

★ Get unlimited access to the best of Medium for less than \$1/week. [Become a member](#)



Javarevisited · [Follow publication](#)

Scenario-based solutions for common challenges in Microservices architecture

11 min read · May 5, 2023



Backend developer

[Follow](#)



Listen



Share



More



Photo by [GR Stocks](#) on [Unsplash](#)

I have gathered ten essential scenario-based microservices questions that individuals typically encounter while designing or debugging their microservice architecture. Although these questions may not cover every possible scenario, they provide significant insights into common issues faced during microservices development.

What measures can be taken to guarantee the scalability and resilience of Microservices?

To ensure that microservices are scalable and resilient, consider the following best practices:

- **Design for scalability:** Build microservices with scalability in mind from the beginning. Use a modular architecture that allows services to be broken down into smaller components that can be scaled independently.
- **Use containerization and orchestration:** Use containerization technologies like Docker to package and deploy microservices. Use orchestration tools like Kubernetes to manage and scale containerized services.
- **Implement fault tolerance:** Design your microservices to handle errors and failures gracefully. Implement retry mechanisms, timeouts, and circuit breakers to ensure that services continue to function even when other services fail.
- **Use monitoring and logging:** Implement monitoring and logging tools to track the health and performance of microservices. Use this data to identify bottlenecks and optimize performance.
- **Use load balancing:** Implement load balancing to distribute traffic evenly across multiple instances of a service. Use auto-scaling to automatically adjust the number of instances based on traffic levels.
- **Implement caching:** Implement caching to reduce the load on backend services and improve response times.
- **Use asynchronous messaging:** Use asynchronous messaging patterns to decouple services and improve scalability. Implement event-driven architectures using message queues or publish-subscribe systems.

By following these best practices, you can ensure that your microservices are scalable, resilient, and able to handle high volumes of traffic and data without compromising performance or reliability.

What is the recommended approach if two Microservices need to update the same database?

- **Use a distributed transaction coordinator :** A distributed transaction coordinator can be used to coordinate transactions across multiple services and ensure that updates are performed atomically. A distributed transaction coordinator like Apache Kafka or Apache Zookeeper can ensure that all changes are committed or rolled back together, thus maintaining consistency across services.
- **Implement optimistic locking :** Optimistic locking is a technique in which a version number is attached to a record in the database. When a service updates the record, it increments the version number. If another service attempts to update the same record concurrently, it checks the version number and rejects the update if the version number has changed. This technique can prevent conflicts and ensure consistency.
- **Use event-driven architecture :** In an event-driven architecture, microservices communicate with each other by publishing events to a message broker. Other

[Open in app](#) ↗

Medium



Search



- **Implement retry and error handling :** When multiple services are updating the same database, it is important to implement retry and error handling mechanisms to ensure that failed updates are retried and that errors are handled appropriately. This can help to prevent data inconsistencies and ensure that updates are eventually successful.

By following these best practices, it is possible to ensure that updates to a shared database are performed in a consistent and reliable manner, even when multiple microservices are involved.

What might be causing the delay in startup time for a Microservice with a large database?

There could be several reasons why a microservice is taking a long time to come up due to a large database:

- **Database schema** : If the database schema is complex and includes many tables, columns, and relationships, it can take a long time for the microservice to initialize and establish a connection to the database.
- **Data volume** : If the database contains a large volume of data, it can take a long time for the microservice to load and cache the data. This can also slow down database queries and other operations.
- **Network latency** : If the microservice and the database are located on different servers or in different data centers, network latency can cause delays in establishing a connection and transferring data between the two.
- **Hardware limitations** : If the hardware used to run the microservice or the database is not powerful enough, it can cause performance issues and slow down startup times.

To address these issues, you could consider:

- Optimizing the database schema by reducing the number of tables, columns, and relationships where possible.
- Implementing pagination or other techniques to limit the amount of data loaded at startup, or using asynchronous loading to load data in the background.
- Moving the microservice and the database to the same server or data center to reduce network latency.
- Upgrading the hardware used to run the microservice and the database to improve performance and reduce startup times.
- Using database connection pooling and other optimization techniques to improve database connection times and query performance.
- Analyzing and optimizing database queries to improve performance and reduce startup times.

