INSTITUTO FEDERAL DE CIÊNCIA E TECNOLOGIA DE PERNAMBUCO

Campus Jaboatão

Departamento de Computação

Curso de Especialização em Desenvolvimento, Inovação e Tecnologias Emergentes

MÁVIA FERREIRA LIMA

ESTUDO DE CASO PARA SOLUÇÃO DE INTEGRAÇÃO COM GRANDES VOLUMES DE DADOS UTILIZANDO PADRÕES DE INTEGRAÇÃO EMPRESARIAL COM O USO DO APACHE CAMEL.

Recife

2021

MÁVIA FERREIRA LIMA

ESTUDO DE CASO PARA SOLUÇÃO DE INTEGRAÇÃO COM GRANDES VOLUMES DE DADOS UTILIZANDO PADRÕES DE INTEGRAÇÃO EMPRESARIAL COM O USO DO APACHE CAMEL.

Trabalho de conclusão de curso de Especialização lato sensu em Desenvolvimento, Inovação e Tecnologias Emergentes do Instituto Federal de Ciência e Tecnologia de Pernambuco, como requisito para parcial para obtenção do título de Especialista.

Orientador: Prof. Josino Rodrigues Neto

Coorientador: Prof. Nilson Cândido de Oliveira Júnior

Recife

2021

FICHA CATALOGRÁFICA

L732e

Lima, Mávia Ferreira.

Estudo de caso para a solução de integração com grandes volumes de dados utilizando padrões de integração empresarial com o uso do Apache Camel. / Mávia Ferreira Lima; Orientador Prof. Josino Rodrigues Neto; coorientador Prof. Nilson Cândido de Oliveira Júnior. Jaboatão dos Guararapes, 2021.

107f.; il.

Trabalho de Conclusão de Curso (Especialização em Desenvolvimento, Inovação e Tecnologias Emergentes) – IFPE - Campus Jaboatão dos Guararapes.

Inclui Referências.

1. Integração de aplicações corporativas (Sistemas de computação) 2. Padrões de Software 3. Processamento de dados I. Lima, Mávia Ferreira. II. IFPE. III. Título.

CDD 004.21

Catalogação na fonte elaborada pelo Bibliotecário Thiago Leite Amaro da Silva - CRB4/ 2.021

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
PERNAMBUCO

Instituto Federal de Educação, Ciência e Tecnologia de Pernambuco
Campus Jaboatão dos Guararapes
Divisão de Pesquisa e Extensão e Pós-graduação

ATA DE REALIZAÇÃO DE BANCA

No dia **30** de **março** de **2021** às **17h** de forma remota, compareceram a banca de defesa do Trabalho de Conclusão de Curso da Especialização *lato sensu* em **Inovação e Tecnologias Emergentes,** o aluno **Mavia Ferreira Lima,** onde defenderam o trabalho intitulado **ESTUDO DE CASO PARA SOLUÇÃO DE INTEGRAÇÃO COM GRANDES VOLUMES DE DADOS UTILIZANDO PADRÕES DE INTEGRAÇÃO EMPRESARIAL COM O USO DO APACHE CAMEL**, os professores que compõem a banca descrita abaixo, e concederam a nota 7,5 sendo o(a) aluno(a) considerado(a) **APROVADO** de acordo com a composição das notas estabelecida pela banca avaliadora.

COMPOSIÇÃO DA BANCA		
	NOTA	**ASSINATURA**
Josino Rodrigues Neto (presidente da banca)	8	gov.br Documento assinado digitalmente Josino Rodrigues Neto Data: 24/04/2021 18:16:11-0300 CPF: 007.550.423-52
Roberto Alencar (1 avaliador)	7	gov.br Documento assinado digitalmente Roberto Luiz Sena de Alencar Data: 25/04/2021 17:52:29-0300 CPF: 055.677.114-17
Julio Cesar Damasceno (2 avaliador)	7,5	gov.br Documento assinado digitalmente Julio Cesar Damasceno Data: 25/04/2021 16:20:07-0300 CPF: 836.514.703-34
NOTA FINAL	7,5	

Documento assinado digitalmente
gov.br
Mavia Ferreira Lima
Data: 18/05/2021 12:20:25-0300
CPF: 591.448.664-53

Mavia Ferreira Lima

Documento assinado digitalmente
gov.br
Nilson Candido de Oliveira Jr
Data: 14/05/2021 15:53:06-0300
CPF: 031.073.834-22

Nilson Cândido de Oliveira Júnior
Coordenador - Pós Graduação
SIAPE: 1829625

AGRADECIMENTOS

Agradeço a meus pais, e minha família pela compreensão e apoio, também agradeço a todos que contribuíram de algum modo no desenvolvimento deste trabalho, e um agradecimento especial ao corpo docente pela dedicação na transmissão do conhecimento.

RESUMO

Este trabalho se propõe a apresentar a utilização de padrões de integração corporativos como solução para o tratamento e transformação de grandes volumes de dados em aplicações distribuídas e heterogêneas, sendo um dos principais desafios: construir uma solução para um problema real de gestão e manipulação de grandes volumes de dados, com um menor tempo de desenvolvimento e com um menor custo de recursos e valor. Neste contexto, objetiva-se apresentar um estudo de caso que utiliza padrões de integração empresarial: Message Routing, Splitter e Load Balancer, como solução implementada para uma aplicação de um caso real, que manipula grandes volumes de dados, oriundos de sistemas bancários, utilizando o Apache Camel como uma alternativa viável e de baixo custo. Procurou-se utilizar um ambiente virtualizado e escalável com plataforma Java/Spring Boot/Kubernetes. Como resultado foi observado que com a utilização do Apache Camel, possibilita trazer benefícios reais com um rápido desenvolvimento, cerca de 68,75%, mais rápido que uma implementação que não utiliza o Apache Camel, trazendo ganhos como: maior coesão de código, através de uma abordagem mais escalável e segura. Buscou-se estruturar este documento de forma a apresentar um roteiro da implementação, para que sirva como ponto de partida, um guia, para futuros desenvolvedores e acadêmicos, para implementação de problemas de integração de aplicações com grandes volumes de dados.

Palavras-chave: Integração de aplicações corporativas (Sistemas de computação). Padrões de Software. Processamento de dados.

ABSTRACT

This work aims to present the use of corporate integration standards as a solution for the treatment and transformation of large volumes of data in distributed and heterogeneous applications. One of the main challenge is building a solution to solve a real problem of management and handling large volumes of data, with less development time and less cost of resources and pricing. In this context, the objective is to present a case study that uses enterprise integration patterns such as Message Routing, Splitter and Load Balancer, as a solution implemented for an application of a real case, that deal large volumes of data, originating from banking systems, and using the Apache Camel as a viable and low cost option for this purpose. A scalable and virtualized environment was used with the java / spring boot / Kubernetes platform. As a result, it was observed that with the use of Apache Camel, it was possible to bring real benefits with a rapid development, about 68.75% faster than an implementation that doesn't use Apache Camel, with greater code cohesion, through an implementation more scalable and secure. This document was structured in order to present an implementation roadmap, so that it serves as a starting point, or a guide, for future developers and academics, for similar problems of implementing application integration with large volumes of data.

Keywords: Enterprise application integration (Computer systems). Software patterns. Data processing.

LISTA DE FIGURAS

LISTA DE ABREVIAÇÕES, ACRÔNIMOS E SIGLAS

A2A - Application to Application

API - Application Programming Interface

B2B - Business to Business

CRM - Customer Relationship Management

DSL - Domain Specific Language

EIA - Enterprise Application Integration

EIP - Enterprise Integration Patterns

ESB - Enterprise Service Bus

FTP - File Transfer Protocol

IDL - Interface Definition Language

IoT - Internet of Things

JMS - Java Message Service

MEP - Message Exchange Pattern

MOM - Message Oriented Middleware

POJO - Plain Old Java Object

RPC - Remote Procedure Call

SI - Sistema de Informação

SOAP - Simple Object Access Protocol

SQS - Simple Queue Service

TCP/IP - Transmission Control Protocol/Internet Protocol

TI - Tecnologia da Informação

TIC - Tecnologia de Informação e Comunicação

URI - Uniform Resource Identifier

SUMÁRIO

1 INTRODUÇÃO

A Web tem cada vez mais se consolidado entre os mais diversos ramos de negócios, seja público ou privado, devido à facilidade de acesso aos dados e ao grande volume de informações disponíveis na Web. Este grande volume de dados e a facilidade de acesso aos mesmos, revolucionaram o desenvolvimento dos sistemas de informação e motivaram o crescimento de aplicações que frequentemente necessitam integrar dados heterogêneos e distribuídos em diferentes fontes de dados. (CAROLINA SALGADO BERNADETTE FARIAS LÓSCIO, 2020)

A crescente demanda das organizações para integrar os sistemas de informações, se justificam por diversos fatores, entre eles: aumento na diversidade e quantidade de sistemas, busca de vantagens competitivas com uma melhor gestão da informação, exigências de órgãos reguladores, maior agilidade no trâmite de informações e tendência por trabalhos organizados de forma colaborativa, que exigem cada vez mais um melhor fluxo organizacional entre as organizações. Diante destes fatores, do aspecto crítico e da importância para as organizações, são desenvolvidas e apresentadas pelos meios acadêmicos, pesquisadores e praticantes, diversas abordagens sistemáticas para integração entre sistemas, que envolvem: técnicas, conceitos, profissionais, ferramentas e metodologias específicas, culminando num ambiente adequado à gestão das integrações ou camada de integração.(SORDI, DE; MARINHO, 2007)

Primeiramente deve-se entender o conceito de integração, para entender os problemas relacionados com a integração de sistemas para grandes volumes de dados. Como também a abordagem de algumas características sobre o problema de integração de dados, questão que já foi amplamente discutida no contexto de bancos de dados, especificamente quando foram analisados os aspectos relativos à interoperabilidade de sistemas heterogêneos. E mais recentemente, este problema foi ampliado abordando o contexto da Web conforme citado pelo trabalho desenvolvido por Carolina Salgado.

Muitos dos problemas encontrados na construção dos sistemas de integração de dados na Web são similares aos problemas encontrados em sistemas de integração de bancos de dados tradicionais. Entretanto, existem algumas diferenças

que precisam ser consideradas quando fontes de dados Web são integradas: - grandes números de fontes de dados, que dificultam os processos de integração e resolução de conflitos; - quando fontes de dados são muito dinâmicas e assim a adição ou remoção destas deve ser feita de maneira a minimizar o impacto na visão integrada; como também, - quando há fontes de dados muito heterogêneas (desde sistemas de gerenciamento de bancos de dados até simples arquivos); e finalmente, - quando há fontes de dados não estruturadas ou semi-estruturadas, muitas vezes sem informações suficientes para a integração." (CAROLINA SALGADO BERNADETTE FARIAS LÓSCIO, 2020).

Como a questão da integração não é algo novo, diversas técnicas foram desenvolvidas nas últimas décadas, cabe então a escolha da técnica ou padrão mais apropriado para cada necessidade ou problemática. Para isto é importante conhecer e aprofundar os estudos nos EIP – ENTERPRISE INTEGRATION PATTERNS (HOHPE; WOOLF, 2002), que traz diversos modelos de implementação de integração. Pode-se implementar os 'padrões' a partir de qualquer linguagem de programação, todavia para cada estratégia de implementação é importante considerar alguns fatores: - recursos disponíveis, custo, pré-requisitos, framework, ambiente tecnológico, entre outros.

Um grande desafio seria implementar uma solução de integração para uma grande volumetria de dados, gerados de forma frequente e constante, e que reunisse os benefícios da adoção de padrões de projeto, atuando entre plataformas e aplicações distintas, com um menor custo, tanto de consumo de recursos quanto de custo financeiro, como também que pudesse permitir escalonar/alocar filas em tempo de execução. Neste contexto, seguindo os pré-requisitos apontados pelo cliente após estudo prévio realizado, foi considerado o Apache Camel como um meio de facilitar o processo de codificação, uma vez que este traz intrinsecamente, quase que de 'um-para-um', os modelos apresentados em *Enterprise Integration Patterns* de Gregor Hohpe e Bobby Woolf, permitindo implementar problemas complexos de integração corporativos, oferecendo grande flexibilidade para adequar às necessidades e com um baixo custo, por ser *open source*.

O restante deste trabalho está estruturado da seguinte forma: no capítulo 2 é apresentada uma contextualização do problema, seguido dos objetivos no capítulo 3, e da metodologia adotada (capítulo 4). No capítulo 5 são apresentados alguns

conceitos básicos e motivações relacionadas à integração de sistemas, bem como à adoção de alguns padrões do *Enterprise Integration Patterns* como também um pouco sobre o Apache Camel e como ele traz facilidade e agilidade no desenvolvimento, além dos padrões que ele já incorpora em seu framework. O capítulo 6 apresenta a descrição do experimento, onde é descrita a configuração, implementação e testes unitários da solução, como também alguns dos principais problemas identificados durante a implementação, apresentando as soluções propostas para cada caso e citando as respectivas referências utilizadas em sua construção.

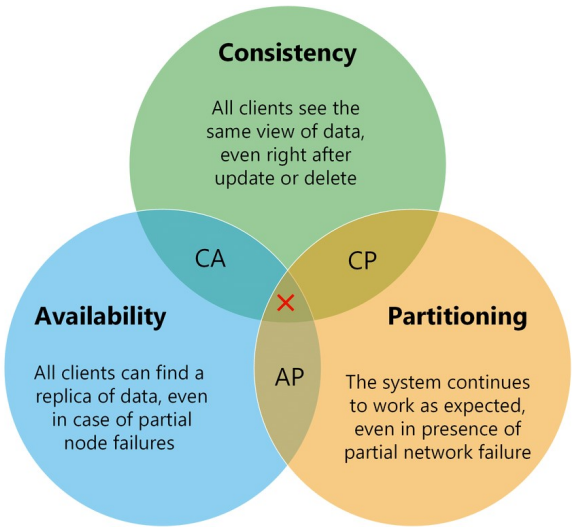
Finalmente no capítulo 7 é apresentada uma breve conclusão sobre o estudo realizado e dos resultados obtidos, com os aprendizados e contribuições bem como sugestões para trabalhos futuros.

Ao final consta uma seção de anexos com os principais trechos de códigos implementados.

2 CONTEXTUALIZAÇÃO DO PROBLEMA ESPECÍFICO

Como cenário principal tem-se um cliente bancário que possui um ambiente heterogêneo, com aplicações CRM que utilizam protocolo SOAP-UI e REST-API, necessitando integrar-se com outras aplicações CRM, orquestradas em Kubernetes. A expectativa de volume de dados é da ordem de 190,73 Gbytes de dados gerados, a partir de 200 milhões de registros oriundos da base de dados de clientes. Devendo ser disparadas campanhas com blocos de 50 mensagens por disparos para os clientes com um *throughput* (taxa em que os dados são transmitidos) 900 Gbps, totalizando 45.000 linhas de disparo por segundo, distribuídos em 3 filas de disparos em paralelo. Então entre 1.2 a 3 horas há uma expectativa de envio de 200 milhões de registros com um tamanho total de 190 Gigabytes.

Figura 1 CAP - *Theorem*



Fonte: (DANNY ZHANG, 2020)

Foram consideradas e analisadas possíveis estratégias de soluções, sobre a ótica do teorema de CAP, conforme **Figura 1.** CAP - *Theorem*, utilizando Kafka, SQS AWS para o disparo das mensagens, associados a funções lambda ou micro-serviços implementados em Java. Algumas destas soluções foram logo descartadas por questões de limitações, expostas pelo cliente:

- O Lambda AWS possui limites quanto ao tempo de processamento (15min), bem como, alocação de memória e poder de processamento, não sendo aconselhado para processamentos massivos, quanto maior o poder de

processamento e alocação de memória maior o custo, sendo mais indicado para processamentos menores, com volumes de recursos menores;

- O Kafka para mensageria foi descartado pois, embora realize o processamento em paralelo não garante a consistência. Como proposta de solução em substituição ao Kafka, foi escolhido o SQS AWS que permite o enfileiramento de mensagens com maior garantia de consistência, e vários *consumers* na mesma fila, e para garantir o paralelismo foi optado em aumentar o número de SQSs atuando em paralelo inicialmente sendo considerada 3 filas, o que poderia ser posteriormente facilmente escalonado;
- Mule Software solução de integração, embora possua uma boa quantidade de conectores não é totalmente *free*, possui apenas uma parte gratuita;
- Spring Integration apresenta um conjunto de conectores limitados. O Camel possui uma gama de conectores muito superior.

Neste contexto o Apache Camel foi colocado como premissa de opção de solução mais adequada e com menor custo para implementação de micro-serviço. Desta forma possibilita o maior controle para balanceamento de campanhas, cargas e/ou *consumers* em paralelo e que permite o escalonamento, e alocação de filas em tempo de execução.

2.1 Ambiente de Desenvolvimento/Tecnologias

- Sistema operacional Windows 10 64bits x64;
- Spring Tools 4.5.1 Release (STS - https://spring.io/tools);
- Java 11;
- Apache Camel 2.23.1;
- Spring Boot 2.2.26. RELEASE;
- JUnit 4 e JUnit 5;
- AWS-S3;
- AWS-SQS;
- Kubernetes;

2.2 Atividades Realizadas

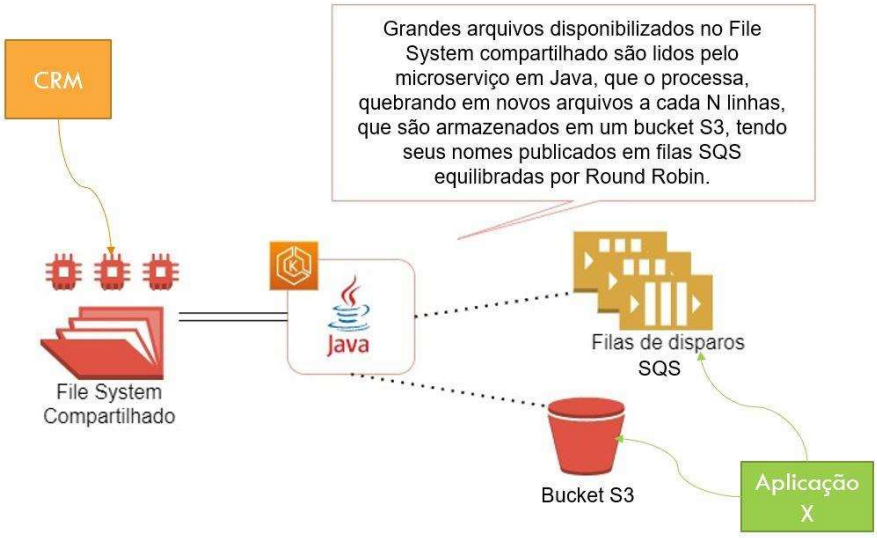
- Plano de atividades da implementação
    - Para um melhor controle da construção da solução desmembramos em tarefas. Sendo elencadas:
        - Estudo dos padrões de integrações implementados pelo Apache Camel a serem utilizados na solução;
        - Identificação das dependências Apache Camel necessárias à implementação em JAVA com Spring Boot;
        - Implementação da solução;
        - Testes unitários.
- Durante o período de desenvolvimento foi possível contar com o apoio de outros membros da equipe, sobre questões relacionadas ao contexto do ambiente do cliente.

