Research Paper

What are the optimal design choices for implementing MicroServices in a .NET-based open video-sharing platform, similar to YouTube?

by Radoslav Radev

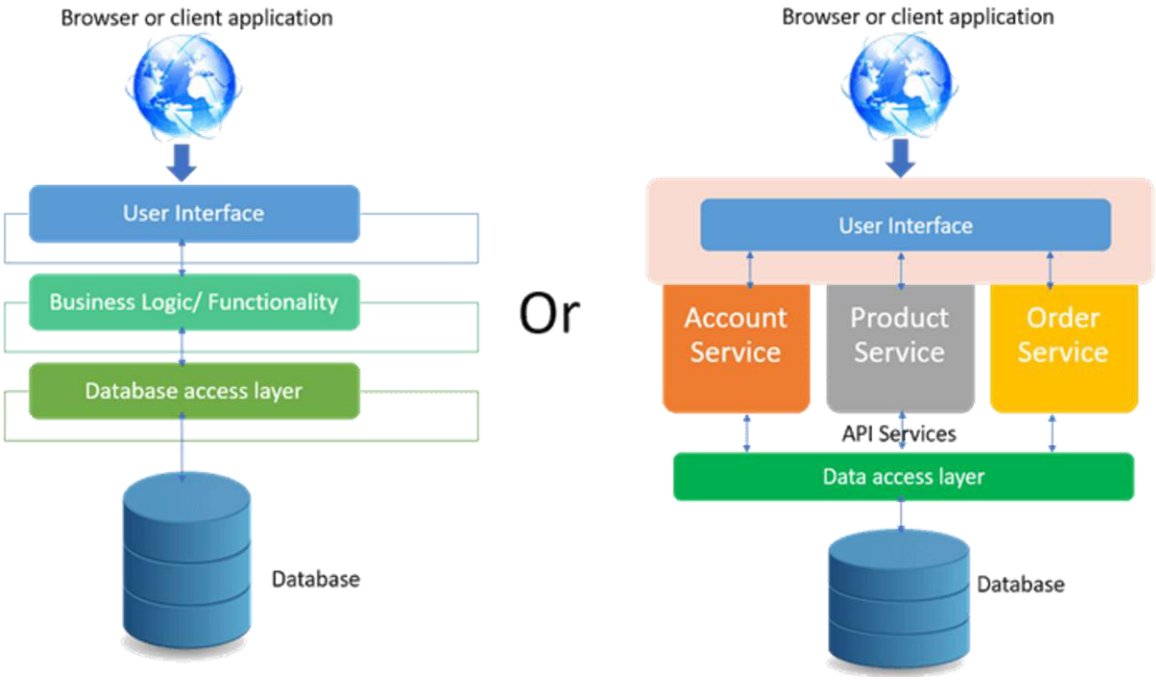
Advanced Software Engineering Semester 6

26/02/2024

ver. 1.0

What do Micro Services offer, and when do they make sense:

in comparison to modular monoliths, Micro Services offer Horizontal Scaling (the ability to scale individual components of the Solution), this comes at the cost of added complexity, and computational expanse in lower-scale applications.



When to use micro services

- When Scalability is needed: Microservices allow for the independent scaling of specific parts of an application, leading to more efficient resource use.
- When Development Speed and Agility are important: They enable faster, independent development, testing, and deployment cycles for different services.
- When Technological Diversity is desired: This architecture supports the use of varying technologies within the same application, optimizing performance for each service.
- When Resilience and Reliability are crucial: The failure of one service in a microservices architecture doesn't necessarily impact the entire application, enhancing overall system stability.

Ref: (MSDN, 13/04/2022), (Bob Reselman,11/01/2022)

The micro Service Architecture relies on a couple of core components:

1. API Gateway:

An API Gateway acts as the front door for all requests from clients to your backend services. It can handle request routing, composition, and protocol translation. popular choices for a .NET environment are:

- **Ocelot**: A lightweight API Gateway built for .NET Core applications. It's easy to configure and extend, making it a good choice for projects where simplicity and performance are key.
- **YARP (Yet Another Reverse Proxy)**: Developed by Microsoft, YARP is a reverse proxy toolkit for building fast proxies in .NET. It's highly customizable and designed to support high-performance scenarios.

