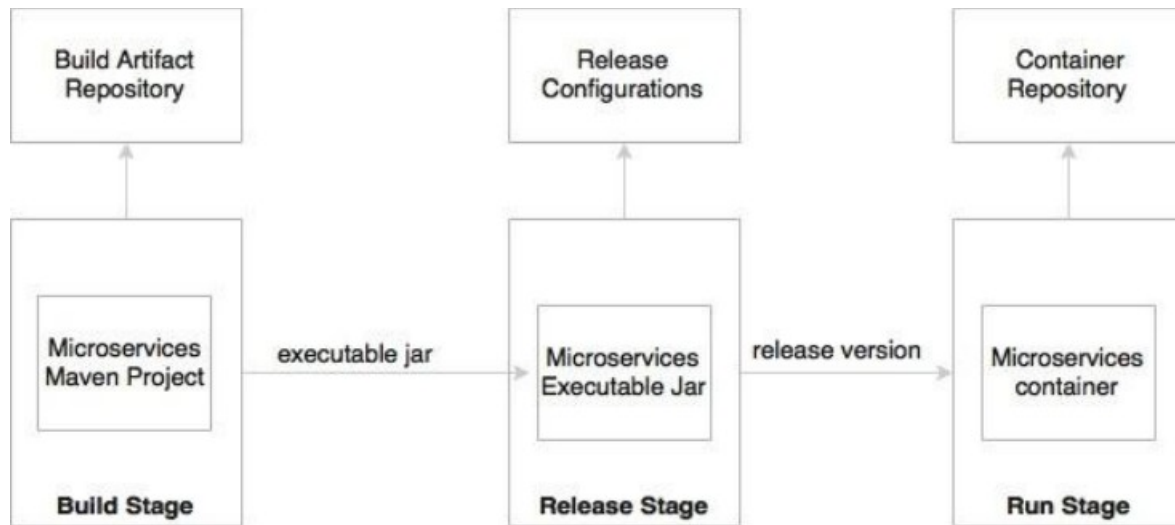
12) Isolation between build, release, and run

This principle advocates a strong isolation between the build, release, and run stages.

The build stage: refers to compiling and producing binaries by including all the assets required. **The release stage:** refers to combining binaries with environment-specific configuration parameters.

The run stage: refers to running application on a specific execution environment.

The pipeline is unidirectional, so it is not possible to propagate changes from the run stages back to the build stage. Essentially, it also means that it is not recommended to do specific builds for production; rather, it has to go through the pipeline.



In MicroServices, the build will create executable JAR files, including the service run time such as an HTTP listener. During the release phase, these executable 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.

Implementing a Micro Service:

Step-1: Configure the application with server port and database configurations as shown below:

```

1##datasource configurations
2spring.datasource.url=jdbc:mysql://localhost:3306/doctor
3spring.datasource.username=root
4spring.datasource.password=root
5spring.datasource.driver-class-name=com.mysql.jdbc.Driver
6
7## server port configuration
8server.port=8081
9
10
11spring.jpa.hibernate.ddl-auto=update
12spring.jpa.show-sql=true
13spring.jpa.properties.hibernate.show_sql=true
14spring.jpa.properties.hibernate.format_sql=true;
15
16## add this property to work with hibernate session factory
17spring.jpa.properties.hibernate.current_session_context_class=org.springframework.orm.hibernate4.SpringSessionContext
18

```

As we are using Hibernate4 to perform the crud with database, we were also have been configured the hibernate related properties.

Step-2: Create HibernateJPASessionFactoryBean as shown below:


```

1 package com.self;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.context.annotation.Bean;
6 import org.springframework.orm.jpa.vendor.HibernateJpaSessionFactoryBean;
7
8 @SpringBootApplication
9 public class DoctorserviceApplication {
10
11     public static void main(String[] args) {
12         SpringApplication.run(DoctorserviceApplication.class, args);
13     }
14
15     @Bean
16     public HibernateJpaSessionFactoryBean sessionFactory() {
17         return new HibernateJpaSessionFactoryBean();
18     }
19 }
20

```

Step-3: Create the domain as shown below:

```

13 /**
14  * @author nsanda
15  *
16  */
17 @Entity
18 public class Doctor implements Serializable{
19
20     /**
21      *
22      */
23     private static final long serialVersionUID = 9110184107227077937L;
24     @Id
25     @GeneratedValue(strategy=GenerationType.AUTO)
26     private int docId;
27     @Override
28     public String toString() {
29         return "Doctor [docId=" + docId + ", docName=" + docName + "]";
30     }
31     private String docName;
32     public int getDocId() {
33         return docId;
34     }
35     public void setDocId(int docId) {
36         this.docId = docId;
37     }
38     public String getDocName() {
39         return docName;
40     }
41     public void setDocName(String docName) {
42         this.docName = docName;
43     }
44 }
45
46

```

Step-4: create the dao and daoimpl as shown below:

```

1 /**
2  *
3  */
4 package com.self;
5
6 import java.util.List;
7
8 /**
9  * @author nsanda
10  *
11  */
12 public interface DoctorDao {
13
14     public Doctor createDoctor(Doctor pat);
15     public List<Doctor> getAllDoctors();
16 }

