Universidade do Minho
Escola de Engenharia

Rui Filipe Castro Leite

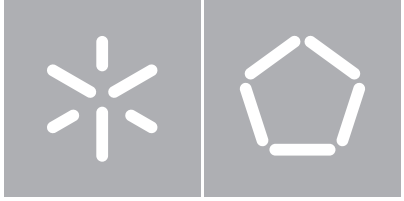
Desenvolvimento de uma plataforma de serviços para facilitar a integração de aplicações empresariais

novembro de 2019

Desenvolvimento de uma plataforma de serviços para facilitar a integração de aplicações empresariais

Rui Filipe Castro Leite

UMinho | 2019



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Rui Filipe Castro Leite

Desenvolvimento de uma plataforma de serviços para facilitar a integração de aplicações empresariais

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor António Luís Pinto Ferreira Sousa
Daniela Duarte da Costa

novembro de 2019

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.
Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

Agradecimentos

Em primeiro lugar, como não poderia deixar de ser, quero prestar um agradecimento especial aos meus pais por me terem fornecido todas as condições que necessitei para a conclusão do meu percurso académico e aos meus irmãos pelo apoio que me deram.

Em segundo lugar, quero deixar um grande agradecimento ao meu orientador, o Professor António Luís Sousa, pelo apoio e acompanhamento que me deu na elaboração deste trabalho e pela oportunidade que me ofereceu.

Agradeço aos meus colegas e diretores da Eurotux Informática pela ajuda e pelo conhecimento que me passaram ao longo do tempo. Um especial obrigado à Sandra por ter estado sempre disponível e interessada em debater ideias e em ajudar a superar alguns dos problemas que encontrei.

Por último, quero prestar um grande agradecimento ao Diogo por toda a ajuda e disponibilidade que me ofereceu e por me ter acompanhado durante todo este percurso.

Resumo

O número de soluções utilizadas pelas empresas para fornecer serviços aos clientes e para fazer a gestão dos processos internos é cada vez maior. Este aumento provocou o aparecimento de novos problemas no que diz respeito à manutenção e à integração de novos serviços, uma vez que grande parte das aplicações não conseguem viver isoladamente.

Com o objetivo de conseguir a integração de diferentes aplicações, surgiu o conceito de *Enterprise Service Bus* (*ESB*) – uma infraestrutura de conectividade que permite a comunicação entre aplicações, que podem ter diferenças a nível das plataformas em que são executadas, das linguagens de programação em que são escritas e dos modelos de dados que utilizam.

A Eurotux Informática, S.A., é uma empresa especialista no planeamento, integração e implementação de sistemas informáticos, onde, devido à existência de diferentes interdependências entre aplicações de apoio ao negócio, surgiu a necessidade de implementar uma solução de integração utilizando um Barramento de Serviços.

Assim, nesta Dissertação de Mestrado, para além do estudo dos padrões de integração de aplicações, é apresentado o processo de planeamento e implementação de uma solução de integração de aplicações num contexto empresarial.

Palavras-chave: Aplicações, EAI, ESB, Integração, SOA

Abstract

The number of solutions that companies use to provide customer service and manage internal processes is growing. This increase has led to new problems in maintaining and integrating new services as most applications cannot live isolated.

In order to achieve the integration of different applications, the concept of *Enterprise Service Bus* emerged – a connectivity infrastructure that allows communication between applications, running on different platforms, written in different programming languages and with different data models.

Eurotux Informática, S.A., is a company specialized in the planning, integration and implementation of computer systems. Due to the existence of different interdependencies between applications that support the company's business, there is a need to implement an integration solution using an *Enterprise Service Bus*.

This Master Thesis aims to study application integration patterns and the planning and implementation of an integration solution in a business context.

Keywords: Applications, EAI, ESB, Integration, SOA

Conteúdo

Lista de Figuras

Acrónimos

A

AMQP Advanced Message Queuing Protocol.
API Application Programming Interface.

B

B2B Business-to-business.
BD Base de Dados.

C

CDM Canonical Data Model.
CPU Central Processing Unit.
CSV Comma-Separated Values.

D

DSL Domain Specific Language.
DSR Design Science Research.

E

EAI Enterprise Application Integration.
ERP Enterprise Resource Planning.
ESB Enterprise Service Bus.
ETL Extract, Transform, Load.

F

FTP File Transfer Protocol.

H

HTTP Hypertext Transfer Protocol.
HTTPS Hyper Text Transfer Protocol Secure.

I

I/O Input/Output.
IDE Integrated Development Environment.
IDL Interface Definition Language.
IP Internet Protocol.
iPaaS Enterprise Integration Platform as a Service.

J

JAR Java ARchive.
JDBC Java Database Connectivity.
JMS Java Message Service.
JMX Java Management Extensions.
JSON JavaScript Object Notation.
JVM Java Virtual Machine.

L

LDAP Lightweight Directory Access Protocol.

M

ML Machine Learning.
MOM Message-Oriented Middleware.
MQTT Message Queuing Telemetry Transport.

N

NIO Non-blocking I/O.

O

OPAIS On-Premises Application Integration Suites.

P

POP3 Post Office Protocol.

R

RAML RESTful API Modeling Language.
REST Representational State Transfer.
RMI Remote Method Invocation.
RPC Remote Procedure Call.

S

SDK Software Development Kit.
SGBD Sistema de Gestão de Base de Dados.
SMTP Simple Mail Transfer Protocol.
SOA Service-Oriented Architecture.
SOAP Simple Object Access Protocol.
SPOF Single Point Of Failure.
SSL Secure Sockets Layer.
SSO Single Sign-On.
STOMP Streaming Text Oriented Messaging Protocol.

T

TCP Transmission Control Protocol.
TLS Transport Layer Security.

U

UDP User Datagram Protocol.
UML Unified Modeling Language.
URI Uniform Resource Identifier.
URL Uniform Resource Locator.

W

WSDL Web Services Description Language.

X

XML Extensible Markup Language.
XPath XML Path Language.
XSL eXtensible Stylesheet Language.

1

Introdução

1.1 Enquadramento

O surgimento de novas tecnologias e a urbanização foram os primeiros passos para o início da constante evolução da indústria, que se tem verificado até aos dias de hoje. O destino dessa evolução ficou voltado para a digitalização, com a automatização de processos industriais, a chegada da Internet e de um grande conjunto de *web-based services*.

A indústria tornou-se cada vez mais dependente de soluções tecnológicas, no sentido de acompanhar a evolução do mercado, de manter a eficiência e a produtividade nos seus processos de negócio e de procurar dar resposta às necessidades dos seus clientes [1].

Também as empresas de desenvolvimento de tecnologia sofreram alterações ao longo do tempo, devido à constante pressão no fornecimento de novos serviços. Com o surgimento de novas necessidades e com o aumento da dimensão das próprias empresas, verificou-se o aumento do número de soluções utilizadas para fornecer serviços aos clientes e também para fazer a gestão dos processos internos.

Este crescimento, de uma forma geral, provocou o aparecimento de novos problemas no que diz respeito à manutenção e à integração de novos serviços. Com isso, tornou-se importante a reutilização e a valorização de sistemas já existentes, bem como a consideração de atributos de qualidade específicos como escalabilidade, modificabilidade e extensibilidade. Para além disso, uma vez que grande parte das aplicações utilizadas por uma empresa não conseguem viver isoladamente, surgiu a necessidade de conseguir que essas aplicações possam interagir entre si, mantendo os atributos de qualidade referidos.

Esta necessidade de integração dos sistemas da empresa resultou também no surgimento de princípios e ferramentas de *middleware* e de plataformas de *Enterprise Application Integration (EAI)*, que culminaram, em meados da década de 90, no aparecimento de conceitos e implementações de *Service-Oriented Architecture (SOA)* [2].

O conceito de SOA assenta numa ideologia em que as aplicações são disponibilizadas como um conjunto de serviços, cada um com uma definição inequívoca da sua responsabilidade, e que procura dar flexibilidade, modularidade e encapsulamento às aplicações.

Nesta arquitetura, um serviço pode ser utilizado por outros serviços ou aplicações – uma relação que foi fundamentada com a definição de protocolos e linguagens *standard*, como *Web Services Description Language (WSDL)*, para uma definição formal das operações que o serviço fornece e dos respetivos parâmetros, e *Simple Object Access Protocol (SOAP)*, como um protocolo de especificação do formato das mensagens [3].

Na perspetiva de conseguir a integração de diferentes aplicações e tendo em consideração os conceitos de SOA e a perceção de que o paradigma de comunicação Ponto a Ponto não se adequa a situações em que existe um conjunto grande de serviços a comunicar entre si, começaram a surgir pela primeira vez, em 2002, pelas mãos de Schulte [4] e de Chappell [5], noções de *Enterprise Service Bus (ESB)*.

Um ESB – também referido como Barramento de Serviços Empresarial ou simplesmente Barramento de Serviços –, de uma forma geral, fornece uma infraestrutura de conectividade que permite a comunicação entre aplicações, executadas em diferentes plataformas, escritas em diferentes linguagens de programação e com diferentes modelos de dados. É uma camada intermédia com uma arquitetura própria que assume do seu lado a responsabilidade de fazer as devidas transformações no formato das mensagens, processá-las e encaminhá-las para um ou mais destinos [2].

A utilização de um ESB permite a redução do esforço e do custo necessário para a integração de diferentes aplicações ou serviços, permite a utilização de protocolos de comunicações distintos, aumenta a abrangência da gama de soluções que a empresa pode utilizar futuramente e permite ainda a definição e reutilização de fluxos de comunicação.

1.2 Motivação

Esta Dissertação de Mestrado tem como propósito o estudo dos requisitos de implementação de uma solução de integração de aplicações num contexto empresarial, a análise de diferentes soluções de ESB existentes e a escolha de uma solução que se enquadre nos requisitos definidos. Com isto, pretende-se que a utilização da solução de ESB escolhida traga benefícios ao processo de integração de aplicações no contexto da empresa Eurotux Informática, S.A., onde o projeto se insere.

No sentido de evitar a complexidade e a falta de fiabilidade que existe na integração de infraestruturas complexas utilizando uma abordagem Ponto a Ponto, pretende-se, como Caso de Estudo, a definição de diferentes cenários de integração e o desenvolvimento de conectores a aplicações em utilização na empresa.

1.3 Objetivos

Face ao enquadramento e à motivação definidos anteriormente, pretende-se, com a elaboração desta Dissertação de Mestrado, atingir os seguintes objetivos:

- Estudar um conjunto de princípios e padrões de integração de aplicações a aplicar num contexto empresarial;
- Identificar e analisar diferentes soluções de ESB existentes no mercado;
- Escolher uma solução de ESB que respeite as necessidades da empresa;
- Identificar, na empresa, as aplicações que necessitam de integração;
- Definir um Caso de Estudo;
- Implementar o Caso de Estudo definido.

1.4 Abordagem metodológica

Estipulados os objetivos desta Dissertação, pretende-se agora discutir a abordagem metodológica a utilizar para garantir a obtenção desses mesmos objetivos.

Existem diferentes abordagens metodológicas que podem ser adotadas no desenvolvimento de projetos com cariz científico. Nesta Dissertação de Mestrado decidiu-se adotar a abordagem *Design Science Research (DSR)* Peffers et al. [6], que constitui um processo rigoroso de definir Artefactos para resolver problemas, avaliar o que foi definido e aplicado e fazer uma discussão dos resultados obtidos.

Na abordagem DSR, os Artefactos podem corresponder a: conceitos de um domínio, que são utilizados para descrever e pensar sobre tarefas; modelos ou proposições, que expressam as relações entre os conceitos de um domínio; ou métodos e respetivos passos para executar uma tarefa [7]. Relativamente à metodologia, segundo Peffers et al. [6], esta é desenvolvida a partir de 6 etapas, nomeadamente:

1. **Identificação do Problema e da Motivação** – Etapa dedicada à definição do problema e da motivação que leva à sua resolução. O Estado da Arte é o recurso necessário para a realização desta etapa.
2. **Definição dos Objetivos** – Tendo como ponto de partida o conhecimento adquirido acerca do problema e do que é viável e fazível, nesta etapa delineiam-se os objetivos da solução a ser desenvolvida. Como recursos necessários para a realização desta etapa tem-se o Estado da Arte e também o conhecimento de possíveis soluções;

3. **Planeamento e desenvolvimento** – Etapa destinada ao desenvolvimento do Artefacto, determinando as funcionalidades e a arquitetura desejada. O conhecimento da teoria que pode ser aplicada numa solução é o recurso necessário para a concretização desta etapa.

4. **Demonstração** – Etapa dedicada à demonstração de utilização do Artefacto desenvolvido, resolvendo uma ou mais instâncias do problema, como Caso de Estudo. Para a concretização desta etapa é necessário possuir o conhecimento efetivo de como utilizar o Artefacto para resolver o problema.

5. **Avaliação** – Nesta etapa deve-se observar e medir quanto o Artefacto satisfaz a resolução do problema, comparando os objetivos definidos com os resultados obtidos. Neste ponto, pode-se voltar às etapas 3 ou 4 para fazer melhorias no Artefacto.

6. **Comunicação** – Etapa onde é feita a divulgação do problema e da relevância da solução encontrada.

