
Semana 14 - Aula 1

Tópico Principal da Aula: Integração e Comunicação entre Sistemas

Subtítulo/Tema Específico: Mensageria e Filas

Código da aula: [SIS]ANO2C2B2S14A1

Objetivos da Aula:

- Implementar sistemas de mensageria e filas para comunicação assíncrona.
- Compreender os conceitos de mensageria e filas no contexto de aplicações back-end.
- Entender a importância da comunicação assíncrona para escalabilidade e resiliência de sistemas.

Recursos Adicionais (Sugestão, pode ser adaptado):

- Caderno para anotações;
- Acesso ao laboratório de informática e/ou internet.

Exposição do Conteúdo:

Referência do Slide: Slide 02 - Linguagens de programação back-end / Integração e comunicação entre sistemas / Mensageria e filas

- **Definição:** **Mensageria e filas são padrões de comunicação assíncrona em sistemas distribuídos, onde componentes se comunicam trocando mensagens através de um intermediário (o sistema de mensageria). As mensagens são armazenadas em filas, garantindo que sejam entregues e processadas, mesmo que o sistema receptor não esteja disponível no momento do envio.**
- **Aprofundamento/Complemento:** Em arquiteturas de software modernas, especialmente microsserviços, a comunicação síncrona (como requisições HTTP diretas) pode levar a problemas de acoplamento, resiliência e escalabilidade. A mensageria e filas surgem como uma solução para desacoplar serviços, permitindo que eles operem de forma independente. Isso significa que um serviço pode enviar uma mensagem e continuar suas operações sem esperar uma resposta imediata, e o serviço receptor processará a mensagem quando estiver pronto. Exemplos comuns de sistemas de mensageria incluem RabbitMQ, Apache Kafka, Apache ActiveMQ e Amazon SQS.

RabbitMQ e Apache Kafka são ambos sistemas de mensageria amplamente utilizados, mas com propósitos e arquiteturas distintas. A escolha entre eles depende muito do tipo de problema que você precisa resolver.

Diferenças Principais:

1. Modelo de Mensageria:

- **RabbitMQ:** É um **Message Broker** tradicional, que funciona como um sistema de fila de mensagens. Ele adota um modelo "push" (empurrar), onde o broker envia as mensagens para os consumidores assim que elas chegam. As mensagens são geralmente consumidas por um único destinatário (ponto a ponto), embora também suporte o modelo publish-subscribe (onde múltiplos consumidores podem receber a mesma mensagem). As mensagens são descartadas após serem entregues e confirmadas pelo consumidor.
- **Apache Kafka:** É uma plataforma de **Stream Processing** (processamento de fluxo de eventos). Ele funciona como um "log de eventos distribuído e persistente". O modelo é "pull" (puxar), onde os consumidores buscam as mensagens dos tópicos. Ele é projetado para que múltiplos consumidores possam ler os mesmos dados independentemente, pois as mensagens são anexadas a um log e permanecem lá por um período de retenção configurável.

2. Persistência de Mensagens:

- **RabbitMQ:** As mensagens são tipicamente removidas da fila assim que são consumidas e confirmadas (acknowledged) pelo consumidor. Embora suporte persistência em disco para garantir que as mensagens não sejam perdidas em caso de falha do broker, o foco principal não é a retenção de histórico.
- **Apache Kafka:** As mensagens são persistidas em disco por um período configurável (dias, semanas, meses ou até indefinidamente). Isso permite que os consumidores reprocesssem os dados a qualquer momento dentro desse período, tornando-o ideal para replay de dados e análise histórica.

3. Ordem das Mensagens:

- **RabbitMQ:** Garante a ordem das mensagens dentro de uma única fila.
- **Apache Kafka:** Garante a ordem das mensagens dentro de uma partição de um tópico. Se você tiver múltiplos produtores ou se um tópico tiver múltiplas partições, a ordem global pode não ser garantida, mas a ordem dentro de cada partição é.

4. Performance e Throughput:

- **RabbitMQ:** Geralmente oferece baixa latência para mensagens individuais e pode lidar com milhares de mensagens por segundo. É otimizado para roteamento complexo e entrega garantida de mensagens.
- **Apache Kafka:** Projetado para alto throughput e processamento de milhões de mensagens por segundo em tempo real. Sua arquitetura particionada e distribuída permite escalabilidade horizontal para lidar com volumes massivos de dados.

