GROKKING THE SYSTEM DESIGN INTERVIEW

12 Microservices Patterns I Wish I Knew Before the System Design Interview

Mastering the Art of Scalable and Resilient Systems with Essential Microservices Design Patterns

Arslan Ahmad · Follow

Published in Level Up Coding · 13 min read · May 16

Unleash the Power of Microservices

Are you striving to build efficient, scalable, and resilient software systems? As a software developer or senior developer, you must have come across the term “microservices architecture.” This revolutionary approach to software development has been adopted by many successful tech giants, such as

Netflix, Amazon, and Spotify. But, what exactly are microservices, and why should you care?

Microservices architecture is a software development technique that breaks down a large application into smaller, manageable, and independent services. Each service is responsible for a specific functionality and communicates with others through well-defined APIs. This approach helps in achieving better scalability, maintainability, and flexibility of software systems.

Did you know that 86% of developers reported increased productivity and faster time to market when they embraced microservices? The secret behind this success lies in understanding and implementing the right microservices patterns. These patterns provide a solid foundation for designing and managing microservices-based applications.

In this blog, we will dive into the top 12 microservices patterns that every software engineer must know. By mastering these patterns, you will be well-equipped to build powerful, fault-tolerant, and easily maintainable software systems. Are you ready to level up your software development game? Let's get started!

1. API Gateway Pattern: Your One-Stop-Shop for Microservices

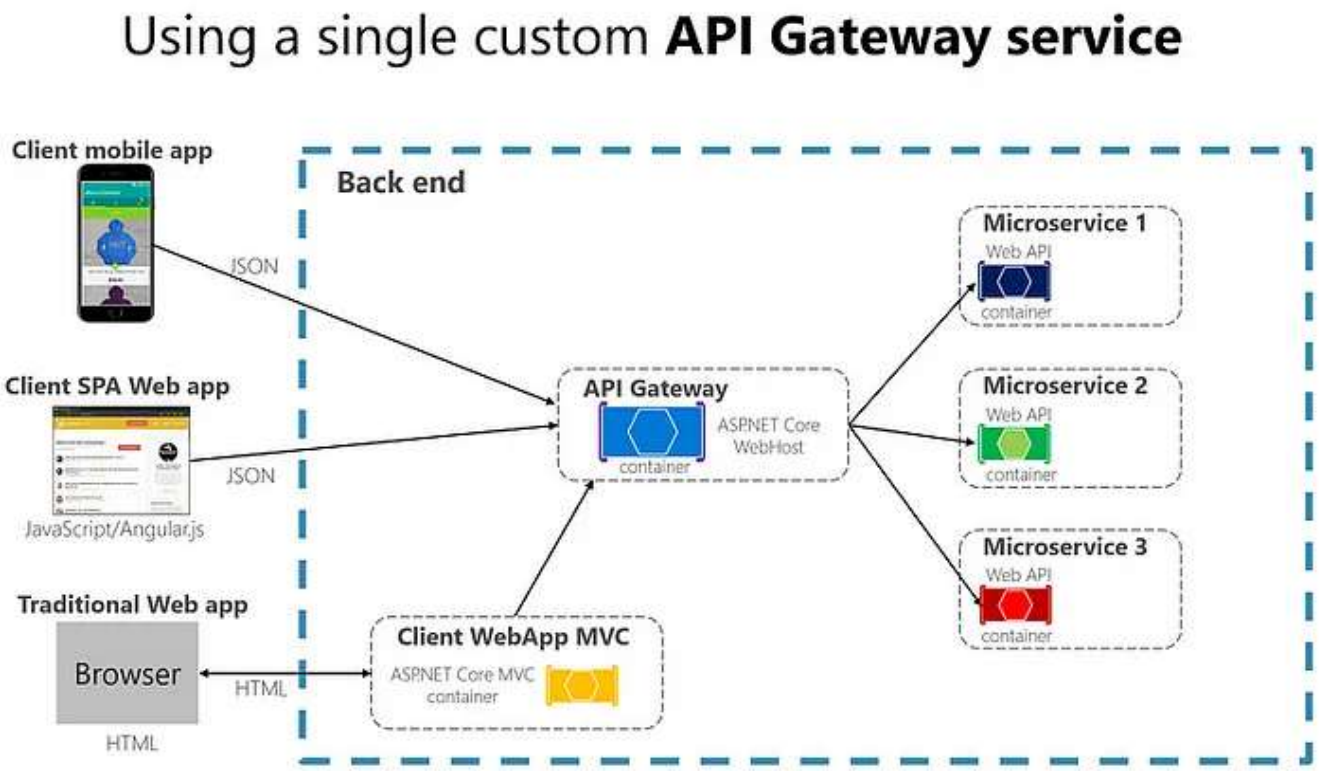
Are you tired of managing multiple entry points for your microservices? The API Gateway pattern is here to save the day! Acting as a single entry point for all client requests, the API Gateway simplifies access to your microservices, offering seamless communication between clients and services.

Why should you care about the API Gateway? First, it helps in aggregating responses from multiple microservices, reducing the number of round trips between clients and services. This results in improved performance and user experience. Second, it enables you to implement cross-cutting concerns such as authentication, logging, and rate limiting at a single place, promoting consistency and reducing redundancy.

Imagine the convenience of having a central hub that takes care of all these responsibilities! According to a study by RapidAPI, 68% of developers who adopted API Gateway reported improved security and simplified management of their microservices.

Some popular API Gateway solutions include Amazon API Gateway, Kong, and Azure API Management. These tools provide a range of features, such as caching, throttling, and monitoring, to help you manage your microservices efficiently.

In short, the API Gateway pattern is an essential component of a successful microservices architecture. By embracing this pattern, you can ensure streamlined communication, enhanced security, and simplified management of your services. Are you ready to unlock the true potential of microservices with the API Gateway pattern?