1.5 Estrutura do Documento

Tal como já referido, esta dissertação tem como principal objetivo a definição de uma possível solução para a integração de aplicações, num contexto empresarial.

Por forma a apresentar essa solução, decidiu-se seguir uma estrutura onde se começa por abordar os conceitos teóricos relacionados com a integração de aplicações, de uma forma geral, para que depois esses mesmos conceitos sejam identificados e aplicados no contexto prático de um Caso de Estudo.

Desta modo, o presente documento está organizado da seguinte forma:

- **Capítulo 2** – Apresentação da revisão da literatura realizada, nomeadamente dos conceitos de SOA, EAI e ESB.

- **Capítulo 3** – Descrição de todos os componentes de uma solução de ESB, o Mule ESB, e a forma como pode ser utilizado para a integração de aplicações.

- **Capítulo 4** – Definição do Caso de Estudo e descrição de todo o procedimento realizado na aplicação dos conceito teóricos e da solução de ESB na integração de um conjunto concreto de aplicações.

- **Capítulo 5** – Conclusões finais e trabalho futuro.

2

Estado da Arte

Este Capítulo é relativo ao estudo do Estado da Arte e Revisão da Literatura, como definido na abordagem metodológica, onde são referidos os conceitos SOA [8], EAI [9] e ESB [5].

2.1 Service-Oriented Architecture

A expressão "orientado a serviços", apesar de ser utilizada em diferentes contextos, está tipicamente associada a uma noção de separação de responsabilidades. Para a concretização desse objetivo, os princípios ditam que a lógica necessária para resolver um problema de grande dimensão deve ser decomposta em peças menores e relacionadas entre si [3].

As empresas começaram a implementar sistemas de *software* utilizando os princípios de SOA em meados da década de 1990 [2]. No domínio da computação, o termo SOA expressa um paradigma de Arquiteturas de *Software* que define o uso de serviços como suporte aos requisitos dos utilizadores de um sistema em implementação [8].

2.1.1 Conceito de Serviço

Um dos princípios subjacentes a este tipo de arquitetura é o de que os recursos são disponibilizados sob a forma de Serviços independentes, que podem ser acedidos em rede sem conhecimento da implementação interna de cada um.

O termo Serviço está associado a uma combinação de características de *design* bem definidas e pode ser considerado uma parte essencial da arquitetura.

> *"A service is a unit of work done by a service provider to achieve desired end results for a service consumer. Both provider and consumer are roles played by software agents on behalf of their owners." He [10]*

Segundo Krafzig et al. [1], um Serviço tipicamente encapsula um conceito de negócio de alto nível, responsável por executar uma determinada tarefa, que usualmente corresponde a uma atividade de negócio no "mundo real". Esta definição de responsabilidade deve ser o mais clara e definida possível, de forma a que, pelas palavras de Josuttis [8], as pessoas que trabalham no negócio "percebam o que é que o Serviço faz".

Um Serviço, segundo Krafzig et al. [1], é constituído por três principais componentes: o contrato, a *interface* e a implementação.

- **Contrato** – Fornece uma especificação formal do propósito, funcionalidade, restrições e modo de utilização do Serviço. O formato desta especificação pode variar dependendo do tipo de Serviço, mas um dos elementos desta especificação é a definição formal da *interface*, baseada em linguagens como WSDL [11].
- **Interface** – Define como é que um *service provider* executa os pedidos de um *service consumer*, expondo aos clientes toda a funcionalidade implementada pelo Serviço.
- **Implementação** – A implementação física do Serviço fornece tanto Lógica de negócio como Dados. Essencialmente, é a realização técnica que satisfaz o contrato do Serviço. Esta implementação consiste em um ou mais artefactos, como programas, ficheiros de configuração e *Bases de Dados (BDs)*.

Segundo Josuttis [8], perceber a composição de um Serviço e ter a perceção de que um Serviço é essencialmente uma *interface* para as diferentes mensagens que retornam informação e/ou alteram o estado de uma entidade associada, é importante para entender o próprio conceito de SOA.

> *"Essentially, SOA is a software architecture that starts with an interface definition and builds the entire application topology as a topology of interfaces, interface implementations and interface calls. SOA would be better-named 'interface-oriented architecture'." Natis [12]*

2.1.2 Classificação de Serviços

Segundo Josuttis [8], existem diferentes formas de classificar um Serviço. Porém, segundo o autor, pode ser considerada uma classificação fundamental, que segue uma abordagem técnica e que divide os Serviços nas seguintes categorias:

- **Basic services** – Fornecem uma funcionalidade básica, que não faz sentido ser dividida por diferentes Serviços, são conceptualmente sem estado (*stateless*) e fornecem a primeira camada de negócio de um sistema. Além disso, encapsulam aspetos específicos de implementação de modo que cada *consumer* os possa invocar sem conhecer a forma como estão implementados.
  Segundo Josuttis [8], a invocação destes Serviços não deve possibilitar o aparecimento de inconsistências nos dados, devendo seguir os princípios ACID [13]: a invocação ou tem sucesso ou então não tem qualquer efeito no estado do sistema (Atomicidade); depois da invocação, o *backend* mantém-se num estado Consistente; a execução de um pedido num Serviço não pode ser influenciada pela invocação de outro Serviço (Isolamento); por último, quando a invocação de um Serviço tem sucesso, o efeito desta invocação tem que persistir no tempo (Durabilidade).

- **Composed Services** – Correspondem a uma composição de outros Serviços que, por sua vez, podem ser *Basic services* ou *Composed services*), e resultam de um processo designado por orquestração. São principalmente utilizados em situações em que uma operação necessita de interagir com múltiplos componentes de um *backend*.

- **Process Services** – Representam um fluxo de atividades de longa duração e interruptível e, ao contrário dos anteriores, mantêm um estado que perdura ao longo das invocações (são *stateful*). Um exemplo desta categoria é um Serviço de *shopping cart*, em que o estado contém os produtos no carrinho e informação sobre o cliente e é mantido ao longo de várias sessões.

Josuttis [8] refere a existência de outras nomenclaturas propostas por autores como: Erl [14], que utiliza o termo *Business services* em vez de *Basic services* e que aglomerou os *Composed services* e *Process services* numa única categoria (designada por *Process services*); e Krafzig et al. [1], que utilizam o termo *Intermediate services* em vez de *Composed services*.

2.1.3 Atributos de qualidade de um Serviço

Sendo o Serviço uma peça importante da abordagem SOA, é igualmente importante ter conhecimento dos seus principais atributos. Nesse sentido, de seguida são apresentados alguns desses atributos, sendo que, como refere Josuttis [8], um Serviço não tem necessariamente que os respeitar todos.

- **Baixo acoplamento** – Apesar de poderem existir algumas dependências, deve-se procurar implementar Serviços o mais auto-contidos possível, i.e., independentes e autónomos em relação aos restantes. Tal é importante para manter a manutenibilidade dos sistemas e minimizar o impacto das alterações no sistema.

- **Reusabilidade** – Evitar a redundância é geralmente um objetivo a cumprir no desenvolvimento de *software*, sendo que cada funcionalidade de um sistema deve ser implementada apenas uma

vez. O princípio da reusabilidade dá destaque ao posicionamento dos Serviços como recursos empresariais que devem ser reutilizáveis, devendo a lógica ser implementada de forma genérica.

- **Granularidade** – A granularidade refere-se ao grau de modularidade ou à quantidade de funcionalidades de negócio que um Serviço expõe. Serviços com uma granularidade fina possuem funções elementares e lidam com mensagens tipicamente pequenas (estruturas de dados simples), tornando a orquestração mais fácil. Por outro lado, Serviços com uma granularidade grossa correspondem a processos de negócio que lidam com mensagens tipicamente mais complexas. Com estes Serviços, consegue-se diminuir o número de mensagens trocadas entre consumidores e produtores, mas a probabilidade de serem reutilizados diminui.
- **Idempotência** – A idempotência é uma propriedade que se atribui a uma operação quando esta pode ser aplicada mais do que uma vez sem que o resultado se altere. Tal significa que invocações sucessivas de um Serviço, com os mesmos parâmetros, devem produzir o mesmo resultado.
- **Compossibilidade** – Este atributo está relacionado com a reusabilidade de um Serviço e mede a sua capacidade de fazer parte de uma composição.

2.1.4 Vantagens e desvantagens

A utilização de uma abordagem SOA traz benefícios a diferentes níveis, tais como:

- Modificabilidade;
- Reutilização de serviços e quebra de redundância;
- Aumento da produtividade;
- Escalabilidade (com a formação de múltiplas instâncias de um Serviço);
- Fiabilidade (a reduzida dimensão dos Serviços facilita os testes);
- Independência de plataforma;

Apesar disso, uma arquitetura deste tipo pode trazer desvantagens como [15]:

- **Aumento da carga computacional** – Sempre que um Serviço interage com outro, tem que ser feita a validação completa dos parâmetros de entrada, o que aumenta o tempo de resposta e a carga do servidor e reduz o desempenho geral do sistema.
- **Gestão complexa dos Serviços** – Com um número elevado de Serviços, a gestão da comunicação torna-se cada vez mais complexa.
- **Integração de Serviços** – A integração dos Serviços pode torna-se num processo difícil se a equipa de desenvolvimento não possuir conhecimentos de integração de aplicações.

2.2 Enterprise Application Integration

Em geral, as empresas tendem a usar um conjunto diverso de sistemas ou aplicações, nos quais os seus funcionários confiam para conduzir os processos de negócio do dia-a-dia. Uma única empresa pode usar diferentes sistemas ao nível dos vários departamentos, que podem ser desenvolvidos internamente ou licenciados por um fornecedor externo.

Em teoria, esta modularização dos processos de negócio é vantajosa, uma vez que permite dividir toda a gestão da empresa em tarefas menores e também porque facilita a implementação de soluções de suporte mais adequadas a cada área e a rápida adaptação às necessidades de negócio que estão constantemente em mudança [16].

Em muitos dos casos, estas aplicações possuem dependências entre si e necessitam de ser integradas. Por exemplo, os sistemas de gestão empresarial *Enterprise Resource Planning (ERP)*, que têm como objetivo englobar todos os dados e processos de uma empresa, podem necessitar de aglomerar dados com origem em diferentes fontes de informação, como sistemas de faturação, de gestão de projeto, de gestão de propostas, etc. Grande parte destes sistemas não conseguem comunicar de forma nativa uns com os outros, o que por vezes leva à replicação dos mesmos dados por vários locais.

O termo *Enterprise Application Integration (EAI)*, enquanto termo técnico, existe desde o início dos anos 2000 e diz respeito ao processo de estabelecer uma ligação entre diferentes aplicações, com o objetivo de simplificar e automatizar processos de negócio na maior medida possível, eficazmente e evitando, ao mesmo tempo, fazer alterações nas aplicações ou nas estruturas de dados já existentes [16].

Segundo Hohpe and Woolf [17], a integração é o processo de fazer com que aplicações separadas "trabalhem" em cooperação para produzir um conjunto único de funcionalidades.

2.2.1 Desafios na integração de aplicações

A integração de aplicações numa empresa não é uma tarefa simples. Geralmente, neste processo tem-se que lidar com diferentes aplicações que são executadas em diferentes plataformas e com determinadas questões técnicas e específicas do contexto.

Além disso, para que a integração das aplicações seja bem sucedida não basta estabelecer a comunicação entre os vários sistemas de *software*. É necessário que as próprias unidades da empresa tenham a perceção de que as aplicações deixam de ser controladas isoladamente e passam a fazer parte de um fluxo geral de aplicações integradas [10].

Uma restrição importante no processo de desenvolvimento de uma solução de integração de aplicações, do ponto de vista de um programador, é o nível limitado de controlo que tem sobre as aplicações a integrar. Isto porque, em diferentes situações, os sistemas em integração são antigos (*legacy systems*) ou *packaged applications* que podem estar preparados para sofrerem alterações no seu comportamento. Por outro lado, quando se está na presença de uma aplicação de código aberto (*open source*), em que existe

a possibilidade de analisar e modificar o seu código fonte [18], ou de uma aplicação que possua uma componente de extensibilidade que, mesmo não sendo *open source*, permita que o seu comportamento seja modificado, embora o programador possua a liberdade para ajustar as aplicações às necessidades do contexto de integração, em contrapartida passa a ter que manter por ele mesmo o código fonte da aplicação e garantir que o comportamento do sistema se mantém correto com as sucessivas atualizações de versão.

Relativamente a este assunto, os autores Hohpe and Woolf [17] referem que algumas destas aplicações podem ser desenvolvidas internamente enquanto outras são desenvolvidas por fornecedores externos e que também é provável que as aplicações estejam a executar em diferentes computadores, em diferentes plataformas e geograficamente dispersas. Também o facto de algumas das aplicações não terem sido desenhadas para serem integradas nem alteradas, dificulta o processo de integração.

Apesar de o desenvolvimento de uma solução EAI seja um desafio por si só, manter essa solução pode ser igualmente complexo. A combinação de tecnologias torna o *deployment*, a monitorização e a manutenção em tarefas mais complexas [10].