5. Escalabilidade:

- **RabbitMQ:** É escalável, mas pode exigir mais recursos (nós) para atingir os mesmos volumes de throughput que o Kafka em cenários de dados massivos.
- **Apache Kafka:** Altamente escalável horizontalmente, distribuindo a carga de trabalho entre vários brokers (servidores) no cluster.

6. Complexidade do Roteamento:

- **RabbitMQ:** Oferece opções de roteamento de mensagens mais complexas e flexíveis através de "exchanges" e "bindings", permitindo direcionar mensagens para filas específicas com base em padrões.
- **Apache Kafka:** O roteamento é mais simples, baseado em tópicos e partições. A complexidade do roteamento é geralmente tratada no lado do consumidor ou com o uso de ferramentas de processamento de stream como Kafka Streams.

7. Linguagens de Programação e Protocolos:

- **RabbitMQ:** Suporta uma ampla variedade de linguagens de programação e protocolos (AMQP, MQTT, STOMP).
- **Apache Kafka:** Principalmente com foco em Java e Scala, mas possui clientes para diversas outras linguagens (Python, Node.js, etc.) e utiliza um protocolo binário sobre TCP.

Aplicações e Casos de Uso:

Aplicações do RabbitMQ:

O RabbitMQ é ideal para cenários onde a entrega confiável de mensagens, o roteamento complexo e a comunicação assíncrona são cruciais, especialmente para tarefas de processamento em segundo plano e comunicação entre serviços.

- **Filas de Tarefas Assíncronas (Background Jobs):** Descarregar tarefas demoradas (envio de e-mails, processamento de imagens, geração de relatórios) para serem executadas em segundo plano, sem bloquear a aplicação principal.

- **Comunicação entre Microserviços:** Habilitar a comunicação assíncrona e desacoplada entre diferentes microserviços em uma arquitetura distribuída.
- **Notificações e Alertas:** Envio de notificações em tempo real (push notifications, SMS, e-mails) para usuários.
- **Processamento de Pedidos em E-commerce:** Gerenciar o fluxo de pedidos, atualizações de estoque e processamento de pagamentos de forma confiável.
- **Aplicações de Chat:** Gerenciar o fluxo de mensagens em sistemas de chat em tempo real.
- **Integração de Sistemas Legados:** Conectar sistemas mais antigos que utilizam protocolos específicos.
- **Garantiar Entrega de Mensagens:** Cenários onde a perda de uma única mensagem é crítica e a confirmação de entrega é essencial.

Aplicações do Apache Kafka:

O Apache Kafka é a escolha preferencial para sistemas que lidam com grandes volumes de dados em tempo real, exigem processamento de fluxo contínuo e a capacidade de reprocessar dados históricos.

- **Processamento de Fluxos de Dados em Tempo Real:** Captura, processamento e análise de grandes volumes de dados em tempo real, como dados de sensores IoT, telemetria, logs de aplicações, etc.
- **Rastreamento de Atividades de Usuário (Website Activity Tracking):** Registrar cliques, visualizações de página, pesquisas e outras interações do usuário para análise em tempo real e construção de perfis.
- **Agregação de Logs e Métricas:** Coletar e centralizar logs e métricas de diversas fontes para monitoramento operacional, análise de segurança (SIEM) e diagnóstico de problemas.
- **Pipelines de Dados:** Construir pipelines de dados escaláveis para mover dados entre diferentes sistemas (por exemplo, de bancos de dados transacionais para data warehouses ou data lakes).
- **Sistemas de Detecção de Fraudes:** Analisar transações em tempo real para identificar padrões suspeitos e prevenir fraudes.
- **Recomendações em Tempo Real:** Gerar recomendações personalizadas para usuários com base em seu comportamento atual.
- **Event Sourcing:** Armazenar todas as mudanças de estado de uma aplicação como uma sequência de eventos imutáveis, permitindo reconstruir o estado em qualquer ponto no tempo.

Em resumo, se você precisa de um "carteiro" confiável que entregue mensagens para destinos específicos com roteamento complexo e garantias de entrega, o RabbitMQ é uma ótima opção. Se você precisa de um "gravador de eventos" de alta performance que armazena fluxos de dados para processamento em tempo real e análise histórica, o Apache Kafka é a escolha mais adequada. É importante notar

que em algumas arquiteturas complexas, eles podem até ser usados em conjunto, aproveitando os pontos fortes de cada um.