If Microservice A communicates with Microservice B, which then communicates with Microservice C, and an exception is thrown by Microservice C, how should the exception be handled?

When Microservice C throws an exception, Microservice B should handle it and return an appropriate response to Microservice A. The exception should be propagated up the call chain from Microservice C to Microservice B, and then to Microservice A.

To handle the exception in Microservice B, you can use a try-catch block or an exception handler. If Microservice C returns an HTTP response with an error code, such as 4xx or 5xx, Microservice B can catch the exception and either rethrow it or wrap it in a new exception with more context information.

For example, if Microservice C returns a 404 Not Found error, Microservice B can catch the exception and wrap it in a new exception with a more descriptive message, such as “Resource not found in Microservice C”. This new exception can then be propagated up to Microservice A along with the appropriate HTTP response code.

It is important to handle exceptions properly in Microservices architecture, as it can impact the overall performance and stability of the system. You should also consider implementing retry mechanisms or fallback strategies in case of exceptions to ensure the system can recover from failures.

Is it necessary for Microservice A to poll Microservice B every time to get the required information, or is there an alternative solution to retrieve only specific parameters?

Instead of polling Microservice B every time to get the information, Microservice A can use a request-response pattern to request only the required parameters from Microservice B. This can be achieved by implementing an API endpoint on Microservice B that returns only the required parameters.

One possible approach is to use a REST API endpoint that accepts the parameters as query parameters or path variables. Microservice A can then make a request to this endpoint to retrieve only the required parameters.

Another approach is to use a message broker or event-driven architecture, where Microservice B publishes events containing the required information, and Microservice A subscribes to these events to retrieve the required parameters. This approach can provide better scalability and performance, as Microservice A doesn't need to poll Microservice B for information.

In both cases, it is important to ensure proper authentication and authorization mechanisms are in place to ensure that only authorized requests are accepted and processed. Additionally, proper error handling and fault tolerance mechanisms should be implemented to handle failures and ensure system reliability.

If I have a cron job in my application, and it is deployed on multiple instances, will the cron job run simultaneously on all instances?

If your application is deployed on multiple instances, each instance will have its own copy of the cron job. Therefore, if the cron job is scheduled to run at a specific time, each instance will independently execute the cron job at that time.

However, if your cron job relies on shared resources or state, running it concurrently on multiple instances could lead to conflicts and inconsistent results. To avoid this, you can use a distributed locking mechanism to ensure that the cron job is executed by only one instance at a time. Alternatively, you can configure your deployment to run the cron job on a single instance only, such as by using Kubernetes' job or singleton deployment patterns.

Explain Circuit Breaker pattern, its application in Microservices architecture to handle service failures, and the issues it addresses?

Let's understand the Circuit Breaker pattern with an example:

Let's say we have a microservice that's responsible for processing payments. Whenever a user wants to make a payment, they send a request to the payment microservice. The payment microservice communicates with the user service to get information about the user making the payment and the account service to retrieve the account information. Once all the information is gathered, the payment microservice processes the payment and sends a response back to the user.

However, one day the user service is experiencing high traffic, and it slows down. As a result, the payment microservice also slows down since it's waiting for a response from the user service. If the payment microservice doesn't handle this properly, it could start queuing up requests and eventually run out of resources, leading to a service failure.

This is where the Circuit Breaker pattern comes in. The Circuit Breaker pattern can be used to detect when a service is failing or not responding and take appropriate action. In this example, the Circuit Breaker pattern would be implemented in the payment microservice, and it would monitor the response times of the user service. If the response times exceed a certain threshold, the Circuit Breaker would trip and stop sending requests to the user service. Instead, it would return an error message to the user or try to fulfill the request using a cached response or a fallback service.