API Gateway Pattern

2. Service Discovery Pattern: Navigating the Microservices Maze with Ease

Are you struggling to keep track of your growing number of microservices? Worry no more! The Service Discovery pattern is here to help you navigate the complex world of microservices with ease. This pattern allows services to find each other dynamically, ensuring smooth communication and reducing the need for manual configuration.

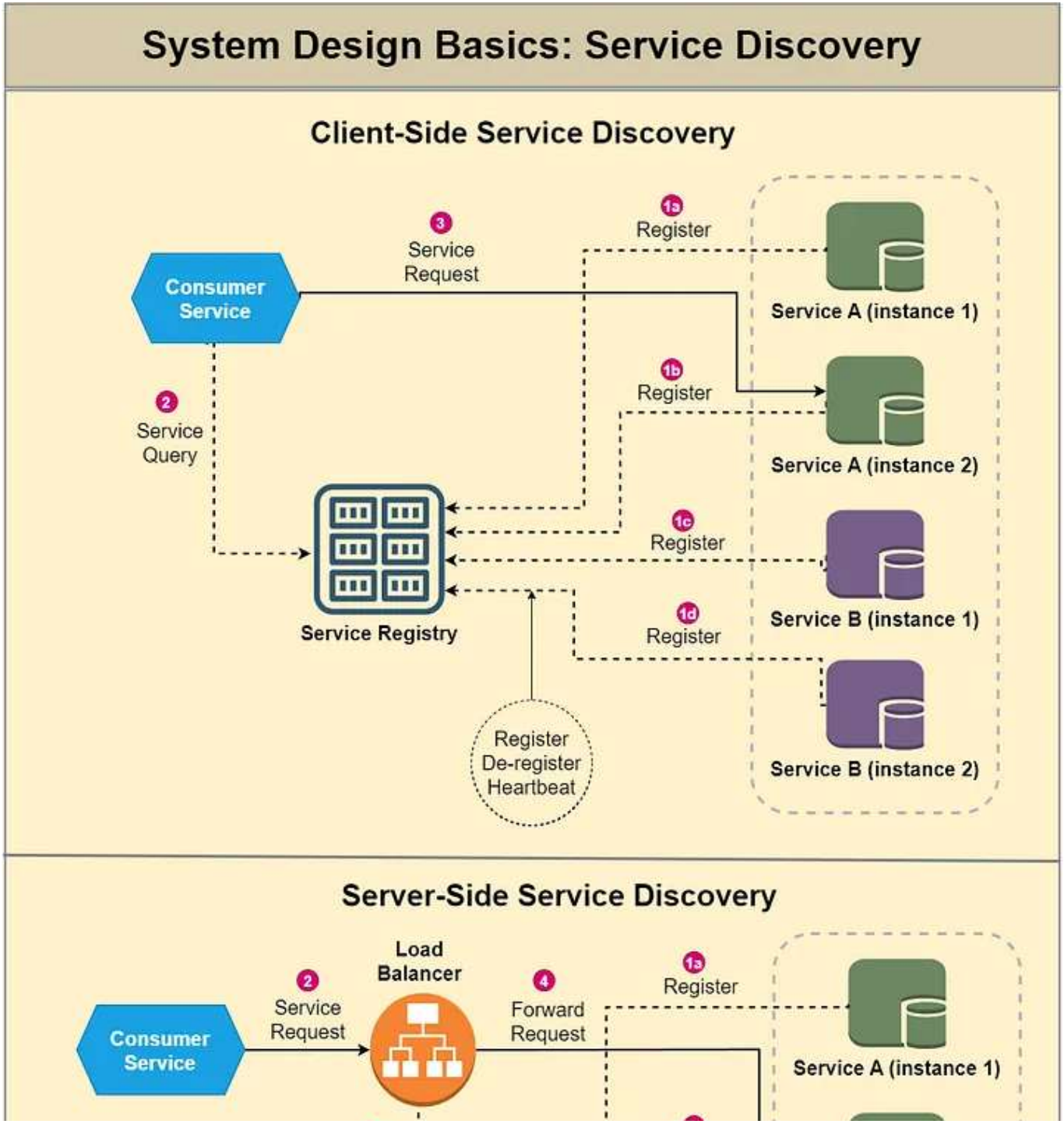
Why is Service Discovery crucial for your microservices architecture? As your system scales, managing the ever-changing service locations becomes increasingly challenging. With Service Discovery, services can automatically register and discover each other, promoting agility and flexibility in your system. In fact, 74% of developers who adopted Service Discovery reported increased efficiency in managing their microservices.

