Cloud Native Daily

A blog for Devs and DevOps covering tips, tools, and developer stories about all things cloud-native

Follow publication

Fault Tolerance in Microservices Architecture

Explaining the patterns, principles, and techniques involved.

Meherban Singh Follow 8 min read · Jun 12, 2023

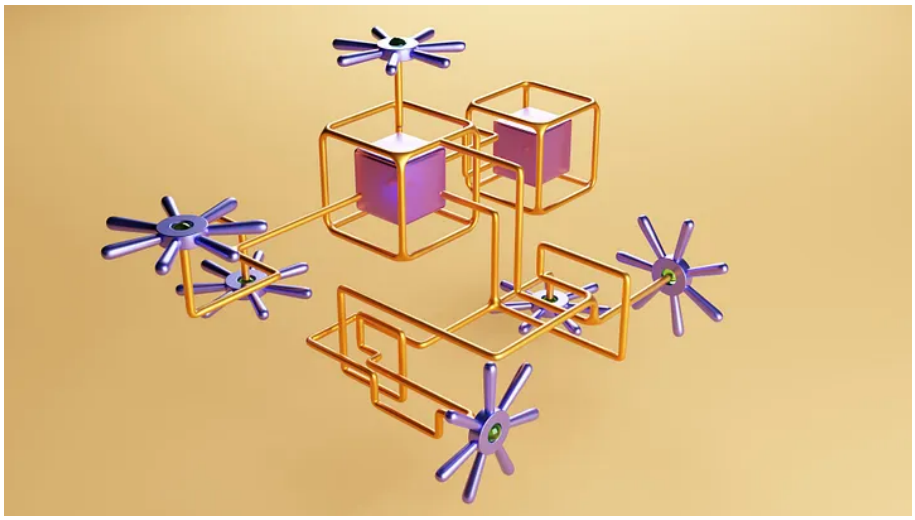


Photo by Growtika on Unsplash

Have you heard of microservices architecture? It's a great way to build complex applications that have grown in popularity recently. Instead of having one large, tightly coupled application, split your application into smaller, independent services that essentially communicate with each other via APIs. Each service is responsible for a specific business function such as user management or payment processing.

The benefits of this approach are very nice. On the one hand, it increases development agility. Different teams can work on different services without stepping on each other, shortening release cycles and facilitating continuous delivery and delivery. Additionally, each service can scale independently, improving resource utilization and fault tolerance. Also, the separation of services allows different technologies and programming languages to be used for each service, which is very convenient.

Of course, there are also some challenges. Communication between services is becoming increasingly important and often requires relying on simple

protocols such as HTTP/REST, Messaging, etc. But overall, microservices architecture is a very nice way to build complex applications and definitely worth trying.

Fault Tolerance

Now Let's talk about fault tolerance. This is very important when designing and running reliable systems, especially in the world of microservice architectures. Fundamentally, fault tolerance means that a system can continue to operate and provide service even if something goes wrong. Believe me, problems can occur in many ways, including hardware malfunctions, software glitches, network issues, and even human error.

Why is fault tolerance so important in microservice architectures? In this type of setup there are many different services that all work independently. This means that each service can experience its own failure, creating a domino effect that can bring down the entire system. not good.

To prevent this, some fault tolerance mechanism should be put in place. But before that, you should understand the different types of errors that can occur. There are two main categories.

Temporary and Permanent Failures.

Temporary outages are temporary and are usually caused by network problems or system congestion. This issue is not permanent and can often be resolved by retrying the process or waiting for the issue to be resolved automatically.

Permanent disability, on the other hand, is more severe. These can be caused by hardware malfunctions, software bugs, or human errors that require manual intervention to correct. In the event of such a failure, a backup system or redundancy mechanism should be put in place to keep things running smoothly.

In summary, fault tolerance is the ability of a system to handle failures without completely collapsing. And this is especially important in microservice architectures. By understanding different types of errors and implementing appropriate strategies, you can build resilient and reliable systems.

Fault Tolerance Principles in Microservices.

Design for Failure :

Microservices architectures are great because they enable scalability, agility, and resilience. But let's be honest, failures are inevitable in distributed systems. That's why it's so important to design your microservices with fault

tolerance in mind. This means making sure the system can properly handle and correct errors. A key principle is to design for failure. It means recognizing that mistakes will happen and planning for them. If you anticipate that a component or service may fail at any time, you can proactively implement strategies to mitigate its impact. Bug-friendly design techniques include:

A. Partition:
Isolate critical services from non-critical services to prevent failure of one service from impacting others. Implementing a bulkhead limits the blast radius of the fault and prevents it from spreading throughout the system. For example, separating user authentication and payment processing into separate services prevents authentication errors from impacting payment processing services.

