Microservices Distributed Messaging Service

Overview:

This architecture work comes from the gchq.gov.uk data communication hub, the official website for the UK's Government Communications Headquarters (GCHQ), a cybersecurity and intelligence agency.

- The platform provides information about GCHQ's mission, history, and initiatives to protect national security through signals intelligence and cyber operations. Secondly, it also promotes career opportunities, public engagement, and resources for individuals and businesses to improve cybersecurity awareness and practices.

- The backbone of the earlier architecture was a Monolithic data harvesting hub built with Ruby on Rails that collected metadata from various systems and put them into one big data set for processing.

- The throughput and efficiency of the architecture began to compromise with bottlenecks when availability couldn't meet peeking and spiking rates, leading to BREXIT resulting from high latencies. This affected delivery of insightful report the stakeholders were expecting.

Key Considerations for adopting Microservices:

Architecture - https://www.figma.com/board/4kkSR0KdbgOCJilUwUAVeI/Architectural-Diagrams?node-id=0-1&node-type=canvas&t=dLnC8yVc6OiJvR0u-0

The key considerations for adopting a Microservice were a result of bottlenecks from the Monolith architecture, which included:

- Tight coupling
- Slow Release Cycle
- Manual Releases
- No Continuous Deployment
- Not Scalable
- Lack of log management, observation and Monitoring

The Monolithic Ruby on Rails had to perform many functionalities at the same time. Namely:
- Routing
- Data Harvesting
- Searching and Indexing
- Data persistence
- etc

No Continuous Deployment but Continuous Integration was using Travis and manual Releases using AWS Ops Works. The System was not designed to work in Containers and, hence not scalable. Many times, engineers had to intervene to scale instances in production, and there was a lack of log management and monitoring resulting in countless hours of debugging.

By leveraging tools like RabbitMQ, Amazon SQS, ElasticSearch, DynamoDB (to host jobs), Docker to host the services, Kubernetes for orchestration, Jenkins for deployments, IaC, NodeJS and Express Framework for API, Prometheus, ELK for Monitoring and Observability, I implemented the Microservice architectures where services communicated asynchronously, allowing them to operate independently and reliably.

Core features for the improved architecture were to enable:

1. Data Harvesting
2. Improved SEARCH and discoverability
3. Scalability

My motivation for adopting a Microservices architecture was a result to improve the efficiency, scalability, usability, high availability, and fault tolerance of the data harvesting hub:

- Small individual components
- Easily maintainable
- Parallel development
- Rapid iteration
- Feature addition - enrichment
- Confidence in CI/CD
- Fault-tolerant
- Language-agnostic

The architecture decoupled functionalities, ensuring scalability and maintaining security and observability. RESTful API for data ingestion and transformation, message brokers (e.g., RabbitMQ) for job queues, and worker nodes for processing.

Containers (Docker) were used for service isolation, orchestrated and scaled with Kubernetes. SQS was leveraged for message queuing between dispatchers and workers, while ElasticSearch handled processed job indexing for real-time search. Logging and monitoring were centralised using Prometheus and the ELK stack to ensure visibility into system health and operations. Security was reinforced with rate limiting, authentication mechanisms (e.g., OAuth), and secure communication protocols, especially for third-party integrations.

Challenges and resolution

Challenges encountered include ensuring message reliability, managing schema evolution, and maintaining consistent processing during service failures.

To address these, I implemented idempotent event handling to prevent duplicate processing, used schema registries (e.g., Confluent Schema Registry) to manage versioning, and set up dead-letter queues to handle poison messages.

When migrating a monolithic architecture to microservices, redundant steps and unnecessary components often include the following:

1. **Tightly Coupled Business Logic:**
   Legacy monoliths typically have tightly coupled modules where unrelated functions are interdependent. For example, a payment module might also handle notifications and reporting, leading to redundant dependencies that make the system harder to scale and maintain.

2. **Centralized Databases:**
   A single database shared across all components often results in tightly coupled data models. This can lead to redundant queries and increased complexity when accessing or updating unrelated data.

3. **Overloaded APIs:**
   Monolithic systems often have large, catch-all APIs that return excessive, irrelevant data or require multiple endpoints for simple operations, increasing redundancy and inefficiency.

4. **Duplicated Functionality:**
   In legacy systems, similar tasks (e.g., validation, authentication) are often implemented multiple times across different parts of the application, leading to maintenance overhead and potential inconsistencies.

5. **Hardcoded Configurations:**
   Many monolithic systems rely on hardcoded configurations for workflows and connections, making it difficult to reconfigure or scale parts of the system independently.

6. **Centralized Error Handling:**
   Legacy systems may have a single error-handling mechanism, making it harder to pinpoint and isolate issues within specific functionalities.

Example Unnecessary Components Found in a Migration Process:

- A *reporting module* baked into the main application instead of being decoupled as a separate service.
- Logging mechanisms tied to the monolith's main runtime, making it challenging to implement distributed logging later.
- Redundant data processing pipelines handling similar transformations across different modules.

By identifying these inefficiencies, migrating to microservices allows the system to decouple functionalities into independent, scalable services. For instance, reporting, authentication, and notification services can become separate components, each managed and deployed independently. This not only streamlines operations but also simplifies scaling and future

Monitoring and debugging were enhanced with Prometheus, Grafana, and ELK stacks to track message flow and latency. These approaches have consistently delivered resilient, high-performing systems that met real-time and high-availability requirements.

Best Practices:

- Authentication and Authorization: Implemented OAuth 2.0 for access delegation and API keys with strict scopes and expiration policies for secure access control.
- Certificate Management: Used mutual TLS (mTLS) for bidirectional authentication between the service system and third-party integrations.
- Rate Limiting and Throttling: Mitigate denial-of-service risks by enforcing rate limits on third-party API calls.
- Regular Audits: Conducted periodic security audits of third-party integrations to ensure compliance with security standards and best practices.
- Data Encryption: Sensitive data was encrypted both in transit (TLS, HTTPS) and at rest AES-256.

The outcome

The resulting architecture provided significant benefits: increased scalability with Kubernetes, better fault isolation, faster deployments through Jenkins CI/CD pipelines, and improved search capabilities with ElasticSearch.

The outcome was a robust, scalable, and secure system that ensured high availability and flexibility, enabling easier future feature integrations and updates.

Alexander Adu-Sarkodie
https://github.com/DataSolutionSoftware/Portfolio
https://github.com/kukuu?tab=repositories