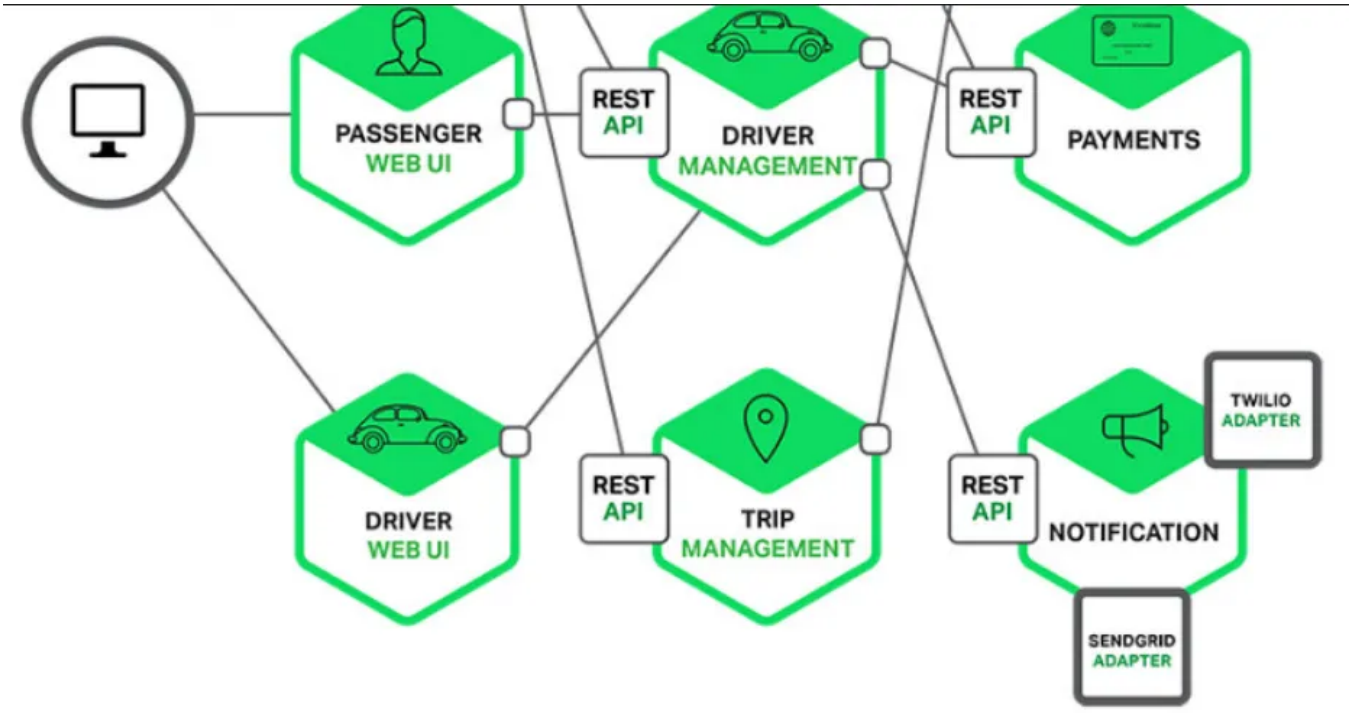
Member-only story

10 Things to Keep in Mind while Designing and Developing Microservices

Best practices and considerations for building effective microservices architecture

Soma · Follow
Published in Javarevisited
14 min read · Apr 1



Hello guys, In last a couple of articles, I have shared both *Microservice architecture interview questions* as well as *Microservice scenario based questions* and today, I am going to talk about a general question about Microservice design and development.

"*What are the things you should keep in mind while designing and developing Microservices?"?* This is one of the most common question I have seen during Tech interviews about System design and Microservices.

When I first saw this question I wasn't sure how to answer it, should I mention about keeping it as small as possible given the name is Microservices or keeping a limit of how many different services we should have? or using which programming language to develop Microservice etc.

But as my experience and knowledge grew, I find better way to answer this question which I am going to share in this article with you. Btw, the discussion is not just important for interview point of view but also for designing and developing Microservices which can withstand test of production.

In recent years, Microservices architecture has become a popular approach to developing scalable and flexible applications.

> *However,* ***implementing Microservices is not as simple as breaking a monolithic application into smaller services.*** *It requires a significant shift in the way we design, develop, and deploy applications.*

In this article, we will discuss the key things you should keep in mind while developing microservices. We'll cover everything from the importance of a good service boundary to the challenges of distributed systems like managing transactions and ensuring data consistency.

Whether you're just starting with Microservices or looking to improve your existing architecture, this guide will provide you with valuable insights to ensure you build reliable, scalable, and maintainable microservices.

Here are 10 things to keep in mind while developing Microservices:

1. **Design your services around business capabilities, not technical capabilities**
2. **Define the boundaries of your microservices**
3. **Design for resiliency**
4. **Use lightweight communication protocols**
5. **Use asynchronous communication between services**
6. **Ensure data consistency between services**
7. **Implement security measures**

8. **Ensure scalability and performance**
9. **Implement monitoring and logging**
10. **Plan for testing and deployment**
11. **Choose the right technology stack**

Now that we know the key points to remember while designing and developing Microservices, let's deep dive into each of them to what they mean and how you can achieve them in your next Microservices app.

And if you haven't joined Medium yet then I also suggest you to **join Medium** to read your favorite stories without interruption and learn from great authors and developers. You can **join Medium here**

10 Things Developers should consider while designing and developing Microservices

As I said before, designing and developing microservices is a complex process that requires careful consideration of a variety of factors. As a developer, there are several things you should keep in mind to ensure that your microservices are scalable, efficient, and easy to maintain.

Here are the 10 key considerations that developers should keep in mind when designing and developing microservices in Java or any other programming language:

1. Design your services around business capabilities, not technical capabilities

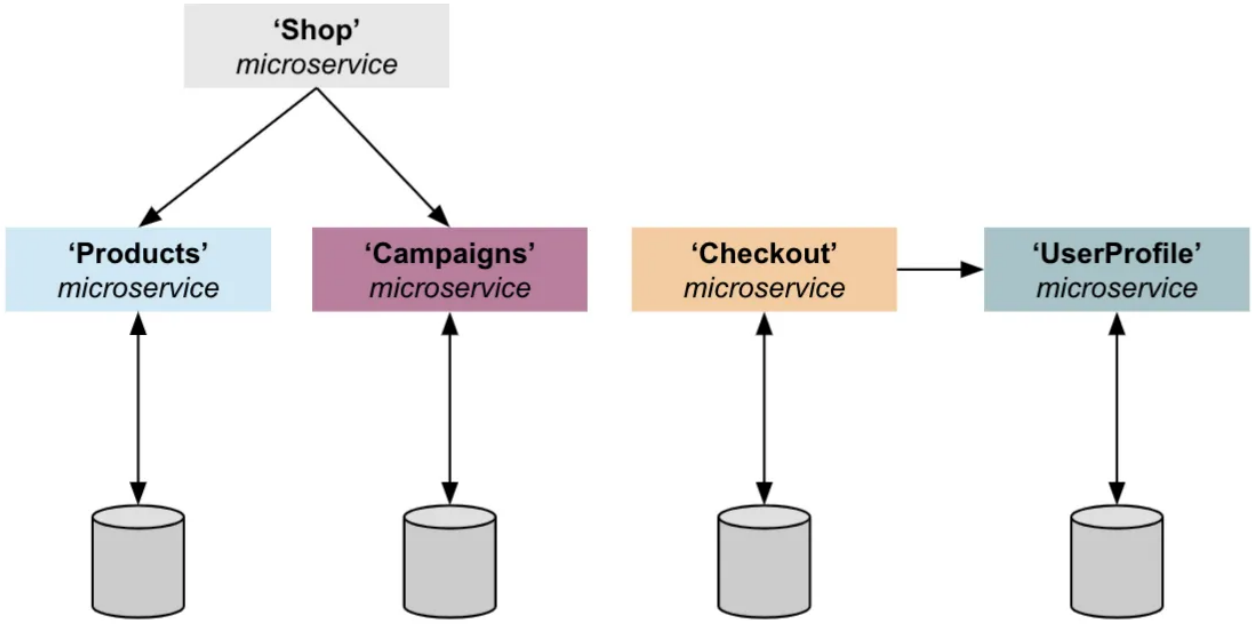
When designing microservices, it's important to focus on the business capabilities of your application, rather than the technical capabilities. For example, instead of

thinking about how to split up a monolithic application into different services based on technical functionality (such as user authentication or data persistence), think about the specific business capabilities that each service provides (such as user management or order processing).