Comparison and Consideration:

- **Performance:** YARP is designed with the latest .NET features for high performance, making it potentially more efficient for high-throughput environments.
- **Flexibility:** Ocelot offers extensive flexibility in request routing and configuration, which can be crucial for complex routing needs.
- **Integration:** YARP might provide smoother integration within an Azure Kubernetes Service (AKS) environment and with other Microsoft technologies. Both gateways can integrate with Consul for service discovery, but their approach and ease of integration may differ.
- **Community Support:** Ocelot has a longer history and a robust community, offering a wealth of resources and solutions. YARP, while newer, is backed by Microsoft, ensuring good documentation and support.

Conclusion:

For my project, I chose Ocelot due to its proven flexibility in routing, robust community support, and comprehensive integration capabilities with services like Consul. This choice offers the adaptability and resources needed for complex service orchestration within my microservices architecture.

It`s also wordy to note that in a future cloud production environment usually the API gateway is replaced with a managed service like: Azure API Management

Ref: (ThreeMammals, 20/10/2023), (YARP development team, 02/06/2022), (Milan Jovanović, 7/12/2023)

2. Communication Between Services:

Message Brokers:

Benefits:

- **Asynchronous Communication**: Message brokers excel at decoupling services by enabling asynchronous message-based communication. This is ideal for scenarios where you don't need an immediate response from the other service.
- **Reliability and Scalability**: They provide advanced features like durable messaging, message routing, and delivery acknowledgments, which are crucial for building reliable, scalable systems.
- **Event-Driven Architectures**: Perfect for supporting event-driven architectures, where services react to events rather than being called directly.

Popular Options:

- **RabbitMQ**: A robust, open-source message broker that supports multiple messaging protocols. It's widely used, well-documented, and offers a variety of features that are essential for building complex, distributed systems.
- **Apache Kafka**: Designed for high throughput and scalability, Kafka is more than just a message broker; it's a distributed event streaming platform. It's suitable for scenarios requiring real-time data processing and analytics.

Comparison and Consideration:

- **Flexibility:** RabbitMQ offers superior flexibility in message routing, delivery mechanisms, and support for multiple messaging protocols, including AMQP, MQTT, and STOMP. This makes it particularly well-suited for complex messaging scenarios and integrations that require specific routing logic or patterns. Kafka, on the other hand, focuses on a publish-subscribe model with a straightforward approach to partitioning and distributing messages across a cluster, optimizing for scalability and durability.

- **Integration**: Kafka integrates seamlessly with big data tools and frameworks, making it an excellent choice for applications that require real-time analytics, data lakes, or integration with stream processing engines like Apache Flink or Apache Storm.

RabbitMQ's integration capabilities are broad, given its support for various messaging protocols, making it easier to integrate with a wide range of applications and services, not necessarily limited to the big data ecosystem.

- **Community Support:** Kafka benefits from a large, active community and a rich ecosystem of tools for monitoring, management, and integration, partly due to its association with the Apache Software Foundation. RabbitMQ also boasts a strong community, with a wide array of plugins and tools for different needs.

- **Use Case Appropriateness:** RabbitMQ is particularly effective in applications requiring complex routing, sophisticated message queuing models, or multi-protocol support. Its architectural design offers the flexibility needed for a broad range of applications, from microservices to traditional enterprise integration. Kafka is better suited for applications that require durable message storage, high throughput, and scalable stream processing capabilities, making it the go-to for event sourcing, logging, and real-time analytics applications.

Conclusion:

For my project, I chose RabbitMQ because of its flexibility in supporting different messaging patterns, which perfectly aligns with my project's varied communication requirements. Its user-friendly setup and management streamline deployment and maintenance. Moreover, RabbitMQ's advanced features, such as message acknowledgments and durable exchanges, guarantee reliable message delivery, making it a dependable backbone for my system's messaging needs. This combination of adaptability, ease of use, and reliability makes RabbitMQ the optimal choice for efficiently handling our messaging infrastructure.