2.3 Detalhamento do Problema

Como problema principal consiste no maior controle para balanceamento de campanhas com envio de mensagens, tem-se que desenvolver uma solução de integração entre um CRM e outra aplicação. O CRM disponibiliza em um diretório de arquivo (*file system*) grande volume de dados, disponibilizados em diversos arquivos com extensão CSV. De posse destes arquivos tem-se que ler e quebrar (desmembrar) cada arquivo em “N” outros arquivos, de acordo com o número específico de linhas por arquivo. A expectativa é de que sejam disponibilizados arquivos com no mínimo 10.000 linhas cada, estes arquivos seriam lidos e quebrados em 10 arquivos de 1.000 linhas cada. Para cada novo arquivo gerado, este receberia o nome do arquivo original mais um UUID e *timestamp* gerados pela rotina de integração. Em seguida cada arquivo gerado, deverá ser encaminhado para um serviço de armazenamento em cloud (Bucket AWS S3), concomitantemente, deverá ter seu nome publicado de forma balanceada em 3 filas FIFO distintas, também disponíveis em ambiente cloud (SQS). Uma outra aplicação seria responsável por consumir o nome dos arquivos

armazenados nas filas SQS, recuperando e processando cada arquivo armazenado no S3. Entenda melhor o fluxo na **Figura 2.** Fluxo do Problema.

Figura 2. Fluxo do Problema.



Fonte: o autor

A solução poderia ter sido desenvolvida utilizando apenas o Java com Spring Boot, o qual foi inicialmente pensado pelo cliente, todavia este caminho demonstrou-se mais dispendioso e complexo, uma vez que tomaria um tempo maior de implementação, com código mais verboso. Como opção de mais rápida implementação, e por estar em aderência à demanda e premissa apresentada pelo cliente, foi optado pela adoção do Apache Camel. Este já traz vários padrões de integração e plugins que facilitam a implementação para este tipo de problema. Uma vez definida a estratégia de implementação, foi necessário entender o uso do padrão apropriado a ser implementado no Apache Camel e a forma correta de sua aplicação, como também sua adequação ao ecossistema existente.

Sem a adoção de um padrão adequado a solução tende a tomar um grau de complexidade muito maior, além de um maior volume de código, como já mencionado. Já com a adoção de padrões esta complexidade se reduz. Desta forma passou-se a

analisar quais padrões poderíamos aplicar para obter o retorno esperado, bem como a forma de implementá-los.

3 OBJETIVO

Neste trabalho tencionou-se contribuir com a comunidade científica e tecnológica na apresentação de uma proposta de integração e uso de padrões de integração de sistemas, através do Apache Camel. Para isto, define-se como objetivo geral:

"Apresentar um estudo de caso de aplicação de padrões de integração em um projeto real com utilização do Apache Camel".

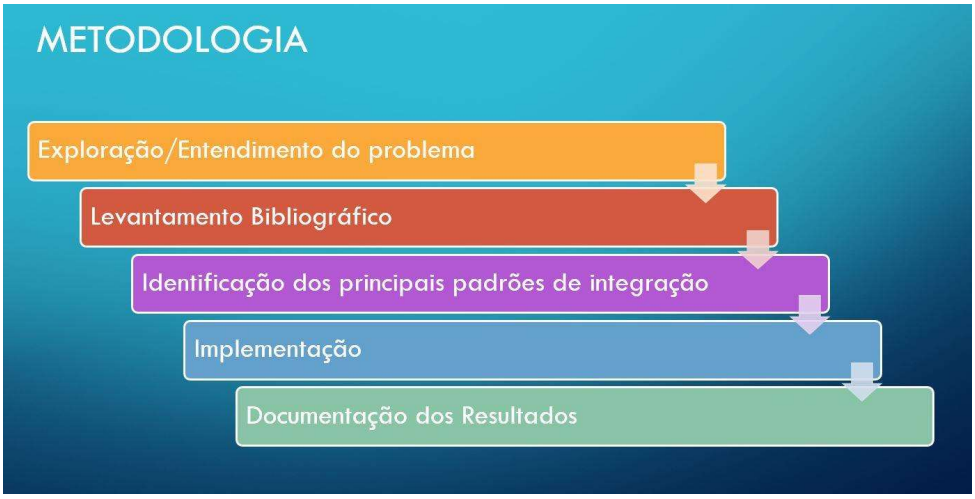
Objetivos específicos:

- Analisar o contexto do ambiente do cliente;
- Definir quais modelos de padrões melhor se aplicam ao problema;
- Avaliar e implementar a solução baseada nos modelos;
- Documentar as dificuldades e benefícios obtidos com a implementação deste estudo de caso.

4 METODOLOGIA

A **Figura 3.** Tópicos da Metodologia, abaixo, representa o esquema de estruturação da metodologia, os quais terá cada tópico detalhado.

Figura 3. Tópicos da Metodologia



Fonte: o autor

- Exploração/Entendimento do problema: realiza-se inicialmente o levantamento e entendimento do problema de integração para grandes volumes de dados, uma vez que é exposto pelo cliente o cenário específico, sendo considerada em adição todo o contexto do cliente, como as limitações de recursos, ambiente/framework disponível, bem como limitações de custos e recursos AWS disponíveis;
- Levantamento bibliográfico: segue-se com pesquisa bibliográfica para embasamento dos conceitos, importância, benefícios e desafios identificados para as empresas que utilizem integração de sistemas. Como também sobre os principais padrões de integração, *Enterprise Integration Patterns, e sobre* a utilização do Apache Camel como alternativa para integração de sistemas;
- Identificação dos principais padrões de integração: a partir das premissas definidas pelo cliente, segue-se com a identificação dos principais padrões que melhor atendem à necessidade apresentada pelo cliente, bem como a forma de aplicação;

- Implementação: implementa-se uma solução de integração, utilizando padrões de integração identificados com o uso do Apache Camel;
- Documentação dos Resultados: Ao final deve-se documentar o estudo realizado, a estratégia adotada e os padrões que foram considerados na implementação, bem como exemplos de códigos utilizados com o Apache Camel na solução proposta. Em adição também deve-se relatar os principais desafios identificados e formas de implementação realizadas.

5 CONCEITOS BÁSICOS

5.1 Integração de Sistemas

5.1.1 O que é Integração de Sistemas

A palavra integração segundo o dicionário Michaelis significa “1 Ato ou efeito de integrar (-se). 2 Condição de constituir um todo pela adição ou combinação de partes ou elementos”. (MICHAELIS, 2021)

Segundo a Red Hat, líder mundial no fornecimento de soluções empresariais open source, “a integração da TI, ou de sistemas, é a conexão de dados, aplicativos, APIs e dispositivos por toda a organização de TI.”, “A integração não só conecta tudo, mas também agrega valor com as novas funcionalidades geradas ao conectar diferentes funções do sistema” sendo primordial para tornar as empresas mais eficientes, produtivas e ágeis. E é essencial em discussões sobre a transformação digital corporativa, pois ela é responsável por fazer com que todos os elementos de TI funcionem bem em conjunto.(RED HAT, 2021a)

Quando se fala em integração, é comum que os processos das empresas passem por várias ferramentas distintas, e cada uma dessas ferramentas guarda informações importantes sobre o negócio. Para que essas informações conversem entre si, pode-se adotar duas abordagens: A integração de aplicações e/ou a integração de dados.

Na integração de aplicações, duas ferramentas são conectadas em um nível funcional, fazendo fluir as informações em tempo real, tipicamente acionadas por algum evento. Um ponto relevante a ressaltar na integração de aplicações, é que podem existir transformações dos dados, e isto deve ocorrer antes do envio destes ao seu destino final, podendo estar ou não no mesmo formato dos dados de entrada. Por outro lado, a integração de dados de diferentes sistemas consiste em reuni-los em um único repositório de dados (*Data Lake* ou *Data Warehouse*), tanto para fins analíticos quanto operacionais. (SOMBRIO, 2020)

Quando se fala em integração com grandes volumes de dados podem-se também conceituar a integração de sistemas como “a utilização de diversas

ferramentas como uma só solução, garantindo que haja uma troca de dados e informações entre as diversas plataformas utilizadas nos processos da empresa". (GAEA, 2020)

5.1.2 Interoperabilidade x Integração

Interoperabilidade pode ser entendida como a capacidade de dois ou mais sistemas se comunicarem, trabalhando em conjunto (inter-operando) de forma transparente com outros sistemas (semelhantes ou não) de modo a garantir que haja a interação para troca de informações de maneira eficaz e eficiente entre pessoas, organizações e sistemas computacionais.

Os conceitos de interoperabilidade e integração muitas vezes são utilizados como sinônimos na área de TIC - Tecnologia da Informação e Comunicação. Todavia, esses conceitos são distintos, embora complementares.

A **integração** refere-se ao processo de conectar dois ou mais sistemas gerando uma dependência tecnológica entre eles. Já a **interoperabilidade** se dá quando há o processo de comunicação de dois ou mais sistemas sem a geração de uma dependência tecnológica entre eles. (PESSOA MELLO; MESQUITA; VIEIRA, 2015).

A integração facilita o acesso à informação e, consequentemente, melhora a comunicação, cooperação e coordenação dentro da empresa, de forma que ela se comporte como um "todo" integrado (VERNADAT, 1997).

A incidência de integrações é maior quando instituições ou empresas optam por escolher sistemas diferentes por setor, e em determinado momento necessitam integrá-los para otimizar os processos ou para a geração de informações de suporte à decisão, deste modo, melhorando a eficácia de seus serviços, podendo até reduzir custos operacionais, para isto é necessário criar uma retaguarda através de uma rápida, confiável e compatível integração com os sistemas a serem integrados.

Atualmente, diante de um cenário altamente competitivo, sistemas de informação nas mais diversas áreas são pré-requisito para suportar os ambientes de negócios de empresas e governos. Provendo de forma rápida e precisa, informações e estatísticas para dar suporte a decisões de negócios, e outras decisões, como também maior agilidade nos processos. (RADES, 2017)

Para um sistema ser considerado interoperável, é muito importante que ele trabalhe com padrões abertos, disponíveis para livre acesso e implementação. Podendo ser um sistema de portal, educacional, de comércio eletrônico ou mais especificamente em sistemas bancários. Para o contexto de sistemas bancários a interoperabilidade tem se tornado um fator de alta geração de valor para as instituições financeiras, devido principalmente ao surgimento e intensificação dos bancos digitais, com a chamada “abertura bancária”.

5.1.3 A importância de ter Integração entre Sistemas

Desde o bug do milênio, foi criada uma cultura nas organizações de atualizar ou substituir seus sistemas de informação (SI) legados para adequá-los às novas necessidades que o mercado e novas tecnologias impõe, como também para a prevenção aos possíveis problemas do “*bug*”. Muitos dos desafios identificados por administradores de sistemas se devem à falta de integração dos SI ou mesmo com o uso de soluções inadequadas.

Alguns dos problemas administrativos identificados por Sordi e Marinho:

- Perda de competitividade, diante da incapacidade de redução do tempo dos processos;
- Restrição da capacidade de organização para se habilitar como fornecedor ou parceiro estratégico de redes, quando exigido domínio de modernas tecnologias de integração;
- Elevação de custos e aumento de riscos de exposição à erros decorrentes de interferência humana no processo de integração (re-digitação de dados);
- Lentidão da organização em identificar e tratar eventos de negócio, limitando capacidade de inovação e proatividade da empresa;
- Dificuldade de evolução e aprimoramento dos processos antigos, por receios e alto custo da mudança de integrações pré-existentes em sistemas legados.

Uma coisa boa é que, desde então, as empresas em sua grande maioria, compreendem que a integração entre SI é um componente crítico ao desempenho dos negócios. Que foi amplamente demonstrado por diversos pesquisadores. Desde

2000, Benamatti e Lederer que analisaram os nove principais desafios do gerenciamento das áreas de tecnologia (TI), já apontavam “novas integrações” como um desafio a ser enfrentado. (SORDI, DE; MARINHO, 2007)

Outro ponto a ser considerado é devido a uma migração da missão principal das áreas de TI, que buscam mais a integração dos sistemas legados do que entregar novos SI. Com uma frequência cada vez maior, surgem eventos nos ambientes de negócios que demandam soluções mais eficazes para integração de sistemas. Abaixo são descritos alguns dos principais eventos de ambiente de negócios que têm motivado a busca de métodos eficazes para integração dos SI. (SORDI, DE; MARINHO, 2007)

- Aumento da diversidade e quantidade de SI (diversidade técnica e/ou sistemas especialistas, compra de sistemas prontos, fusões e aquisições);
- Busca por vantagens competitivas, levando a uma melhor gestão, melhor integração, para sistemas internos A2A – *Application to Application*, ou externos B2B – *Business to Business*;
- Exigências de órgãos reguladores, como maior agilidade, processos de uso e manuseio de informações, impactando diretamente nos requisitos de integração;
- Aumento das entidades que necessitam trocar informações com as organizações. Algumas vezes sequenciais e algumas de características recíprocas.

Nos dias atuais, além dos pontos abordados, não podemos deixar de citar como um grande desafio corporativo, lidar com grandes volumes de dados, muitas vezes conhecido pelo termo “*big-data*”, que frequentemente é usado para indicar o tamanho e a variedade das fontes de dados. O fato de ter um grande volume de dados em formatos variados e não padronizados torna-se muito valioso para os negócios, mas apenas quando é possível realizar a integração deles. A Internet das Coisas (IoT) também é uma nova oportunidade para se conectar com os clientes e analisar dados úteis gerados por dispositivos de uso cotidiano. No entanto, é preciso filtrar o que necessariamente vai ser aproveitado. As aplicações web são um outro fator que aumenta ainda mais a complexidade das estratégias de integração corporativa,

principalmente quando é necessário integrar aplicações legadas a componentes com arquitetura baseada em serviços, como os micros serviços. (RED HAT, 2021b)

5.1.4 Benefícios da Integração para a Gestão Corporativa

A integração de sistemas traz uma série de benefícios associados à sua implementação, e na forma como as empresas tomam suas decisões. Segue abaixo algumas das principais vantagens da integração (GAEA, 2020):

- **Queda de retrabalho**: é uma consequência direta à integração, uma vez que se tem uma única entrada de dados, deixando estes dados disponíveis para outros setores com acesso à informação. Essa redução de retrabalho acarreta menor índice de erros de digitação e a melhor distribuição de tempo dos colaboradores;
- **Processos mais Ágeis**: uma das principais vantagens, é o ganho em agilidade nos processos produtivos da empresa. Sem esperar por respostas, os acessos passam a ser automatizados. Além disto, tem-se a automatização de tarefas, que permite uma maior dedicação dos funcionários para outras atividades, melhorando os resultados;
- **Informações mais confiáveis**: com a redução da manipulação das informações, ou seja, através de redução da interação humana, melhora a confiabilidade nos dados apresentados pela ferramenta. Em consequência facilita o trabalho do gestor, que passa a poder tomar decisões estratégicas para o futuro do negócio de forma mais simples, com acesso a dados concretos e de procedência confiável;
- **Diminuição de custos**: O custo total para manter as operações da empresa tende a reduzir, devido aos ganhos em produtividade e otimização de processos que poderão ser obtidos com a integração das informações. Como também as oportunidades de melhoria que surgirão devido ao monitoramento constante, além da redução de perdas e prejuízos. E a automatização de uma série de tarefas, que também reduzem gastos contratuais de mão de obra, entre outros;

- **Otimização de processos:** A visão macro das atividades desenvolvidas por todos os setores da empresa permite ao gestor identificar falhas ou processos que não estejam entregando o que se espera. Permitindo a busca de alternativas para melhorias, para alcançar as expectativas com relação aos seus resultados. Num sistema de gestão integrada, esta é uma das maiores vantagens que podemos elencar, pois permite um aumento de produtividade e eficiência em todos os setores da empresa;
- **Redução de erros:** grandes ecossistemas de softwares independentes tem uma grande desvantagem em relação à gestão integrada: a ocorrência sucessiva de erros. Isto ocorre muitas vezes por causa da inserção de dados de forma manual em cada um dos sistemas. A integração promove a redução desses erros de entrada de informações, já que os dados serão inseridos uma única vez no sistema, ficando disponíveis para todos os outros setores;
- **Redução do uso de softwares:** Com a integração, pode-se reduzir também a utilização e gestão de diferentes softwares, que podem trazer problemas diversos: — entre eles, um dos mais críticos é a necessidade de treinamento para a equipe. Nesse sentido também é reduzida a realização de capacitação;
- **Melhoria no controle dos processos**: Uma vez informatizados os processos, seu registro e controle e fluxo se torna muito mais fácil de ser monitorado de forma consistente, possibilitando uma melhor visão e detecção de processos fora da curva;
- **Uso de *dashboards*:** administração centralizada, através de gestão integrada com uso de *dashboards* que podem ser montados conforme a necessidade da empresa, dando ênfase às informações que a administração acredita serem de maior relevância. Os painéis de controle permitem uma visualização departamental com andamentos de atividades e processos realizados por estes, auxiliando o monitoramento e observações de possíveis gargalos que precisam de correção.