Once the user service has recovered and response times have improved, the Circuit Breaker would close and start sending requests to the user service again.

In this way, the Circuit Breaker pattern helps to handle service failures in a Microservices architecture and prevent cascading failures by isolating the failing service and protecting the system from further degradation.

What is Command Query Responsibility Segregation (CQRS) pattern and when is it appropriate to use in Microservices architecture? Explain with an example.

Command Query Responsibility Segregation (CQRS) is a design pattern that separates the operations that read data from those that write data in a microservices architecture. It proposes that commands, which modify data, should be separated from queries, which retrieve data. This separation allows for optimized processing and scalability of each operation, as they have different performance and scaling requirements.

CQRS is appropriate to use when dealing with complex data models or high-performance systems, where the query and write patterns are different, and the system requires a highly responsive and scalable architecture. It also enables the creation of different models for read and write operations, allowing each to evolve independently.

For example, consider a system that manages e-commerce orders. The write operations, such as placing an order or canceling an order, require high consistency and reliability. On the other hand, read operations, such as fetching a customer's order history or product inventory, are more frequent and require high performance.

With CQRS, the write operations can be handled by a separate service that ensures data consistency and reliability. Meanwhile, read operations can be handled by a separate service that optimizes for high performance, such as caching frequently

accessed data or using precomputed views. This separation allows for scalability, performance optimization, and evolution of each service independently.

How can service deployment and rollback be managed in a microservices architecture?

In a microservices architecture, service deployment and rollback require careful planning and execution to ensure smooth and efficient operations. Here are some key considerations:

- **Containerization:** Containerization is an important step in service deployment and rollback. By using containers, you can package your microservices into a single image, including all dependencies and configurations. This makes it easier to deploy and rollback services.
- **Version Control:** It is essential to maintain version control of all microservices. This will help in identifying the differences between the current and previous version and will help to rollback the changes if necessary.
- **Blue-Green Deployment:** This approach involves deploying a new version of the microservice alongside the old version, testing it, and then routing traffic to the new version once it has been verified. If any issues arise, traffic can be easily rerouted back to the previous version.
- **Canary Deployment:** In this approach, a small percentage of users are routed to the new version of the service, while the rest are still using the old version. This allows for gradual testing and identification of any issues before a full rollout is done.
- **Automated Testing:** Automated testing is an important part of service deployment and rollback. Unit, integration, and end-to-end tests should be performed to ensure that the microservice is functioning as expected.
- **Monitoring and Logging:** Monitoring and logging play a critical role in identifying issues with microservices. Logs and metrics should be collected and analyzed in real-time to detect any anomalies or failures.
- **Rollback Plan:** A rollback plan should be in place in case of any issues with the new version of the microservice. This plan should include steps for rolling back

to the previous version, testing it, and identifying the root cause of the issue before attempting another deployment.

By following these best practices, you can ensure smooth service deployment and rollback in a microservices architecture.

How can Blue-Green Deployment be implemented in OpenShift?

Blue-Green deployment is a deployment strategy that reduces downtime and risk by deploying a new version of an application alongside the current version, then switching traffic over to the new version only after it has been fully tested and verified to be working correctly. OpenShift provides built-in support for blue-green deployments through its routing and deployment features. Here's how you can implement blue-green deployment in OpenShift:

- 1. Create two identical deployments:** Start by creating two identical deployments in OpenShift, one for the current version (blue) and one for the new version (green).
- 2. Configure route:** Next, create a route that points to the blue deployment so that incoming traffic is directed to it.
- 3. Test the green deployment:** Deploy the new version (green) alongside the current version (blue), but do not make it publicly available yet. Test the new deployment thoroughly, to ensure that it is working correctly.
- 4. Update the route:** Once the new deployment (green) has been tested and verified, update the route to point to the green deployment.
- 5. Monitor the deployment:** Monitor the new deployment (green) closely, to ensure that it is working correctly and that there are no issues.
- 6. Rollback if necessary:** If any issues are detected, or if the new deployment (green) is not performing as expected, roll back the deployment by updating the route to point back to the blue deployment.