- **Exemplo Prático: Imagine um sistema de e-commerce. Quando um cliente faz um pedido, o serviço de pedidos envia uma mensagem para uma fila (ex: "pedidos_processar"). Outros serviços, como o de estoque, o de pagamento e o de envio, ouvem essa fila. O serviço de estoque pode consumir a mensagem para verificar a disponibilidade dos produtos, o de pagamento para processar a transação e o de envio para preparar a logística. Se o serviço de pagamento estiver temporariamente offline, a mensagem do pedido permanece na fila até que ele volte a operar, evitando que o pedido seja perdido.**
- **Links de Vídeos:**
  - **Processamento assíncrono com mensageria - Escalando aplicações Web | Dias de Dev**
  - **https://youtu.be/Rx80QRZRgHc?si=3n05CHGpjy2x5ovi**
  -
  - **Kafka (Plataforma de Mensageria e Streaming) // Dicionário do Programador**
  - https://youtu.be/qOqXz5Qv_-8?si=9spVSIOX8XBoTb89

Semana 14 - Aula 2

Tópico Principal da Aula: Integração e Comunicação entre Sistemas

Subtítulo/Tema Específico: Integração com serviços externos

Código da aula: [SIS]ANO2C2B2S14A2

Objetivos da Aula:

- Integrar aplicações back-end com serviços externos.
- Compreender os desafios e melhores práticas na integração com APIs e serviços de terceiros.
- Desenvolver a capacidade de consumir e prover APIs RESTful para comunicação entre sistemas.

Recursos Adicionais (Sugestão, pode ser adaptado):

- Caderno para anotações;
- Acesso ao laboratório de informática e/ou internet.

Exposição do Conteúdo:

Referência do Slide: Slide 02 - Linguagens de programação back-end / Integração e comunicação entre sistemas / Integração com serviços externos

- **Definição: A integração com serviços externos refere-se à capacidade de uma aplicação back-end de se comunicar e interagir com sistemas, APIs ou plataformas que não fazem parte do seu próprio ambiente de execução, mas que oferecem funcionalidades ou dados necessários para o seu funcionamento.**
- **Aprofundamento/Complemento:** No desenvolvimento de software moderno, é raro que uma aplicação funcione de forma completamente isolada. A integração com serviços externos é crucial para estender funcionalidades (ex: serviços de pagamento, APIs de mapas, envio de SMS), obter dados (ex: dados climáticos, informações de bancos) ou interagir com sistemas legados. Os métodos de integração mais comuns incluem o uso de APIs (Application Programming Interfaces), como REST (Representational State Transfer) e SOAP (Simple Object Access Protocol), e, em menor grau, transferências de arquivos ou integrações diretas com bancos de dados. A segurança (autenticação, autorização), tratamento de erros (timeouts, retries) e monitoramento são aspectos críticos a serem considerados na integração.
- **Exemplo Prático:** Um aplicativo de entrega de comida precisa integrar-se com um serviço de mapas externo (como Google Maps ou OpenStreetMap) para calcular distâncias e estimar tempos de entrega. O aplicativo envia as coordenadas do restaurante e do cliente para a API do serviço de mapas, que retorna as informações necessárias. Além disso, pode integrar-se com um gateway de pagamento (como Stripe ou PagSeguro) para processar as transações financeiras.

API Externa

Uma **API externa** (Interface de Programação de Aplicações) é um conjunto de regras e ferramentas que permite que diferentes softwares se comuniquem e interajam entre si, mesmo que tenham sido desenvolvidos por empresas distintas. Pense nela como um "garçom" que leva seu pedido (uma solicitação) para a cozinha (o software externo) e traz a resposta (os dados ou a funcionalidade).

Exemplo prático: Quando você usa um aplicativo de viagens como o Kayak ou o Decolar, ele não tem todos os preços de voos e hotéis em seu próprio banco de dados. Em vez disso, ele usa **APIs externas** de companhias aéreas (LATAM, GOL, etc.) e redes de hotéis (Hilton, Accor) para buscar informações em tempo real e mostrar a você as opções disponíveis. O aplicativo faz uma "chamada" para a API da companhia aérea solicitando os voos de São Paulo para o Rio de Janeiro em uma data específica, e a API retorna os resultados.

Notificações de Status via Webhook

São uma forma de comunicação automatizada entre sistemas, onde um serviço envia informações (uma "notificação") para outro serviço sempre que um evento específico ocorre. Em vez de você ter que "perguntar" constantemente se algo aconteceu, o webhook te "avisa" proativamente.