5.1.5 Desafios em Integrar Sistemas com Grandes Volumes de Dados

São diversos desafios que devem ser superados na realização de uma integração de sistemas de sucesso. Segue abaixo alguns desafios que devem ter uma maior atenção durante a implementação de solução integrada (GAEA, 2020):

- **Monitoramento**: Para um monitoramento consistente, é necessário o constante monitoramento de todos os processos e atividades durante uma integração, permitindo a visualização de falhas que antes não eram vistas;
- **Atualizações:** A evolução constante da tecnologia, acarreta num maior e constante número de novas atualizações. Manter seus sistemas atualizados também protege sua empresa e suas informações de ataques de hackers e cibercriminosos que se utilizam de sistemas legados para realizar invasões e roubos de dados;
- **Manutenções periódicas:** As atualizações realizadas em sistemas integrados, podem acabar por gerar uma certa lentidão geral no ecossistema, devido a alterações na maneira como tal software se relaciona com o banco de dados comum. Por isso, a necessidade de realizar manutenções periódicas em cada um dos módulos da plataforma, com o intuito de evitar grandes manutenções. Ao realizar constantes revisões, é possível garantir a continuidade das operações e evitar possíveis interrupções dos serviços, que geram prejuízos;
- **Informações repetidas:** Sistemas legados antes de uma integração, tendem a possuir acessos a informações em comum, podendo haver dados repetidos. É necessário criar um plano de ação com o objetivo de verificar todos os dados contidos em cada um dos *databases* das aplicações em busca de informações repetidas a fim de eliminá-las, evitando possíveis duplicidades ao fim da integração dos sistemas;
- **Backup confiável:** Ao integrar todos os sistemas da empresa, compartilhando informações constantemente, será utilizado apenas um banco de dados comum, aumentando o risco quanto a perda de dados por falha de servidor ou parada dos serviços por motivo fortuito. Sendo preciso investir ainda mais em rotinas confiáveis de backup e nas boas práticas,

estando atento à criação de diversas cópias de segurança, locais de armazenamento, tempo de *restore* e demais rotinas do *backup*;

- **Banco de dados legado:** Um dos grandes desafios é quando o banco de dados legado possui características que pode não ser aceito por algum dos softwares utilizados. Existem duas opções nesse caso. A primeira seria iniciar um novo banco de dados em uma plataforma mais moderna e aceita por todos os sistemas em utilização na empresa, o que nem sempre é viável. A segunda opção seria a importação de todos os dados para um novo banco de dados, mais moderno e aceito por todas as soluções em uso na empresa, contudo isto também pode gastar muitos recursos na transação. Devido à diversidade em como cada banco de dados trata as informações, exigindo muitas vezes alterações em conjuntos de dados antes de uma importação.

5.1.6 Uso de Padrões nas Integrações

Um problema recorrente na integração de sistemas, era o fato destes não serem projetados para se integrarem entre si. Como também as primeiras tentativas de integração não seguirem padrões ou normas técnicas, dada a sua inexistência. Com a crescente necessidade do uso de técnicas e metodologias nesta área e, ainda, a própria evolução tecnológica, impulsionaram a criação das primeiras especificações, que têm sido aperfeiçoadas.(CUNHA; JUNIOR; DORNELAS, 2014)

Como o foco em questão é a integração entre SI, apresenta-se abaixo, um resumo dos principais recursos utilizados para integração entre SI, classificados a partir de três principais mecanismos: chamadas (*call interface*), mensagens (*messaging*) e transferência entre arquivos (*data access / file transfer*) (SORDI, DE; MARINHO, 2007).

Figura 4. Tecnologias que habilitam a integração entre SI

MENSAGENS (messaging) - aplicações são integradas pelo envio e recebimento de mensagens, utilizando tecnologias que empregam mecanismos de fila de mensagens (message queue). Exemplos de tecnologias que operam messaging: sistemas de e-mail e produtos para workgroup, como Microsoft Outlook e Lotus Notes; produtos específicos para integração entre sistemas via mensagem, como IBM MQ Series, Tibco, Vitria e Microsoft MQMS.
TRANSFERÊNCIA ENTRE ARQUIVOS OU COMPARTILHAMENTO DE DADOS (data acess / file transfer) - neste mecanismo, as aplicações são integradas via acesso direto as suas bases de dados ou via transferência de arquivos. Exemplos de técnicas para data acess ou file transfer: extração da base de dados fonte, transferência de arquivos e carga de dados em batch; leitura e gravação direta na base de dados, utilizando chamadas à base de dados ou gateways (ODBC ou EDA/SQL); replicação de base de dados.
CHAMADAS (call interface) - aplicações provêem interface possível de ser chamada, denominada API (application programmable interface). Exemplos de tecnologias que operam call interfaces: interfaces de processamento transacional como CICS da IBM e o Tuxedo da BEA; interfaces para aplicações do tipo "pacotes", como é o caso da BAPI para o SAP R/3; interfaces baseadas em objetos CORBA ou COM ou JavaBeans.

Fonte: (SORDI, DE; MARINHO, 2007)

Não há uma técnica de integração melhor ou pior para uma organização, cada uma delas possui vantagens e desvantagens. Devendo-se selecionar a técnica mais apropriada para cada nova necessidade de integração. As aplicações podem integrar-se, empregando várias técnicas de integração, de forma a obter vantagens daquela que melhor atenda a cada nova demanda. Assim, a camada de integração entre SI pode ser entendida como um ambiente híbrido composto por diferentes técnicas de integração (HOHPE; WOOLF, 2002).

As ações de integração podem ocorrer em diversos locais ou pontos do ambiente computacional, devido à diversidade de recursos tecnológicos. Gerando uma complexidade que requer pessoas, ferramentas e procedimentos específicos, para composição de um ambiente propício à gestão das integrações. (SORDI, DE; MARINHO, 2007)

5.2 EIP – Enterprise Integration Patterns

Como a questão da integração não é algo novo, diversas técnicas foram desenvolvidas nas últimas décadas. Serão apresentadas aqui algumas das mais comumente utilizadas, destacando seus pontos fortes e fracos. (HOHPE; WOOLF, 2002)

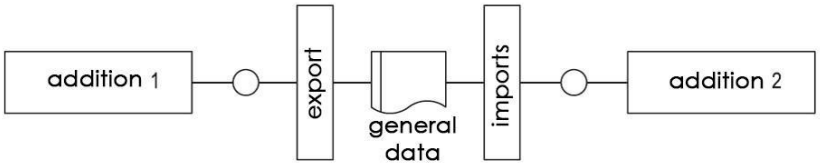
5.2.1 Estratégias de Integração

5.2.1.1 Batch Data Exchange – Troca de dados em lote

A maior parte dos esforços iniciais para múltiplas integrações entre vários sistemas de computadores foram baseadas na transferência de dados em arquivos. Em Sistemas Mainframe, era comum a realização de ciclos de operações noturnas para execução de arquivos em lote. Onde eram processadas as transferências de dados.

Tendo como vantagens o desacoplamento físico entre os diferentes processos, se o sistema de destino não está disponível de imediato para receber o arquivo. E também, pelo fato de o arquivo ser uma forma de linguagem independente, que pode utilizar um conjunto de caracteres comuns. Como desafios: o fato de que dados alterados em um sistema podem não estar atualizados no outro sistema até o próximo dia. O que pode confundir os usuários e causar problemas de integridade entre sistemas. Adicionalmente, transferências em *batch* podem gerar transmissões desnecessárias de dados, quando processam mais informações do que as que foram efetivamente alteradas. (HOHPE; WOOLF, 2002)

Figura 5. Troca de dados em lote.

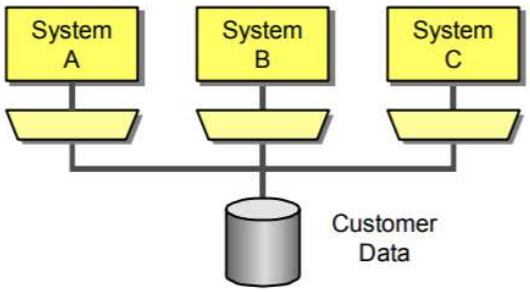


Fonte: (HOHPE; WOOLF, 2002)

5.2.1.2 Shared Database – Compartilhamento de base de dados

Em uma tentativa de eliminar problemas de sincronização de dados e a replicação de grandes quantidades de dados, algumas empresas criaram recursos de base de dados compartilhados. Permitindo ter todos os dados em um único lugar sem a necessidade de duplicação. Problemas de sincronização e bloqueios eram facilmente provocados pelos próprios sistemas de banco de dados.

Figura 6. Compartilhamento de base de dados.



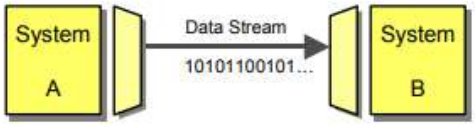
Fonte: (HOHPE; WOOLF, 2002)

A principal desvantagem dessa solução está na dificuldade de definir um modelo de dados que se adapte a todos os aplicativos. Muitos aplicativos são construídos ou adquiridos com modelos de dados proprietários e muitas tentativas de transformar todos esses modelos em um único modelo é uma tarefa interminável. Como também o acesso a uma única base de dados pode causar sérios problemas de performance. (HOHPE; WOOLF, 2002)

5.2.1.3 Raw Data Exchange – Troca de dados brutos

Troca direta de dados por meio de protocolos de transferência de dados de rede, como TCP/IP, esses mecanismos permitem a troca direta de dados entre dois sistemas em tempo real. As informações podem ser propagadas à medida que os dados são modificados no sistema de origem. A replicação em tempo real reduz a chance de os dados ficarem fora de sincronia.

Figura 7. Troca de dados brutos.



Fonte: (HOHPE; WOOLF, 2002)

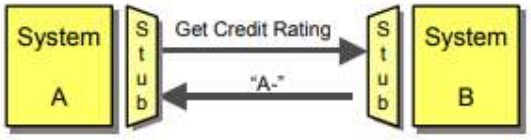
Esta forma direta de troca de dados também trouxe novos desafios. Como ocorre de forma síncrona, caso haja algum problema em algum dos sistemas a mensagem não será recebida/entregue. Isto exige que os desenvolvedores criem sofisticados mecanismos de controle de entrega e novas tentativas de envio.

Como também, por serem estruturados *byte-a-byte*, os dados enviados também devem ser representados de forma simples, ficando a cargo dos desenvolvedores criar código para converter todos os dados de origem em fluxos de caracteres e convertê-los de volta em estruturas de dados na extremidade receptora. Sem uma infraestrutura de gerenciamento, esta abordagem é propensa a erros e difícil de manter. (HOHPE; WOOLF, 2002)

5.2.1.4 Remote Procedure Calls – Chamada a procedimento remoto

Os mecanismos de chamada de procedimento remoto isolam a aplicação dos mecanismos de troca de dados brutos por meio de uma camada adicional. Essa camada gerencia o *marshalling* de tipos de dados complexos. Como resultado, o RPC permite que um aplicativo invoque de forma transparente uma função implementada em outro aplicativo. Os mecanismos de empacotamento dependem de *stubs* gerados a partir da especificação da interface em uma linguagem neural IDL – *Interface Definition Language.*

Figura 8. Chamada a procedimento remoto.



Fonte: (HOHPE; WOOLF, 2002)

Embora os sistemas baseados em RPC tornem o tratamento da integração muito mais simples, esses sistemas ainda compartilham as desvantagens da comunicação não confiável e síncrona com os métodos de troca de dados brutos. Os sistemas baseados em RPC também implicam conexões ponto a ponto frágeis entre remetentes e recebedores, que se tornam difíceis de manter à medida que o número

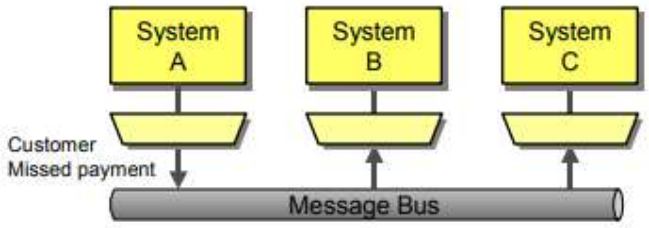
de participantes aumenta. Além disso, a maioria das implementações RPC fornecidas pelos fornecedores não são interoperáveis com outros produtos. (HOHPE; WOOLF, 2002)

5.2.1.5 Messaging - Mensagem

Os sistemas de mensagens tentam superar as desvantagens das soluções anteriores, fornecendo transferência de dados assíncrona e confiável. Um aplicativo pode publicar dados na camada de integração e pode ter certeza de que os dados serão entregues ao(s) destinatário(s).

O sistema de origem não precisa esperar por uma confirmação e pode continuar o fluxo do processo primário. Os sistemas de mensagens também incorporam esquemas de endereçamento que evitam a difícil manutenção de conexões ponto a ponto, típicas dos sistemas RPC.

Figura 9. Mensagem.



Fonte: (HOHPE; WOOLF, 2002)

No momento, parece que os sistemas de mensagens fornecem a melhor base para uma solução de integração. Todavia, existem alguns desafios a serem superados. Os sistemas de mensagens ainda são relativamente novos e o corpo de conhecimento em torno da arquitetura desses sistemas ainda está em evolução. Os sistemas assíncronos exigem uma abordagem diferente para a arquitetura e design de sistemas do que os sistemas síncronos. Além disso, a natureza assíncrona da interação pode tornar o teste e a depuração mais difíceis também. (HOHPE; WOOLF, 2002)

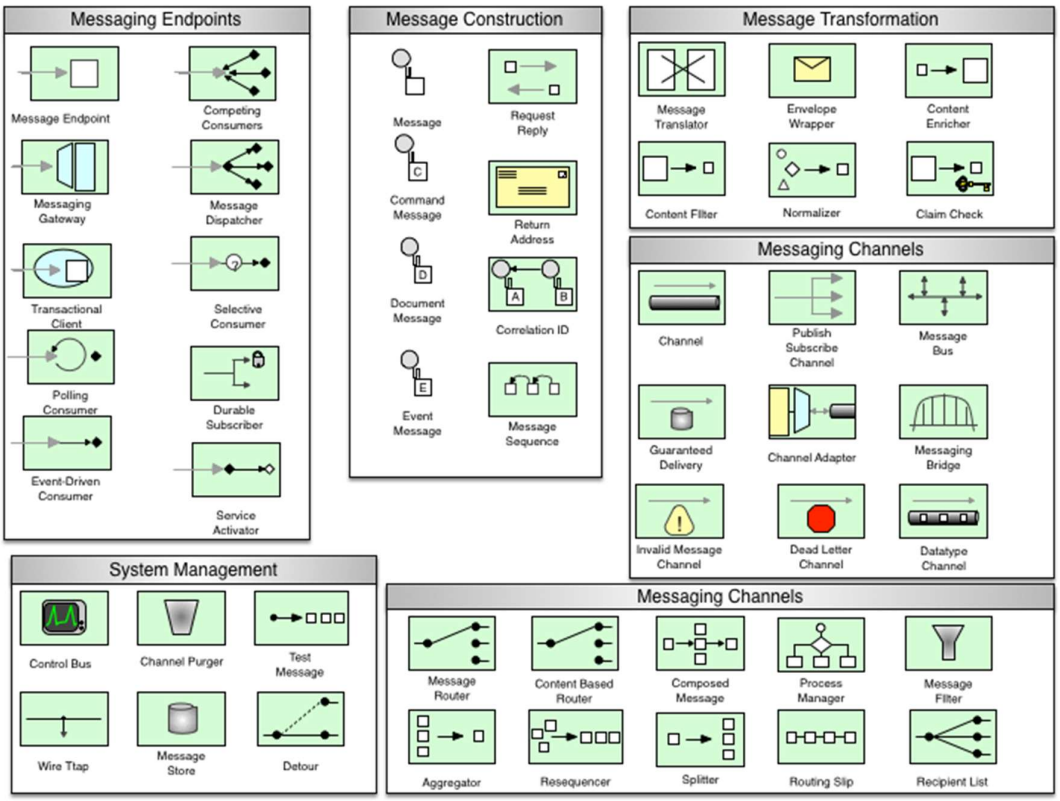
Existem muitas implementações diferentes de sistemas de integração ou *middleware*. Mesmo no âmbito das mensagens, existem muitas abordagens e tecnologias diferentes. Por exemplo, muitos fornecedores de EAI diferentes fornecem soluções de "mensagem", assim como as implementações JMS ou soluções baseadas em protocolos de serviços da web, como SOAP. Para permanecer independente de implementações técnicas específicas, definimos *middleware* orientado a mensagens (MOM) com base nas seguintes qualidades de definição (HOHPE; WOOLF, 2002):

- Comunicação entre sistemas diferentes (***Dispares Systems***): *Message-Oriented Middleware* precisa habilitar a comunicação entre sistemas que são escritos em linguagens de programação distintas, executam em diferentes plataformas de computação em diferentes localizações;
- Transporte pacotes discretos de informações (***messages***): *Message-Oriented Middleware,* a informação é enviada em uma mensagem discreta, diferente de fluxos de dados, cada mensagem é independente e pode ser roteada e processada independentemente;
- Uso de endereçamento (***addressable conduits***): As mensagens são encaminhadas por canais, e podem ser localizadas por esquema de endereçamento. O esquema de endereçamento pode ser baseado em uma série de princípios, sendo os esquemas simples ou hierárquicos os mais comuns. Implementações físicas normalmente empregam uma forma de Identificadores de Recursos (URIs) universais, como nomes de filas, nomes de canais ou assuntos de mensagens como endereços;
- Comunicação síncrona ou assíncrona (***Synchronous or Asynchronous communication***): Os canais podem enfileirar as mensagens se um sistema estiver temporariamente indisponível (quando assíncrono) ou não (quando síncrono). Os canais podem também fornecer distribuição *multicast* (vários destinatários para a mesma mensagem) ou distribuição de ponto único (um único destinatário por mensagem).

5.2.2 Padrões de Categorias

Integração empresarial abrange muitos problemas de domínio e níveis de abstração, fazendo sentido dividi-los em categorias que refletem o escopo e a abstração dos padrões. (HOHPE; WOOLF, 2002)

Figura 10. Padrões de Integração



Fonte: (FRANZINI, 2017)

5.2.2.1 Message Routing

O padrão *Message Router* apresenta mecanismos para direcionar mensagens de um remetente para o destinatário correto. Integração baseada em mensagens usam o endereçamento do canal para enviar as mensagens entre os sistemas. O remetente e o destinatário concordam em um esquema de endereçamento comum para que uma mensagem enviada para um determinado endereço seja recebida pelo destinatário correto (ou conjunto de destinatários). Em muitos casos, o destinatário da mensagem é desconhecido para o remetente ou é derivado dinamicamente com base

em um conjunto de condições. Além disso, as mensagens podem ser divididas, encaminhadas separadamente e reunidas posteriormente. Geralmente, deixando o conteúdo da mensagem inalterado, mas movendo a mensagem de um endereço para outro. (HOHPE; WOOLF, 2002)

5.2.2.2 Message Transformation

Os padrões de transformação da mensagem alteram o conteúdo das informações de uma mensagem. Em muitos casos, um formato de mensagem precisa ser alterado devido às diferentes necessidades do sistema de envio ou recebimento. Os dados podem ter que ser adicionados, retirados ou os dados existentes podem ter que ser reorganizados. Essas tarefas são realizadas por transformadores de mensagens. (HOHPE; WOOLF, 2002)

5.2.2.3 Message Management

Os Padrões de Gerenciamento de Mensagens fornecem as ferramentas para manter um sistema complexo baseado em mensagens em execução. Uma solução de integração baseada em mensagens pode processar milhares ou até milhões de mensagens em um dia. As mensagens são geradas, roteadas, transformadas e consumidas. A solução tem que lidar com condições de erro, gargalos de desempenho e mudanças nos sistemas participantes. Os padrões de gerenciamento de mensagens atendem a esses requisitos. (HOHPE; WOOLF, 2002)

5.2.2.4 Padrão de Descrição e Notação

As descrições dos padrões devem seguir um formato consistente. Vários formatos de padrão foram estabelecidos na comunidade de padrões, por exemplo [PatternForms].