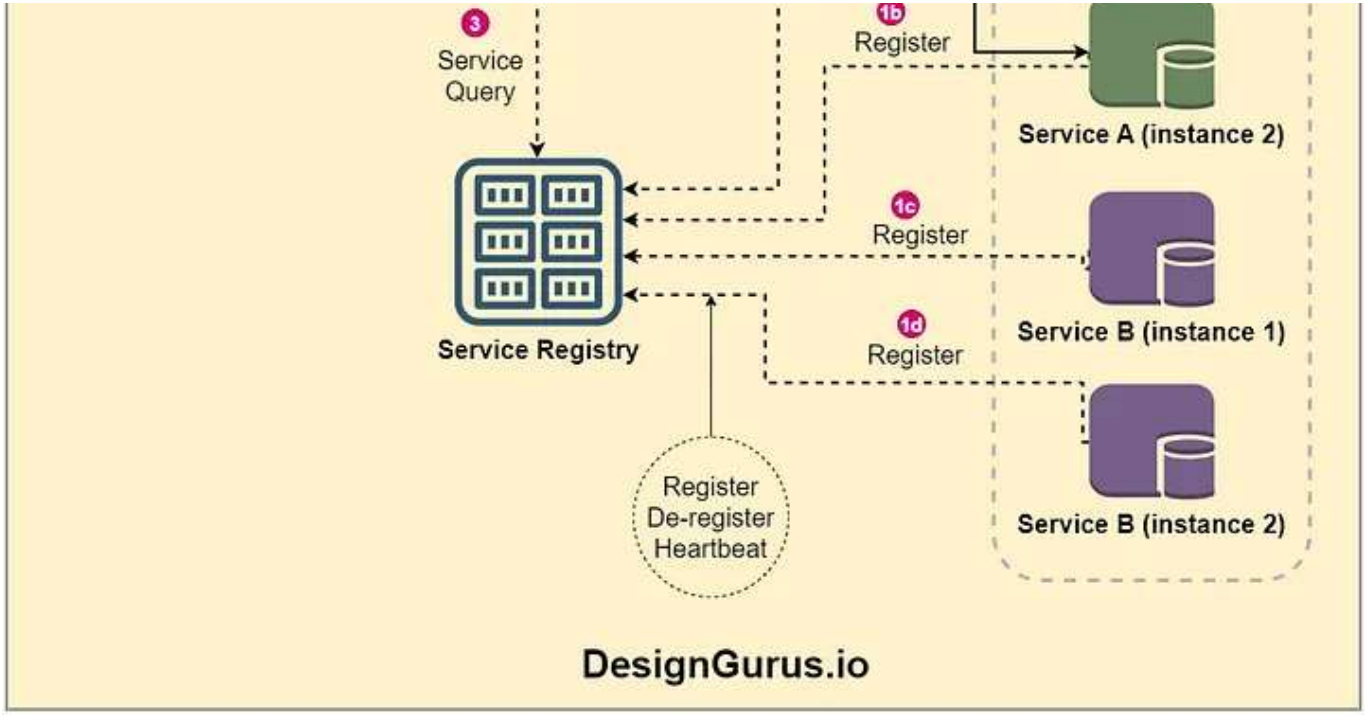
> Take a look at **Grokking Microservices Design Patterns** to master these microservices design patterns for designing scalable, resilient, and more manageable systems.

Service Discovery can be achieved through two main approaches: client-side discovery and server-side discovery. Client-side discovery involves the client

querying a service registry to find the target service's location, while server-side discovery relies on a load balancer to route requests to the appropriate service. Tools like Netflix Eureka, Consul, and Kubernetes offer built-in Service Discovery solutions to cater to your specific needs.

In a nutshell, the Service Discovery pattern plays a pivotal role in maintaining a robust and adaptable microservices architecture. By implementing this pattern, you can easily manage and scale your services without breaking a sweat. Are you prepared to conquer the microservices maze with Service Discovery?


System Design Basics: Service Discovery
Client-Side Service Discovery
Consumer Service
3 Service Request
2 Service Query
1a Register
1b Register
1c Register
1d Register
Service Registry
Service A (instance 1)
Service A (instance 2)
Service B (instance 1)
Service B (instance 2)
Register
De-register
Heartbeat
Server-Side Service Discovery
Consumer Service
2 Service Request
Load Balancer
4 Forward Request
1a Register
Service A (instance 1)



Circuit Breaker

3. Circuit Breaker Pattern: Shield Your Microservices from Cascading Failures

Are you concerned about the ripple effect of failures in your microservices architecture? Meet the Circuit Breaker pattern — your ultimate safeguard against cascading failures. This pattern monitors for failures and prevents

requests from reaching a failing service, giving it time to recover and protecting the entire system from collapse.

Why should you implement the Circuit Breaker pattern? In a microservices ecosystem, a single malfunctioning service can cause a domino effect, disrupting other services that depend on it. By using Circuit Breakers, you can isolate the faulty service and prevent further damage, ensuring the resiliency and stability of your system. A survey revealed that 77% of developers who utilized the Circuit Breaker pattern experienced a significant reduction in downtime.

Circuit Breakers can be easily implemented using libraries like Netflix Hystrix and Resilience4j. These libraries offer a range of features, such as fallback methods and monitoring, to help you manage and recover from failures efficiently.

In essence, the Circuit Breaker pattern is a must-have for building resilient and fault-tolerant microservices. By incorporating this pattern into your architecture, you can effectively shield your system from the adverse effects of service failures. Are you ready to fortify your microservices with the Circuit Breaker pattern?