Designing microservices around business capabilities, rather than technical capabilities, has several benefits, including:

1. **Business alignment**
   Aligning microservices with business capabilities ensures that the services are designed to meet business requirements, which helps in achieving business goals.
2. ***Flexibility***
   Services based on business capabilities can be changed or modified more easily to meet changing business needs without affecting other services.
3. **Reusability**
   Services designed around business capabilities are more likely to be reusable across different parts of the organization, reducing development time and effort.
4. **Scalability**
   Business capabilities tend to be more stable over time, which makes services based on these capabilities more scalable.
5. **Collaboration**
   Designing services around business capabilities encourages collaboration between business and IT teams, helping to ensure that IT solutions meet business needs.

Overall, designing microservices around business capabilities helps to align technology with business goals, improve agility and scalability, and promote collaboration between different teams within an organization.



2. Define the boundaries of your Microservices

Like the previous thing about designing Microservices on business capabilities rather than technical capabilities,defining the boundaries of microservices is an important aspect of microservices architecture as it helps to ensure that each microservice has a clearly defined purpose and scope.

This enables developers to work on individual microservices in isolation, without worrying about the impact of changes on other parts of the system.

Defining the boundaries **also helps to improve the overall scalability and maintainability of the system.** By breaking down a monolithic application into smaller, independent services, it becomes easier to manage and scale individual components as required.

Additionally, it allows for easier maintenance and updating of individual services without affecting the entire system.

Let's understand this concept with an example. Assume that you are developing an e-commerce application which consists of multiple functionalities such as product catalog, order management, payment processing, and customer management.

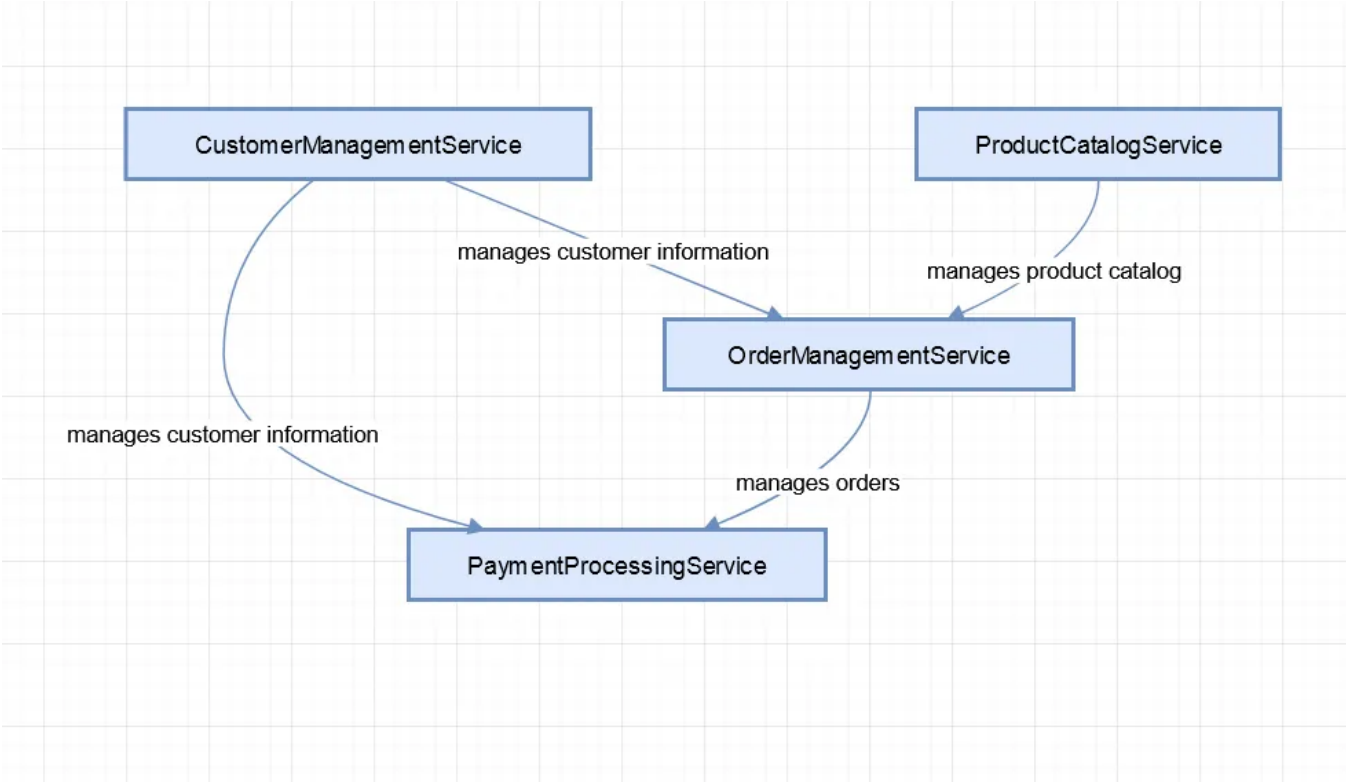
Instead of building a monolithic application that contains all these functionalities, you can break them down into separate microservices.

For example, **you can create a product catalog microservice that manages the product catalog, a payment processing microservice that handles payment processing, an order management microservice that manages orders**, and a **customer management microservice that manages customer information**.

By defining these clear boundaries, each microservice can focus on its specific functionality, making it easier to develop, test, deploy, and scale. It also allows for more flexibility, as changes can be made to one microservice without affecting the others.

Overall, defining the boundaries of microservices helps to ensure that each service is focused on a specific task or business capability, leading to a more efficient and scalable system.

Here is a diagram which explains this concept in bit more detail.



In this diagram, the four microservices are represented by their respective nodes: `CustomerManagementService`, `OrderManagementService`, `PaymentProcessingService`, and `ProductCatalogService`. The arrows between the nodes represent the relationships between the microservices: for example, the `CustomerManagementService` manages

customer information for both the `OrderManagementService` and the `PaymentProcessingService`.