Christopher Alexander define padrões no mais alto nível usando uma estrutura de solução de problema de contexto, aumentada por imagens, diagramas e ponteiros para padrões relacionados.

Os padrões de integração que apresentados neste estudo seguem a forma Alexandrina de apresentação de padrões. Cada padrão consiste no nome do padrão, uma breve descrição do contexto, a declaração do problema, uma elaboração das forças, a solução proposta seguida por um diagrama, além de uma discussão de padrões relacionados e um exemplo opcional (HOHPE; WOOLF, 2002).

5.2.3 Padrões de Integração no Apache Camel

Quando é realizada integração adotando-se padrões de projeto, intrinsecamente tem-se o reuso de experiência adquirida em problemas corriqueiros, refletindo diretamente no aumento da qualidade do código, tornando-o mais flexível, elegante e de fácil reuso. Não é necessário construir algo novo quando já existe uma solução estruturada para determinados problemas. E nada melhor do que os padrões apresentados em "*Enterprise Integration Patterns*, Gregor Hohpe and Bobby Woolf" que permitem implementar problemas complexos de integração corporativos.

No contexto deste estudo de caso será abordada a adoção do Apache Camel que engloba o uso de *Enterprise Integration Patterns*, nas integrações de projetos corporativos.

5.3 Apache Camel

5.3.1 Sobre o Apache Camel

O Apache Camel é um *framework* de integração *open source* maduro que surgiu em 2007, que realiza a integração entre sistemas de aplicações de forma fácil e simplificada. O foco do Camel é simplificar a integração.

O Camel difere de um ESB por não ter um *container*, embora alguns chamem desta forma por ele apoiar o roteamento, transformação, orquestração, monitoramento e assim por diante. É preferível se referir ao Camel como estrutura de integração.

O Camel introduz algumas novas ideias na área de integração, razão pela qual ele foi criado. Este relato não pretende se aprofundar em todos, mas apenas naqueles

que são necessários ao entendimento do desenvolvimento da solução, abaixo algumas das características do Camel (IBSEN, Claus; ANSTEY, 2018):

- Mecanismo de roteamento e mediação;
- Extensa biblioteca de componentes;
- Padrões de integração corporativa (EIPs);
- Idioma específico do domínio (DSL);
- Roteador agnóstico de carga;
- Arquitetura modular e plugável;
- *Plain Old Java Object* (POJO) model;
- Configuração fácil;
- Conversores automáticos de tipo;
- *Lightweight core* ideal para microsserviços;
- Nuvem pronta;
- Kit de teste.

5.3.1.1 Arquitetura Camel

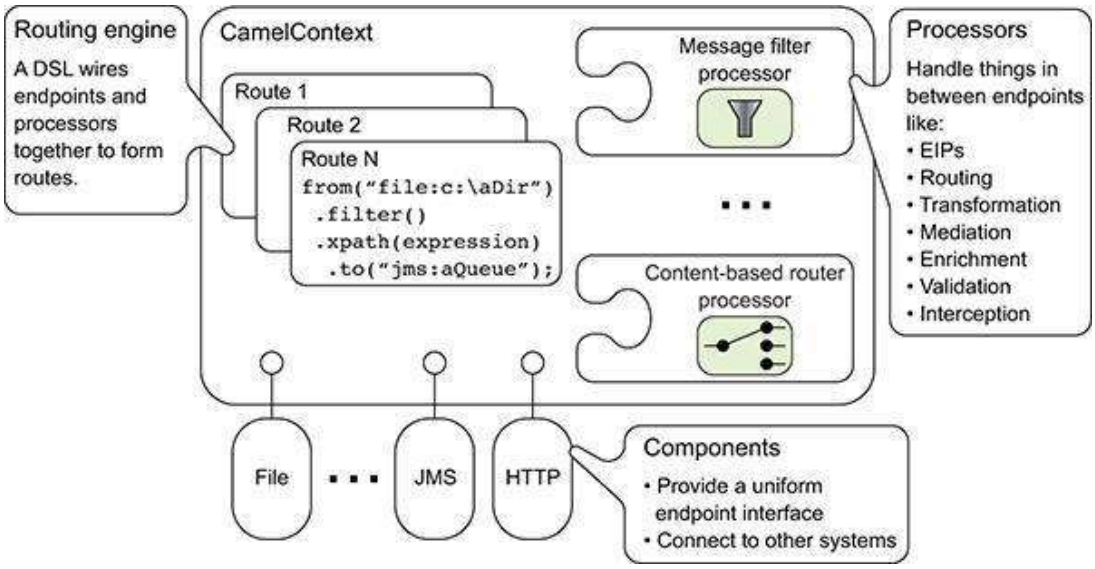
Apresenta-se aqui uma visão de alto-nível (CamelContext) da arquitetura do Camel e seus componentes e os principais conceitos. O Apache Camel já traz recursos, funcionalidades plugáveis que poupam tempo de implementação. Destacando-se:

- Roteamento: possui um engine de roteamento de mensagens, processadores de mensagens permitindo aplicar roteamento baseado em conteúdo;
- Pré-Processamento e Transformação: possui diversos conversores de tipo, entre outros recursos;
- Protocolos: facilitam a comunicação com outras aplicações por já trazer protocolos de comunicação como File, JMS, HTTP.

De forma geral, ele apoia o roteamento, transformação, orquestração e monitoramento do processo. É considerado um framework leve, que permite a

integração praticamente qualquer tipo de aplicação, com uso de uma grande diversidade de plugins, e também por estar pronto para uso em aplicações em nuvem (IBSEN, ClauS; ANSTEY, 2018).

Figura 11. Composição do Camel em alto nível (CamelContext)



Fonte:(IBSEN, Claus; ANSTEY, 2018)

O mecanismo de roteamento usa rotas para indicar onde as mensagens são enviadas. As rotas são definidas usando um dos DSLs do Camel. *Processors* são usados para transformar e manipular mensagens durante o roteamento, como também implementa todos os EIPs, que correspondem aos nomes no DSLs. Os componentes são pontos de extensão para adicionar conectividade a outros sistemas. Para expor estes sistemas para o restante do Camel, os componentes fornecem uma interface de *endpoin*t.

5.3.1.2 Mecanismo de Roteamento e Mediação

O ponto principal do framework Camel é o mecanismo de criação de rotas, um motor de roteamento. Ele permite a definição de suas próprias regras de rotas, dando uma oportunidade de integrar quaisquer tipos de sistemas, sem a necessidade de converter seus dados em um formato canônico, bem como determinar e processar como enviar as mensagens. Através de uma linguagem de integração que permite

definir regras complexas de roteamento como um processo de negócio. Conforme imagem.

Figura 12. Ligação do Camel entre sistemas.

Fonte: (IBSEN, Claus; ANSTEY, 2018)

5.3.1.3 Domain-Specific Language

No início, a linguagem de domínio específico (DSL) do Camel foi uma contribuição importante para o espaço de integração. Desde então, várias outras estruturas de integração seguiram o exemplo e agora apresentam DSLs em Java, XML ou linguagens personalizadas. O objetivo da DSL é permitir que o desenvolvedor se concentre no problema de integração em vez de na ferramenta - a linguagem de programação. Aqui estão alguns exemplos de DSL usando formatos diferentes e mantendo-se funcionalmente equivalente (IBSEN, Claus; ANSTEY, 2018):

- Java DSL

```
from("file:data/inbox").to("jms:queue:order");
```

- XML DSL

```
<route>
  <from uri="file:data/inbox"/>
  <to uri="jms:queue:order"/>
</route>
```

5.3.1.4 Arquitetura Modular and Pluggable

O Camel tem uma arquitetura modular, que permite que qualquer componente seja carregado no Camel, independentemente de o componente ser enviado com o Camel, de terceiros ou de sua própria criação. Também é possível configurar quase

tudo no Camel. Muitos de seus recursos são plugáveis e configuráveis - qualquer coisa, desde geração de ID, gerenciamento de thread, sequenciador de desligamento, cache de fluxo e outros enfeites. Os conceitos foram baseados a partir do trabalho de Ibsen e Anstey (IBSEN, Claus; ANSTEY, 2018).

5.3.1.4.1 Modelo POJO

Os Java beans (ou *Plain Old Java Objects*, POJOs) são muito bem considerados no Camel, tanto que são incentivados pelo Camel que permite o uso de *beans* em qualquer lugar e a qualquer hora em seus projetos de integração. Em muitos lugares, pode-se estender a funcionalidade integrada do Camel com código personalizado/customizado.

5.3.1.4.2 Conversão de Tipo Automática

O Camel possui um mecanismo de conversor de tipo embutido que vem com mais de 350 conversores. Não sendo necessário configurar regras de conversão de tipo, por exemplo para ir de matrizes de *bytes* para *strings*. E caso precise converter para tipos que o Camel não suporte, pode-se criar conversor de tipo customizado.

5.3.1.4.3 Lightweight Core Ideal for Microservices

O Camel é considerado leve, sua biblioteca total chega a cerca de 4,9 MB e tendo apenas 1,3 MB de dependências em tempo de execução. Isso torna o Camel fácil de incorporar ou implantar em qualquer lugar que você quiser, como em um aplicativo independente, micro-serviço, aplicativo da web, aplicativo Spring, aplicativo Java EE, OSGi, Spring Boot, WildFly e em plataformas de nuvem, como AWS, Kubernetes e Cloud. O Camel foi projetado não para ser um servidor ou ESB, mas para ser integrado em qualquer tempo de execução que se escolha, só precisando do Java.

5.3.1.4.4 Nuvem Pronta

Em adição o Camel é *cloud-native*, fornecendo vários componentes para conexão com provedores SaaS. Neste relato será feito o uso do conector Amazon S3.

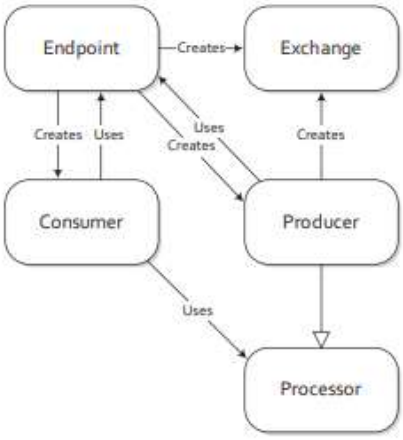
5.3.1.4.5 Test Kit

O Camel oferece um kit de teste que torna mais fácil testar seus próprios aplicativos Camel. O mesmo kit de teste é usado extensivamente para testar o próprio Camel e inclui mais de 18.000 testes de unidade. O kit de teste contém componentes específicos de teste que podem ajudá-lo a simular *endpoints* reais. Ele também permite a definição de expectativas que o Camel pode usar para determinar se um aplicativo atendeu aos requisitos ou falhou.

5.3.1.4.6 Processor

O *processor* faz parte do conceito central do Camel que representa um nó capaz de usar, criar ou modificar trocas de informações. Durante o roteamento, as trocas fluem de um processador para outro. Os processadores podem ser implementações de EIPs, produtores de componentes específicos ou sua própria criação personalizada. A **Figura 13.** Como *end-point* trabalha com *Producers, Consumers* e *Exchange*, mostra o fluxo entre os processadores.

Figura 13. Como *end-point* trabalha com *Producers, consumers* e *Exchange*



Fonte: (IBSEN, Claus; ANSTEY, 2018)

5.3.1.4.7 Component

Os componentes são o principal ponto de extensão no Camel. Até o momento, o ecossistema Camel tem mais de 280 componentes que variam em função de transporte de dados a DSLs, formatos de dados e assim por diante. Pode-se até criar seus próprios componentes para o Camel.

Do ponto de vista da programação, os componentes são bastante simples: são associados a um nome que é usado em um URI e atuam como uma fábrica de terminais. Por exemplo, File Component é referido por "**file**" em um URI e cria File Endpoints. O EndPoint é talvez um conceito ainda mais fundamental no Camel.

5.3.1.5 Enterprise Integration Patterns

Os padrões de integração corporativa, ou EIPs, são úteis não apenas porque fornecem uma solução comprovada para um determinado problema, mas também porque ajudam a definir e comunicar o próprio problema. Os padrões têm semântica conhecida, o que torna a comunicação de problemas muito mais fácil. O Camel é fortemente baseado em EIPs. Embora os EIPs descrevam problemas e soluções de integração e forneçam um vocabulário comum, o vocabulário não é formalizado. O

Camel tenta preencher essa lacuna fornecendo uma linguagem para descrever as soluções de integração. Existe uma relação quase **um-para-um** entre os padrões descritos em Enterprise *Integration Patterns* e o Camel DSL (IBSEN, Claus; ANSTEY, 2018) (**Figura 10.** Padrões de Integração).

O Apache Camel implementa a maioria dos padrões empresariais EIPs do livro de Gregor Hohpe e Bobby Woolf (HOHPE; WOOLF, 2002), que são essenciais para a construção dos blocos de rotas Camel. Os padrões de integração podem ser agrupados conforme a classificação: (IBSEN, Claus; ANSTEY, 2018)

- **Canais de Mensagens (*Messaging Channels*):** oferecem um meio para que aplicações possam trocar dados através de um canal conhecido, diminuindo bastante o acoplamento entre os envolvidos;
- **Construção de Mensagem (*Message Construction* ou *Messaging Patterns*):** oferecem soluções que descrevem as várias formas, funções e atividades que envolvem a criação e transformação da mensagem a ser trafegada entre os sistemas;
- **Roteamento de Mensagens (*Messaging Routing*):** oferecem mecanismos para direcionar mensagens de um remetente para um destinatário através de canais de comunicação como, por exemplo, filas e tópicos. As mensagens roteadas podem ser divididas, redirecionadas separadamente e reagrupadas posteriormente. Todavia, vale ressaltar que não alteram o conteúdo da mensagem, apenas à move de um canal para outro;
- **Transformação de Mensagem (*Message Transformation*):** visam mudar o conteúdo de uma mensagem para diferentes necessidades de envio e recebimento. Dependendo da situação, há alteração dos dados, com retiradas ou adições de informações para que as mensagens sejam reconstruídas;
- **Terminais de Mensagens (*Messaging Endpoints*):** esses padrões oferecem uma interface entre a aplicação e a API do sistema. É então responsabilidade do CamelContext criar e ativar as instâncias de *Endpoint* necessárias usando as implementações de Component disponíveis;
- **Gerenciamento de Sistema (*System Management*):** esse padrão descreve como monitorar, testar e administrar as mensagens do sistema complexo,

baseado em mensagens em execução, incluindo lidar com condições de erro, gargalos de desempenho e alterações nos sistemas participantes.

Neste relato serão apresentados alguns dos padrões mais ricos e poderosos, entre estes alguns que foram utilizados na solução implementada (IBSEN, Claus; ANSTEY, 2018).

- *Message channel;*
- *Aggregator;*
- *Splitter;*
- *Dynamic Router;*
- *Load Balancer;*
- *Routing Slip;*
- *Composed Message Processor.*

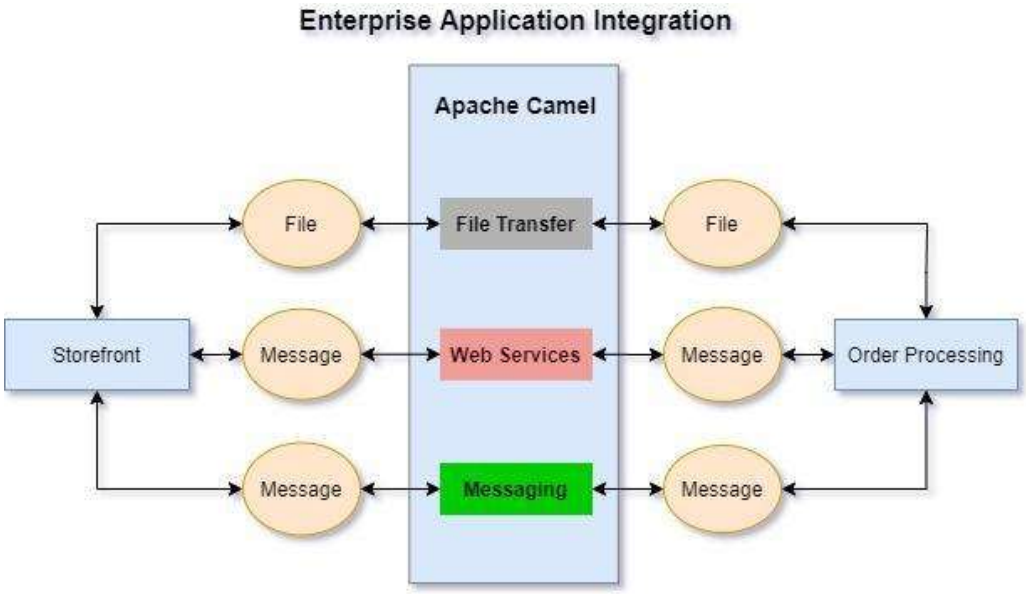
5.3.1.5.1 Conceitos Principais

Primeiramente, vejamos alguns conceitos importantes no Camel:

- *Mediador engine*: realiza a mediação, ou processamento, integrando a comunicação entre *endpoints;*
- *Routing engine*: responsável pela construção das rotas entre *endpoints;*
- *Endpoints*: ponto de extremidade, é um dispositivo final conectado a um terminal de rede, o camel suporta uma variedade de *endpoints* (nuvem, rest, db, etc).