Exemplo prático: Imagine que você tem uma loja online e usa um serviço de pagamentos (como o Stripe ou o PagSeguro). Quando um cliente finaliza uma compra e o pagamento é aprovado, o serviço de pagamentos pode enviar uma **notificação de status via webhook** para o sistema da sua loja. Essa notificação contém informações sobre o pagamento bem-sucedido, e o sistema da sua loja então processa o pedido, atualiza o status do estoque e envia um e-mail de confirmação para o cliente. Você não precisa ficar verificando manualmente se o pagamento foi aprovado; o webhook faz isso por você.

Armazenamento de Imagens de Produtos em um Serviço em Nuvem

Refere-se à prática de guardar as fotos dos seus produtos (ou qualquer outro arquivo) em servidores remotos, gerenciados por um provedor de nuvem (como a Amazon Web Services - AWS, Google Cloud ou Microsoft Azure), em vez de armazená-los diretamente no seu próprio servidor local.

Exemplo prático: Se você tem um e-commerce com milhares de produtos, cada um com várias imagens, armazenar todas essas fotos no seu próprio servidor pode ser caro, lento e exigir muita manutenção. Ao utilizar um serviço em nuvem como o **Amazon S3** (Simple Storage Service) ou o **Google Cloud Storage**, você faz o upload das imagens para esses serviços. Quando um cliente acessa a página de um produto na sua loja, as imagens são carregadas diretamente da nuvem, o que garante:

- **Alta disponibilidade:** As imagens estão sempre acessíveis.
- **Escalabilidade:** Você pode armazenar um volume ilimitado de imagens.
- **Performance:** As imagens carregam rapidamente para os usuários, independentemente de onde eles estejam.
- **Segurança:** Os provedores de nuvem oferecem robustos recursos de segurança.

Links de Vídeos:

- **Integração entre sistemas**
- https://youtu.be/46Fss6sRZ6Y?si=vKpi16oK87gBfAH3

Semana 14 - Aula 3

Tópico Principal da Aula: Integração e Comunicação entre Sistemas

Subtítulo/Tema Específico: Orquestração de microsserviços

Código da aula: [SIS]ANO2C2B2S14A3

Objetivos da Aula:

- Implementar orquestração de microsserviços para coordenação de tarefas.
- Compreender os conceitos e padrões de orquestração em arquiteturas de microsserviços.
- Analisar as vantagens e desvantagens da orquestração versus coreografia de microsserviços.

Recursos Adicionais (Sugestão, pode ser adaptado):

- Caderno para anotações;
- Acesso ao laboratório de informática e/ou internet.

Exposição do Conteúdo:

Referência do Slide: Slide 02 - Linguagens de programação back-end / Integração e comunicação entre sistemas / Orquestração de microsserviços

- **Definição:** **A orquestração de microsserviços é um padrão de coordenação onde um serviço central (o "orquestrador") é responsável por coordenar a execução de uma sequência de operações entre múltiplos microsserviços para completar um processo de negócio. O orquestrador define a ordem e as condições sob as quais cada microsserviço deve ser chamado, aguardar suas respostas e decidir o próximo passo.**
- **Aprofundamento/Complemento:** Em uma arquitetura de microsserviços, um processo de negócio complexo pode exigir a interação de vários serviços. **A orquestração é uma abordagem para gerenciar essa complexidade, onde um serviço "maestro" (o orquestrador) envia comandos para os serviços "escravos" e espera por suas respostas.** Isso contrasta com a coreografia, onde os serviços interagem uns com os outros de forma independente, reagindo a eventos. A orquestração oferece maior controle centralizado sobre o fluxo de trabalho, facilitando o rastreamento e a depuração. No entanto, pode criar um ponto único de falha e acoplamento entre o orquestrador e os serviços. Ferramentas como o Apache Camel, Netflix Conductor e orquestradores de contêineres como Kubernetes são usados para gerenciar e coordenar microsserviços.

- **Exemplo Prático:** Em um sistema de processamento de pedidos online, a orquestração pode funcionar da seguinte forma: o orquestrador de pedidos recebe um novo pedido. Ele chama o serviço de validação de estoque, depois o serviço de pagamento, em seguida o serviço de envio e, por fim, o serviço de notificação do cliente. Se qualquer um desses passos falhar, o orquestrador é responsável por lidar com a falha (ex: reverter a transação, notificar o usuário) ou tentar novamente.

O Kubernetes (também conhecido como K8s) é um sistema de orquestração de contêineres de código aberto que automatiza o deployment, escalonamento e gerenciamento de aplicações conteinerizadas. Originalmente desenvolvido pelo Google, ele se tornou a ferramenta padrão da indústria para gerenciar workloads em ambientes de nuvem e on-premises.