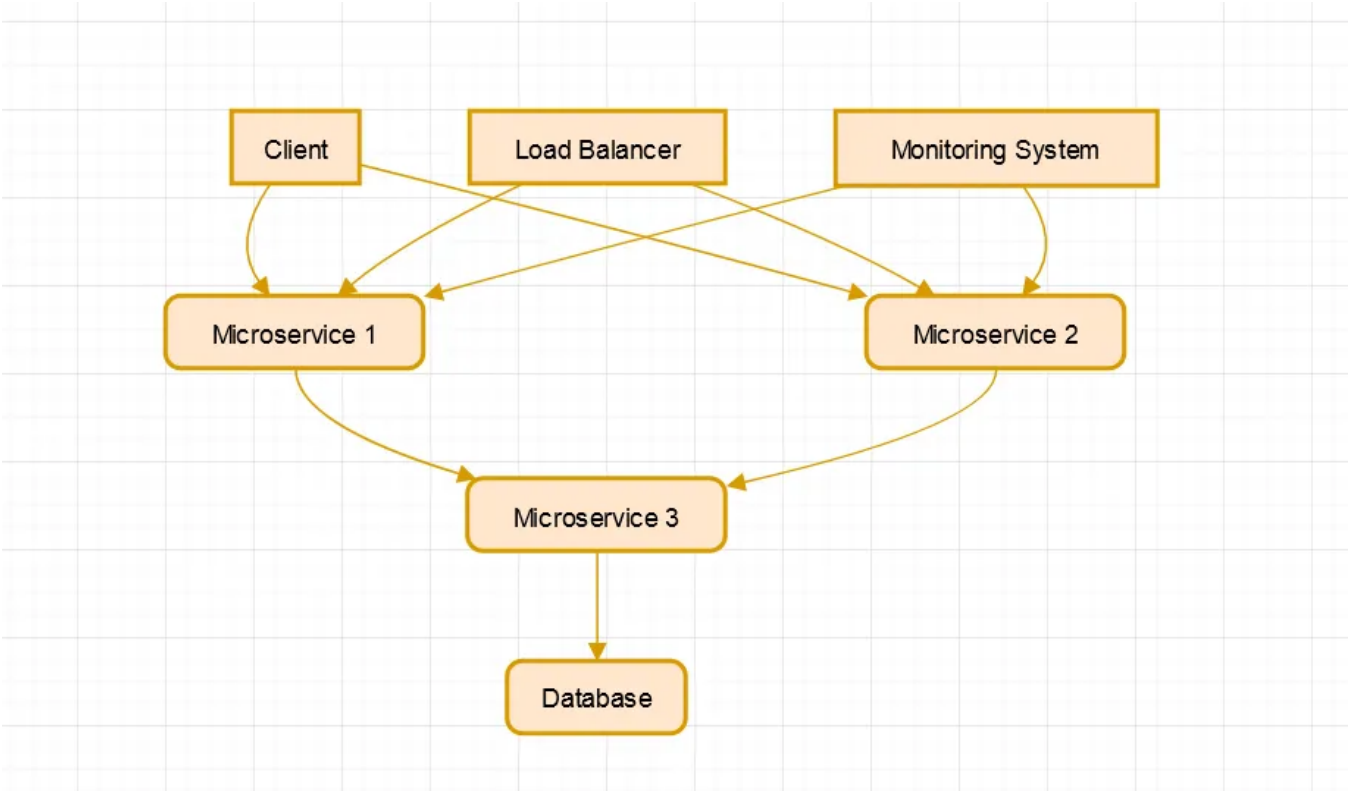
Similarly, the `OrderManagementService` manages orders for the `PaymentProcessingService`, and the `ProductCatalogService` manages the product catalog for the `OrderManagementService`.

3. Design for Resiliency

Microservices should always be designed to be resilient in the face of failures. This means designing for failure at every level, from individual services to the entire system.

For example, services should be designed to gracefully handle network timeouts, and the system should have redundancies in place to ensure that a single point of failure does not bring down the entire system.

Here is a **diagram** which explains how to design a Microservice for Resiliency:



In this diagram, we have a **client that interacts with two microservices (Microservice 1 and Microservice 2)**, which then communicate with a third

microservice (Microservice 3) that interacts with a database. The microservices are load-balanced by a load balancer, and the entire system is monitored by a monitoring system.

To design for resiliency in this architecture, we can take several measures, such as:

- **Implementing retries**
  If a microservice is unavailable, we can implement retries in the client or load balancer to try again after a certain interval.
- **Circuit breakers**
  We can implement circuit breakers to prevent cascading failures. If a microservice is unavailable, the circuit breaker trips and the requests are routed to a fallback service.
- **Graceful degradation**
  If a microservice is experiencing high load or is unavailable, we can implement graceful degradation by returning a reduced functionality response or a cached response.
- **Monitoring**
  We can use a monitoring system to track the performance and availability of the microservices, and set up alerts to notify us when there are issues.

By designing for resiliency in this way, we can ensure that our microservices are able to handle failures and continue to provide a reliable and responsive service to our clients.

4. Use lightweight communication protocols for Microservices

In Microservice architecture, you are bound to have communication between multiple Microservices and in that case you should use a lightweight communication protocol.

Using lightweight communication protocols, such as REST, helps to reduce the complexity and overhead of communication between microservices. These protocols are simple to implement and use, which makes it easier to develop and maintain microservices-based systems.

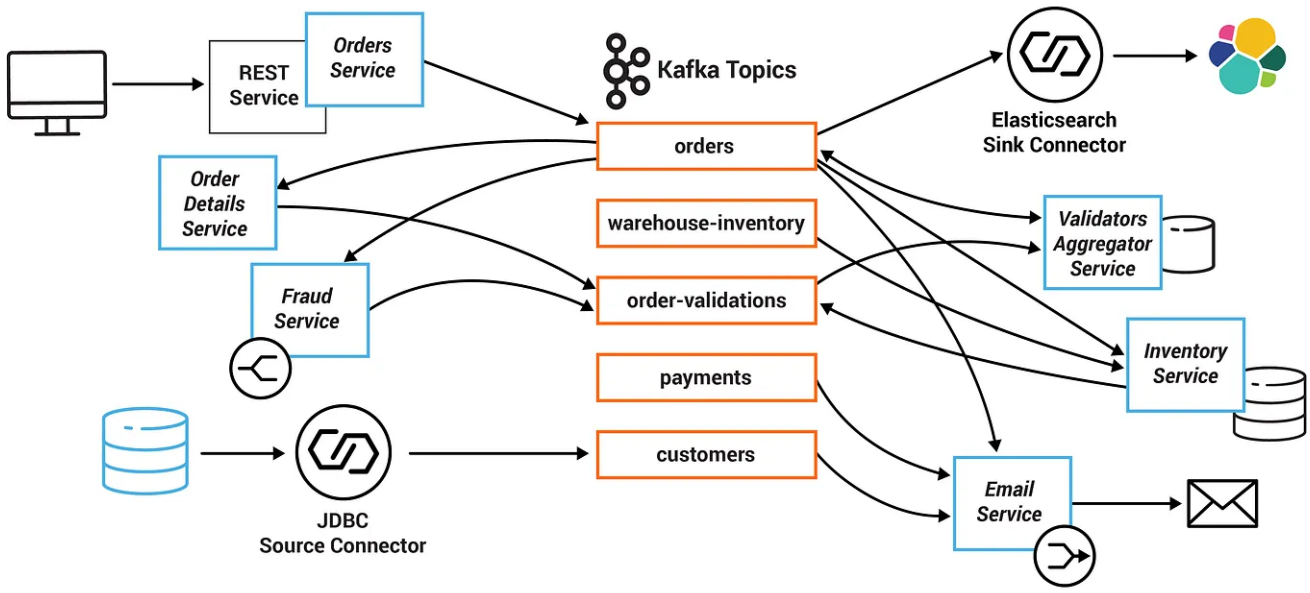
Lightweight protocols also facilitate interoperability between different programming languages and platforms, which can be important in distributed systems where different technologies may be used for different microservices.

