1.What are microservices and why would someone want to look at implementing it.

Microservices is an architectural style which structures and application as a collection of loosely coupled, independently maintainable, testable and deployable services which are organized around business capabilities.

If you have a business focus and you want to solve a use case or a problem efficiently without the boundaries of technology, want to scale an independent service infinitely, highly available stateless services which are easy to maintainable and managed as well as independently testable then we would go ahead and implement Microservices architecture.

2.When should one consider microservice kind of architecture?

There are two cases.

If you already have a monolith application and it grows to an extent where there are problems in scaling or we are not able to reutilize the components/modules/services across different projects/platforms and there is a need to do so. As well as at the same time implementing new features is painful and more error-prone and it is difficult to scale further.

For new applications where implementation has not started yet started, we can think of a business case to be efficiently implemented, which can be easily maintainable, testable and scalable in the future and might be used across other projects/products/platforms at the same time.

3.How would you test microservice based architecture?

One should have unit and integration tests where all the functionality of a microservice can be tested. One should also have component based testing.

One should have contract tests to assert that the expectations by the client is not breaking. End-to-end test for the microservices, however, should only test the critical flows as these can be time-consuming. The tests can be from two sides, consumer-driven contract test and consumer-side contract test.

You can also leverage Command Query Responsibility Segregation to query multiple databases and get a combined view of persisted data.

4.What is service discovery? And how is it helpful?

In a cloud environment where docker images are dynamically deployed on any machine or IP + Port combination, it becomes difficult for dependent services to update at runtime. Service discovery is created due to that purpose only.

Service discovery is one of the services running under microservices architecture, which registers entries of all of the services running under the service mesh. All of the actions are available through the REST API. So whenever the services are up and running, the individual services registers themselves to service discovery service and service discovery services maintains heartbeat to make sure that those services are alive. That also serves the purpose of monitoring services as well. Service discovery also helps in distributing requests across services deployed in a fair manner.

5.What is client side and server side service discovery?

Instead of clients directly connecting to load balancer, in this architectural pattern the client connects to the service registry and tries to fetch data or services from it.

Once it gets all data, it does load balancing on its own and directly reaches out to the services it needs to talk to.

This can have a benefit where there are multiple proxy layers and delays are happening due to the multilayer communication.

In server-side discovery, the proxy layer or API Gateway later tries to connect to the service registry and makes a call to appropriate service afterward. Over here client connects to that proxy layer or API Gateway layer.

6.Explain how can you scale a microservice based system?

Assuming that the majority of providers using microservices architecture,

One can scale the system by increasing the number of instances of service by bringing up more containers.

One can also apply to cache at microservice layer which can be easy to manage as an invalidation of the cache can be done very easily as the microservice will be the single source of truth.

Caching can also be introduced at the API Gateway layer where one can define caching rules like when to invalidate the cache.

One can also shut down some containers when the requirement is less. That is, scale down

**[7.Define Microservices Architecture?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3580)**

Microservices Architecture is a style of developing a scalable, distributed & highly automated system made up of many small autonomous services. It is not a technology but a new trend evolved out of SOA.

There is no single definition that fully describes the term "microservices". Some of the famous authors have tried to define it in the following way:

1. Microservices are small, autonomous services that work together.
2. Loosely coupled service-oriented architecture with bounded contexts.
3. Microservice architecture is a natural consequence of applying the single responsibility principle at the architectural level.

**[8.](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3581)**

**[Difference between Microservices and SOA](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3581)**

***Microservices are a continuation to SOA.***

SOA started gaining ground due to its distributed architecture approach and it emerged to combat the problems of large monolithic applications, around 2006.