2.2.2 Critérios a considerar na integração de aplicações

Existe um conjunto de critérios e de consequências que devem ser tidos em consideração no processo de integração de aplicações. A seguir estão presentes alguns desses critérios definidos por Hohpe and Woolf [17]:

- **Acoplamento das aplicações** – Mesmo aplicações que tenham passado por um processo de integração devem possuir o mínimo possível de dependências entre si, de forma a que cada uma possa evoluir sem causar problemas às restantes. Normalmente, aplicações com um alto nível de acoplamento possuem uma série de pressupostos sobre como as outras aplicações funcionam. O problema é que, quando uma aplicação é alterada, esses pressupostos são quebrados.

- **Simplicidade** – Segundo o autor, é típico que seja necessário realizar alterações numa aplicação ao longo do tempo e que as abordagens de integração com menos impacto na aplicação podem não ser as mais simples de implementar. No entanto, os programadores devem esforçar-se por minimizar essas alterações.

- **Tecnologia** – Diferentes técnicas de integração requerem diferentes tipos de *software* especializado. Estas ferramentas podem ter um elevado custo e aumentar a carga dos programadores na procura de perceber como é que as podem utilizar para integrar as aplicações.

- **Formato dos dados** – Aplicações integradas devem possuir uma "concordância" no formato dos dados que trocam ou deve existir um "tradutor" intermediário que unifique os diferentes formatos

de dados. Um problema que está relacionado com esta questão é o impacto que a evolução e a extensibilidade dos formatos de dados têm nas aplicações e na solução de integração.

- **Latência na partilha de dados** – A solução de integração deve minimizar o tempo entre o momento em que uma aplicação disponibiliza uma determina informação e o momento em que outra aplicação obtém essa mesma informação. Os dados devem ser trocados com frequência e em blocos pequenos, em vez de se esperar pela troca de um grande conjunto de dados. Segundo o autor, a latência na partilha de dados deve ser levada em consideração na solução de integração, uma vez que quanto mais tempo a partilha demorar, maior é a probabilidade de os dados se tornarem obsoletos e maior é a complexidade da solução.

2.2.3 Cenários de integração

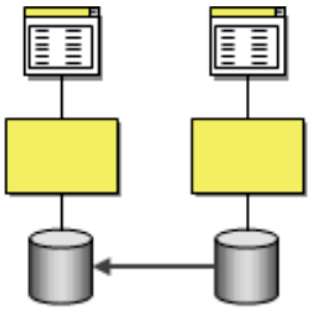
Por forma a estudar a definição de integração, é pertinente fazer uma análise de diferentes cenários onde esta pode ser necessária. Nesse sentido, de seguida apresentam-se cinco padrões de exemplo que representam um cenário de integração concreto (Figuras adotadas de [17]).

Portal informativo

É frequente que um colaborador de uma empresa necessite de consultar mais do que um sistema para executar uma determinada tarefa ligada com o negócio ou para responder a uma questão de um cliente, por exemplo. Em vez de ter que consultar diferentes sistemas, o colaborador poderá utilizar um portal que agregue toda a informação necessária para oferecer uma resposta completa ao cliente [17].

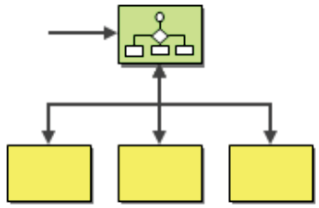
Replicação de dados

Diferentes sistemas podem ter que aceder aos mesmos dados. Por exemplo, a morada de um cliente pode ser utilizada tanto num sistema de apoio ao atendimento como num sistema de faturação, mas ambos os sistemas devem ter as suas próprias BDs. Porém, quando um cliente pretende fazer a alteração dos seus dados pessoais, ambos os sistemas têm de atualizar a sua cópia da morada. Para isto, existem várias soluções: implementar um mecanismos de sincronização de BDs; fazer a exportação e importação de dados através de ficheiros; utilizar um sistema de mensagens [17].



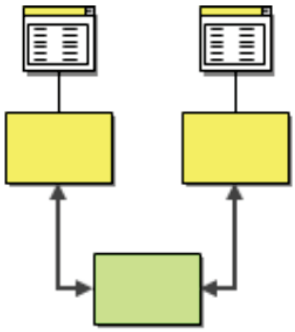
Processo de negócio descentralizado

Uma das principais motivações para a integração de aplicações está relacionada com o facto de que uma única transação (por exemplo, "a compra de um produto") poder estar dispersa por sistemas distintos (por exemplo, um sistema de gestão de *stock*, um sistema de faturação, um sistema de gestão de encomendas a fornecedores, etc). Estando estas funções dispersas, é necessário acrescentar à arquitetura um componente que faça a coordenação destes processos, o que se considera um cenário de integração [17].



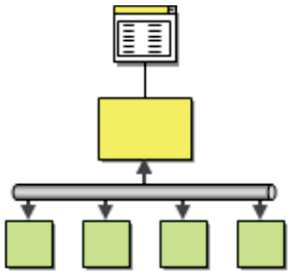
Processo de negócio partilhado

Da mesma forma que podem existir dados redundantes no conjunto de aplicações utilizadas numa empresa, também podem existir implementações redundantes de uma ou mais funcionalidades. Por exemplo, a verificação do nível de *stock* de um produto pode ser uma funcionalidade disponibilizada sob a forma de um serviço a outros sistemas. Esta é também uma solução para o cenário de **Replicação de dados** – poderia ser implementada uma função partilhada "Obter morada do cliente" que fornecesse a morada do cliente a vários sistemas em vez de serem armazenadas cópias redundantes [17].



Arquitetura Orientada a Serviços

Um serviço corresponde a uma entidade de negócio bem definida e universalmente disponível, que responde a pedidos de consumidores. Em primeiro lugar, os serviços dependem uns dos outros e é necessário que haja uma plataforma de serviços (uma espécie de lista de todos os serviços disponíveis) para que possam ser consumidos. Em segundo lugar, cada serviço deve descrever a sua *interface* e a forma como pode ser consumido. A implementação de uma nova aplicação pode ser feita recorrendo a invocações de serviços remotos já existentes [17].



2.2.4 Diferentes abordagens de integração

Segundo Hohpe and Woolf [17], não existe uma abordagem de integração que satisfaça todo o conjunto de vantagens ao mesmo tempo, mas determinadas abordagens podem ser melhores do que outras em

determinados cenários. Assim sendo, os autores referem a existência de quatro principais categorias ou abordagens de integração:

- **Partilha de ficheiros** – Quando as aplicações produzem ficheiros de dados partilhados para que outras aplicações possam consumir.
- **Invocação de procedimentos remotos** – Quando as aplicações expõe algumas das suas operações numa *Application Programming Interface (API)*, de forma a que possam ser invocadas remotamente. Outras aplicações invocam essas operações para realizar determinadas tarefas ou para obter dados.
- **BD partilhada** – Quando as aplicações gravam os dados numa BD de acesso comum, de forma a que a informação seja partilhada.
- **Sistema de Mensagens** – Quando as aplicações partilham dados e invocam operações através da utilização de mensagens.

Segundo os autores, o que diferencia cada um destes padrões dos restantes é a procura por uma implementação mais "elegante", porém os mesmos referem que o objetivo desta análise não é levar à escolha de um padrão que deva ser usado sempre, mas sim escolher um ou mais padrões que se adaptem a uma situação em particular, i.e., aqueles que oferecem mais vantagens para a solução.

Apesar de considerarem que cada padrão possui vantagens e desvantagens, os autores acreditam que um Sistema de Mensagens é a melhor solução para a integração de aplicações, uma vez que:

- Tanto a Partilha de ficheiros como a Partilha de uma BD expõe os dados de uma aplicações, mas não expõe as funcionalidades.
- A utilização de uma BD partilhada e da Invocação de procedimentos remotos levam ao aumento do acoplamento entre as aplicações.
- Um Sistema de Mensagens permite o encapsulamento dos dados, ao contrário da utilização de uma BD partilhada;
- Uma solução de Partilha de ficheiros não produz resultados imediatos.
- Um Sistema de Mensagens reduz a ocorrência de problemas de inconsistência de dados, que podem ocorrer na Partilha de ficheiros.
- Um Sistema de Mensagens permite o processamento assíncrono de mensagens, ao contrário da Invocação de procedimentos remotos.
- Um Sistema de Mensagens fornece mecanismos de tolerância a falhas.

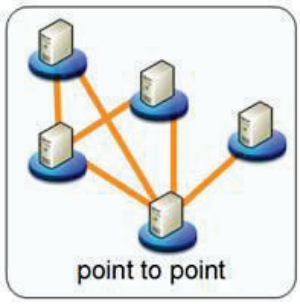
Por ser considerada pela literatura uma melhor solução, nesta Dissertação é explorada a utilização de um Sistema de Mensagens na integração de aplicações.

2.2.5 Principais topologias de integração

Ao longo dos anos foram surgindo diferentes topologias de integração de aplicações, que foram evoluindo na procura de remover as principais desvantagens, como é a existência de interdependências entre aplicações. Nesta Secção, encontram-se três principais topologias (Figuras adaptadas de NeuronESB [19]).

Topologia Ponto a Ponto

Na topologia Ponto a Ponto [20] as soluções de integração são definidas para um par de aplicações de cada vez, tendo que ser implementado um componente responsável pela conexão, i.e., um conector, para cada par de aplicações ou sistemas que necessitam de comunicar. Nesta abordagem, o conector é responsável por controlar a transformação dos dados e as operações necessárias para a comunicação entre as duas aplicações específicas.



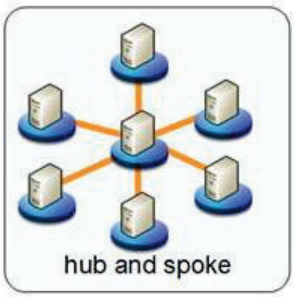
Quando usado em infraestruturas pequenas, por exemplo com dois sistemas, este modelo poder-se-à ajustar às necessidades da infraestrutura, fornecendo uma solução de integração de baixo custo. Porém, à medida que forem adicionados novos componentes, o número de conexões Ponto a Ponto necessárias para construir uma arquitetura em integração aumenta exponencialmente [21] e torna-se naquilo a que se chama "topologia esparguete" (uma alusão ao equivalente em programação a "código esparguete").

O número de ligações necessárias (ou de conectores) para que se tenha uma topologia Ponto a Ponto com n aplicações completamente interligadas é dado por: $n \times (n-1)/2$. Por exemplo, para se conseguir ter 8 aplicações totalmente integradas com este tipo de topologia, são necessárias $8 \times 7/2 = 28$ ligações [21, 22].

De realçar que cada um destes conectores tem que ser desenvolvido e mantido separadamente, tendo em conta as alterações de versão dos sistemas específicos. Com isto, a topologia de integração Ponto a Ponto facilmente se torna desajustada aos cenários de integração num contexto empresarial.

Topologia Hub & Spoke

As primeiras soluções de EAI a surgir no mercado seguiram a ideia de, literalmente, unificar a integração das aplicações e de incorporar toda a funcionalidade num ponto central.



Na topologia *Hub & Spoke* [20] existe um *Hub* ao qual todas as aplicações se ligam e que é responsável pela transformação de mensagens, pelo encaminhamento e qualquer outra funcionalidade necessária à integração das aplicações.

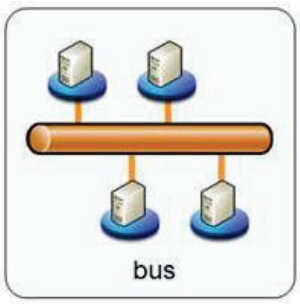
Cada uma das aplicações ligam-se ao ponto central através de um conector (*Spoke*). Segundo Mason [23], normalmente, o ponto central lida diretamente com o formato de dados das aplicações, não existindo um Modelo Canónico[1].

O principal problema que está associado à topologia Ponto a Ponto fica reduzido com a adoção deste modelo. Em vez de a integração ser para um par de aplicações, com este modelo passa-se a desenvolver um único conector entre cada aplicação e o ponto centrak, diminuindo as interdependências entre as aplicações [16].

Este modelo possui algumas desvantagens, principalmente relacionados com a sua "centralização". Em primeiro, se o *Hub* falhar, toda a topologia de integração também falha, o que constitui um *Single Point Of Failure (SPOF)*. E em segundo, por ser um único ponto de entrada de todas as mensagens, o ponto central pode tornar-se num ponto de congestão (*bottleneck*) [20, 22, 25].

Mason [23], fundador da MuleSoft, afirma que esta topologia é adequada para se iniciar um processo de integração, como uma prova de conceito, ou para ser utilizada em projetos de integração pequenos. Já os autores Rahimsoomro and Awan [26] referem que muitos dos projetos que adotaram o modelo *Hub & Spoke* não foram bem sucedidos pela falta de *standards* e pelo elevado custo monetário das soluções no mercado. Em 2003, um estudo estimou que 70% dos projetos de integração não tiveram sucesso devido a estas falhas nas primeiras soluções de EAI.

Barramento de Mensagens



Na tentativa de ultrapassar os problemas causados pela topologia *Hub & Spoke*, surgiu um novo modelo de EAI – o Barramento de Mensagens (ou *Message Bus*) [20].

Este modelo procura diminuir a carga de responsabilidade colocada num único componente, distribuindo-a por outros subcomponentes da arquitetura [16, 26]. Estes componentes podem ser agrupados em diferentes composições por forma a lidarem com qualquer cenário de integração, na procura de uma implementação mais eficiente. Além disso, o modelo foi pensado numa perspetiva de escalabilidade, de forma a que os componentes possam ser replicados na medida do necessário.