In addition, using lightweight communication protocols can help to improve the scalability and performance of microservices-based systems. Since these protocols are designed to be **stateless and cacheable**, they can reduce the amount of network traffic and improve response times.

This is particularly important in systems where microservices may be deployed across different geographical locations or data centers.

Overall, using lightweight communication protocols is an important best practice in microservices-based systems that can help to improve their scalability, performance, and maintainability.

Here is a nice diagram of a Microservice architecture where different Microservices are communicating using **Apache Kafka**



5. Use asynchronous communication between services?

Microservices should communicate with each other **asynchronously**, using technologies such as **messaging queues** or **event-driven architectures.** This allows for greater scalability and flexibility, as services can be scaled up or down

independently of each other, and communication can be decoupled from the rest of the system.

Here are the main benefit of using asynchronous communication between Microservices:

1. **Improved Scalability**
   Asynchronous communication allows for better scalability of microservices because services can process requests independently of each other. This allows for a greater degree of parallelism and reduces bottlenecks in the system.

2. **Increased Fault Tolerance**
   Asynchronous communication helps to make the system more resilient to failures. If a service goes down or becomes unavailable, requests can be queued and processed once the service is available again.

3. **Reduced Coupling**
   Asynchronous communication reduces coupling between services because each service can operate independently and does not need to know about the internal workings of other services.

4. **Improved Performance**
   Asynchronous communication can improve the performance of the system by reducing the overhead of synchronous communication. Services can respond immediately with an acknowledgment, and then process requests in the background.

5. **Better User Experience**
   Asynchronous communication can help to improve the user experience by allowing requests to be processed in the background, which can reduce wait times and increase responsiveness.

In short, you should try for async communication as much as possible between Microservices.

6. Ensure data consistency between Microservices

This one is another important thing to consider while designing and developing Microservices because Microservices typically operate on their own data stores, it's important to ensure data consistency between services.

This can be achieved through techniques such as using a distributed transaction coordinator, or by using eventual consistency and compensating transactions.

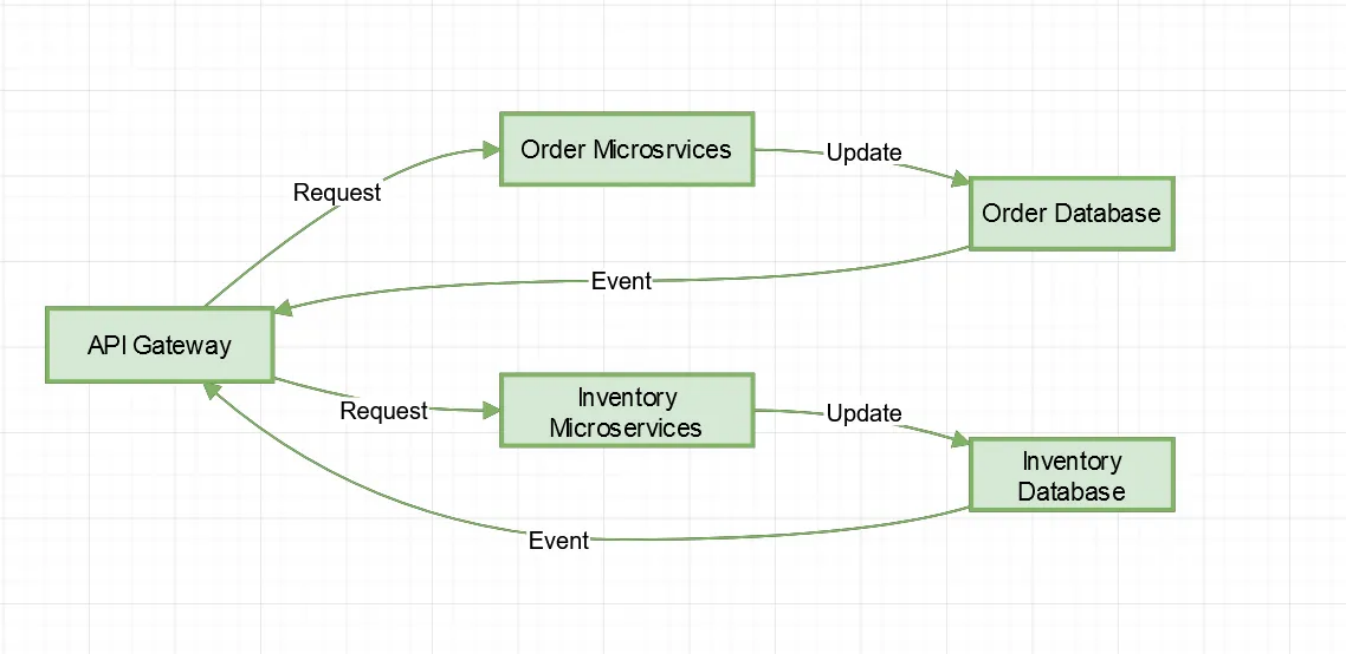
Let's say you have two microservices: Order Management and Inventory Management. Whenever a new order is placed, the Order Management service needs to check if the requested items are available in the inventory or not. If the items are available, the Order Management service should deduct the quantity of items from the inventory.

To ensure data consistency between the two microservices, you can use a distributed transaction management system like **Atomikos** or **Spring Cloud** Sleuth.

This way, when the Order Management service initiates a transaction to deduct the quantity of items from the inventory, the Inventory Management service will participate in the same transaction.

This ensures that either both services commit the transaction or both services roll back the transaction in case of any failure. This way, you can maintain data consistency between the two microservices.

Here is a nice diagram which explains how you can ensure data consistency in Microservices:



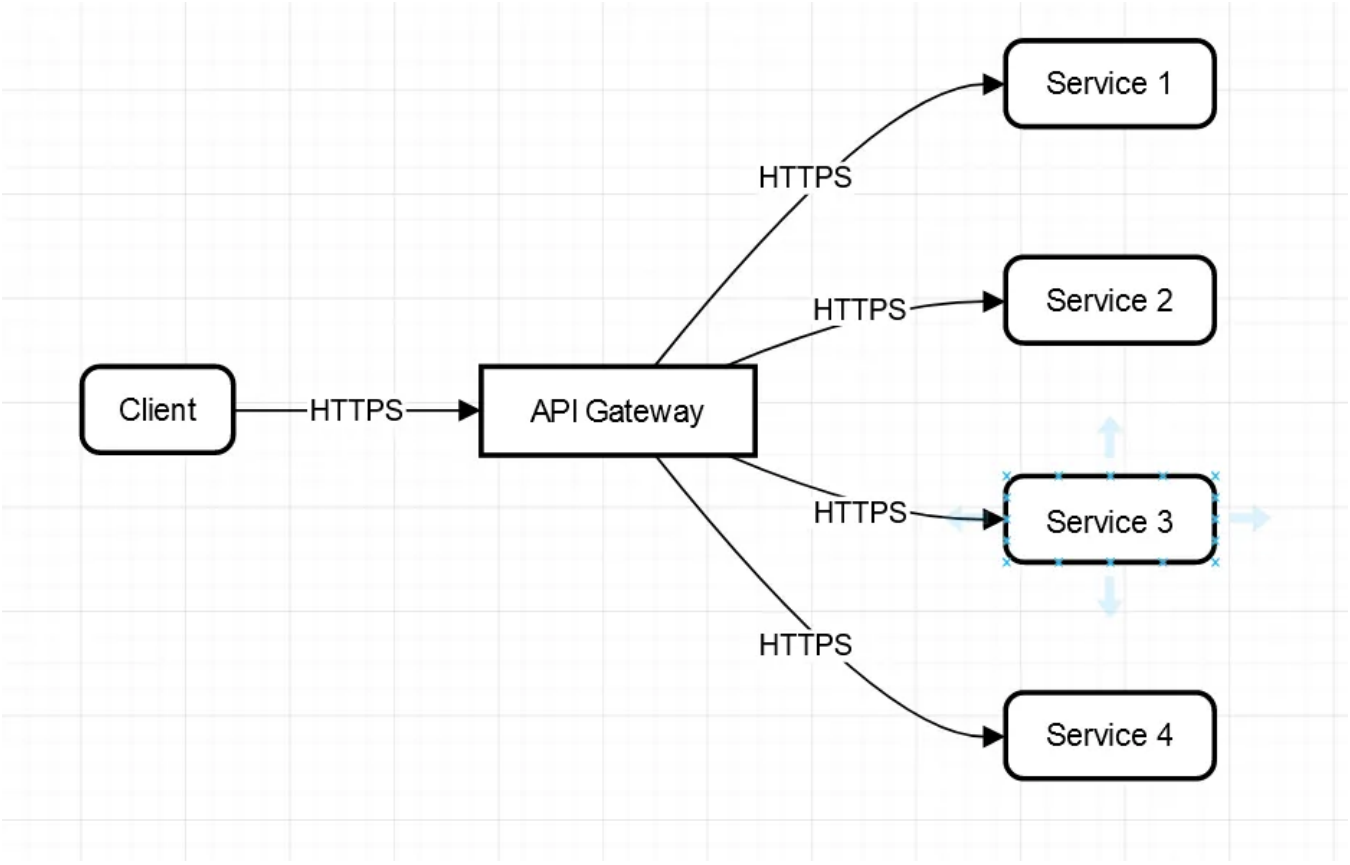
7. Implement Security measures

Security is always an important concern while designing software system and Microservices is no exception. Implementing security measures in Microservices is important to ensure the safety and protection of the sensitive data and functionality that they manage.

Without adequate security measures, Microservices are vulnerable to attacks such as data breaches, unauthorized access, and denial of service.

By implementing security measures, such as authentication, authorization, encryption, and other security protocols, Microservices can protect against these threats and ensure the confidentiality, integrity, and availability of the data and services they manage.

Additionally, implementing security measures can help organizations comply with regulatory requirements and industry standards related to data security and privacy.



In this diagram, the client communicates with the API Gateway over HTTPS. The API Gateway then communicates with multiple microservices, including

Authentication, Authorization, Accounting, and Inventory, also over HTTPS.

This communication is secured with encryption to ensure the confidentiality and integrity of the data being transmitted.

By implementing security measures like HTTPS encryption, microservices can protect sensitive information and prevent unauthorized access to their systems.

8. Monitor and debug effectively

Monitoring and debugging are crucial in Microservices architecture for the following reasons:

1. **Identifying bottlenecks**
   Microservices architecture involves multiple services working together, and a bottleneck in one service can affect the performance of the entire system. Effective monitoring helps identify such bottlenecks and enables developers to address them quickly.
2. **Troubleshooting**
   As Microservices are distributed across different servers, troubleshooting can be challenging. Effective monitoring and logging help identify issues quickly and enable developers to troubleshoot effectively.
3. **Proactive issue resolution**
   Monitoring enables developers to identify issues before they become critical. Proactive issue resolution ensures that the system is running smoothly and helps avoid service outages.
4. **Capacity planning**
   Monitoring helps identify resource usage trends, which can aid in capacity planning. Developers can identify which services are underutilized or over-utilized and adjust resource allocation accordingly.

With many small services working together, it's important to have effective monitoring and debugging tools in place to quickly identify and resolve issues. This can include tools such as centralized logging, distributed tracing, and metrics aggregation.

9. Automate deployment and testing

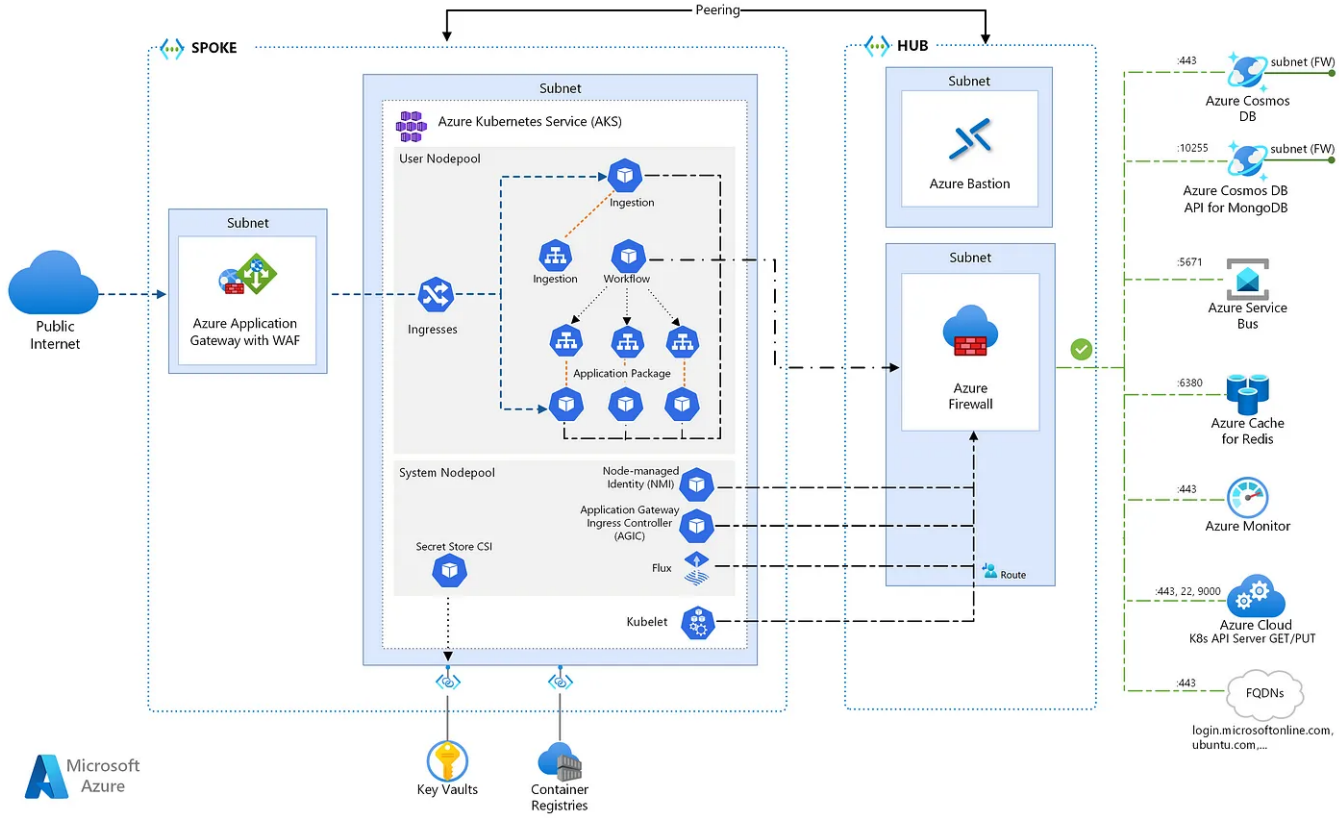
Because microservices are typically deployed independently of each other, it's important to have robust automation in place for deployment and testing.