Both (SOA and Microservices) of these architectures share one common thing that they both are distributed architecture and both allow high scalability. In both, service components are accessed remotely through remote access protocol (RMI, REST, SOAP, AMQP, JMS, etc.). both are modular and loosely coupled by design and offer high scalability. Microservices started gaining buzz in late 2000 after the emergence of lightweight containers, Docker, Orchestration Frameworks (Kubernetes, Mesos). Microservices differ from SOA in a significant manner conceptually -

1. SOA uses Enterprise Service Bus for communication, while microservices uses REST or some other less elaborate messaging system (AMQP, etc). Also, microservice follow "Smart endpoints and dumb points", which means that when a microservice needs another one as a dependency, it should use it directly without any routing logic/components handling the pipe.
2. In microservices, service deployment and management should be fully automated, whereas SOA services are often implemented in deployment monoliths.
3. Generally, microservices are significantly smaller than what SOA tends to be. Here we are not talking about the codebase here because few languages are more verbose than the other ones. We are talking about the scope (problem domain) of the service itself. Microservices generally do one thing in a better way.
4. Microservices should own their own data while SOA may share a common database. So one Microservices should not allow another Microservices to change/read its data directly.
5. Classic SOA is more platform-driven, while we have a lot of technology independence in case of microservices. Each microservice can have its own technology stack based on its own functional requirements. So microservices offers more choices in all dimensions.
6. Microservices make an as little assumption as possible on the external environment. A Microservice should manage its own functional domain and data model.

**[9.What is Bounded Context?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3582)**

Bounded Context is a central pattern in Domain-Driven Design. In Bounded Context, everything related to the domain is visible within context internally but opaque to other bounded contexts. DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.

*Monolithic Conceptual Model Problem*

A single conceptual model for the entire organization is very tricky to deal with. The only benefit of such a unified model is that integration is easy across the whole enterprise, but the drawbacks are many, for example:

1. At first, it's very hard to build a single model that works for the entire organization.
2. It's hard for others (teams) to understand it.
3. It's very difficult to change such a shared model to accommodate the new business requirements. The impact of such a change will be widespread across team boundaries.
4. Any large enterprise needs a model that is either very large or abstract.
5. Meaning of a single word may be different in different departments of an organization, so it may be really difficult to come up with a single unified model. Such a model, even if created, will lead to a lot of confusion across the teams.

**[10.Characteristics of a microservices architecture](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3583)**

1. **High Cohesion** - Small and focussed on doing one thing well. Small does not mean less number of lines of code because few programming languages are more verbose than others, but it means the smallest functional area that a single microservices caters to.
2. **Loose Coupling** - Autonomous - the ability to deploy different services independently, and reliability, due to the ability for a service to run even if another service is down.
3. **Bounded Context** - A Microservice serves a bounded context in a domain. It communicates with the rest of the domain by using an interface for that Bounded context.
4. Organisation around business capabilities instead of around technology.
5. Continuous Delivery and Infrastructure automation.
6. Versioning for backward compatibility. Even multiple versions of same microservices can exist in a production environment.
7. **Fault Tolerance** - if one service fails, it will not affect the rest of the system. For example, if a microservices serving the comments and reviews for e-commerce fails, the rest of the website should run fine.
8. Decentralized data management with each service owning its database rather than a single shared database. Every microservice has the freedom to choose the right type of database appropriate for its business use-case (for example, RDBMS for Order Management, NoSql for catalogue management for an e-commerce website)
9. **Eventual Consistency** - event-driven asynchronous updates.
10. **Security** - Every microservice should have the capability to protect its own resources from unauthorized access. This is achieved using stateless security mechanisms like JSON Web Token (JWT pronounced as jot) with OAuth2.

**[11.What are the benefits of using microservices architecture?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3584)**

Embracing microservices architecture brings many benefits compared to using monolith architecture in your application, including:

* **Autonomous Deployments**

The decentralized teams working on individual microservices are mostly independent of each other, so changing a service does not require coordination with other teams. This can lead to significantly faster release cycles. It is very hard to achieve the same thing in a realistic monolithic application where a small change may require regression of the entire system.

* **Culture Shift**

Microservices style of system architecture emphasizes on the culture of freedom, single responsibility, autonomy of teams, faster release iterations and technology diversification.

* **Technology Diversification**

Unlike in monolithic applications, microservices are not bound to one technology stack (Java, .Net, Go, Erlang, Python, etc). Each team is free to choose a technology stack that is best suited for its requirements. For example, we are free to choose Java for a microservice, c++ for others and Go for another one.