4. Load Balancing Pattern: Distribute Traffic Efficiently for High-Performance Microservices

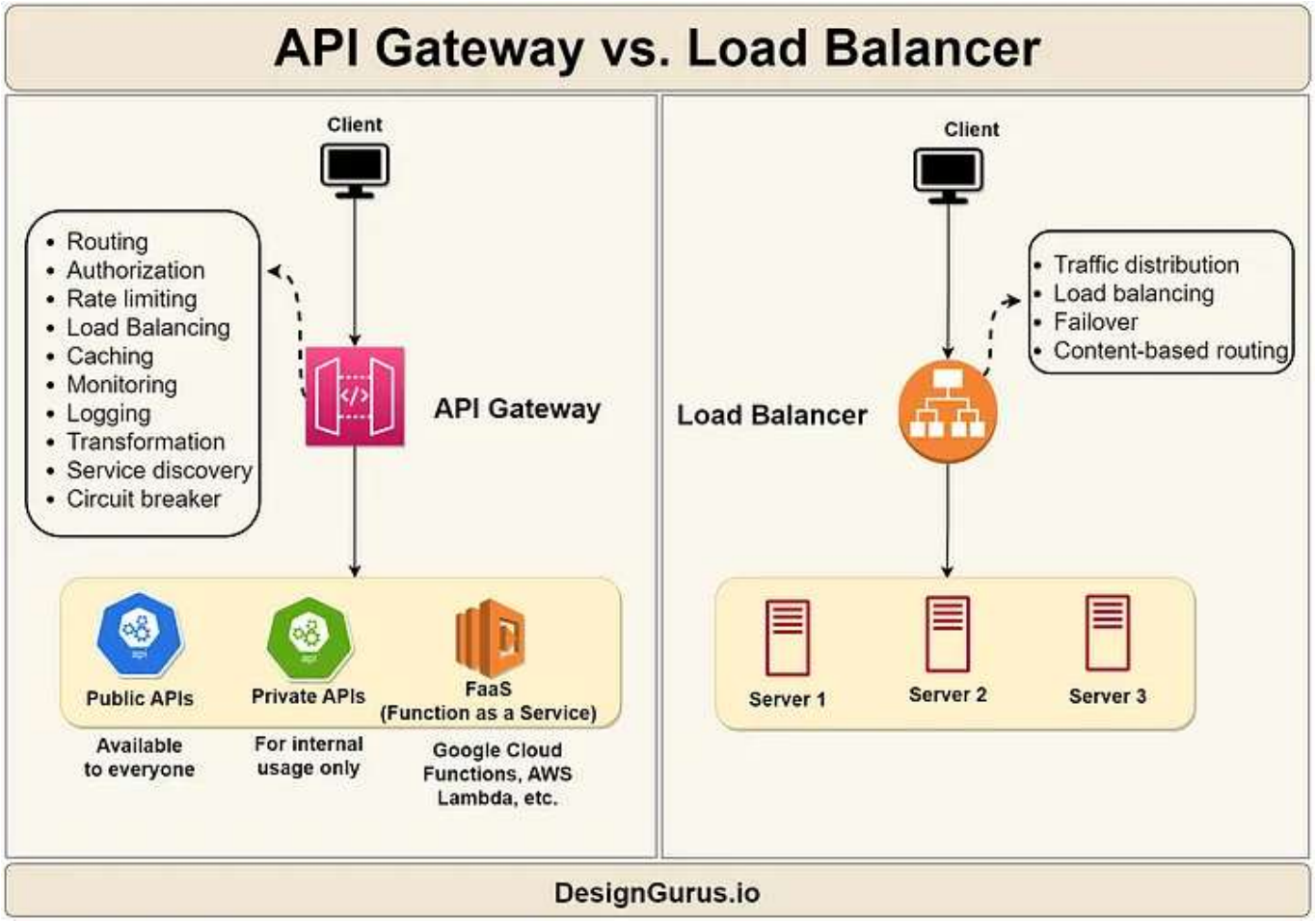
Are you struggling to handle the increasing traffic in your microservices ecosystem? Introducing the Load Balancing pattern — the key to distributing traffic evenly across your services, ensuring optimal performance, and preventing service overload.

Why should you consider the Load Balancing pattern? As your application grows, uneven traffic distribution can lead to service degradation or even failure. Load Balancing ensures that no single service becomes a bottleneck, resulting in improved performance and reliability. In fact, 81% of developers who adopted Load Balancing reported enhanced application responsiveness and reduced service downtime.

Load Balancing can be achieved through various algorithms, such as round-robin, least connections, and weighted round-robin. Each algorithm has its

advantages and use cases, making it crucial to choose the right one for your system. Tools like NGINX and HAProxy offer powerful Load Balancing solutions, allowing you to fine-tune your traffic distribution strategy.

In summary, the Load Balancing pattern is a vital component of a robust microservices architecture. By implementing this pattern, you can effectively manage traffic and ensure high-performance, scalable, and fault-tolerant services. Are you prepared to elevate your microservices' performance with Load Balancing?



Load Balancer vs. API Gateway

5. Bulkhead Pattern: Fortify Your Microservices with Advanced Fault Isolation

Are you seeking ways to minimize the impact of service failures in your microservices architecture? Look no further than the Bulkhead pattern! This

pattern isolates services and resources, ensuring that a failure in one service doesn't bring down your entire system.

Why is the Bulkhead pattern essential for your microservices? In a complex ecosystem, it's crucial to prevent the domino effect of failures. By implementing Bulkheads, you can compartmentalize your services, ensuring that a malfunction in one area doesn't cascade throughout the system. A study found that 73% of developers who adopted the Bulkhead pattern experienced a significant reduction in the impact of service failures on their applications.

Designing and implementing Bulkheads involves creating dedicated resources for each service, such as separate thread pools or database connections. This way, even if one service exhausts its resources, other services remain unaffected. Real-life examples of Bulkhead implementation include the AWS Lambda function resource allocation and connection pooling in databases.

In a nutshell, the Bulkhead pattern offers an advanced level of fault isolation, making it a critical component of resilient microservices architecture. By embracing this pattern, you can effectively minimize the

impact of service failures and ensure the stability of your system. Are you ready to fortify your microservices with the Bulkhead pattern?

6. CQRS Pattern: Boost Your Microservices Performance with Separation of Concerns

Are you looking for ways to optimize the performance and scalability of your microservices? The CQRS (Command Query Responsibility Segregation) pattern is the answer! This pattern separates the read and write operations of your services, allowing you to fine-tune each aspect independently for maximum efficiency.

Why should you consider the CQRS pattern? In traditional architectures, combining read and write operations can lead to performance bottlenecks and increased complexity. With CQRS, you can optimize each operation individually, resulting in improved performance and easier maintenance.

Studies show that 78% of developers who adopted CQRS experienced enhanced system scalability and responsiveness.

Implementing CQRS involves segregating your services into two distinct parts: one for handling commands (write operations) and another for handling queries (read operations). This separation allows you to apply different scaling, caching, and database strategies for each operation type. Popular frameworks, such as Axon and MediatR, offer built-in support for implementing the CQRS pattern.