O Barramento de Mensagens fornece uma infraestrutura de comunicação comum entre as aplicações, que atua de forma independente de plataforma e de linguagem. Cada aplicação interage com o Barramento utilizando um conector, que é responsável pelas transformações da mensagem para um formato utilizado pela respetiva aplicação.

Podem ser identificados neste modelo outras funcionalidades, como o processamento de transações, o controlo de erros e a possibilidade de conceção de uma solução com pouco desenvolvimento e sem alterações às aplicações em integração, tornado-o adequado a cenários empresariais [26].

1 Um Modelo Canónico é um padrão utilizado para comunicar entre diferentes formatos de dados. Essencialmente, é criado um modelo de dados *standard* que é um superconjunto de todos os outros ("canónico") e que define o conteúdo e estrutura de uma mensagem [24].

A principal distinção entre este modelo e o modelo *Hub & Spoke* é que o mecanismo de integração que trata da transformação das mensagem e do encaminhamento é distribuído pelos conectores das aplicações e não centralizado no *Hub* [20].

2.2.6 Sistema de Mensagens

Um Sistema de Mensagens (ou *Messaging System*) é uma tecnologia de *middleware* – *Message-Oriented Middleware (MOM)* – que permite a comunicação assíncrona entre aplicações com entrega fiável. As aplicações comunicam através do envio de pacotes de dados designados por "Mensagens" [27].

Os Canais de Mensagens, também conhecidos por *Channels* ou *Queues*, são caminhos lógicos que conectam um Remetente a um ou mais Recetor e que transmitem as mensagens entre eles. Um Canal pode ser visto como uma coleção partilhada de Mensagens.

O Emissor ou Produtor corresponde à entidade que envia Mensagens através do Canal. Enquanto que o Recetor ou Consumidor (*Receiver* ou *Consumer*) corresponde à entidade que recebe e lê as mensagens de um Canal. Ao longo desta Secção é utilizado o termo "entidades" para referir tanto a Emissores como Recetores.

Uma Mensagem pode ser considerada, de forma simplificada, uma estrutura de dados, tal como uma *String*, um Vetor ou um Objeto, e pode ser interpretada como a representação de um comando a ser invocado no respetivo Recetor ou como a descrição de um evento que ocorreu no próprio Emissor.

Tecnicamente, uma Mensagem é composta por duas partes: o cabeçalho, que contém meta-informação sobre a mensagem, e o corpo, que contém os dados transmitidos.

O Sistema de Mensagens é um sistema de *software* autónomo designado que coordena o envio e receção das Mensagens, na medida em que deve garantir que mesmas são efetivamente enviadas e entregues ao Recetor.

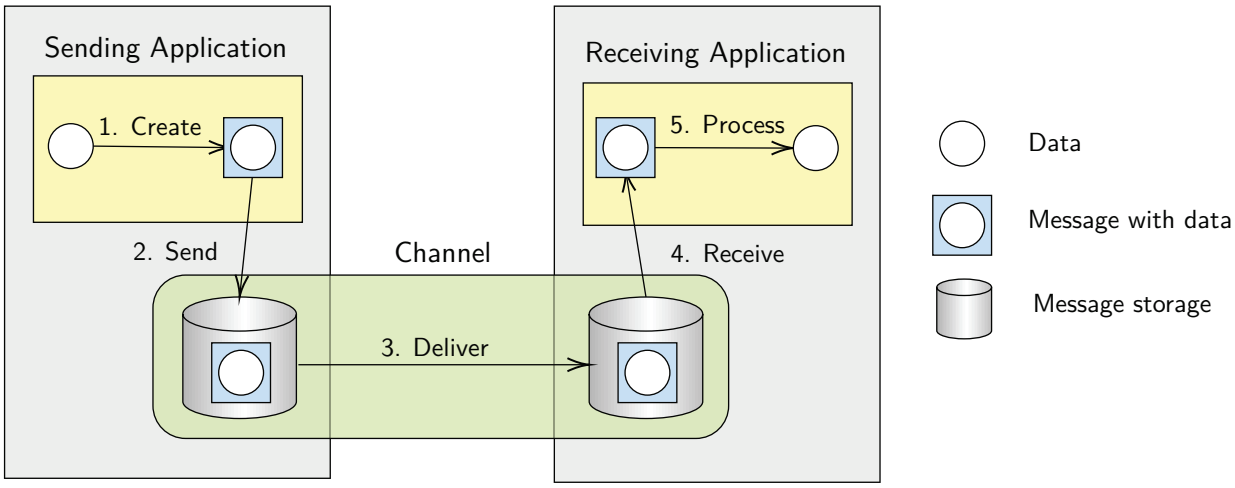


Figura 1: Transmissão de uma mensagem (**Adaptado de:** *Hohpe and Woolf [17]*)

Para a garantia de entrega, o sistema tem que saber lidar com algumas situações excecionais, como quando o Recetor não está pronto para receber uma Mensagem ou o quando existem problemas a nível da rede. O Sistema de Mensagens tem a responsabilidade de superar estas limitações e de tentar transmitir a Mensagem até obter sucesso.

Teoricamente, tal como se pode ver representado na Figura 1, uma mensagem é transmitida em 5 passos:

1. **Instanciação** – O Emissor cria a Mensagem com o devido conteúdo.
2. **Envio** – O Emissor submete a Mensagem para um canal.
3. **Entrega** – O Sistema de Mensagens entrega a Mensagem ao Recetor, deixando-a disponível para ser consumida.
4. **Receção** – O Recetor obtém e lê a mensagem do Canal.
5. **Processamento** – O Recetor extrai os dados presentes no corpo da Mensagem.

Para além destes passos, neste diagrama estão também ilustrados dois conceitos importantes: o *Send and Forget* e o *Store and Forward*.

- **Send and Forget** – No passo 2, o Emissor envia a mensagem para o Canal mas não fica à "espera" que a mesma seja entregue ao Recetor. Deste modo, o Sistema de Mensagens pode transmitir a Mensagem em segundo plano;
- **Store and Forward** – No passo 2, quando o Remetente envia a mensagem para o Canal, o Sistema de Mensagens armazena a Mensagem em disco ou em memória. No passo 3, o sistema entrega a Mensagem ao Recetor e, em seguida, armazena-a novamente. Este processo de *Store and Forward* é importante para conseguir a entrega fiável e garantida das Mensagens.

Vantagens de utilização

Segundo Hohpe and Woolf [17], existe um conjunto diverso de vantagens na utilização de um Sistema de Mensagens para estabelecer a comunicação entre duas entidades:

- **Comunicação remota** – O Sistema de Mensagens permite que entidades distintas possam comunicar e transferir objetos entre si, desde que os mesmos sejam "*serializable*", i.e., que possam ser convertidos para uma simples *Stream* de *Bytes*, que pode ser enviada através da rede. Porém, o autor salienta que o nível dificuldade de enviar dados para uma aplicação remota é maior quando comparado com uma situação em que os objetos residem no mesmo processo. Segundo o autor, caso a comunicação remota não seja um requisito da solução, o Sistema de Mensagens não é necessário – uma simples solução com estruturas de dados concorrentes ou memória partilhada é suficiente.

- **Comunicação assíncrona** –Tal como referido anteriormente, o Sistema de Mensagens permite a adoção de uma abordagem *Send and Forget* na comunicação entre duas ou mais entidades. O Emissor não tem de "esperar" que o sistema envie a mensagem nem que o Recetor a receba e processe. Tem apenas que aguardar que a mensagem seja introduzida no Canal e está livre para continuar a executar outra tarefa. Caso o Recetor proceda com o envio de uma mensagem de confirmação, esse envio terá que ser detetado e processado por um mecanismo de *callback* no Emissor. A comunicação assíncrona tem a vantagem, ao contrário da comunicação síncrona, de permitir que as aplicações sejam executadas no seu rendimento máximo e que não desperdicem tempo à espera de uma resposta a um pedido previamente feito.

- **Independência de linguagem ou plataforma** – Ao estabelecer uma conexão entre duas aplicações remotas, a linguagem, a tecnologia e a plataforma que as aplicações utilizam são possivelmente diferentes. A utilização de um Sistema de Mensagens permite que a solução implementada seja independente destas questões.

- **Garantia de entrega** – Um Sistema de Mensagens permite uma comunicação fiável entre as entidades, através de mecanismos como o *Store and Forward*. As Mensagens, enquanto unidades atómicas e independentes, são armazenadas em disco ou em memória, tanto no Emissor como no Recetor. Esta abordagem permite que quer uma entidade como a outra não tenham que lidar com determinados problemas da comunicação remota, como falhas na rede.

- **Operações offline** – Algumas aplicações foram especificamente concebidas para executar em modo *offline* e para iniciarem um processo de sincronização com os servidores quando uma ligação à Internet estiver disponível. Segundo Hohpe and Woolf [17], um Sistema de Mensagens é o ideal para permitir esta sincronização, uma vez que os dados podem ser adicionados a um Canal à medida que são criados e o próprio Sistema de Mensagens trata de os enviar quando existir uma ligação à rede.

Desafios na utilização

Segundo Hohpe and Woolf [17], é verdade que um Sistema de Mensagens resolve alguns dos desafios da integração de sistemas, mas também é verdade que traz novos. Alguns deles estão inerentes ao modelo assíncrono do envio de Mensagens, enquanto que outros dependem de questões específicas de implementação.

- **Modelo de programação complexo** – A assincronia no envio e receção das Mensagens leva a que os programadores tenham que pensar as suas soluções de uma forma orientada a eventos (*Event-driven programming* [28]). A lógica da aplicação passa a consistir num fluxo de eventos, com origem noutras aplicações. Segundo o autor, este tipo de sistema é mais complexo e mais difícil de desenvolver e de testar.

- **Ordem de entrega** – Tal como já referido, uma das vantagens da utilização de um Sistema de Mensagens é a garantia de entrega. Porém, este sistema não garante quando é que as Mensagens são entregues ao Recetor, o que pode fazer com que sejam processadas por uma ordem distinta daquela com que foram enviadas. Em situações em que a ordem de processamento das Mensagens é importante, tem que ser utilizado um mecanismo de sequenciação.

- **Desempenho** – Os Sistemas de Mensagens adicionam alguma carga à comunicação, nomeadamente na criação de uma entidade Mensagem e no envio da mesma. O autor refere que se for necessário transferir uma grande quantidade de dados, dividir esses dados num conjunto grande de pacotes, cada um com um tamanho reduzido, pode não ser uma boa opção. Por exemplo, se uma solução de integração envolver a sincronização de uma grande quantidade de dados entre dois sistemas, Hohpe and Woolf [17] refere que as ferramentas de *Extract, Transform, Load (ETL)* podem ser mais eficientes do que os Sistemas de Mensagens, sendo estes mais adequados para manter a sincronização a funcionar após uma replicação inicial.

2.2.7 Elementos básicos de uma solução de integração

Por forma a conseguir a criação de uma camada intermédia que integre dois ou mais sistemas, há um conjunto de passos a tomar.

De uma forma geral, é necessário permitir que sejam transportados dados desde um sistema para outro. Esses dados podem corresponder, por exemplo, ao registo da morada de um cliente que precisa de ser replicado para outro sistema ou a uma invocação a um serviço remoto. O que é importante é que o corpo de uma Mensagem seja entendido por ambas as partes, independentemente de qual seja, e que a Mensagem possa ser transferida através da rede.

Tendo por base os princípios de um Sistema de Mensagens apresentados na Secção 2.2.6, existem dois elementos básicos que permitem a comunicação de Mensagens. Em primeiro lugar, é necessário um Canal de comunicação para transportar dados de uma aplicação para outra. Este Canal pode corresponder a um conjunto de ligações *TCP/IP*, a um ficheiro partilhado, a uma Base de Dados partilhada, etc.

Dentro do Canal de comunicação é inserida uma Mensagem, correspondente a um fragmento de dados com um significado comum entre as aplicações. Esse fragmento pode ter uma dimensão reduzida, como a morada de um cliente , ou uma grande dimensão, como a lista completa dos clientes.

Sendo possível o envio da Mensagem através do Canal de Comunicação, pode-se começar a pensar numa solução básica para a integração das aplicações. Porém, tem que se ter em atenção alguns dos desafios que surgem nesse processo.

Um desses desafios é a falta de controlo sobre o modelo de dados que é utilizado pelas aplicações. Por exemplo, uma aplicação pode usar uma estrutura com dois campos para armazenar o nome de um cliente – `FIRST_NAME` e `LAST_NAME` – enquanto que outra aplicação pode usar um único campo –

`CUSTOMER_NAME`. Ou então, uma aplicação no seu modelo de dados pode permitir que um cliente possua mais do que uma morada, enquanto que o modelo de dados utilizado por outra aplicação pode considerar que um cliente possui apenas uma morada.

Dado que em grande parte das situações não é possível alterar o modelo de dados utilizado, há a necessidade de existir na camada intermédia responsável pela integração um mecanismo de conversão entre um modelo de dados e outro. Este passo é chamado de Transformação [17] .