5.3.1.5.2 Abordagens de Integração

Figura 14. Integração de aplicativos empresariais



Fonte: (HOFFMAN, 2020)

Segundo Michael Hoffman podemos agrupar as abordagens de integrações em:

- *File Transfer*: Integração através da troca de arquivos através de FTP;
- *Web Services*: Integração pela web, expondo um método que executa a lógica de negócios;
- *Messaging*: Integração através da troca de mensagens, muitas vezes assíncrona. Fornece maior flexibilidade, suportando um nível mais alto da entrega garantida e interoperabilidade;

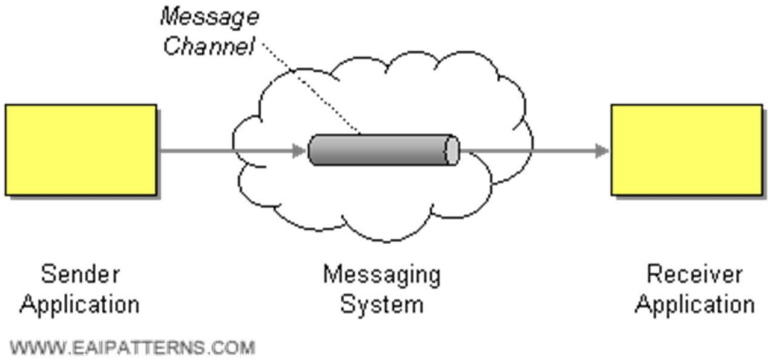
5.3.1.5.3 Exposição do EIP: Message Channel

O EIP *Patterns Message Channel* é suportado pelo Camel.

Oferece um meio para que aplicações possam trocar dados através de um canal conhecido, diminuindo bastante o acoplamento entre os envolvidos, todas as interações com o canal de mensagem são realizadas pelas interfaces dos *endpoints*.

Neste modelo, há garantia do envio e entrega à aplicação correta em virtude da utilização de um canal em comum, mesmo que a aplicação desconheça os destinatários (HOHPE; WOOLF, 2002).

Figura 15. Canal de mensagem.



Fonte: (HOHPE; WOOLF, 2002)

E o seguinte exemplo abaixo mostra um pequeno trecho de uma rota Java:

```
from("direct:foo")
    .to("jms:queue:foo")
```

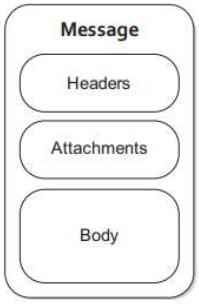
Modelo da Mensagem no Camel

O Camel suporta o padrão de integração através do modelo de mensagem. O modelo é composto por duas interfaces (KOLB, 2008).

Message interface representa a entidade que contém os dados que são roteados. A estrutura que se deve esperar da implementação.

- *Header* (cabeçalho): composto com um Java hashmap com chaves únicas (case-insensitive) e objetos java de qualquer tipo, valor dos metadados – origem, destino
- *Body* (corpo): tipo de objeto java, que armazena o conteúdo da mensagem
- *Fault*: definindo um *fault* na mensagem, indicando uma falha de resultado. Ela é parte do contrato entre o cliente e o servidor sendo tratadas no nível de aplicativo.

Figura 16. Arquitetura da mensagem.

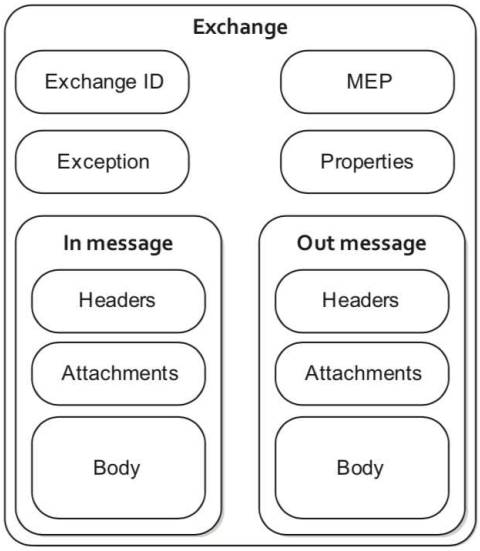


Fonte: (KOLB, 2008)

Exchange Interface representa o container da mensagem para rotear a entrada e saída das mensagens que são transferidas entre o produtor e o consumidor.

- Container com as informações de entrada da mensagem e retorno do processamento da mensagem na rota. Com metadados armazenados como propriedades do Exchange, similar ao header da mensagem, porém funcionam como um metadado de nível global dentro de toda a rota. Algumas das propriedades são o *unique* ID chamado de Exchange ID, *Exception* entre outras;
- Message Exchange Pattern (MEP): Identifica o estilo da mensagem que está sendo usada.

Figura 17. Arquitetura da Exchange

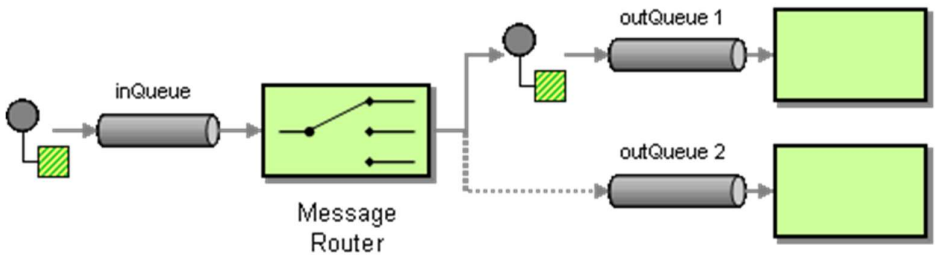


Fonte: (KOLB, 2008)

5.3.1.5.4 Exposição do EIP: Message Router

O EIP *Patterns Message Router* permite o consumo de uma mensagem de entrada, avalie algum predicado e depois escolha o destino de saída correto (HOHPE; WOOLF, 2002).

Figura 18. Roteador de mensagens – Message router.



Fonte: (HOHPE; WOOLF, 2002)

O exemplo a seguir mostra como rotear uma solicitação de uma fila de entrada: um *endpoint* para qualquer **fila: b**, **queue: c** ou **queue: d** dependendo da avaliação de várias expressões de predicado.

```
RouteBuilder builder = new RouteBuilder() {
  public void configure() {
    from("direct:a")
      .choice()
        .when(simple("${header.foo} == 'bar'"))
          .to("direct:b")
        .when(simple("${header.foo} == 'cheese'"))
          .to("direct:c")
        .otherwise()
          .to("direct:d");
  }
```

```
};
```

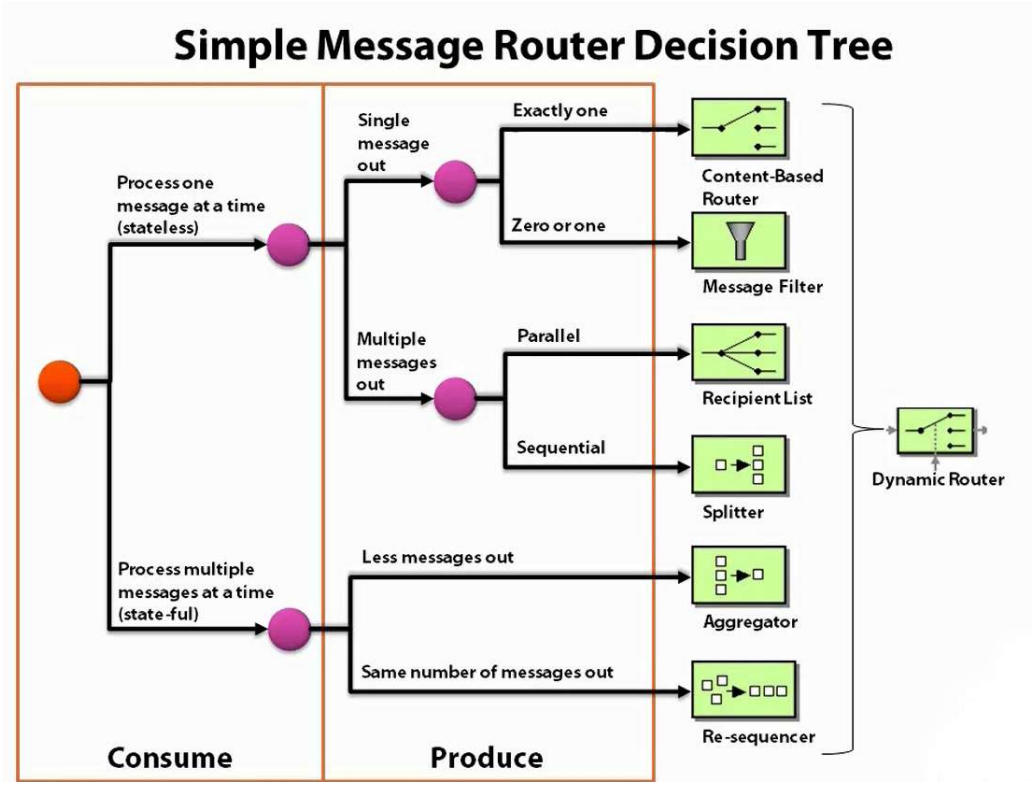
Tipos de Rotas

Segundo Michael Hoffman podemos estruturar as rotas (HOFFMAN, 2020):

- Simples (*Simple*): para um canal de entrada para um ou vários canais de saída. Possui 6 tipos de padrões de roteador
  - Baseado em conteúdo;
  - Filtro de mensagem;
  - Lista de destinatários;
  - Divisor;
  - Agregador;
  - Re-sequenciador.
- Composto (*Composed*): combina vários roteadores simples para fluxos de mensagens mais complexos. Possui 4 tipos de padrões de roteador
  - Roteador de mensagem composto;
  - *Scatter-gather;*
  - Roteamento;
  - Gerenciador de processos.

A árvore de decisão do "***Simple Message Router***" ajuda a decidir o melhor padrão de integração a ser adotado para cada situação. Vejamos:

Figura 19. Árvore de decisão do roteador de mensagem simples.



Fonte: (HOFFMAN, 2020).

Segundo Michael Hoffman, na primeira parte da *Simple Message Route Decision Tree* você decide quantas mensagens a rota consumirá:

- Uma mensagem por vez sem estado, quando não precisa manter metadados da mensagem que está consumindo;
- Várias mensagens por vez com controle de estado, precisa controlar o estado de cada mensagem que está sendo processada.

A segunda parte da árvore de decisão apresenta quantas mensagens irá produzir de acordo com o tipo de roteador:

- Baseado em conteúdo (*Content-Based Router*)
  - consome uma única mensagem e sem estado;
  - produz exatamente uma única mensagem.

- Filtro de mensagem (*Message Filter*)
    - consome uma única mensagem e sem estado;
    - pode produzir ou não uma mensagem;
    - filtra o que não é de interesse (ex. Filtrar mensagens com mais de 30d).
- Lista de destinatários (Recipient List)
    - consome uma única mensagem e sem estado;
    - produz um ou várias mensagens, para vários canais ou quando a mensagem é uma coleção de objetos que precisam ser processadas separadamente;
    - objetiva produzir mensagens em paralelo, exemplo mensagem de e-mail.
- Divisor (*Splitter*)
    - consome uma única mensagem e sem estado;
    - produz um ou várias mensagens, para vários canais ou quando a mensagem é uma coleção de objetos que precisam ser processados separadamente de forma diferente e sequencial (principal diferença entre o *Recipient List*);
    - Como exemplo seria consumir uma lista de pedidos tendo que gerar um identificador sequencial.
- Agregador (*Aggregator*)
    - consome várias mensagens por vez com controle de estado;
    - A intenção é produzir menos mensagens do que foram consumidas. Ex.: pode-se usar para agregar mensagens para o SSH. Não faz sentido enviar um pedido por vez, em vez disso usa-se o agregador para processá-los como uma coleção.
- Resequenciamento (*Re-sequencer*)
    - consome várias mensagens por vez com controle de estado;
    - A intenção é produzir o mesmo número de mensagens, mas numa ordem necessária. Ex.: ordenar mensagens por hora.

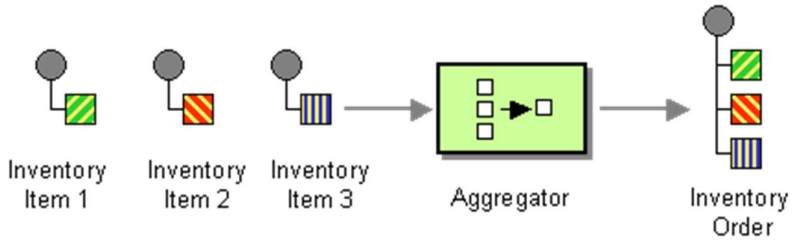
Existe um padrão adicional associado à árvore de decisão:

- *Dynamic Router* (roteador dinâmico)
  - Contempla os 6 tipos de padrões de roteamento apresentados anteriormente;
  - Permite configurar dinamicamente o destino de uma mensagem.

5.3.1.5.5 Exposição do EIP: Aggregate

O EIP Patterns Aggregator é usado para combinar um número de mensagens de entrada em uma única mensagem de saída. Pode-se visualizar isso como o inverso do padrão *Splitter* (HOHPE; WOOLF, 2002).

Figura 20. Agregado – Aggregate.



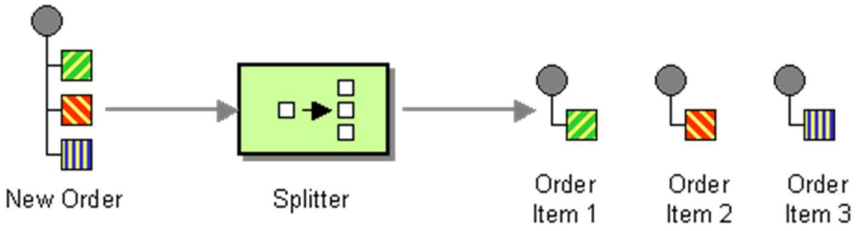
Fonte: (HOHPE; WOOLF, 2002)

Uma expressão de correlação é usada para determinar as mensagens que devem ser agregadas. Se deseja agregar todas as mensagens em uma única mensagem, basta usar uma expressão constante. Uma *Aggregation Strategy* é usada para combinar todas as trocas de mensagens para uma única chave de correlação em uma única troca de mensagens.

5.3.1.5.6 Exposição do EIP: Splitter

O EIP *patterns Splitter* permite fatiar a *message* em várias partes e processá-las individualmente.

Figura 21. Divisão - Split.



Fonte: (HOHPE; WOOLF, 2002)

Você precisa especificar um 'divisor' como parâmetro *split*().

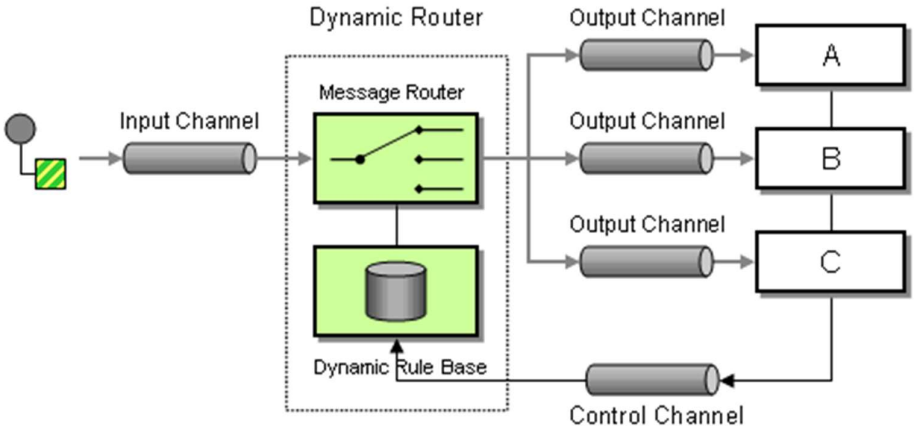
O exemplo abaixo mostra como pegar uma requisição de um endpoint **direct:a** e o fatia e partes usando uma expressão e enviando cada parte para o endpoint **direct:b**

```
from("direct:a")
   .split(body(String.class).tokenize("\n"))
   .to("direct:b");
```

5.3.1.5.7 Exposição do EIP: Dynamic Router

O EIP *Patterns Dynamic Router* permite rotear as mensagens, evitando a dependência do roteador em todos os destinos possíveis, enquanto mantém sua eficiência.

Figura 22. Roteador Dinâmico – Dynamic Router.



Fonte: (HOHPE; WOOLF, 2002)

O *Dynamic Router* no DSL avalia a rota de forma dinâmica direcionando a cada mensagem para rotas específicas.

Deve-se ter uma atenção especial para a expressão usada para o *dynamic Router*, em formato de *bean*, que deve retornar *null* para indicar o fim. Caso contrário o *dynamic Router* repetirá indefinidamente.

5.3.1.5.8 Exposição do EIP: Load Balance

O EIP *Patterns Load Balancer* permite que o processamento seja delegado ou direcionado para um dos diversos *endpoints* usando uma variedade de políticas de balanceamento (HOHPE; WOOLF, 2002).

Políticas de balanceamento: Round Robin, Randon, Sticky, Topic, Failover, Weighted Round-Robin, Weighted Randon, Custom.

Política de Balanceamento Round Robin

O balanceador de carga “Round Robin” não foi feito para funcionar com *failover*, caso precise, deve usar o balanceador de carga ***failover*** dedicado. O balanceador Round Robin somente irá mudar para o próximo endpoint em cada

mensagem. Ele é *stateful* pois ele mantém o estado do próximo *endpoint* a ser utilizado.

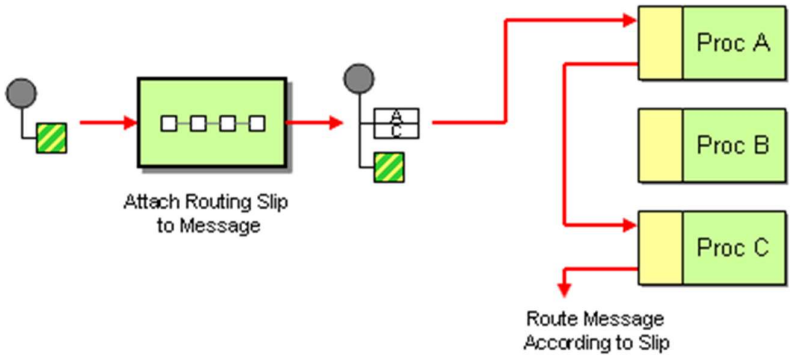
Aqui um pequeno exemplo:

```
from("direct:start")
    .loadBalance().roundRobin()
        .to("mock:x")
        .to("mock:y")
        .to("mock:z")
    .end() // end load balancer
```

5.3.1.5.9 Exposição do EIP: Routing Slip

Usado para rotear uma mensagem em uma série de etapas. A sequência de passos não é conhecida em tempo de design e pode variar para cada mensagem.