Kubernetes e a Orquestração de Contêineres

A orquestração de contêineres é o processo de automatizar o ciclo de vida dos contêineres, desde o deployment até o escalonamento e o gerenciamento da rede. Em um ambiente de microsserviços, onde as aplicações são divididas em componentes menores e independentes, a orquestração manual se torna impraticável. O Kubernetes entra em cena para simplificar essa complexidade, oferecendo recursos como:

- **Automação de Deployment:** O Kubernetes permite descrever o estado desejado da sua aplicação usando arquivos YAML ou JSON. Ele então garante que esse estado seja mantido, implantando os contêineres necessários, configurando as redes e gerenciando os logs.
- **Escalonamento:** Ele pode aumentar ou diminuir automaticamente o número de instâncias de um serviço (pods) com base na demanda, garantindo que sua aplicação possa lidar com picos de tráfego.
- **Gerenciamento do Ciclo de Vida:** O Kubernetes monitora a saúde dos contêineres e dos nós, reiniciando-os automaticamente em caso de falha e garantindo a alta disponibilidade da aplicação.
- **Service Discovery:** Ele atribui endereços IP e nomes DNS estáveis aos serviços, permitindo que os microsserviços se comuniquem uns com os outros sem a necessidade de conhecer seus endereços IP físicos.
- **Balanceamento de Carga:** O Kubernetes distribui o tráfego entre as instâncias disponíveis de um serviço, garantindo que nenhum pod seja sobrecarregado.
- **Armazenamento Persistente:** Ele oferece mecanismos para que os contêineres acessem armazenamento persistente, o que é crucial para aplicações que precisam de dados permanentes, como bancos de dados.

- **Como o Kubernetes Ajuda a Gerenciar Serviços em uma Arquitetura de Microsserviços**

  Em uma arquitetura de microsserviços, cada funcionalidade da aplicação é um serviço independente. O Kubernetes é ideal para gerenciar esses serviços devido aos seguintes pontos:

- **Isolamento:** Cada microsserviço pode ser empacotado em seu próprio contêiner, garantindo isolamento de dependências e ambientes. O Kubernetes gerencia a execução desses contêineres em pods, a menor unidade implantável no Kubernetes.
- **Escalonamento Independente:** Microsserviços podem ser escalados independentemente com base em suas próprias demandas de recursos. Se o serviço de autenticação tiver um pico de uso, apenas ele pode ser escalado, sem afetar outros serviços.
- **Resiliência:** O Kubernetes oferece recursos de auto-recuperação (self-healing). Se um contêiner ou nó falhar, o Kubernetes detecta a falha e reinicia os contêineres afetados em outros nós saudáveis, minimizando o tempo de inatividade.
- **Desacoplamento:** Microsserviços se comunicam via APIs bem definidas, e o Kubernetes fornece o Service Discovery e o balanceamento de carga necessários para que essa comunicação ocorra de forma transparente. Isso elimina a necessidade de codificar endereços IP diretamente na aplicação.
- **Deployments Simplificados:** Com os Deployments do Kubernetes, é possível realizar atualizações de forma gradual (rolling updates) ou com rollback automático em caso de falha, minimizando o impacto no usuário final.
- **Gerenciamento de Configuração e Segredos:** O Kubernetes oferece ConfigMaps para dados de configuração não sensíveis e Secrets para informações sensíveis (senhas, chaves de API), permitindo que essas informações sejam gerenciadas de forma centralizada e segura.

Planejamento da Configuração Inicial de Cada Serviço no Kubernetes

Vamos planejar a configuração inicial de um serviço de exemplo em uma arquitetura de microsserviços. Imagine um aplicativo de e-commerce com os seguintes serviços: `servico-autenticacao`, `servico-produtos` e `servico-pedidos`.

Para cada serviço, usaremos os seguintes objetos Kubernetes:

1. **Deployment:** Descreve como os pods do serviço devem ser criados e mantidos.
2. **Service:** Define como acessar o serviço dentro e fora do cluster.
3. **ConfigMap (Opcional):** Para configurações não sensíveis.

4. **Secret (Opcional):** Para informações sensíveis.
5. **HorizontalPodAutoscaler (HPA):** Para escalonamento automáticos.

- **Links de Vídeos:**
- **Microservices, o que usar? (Coreografia ou Orquestração)**
- **https://youtu.be/QwDY0wla13Y?si=2cl2n890jeAO6xDM**
-