```

```

1+ /**
4  package com.self;
5
6+ import java.util.List;
15
16- /**
17  * @author nsanda
18  *
19  */
20 @Repository
21 @Transactional
22 public class DoctorDaoImpl implements DoctorDao {
23
24-     @Autowired
25     private SessionFactory sf;
26-     /* (non-Javadoc)
27      * @see com.self.PatientDao#createPatient(com.self.Patient)
28      */
29-     @Override
30     public Doctor createDoctor(Doctor pat) {
31         Session ses = sf.getCurrentSession();
32         ses.save(pat);
33         return pat;
34     }
35
36-     @Override
37     public List<Doctor> getAllDoctors() {
38         Session ses = sf.getCurrentSession();
39         org.hibernate.Query q = ses.createQuery("from Doctor d");
40         return q.list();
41     }
42
43 }
44

```

Step-5: Now create the micro service as below:

```

1  /**
4  package com.self;
5
6  import java.util.List;
17
18
19  /**
20   * @author nsanda
21   *
22   */
23  @RestController
24  @RequestMapping("/docService")
25  public class DoctorServiceCntrl {
26
27      @Autowired
28      private DoctorDao docDao;
29
30      @PostMapping(path="/createDoctor",
31                  consumes=MediaType.APPLICATION_JSON_VALUE,
32                  produces=MediaType.APPLICATION_JSON_VALUE)
33      public ResponseEntity<Doctor> createPatient(@RequestBody Doctor doc){
34          System.out.println("doctor details are:\t"+doc);
35          doc = docDao.createDoctor(doc);
36          return new ResponseEntity<Doctor>(doc, HttpStatus.OK);
37      }
38
39      @GetMapping(path="/getAllDoctors",produces=MediaType.APPLICATION_JSON_VALUE)
40      public ResponseEntity<List<Doctor>> getAllDoctors(){
41          List<Doctor> docList = docDao.getAllDoctors();
42          return new ResponseEntity<List<Doctor>>(docList, HttpStatus.OK);
43      }
44  }
45

```

Step-6: Now the Application.java file should be as below:

```

1 package com.self;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.context.annotation.Bean;
6 import org.springframework.orm.jpa.vendor.HibernateJpaSessionFactoryBean;
7
8 @SpringBootApplication
9 public class DoctorserviceApplication {
10
11     public static void main(String[] args) {
12         SpringApplication.run(DoctorserviceApplication.class, args);
13     }
14
15     @Bean
16     public HibernateJpaSessionFactoryBean sessionFactory() {
17         return new HibernateJpaSessionFactoryBean();
18     }
19 }

```

when we run the application we should see the log as below:

```

main] com.self.DoctorserviceApplication : No active profile set, falling back to default profiles: default
main] ationConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@291a7e3c: startup
main] o.s.cloud.context.scope.GenericScope : BeanFactory id=11801ede-23e0-397e-b92e-def1320b0081
main] f.a.AutowiredAnnotationBeanPostProcessor : JSR-330 'javax.inject.Inject' annotation found and supported for autowiring
main] trationDelegate$BeanPostProcessorChecker : Bean 'org.springframework.transaction.annotation.ProxyTransactionManagementConfiguration' of type [org.springframework
main] trationDelegate$BeanPostProcessorChecker : Bean 'org.springframework.cloud.netflix.metrics.MetricsInterceptorConfiguration$MetricsRestTemplateConfiguration' of
main] trationDelegate$BeanPostProcessorChecker : Bean 'org.springframework.cloud.autoconfigure.ConfigurationPropertiesRebinderAutoConfiguration' of type [org.springf
main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s): 8081 (http)
main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
main] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/8.5.16
Stop-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
Stop-1] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 1585 ms
Stop-1] o.s.b.w.servlet.ServletRegistrationBean : Mapping servlet: 'dispatcherServlet' to [/]
Stop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'metricsFilter' to: [/]
Stop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'characterEncodingFilter' to: [/]
Stop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'hiddenHttpMethodFilter' to: [/]
Stop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'httpPutFormContentFilter' to: [/]
Stop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'requestContextFilter' to: [/]
Stop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'webRequestLoggingFilter' to: [/]
Stop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'applicationContextIdFilter' to: [/]
main] j.LocalContainerEntityManagerFactoryBean : Building JPA container EntityManagerFactory for persistence unit 'default'
main] o.hibernate.jpa.internal.util.LogHelper : HH000204: Processing PersistenceUnitInfo [

```

Now we can access the data as shown:

GET

http://localhost:8081/docService/getAllDoctors

Authorization

Headers

Body

Pre-request Script

Tests

Type

No Auth

Body

Cookies

Headers (4)

Tests

Pretty

Raw

Preview

JSON

1

2

3

4

5

6

7

8

9

10

[

{

"docId": 1,

"docName": "Narsi"

},

{

"docId": 2,

"docName": "Sava"

}

]

Lets consider another micro service I.e.Patient Micro service which is running on the port 8080

```

/**
 * @author nsanda
 *
 */
@RestController
@RequestMapping("/patService")
public class PatientServiceCntrl {

    @Autowired
    private PatientDao patDao;

    @PostMapping(path="/createPatient",
        consumes=MediaType.APPLICATION_JSON_VALUE,
        produces=MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<Patient> createPatient(@RequestBody Patient pat){
        System.out.println("pat details are:\t"+pat);
        pat = patDao.createPatient(pat);
        return new ResponseEntity<Patient>(pat, HttpStatus.OK);
    }

}

```

Which can be accessible as:

POST ▾

http://localhost:8080/patService/createPatient

Authorization

Headers (1)

Body ●

Pre-request Script

Tests

☐ form-data

☐ x-www-form-urlencoded

☒ raw

☐ binary

JSON (application/json) ▾

1 ▾ {

2 "patName": "kane"

3 }

Body

Cookies

Headers (3)


Tests

Pretty

Raw

Preview

JSON ▾



1 ▾ {

2 "patId": 2,

3 "patName": "kane"

4 }