* **DevOps Culture**

The term comes from an abbreviated compound of "development" and "operations". It is a culture that emphasizes effective communication and collaboration between product management, software development, and operations team. DevOps culture, if implemented correctly can lead to shorter development cycles and thus faster time to market.

**[12.](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3585)**

**[What is polyglot persistence? Can this idea be used in monolithic applications as well?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3585)**

Polyglot persistence is all about using different databases for different business needs within a single distributed system. We already have different database products in the market each for a specific business need, for example:

* **RDBMS**

Relational databases are used for transactional needs (storing financial data, reporting requirements, etc.)

* **MongoDB**

Documented oriented databases are used for documents oriented needs (for e.g. Product Catalog). Documents are schema-free so changes in the schema can be accommodated into the application without much headache

* **Cassandra/Amazon DynamoDB**

Key-value pair based database (User activity tracking, Analytics, etc.). DynamoDB can store documents as well as key-value pairs.

* **Redis**

In memory distributed database (user session tracking), its mostly used as a distributed cache among multiple microservices.

* **Neo4j**

Graph DB (social connections, recommendations, etc)

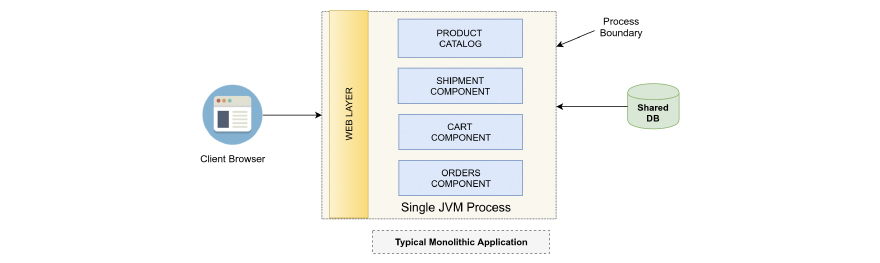
Benefits of *Polyglot Persistence*are manifold and can be harvested in both monolithic as well as microservices architecture. Any decent-sized product will have a variety of needs which may not be fulfilled by a single kind of database alone. For example, if there are no transactional needs for a particular microservice, then it's way better to use a key-value pair or document-oriented NoSql rather than using a transactional RDBMS database

**[13.What are the challenges in microservices?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3587)**

1. DevOps is a must, because of the explosion of a number of processes in a production system. How to start and stop the fleet of services?
2. The complexity of distributed computing such as “network latency, fault tolerance, message serialization, unreliable networks, handling asynchronous o/p, varying loads within our application tiers, distributed transactions, etc.”
3. How to make configuration changes across the large fleet of services with minimal effort?
4. How to deploy multiple versions of single microservice and route calls appropriately?
5. How to disconnect a microservice from ecosystem when it starts to crash unexpectedly?
6. How to isolate a failed microservice and avoid cascading failures in the entire ecosystem?
7. How to discover services in an elastic manner considering that services may be going UP or DOWN at any point in time?
8. How to aggregate logs/metrics across the services? How to identify different steps of a single client request spread across a span of microservices?

**[14.Why Microservices are better than Monoliths?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3588)**

Microservices architecture is meant for developing large distributed systems that scale with safely. There are many benefits of microservices architecture over monoliths, for example:

1. Monolith application is built as a single unit, it is usually composed of 3 components – a database (usually a RDBMS), a server-side executable (war file deployed in tomcat, websphere etc) and a client interface (JSP, etc.)
2. Whenever we want to add/update functionality, developers need to change at least one of these three components and deploy the new version to production. The entire system is tightly coupled, have limitations in choosing technology stack, have low cohesion.
3. When we need to scale a monolith, we deploy the same version of the monolith on multiple machines, by copying the big war/ear file again and again. Everything is contained into a single executable file.
4. Microservices Architecture, on the other hand, is composed of small autonomous services, divided over business capabilities that communicate with each other over network mostly in async fashion.
5. 
6. As illustrated in the above example, a typical monolith eShop application is usually a big war file deployed in a single JVM process (tomcat/jboss/websphere, etc). Different components of a monolith communicate with each other using in-process communication like direct method invocation. One or more databases are shared among different components of a monolith application.