Além do problema da compatibilidade entre modelos de dados, há um outro desafio a ter em mente. Numa situação em que existam diversos sistemas integrados, em que ponto da arquitetura deve ser definido quais são os Recetores de cada Mensagem, i.e., para que sistemas devem ser enviados os dados? Uma possível solução é cada sistema especificar o(s) sistema(s) Recetores. Porém, essa não é a melhor abordagem pelo facto de exigir que cada sistema tenha conhecimento da existência dos restantes e assim contrariar o critério do Acoplamento entre sistemas, defino na Secção 2.2.2. Uma outra solução é passar a responsabilidade para a camada intermédia e assim eliminar as interdependências entre os sistemas. Este passo é chamado de Encaminhamento das Mensagens [17].

Por terem que lidar com múltiplos sistemas, cada um com um modelo de dados distinto, a complexidade das soluções de integração de aplicações facilmente aumenta. Além disso, existe um outro aspeto importante, apontado por Hohpe and Woolf [17]: a maioria das aplicações não estão preparadas para participar de forma nativa numa solução de integração, sendo necessária a existência de um componente adicional que interligue a aplicação em integração à camada intermédia – um Conector.

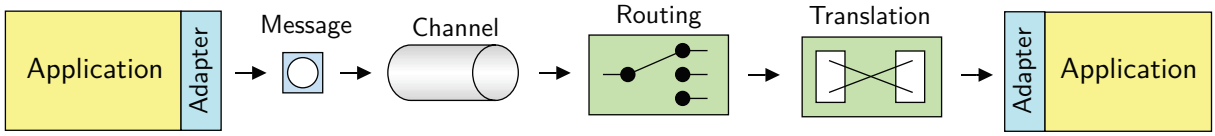


Figura 2: Elementos básicos de uma solução de integração (**Adaptado de:** *Hohpe and Woolf [17]*)

Em suma, pode-se então definir os seguintes principais conceitos numa solução de integração, tendo por base os princípios de um Sistema de Mensagens:

- **Canal de comunicação** – Meio por onde o qual é feita a troca de informação.
- **Mensagem** – Pacote de dados atómico transmitido por um Canal de comunicação.
- **Multi-step delivery** – Em diferentes situações é necessário executar algum tipo de processamento sobre cada uma das Mensagens, quer seja antes ou depois de serem entregues ao Recetor. Estes procedimentos estão associados aos princípios dos padrões de *Pipes and Filters* [29].
- **Encaminhamento** – Idealmente, o Emissor de uma Mensagem não se deve preocupar com a entrega ao(s) Recetor(es). A camada intermédia deve possuir um componente de Encaminhamento que trate desse processo.

- **Transformação** – Diferentes aplicações podem usar diferentes modelos de dados para representar o mesmo conceito. Para acomodar estas situações, a Mensagem tem que passar por um componente intermédio de Transformação.

- **Conector** – Uma aplicação pode não ter a capacidade nativa de utilizar o Canal de comunicação, pelo que deve ser possível a implementação de um componente que estabeleça uma "ponte" entre a aplicação e a camada intermédia. Este componente deve ter conhecimento da forma como a aplicação lida com pedidos remotos.

Na Figura 3 encontra-se representada a integração de duas aplicações: é utilizado um Canal de comunicação para o envio de uma Mensagem entre uma Aplicação A e uma Aplicação B, cada uma com um modelo de dados distinto. É possível encontrar não só os Conectores de cada aplicação, como também a utilização de um componente de Transformação de Mensagens.

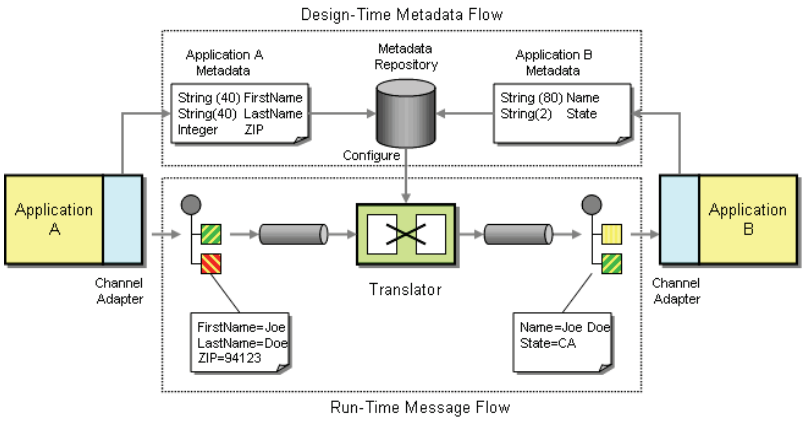


Figura 3: Exemplo de integração entre duas aplicações (**Adotado de:** *Hohpe and Woolf [17]*)

2.3 Enterprise Service Bus

Inicialmente, os sistemas de *software* que seguiam os princípios da abordagem SOA foram planeados para usarem um intermediário central de integração, como o modelo *Hub and Spoke* (Secção 2.2.5). Mas, com o crescimento das empresas e com a adoção de mais aplicações e serviços, foi identificada a necessidade de um novo modelo de infraestrutura que combine os princípios de MOM [27] e de Transformação e Encaminhamento de Mensagens como componente principal.

Surgiu então um novo modelo de integração baseado num Barramento de Serviços (ESB) e com ele apareceram várias soluções disponibilizadas por reconhecidas empresas e organizações, tais como:

- A IBM, com o IBM Integration Bus, versão 10.0 lançada em 2014 [30].

- A Oracle, com o Oracle Service Bus, versão 12.2 lançada em junho de 2018 [31].

- A Microsoft, com o Microsoft BizTalk ESB, um *toolkit* do Microsoft BizTalk Server cuja última versão foi lançada em 2016 [32].
- A MuleSoft, com o Mule ESB, versão 4.2 lançada em julho de 2019 [16].
- a WSO2, com o Enterprise Integrator, versão 7.0 lançada em outubro de 2019 [33].
- a Red Hat, com o Red Hat Fuse, versão 7.4 lançada em setembro de 2019 [34].

No seguimento daquilo que foi apresentado nas Secções anteriores sobre EAI e SOA, nesta Secção pretende-se especificar noções importantes sobre a arquitetura de um ESB e sobre o que se pode esperar de uma solução que implemente essa arquitetura.

2.3.1 Definição

Segundo Menge [35], é difícil encontrar uma definição concreta para "o que é um Enterprise Service Bus", isto porque, segundo o autor, não existe um consenso geral para uma definição comum sobre o termo, mas refere que o mesmo foi criado por analistas da Gartner em 2002 [4].

Um ESB é uma infraestrutura de integração distribuída baseada em SOA, em Sistemas de Mensagens e em padrões de integração e atua como uma camada intermédia através da qual um conjunto de *endpoints* e aplicações é disponibilizado [35].

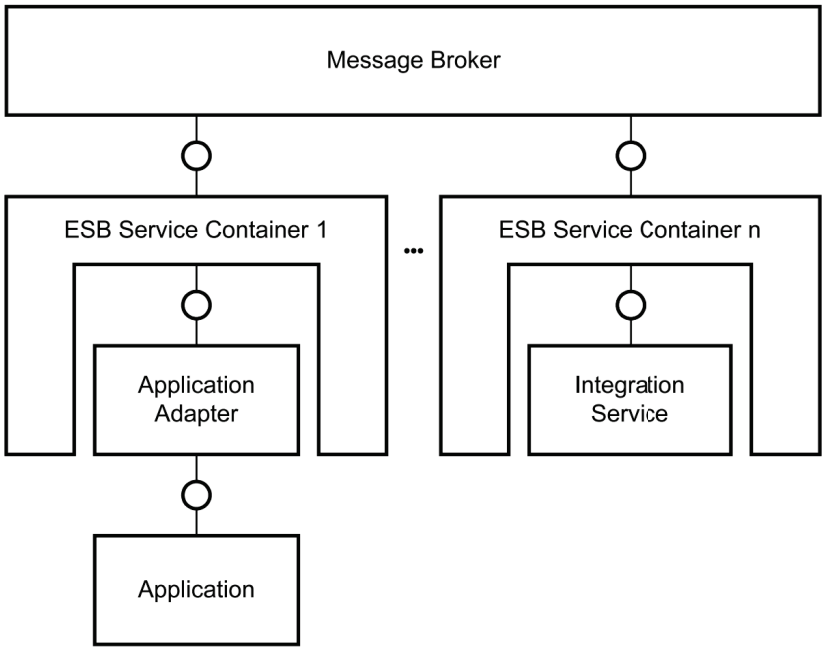


Figura 4: Composição simples de um ESB (**Adotado de:** *Menge [35]*)

Segundo De Leusse et al. [36], um ESB fornece uma camada de abstração que atua como ponto de entrada para um Barramento. Quando as Mensagens chegam a esse ponto de entrada, há uma série

de ações que podem ocorrer dentro do Barramento. Essas ações são utilizadas tendo em conta diversos fatores como o conteúdo da mensagem, a origem ou o destino. Menge [35] refere que um ESB fornece componentes de Encaminhamento, Invocação e Mediação para facilitar a interação, de forma segura e confiável, entre as diferentes aplicações, que podem estar em execução em diferentes plataformas e escritas em diferentes linguagens.

As soluções de ESB são geralmente compostas por *service containers* (Figura 4). Cada um dos *service containers* encapsula serviços de integração como serviços de Encaminhamento, serviços de Transformação de Mensagens, Conectores para uma aplicação externa ou *MOM bridges*. As aplicações são conectadas ao Barramento recorrendo ao respetivo Conector ou então a um dos mecanismos de *messaging* suportados pelo próprio ESB [35].

O Barramento de Serviços deve coordenar as interações entre os vários recursos e fornecer suporte transacional. Além disso, os componentes fornecidos pelo Barramento devem ser componentes genéricos de forma a que possam ser configurados e adequados a diferentes cenários.

2.3.2 Padrões de EAI no contexto de um ESB

Nesta Secção pretende-se fazer uma descrição das funcionalidades base de um ESB, que estão fortemente relacionadas com alguns dos padrões de integração apresentados ao longo das Secções 2.2.6 e 2.2.7, referidos por Hohpe and Woolf [17] e por Menge [35].

Algumas destas características podem ser classificadas em duas categorias: características base, geralmente oferecidas pelas soluções de ESB, e características adicionais.

Invocação

A Invocação é a capacidade de um ESB fazer pedidos e obter respostas de serviços e recursos integrados. Os autores Menge [35] e De Leusse et al. [36] referem que um ESB deve fornecer um barramento de comunicação através do qual possam ser feitas todas as trocas de Mensagens e deve suportar os padrões de comunicação de serviços, como SOAP [37], WSDL [38] e a API *JMS* [39], para integração com sistemas MOM. Além disso, o ESB deve ser capaz de lidar com protocolos como *TCP* [40], *UDP* [41], *HTTP* [42] e *TLS* [43] e também com alguns mecanismos de comunicação adicionais como *RMI* [44], *RPC* [45], *JDBC* [46], *SMTP* [47], *POP3* [48] e *FTP* [49].

Encaminhamento

Assim como já falado na Secção 2.2.7 no âmbito dos Sistemas de Mensagens, o Encaminhamento é a capacidade de decidir o destino de uma Mensagem durante o seu envio. Os serviços de Encaminhamento constituem um recurso essencial de um ESB, uma vez que permitem separar o Emissor de uma Mensa-

gem do(s) respetivo(s) Recetor(es), i.e., permite que o sistema que envia a Mensagem tenha completo desconhecimento do Recetor, ficando a responsabilidade do lado da camada intermédia.

O Encaminhamento de uma Mensagem pode ser feito de diferentes formas. De seguida apresentam-se os principais serviços de Encaminhamento que devem ser oferecidos por um ESB.

- **Recipient list Router** – Permite o envio de uma Mensagem para diferentes Recetores ao mesmo tempo (Figura 5).

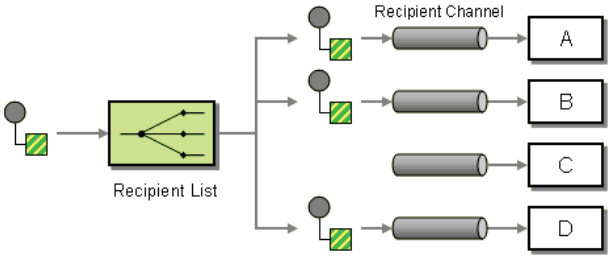


Figura 5: *Recipient list Router* (**Adotado de:** *Hohpe and Woolf [17]*)

- **Content-based Router** – O destino da Mensagem é decidido através do teste de uma determinada condição aplicada ao seu conteúdo, por exemplo, pelo valor de um determinado campo (Figura 6).

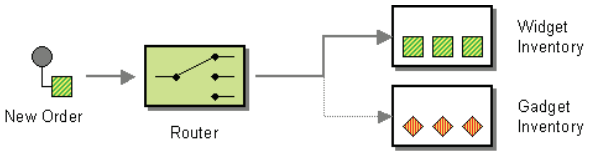


Figura 6: *Content-based Router* (**Adotado de:** *Hohpe and Woolf [17]*)

- **Filtro de Mensagens** – Existindo apenas um possível Recetor, um Filtro decide se envia ou não a Mensagem, tendo em conta um determinado critério [17].

- **Scatter-Gather** – Este componente é utilizado quando se pretende enviar uma Mensagem para vários Recetores ao mesmo tempo e agregar todas as respostas numa única Mensagem, através de um Agregador (Figura 7).