Ref: (Tanzu, 18/10/2023), (Simplilearn, 17/10/2023)

Non-Brokered Communication :

- **gRPC:**
    - **High-Performance, Synchronous Communication**: gRPC is designed for low latency and high throughput communication. It uses HTTP/2 for transport, offering advantages like multiplexing multiple requests over a single connection.
    - **Strict API Contracts**: Utilizes Protocol Buffers for defining service contracts, ensuring clear API specifications and compatibility between services.
    - **Streaming Capabilities**: Supports server-side streaming, client-side streaming, and bidirectional streaming, enabling more sophisticated interaction patterns between services.

- **SignalR:**
    - **Real-Time Web Functionality**: SignalR simplifies adding real-time web capabilities to your applications. It enables two-way communication between client and server over WebSockets, with fallbacks for older browsers.
    - **Dynamic, Live Content**: Great for scenarios where we need to push updates from the server to the client in real-time, such as live dashboards, notifications, or collaborative applications.
    - **Simplified Real-Time Communication**: Abstracts the complexity of handling connections, grouping, and broadcasting to clients, making it easier to develop real-time features.
- HTTP Endpoints (Json)

Decision Factors:

- **Communication Pattern**: Choosing message brokers for asynchronous and decoupled communication; gRPC for synchronous, high-performance communication; and SignalR for real-time updates and interactions.
- **Reliability Needs**: If message durability and delivery guarantees are critical, message brokers are the best choice.
- **Performance and Scalability**: gRPC offers significant performance benefits for high-throughput, low-latency communications, making it suitable for performance-critical applications.
- **Real-Time Interaction**: SignalR is the go-to for scenarios requiring real-time client-server communication.

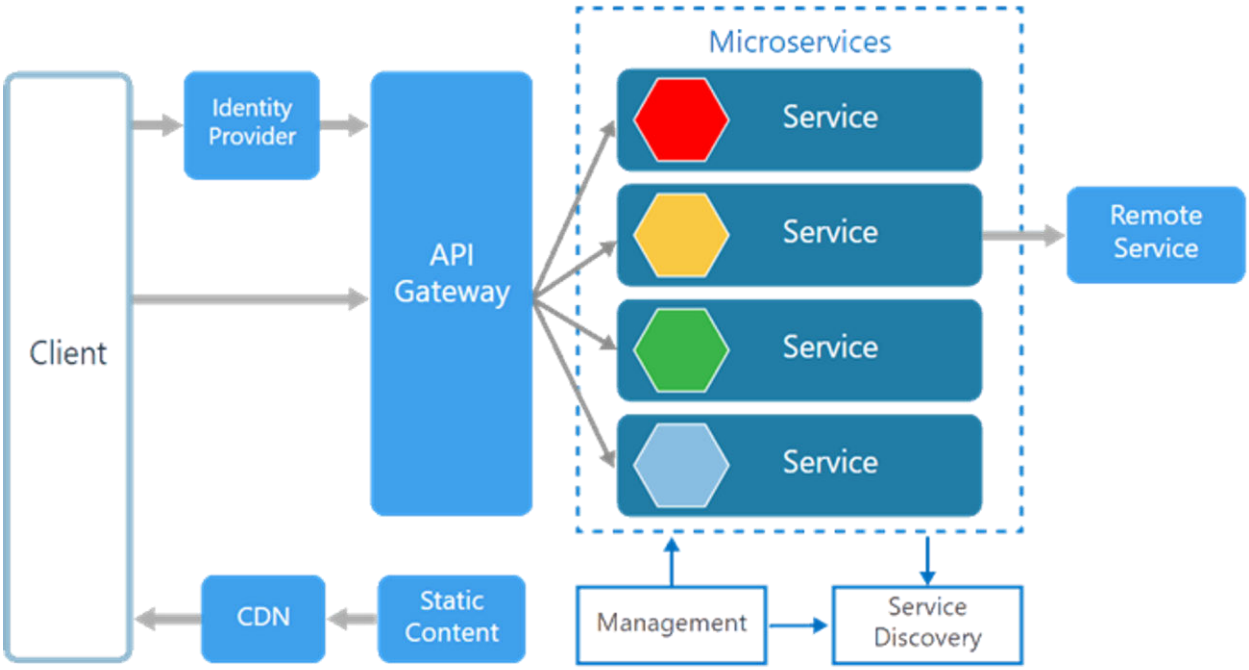
Conclusion:

For my project, I will use a combination of different communication protocols:

- Restful HTTP endpoints for general API use cases
- gRPC for large bandwidth data streaming
- SignalR for real-time communication
- RabbitMQ for reliable communication between the MicroServices, and client requests which result in an Async operation

3. Microservice Discovery and Management

Service discovery tools help services find and communicate with each other in dynamic environments.



. For managing microservices, the options are:

- **Consul**: Offers service discovery, along with health checking, KV storage, and secure service communication. It's a robust solution that can be used across multiple environments.
- **Eureka**: Originally developed by Netflix and part of the Spring Cloud suite for Java, Eureka can also be used in .NET applications through Steeltoe, a toolkit for developing microservices on .NET. Eureka provides service discovery to help microservices find and communicate with each other.

Comparison and Consideration:

- **Performance:** Both Eureka and Consul are designed for high availability and fault tolerance in distributed systems. Eureka focuses on resilience, with features like self-preservation during network failures, making it robust in facing service registry inconsistencies. Consul emphasizes reliable service discovery with integrated health checking to ensure only healthy instances are used, offering a more proactive approach to maintaining service health.

- **Flexibility:** Eureka offers simplicity and ease of use, particularly within the Java ecosystem, with its integration with Spring Cloud providing a seamless experience for developers. Its configuration is straightforward, catering primarily to service registration and discovery. Consul, however, provides a broader set of features beyond service discovery, including key/value storage, service mesh capabilities, and detailed health checks, making it more versatile for complex infrastructures.
- **Community Support:** Eureka, being a part of the Netflix OSS suite, has strong community support, especially among Spring and Java developers. Consul, developed by HashiCorp, benefits from a broad and active community, with extensive documentation and support for a wide array of use cases, from simple service discovery to complex service mesh architectures.
- **Use Cases:** Eureka is best suited for applications deeply integrated with the Spring ecosystem, offering a focused solution for service discovery with less operational complexity. It shines in environments where simplicity and integration with Spring are prioritized. Consul is ideal for more diverse and complex environments that require a comprehensive toolset for service discovery, configuration, and networking across multiple platforms and data centers.

Conclusion:

For my project, I chose Consul for its superior integration with the .NET ecosystem and its extensive feature set that goes beyond what Eureka offers. Consul's comprehensive capabilities such as service discovery, health checking, key/value storage, and service mesh functionalities, make it a versatile and powerful choice for my project.

Ref: (Stackshare, n.d.), (Shivam Roy, 28/03/2022)

References:

The MicroService Architecture:

MSDN (13/04/2022), Design a microservice-oriented application, learn.microsoft.com

Bob Reselman (11/01/2022), *5 design principles for microservices*, developers.redhat.com

API Gateways:

ThreeMammals (the development team of Ocelot) (20/10/2023), Ocelot Docs github.com/ThreeMammals/Ocelot

YARP development team (02/06/2022), YARP documentation, microsoft.github.io

Milan Jovanović (7/12/2023), “Ocelot probably has more features, as YARP is a recent project. However, YARP is more of a reverse proxy. Lightweight. Ocelot brands itself as an API gateway, so it has features that are common for gateways.”

Message brokers:

Tanzu (18/10/2023), Understanding the Differences Between RabbitMQ and Kafka, Tanzu.vmware.com

Simplilearn (17/10/2023), Kafka vs RabbitMQ: Biggest Differences and Which Should You Learn? , simplilearn.com

Microservice Discovery and Management:

Stackshare (Unknown Date), Consul vs Eureka: What are the differences?,stackshare.io

Shivam Roy (28/03/2022), Comparing Eureka vs. Consul, blog.knoldus.com