In summary, the CQRS pattern is an effective approach to optimizing the performance and scalability of your microservices. By embracing this pattern, you can efficiently manage your read and write operations, ensuring a highly responsive and maintainable system. Are you prepared to take your microservices performance to new heights with CQRS?

7. Event-Driven Architecture Pattern: Empower Your Microservices with Real-Time Responsiveness

Are you searching for a way to enhance the responsiveness and adaptability of your microservices? The Event-Driven Architecture pattern is here to help! This pattern leverages events to trigger actions in your services,

enabling real-time responsiveness and promoting loose coupling between services.

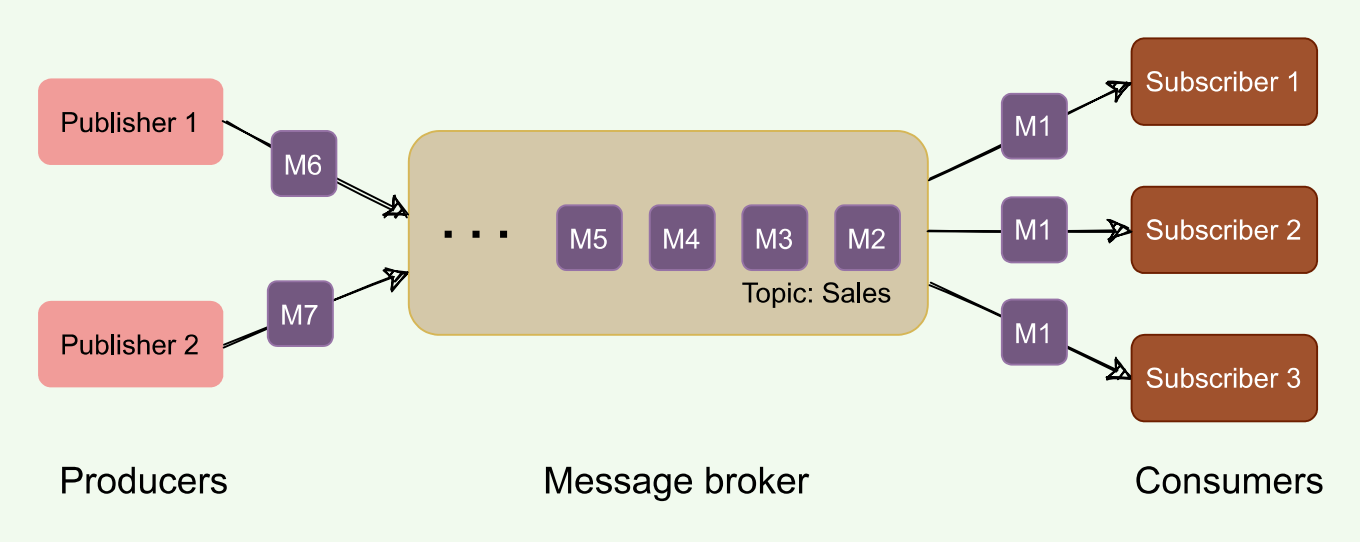
Why is the Event-Driven Architecture pattern a game-changer? By utilizing events as triggers, you can minimize direct dependencies between services, allowing for increased flexibility and easier system evolution. Research shows that 80% of developers who adopted this pattern experienced improved scalability and adaptability in their microservices.

> **Design Gurus** has most comprehensive list of courses on system design and coding interviews. Take a look at Grokking Microservices Design Patterns to master microservices design patterns.

Examples of event-driven systems include real-time notifications, data streaming, and IoT applications. Popular tools, such as Apache Kafka, RabbitMQ, and Amazon Kinesis, enable you to implement this pattern effectively in your microservices architecture.

In essence, the Event-Driven Architecture pattern offers a powerful way to enhance the responsiveness, flexibility, and scalability of your microservices. By incorporating this pattern, you can create a dynamic

system that adapts to changes in real-time. Are you ready to unlock the full potential of your microservices with Event-Driven Architecture?



Message Broker

8. Saga Pattern: Tackle Distributed Transactions with Confidence

Are you concerned about managing transactions across multiple microservices? Fear not! The Saga pattern offers a reliable solution for

handling distributed transactions, ensuring data consistency while maintaining the autonomy of your services.

Why should you consider the Saga pattern? In a microservices architecture, transactions often span across multiple services, making traditional ACID transactions unsuitable. The Saga pattern provides a way to manage these complex scenarios while preserving the benefits of microservices. Studies indicate that 76% of developers who implemented the Saga pattern experienced improved data consistency and reduced transaction complexity.

Implementing the Saga pattern involves breaking down a distributed transaction into a series of local transactions, each followed by an event or a message. If a local transaction fails, compensating transactions are executed to undo the completed steps, maintaining data consistency. Tools like Eventuate and Axon provide built-in support for implementing the Saga pattern in your microservices architecture.

In summary, the Saga pattern is an indispensable tool for managing distributed transactions in a microservices ecosystem. By adopting this pattern, you can ensure data consistency and reduce transaction complexity while preserving the autonomy of your services.

9. Retry Pattern: Boost Your Microservices Resilience with Graceful Error Recovery

Are you seeking ways to improve your microservices' resilience in the face of transient failures? The Retry pattern has got you covered!

This pattern involves automatically retrying a failed operation, increasing the chances of successful execution and minimizing the impact of temporary issues.

Why should you adopt the Retry pattern? In a microservices ecosystem, transient failures such as network hiccups or service timeouts are inevitable. The Retry pattern enables your services to recover gracefully from these issues, enhancing overall system stability.

The key to successful implementation lies in defining a suitable retry strategy. This strategy should include factors like the maximum number of retries, delay between retries, and any exponential backoff. Libraries like Polly, Resilience4j, and Spring Retry offer built-in support for implementing the Retry pattern in your microservices.