This can include using c**ontainerization technologies** such as **Docker**, and using continuous integration and delivery pipelines to automate the build, test, and deployment process.

Also, when it comes to cloud, there are several ways to **automatically deploy Microservices in the cloud, depending on the cloud provider and the tools used for deployment.**

One common approach is to use containerization technologies such as Docker and container orchestration platforms like **Kubernetes**. In this approach, microservices are packaged as Docker containers, and Kubernetes is used to manage and scale the containers across a cluster of machines.

Another approach is to use serverless computing platforms such as AWS Lambda or Azure Functions. With serverless computing, developers write code as individual functions that can be deployed and run in response to events triggered by other services. The cloud provider manages the underlying infrastructure and scaling, allowing developers to focus on writing code.



Some cloud providers also offer managed services for deploying microservices, such as AWS Elastic Beanstalk, Azure Service Fabric, and Google App Engine. These services provide pre-configured environments for deploying microservices, along with tools for managing and scaling the services.

Overall, the choice of deployment stack will depend on factors such as the complexity of the microservices, the desired level of control over the infrastructure, and the cloud provider being used.

10. Choose the right technology stack

With so many different technologies available for microservices, it's important to choose the right stack for your specific needs. This might include choosing a language and framework that's well-suited to your application's requirements, as well as choosing the right tools and platforms for deployment, monitoring, and testing.

Now, if you are thinking which technology stack to use then I would say it mainly depends upon the programming language and platform you choose.

There are several technology stacks available to develop microservices, and the choice of stack depends on various factors such as the specific requirements of the project, the team's expertise, and the available resources. However, some of the popular stacks for developing microservices are:

1. **Spring Boot and Java**
   Spring Boot is a widely used Java-based microservices framework that offers several features such as auto-configuration, easy deployment, and quick setup.

2. **Node.js and Express**
   Node.js is a popular JavaScript runtime environment that can be used to build scalable and high-performance microservices. Express is a web application framework for Node.js that simplifies the development of APIs and web applications.

3. **Python and Flask**
   Python is a versatile programming language that can be used to build microservices. Flask is a lightweight and flexible microservices framework for Python that provides several features such as routing, request handling, and session management.

4. **Go and Gin**
   Go is a high-performance programming language that is becoming increasingly popular for developing microservices. Gin is a lightweight and fast microservices framework for Go that offers several features such as routing, middleware, and error handling.

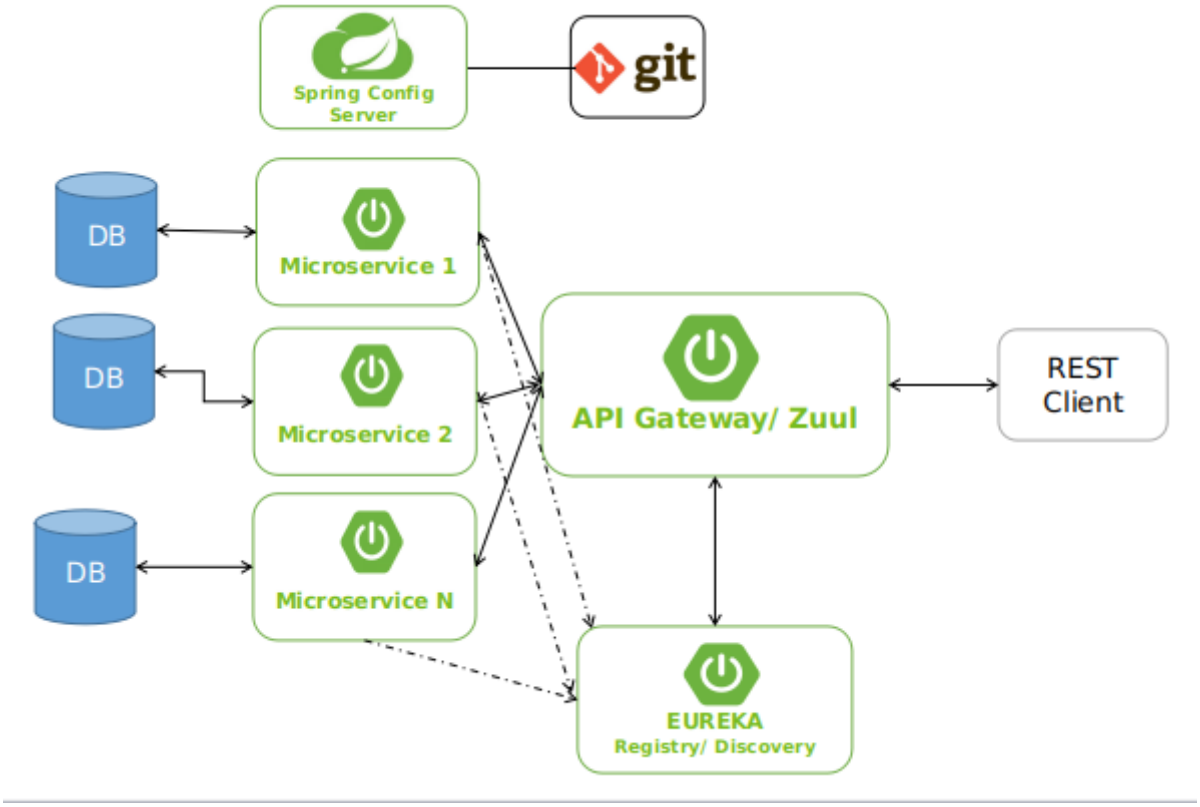
5. **Ruby on Rails**
   Ruby on Rails is a popular web application framework that can also be used to develop microservices. It provides several features such as a built-in ORM, MVC architecture, and RESTful routing.

6. **.NET Core**
   .NET Core is a cross-platform, open-source framework that can be used to build microservices using C# or F#. It provides several features such as high performance, scalability, and security.

These are just some examples of the technology stacks available for developing microservices. Ultimately, the choice of stack depends on the specific needs and

requirements of the project, as well as the expertise and resources available to the development team.

Here is an example of Java Microservices using Spring Boot and Spring Cloud:



Java and Spring Interview Preparation Material

Before any Java and Spring Developer interview, I always use to read the below resources

Grokking the Java Interview

Grokking the Java Interview: click here

I have personally bought these books to speed up my preparation.

You can get your sample copy here, check the content of it and go for it

Grokking the Java Interview [Free Sample Copy]: click here

> *If you want to prepare for the Spring Boot interview you follow this consolidated ebook, it also contains microservice questions from spring boot interviews.*

Grokking the Spring Boot Interview

You can get your copy here — **Grokking the Spring Boot Interview**

Conclusion

That's all about the **10 important things you should consider while designing and developing Microservices** for today's need. Microservices architecture has become increasingly popular due to its benefits of scalability, agility, and flexibility. However, it also presents unique challenges that must be taken into consideration during development.

By keeping in mind the key **Microservices design principles and best practices** outlined above, developers can successfully navigate these challenges and create highly functional and reliable microservices.

Ultimately, this will help to drive business success and meet the growing demand for efficient and effective software solutions.

And , if you like this Microservice article and you are not a Medium member then I highly recommend you to join Medium and many such articles from many Java and Microservice experts. You can **join Medium here**

Join Medium with my referral link - Soma

Read this and every story from thousands of great writers on Medium). Read more from great writers on Medium like...

medium.com

Also, other **Microservices articles** you may like to explore:

Top 10 Microservices Problem Solving Questions for 5 to 10 years experienced Developers

From Service Discovery to Security: 10 Problem-Solving Questions to Test the Microservices Skills of Developers with 5...

medium.com

What is API Gateway Pattern in Microservices Architecture? What Problem Does it Solve?