Figura 23. Deslizamento – Routing Slip.



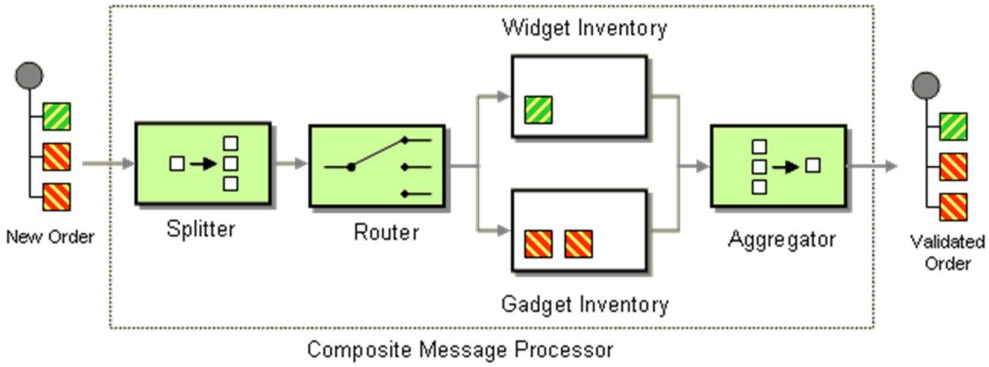
Fonte: (HOHPE; WOOLF, 2002)

5.3.1.5.10Exposição do EIP: Composed Message Processor

O EIP *Patterns Composed Message* Processor é suportado pelo Camel, e deve ser usado para processar mensagens compostas. O *Composed Message Processor* divide a mensagem, roteia as sub mensagens para os destinos apropriados e re-agregam as respostas em uma única mensagem.

Com o Camel, esse padrão é implementado pelo *Splitter*, que possui agregação embutida para re-agregar as respostas em uma única mensagem.

Figura 24. Processador de Mensagem Composto



Fonte: (HOHPE; WOOLF, 2002)

```
Exemplo:
// Esta rota inicia do endpoint direct:start
//O corpo é então dividido baseado no separador @
// O divisor em Camel também suporta InOut e para o que precisamos para ser capaz de agregar qual resposta precisamos enviar de volta, então é fornecida a própria estratégia com a classe MyOrderStrategy
from("direct:start")
    .split(body().tokenize("@"), new MyOrderStrategy())
// cada mensagem dividida é então enviada para este bean onde nós podemos processá-lo
        .to("bean:MyOrderService?method=handleOrder")
//Isso é importante para terminar a rota do divisor, pois não queremos mais fazer roteamento para cada mensagem dividida.
    .end()
//Depois de dividir e lidar com cada mensagem, queremos enviar um único conjunto de resposta de volta ao chamador original, então deixamos este bean
```

```
        construí-lo para nós. O bean receberá o resultado da estratégia de agregação:
        MyOrderStrategy
    .to("bean:MyOrderService?method=buildCombinedResponse")
```

6 DESCRIÇÃO DO EXPERIMENTO

Ao receber o problema, após a análise do contexto e premissas fornecidas pelo cliente, foi considerado a implementação apenas com o uso do Java com o framework Spring Boot, e como opção, a adoção do Apache Camel. Após algumas tentativas utilizando apenas o Java, chegou-se à conclusão de tempo estimado de implementação aproximado de 320 h, e o grau de complexidade do código estavam elevados, o que ratificou a decisão em adotar o Apache Camel na implementação (estimado em 100 h, desconsiderando-se o tempo de aprendizagem).

Em sequência apresenta-se um descritivo de como foi realizado o experimento, detalhes de código também podem ser encontrados no GitHub (https://github.com/MaviaLima/RouteBalanceator.git).

6.1 Incompatibilidades de Versão

Primeiramente, ao utilizar o Apache Camel com o Spring Boot, foram verificados alguns erros relacionados à versão. Sendo necessário verificar se as versões do Spring Boot e do Apache Camel são compatíveis entre si, caso contrário poderá incorrer no erro "**RelaxedPropertyResolved error**".

Segue abaixo algumas das versões do Apache Camel que são compatíveis com a versão do Spring Boot (IBSEN, Claus, 2018):

- Camel 2.21.x or older is Spring Boot 1.5.x;
- Camel 2.22.x is Spring Boot 2.0.x only;
- Camel 2.23.x is Spring Boot 2.1.x (and potentially also Spring Boot 2.0.x).

Para este estudo de caso foi utilizada a versão do **Spring Boot 2.2.6. RELEASE** e a **versão 2.23.1 do Apache Camel**.

6.2 Configuração do POM

O Apache Camel é uma excelente solução para integrações de sistemas, onde além do projeto "core" que fornece diversos padrões de integrações, ele também

disponibiliza uma série de subprojetos para integrações com tecnologias comuns como FTP, Mail, JMS, HTTP, entre outros, foram usadas várias referências no decorrer das configurações do Pom (CAMEL, 2021) (HOFFMAN, 2020)(IBSEN, Claus; ANSTEY, 2018).

6.2.1 Principais dependências

A partir do Camel 2.18 em diante é requerido o JDK 1.8. Neste estudo de caso foi utilizado o Camel 2.23.1 e o JDK 11 e o Maven (CAMEL, 2021).

```
<properties>
    <java.version>11</java.version>
    <camel.version>2.23.1</camel.version>
    <project.build.sourceEnconding>UTF-8</project.build.sourceEnconding>
    <project.reporting.outputEnconding>UTF-
8</project.reporting.outputEnconding>
 </properties>
```

Quando adotado o “Maven” deve-se adicionar a dependência “camel-spring-boot” no arquivo “pom.xml” (CAMEL, 2021a) conforme detalhe abaixo:

```
<dependency>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-spring-boot</artifactId>
        <version>${camel.version}</version><!-- use the same version as your
Camel core version -->
</dependency>
```

Camel-spring-boot .jar vem com o arquivo **spring.factories**, tão logo adicionado no classpath, automaticamente o Spring Boot já configura o Camel.

Componentes do Spring Boot fornecem auto configuração para o Apache Camel. A autoconfiguração do contexto Camel detecta automaticamente as rotas Camel disponíveis no contexto Spring e registra os principais utilitários do Camel (como template produtor, template consumidor e conversor de tipo) como java beans.

A partir de 2015, o Camel começa a fornecer suporte ao Spring Boot seguindo o conceito de dependências starters (camel-spring-boot-starter), que quando adicionada essa dependência, o Apache Camel inicializa no startup da aplicação, devido o recurso de auto configuração do Spring Boot.

As dependências do Camel são bem modulares, dessa forma, é preciso adicionar o camel-spring-boot-starter para adicionar a dependência de integração com o Spring Boot e o Camel Core.

O Apache Camel vem com um módulo **Spring Boot Starte**r, disponível a partir do Camel 2.17, que permite desenvolver aplicativos Spring Boot usando starters. Para usar o iniciador, adicione a dependência “camel-spring-boot-starter” no arquivo pom.xml do Spring Boot, conforme exemplo abaixo:

```
<dependency>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-spring-boot-starter</artifactId>
        <version>${camel.version}</version>
</dependency>
```

Camel-Core é o módulo básico do Apache Camel, ele contém a API pública e o Java DSL além de vários pacotes de implementação.

```
<dependency>
```

```
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-core</artifactId>
        <version>${camel.version}</version>
</dependency>
```

O componente AWS permite o trabalho com uma grande paleta de diferentes componentes, como elastic, e-mail, serviços de fila, entre outros. O principal motivo para usar a AWS é sua plataforma de computação em nuvem. Para utilização destes componentes, usuários do Maven deve-se adicionar a dependência “camel-aws” ao seu pom.xml conforme abaixo:

```
<dependency>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-aws</artifactId>
        <version>${camel.version}</version>
</dependency>
```

Para utilizar o S3 e o SQS em aplicações Java, a Amazon disponibiliza as libs específicas “aws-java-sdk-s3” e “aws-java-sdk-sqs”, são onde estão as funcionalidades para utilizar respectivamente os serviços do S3 e o SQS. O componente S3 oferece suporte ao armazenamento e recuperação de objetos de/para o serviço S3 da Amazon. O componente SQS oferece suporte ao envio e recebimento de mensagens para o serviço SQS da Amazon. O “aws-java-sdk-core” traz as principais funcionalidades relacionadas ao AWS e Java. Para o acesso aos serviços Amazon, deve-se ter uma conta válida de desenvolvedor de Amazon Web Services e estar inscrito para usar o Amazon.

```
<dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>aws-java-sdk-core</artifactId>
        <type>test-jar</type>
</dependency>
<dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>aws-java-sdk-s3</artifactId>
</dependency>
<dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>aws-java-sdk-sqs</artifactId>
</dependency>
```

Outras dependências necessárias e não menos importantes são as relacionadas aos testes unitários e às questões de gerenciamento de sistemas. O Camel fornece uma rica biblioteca com facilidades para os testes do projeto. Para Camel com Spring use “camel-test-spring”**,** o kit de testes é muito utilizado para a realização dos testes do próprio Camel.(IBSEN, ClauS; ANSTEY, 2018)

Além das dependências utilizadas para o Spring listadas abaixo, também podemos adicionar o “camel-test**”** além da “camel-test-spring**”**:

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
        <exclusions>
```

```
                <exclusion>
                        <groupId>org.junit.vintage</groupId>
                        <artifactId>junit-vintage-engine</artifactId>
                </exclusion>
        </exclusions>
</dependency>
<dependency>
        <groupId>org.junit.plataform</groupId>
        <artifactId>junit-plataform-runner</artifactId>
        <scope>test</scope>
</dependency>
<dependency>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-test-spring</artifactId>
        <version>${camel.version}</version>
        <scope>test</scope>
</dependency>
<dependency>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-test</artifactId>
        <version>${camel.version}</version>
        <scope>test</scope>
</dependency>
```

6.2.2 Dependências necessárias para Actuator

Uma aplicação Camel Spring-Boot a partir da versão 2.18.0 automaticamente fornece verificações de integridade confiáveis no endpoint “/health”, aproveitando o módulo atuador Spring-Boot. Esse é um requisito fundamental para qualquer micro serviço em um ambiente de nuvem, pois permite que a plataforma de nuvem (por exemplo, Kubernetes) detecte qualquer anomalia e execute ações corretivas. Bastando incluir a dependência “spring-boot-starter-actuator” no arquivo POM (CAMEL, 2021b).

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Esta configuração é essencial para conseguir a correta compilação em ambiente de nuvem.

Outra questão importante que vale ressaltar é que, caso deseje gerenciar quais rotas serão expostas, deverá ser incluído no “application.properties” as propriedades abaixo:

```
enabled=true
management.endpoint.metrics.enabled=true
management.endpoint.web.exposure.include =* #Para expor todos os endpoints
#O ideal é expor apenas os endpoints que sejam estritamente necessários.
management.endpoint.shutdown.enabled=true
management.endpoint.info.enabled=true
```

6.3 Implentação da Solução

6.3.1 Considerações

Auto-Configured Camel Context

A parte mais importante da funcionalidade fornecida pela configuração automática do Camel é a instância do CamelContext. A configuração automática do Camel cria um *SpringCamelContext* que cuida da inicialização e encerramento adequados desse contexto. O contexto Camel criado também é registrado no contexto do aplicativo Spring (sob o nome do bean camelContext), portanto, pode-se acessar como qualquer outro bean Spring (KONSEK, 2019).

Auto-Detecting Camel Routes

A configuração automática do Camel coleta todas as instâncias do RouteBuilder do contexto do Spring e as injeta automaticamente no CamelContext fornecido. Isso significa que é criada nova rota Camel com o iniciador Spring Boot é tão simples quanto adicionar a classe anotada @Component ao seu classpath (KONSEK, 2019).

6.3.2 Application Class

Spring Boot não requer nenhum layout de código específico para funcionar. No entanto, existem algumas práticas recomendadas que ajudam.

Geralmente, recomenda-se que posicione a classe principal do aplicativo em um pacote raiz acima de outras classes. Neste caso, o arquivo “Application.java” deve declarar o método principal, junto com o básico @SpringBootApplication, da seguinte maneira:

```
package com.camel;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
@SpringBootApplication
public class MicroServiceCamelApplication {

        public static void main(String[] args) {
      SpringApplication.run(MicroServiceCamelApplication.class, args);
   }
}
```

MicroServiceCamelApplication.java

6.3.3 Ingestão de Variáveis de Ambiente

Talvez a ***@Value*** annotation seja familiar. Ela pode ser usada para injetar o valor das propriedades em um Bean. Também pode-se usar essa anotação para colocar o valor da variável de ambiente, por exemplo, "***region***" com a sintaxe @Value("${region }").

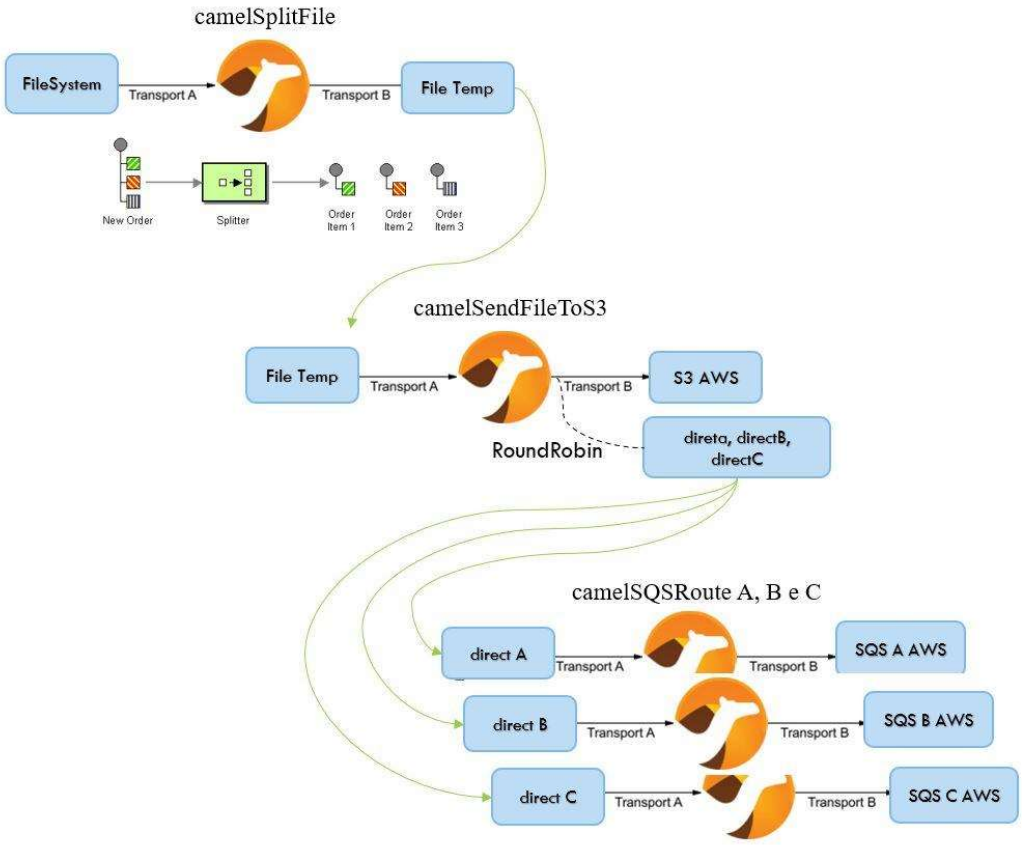
```
@Value("${awsRegion}")
private String region;
```

O Camel reconhece as variáveis de ambiente em tempo de execução, E na rota pode-se usar a sintaxe de espaço reservado {{..}} para se referir a propriedade, conforme exemplo abaixo.

```
 .to("aws-
s3://{{outBucket}}?deleteAfterWrite=false=false&AmazonS3Client=#s3Client")
```

6.3.4 Estruturação das Rotas

Figura 25. Processador de Mensagem Composto



Fonte: Autor

Para a implementação da solução o problema foi dividido em 3 blocos com rotas distintas a serem realizadas pelo Camel (ALRABADI, 2014). Vejamos como foi implementado cada trecho de rota e quais desafios para cada situação:

- *camelSplitFile*: responsável pela primeira parte, lê o arquivo ".csv" do file system e realizar a operação de split, direcionando para um diretório temporário;
- *camelSendFileToS3*: responsável por enviar os arquivos processados pelo camelSplitFile para o S3 AWS e distribuir o nome dos arquivos processados para 3 novas rotas (directA, directB e directC) de forma balanceada através de um Round Robin;

- *camelSQSRouteA*: responsável por enviar a mensagem com o nome dos arquivos processados e direcionados para esta rota via AWS-SQS para a fila 1 (directA);
- *camelSQSRouteB*: responsável por enviar a mensagem com o nome dos arquivos processados e direcionados para esta rota via AWS-SQS para a fila 2 (directB);
- *camelSQSRouteC*: responsável por enviar a mensagem com o nome dos arquivos processados e direcionados para esta rota via AWS-SQS para a fila 3 (directC).

O código completo da implementação do fluxo de roteamento proposto pode ser verificado no GitHub ou no Anexo I. Vejamos cada trecho da rota com mais detalhe.

6.3.5 Construindo uma rota Camel com Splitter

O primeiro passo é a extensão da classe *RouteBuider* e a implementação do método configure, conforme abaixo:

```
class MyRouteBuilder extends RouteBuilder {
        public void configure() throws Exception { ...
        }
}
```

Dentro do método configure() são implementadas as rotas camel.

Vamos analisar o primeiro trecho da rota "**camelSplitFile**"; nesta solução podemos observar o uso do padrão splitter (SATISH, 2015) conforme **Figura 21.** Divisão - Split.

O aplicativo basicamente lê os arquivos um-por-um a partir de um caminho configurado para um diretório, em seguida, faz o processamento dividindo o arquivo e re-agrupando, de acordo com o número de linhas passadas por parâmetro, no final, salva este novo arquivo em um diretório de destino.