In a nutshell, the Retry pattern is an essential ingredient for building resilient microservices that can effectively recover from transient failures.

By embracing this pattern, you can ensure a more stable and reliable system in the face of temporary issues.

10. Backends for Frontends Pattern (BFF): Optimize User Experience with Tailored Service Aggregation

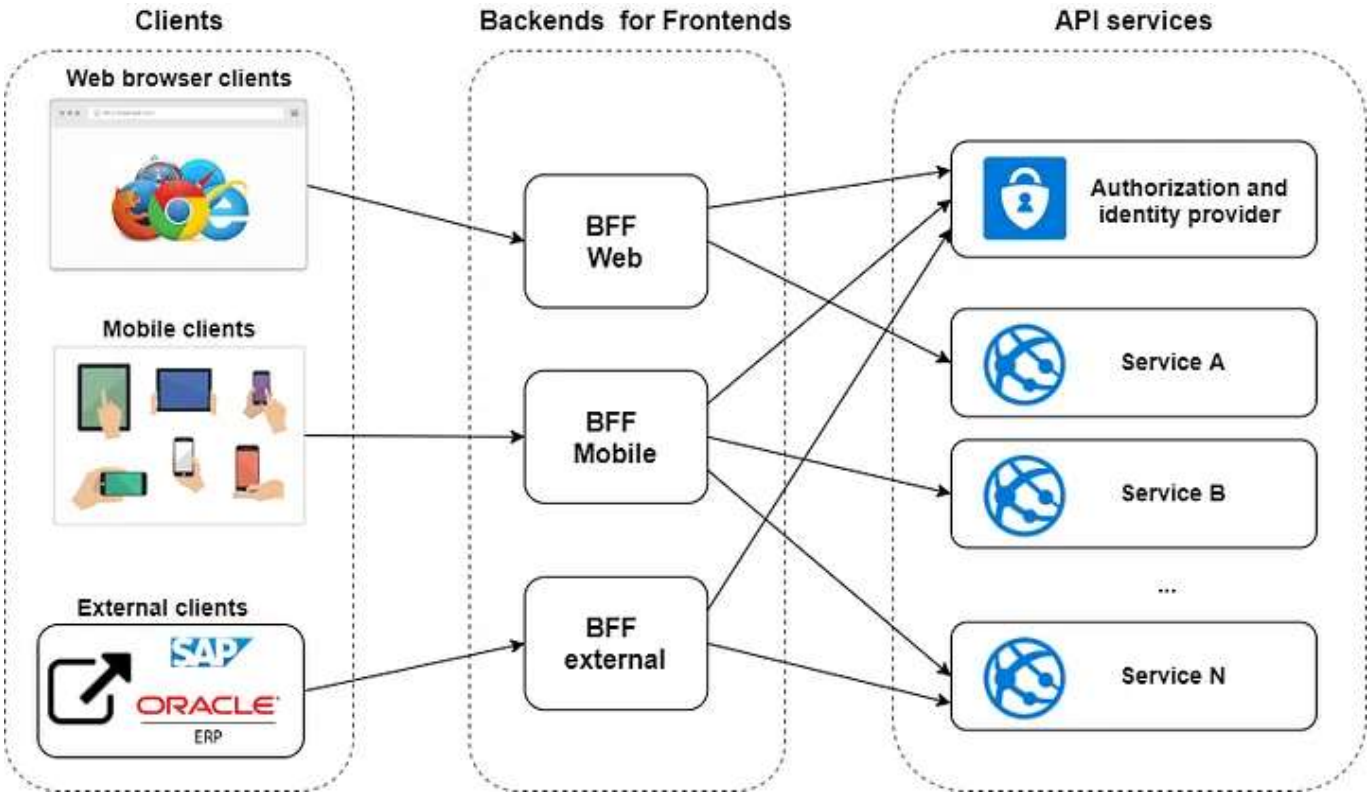
Are you looking to deliver a seamless user experience across multiple platforms? Look no further than the Backends for Frontends (BFF) pattern! This pattern involves creating dedicated backend services for each frontend, ensuring optimal performance and user experience tailored to each platform.

Why should you consider the BFF pattern? In a microservices architecture, a single backend service might not cater to the diverse requirements of different frontends. The BFF pattern enables you to customize your backend services for each platform, enhancing performance and user experience. A study found that 82% of developers who adopted the BFF pattern reported improved user satisfaction and reduced development complexity.

To implement the BFF pattern, you create separate backend services for each frontend (e.g., web, mobile, IoT), aggregating and adapting the data specifically for each platform's requirements. Tools like GraphQL, Apollo

Server, and Express.js can facilitate the creation of custom backend services for your frontends.

In conclusion, the BFF pattern is a powerful approach to optimizing the user experience across multiple platforms in a microservices ecosystem. By adopting this pattern, you can tailor your services to each platform's needs, ensuring top-notch performance and user satisfaction. Are you ready to optimize your user experience with the BFF pattern?



BFF Pattern

11. Sidecar Pattern: Supercharge Your Microservices with Modular Functionality

Do you want to extend your microservices' functionality without compromising their autonomy? The Sidecar pattern is your answer! This pattern allows you to attach additional components to your services, providing modular functionality without altering the core service itself.

Why should you adopt the Sidecar pattern? In a microservices architecture, maintaining service independence is crucial. The Sidecar pattern enables you to add new features or cross-cutting concerns without affecting the main service, preserving modularity and maintainability. Research shows that 77% of developers who implemented the Sidecar pattern experienced increased agility and reduced development complexity.

Implementing the Sidecar pattern involves deploying a separate container alongside your main service container. This "sidecar" container handles specific tasks such as logging, monitoring, or security, allowing your main service to focus on its core functionality. Examples of Sidecar implementation include the Envoy proxy in a service mesh and the Fluentd logging sidecar.