The API Gateway can help in managing authentication, request routing, load balancing, and caching in Microservices...

medium.com

Microservices Java Programming Microservice Architecture

Software Engineering

Follow

Written by Soma

4.1K Followers · Editor for Javarevisited

Java and React developer, Join Medium (my favorite Java subscription) using my link 👉
https://medium.com/@somasharma_81597/membership

More from Soma and Javarevisited

Messaging Model	Traditional	Publish/Subscribe	Traditional
Scalability	Clustering/Network of Brokers	Partitioning	Clustering/Network of Brokers
Performance	Moderate	High	High
Data Persistence	On Disk (default), In-memory	On Disk	On Disk (default), Database
Integration	Programming Languages, Databases, Web Servers	Data Processing Systems, Databases, Data Sources	JMS Clients, Apache Camel, Apache CXF
Suitable For	Strict Ordering, Reliable Delivery, Moderate-High	Streaming Data, High	High Data Durability,

Soma in Javarevisited

Difference between RabbitMQ, Apache Kafka, and ActiveMQ

Hello folks, if you are preparing for Java Developer interviews the along with Spring Boot, and Microservices, you should also prepare...

· 8 min read · May 19

208 4

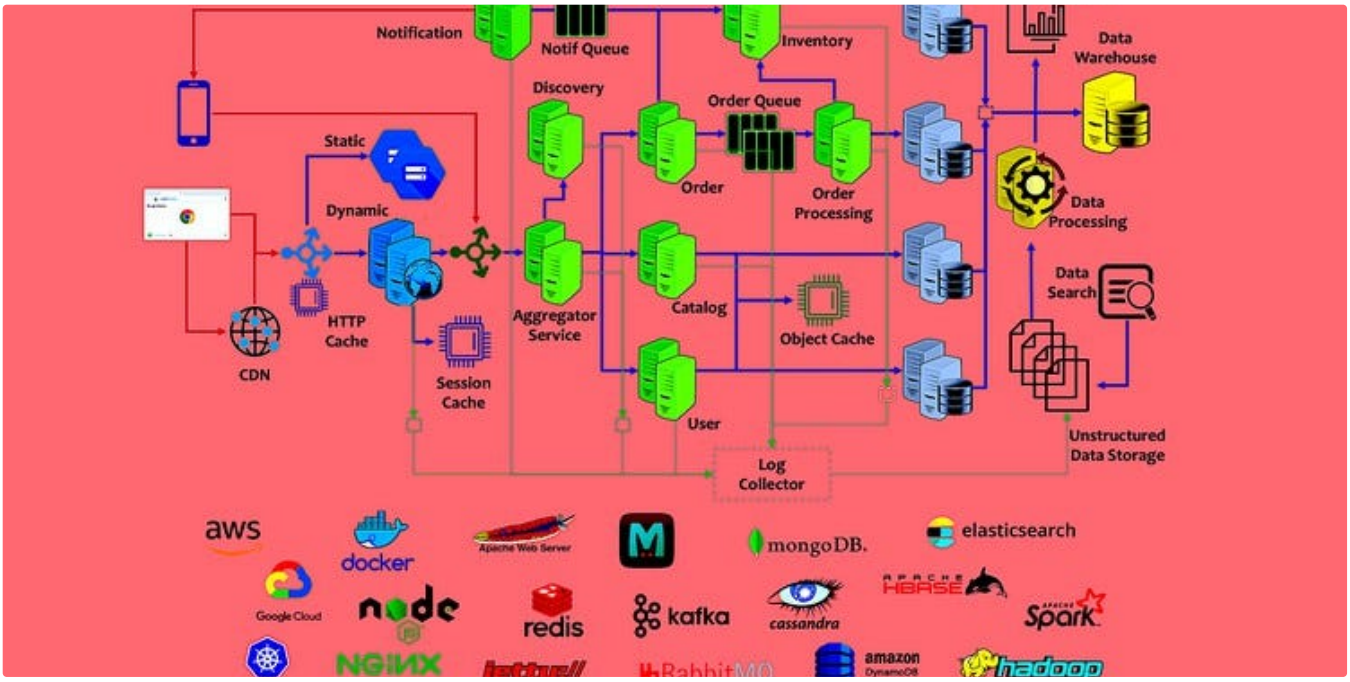
Ajay Rathod in Javarevisited

Comprehensive Spring-Boot Interview Questions and Answers for Senior Java Developers: Series-25

Greetings everyone,

9 min read · 6 days ago

79 1



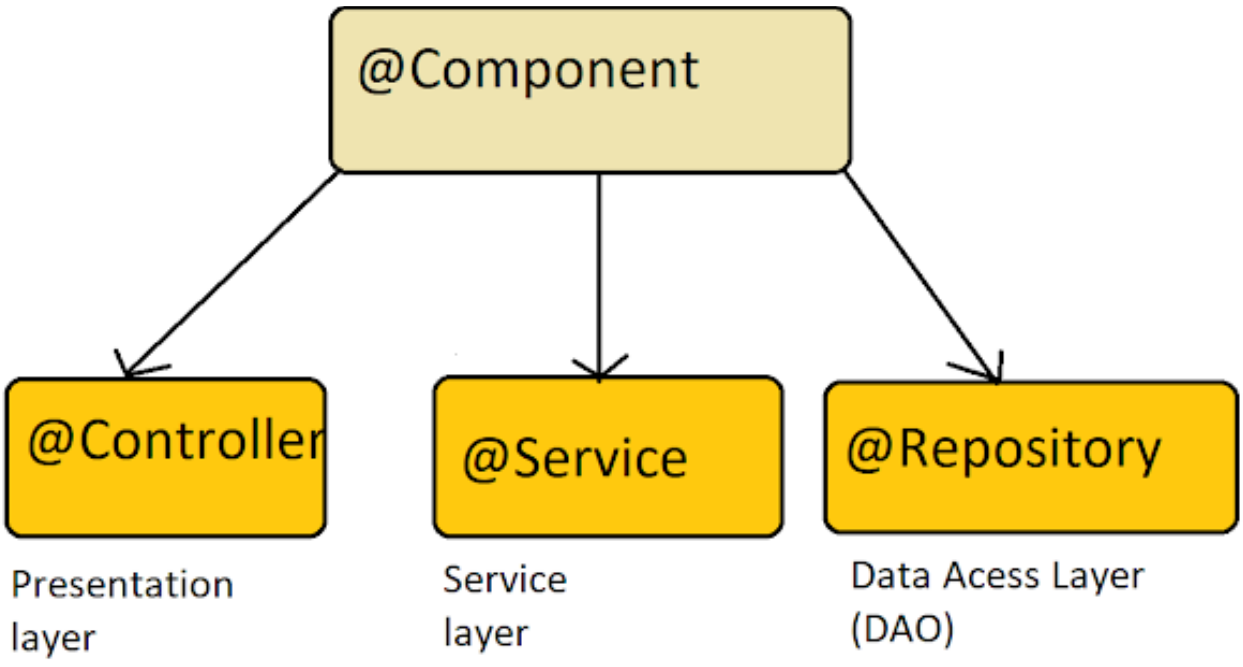
javinpaul in Javarevisited

My Favorite Udemy Courses to Learn System Design in 2023

These are the best System Design courses you can join to not only prepare for System Design interviews but also to learn Software...