B. Circuit breaker:
This is a pattern for detecting and handling service outages. If the service fails repeatedly, the circuit breaker will trip and redirect the request to an alternative service or return a predefined fallback response. This pattern helps prevent cascading failures and gracefully degrades systems when services become unavailable.

Meherban Singh highlighted

C. Graceful Degradation:
This means that services should be designed so that their functionality degrades gracefully in the event of a failure. Instead of failing completely, the service may degrade its functionality and provide a limited feature set. For example, an e-commerce application may disable non-essential features.

Designing for failures is like implementing protection switches, and enabling soft degradation ensures that your microservices architecture is fault-tolerant and handles failures gracefully.

Decentralization

It is basically a way to avoid single points of failure in a system by distributing services across multiple nodes and data centers. This means that the failure of one part of the system does not lead to the collapse of the whole system.

There are several decentralization strategies that can be used. One is service replication, which creates copies of important services and distributes them to different instances. This ensures that your service is always available even if your instance goes down. Load balancing techniques can be used to distribute incoming requests to different replicas.

Another strategy is service discovery.Service discovery is the process of automatically detecting and registering services that are available in a microservices architecture. In a microservices architecture, each service

runs independently, and they need to communicate with each other to complete a task. Service discovery helps in locating the services and their addresses so that the communication between them is seamless. .

Finally, distributed data management. When dealing with distributed systems, it is important to spread data across multiple nodes to ensure availability and fault tolerance. Techniques such as database sharding help by distributing data across multiple databases or instances.

Redundancy

Redundancy is having a backup resource in case something goes wrong. This may mean duplicating critical components or services to ensure continued operations. For example, databases can be replicated to maintain data availability and durability. A master/slave or master/master,masterless setup using technologies such as MySQL or PostgreSQL,Cassandra,MongoDb.

Isolation

This is a very important concept in maintaining system stability and preventing bugs from spreading like wildfire. Fundamentally, isolation means containing failures within individual services from affecting the rest of the system.

There are several ways to implement isolation, such as service isolation. Service isolation means that each microservice operates independently on its own resources. That way, failure of one service doesn't lead to failure of other services.

Another technique is the safety net-like circuit breaker pattern. When the service stops responding, it trips the circuit breaker and prevents further requests from being forwarded. Instead, they are routed to a backup plan or return a preconfigured response.

Finally, it's important to ensure that each microservice can be deployed and scaled independently. Therefore, any problem during an update or scaling operation will not affect the system as a whole.

In a word, isolation. Please try to keep these errors at bay. The system will thank you for that.

Fail-Fast

Basically, it's important to detect errors early and react as soon as possible. Rather than allowing bugs to spread and wreak havoc, we want to nip them in the bud and minimize their impact. Here are some important points to keep in mind:

First, a solid monitoring and warning system should be put in place. This allows you to monitor your microservices and spot strange behavior before it spirals out of control. Metrics, logs and health checks are great tools for this. Then automated testing is on your side. By thoroughly testing your code before deploying it, you can identify potential problems and prevent major problems down the road. Unit testing, integration testing, and end-to-end testing are important pieces of the puzzle.

Speed is of the essence when it comes to detecting and responding to errors. We want to be able to quickly identify problematic services and take appropriate action. Timeouts and retries are very useful here as they help detect unresponsive or slow services and invoke fallback mechanisms or circuit breakers.

Finally, continuous integration and delivery (CI/CD) methods help automate the process of deploying and updating microservices. This means that bugs can be caught early and prevented from entering production. It also saves you time and effort in the long run. Keeping these tips in mind will help you catch bugs early and keep your microservices running smoothly.

Resilience Patterns

Circuit Breaker Pattern : https://eksimtech.com/circuit-breaker-pattern-36dae90f2acb

Retry Pattern : https://eksimtech.com/retry-pattern-resilience-design-patterns-df6e97092c49

Bulkhead Pattern : https://eksimtech.com/bulkhead-pattern-5acaf3ea00b8

Testing for Fault Tolerance Strategies :

As a developer, if you want to ensure that your microservice application is production-ready, there are several strategies you can use to test its fault tolerance. All of these strategies are about identifying system weaknesses and allowing the system to handle failures without crashing.

First is the **unit test**. Here, individual components of the system are tested individually to identify problems before they become widespread. We want to cover both expected and unexpected scenarios, such as edge cases and error conditions. Next is the integration test. The purpose is to test how the various components of the system work together, which is especially important in a microservices architecture. I want to test both normal and abnormal scenarios. B. Network Outages and Service Outages.