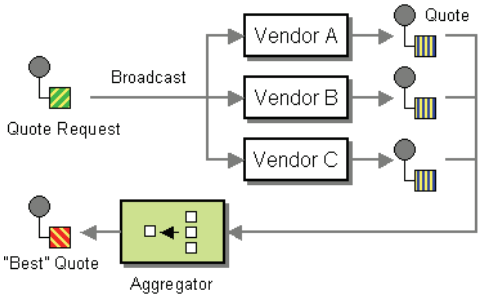


Figura 7: *Scatter-Gather* (**Adotado de:** *Hohpe and Woolf [17]*)

- **Splitter** – Utilizado nas situações em que uma Mensagem é composta por várias partes e em que se pretende dividi-la em várias Mensagens, de forma a poderem serem processadas e encaminhadas individualmente (Figura 8).

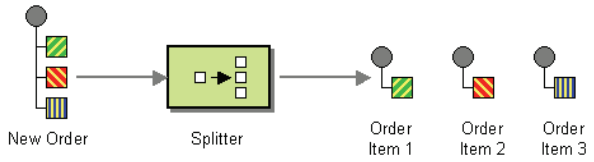


Figura 8: *Splitter* (**Adotado de:** *Hohpe and Woolf [17]*)

- **Agregador** – Ao contrário do *Splitter*, permite a agregação de Mensagens individuais numa única Mensagem, que pode ser encaminhada de uma só vez.

- **Resequencer** – Utilizado em situações em que as Mensagens possuem um determinado número de sequência e em que se pretende que sejam encaminhadas de forma ordenada (Figura 9).

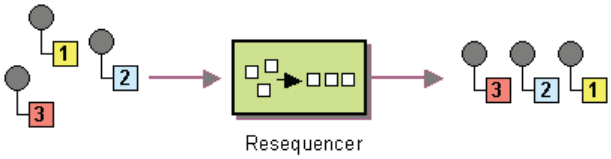


Figura 9: *Resequencer* (**Adotado de:** *Hohpe and Woolf [17]*)

Transformação de Mensagens

Segundo Menge [35], De Leusse et al. [36], a mediação refere-se à capacidade de Transformar Mensagens ou parte delas em diferentes formatos. Tal inclui a Transformação do modelo de dados utilizado por uma aplicação para outro qualquer (Figura 10).

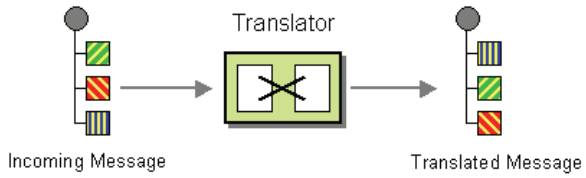


Figura 10: *Message Translator* (**Adotado de:** *Hohpe and Woolf [17]*)

A Transformação de mensagens pode ser associada a diferentes padrões de integração, como:

- **Content Enricher** – Utiliza a informação contida na Mensagem de entrada (por exemplo, um ID) para obter dados de uma fonte externa e incluir esses novos dados na Mensagem de saída.

- **Content Filter** – Ao contrário do anterior, o Filtro de conteúdo tem como objetivo simplificar uma Mensagem, removendo informação irrelevante ou fazendo alguma agregação de informação.

- **Uniformizador** – Tem o objetivo de uniformizar o formato das Mensagens, i.e., consoante a Mensagem de entrada, o componente aplica uma determinada Transformação que faz com que todas as Mensagens de saída tenham o mesmo formato.
- **Modelo de dados canónico** – Padrão aplicado nas situações em que se está na presença de diferentes modelos de dados e em que o objetivo é a definição de um modelo de dados genérico (canónico). Quando é introduzido um novo modelo é apenas necessária a Transformação do novo modelo para o modelo canónico.

Conectores

Algumas das soluções de ESB existentes fornecem um conjunto amplo de Conectores para aplicações externas. Cada um dos Conectores estabelece uma conexão com a *interface* nativa da aplicação e expõe um conjunto de funcionalidades que interagem com a mesma (Figura 11).

Apesar de serem disponibilizados um conjunto de Conectores "prontos a usar", as soluções de ESB tipicamente permitem que sejam desenvolvidos novos Conectores para aplicações específicas.

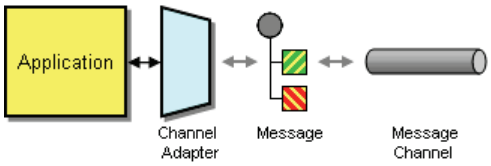


Figura 11: Conector (**Adotado de:** *Hohpe and Woolf [17]*)

Segurança

Uma solução de integração que seja implementada num contexto empresarial deve fornecer um mecanismo de segurança para a troca de Mensagens. Tal significa que o ESB deve permitir que o conteúdo das Mensagens seja cifrado e decifrado, bem como permitir a autenticação e controlo de acesso aos *endpoints*.

Gestão e Monitorização

As soluções de ESB podem fornecer recursos de monitorização da solução, como a consulta de forma centralizada dos *logs* gerados pelos fluxos de Mensagens. Adicionalmente, podem ser disponibilizadas ferramentas de configuração do próprio Barramento e da execução dos processos.

Ferramenta de desenvolvimento

Os autores Hohpe and Woolf [17] salientam que deve ser disponibilizada uma ferramenta específica para o desenvolvimento, teste e *deploy* dos Conectores.

2.3.3 Análise de Soluções de ESB existentes

No sentido de perceber as funcionalidades típicas de um ESB e de que forma os padrões de integração encontrados na teoria de EAI foram implementados, decidiu-se fazer um levantamento das soluções existentes no mercado. Para tal, recorreu-se à análise de alguns dos Quadrantes Mágicos da consultora Gartner.

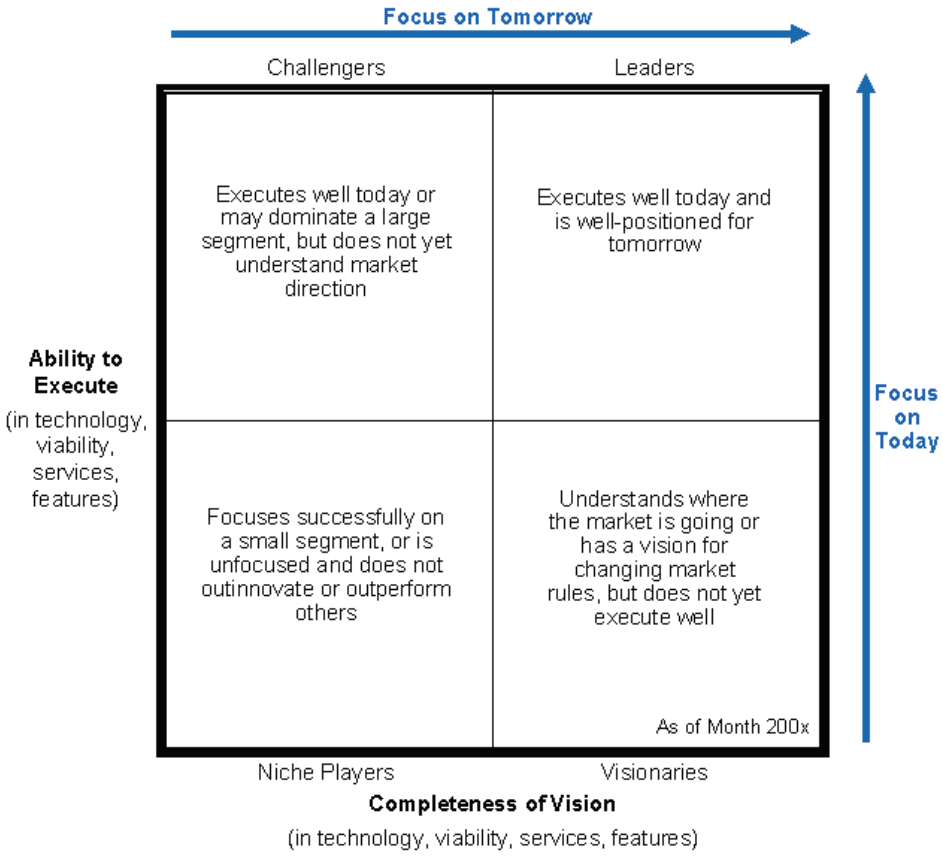


Figura 12: Composição de um Quadrante Mágico (**Adotado de:** *Smulders [50]*)

Um Quadrante Mágico da Gartner oferece uma visão do mercado e das posições relativas dos principais intervenientes numa determinada área, através de um formato gráfico.

A Gartner classifica os fornecedores, empresas e produtos de tecnologia segundo dois critérios: Completude de Visão e Capacidade de Execução. Através de um metodologia própria, essa classificação leva à definição de uma posição num dos quatro quadrantes [50] (Figura 12):

- **Líderes** – Os intervenientes neste quadrante têm as pontuações mais altas para a Completude da Visão e Capacidade de Execução, o que significa que possuem a participação no mercado, a credibilidade e os recursos de *marketing* e de venda próprios do mercado, demonstrando um claro entendimento das necessidades.

- **Desafiantes** – Um interveniente neste quadrante participa no mercado bem o suficiente para ser uma "ameaça" aos seus concorrentes no quadrante dos Líderes. Além disso, possui produtos fortes e recursos de mercado confiáveis para sustentar um crescimento contínuo. A estes intervenientes falta o tamanho e a influência dos intervenientes no quadrante Líderes.
- **Visionários** – Um interveniente no quadrante Visionários oferece produtos inovadores que abordam, em larga escala, problemáticas do utilizador final, importantes tanto em termos operacionais como em termos financeiros. Porém, estes intervenientes ainda não demonstraram a capacidade de assegurar uma participação no mercado ou uma sustentabilidade lucrativa.
- **Niche players** – Estes intervenientes são frequentemente focados, de forma restrita, em mercados específicos. Ou os seus produtos estão num processo de adaptação para entrar num mercado mais amplo; ou ainda estão com dificuldade em desenvolver e por em prática a sua visão.

Uma das áreas para o qual existe um Quadrante Mágico da Gartner está com relacionado com *On-Premises Application Integration Suites (OPAIS)*. A consultora referiu que as empresas necessitam de um procedimento sistemático de integração de aplicações que suporte um Sistema de Mensagens robusto, vários protocolos *Business-to-business (B2B)* e *cloud* APIs. Na Figura 13 é possível consultar o Quadrante Mágico publicado no ano de 2014.

Figura 13: Quadrante Mágico para *OPAIS* (2014)

Para além desta, outra das áreas que pode ser referida no contexto da integração de sistemas está relacionada com *Enterprise Integration Platform as a Service (iPaaS)*. Uma iPaaS é uma plataforma que fornece recursos para permitir que sejam implementados projetos de integração de dados, de aplicações e de processos, abrangendo *endpoints* na *cloud* e *endpoints* locais. Nas Figuras 14 e 15 podem ser consultados os Quadrantes Mágicos de iPaaS dos anos entre 2015 e 2019.

Para determinar se existe alguma solução de ESB existente no mercado que possa resolver um determinado problema de integração, podem ser utilizados estes resultados da Gartner como um ponto de partida. Porém, tal não evita que seja feito um estudo às soluções afim de perceber realmente o proveito que se pode tirar delas.

(a) *Ano de 2015* **(b)** Ano de 2016

(c) Ano de 2017 **(d)** Ano de 2018

Figura 14: Quadrantes Mágicos para iPaaS (2016 – 2019)

Figura 15: Quadrante Mágico para iPaaS (2019)

2.3.4 Estudo sobre duas das soluções de ESB

Por forma a ter opinião sobre uma possível solução a adotar para a resolução de um Caso de Estudo, decidiu-se fazer um estudo sobre duas soluções *open source* de ESB: a solução da MuleSoft, o Mule ESB, [21] e a solução da WSO2, o Enterprise Integrator [33].

Para cada uma destas soluções foi realizado um teste prático, do qual se obtiveram as considerações apresentadas de seguida. O teste realizado consistiu na instalação local da solução, na preparação do ambiente de desenvolvimento e na definição de um exemplo de integração, tendo por base a documentação e os guias práticos disponibilizados nas páginas da Internet da cada solução.

Mule ESB

A solução da MuleSoft corresponde a um ESB *open source* baseada em Java. De uma forma geral considerou-se que esta solução permite a integração de novos sistemas de uma forma rápida e intuitiva e que pode ser utilizado com um simples configuração e instalação. A solução gere todas as interações entre os sistemas e componentes de forma transparente, independentemente do protocolo de comunicação subjacente utilizado [21].

De seguida apresentam-se algumas considerações sobre o Mule ESB:

- Permite a invocação de serviços remotos, de forma independente de formato e protocolo de comunicação.

- Possibilita o encaminhamento, filtro, agregação e reordenação de mensagens, tendo em conta o conteúdo ou outras regras de negócio.
- Permite a transformação de mensagens em diferentes formatos.
- Possui um conjunto vasto de conectores prontos a usar, que permitem a conexão a sistemas como: Salesforce [51], Amazon S3 [52], Amazon EC2 [53], Amazon SNS [54], MongoDB [55], Neo4j [56], Redis [57], ServiceNow [58], Slack [59], Apache Kafka [60], entre outros.
- Fornece uma *framework* de desenvolvimento de novos conectores e módulos.
- Permite a reutilização de componentes.
- Permite a exposição de serviços *web*, recorrendo à linguagem *RAML* [61].