Here's an example of how you can use the OpenShift CLI to perform a blue-green deployment:

1. Create two deployments:

```
oc new-app my-image:v1 --name=my-app-blue  
oc new-app my-image:v2 --name=my-app-green
```

2. Create a route that points to the blue deployment:

```
oc expose service my-app-blue --name=my-app-route --hostname=my-app.example.com
```



3. Test the green deployment:

```
oc patch route my-app-route -p '{"spec":{"to":{"name":"my-app-green"}}}'
```

4. Update the route:

```
oc patch route my-app-route -p '{"spec":{"to":{"name":"my-app-blue"}}}'
```

5. Monitor the deployment.

The entire blue-green deployment process can also be automated using OpenShift templates and scripts to ensure consistency and reduce errors.

This is not an all-encompassing compilation. The following are just some introductory questions that any backend developer should be familiar with. I will endeavor to obtain more complex questions related to these topics. Keep an eye out!

Backend Development



Follow

Published in Javarevisited

42K followers · Last published 10 hours ago

A humble place to learn Java and Programming better.



Follow

Written by Backend developer

335 followers · 60 following

No responses yet



Anupriya

What are your thoughts?

More from Backend developer and Javarevisited



In Geek Culture by Backend developer

The Eight Fallacies of Distributed Computing

Thanks to modern computing, the 8 fallacies of distributed computing are being rendered obsolete

Sep 30, 2021 🖱️ 74 💬 1



In Javarevisited by Mohit Bajaj


How I Optimized a Spring Boot Application to Handle 1M Requests/Second 🚀

Discover the exact techniques I used to scale a Spring Boot application from handling 50K to 1M requests per second. I'll share the...

 Mar 2  2.3K  64


	Before Compression	After Con
	~1.2 MB	~180 KB
avg)	1.3s	380ms
	Laggy	Instant
	Sluggish	Snappy 🚀

 In Javarevisited by Meena Jadhav

This Spring Boot Trick Made My REST API 3x Faster (No Code Rewrite Required)

What if I told you there's a one-line Spring Boot trick that can make your API significantly faster —without touching your code logic?

 May 5  329  5



 In CodeX by Backend developer

Client-Server Architecture

This article is related to my previous post which was about Architectures in Distributed System. Through this article, I would like to...