11 min read · 6 days ago

108



Soma in Javarevisited

Difference between @Controller, @Service, and @Repository Annotations in Spring Framework?

While all three are stereotype annotation in Spring and can be use to represent bean where exactly they are used is the key for answering...

✦ · 7 min read · Mar 23

238 1

See all from Soma

See all from Javarevisited

Recommended from Medium

Anto Semeraro in Level Up Coding

Microservices Orchestration Best Practices and Tools

Optimizing microservices communication and coordination through orchestration pattern with an example in C#

· 10 min read · Mar 27

69

Daniel Zielinski

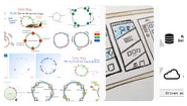
Senior Java Software Developer Interview Questions—part 1

1. Anemic Model vs Rich Model ?

· 3 min read · Feb 18

17

Lists

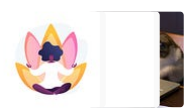


General Coding Knowledge

20 stories · 204 saves

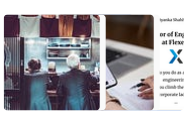
It's never too late or early to start something

13 stories · 69 saves



Stories to Help You Grow as a Software Developer

19 stories · 271 saves



Leadership

35 stories · 92 saves

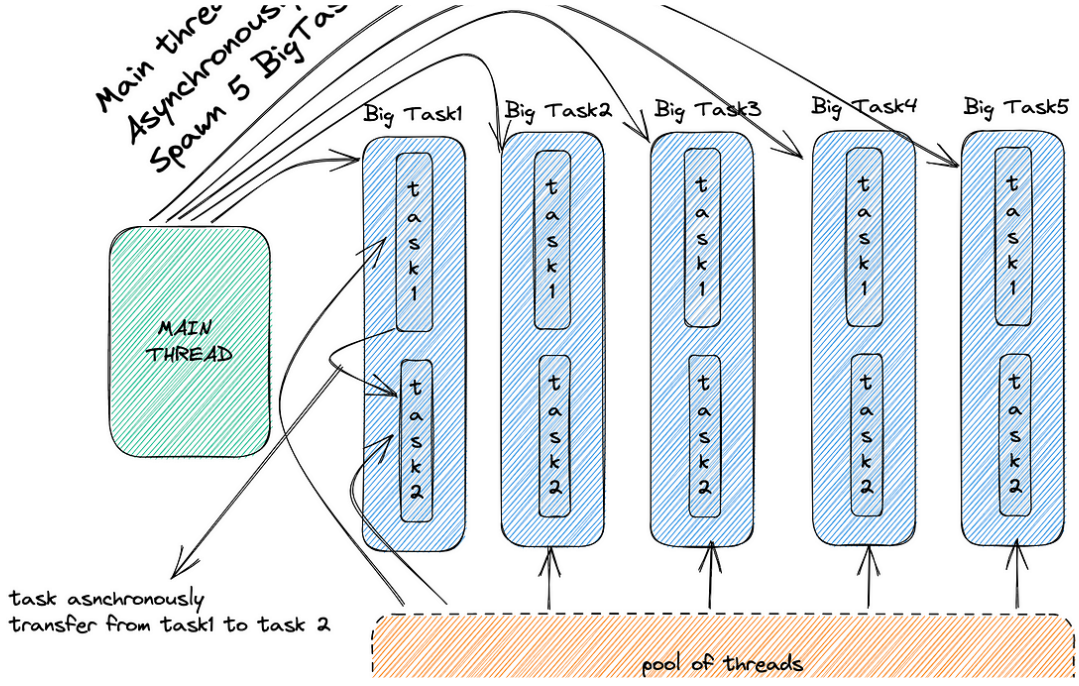
Rupert Waldron

Create a non-blocking REST Api using Spring @Async and Polling

Get the great asynchronous, non-blocking experience you deserve with Spring's basic Rest Api, polling and @Aysnc annotation.

12 min read · Feb 26

23



anil gola

Why have a CompletableFuture when we have an ExecutorService ?

I always touch on this topic while interviewing for Java. Why do we need CompletableFuture when we have ExecutorService? Which problem did...

4 min read · Feb 21

404 5

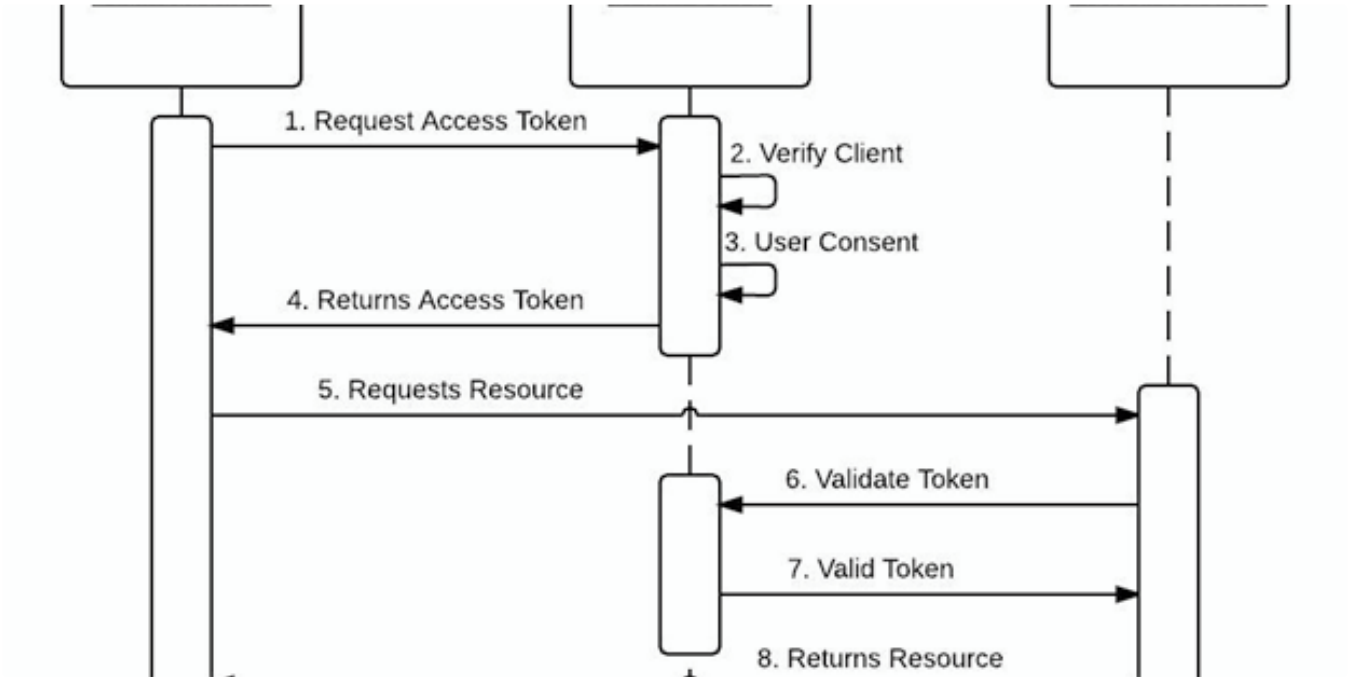
Daryl Goh

Spring Boot: RequestEntity vs ResponseEntity | RequestBody vs ResponseBody

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can “just run”.

· 5 min read · May 21

40



Somal Chakraborty in Dev Genius

Securing a Microservice with OAuth 2.0 (Part 2 : How Token Based Security Works Internally)

Table of content

5 min read · May 1

31 1