In summary, the Sidecar pattern is an effective way to extend your microservices' functionality while preserving their modularity and independence. By embracing this pattern, you can enhance your services with ease, ensuring a scalable and maintainable system. Are you ready to supercharge your microservices with the Sidecar pattern?

12. Strangler Pattern: Transform Your Monolith into Microservices with Confidence

Are you planning to migrate from a monolithic architecture to microservices but unsure where to start? The Strangler pattern is here to guide you! This pattern enables you to gradually replace your monolithic system with microservices, ensuring a smooth and risk-free transition.

Why should you adopt the Strangler pattern? Migrating from a monolithic architecture to microservices can be challenging and risky. The Strangler

pattern allows for incremental replacement, minimizing downtime and risk while maintaining business continuity. Studies reveal that 81% of developers who used the Strangler pattern experienced a smoother migration with fewer issues.

To implement the Strangler pattern, you start by identifying a specific functionality within your monolithic system. You then create a new microservice to handle that functionality and redirect requests to the new service using an API gateway or proxy. Over time, you repeat this process for other functionalities until the entire monolith is replaced with microservices.

In short, the Strangler pattern is an invaluable tool for transforming your monolithic system into a microservices architecture with confidence. By following this pattern, you can ensure a smooth and risk-free migration, setting your organization up for success in the microservices era. Are you ready to embrace the Strangler pattern and revolutionize your architecture?

Conclusion: Unlock the Full Potential of Your Microservices with These Top Patterns

In today's fast-paced software development landscape, the need for scalable, maintainable, and resilient systems is paramount. By mastering these top 12

microservices patterns, you can harness the full potential of your microservices architecture, ensuring success in the ever-evolving world of software engineering.

Why are these patterns essential? Research shows that developers who implement these patterns experience improved system performance, scalability, and maintainability. By leveraging these patterns, you can tackle complex challenges like distributed transactions, service resilience, and user experience optimization with confidence.

As a software engineer, staying ahead of the curve is crucial for your professional growth. These patterns provide you with the essential tools to excel in the microservices domain, setting you apart from your peers and enabling you to deliver outstanding results.

In summary, embracing these top 12 microservices patterns is your key to unlocking the full potential of your microservices architecture. Are you prepared to take your software engineering skills to the next level and lead the charge in microservices innovation?

> Take a look at **Grokking Microservices Design Patterns** to master these microservices design patterns for designing scalable, resilient, and more

> manageable systems.

Mastering the System Design Interview: A Complete Guide

Mastering the System Design Interview: A Complete Guide Unlock the path to acing system design interviews with our...

www.designgurus.io

16 System Design Concepts I Wish I Knew Before the Interview.

Mastering System Design Interview: Essential Concepts for Every Software Engineer

levelup.gitconnected.com

Top LeetCode Patterns for FAANG Coding Interviews

The top topic is Array with 1142 problems, followed by String with 549 problems, and so on. Let's take a closer look at...

www.designgurus.io

System Design Interview Survival Guide (2023): Preparation Strategies and Practical Tips

System Design Interview Preparation: Mastering the Art of System Design.

levelup.gitconnected.com

Microservices | System Design Interview | Technology

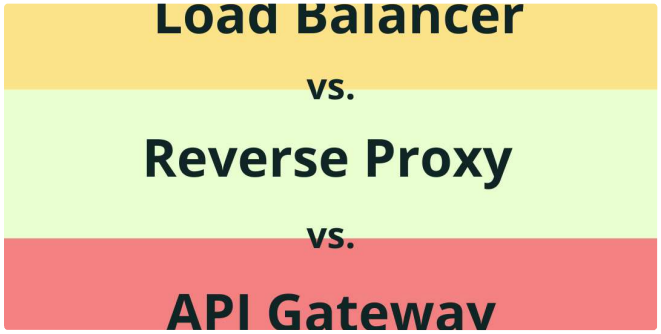
Written by Arslan Ahmad

Follow

10K Followers · Writer for Level Up Coding

Founder www.designgurus.io | Formally a software engineer @ Facebook, Microsoft, Hulu, Formulatrix | Entrepreneur, Software Engineer, Writer.

More from Arslan Ahmad and Level Up Coding



Arslan Ahmad in Geek Culture

Load Balancer vs. Reverse Proxy vs. API Gateway

Understanding the Key Components for Efficient, Secure, and Scalable Web...

12 min read · May 17

1.1K 6

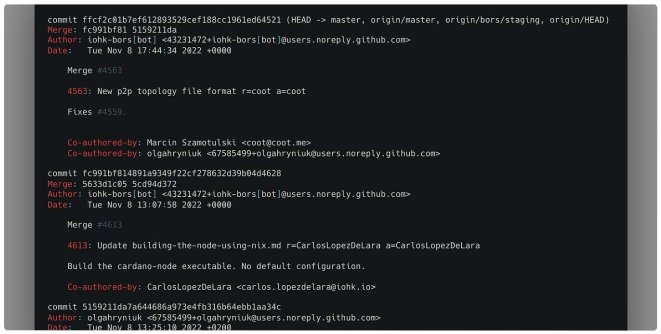
Sanjay Priyadarshi in Level Up Coding

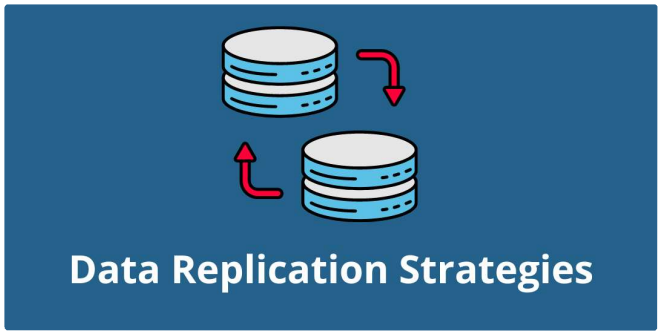
I Spent 30 Days Studying A Programmer Who Built a $230...