WSO2 Enterprise Integrator

A solução da WSO2 estudada é também um ESB *open source*, que fornece recursos para a integração de aplicações através da definição de fluxos de mensagens, com ou sem estado, através de mecanismos de encaminhamento e transformação de mensagens [33].

De seguida apresentam-se algumas considerações sobre a solução da WSO2:

- Permite a comunicação entre sistemas, com diferentes formatos de dados e protocolos de comunicação.
- Possui uma grande quantidade de conectores prontos a usar, que permitem a conexão a sistemas como: Salesforce [51], Slack [59], Google Firebase [62], Apache Kafka [60], Google Cloud Pub/Sub [63], Amazon S3 [52], Amazon SNS [54], Redis [57], ElasticSearch [64], entre outros.
- Permite a implementação de novos conectores.
- Fornece mecanismos para a monitorização da instância e controlo de acessos.
- Permite o encaminhamento e filtro de mensagens.
- Oferece mecanismos para a transformação de mensagens em diferences formatos.

Comparação entre as soluções

De uma forma geral, ambas as soluções são *open source* e seguem as mesmas ideologias e padrões de integração apresentados até este ponto. Oferecem componentes de Encaminhamento, Transformação de Mensagens, Invocação, Segurança e Processamento de eventos. Além disso, também existe em ambas a noção de Conector, para permitir a integração com sistemas externos.

Relativamente aos Conectores, é de notar a atualização dos Conectores para o Mule ESB por parte da própria MuleSoft e da sua comunidade, reflectindo a evolução dos sistemas externos, a correção de erros e a melhorias das operações.

Além disso, ambas as soluções possuem à disposição um *Integrated Development Environment (IDE)* baseado em Eclipse para a implementação dos Conectores (em Java e *XML*) e para a definição da integração, permitindo uma a definição de fluxos de interação entre diferentes aplicações.

Em relação à monitorização, o WSO2 Enterprise Integrator possui um *dashboard* pré-concebido para a gestão e monitorização do Barramento, o que constitui uma vantagens em relação à solução da MuleSoft, que apenas permite a monitorização através de *Java Management Extensions (JMX)* [65]. Porém considerou-se que o WSO2 Enterprise Integrator é uma solução de difícil utilização, quando comparado com o Mule ESB.

De uma forma geral, considerou-se que qualquer uma das soluções analisadas poderia ser adotada para fazer a integração de aplicações num contexto empresarial. Porém, a solução da MuleSoft distinguiu-se pela documentação, pela quantidade de contribuições da comunidade, pelo número de conectores pré-concebidos e mantidos pela próprio MuleSoft, e pela facilidade com que pode ser utilizada. De salientar que a facilidade de utilização e a existência de uma curva de aprendizagem suave é importante pelo facto de permitir que qualquer programador dentro de uma empresa possa trabalhar e usufruir da solução de integração com o máximo rendimento possível.

Também é de referir o Mule ESB é uma solução que possui uma grande visibilidade a nível do mercado: a MuleSoft manteve, ao longo dos anos, a sua posição de Líder no Quadrante Mágico da Gartner para a categoria de iPaaS e, segundo a Gartner, é uma empresa que continua em crescimento, uma vez que dobrou a quantidade de clientes no ano de 2018.

3

Integração de sistemas com o Mule ESB

3.1 Visão global

A MuleSoft é uma empresa sediada em San Francisco, adquirida em 2018 pela Salesforce, cujo foco principal é o desenvolvimento de *software* para a integração de aplicações. Inicialmente, a empresa centrava a sua produção no desenvolvimento de um Sistema de Mensagens, mas mais tarde expandiu a sua oferta para uma plataforma de integração, com vários componentes (iPaaS) – a Anypoint Platform –, que se encontram a seguir:

- **Anypoint Design Center** – permite o desenho e construção de APIs;
- **Anypoint Connectors** – um conjunto de Conectores prontos a usar para a integração de aplicações;
- **Anypoint Analytics** – uma ferramenta que permite a agregação e análise de métricas, assim como a construção de gráficos e de *dashboards* para a visualização da *performance* da solução;
- **Anypoint Visualizer** – uma ferramenta que permite ter uma visualização gráfica sobre as APIs e as suas dependências, em tempo real;
- **Anypoint Exchange** – uma biblioteca onde uma equipa de desenvolvimento pode guardar e aceder as suas APIs, Conectores, documentação e outros recursos;

Além desta plataforma, a MuleSoft fornece separadamente o Mule Runtime (o *engine* da Anypoint Platform) para a integração de aplicações. Este *engine* é disponibilizado numa versão *open-source*, extensível, e será utilizado para a implementação do caso de estudo, no Capítulo 4. A MuleSoft lançou no final

do ano de 2017 a versão 4 do Mule Runtime, onde reformulou a arquitetura e a forma de desenvolvimento dos Conectores (que é abordada nas próximas Secções deste Capítulo). Desde então, tem vindo a lançar novas atualizações do *engine*: a última, à data de escrita desta Dissertação, a versão 4.2.1, foi lançada em julho de 2019.

Para além do *engine*, a MuleSoft oferece também duas outras ferramentas úteis para este processo, nomeadamente:

- **API Designer** – uma aplicação *web* para o desenho e documentação de uma API, com a possibilidade de partilha com os membros da equipa e de reutilização de componentes;
- **Anypoint Studio** – um ambiente de desenvolvimento (IDE), baseado em Eclipse, para a conceção e teste da solução de integração. À data da escrita desta Dissertação, a versão mais recente do Anypoint Studio era a 7.3. Esta versão tem vindo a ser atualizada ao longo do tempo com correções e melhorias.

3.2 Noção de Evento

No contexto do Mule Runtime, uma Mensagem está encapsulada numa estrutura bem definida designada por Evento (Figura 16).

A entidade Mensagem, por sua vez, é constituída por dois componentes: o Corpo (ou *payload*) e os Atributos (ou *metadata*) (semelhante àquilo que foi referido anteriormente no contexto dos Sistemas de Mensagens),

O corpo da Mensagem pode corresponder, por exemplo, ao conteúdo de um ficheiro, enquanto que os atributos podem corresponder aos meta-dados, como o nome, o tamanho, a data de modificação, etc.

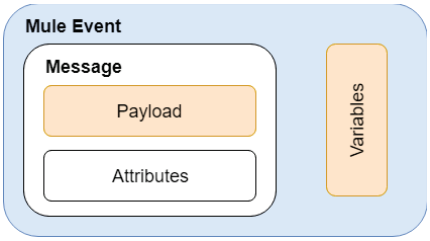


Figura 16: Estrutura de um Evento no contexto do Mule Runtime (**Adotado de:** *MuleSoft [21]*)

Para além da Mensagem, um Evento possui também um conjunto de Variáveis, que podem ser consultadas e alteradas ao longo do processamento da Mensagem, dependendo da lógica de negócio que seja implementada.

3.3 Mule Runtime e Aplicações Mule

A maioria das integrações requerem a Transformação de uma determinada estrutura de dados, à medida que esta se "move" desde a origem até ao destino. Com a solução da MuleSoft, é possível fazer *drag-n-drop* das diferentes operações que os conectores ou módulos expõe para um , como a operação de Transformação de mensagens (*Message Transformer*), e assim definir qual deve ser o processamento a aplicar em cada Mensagem.

A definição de um Fluxo pode ser feita num Projeto Mule, criado a partir do Anypoint Studio ou do Anypoint Design Center, e sua compilação dá como resultado uma Aplicação Mule, que pode ser executada pelo Mule Runtime.

É através de uma Aplicação Mule que os programadores podem estabelecer conexões entre sistemas, serviços e APIs, seguindo os princípios básicos de EAI e evitando as integrações Ponto a Ponto. Em cada uma destas Aplicações podem ser utilizadas as funcionalidades fornecidas pelo ESB, como o Encaminhamento, a Transformação, a Invocação, entre outras [21].

O Mule Runtime, como já referido, é um *engine* de integração, desenvolvido em Java, capaz de executar e servir Aplicações Mule e é o que permite estabelecer conexões entre diferentes sistemas seguindo os padrões de EAI.

O *runtime* da MuleSoft utiliza o Java Service Wrapper, que, para além de permitir controlar a *Java Virtual Machine (JVM)* a partir do Sistema Operativo, oferece às aplicações Java vantagens e funcionalidades, como:

- A possibilidade de instalar, executar e remover a aplicação Java como um serviço em Windows ou um *daemon* nos sistemas Unix;
- Uma configuração flexível;
- Manipulação dos sinais do Sistema Operativo e dos parâmetros de arranque;
- Monitorização da JVM e reinício automático da aplicação (após a deteção de um *deadlock*, por exemplo);
- *Logging* configurável e envio de alertas.

No caso do Mule Runtime, o *wrapper* pode ser invocado através de um *script* presente na diretoria `$MULE_HOME/bin`, que oferece como principais comandos: `install` (para a instalação do serviço ou do *daemon*), `remove` (para a remoção), `start` (para o arranque em segundo plano), `stop`, `restart` e `status`.

Em cada instância do *runtime* existe a possibilidade de se ter diferentes Aplicações Mule em execução, o que permite a existência de determinadas vantagens como:

- ser possível "'dividir" uma Aplicação Mule complexa em diferentes Aplicações, cada uma com a sua lógica específica, e fazer *deploy* das mesmas numa única instância;
- a quebra dos limites sobre as configurações que se deseja implementar;
- a possibilidade de se ter Aplicações Mule com dependências e versões diferentes, ao mesmo tempo.

De referir que o *runtime* faz um isolamento ao nível do *classloader* entre as diferentes Aplicações Mule, mas é possível fazer a partilha de recursos entre as mesmas.

3.4 Estrutura de uma Aplicação Mule

A estrutura do Projeto que permite o desenvolvimento de uma Aplicação Mule segue a estrutura básica de um projeto Maven. O Apache Maven é uma ferramenta de *software* que tem o objetivo facilitar a criação, automatização e gestão de dependências de projetos Java e que permite tornar mais fácil o desenvolvimento e a compilação de aplicações.

Na Figura 17 encontra-se representada a estrutura de ficheiros de uma Aplicação Mule, assim como uma descrição sucinta dos principais ficheiros e diretorias:

- `mule-app/pom.xml` – definição das dependências e dos módulos que são necessários para a compilação do Projeto;
- `mule-app/mule-artifact.json` – descrição da Aplicação (nome, versão mínima do *rumtime*, etc);
- `mule-app/src/main` – código fonte e recursos (*resources*) que fazem parte do *Artifact* (o resultado da compilação);
- `mule-app/src/test` – código fonte e recursos de teste à aplicação;

```
mule-app
├── mule-artifact.json
├── pom.xml
└── src
    ├── main
    │   ├── java
    │   ├── mule
    │   │   └── mule-app.xml
    │   └── resources
    │       ├── api
    │       └── log4j2.xml
    └── test
        ├── java
        ├── munit
        └── resources
            └── log4j2-test.xml
```

Figura 17: Estrutura de ficheiros base de uma Aplicação Mule

3.5 Configuração de uma Aplicação Mule

De toda a estrutura de uma Aplicação Mule, a principal componente que define a integração reside na diretoria `mule-app/src/main/mule` e consiste num conjunto de ficheiros *Extensible Markup Language (XML)*, escritos através de uma *Domain Specific Language (DSL)*. Em cada um destes ficheiros encontra-se a definição de diferentes elementos que representam uma configuração para uma determinada entidade dentro do sistema Mule Runtime. Ao longo desta Secção são apresentados os principais elementos.

3.5.1 Configuração Global

Uma Configuração Global corresponde a uma definição genérica que é aplicada a toda a Aplicação e que é declarada ao nível da raiz do ficheiro XML. Neste elemento podem ser configurados os seguintes atributos:

```
<configuration
    defaultTransactionTimeout="30000"
    defaultResponseTimeout="1000"
    shutdownTimeout="5000"
    maxQueueTransactionFilesSize="500">
</configuration>
```

3.5.2 Propriedades

Podem ser declaradas numa Aplicação diferentes propriedades para futura referência (através do *Ant-style*) ou a importação de um ficheiro `.yaml` ou `.properties`.

```
<configuration-properties file="smtp.yaml"/>
<global-property name="smtp.host" value="smtp.mail.com"/>
<global-property name="smtp.subject" value="Subject of Email"/>
...
<email:smtp-config name="config">
    <email:smtp-connection host="${smtp.host}" port="${smtp.port}"/>
</email:smtp-config>
```

A declaração e utilização destas propriedades numa Aplicação Mule permite uma configuração para ambientes distintos (como um ambiente de testes e outro de produção), sem que o código seja alterado.

```
<global-property name="env" value="dev"/>

<configuration-properties file="${env}-properties.yaml"/>
```

3.5.3 Fluxos e Subfluxos

As Aplicações Mule processam as Mensagens através de componentes implementados por Conectores e Módulos. Esses componentes são colocados em sequência para definir o conjunto de Operações que devem ser aplicadas a cada Mensagem, formando aquilo se chama de Fluxo, como pode ser visto na Figura 18.

Uma Aplicação pode consistir num único Fluxo ou pode ter o processamento repartido por diferentes Fluxos e Subfluxos, organizados por Módulos funcionais, por exemplo.