**[16.](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3589)**

**[How to convert the large application into microservices architecture?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3589)**

Microservices should be autonomous and divided based on business capabilities. Each software component should have single well-defined responsibility (a.k.a Single Responsibility Principle) and the principle of Bounded Context (as defined by Domain Driven Design) should be used to create highly cohesive software components.

For example, an e-commerce site can be partitioned into following microservices based on its business capabilities:

* **Product catalogue**

Responsible for product information, searching products, filtering products & products facets.

* **Inventory**

Responsible for managing inventory of products (stock/quantity and facet).

* **Product review and feedback**

Collecting feedback from users about the product

* **Orders**

Responsible for creating and managing orders.

* **Payments**

Process payments both online and offline (Cash On Delivery).

* **Shipments**

Manage and track shipments against orders.

* **Demand generation**

Market products to relevant users.

* **User Accounts**

Manage users and their preferences.

* **Recommendations**

Suggest new products based on the user’s preference or past purchases.

* **Notifications**

Email and SMS notification about orders, payments, and shipments.

The client application (browser, mobile app) will interact with these services via API gateway and render the relevant information to the user.

#### [17.](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3590)

**[How to stop a Spring Boot based microservices at startup if it can not connect to the Config server during bootstrap?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3590)**

If you want to halt the service when it is not able to locate the config-server during bootstrap, then you need to configure the following property in microservice’s bootstrap.yml:

spring:

      cloud:

         config:

             fail-fast: **true**

Using this configuration will make microservice startup fail with an exception when config-server is not reachable during bootstrap.

We can enable a retry mechanism where microservice will retry 6 times before throwing an exception. We just need to add spring-retry and spring-boot-starter-aop to the classpath to enable this feature.

build.gradle:-

...

 dependencies {

   compile('org.springframework.boot:spring-boot-starter-aop')

   compile('org.springframework.retry:spring-retry')

   ...

}

**[18.](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3591)**

**[How to partition a large application into microservices architecture?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3591)**

Microservices should be autonomous and divided based on business capabilities. Each software component should have single well-defined responsibility (a.k.a Single Responsibility Principle) and the principle of Bounded Context (as defined by Domain Driven Design) should be used to create highly cohesive software components.

For example, an e-shop can be partitioned into following microservices based on its business capabilities:

* **Product catalogue**

Responsible for product information, searching products, filtering products & products facets.

* **Inventory**

Responsible for managing inventory of products (stock/quantity and facet).

* **Product review and feedback**

Collecting feedback from users about the products

* **Orders**

Responsible for creating and managing orders.

* **Payments**

Process payments both online and offline (Cash On Delivery).

* **Shipments**

Manage and track shipments against orders.

* **Demand generation**

Market products to relevant users.

* **User Accounts**

Manage users and their preferences.

* **Recommendations**

Suggest new products based on the user’s preference or past purchases.

* **Notifications**

Email and SMS notification about orders, payments, and shipments.  
The client application (browser, mobile app) will interact with these services via the API gateway and render the relevant information to the user.

#### [19.](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3592)

**[How big a single microservice should be?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3592)**

A good, albeit non-specific, rule of thumb is as small as possible but as big as necessary to represent the domain concept they own said by Martin Fowler

Size should not be a determining factor in microservices, instead bounded context principle and single responsibility principle should be used to isolate a business capability into a single microservice boundary.

Microservices are usually small but not all small services are microservices. If any service is not following the Bounded Context Principle, Single Responsibility Principle, etc. then it is not a microservice irrespective of its size. So the size is not the only eligibility criteria for a service to become microservice.

In fact, size of a microservice is largely dependent on the language (Java, Scala, PHP) you choose, as few languages are more verbose than others.

**[20.](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3593)**

**[How do microservices communicate with each other?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3593)**

Microservices are often integrated using a simple protocol like REST over HTTP. Other communication protocols can also be used for integration like AMQP, JMS, Kafka, etc.

The communication protocol can be broadly divided into two categories- synchronous communication and asynchronous communication.