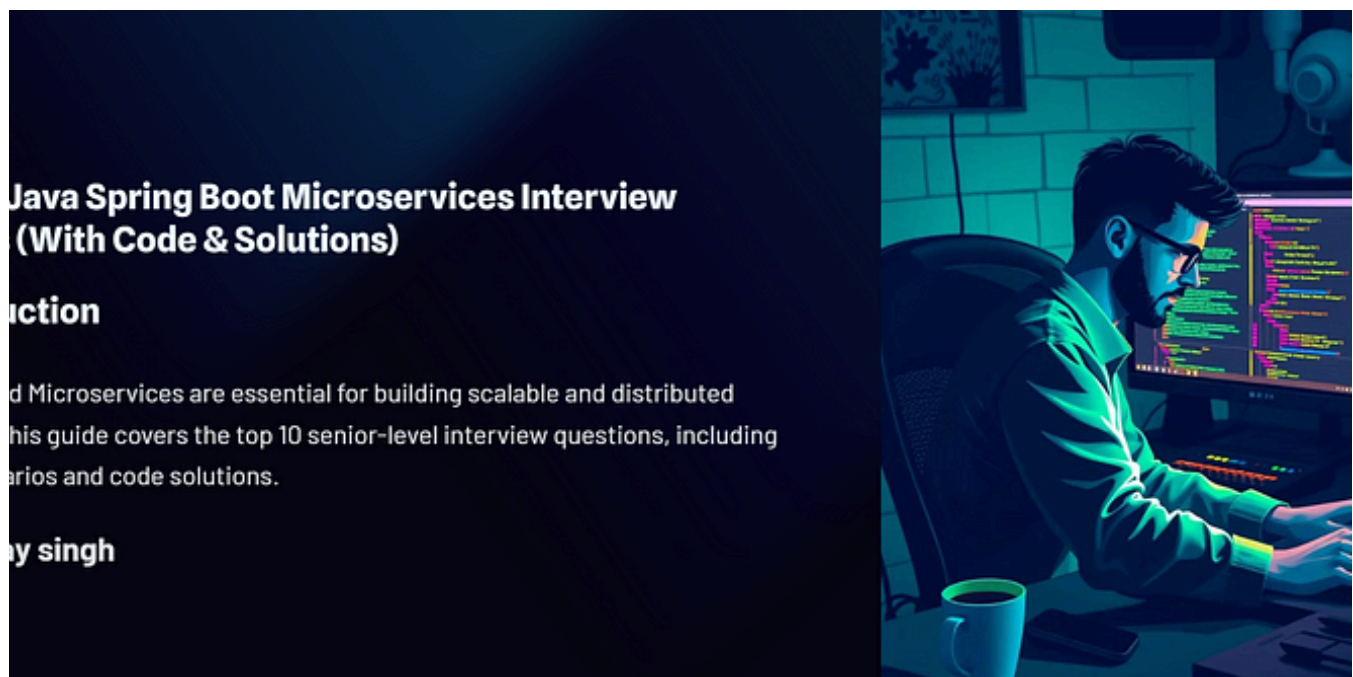
Aug 11, 2021  52



See all from Backend developer

See all from Javarevisited

Recommended from Medium



Sanjay Singh

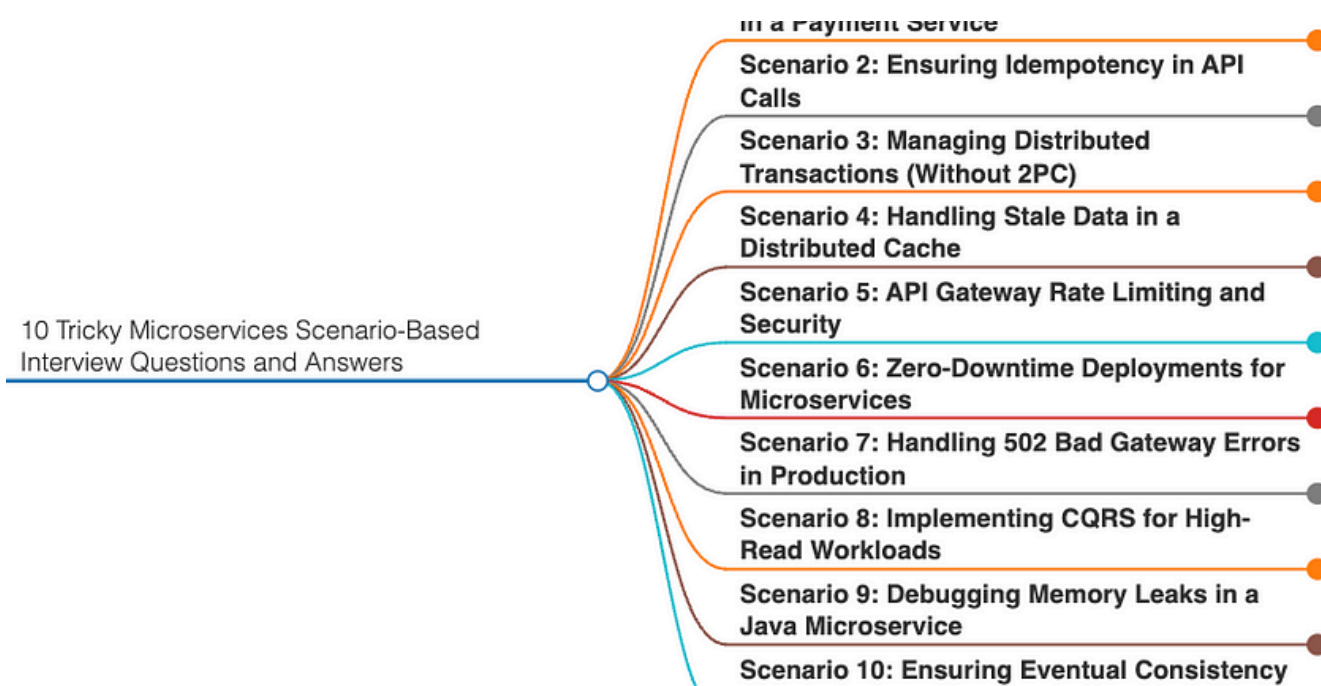


Top 10 Java Spring Boot Microservices Interview Questions (With Code & Solutions)



Introduction

★ Feb 25 🖱 54



Arvind Kumar

10 Tricky Microservices Scenario-Based Interview Questions and Answers

Microservices architecture introduces complex scalability, resilience, data consistency, and observability challenges. Below are 10...