```
from("file:" + SOURCE_FOLDER +"?delete=true")
                    .routeId("camelSplitFile")
                    .log("Start processing ...")
                    .multicast()
                    .marshal()
                    .string("UTF-8")
                    .split()
                    .tokenize("\n", rangeLines)
                    .process(e -> e.getIn().setHeader("uid",
UUID.randomUUID().toString()))
                              .setHeader(Exchange.FILE_NAME,
simple("${file:name.noext}-${in.header.uid}-
${data:now:yyyyMMddHHmmssSSS}.${file:ext}"))
                    .to("file://"+ DESTINATION_FOLDER );
```

Outras funções foram agregadas para a correta manipulação do arquivo, vamos entender cada uma:

A identificação da rota é importante para podermos referenciar em testes e logs por exemplo, isto é feito através do ***routeId***, caso não seja definido um nome default é atribuído pelo sistema.

Os logs da rota podem ser registrados pelo parâmetro .***log***().

O ***multicast***() refere-se a um *Aggregation Strategy* a ser usado para reunir as respostas dos multicast em uma única mensagem de saída do *multicast* . Por padrão, o Camel usará a última resposta como mensagens de saída.

O ***marchal***(), no DSL pode-se usar uma instância de *DataFormat*, pode configurar o *DataFormat* dinamicamente usando o DSL ou pode referir-se a uma instância nomeada do formato especificado. Nesta solução foi possível verificar que o Camel não consegue processar se o arquivo não estiver no formato **string("UTF-8")**, por este motivo, deve-se utilizar este recurso para garantir a conversão para o formato esperado.

O ***split***() é a função principal responsável pela divisão do arquivo lido.

O ***tokenize***("\n", rangeLines) é o responsável por definir o formato da divisão do arquivo lido em um novo grupo de opções que permite agrupar N partes, por exemplo, dividir arquivos de 10.000 linhas em 10 arquivos de 10 linhas cada.

O ***process***(e-->e.getIn().setHeader("uid", UUID.randomUUID().toString())): o process permite a execução de códigos Java, customizados, que não fazem parte dos componentes do Camel. Foi utilizado este recurso para que fosse gerado um novo UUID para cada arquivo. Observamos que o UUID próprio do Camel gerava apenas uma vez no início da rota e todos os arquivos gerados continham o mesmo UUID, e só era gerado um novo caso o contexto fosse reiniciado. (NPEDER, 2016)

O Header contém um conjunto de informações relacionadas a transação que está sendo realizada, neste contexto foi incorporado através do ***setHeader***() para a chave, o nome do arquivo (Exchange.FILE_NAME), e seu respectivo valor: simple("${file:name.noext}-${in.header.uid}-${data:now:yyyyMMddHHmmssSSS}.${file:ext}")).

6.3.6 Construindo uma rota com conector AWS-S3 e Load Balancer

Primeiramente se faz necessário construir a conexão ao serviço Amazon através do uso do conector AWS-S3. No código abaixo há um método ***init***() e sobre este a tag @PostConstruct, neste método é instanciado o objeto **s3Client** do tipo AmazonS3, através do uso do construtor padrão Amazon, que implicitamente chama o método *getAmazonS3Client*() (KONSEK, 2019).

A tag de anotações @PostConstruct permite que o Spring chame apenas uma vez, logo após a inicialização das propriedades do bean, este método é executado

mesmo se não houver nada para inicializar. (*Spring PostConstruct and PreDestroy Annotations | Baeldung*, n.d.)

No método *getAmazonS3Client*() é referenciado o nome do *bean* "s3Client" que será registrado ao CamelContext, através do método simpleRegistry() dentro do método init(). O Camel mantém um registro que permite consultar todos os *beans* nele registrados.

```
private AmazonS3 s3Client;
SimpleRegistry simpleRegistry;

@PostConstruct
private void init() {
        s3Client = AmazonS3ClientBuilder.standard().withRegion(region).build();
        simpleRegistry = new SimpleRegistry();
        simpleRegistry.put("s3Client", s3Client);
}

@Bean(name = "s3Client")
public AmazonS3 getAmazonS3Client() {
        return s3Client;
}
```

O conector "aws-s3" faz uso do *bean* "s3Client" e este deve estar corretamente registrado no *CamelContext* através do *SimpleRegistry*, conforme demonstrado acima, caso contrário, ocorrerá um erro de identificação do tipo de dado, uma vez que o Camel interpretará o parâmetro como um simples String e não reconhecerá o objeto do tipo AmazonS3Client, quando este parâmetro é passado na rota ".to(..#s3Client.)" detalhada abaixo.

```
to("aws-
s3://{{outBucket}}?deleteAfterWrite=false=false&AmazonS3Client=#s3Client")
```

Vejamos o segundo trecho da rota “**camelSendFileToS3**”, nesta solução podemos observar o uso do padrão LoadBalancer, além de utilizar um conector aws-s3**:**

```
from("file://"+ DESTINATION_FOLDER )
        .routeId("camelSendFileToS3")
        .log("Start sending index ${header.CamelSplitIndex} splited file to s3 ...")
         .setHeader(S3Constants.CONTENT_LENGTH,
simple("${in.header.CamelFileLength}"))
         .setHeader(S3Constants.KEY, simple("${in.header.CamelFileNameOnly}"))
                  .to("aws-
s3://{{outBucket}}?deleteAfterWrite=false=false&AmazonS3Client=#s3Client")
         .log("Start direct message to SQS by roundRobin ...")
         .loadBalance().roundRobin().to("direct:a", "direct:b", "direct:c")
         .end();
```

Neste trecho o aplicativo basicamente lê os arquivos um-por-um, disponibilizados no mesmo diretório de saída da primeira rota, em seguida, processa cada um, encaminhando para o conector aws-s3 e paralelamente redireciona para uma nova rota de forma balanceada, através do loadBalance, equilibradamente para os “direct:a”, “direct:b” e “direct:c”.

Neste contexto é importante ressaltar a importância de acrescentar as informações de Header **CONTENT_LENGTH** e **KEY** para cada arquivo antes de seu envio para o componente “aws-s3” conforme apresentado no exemplo acima.

6.3.7 Construindo uma rota implementando o Process

Houveram algumas dificuldades em identificar a compatibilidade entre as dependências do Camel e o Spring Boot, com a utilização do conector AWS-SQS para uma fila FIFO concomitantemente com outro *bean* já existente, o AWS-S3 que levaram a uma implementação de um componente customizado. De toda a forma segue abaixo

um exemplo de como pode ser implementado o envio para uma fila FIFO AWS-SQS pelo Camel utilizando conector AWS-SQS, conforme exemplo abaixo:

Figura 26. Exemplo de rota connector AWS-SQS

```
from("direct:a").routeId("camelSQSRouteA")
    .setHeader("messageGroupIdStrategy", constant("1596550402"))
    .setBody(constant("${headers.CamelFileName}"))
    .to("aws-sqs://{{smsQueue1}}?amazonSQSClient=#sqsClient)")
    .log("${in.header.CamelFileNameOnly} succesfully uploaded to S3 {{smsQueue1}} bucket");
```

Fonte: o autor

Como uma opção de solução, é possível fazer uso de uma implementação customizada, neste caso pode-se realizar o acesso aos serviços AWS, sem acionar os plugins do Camel. Através do uso do *Process* (RED HAT, 2021c), que permite a chamada de um código customizado em java.

Para isso foi implementado através do *Process* a execução do envio de mensagens para a fila FIFO AWS-SQS. Como as três rotas “direct:a”, “direct:b” e “direct:c” fazem uso de um mesmo serviço, “processWithQueue”, o envio da mensagem para a fila na AWS foi utilizado com a mesma estrutura de código, diferenciando apenas o parâmetro de entrada “nameQueue”.

Exemplo de implementação customizada (process):

```
public void processWithQueue(Exchange exchange, String nameQueue) throws
Exception {
        Map<String, Object> headers = exchange.getIn().getHeaders();
        String fileName = (String) headers.get("CamelFileName");
        String result = sqsService.sendMenssageToQueue(fileName,
sqsService.getQueueURL(nameQueue));
        exchange.getOut().setBody(result);
}
```

Exemplo de rota:

```
from("direct:a").routeId("camelSQSRouteA")
```

```
        .log("Starting send ${in.reader.CamelFileName} to {{outQueue1}}
queue")
        .process(new Processor(){
            public void process(Exchange exchange) throws Exception {
                processWithQueue(exchange, outQueue1);
            }
        })
        .log("Filename successfully send to {{outQueue1}} queue");
```

O método **SendMessageToQueue**() abaixo implementado, envia a mensagem para uma fila FIFO AWS-SQS.

```
public String sendMessageToQueue(String message, String queueUrl) throws
Exception {
        SendMessageRequest sendMessageRequest = new
SendMessageRequest()
                                .withQueueUrl(queueUrl);

        String timeStamp = new
SimpleDateFormat("yyyy.MM.dd.HH.mm.ss").format(new Date());
        sendMessageRequest.setMessageBody(message);
        sendMessageRequest.setMessageGroupId(timeStamp);

        SendMessageResult sendMessageResult =
sqsClient.sendMessage(sendMessageRequest);

        return sendMessageResult.getMessageId();
```

```
}
```

O Amazon SQS (Simple Queue Service) é uma solução de fila para troca de mensagens entre aplicações hospedada pela própria Amazon, assim seu gerenciamento é todo através de serviços, o que faz com que toda a complexidade da infraestrutura seja terceirizada para AWS, dessa forma, deixando apenas o foco para o uso da solução em si.

As filas *FIFO (ordem de chegada)* são projetadas para aprimorar o sistema de mensagens entre aplicativos, quando a ordem das operações e dos eventos é crítica, ou quando duplicatas não podem ser toleradas. As mensagens podem ser agrupadas através de uma tag (ID do grupo de mensagens) que especifica que a mensagem pertence a um grupo específico. As mensagens que pertencem ao mesmo grupo de mensagens são processadas sempre uma-por-uma, em uma ordem estrita em relação ao grupo de mensagens (porém, as mensagens que pertençam a grupos diferentes de mensagens podem ser processadas fora de ordem), veja o código no ANEXO II - SqsAWSService.

A classe **AmazonSQSConfig** implementa o método **retClientSQS** que realiza a conexão/autenticação com o serviço e retorna um objeto do tipo **AmazonSQS**:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;


@Configuration
public class AmazonSQSConfig {


    @Bean
```

```
        public AmazonSQS retClientSQS {
            return AmazonSQSClientBuilder
                .standard()
                .withRegion(Regions.SA_EAST_1)
                .build();
}
```

6.3.8 Implementando os testes unitários

O aplicativo Camel pode ser utilizado em muitos ambientes/frameworks. Neste estudo de caso foi utilizado Camel com Spring Boot. O Spring Boot torna mais fácil o desenvolvimento, execução e testes dos aplicativos, para isto deve-se adicionar dependências do Camel para Spring Boot conforme mencionado no capítulo 6.2.1 Principais dependências, e utilizar a *annotation* @SpringBootTest.

Para executar o teste com o JUnit, deve usar o CamelSpringBootRunner como @RunWith, em seguida, referir a classe do aplicativo Spring Boot “MyApplication”. No restante deve-se injetar a dependência CamelContext e ProducerTemplate para torná-los disponíveis para uso em seus métodos de teste. Veja em “Anexos” a implementação dos testes.

6.3.8.1 Construção da classe de testes para o Camel

Conforme mencionado para executar o teste com o JUnit deve utilizar o “CamelSpringBootRunner” como @RunWith, em seguida, referir a classe de aplicativo Spring Boot “MicroserviceCamelApplication”.

Em adição pode-se utilizar o @TestPropertySource que permite que sejam injetados valores específicos para a realização dos testes no lugar das variáveis previamente injetadas na aplicação.

Isto muitas vezes é necessário pois não é possível utilizar as rotas reais na realização dos testes ou mesmo outros parâmetros utilizados. Vejamos exemplo abaixo:

```
@RunWith(CamelSpringBootRunner.class)
@SpringBootTest(classes = MicroServiceCamelApplication.class)
@TestPropertySource(properties = {"folderPathIn=//target/inbox",
"folderPathOut=//target/outbox", "rangeLines=2"})
@UseAdviceWith
public class RouteBalanceatorTests {
        @Autowired
    private CamelContext context;

    @Autowired
    private ProducerTemplate template;

    @EndpointInject(uri = "mock:s3")
    private MockEndpoint mocks3;

    @MockBean
    AmazonS3 s3Client;

    @MockBean
    private SqsAWSService sqsService;
```

6.3.8.2 Teste de movimentação do arquivo

A interface “ProducerTemplate” permite a trocas de mensagens para terminais de várias maneiras diferentes, para facilitar o trabalho com instâncias do Camel Endpoint a partir do código Java. Neste caso é montado um simples teste de movimentação do arquivo. Pode ser configurado com um endpoint padrão caso deseje apenas enviar muitas mensagens para o mesmo endpoint; ou pode-se especificar um Endpoint ou uri como o primeiro parâmetro. O segundo parâmetro é o conteúdo do arquivo e o terceiro parâmetro o nome do arquivo.

```
@Test
  public void testMoveFile() throws Exception {
      template.sendBodyAndHeader("file://target/inbox", "Hello World",
Exchange.FILE_NAME, "hello.csv");

      Thread.sleep(2000);

      File target = new File("target/outbox");
      assertTrue("File not moved", target.exists());
  }
```

6.3.8.3 Teste de Rota

O FluentProducerTemplate é um modelo para trabalhar com Camel e envio de mensagens através do uso do Exchange a um determinado Endpoint usando um estilo de construção *fluente*. Neste modelo é possível referenciar qual o trecho específico de rota deseja-se testar. (CAMEL, 2021b)

```
@Test
```

```
    public void testRoute() throws Exception {


        FluentProducerTemplate fluentTemplate =
context.createFluentProducerTemplate();
        Exchange exchange = fluentTemplate.withProcessor(e->{
                e.getIn().setHeader(Exchange.FILE_NAME,"helloword.csv");
                e.getIn().setBody("Hello World/n"+
                "Hi linha2/n"+
                "Hi linha 3/n"+
                "Hi linha 4/n");
        }).to("file://target/inbox")
                        .send();


        Object resposta = exchange.getIn().getBody();
        assertNotNull(resposta);
        assertTrue(resposta.toString().toLowerCase().contains("hello world"));
    }
```

6.3.8.4 Teste routeBalance

Este teste já apresenta a utilização de mocks, uma vez que o resultado depende de um serviço externo (SQS Amazon).

```
@Test
    public void WhenSendMessageToSQS_A() throws Exception {
      String resultExpected = "test";


      when(sqsService.getQueueUrl(Mockito.anyString())).thenReturn("url");
```

```
    when(sqsService.sendMessageToQueue(Mockito.anyString(),
Mockito.anyString())).thenReturn(resultExpected);


    FluentProducerTemplate fluentTemplate =
context.createFluentProducerTemplate();
    Exchange exchange = (Exchange) fluentTemplate.withProcessor(e-> {
       e.getIn().setHeader(Exchange.FILE_NAME, "helloworld.csv");
    }).to("direct:a");


    String resultMessage = exchange.getIn().getBody(String.class);
    assertNotNull(resultMessage);
    assertTrue(resultExpected.equalsIgnoreCase(resultMessage));
  }
```

Veja na seção de anexos o código completo da classe **RouteBalanceatorTests.**

Destaca-se também a implementação dos testes das classes “não Camel”, veja exemplo **SqsAWSServiceTests,** conforme ANEXO IV. Um ponto de atenção é que todas as classes de testes do projeto devem estar alinhadas no uso de uma mesma versão de JUnit, caso contrário estas não serão reconhecidas. Ex. JUnit 4

7 CONCLUSÃO

O estudo de padrões de integração e seu uso deve ser constantemente abordado e estimulado, em busca de otimizações e melhorias no processo de desenvolvimento de softwares, principalmente para grandes volumes de dados.

Existem diversas formas de implementar soluções de integração utilizando padrões de projeto, todavia, deve-se levar em consideração os fatores de recursos disponíveis, uso de memória, custo, taxa de envios de mensagens entre outras. Este estudo teve premissas que foram colocadas pelo cliente final, sendo uma delas a adoção do Apache Camel. Bem como foram descartadas outras possíveis soluções como Kafka, Lambda AWS, Mule Software, Spring Integration por questões de prerrogativas e limitações destas soluções.

Sobre o Apache Camel, embora traga diversas facilidades, seu entendimento não é tão trivial, a documentação padrão do https://camel.apache.org/ embora muito rica, não traz muitos exemplos práticos, o que gera muitas dúvidas. Outro fato que vale ressaltar é sobre a quase inexistência de documentação em nossa língua nativa, o que também pode vir a ser um fator de dificuldade para algumas pessoas.

Dentre as principais dificuldades encontradas podemos ressaltar a necessidade de entendimento dos padrões de projeto para o melhor aproveitamento dos recursos. O Apache Camel se apresentou como a melhor opção para implementação de micro-serviço para atender a gestão e manipulação de grandes volumes de dados, com uma rápida implementação e menor custo. Onde o tempo para implementação sem o uso de Apache Camel, utilizando apenas rotinas Java/Spring Boot, foi estimado em 320h enquanto a implementação utilizando o Apache Camel foi estimado em 100h, um ganho de tempo estimado em 68,75%, este tempo não considera o tempo de aprendizagem e entendimento da ferramenta e conceitos sobre os padrões utilizados.

Este trabalho resultou em uma implementação com sucesso para um problema de integração real em um cliente de instituição bancária, com transformação de grandes volumes de dados entre plataformas distintas. A solução implementada permitiu a manipulação e envio de dados de forma que, o tempo esperado para o cliente receber um volume de 200 milhões de registros com um tamanho total de 190

Gigabytes, ficou entre 1,2 horas e 3 horas. Foi reconhecido pelo cliente como um trabalho de grande entrega de valor, com agilidade na implementação, coesão de código, e simplicidade no entendimento do processo, além de outros benefícios agregados pela adoção dos padrões que já são incorporados pelo Apache Camel como segurança e performance.

Neste estudo foram utilizados padrões de integração entre sistemas, como Message, *Splitter*, e *Load Balance* entre outras. O uso destes padrões possibilitou uma grande coesão de código, e uma melhor visibilidade do fluxo do processo. Buscou-se estruturar o documento de forma a apresentar um roteiro de implementação, para que sirva de guia para futuros desenvolvedores e acadêmicos. Um arcabouço de conteúdo ou base, para que outras pessoas que enfrentam problemas de integração de aplicações, possam ter este trabalho como ponto de partida.

A adoção do Apache Camel na implementação da solução trouxe:

- Uma abordagem mais escalável;
- Rapidez na implementação;
- Coesão de código;
- Confiabilidade do processo.

Como trabalhos futuros pode-se sugerir:

- O aprofundamento no estudo e adoção de padrões de integração utilizando o Apache Camel como ferramenta facilitadora;
- Complementação do presente estudo de caso, com possíveis melhorias de código;
- Apresentação de implementação de estudo de caso utilizando outras soluções: Spring Integration, Lambda AWS.