* **Synchronous Communication**

RestTemplate, WebClient, FeignClient can be used for synchronous communication between two microservices. Ideally, we should minimize the number of synchronous calls between microservices because networks are brittle and they introduce latency. Ribbon - a client-side load balancer can be used for better utilization of resource on the top of RestTemplate. Hystrix circuit breaker can be used to handle partial failures gracefully without a cascading effect on the entire ecosystem. Distributed commits should be avoided at any cost, instead, we shall opt for eventual consistency using asynchronous communication.

* **Asynchronous Communication**

In this type of communication, the client does not wait for a response, instead, it just sends the message to the message broker. AMQP (like RabbitMQ) or Kafka can be used for asynchronous communication across microservices to achieve eventual consistency

**[21.](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3594)**

**[What should be preferred communication style in microservices: synchronous or asynchronous?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3594)**

1. You must use asynchronous communication while handling HTTP POST/PUT (anything that modifies the data) requests, using some reliable queue mechanism (RabbitMQ, AMQP, etc.)
2. It's fine to use synchronous communication for Aggregation pattern at API Gateway Level. But this aggregation should not include any business logic other than aggregation. Data values must not be transformed at Aggregator, otherwise, it defeats the purpose of Bounded Context. In Asynchronous communication, events should be published into a Queue. Events contain data about the domain, it should not tell what to do (action) on this data.
3. If microservice to microservice communication still requires synchronous communication for GET operation, then seriously reconsider the partitioning of your microservices for bounded context, and create some tasks in backlog/technical debt.

**[What is the difference between Orchestration and Choreography in microservices context?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3595)**

In Orchestration, we rely on a central system to control and call other Microservices in a certain fashion to complete a given task. The central system maintains the state of each step and sequence of the overall workflow. In Choreography, each Microservice works like a State Machine and reacts based on the input from other parts. Each service knows how to react to different events from other systems. There is no central command in this case.

Orchestration is a tightly coupled approach and is an anti-pattern in a microservices architecture. Whereas, Choreography’s loose coupling approach should be adopted where-ever possible.

***Example***

Let’s say we want to develop a microservice that will send product recommendation email in a fictitious e-shop. In order to send Recommendations, we need to have access to user’s order history which lies in a different microservices.

In Orchestration approach, this new microservice for recommendations will make synchronous calls to order service and fetch the relevant data, then based on his past purchases we will calculate the recommendations. Doing this for a million users will become cumbersome and will tightly couple the two microservices.

In Choreography approach, we will use event-based Asynchronous communication where whenever a user makes a purchase, an event will be published by order service. Recommendation service will listen to this event and start building user recommendation. This is a loosely coupled approach and highly scalable. The event, in this case, does not tell about the action, but just the data

**[23.](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3596)**

**[How frequent a microservice be released into production?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3596)**

There is no right answer to this question, there could be a release every ten minutes, every hour or once a week. It all depends on the extent of automation you have at a different level of the software development lifecycle - build automation, test automation, deployment automation and monitoring. And of course on the business requirements - how small low-risk changes you care making in a single release.

In an ideal world where boundaries of each microservices are clearly defined (bounded context), and a given service is not affecting other microservices, you can easily achieve multiple deployments a day without major complexity.

**Examples of deployment/release frequency**

1. Amazon is on record as making changes to production every 11.6 seconds on average in May of 2011.
2. Github is well known for its aggressive engineering practices, deploying code into production on an average 60 times a day.
3. Facebook releases to production twice a day.
4. Many Google services see releases multiple times a week, and almost everything in Google is developed on mainline.
5. Etsy Deploys More Than 50 Times a Day.

**[24.](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3597)**

**[What are Cloud-Native applications?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3597)**

Cloud-Native Applications (NCA) is a style of application development that encourages easy adoption of best practices in the area of continuous delivery and distributed software development. These applications are designed specifically for a cloud computing architecture (AWS, Azure, CloudFoundary, etc).

DevOps, continuous delivery, microservices, and containers are the key concepts in developing cloud-native applications.

Spring Boot, Spring Cloud, Docker, Jenkins, Git are a few tools that can help you write Cloud-Native Application without much effort.

* **Microservices**

It is an architectural approach for developing a distributed system as a collection of small services. Each service is responsible for a specific business capability, runs in its own process and communicates via HTTP REST API or messaging (AMQP).