Mar 12 🖱 91 💬 1





In Stackademic by Lets Learn Now

Ace Your Interview: Must-Know Java, Microservices, AWS & System Design Questions

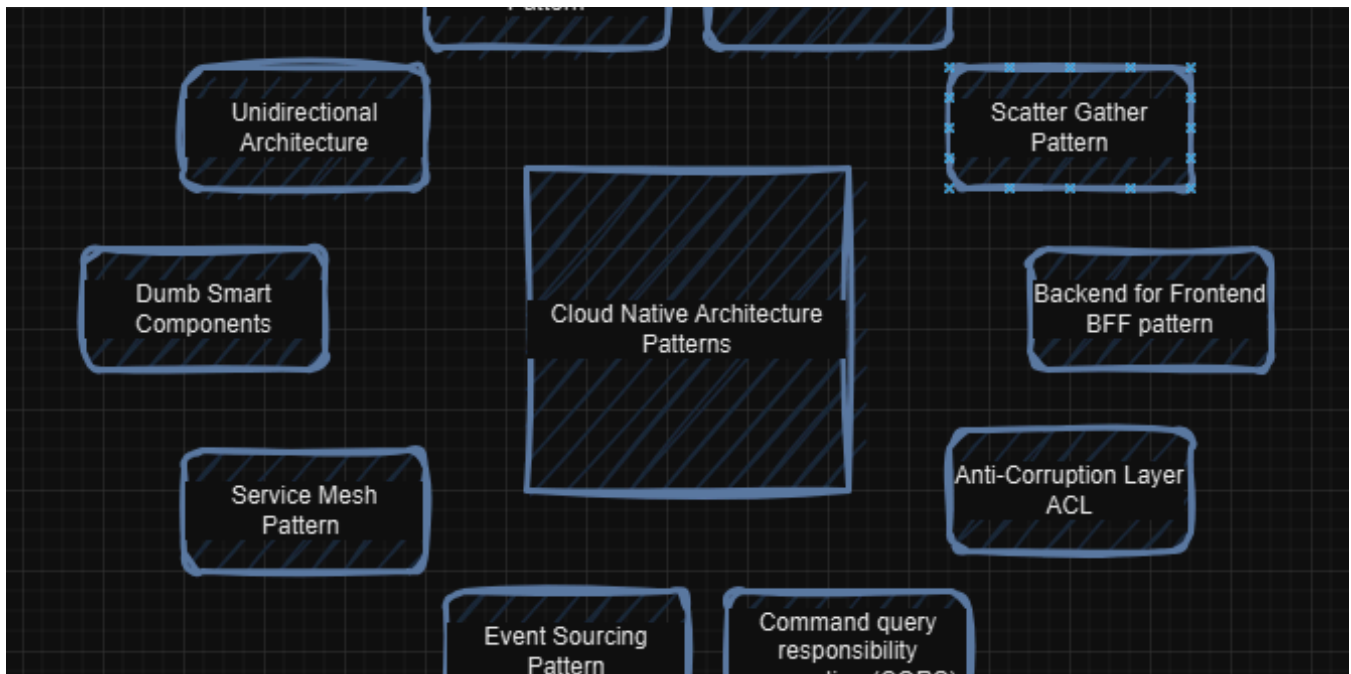
Key features of Java that make it suitable for large-scale applications



May 14



32



Archana Goyal

10 Must-Know Cloud Native Architecture Patterns

Modern cloud-native applications require scalable, resilient, and modular architectures. These architecture patterns help in designing...