Next is **chaos engineering**. Here we intentionally introduce a bug into the system to see how it handles. By simulating different failure scenarios, you can identify weaknesses and make changes to improve system resilience.

Load testing is also important for testing fault tolerance. Here we test how the system behaves under heavy load, where failures are most likely to occur. We want to test both normal load and peak load scenarios to make sure the system can handle failures under heavy load.

Finally, there are tests for **disaster recovery.** Here we test how the system recovers after a catastrophic failure. Restoring from backups, restoring data, etc. should be tested to ensure that the system is back to normal after a serious problem. For better understanding please follow **Disaster Recovery Guide**

Monitoring and Observability

Monitoring is the collection and analysis of data to detect problems or anomalies in a system or application. This data comes from metrics, logs, and alerts and focuses on specific things that can be tracked over time.

Observability, on the other hand, is understanding how a system or application behaves from the outside. It is important not only to collect data but also to analyze it in real-time to get a complete picture of what is happening. This means looking at things like logs, traces, and other data sources to get a real sense of overall system health and performance.

In summary, Monitoring is about tracking something over time, while observability is about fully understanding how a system works by examining different data sources in real-time.

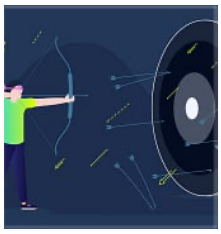
For a complete understanding of Monitoring and Observability please follow this guide.

Further Reading:

5 Microservices Challenges and Blindspots for Developers

Microservices dev top 5 challenges and solutions: testing, observability, troubleshooting & debugging, services...

gethelios.dev



Microservices Monitoring: Cutting Engineering Costs and Saving Time

A few ways fort leveraging Helios to save on engineering costs and dev time for a more resource-efficient organization...

gethelios.dev

Testing Microservices - Trace Based Integration Testing Example

Microservices architectures require a new type of testing. Here's why traditional testing fail and the new automated...

gethelios.dev

API observability: Leveraging OTel to improve developer experience

A Helios developer shares her experience applying API observability to improve API discovery, enforcement and...

gethelios.dev

Software Engineering | Microservices | Resilience | Distributed Systems | Software Development

Published in Cloud Native Daily

1.3K followers · Last published Apr 4, 2025

Follow

A blog for Devs and DevOps covering tips, tools, and developer stories about all things cloud-native

Written by Meherban Singh

163 followers · 89 following

Follow

Responses (1)

Ishu

What are your thoughts?

Sohail Shah
Jul 16, 2023

Great article, gives a lot of amazing information.

1 Reply

More from Meherban Singh and Cloud Native Daily

In Cloud Native Daily by Meherban Singh

Microservices Monitoring and Observability in Depth

Exploring Monitoring and Observability in Microservices.

Jun 12, 2023 8

In Cloud Native Daily by Maryam Naveed

Blue-Green Deployments with Kubernetes: A Comprehensive...

In the modern world of software development, deploying new versions of...

May 7, 2023 111 10

In Cloud Native Daily by SANKET RAI

Monitoring Kubernetes Pods Resource Usage with Prometheu...

In this article, we will explore how Prometheus and Grafana can be leveraged to monitor...

May 20, 2023 53 1

Meherban Singh

How Vector Databases & Semantic Search Can Change Reliance Stoc...

What is a Vector DB?

Sep 17

See all from Meherban Singh | See all from Cloud Native Daily

Recommended from Medium

In Stackademic by Shanvika Devi

Stop Saying ‘Immutable Means Can’t Change’—Java Interviewers...

The one concept that separates “I know Java” from “I understand Java.”

4d ago 302 8

Ankit Kumar Srivastava

Design Uber Backend

System requirements

May 25 2

In ITNEXT by Animesh Gaitonde

System Design: Building TikTok-Style Video Feed for 100 Million...

A deep dive into architecture, scalability, and real-time data delivery at scale

May 20 787 23

FullStack With Ram

Spring Boot Starts Here: Mastering Annotations in the Main Class

In a Spring Boot application, the main class typically contains several key annotations...

Jul 29 1

Gaddam.Naveen

5 Threads That Saved My Microservice from a 2 AM Outage...

if you are not a medium member then Click here to read free

Oct 21 11

Agam Kakkar

Achieving Data Consistency in Microservices: A Practical Guide

One of the biggest challenges when designing microservices is ensuring data...

Aug 25 1

See more recommendations