* **DevOps**

It is a collaboration between software developers and IT operations with a goal of constantly delivering high-quality software as per customer needs.

* **Continuous Delivery**

Its all about automated delivery of low-risk small changes to production, constantly. This makes it possible to collect feedback faster.

* **Containers**

Containers (e.g. Docker) offer logical isolation to each microservices thereby eliminating the problem of "run on my machine" forever. It’s much faster and efficient compared to Virtual Machines.

#### [25.](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3598)

**[How will you develop microservices using Java?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3598)**

Spring Boot along with Spring Cloud is a very good option to start building microservices using Java language. There are a lot of modules available in Spring Cloud that can provide boiler plate code for different design patterns of microservices, so Spring Cloud can really speed up the development process. Also, Spring boot provides out of the box support to embed a servlet container (tomcat/jetty/undertow) inside an executable jar (uber jar), so that these jars can be run directly from the command line, eliminating the need of deploying war files into a servlet container.

You can also use Docker container to ship and deploy the entire executable package onto a cloud environment. Docker can also help eliminate "works on my machine" problem by providing logical separation for the runtime environment during the development phase. That way you can gain portability across on-premises and cloud environment.

**[27.](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3600)**

**[What is API Gateway?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3600)**

API Gateway is a special class of microservices that meets the need of a single client application (such as android app, web app, angular JS app, iPhone app, etc) and provide it with single entry point to the backend resources (microservices), providing cross-cutting concerns to them such as security, monitoring/metrics & resiliency.

Client Application can access tens or hundreds of microservices concurrently with each request, aggregating the response and transforming them to meet the client application’s needs. Api Gateway can use a client-side load balancer library (Ribbon) to distribute load across instances based on round-robin fashion. It can also do protocol translation i.e. HTTP to AMQP if necessary. It can handle security for protected resources as well.

Features of API Gateway

1. Spring Cloud DiscoveryClient integration
2. Request Rate Limiting (available in Spring Boot 2.x)
3. Path Rewriting
4. Hystrix Circuit Breaker integration for resiliency

**[28.](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3601)**

**[How to achieve zero-downtime during the deployments?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3601)**

As the name suggests, zero-downtime deployments do not bring outage in a production environment. It is a clever way of deploying your changes to production, where at any given point in time, at least one service will remain available to customers.

* **Blue-green deployment**

One way of achieving this is blue/green deployment. In this approach, two versions of a single microservice are deployed at a time. But only one version is taking real requests. Once the newer version is tested to the required satisfaction level, you can switch from older version to newer version.

You can run a smoke-test suite to verify that the functionality is running correctly in the newly deployed version. Based on the results of smoke-test, newer version can be released to become the live version.

* **Changes required in client code to handle zero-downtime**

Lets say you have two instances of a service running at the same time, and both are registered in Eureka registry. Further, both instances are deployed using two distinct hostnames:

* **Changes required in client code to handle zero-downtime**

Lets say you have two instances of a service running at the same time, and both are registered in Eureka registry. Further, both instances are deployed using two distinct hostnames:

*/src/main/resources/application.yml*

  spring.application.name: ticketBooks-service

  ---

  spring.profiles: blue

eureka.instance.hostname: ticketBooks-service -blue.example.com

  ---

  spring.profiles: green

  eureka.instance.hostname: ticketBooks-service -green.example.com

Now the client app that needs to make api calls to books-service may look like below:

Now the client app that needs to make api calls to books-service may look like below:

@RestController

@SpringBootApplication

@EnableDiscoveryClient

 public class ClientApp {

@Bean

@LoadBalanced

public RestTemplate restTemplate() {

return new RestTemplate(); }

@RequestMapping("/hit-some-api")

public Object hitSomeApi() {

return restTemplate().getForObject("https://ticketBooks-service/some-uri", Object.class);  }

Now, when ticketBooks-service-green.example.com goes down for upgrade, it gracefully shuts down and delete its entry from Eureka registry. But these changes will not be reflected in the ClientApp until it fetches the registry again (which happens every 30 seconds). So for upto 30 seconds, ClientApp’s @LoadBalanced RestTemplate may send the requests to ticketBooks-service-green.example.com even if its down.