REFERÊNCIAS

ALRABADI, G. How to Implement the Splitter and Aggregator Patterns with Apache Camel. **Source Allies**, 2014. Disponível em: <https://www.sourceallies.com/2014/01/how-to-implement-the-splitter-and-aggregator-patterns-with-apache-camel/>. Acesso em: 12 abr. 2021.

CAMEL, A. Enterprise Integration Patterns :: Apache Camel. **Apache Software Foundation**, 2021a. Disponível em: <https://camel.apache.org/components/latest/eips/enterprise-integration-patterns.html>. Acesso em: 29 ago. 2020.

_____. APACHE CAMEL 2.18.0 RELEASE. **Apache Software Foundation**, 2021b. Disponível em: <https://camel.apache.org/releases/release-2.18.0/>. Acesso em: 29 ago. 2020.

CAROLINA SALGADO BERNADETTE FARIAS LÓSCIO, A. **Integração de Dados na Web**. Disponível em: <https://www.cin.ufpe.br/~if696/referencias/integracao/JAI01.pdf>. Acesso em: 29 nov. 2020.

CUNHA, M. X. C.; JUNIOR, M. F. S.; DORNELAS, J. S. **O uso da arquitetura SOA como estratégia de integração de sistemas de informação em uma instituição pública de ensino.** Alagoas/Recife: SEGeT, 2014. p. 13.

DANNY ZHANG. System Design Topics: CAP Theorem. **D.Z notes**, 2020. Disponível em: <http://dannyzhang.run/2020/03/21/system-desing-1/>. Acesso em: 10 abr. 2021.

FRANZINI, F. Java Frameworks – Enterprise Integration Patterns. **Fernando Franzini Blog**, 2017. Disponível em: <https://fernandofranzini.wordpress.com/2017/09/08/java-frameworks-enterprise-integration-patterns/>. Acesso em: 11 abr. 2021.

GAEA. Integração de sistemas com grande volume de dados. **Gaea Consulting**, 2020. Disponível em: <https://gaea.com.br/integracao-de-sistemas-com-grande-volume-de-dados/>. Acesso em: 3 nov. 2020.

HOFFMAN, M. Introduction to Integration With Apache Camel. **Pluralsight**, 2020. Disponível em: <https://app.pluralsight.com/player?course=apache-camel-intro-integration&author=michael-hoffman&name=apache-camel-intro-integration-m1&clip=0&mode=live>. Acesso em: 25 ago. 2020.

HOHPE, G.; WOOLF, B. **Enterprise Integration Patterns**. PLoP 2002 conference, 2002

IBSEN, Claus. incompatible version of apache.camel with Spring.Boot v2.0 - Stack

Overflow. **Stack Overflow**, 2018. Disponível em: <https://stackoverflow.com/questions/53920513/incompatible-version-of-apache-camel-with-spring-boot-v2-0>. Acesso em: 30 ago. 2020.

_____; ANSTEY, J. **Camel in Action**. 2nd Edition, 2018.

IBSEN, ClauS; ANSTEY, J. **Camel in Action, Second Edition**. 2018.

KOLB, P. **Realization of EAI Patterns with Apache Camel**. 2008. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.214.3580&rep=rep1&type=pdf>.

KONSEK, H. Spring Boot - Apache Camel - Apache Software Foundation. **Atlassian Confluence**, 2019. Disponível em: <https://cwiki.apache.org/confluence/display/CAMEL/Spring+Boot>. Acesso em: 30 ago. 2020.

MICHAELIS. Integração | Dicionário Brasileiro da Língua Portuguesa. **Editora Melhoramentos Ltda.**, 2021. Disponível em: <http://michaelis.uol.com.br/busca?id=BVqRl>. Acesso em: 13 dez. 2020.

NPEDER. java - Camel route fails to generate random UUID - Stack Overflow. **Stack Overflow**, 2016. Disponível em: <https://stackoverflow.com/questions/35745371/camel-route-fails-to-generate-random-uuid>. Acesso em: 9 set. 2020.

PESSOA MELLO, A. P. (SLTI/MP); MESQUITA, H. (SLTI/MP); VIEIRA, C. E. (SLTI/MP). Introdução à Interoperabilidade. **Enap - Escola Nacional de Administração Pública**, 2015. Disponível em: <https://repositorio.enap.gov.br/bitstream/1/2399/1/Módulo_1_EPING.pdf>. Acesso em: 11 fev. 2021.

RADES, P. R. Quando interoperar e quando integrar? Existe diferença? **Interopera**, 2017. Disponível em: <http://interopera.esy.es/interoperabilidade/>. Acesso em: 13 dez. 2020.

RED HAT. What is integration? **Red Hat, Inc**, 2021a. Disponível em: <https://www.redhat.com/pt-br/topics/integration/what-is-integration>. Acesso em: 13 dez. 2020.

_____. Introdução à integração corporativa. **Red Hat, Inc**, 2021b. Disponível em: <https://www.redhat.com/pt-br/topics/integration>. Acesso em: 3 nov. 2020.

_____. Chapter 11. System Management Red Hat Fuse 7.0. **Red Hat, Inc**, 2021c. Disponível em: <https://access.redhat.com/documentation/en-us/red_hat_fuse/7.0/html/apache_camel_development_guide/sysman>. Acesso em:

29 ago. 2020.

SATISH, R. Apache Camel Split XML File Example. **Java Articles**, 2015. Disponível em: <https://www.javarticles.com/2015/07/apache-camel-split-xml-file-example.html>. Acesso em: 12 abr. 2021.

SOMBRIO, J. Integração de Aplicações e Integração de Dados: Entenda a Diferença. **Kondado**, 2020. Disponível em: <https://kondado.com.br/blog/blog/2020/08/04/integracao-de-aplicacoes-e-integracao-de-dados-entenda-a-diferenca/>. Acesso em: 13 dez. 2020.

SORDI, J. O. DE; MARINHO, B. De L. Integração entre sistemas: Análise das abordagens praticadas pelas corporações brasileiras. **Revista Brasileira de Gestao de Negocios**, 2007. v. 9, n. 23, p. 78–93.

VERNADAT, F. B. Book Reviews : Enterprise Modeling and Integration Principles and Applications, Authored by François B. Vernadat; Published by Chapman and Hall, London, UK © 1996 ISBN 0-412-60550-3. **Concurrent Engineering**, 26 jun. 1997. v. 5, n. 2, p. 195. Disponível em: <http://journals.sagepub.com/doi/10.1177/1063293X9700500211>.

ANEXOS

ANEXO I - RouteBalanceator

```
package com.camel;

import java.util.Map;

import java.util.UUID;

import javax.annotation.PostConstruct;

import org.apache.camel.Exchange;

import org.apache.camel.LoggingLevel;

import org.apache.camel.Processor;

import org.apache.camel.builder.RouteBuilder;

import org.apache.camel.component.aws.s3.S3Constants;

import org.apache.camel.impl.SimpleRegistry;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.context.annotation.Bean;

import org.springframework.stereotype.Component;


import com.amazonaws.services.s3.AmazonS3;

import com.amazonaws.services.s3.AmazonS3ClientBuilder;

import com.camel.services.SqsAWSService;


@Component

public class RouteBalanceator extends RouteBuilder {


	@Autowired

	private SqsAWSService sqsService;
```

```
@Value("${rangeLines}")
private Integer rangeLines;

@Value("${awsRegion}")
private String region;

String queueURL;

@Value("${outBucket}")
private String outBucket;

@Value("${outQueue1}")
private String outQueue1;

@Value("${outQueue2}")
private String outQueue2;

@Value("${outQueue3}")
private String outQueue3;

@Value("${folderPathIn}")
private String SOURCE_FOLDER;

@Value("${folderPathOut}")
private String DESTINATION_FOLDER;

private AmazonS3 s3Client;
```

```
        SimpleRegistry simpleRegistry;

        @PostConstruct
        private void init() {
                s3Client =
AmazonS3ClientBuilder.standard().withRegion(region).build();

                simpleRegistry = new SimpleRegistry();
                simpleRegistry.put("s3Client", s3Client);
        }

        @Bean(name = "s3Client")
        public AmazonS3 getAmazonS3Client() {
                return s3Client;
        }

        @Override
           public void configure() throws Exception {

              onException(Exception.class)
                 .routeId("camelException")
                 .log(LoggingLevel.ERROR, "Exception in Camel")
                 .handled(true)
                 .end();

              from("file:" + SOURCE_FOLDER +"?delete=true")
                 .routeId("camelSplitFile")
```

```
                .log("Start processing ...")
                .multicast()
                .marshal()
                .string("UTF-8")
                .split()
                .tokenize("\n", rangeLines)
                .process(e -> e.getIn().setHeader("uid",
UUID.randomUUID().toString()))
                        .setHeader(Exchange.FILE_NAME,
                    simple("${file:name.noext}-${in.header.uid}-
${data:now:yyyyMMddHHmmssSSS}.${file:ext}"))
                .to("file://"+ DESTINATION_FOLDER );


            from("file://"+ DESTINATION_FOLDER )
                .routeId("camelSendFileToS3")
                .log("Start sending index ${header.CamelSplitIndex} splited file to s3
...")
                .setHeader(S3Constants.CONTENT_LENGTH,
simple("${in.header.CamelFileLength}"))
                .setHeader(S3Constants.KEY,
simple("${in.header.CamelFileNameOnly}"))
                .to("aws-
s3://{{outBucket}}?deleteAfterWrite=false=false&AmazonS3Client=#s3Client")
                .log("Start direct message to SQS by roundRobin ...")
                .loadBalance().roundRobin().to("direct:a", "direct:b", "direct:c")
                .end();
```

```
            from("direct:a").routeId("camelSQSRouteA")
                .log("Starting send ${in.reader.CamelFileName} to {{outQueue1}}
queue")
                .process(new Processor(){
                    public void process(Exchange exchange) throws Exception {
                        processWithQueue(exchange, outQueue1);
                    }
                })
                .log("Filename succesfully send to {{outQueue1}} queue");

            from("direct:b").routeId("camelSQSRouteB")
                .log("Starting send ${in.reader.CamelFileName} to {{outQueue2}}
queue")
                .process(new Processor(){
                    public void process(Exchange exchange) throws Exception {
                        processWithQueue(exchange, outQueue2);
                    }
                })
                .log("Filename succesfully send to {{outQueue2}} queue");

                from("direct:c").routeId("camelSQSRouteC")
                .log("Starting send ${in.reader.CamelFileName} to {{outQueue3}}
queue")
                .process(new Processor() {
                    public void process(Exchange exchange) throws Exception {
                        processWithQueue(exchange, outQueue3);
                    }
                })
```

```
                .log("Filename succesfully send to {{outQueue3}} queue");
        }

    public void processWithQueue(Exchange exchange, String nameQueue)
throws Exception {
            Map<String, Object> headers = exchange.getIn().getHeaders();
            String fileName = (String) headers.get("CamelFileName");
            String result = sqsService.sendMenssageToQueue(fileName,
sqsService.getQueueURL(nameQueue));
            exchange.getOut().setBody(result);
    }
}
```

ANEXO II - SqsAWSService

```
package com.camel.services;
import java.text.SimpleDateFormat;
import java.util.Date;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.model.SendMessageRequest;
import com.amazonaws.services.sqs.model.SendMessageResult;

@Service
public class SqsAWSService {
	@Autowired
	private AmazonSQS sqsClient;

	public String getQueueUrl(String queueName) {
		return sqsClient.getQueueUrl(queueName).getQueueUrl();
	}

	public String sendMessageToQueue(String message, String queueUrl) throws
Exception {
	SendMessageRequest sendMessageRequest = new SendMessageRequest()
			.withQueueUrl(queueUrl);
```

```
        String timeStamp = new SimpleDateFormat("yyyy.MM.dd.HH.mm.ss").format(new Date());
        sendMessageRequest.setMessageBody(message);
        sendMessageRequest.setMessageGroupId(timeStamp);

        SendMessageResult sendMessageResult = sqsClient.sendMessage(sendMessageRequest);

            return sendMessageResult.getMessageId();
        }
}
```

ANEXO III - RouteBalanceatorTests

```
package com.camel;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;
import static org.mockito.Mockito.when;

import org.apache.camel.CamelContext;
import org.apache.camel.EndpointInject;
import org.apache.camel.Exchange;
import org.apache.camel.FluentProducerTemplate;
import org.apache.camel.ProducerTemplate;
import org.apache.camel.component.mock.MockEndpoint;
import org.apache.camel.test.spring.CamelSpringBootRunner;
import org.apache.camel.test.spring.UseAdviceWith;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.runner.RunWith;
import org.mockito.Mockito;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.context.TestPropertySource;

import java.io.File;
import com.amazonaws.services.s3.AmazonS3;
import com.camel.MicroServiceCamelApplication;
```

```
import com.camel.services.SqsAWSService;

@RunWith(CamelSpringBootRunner.class)
@SpringBootTest(classes = MicroServiceCamelApplication.class)
@TestPropertySource(properties = {"folderPathIn=//target/inbox",
"folderPathOut=//target/outbox", "rangeLines=2"})
@UseAdviceWith
public class RouteBalanceatorTests {
    @Autowired
    private CamelContext context;

    @Autowired
    private ProducerTemplate template;

    @EndpointInject(uri = "mock:s3")
    private MockEndpoint mocks3;

    @MockBean
    AmazonS3 s3Client;

    @MockBean
    private SqsAWSService sqsService;

    @Test
    public void onContextUpReturnS3Client() {
        assertNotNull(s3Client);
    }
```

```
    @BeforeEach
    public void setUp() throws Exception {
        deleteDirectory("target/outbox");
        deleteDirectory("target/inbox");
    }

    private void deleteDirectory(String strPath) {
        File dir = new File(strPath);

        for (File file: dir.listFiles()) {
            if (!file.isDirectory()) {
                if (file.delete()) {
                    System.out.println("file deleted successfully");
                }else {
                    System.out.println("Failed to delete the file");
                }
            }
        }
    }

    @Test
    public void testMoveFile() throws Exception {
        template.sendBodyAndHeader("file://target/inbox", "Hello World",
Exchange.FILE_NAME, "hello.csv");

        Thread.sleep(2000);
```

```
        File target = new File("target/outbox");
        assertTrue("File not moved", target.exists());
    }

    @Test
    public void testRoute() throws Exception {

        FluentProducerTemplate fluentTemplate =
context.createFluentProducerTemplate();
        Exchange exchange = fluentTemplate.withProcessor(e->{
                e.getIn().setHeader(Exchange.FILE_NAME,"helloword.csv");
                e.getIn().setBody("Hello World/n"+
                "Hi linha2/n"+
                "Hi linha 3/n"+
                "Hi linha 4/n");
        }).to("file://target/inbox")
                        .send();

        Object resposta = exchange.getIn().getBody();
        assertNotNull(resposta);
        assertTrue(resposta.toString().toLowerCase().contains("hello world"));
    }

    @Test
    public void WhenSendMessageToSQS_A() throws Exception {
      String resultExpected = "test";
```

```
        when(sqsService.getQueueUrl(Mockito.anyString())).thenReturn("url");
        when(sqsService.sendMessageToQueue(Mockito.anyString(),
Mockito.anyString())).thenReturn(resultExpected);

        FluentProducerTemplate fluentTemplate =
context.createFluentProducerTemplate();
        Exchange exchange = (Exchange) fluentTemplate.withProcessor(e-> {
            e.getIn().setHeader(Exchange.FILE_NAME, "helloworld.csv");
        }).to("direct:a");

        String resultMessage = exchange.getIn().getBody(String.class);
        assertNotNull(resultMessage);
        assertTrue(resultExpected.equalsIgnoreCase(resultMessage));
    }

    @Test
    public void WhenSendMessageToSQS_B() throws Exception {
        String resultExpected = "test";

        when(sqsService.getQueueUrl(Mockito.anyString())).thenReturn("url");
        when(sqsService.sendMessageToQueue(Mockito.anyString(),
Mockito.anyString())).thenReturn(resultExpected);

        FluentProducerTemplate fluentTemplate =
context.createFluentProducerTemplate();
        Exchange exchange = (Exchange) fluentTemplate.withProcessor(e-> {
            e.getIn().setHeader(Exchange.FILE_NAME, "helloworld.csv");
```

```
        }).to("direct:b");

        String resultMessage = exchange.getIn().getBody(String.class);
        assertNotNull(resultMessage);
        assertTrue(resultExpected.equalsIgnoreCase(resultMessage));
    }


    @Test
    public void WhenSendMessageToSQS_C() throws Exception {
        String resultExpected = "test";

        when(sqsService.getQueueUrl(Mockito.anyString())).thenReturn("url");
        when(sqsService.sendMessageToQueue(Mockito.anyString(),
Mockito.anyString())).thenReturn(resultExpected);

        FluentProducerTemplate fluentTemplate =
context.createFluentProducerTemplate();
        Exchange exchange = (Exchange) fluentTemplate.withProcessor(e-> {
            e.getIn().setHeader(Exchange.FILE_NAME, "helloworld.csv");
        }).to("direct:c");

        String resultMessage = exchange.getIn().getBody(String.class);
        assertNotNull(resultMessage);
        assertTrue(resultExpected.equalsIgnoreCase(resultMessage));
    }
}
```

ANEXO IV - SqsAWSServiceTests

```
package com.camel.service;

import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.junit.platform.runner.JUnitPlatform;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.junit.jupiter.MockitoExtension;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.model.GetQueueUrlResult;
import com.amazonaws.services.sqs.model.SendMessageRequest;
import com.amazonaws.services.sqs.model.SendMessageResult;
import com.camel.services.SqsAWSService;

@RunWith(JUnitPlatform.class)
@ExtendWith(MockitoExtension.class)
```

```
public class SqsAWSServiceTest {

@InjectMocks
private SqsAWSService sqsService;

@Mock
private AmazonSQS sqsClient;

@Test
public void returnValidUrl() {
        GetQueueUrlResult getQueueUrlResult = mock(GetQueueUrlResult.class);
        when(getQueueUrlResult.getQueueUrl()).thenReturn("url");

                when(sqsClient.getQueueUrl(Mockito.anyString())).thenReturn(getQueu
        eUrlResult);
        assertEquals("url", sqsService.getQueueUrl("queue1"));
}

@Test
public void returnMessageIdValid()throws Exception {

        SendMessageResult sendMessageResult =
mock(SendMessageResult.class);
        when(sendMessageResult.getMessageId()).thenReturn("messageId");
        when(sqsClient.sendMessage((SendMessageRequest)
Mockito.any())).thenReturn(sendMessageResult);
```

```
        String resultMessage = sqsService.sendMessageToQueue("message",
"queue");
        assertEquals("messageId", resultMessage);
        }
}
```