Feb 7



11



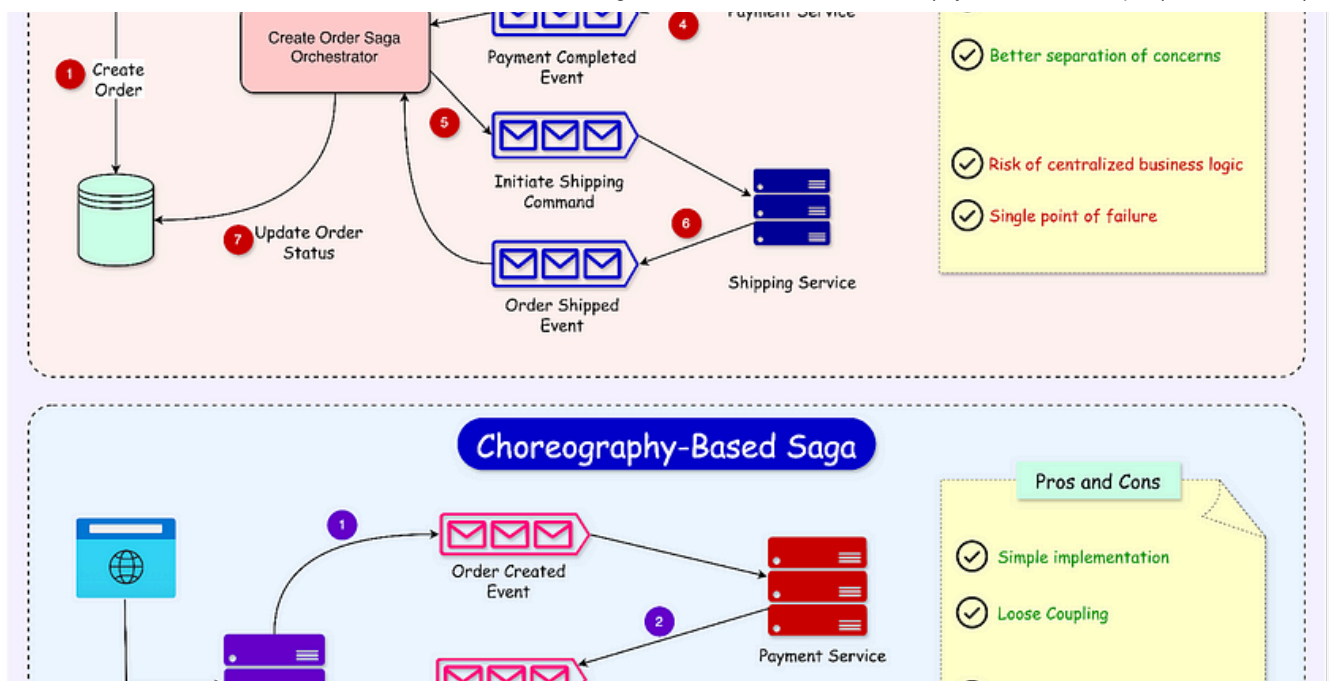


 Zudonu Osomudeya

50 Scenario-Based Interview Questions and Answers

★ Apr 4 🖱️ 17 💬 1





 Ajay Rathod

Top 10 Microservice Interview Questions Most Likely to Appear in an Interviews in 2025

These are top 10 Microservice interview questions mostly likely to appear in an interview.

★ Dec 23, 2024 🖱️ 170 💬 5



See more recommendations