To fix this, we can use Spring Retry support in Ribbon client-side load balancer. To enable Spring Retry, we need to follow the below steps:

*Add spring-retry to build.gradle dependencies*

compile("org.springframework.boot:spring-boot-starter-aop")

compile("org.springframework.retry:spring-retry")

Now enable spring-retry mechanism in ClientApp using @EnableRetry annotation, as shown below:

To fix this, we can use Spring Retry support in Ribbon client-side load balancer. To enable Spring Retry, we need to follow the below steps:

*Add spring-retry to build.gradle dependencies*

compile("org.springframework.boot:spring-boot-starter-aop")

compile("org.springframework.retry:spring-retry")

Now enable spring-retry mechanism in ClientApp using @EnableRetry annotation, as shown below:

@EnableRetry @RestController @SpringBootApplication @EnableDiscoveryClient public class ClientApp {

... }

Once this is done, Ribbon will automatically configure itself to use retry logic and any failed request to ticketBooks-service-green.example.com com will be retried to next available instance (in round-robins fashion) by Ribbon. You can customize this behaviour using the below properties:

*/src/main/resources/application.yml*

ribbon:

MaxAutoRetries: 5

MaxAutoRetriesNextServer: 5

OkToRetryOnAllOperations: true

OkToRetryOnAllErrors: true

**[29.](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3602)**

**[How to achieve zero-downtime deployment(blue/green) when there is a database change?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3602)**

The deployment scenario becomes complex when there are database changes during the upgrade. There can be two different scenarios: 1. database change is backward compatible (e.g. adding a new table column) 2. database change is not compatible with an older version of the application (e.g. renaming an existing table column)

1. **Backward compatible change**: This scenario is easy to implement and can be fully automated using Flyway. We can add the script to create a new column and the script will be executed at the time of deployment. Now during blue/green deployment, two versions of the application (say v1 and v2) will be connected to the same database. We need to make sure that the newly added columns allow null values (btw that’s part of the backward compatible change). If everything goes well, then we can switch off the older version v1, else application v2 can be taken off.
2. **Non-compatible database change**: This is a tricky scenario, and may require manual intervention in-case of rollback. Let's say we want to rename first\_name column to fname in the database. Instead of directly renaming, we can create a new column fname and copy all existing values of first\_name into fname column, keeping the first\_name column as it is in the database. We can defer non-null checks on fname to post-deployment success. If the deployment goes successful, we need to migrate data written to first\_name by v1 to the new column (fname) manually after bringing down the v1. If the deployment fails for v2, then we need to do the otherwi

**[30.](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3603)**

**[How to maintain ACID in microservice architecture?](https://www.knowledgehut.com/interview-questions/microservices" \l "collapse-beginner-3603)**

ACID is an acronym for four primary attributes namely atomicity, consistency, isolation, and durability ensured by the database transaction manager.

* **Atomicity**

In a transaction involving two or more entities, either all of the records are committed or none are.

* **Consistency**

A database transaction must change affected data only in allowed ways following specific rules including constraints/triggers etc.

* **Isolation**

Any transaction in progress (not yet committed) must remain isolated from any other transaction.

* **Durability**

Committed records are saved by a database such that even in case of a failure or database restart, the data is available in its correct state. Any transaction in progress (not yet committed) must remain isolated from any other transaction.

* **Durability**

Committed records are saved by a database such that even in case of a failure or database restart, the data is available in its correct state.

In a distributed system involving multiple databases, we have two options to achieve ACID compliance:

1. One way to achieve ACID compliance is to use a two-phase commit (a.k.a 2PC), which ensures that all involved services must commit to transaction completion or all the transactions are rolled back.
2. Use eventual consistency, where multiple databases owned by different microservices become eventually consistent using asynchronous messaging using messaging protocol. Eventual consistency is a specific form of weak consistency.

2 Phase Commit should ideally be discouraged in microservices architecture due to its fragile and complex nature. We can achieve some level of ACID compliance in distributed systems through eventual consistency and that should be the right approach to do it.