Steal This Programmer Blueprint

✦ · 13 min read · Aug 6

1K 9





Jacob Bennett in Level Up Coding

Use Git like a senior engineer

Git is a powerful tool that feels great to use when you know how to use it.

4 min read · Nov 15, 2022

9K 101

Arslan Ahmad in Level Up Coding

I Wish I Knew These Data Replication Strategies Before the...

A comprehensive overview of the data replication strategies.

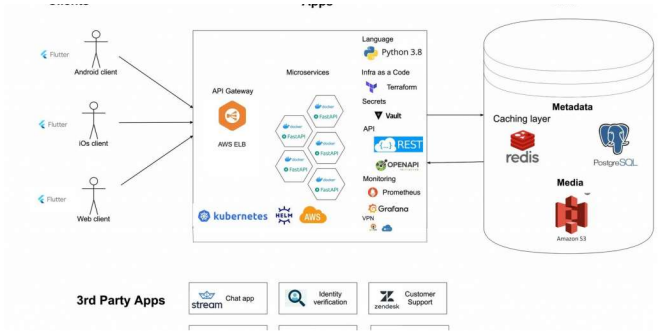
10 min read · Aug 6

302 1

See all from Arslan Ahmad

See all from Level Up Coding

Recommended from Medium



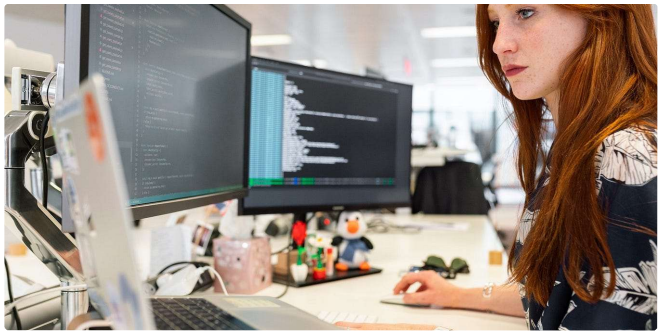
Dmitry Kruglov in Better Programming

The Architecture of a Modern Startup

Hype wave, pragmatic evidence vs the need to move fast

16 min read · Nov 7, 2022

4.5K 45



The Coding Diaries in The Coding Diaries

Why Experienced Programmers Fail Coding Interviews

A friend of mine recently joined a FAANG company as an engineering manager, and...

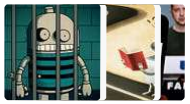
· 5 min read · Nov 2, 2022

6.7K 133

Lists

ChatGPT prompts

24 stories · 264 saves



AI Regulation

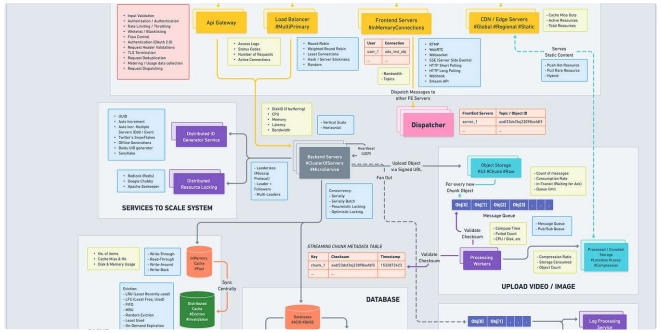
6 stories · 81 saves

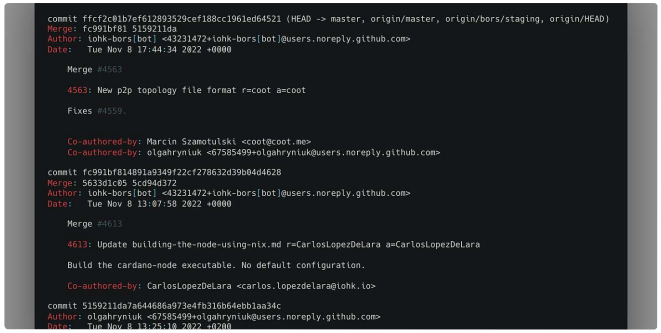
ChatGPT

21 stories · 112 saves

Generative AI Recommended Reading

52 stories · 168 saves





Love Shar... in ByteByteGo System Design Allian...

System Design Blueprint: The Ultimate Guide

Developing a robust, scalable, and efficient system can be daunting. However,...

✦ · 9 min read · Apr 20

7.6K 54

Jacob Bennett in Level Up Coding

Use Git like a senior engineer

Git is a powerful tool that feels great to use when you know how to use it.

✦ · 4 min read · Nov 15, 2022

9K 101

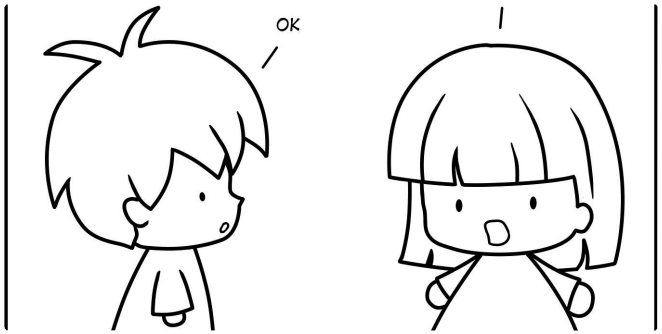
Denat Hoxha

Sharing Data Between Microservices

Robust distributed systems embrace eventual consistency to share data between...

7 min read · Oct 24, 2022

1.1K 26



Julie Zhuo in The Year of the Looking Glass

Average Manager vs. Great Manager

Explained in 10 sketches

2 min read · Aug 11, 2015

21K 189

See more recommendations