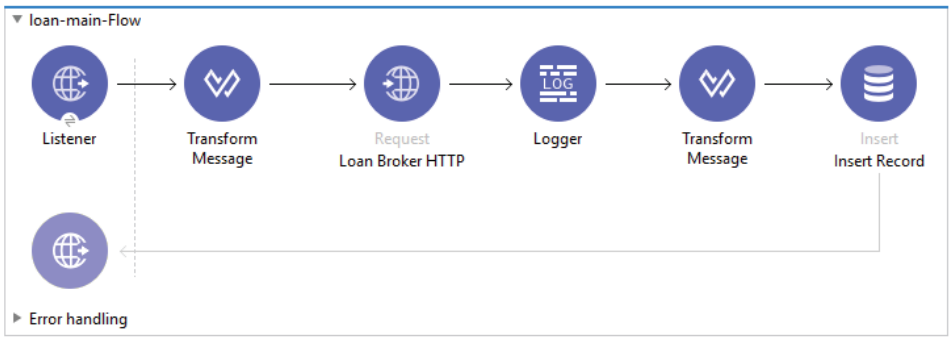


Figura 18: Fluxo exemplo (**Adotado de:** *Profit [66]*)

Referência a outros Fluxos

Uma vantagem da separação da Aplicação em diferentes Fluxos está relacionada com a reutilização da lógica de processamento.

Através de uma Operação designada *Flow Reference* (Figura 19) é possível, num determinado Fluxo, fazer referência a um outro Fluxo. A Mensagem é encaminhada para o Fluxo referenciado, que a processa e devolve o resultado para o Fluxo que o referenciou. Pode-se olhar para esta Operação como a invocação de uma função.

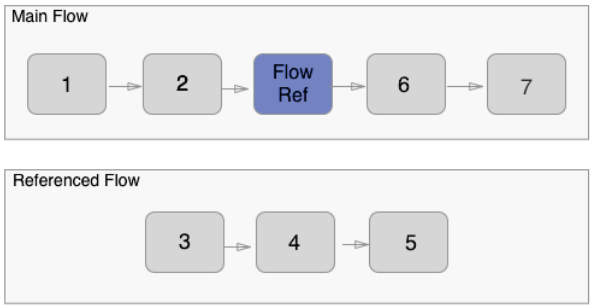


Figura 19: Operação *Flow Reference* (**Adoptado de:** *MuleSoft [21]*)

Execução assíncrona

Existem disponíveis alguns componentes que permitem a execução de um determinado conjunto de Operações de forma assíncrona, isto é, componentes que permitem que o processamento de uma ou mais Mensagens seja feito em *threads* separadas, permitindo que a execução do Fluxo continue paralelamente.

Um dos componentes que permitem a execução assíncrona de Operações é o *Async scope*. Com este componente, as Operações são executadas simultaneamente com o Fluxo principal e é útil para a execução de Operações demoradas e que não requerem o envio de uma resposta de volta.

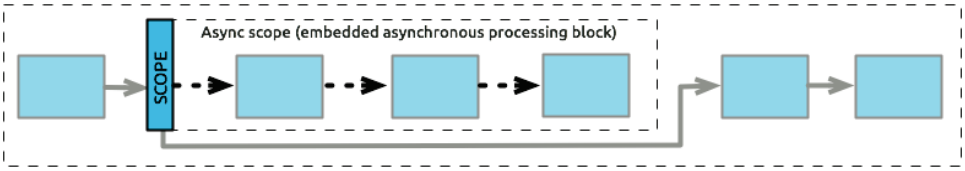


Figura 20: Componente *Async scope* (**Adoptado de:** *MuleSoft [21]*)

Um outro componente que permite a execução de Operações de forma assíncrona é o *Scatter-Gather Router*. Este é um componente de Encaminhamento, que faz uma cópia da Mensagem de entrada para dois ou mais processadores de Mensagens. Cada um destes processadores corresponde a uma sequência de um ou mais componentes (semelhante a um Subfluxo) e a sua execução é feita em paralelo.

O componente *Scatter-Gather* faz a consolidação (*gathers*) das Mensagens de cada uma das *threads* numa nova Mensagem. Quando todas as *threads* terminam o processamento, a Mensagem consolidada é passada para o próximo componente no Fluxo.

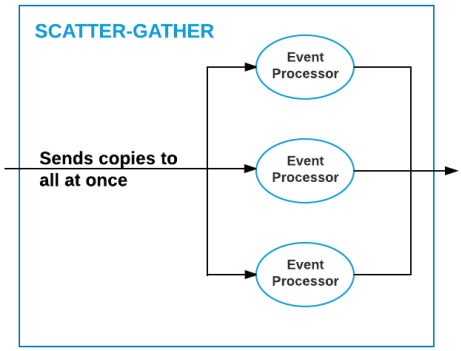


Figura 21: Componente *Scatter-Gather* (**Adoptado de:** *MuleSoft [21]*)

Tratamento de erros

Para além do fluxo normal de execução, num Fluxo pode ser definida a lógica que deve ser executada em caso de ocorrência de erros. Essa definição pode ser feita ao nível do Fluxo ou mais especificamente em determinadas Operações, através do componente *Try scope*, à semelhança do que é possível fazer em Java com os blocos *Try-Catch*.

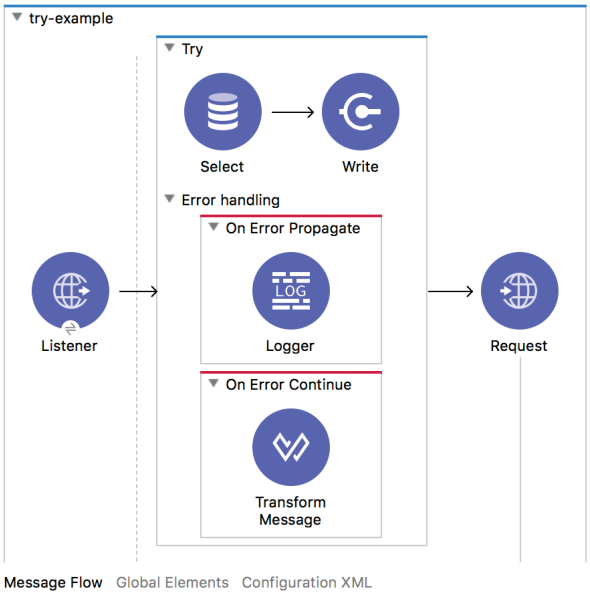


Figura 22: Componente *Try Scope* (**Adoptado de:** *MuleSoft [21]*)

Tal como é possível ver na Figura 22, existem dois tipos de tratamento de erros:

- *On Error Continue* – Em que é feita a execução das Operações de tratamento de erro e, no final dessas Operações, a execução do fluxo normal continua.
- *On Error Propagate* – Em que a execução das Operações de tratamento de erro são executadas e, no final, o erro é propagado.

3.5.4 Configuração de Conectores e de Módulos

Os Conectores facilitam a integração com sistemas de terceiros e os Módulos são extensões que adicionam funcionalidades ao Mule Runtime.

Cada um destes componentes necessita de ser configurado para desempenhar a sua função. Por exemplo, existe um Conector pronto a usar, mantido pela MuleSoft, que permite o envio de pedidos *Hypertext Transfer Protocol (HTTP)* a um servidor. Para que este Conector possa ser utilizado num Fluxo, é necessária a definição de uma configuração como a que se segue.

```
<http:request-config name="http-config" basePath="/api">
    <http:request-connection host="localhost" port="8080">
        <http:authentication>
            <http:basic-authentication username="${user}" password="${passwd}"/>
        </http:authentication>
    </http:request-connection>
</http:request-config>
```

De referir que podem ser declaradas diferentes configurações para o mesmo Conector ou Módulo numa mesma aplicação Mule. Quando é feita a invocação de uma Operação desse Conector é escolhida qual a configuração que se pretende usar.

3.5.5 Operações

As Operações são um dos conceitos mais importares no Mule ESB. Uma Operação representa uma ação que pode ser utilizada dentro de um Fluxo para processar uma Mensagem, através de uma determinada lógica de negócio implementada por um Conector.

Por exemplo, o Conector de HTTP, já referido anteriormente, expõe uma Operação que permite fazer pedidos HTTP a um servidor. Essa Operação pode ser declarada dentro de um Fluxo da seguinte forma:

```
1 <flow name="mule-appFlow">
2     ...
3     <http:request method="GET" path="/clients/1" config-ref="http-config"/>
4     ...
5 </flow>
```

Esta Operação faz uso da configuração do Conector declarada anteriormente, pelo atributo `config-ref="http-config"` que corresponde ao `name` da configuração. Neste caso, será feito um pedido HTTP GET, para o *endpoint* `http://localhost:8080/api/clients/1`.

3.5.6 Fonte

Uma Fonte é um componente responsável por criar Mensagens para serem processadas pelo Mule Runtime, sendo sempre o ponto de entrada de um Fluxo. Como exemplo, o Conector de HTTP implementa a *Source HTTP Listener*, que permite a inicialização de um servidor HTTP, numa determinada porta, que faz desencadear a execução de um Fluxo sempre que recebe um pedido. O corpo da Mensagem que é enviada no pedido HTTP é utilizado para construir a Mensagem publicada no Fluxo.

Um outro exemplo de Fonte é o componente *JMS Listener* do Conector de *Java Message Service (JMS)*, que permite estabelecer uma conexão com um *broker* e despoletar a execução de um Fluxo sempre que for publicada uma Mensagem numa fila.

3.6 Linguagem DataWeave

DataWeave é uma linguagem de *scripting* que a MuleSoft criou para aceder e fazer Transformações na informação que "passa" no Barramento.

Geralmente, os *scripts* DataWeave são utilizados nas Aplicações Mule para alterar o corpo das Mensagens, como por exemplo para remover campos desnecessários e/ou acrescentar informação ao corpo da Mensagem, numa alusão aos princípios de *Content Enricher* e *Content Filter* falados na Secção 2.3.2.

Esta linguagem, à semelhança de outras, lida com diferentes tipos de dados, como `String`, `Number`, Tempos e Datas (`Date`, `DateTime`, `Period`, `Time`, ...), `Enums`, `Iterator`, `Boolean`, `Array`, `Object`, entre muitos outros. Para cada um destes tipos de dados existe um grande conjunto de operações que podem ser usadas.

A utilização desta linguagem pode ser feita de duas formas: através da escrita de *scripts* em componentes de Transformação de Mensagens (como o *Message Transformer*, que pode ser visto na Figura 22, ou o *Set Payload*); ou através da escrita de expressões *inline* para a Transformação da informação *in-place* ou para a atribuição dinâmica de propriedades, como atributos de configuração das Operações (por exemplo, `<example value="#[now()]"`).

A estrutura de um *script* DataWeave é composta por duas partes separadas por `---`: um Cabeçalho, onde se definem as diretivas a aplicar; e um Corpo, onde está contida a expressão que vai gerar a estrutura de saída.

```
output application/xml
---
example: {
    user: { firstName: payload.user_firstname, lastName: payload.user_lastname },
    code: upper(payload.code),
    products: payload.products mapObject ((value, key) ->
        product: {id: key, price: value}
    )
}
```

De notar que a Mensagem de saída do *script* anterior terá o formato XML e que se se pretender que a Mensagem tenha outro formato, basta alterar o Cabeçalho do *script*, sem que se tenha que alterar o código DataWeave.

Outra das vantagens de utilização desta linguagem é o facto de ser possível ter uma pré-visualização do resultado das operações, bastando para isso definir um exemplo de entrada na Aplicação Mule, como se pode ver na Figura 23.

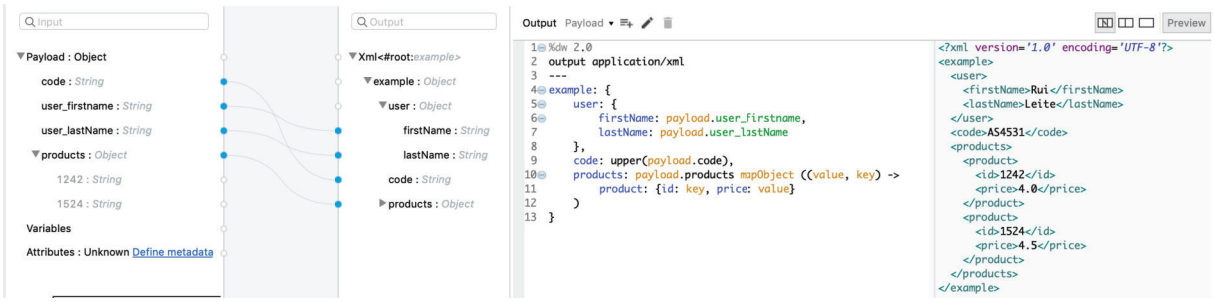


Figura 23: Pré-visualização de uma Transformação no Anypoint Studio

Além da escrita de código DataWeave, também é possível invocar métodos Java diretamente a partir de um *script*, bastando para isso implementar os métodos Java na diretoria `mule-app/src/main/java`. No seguinte exemplo, encontra-se a utilização de um método `appendRandom()`, da classe `utils.MyUtils`.

```
1 import java!utils::MyUtils
2 output application/json
3 ---
4 {
5     a: MyUtils::appendRandom("myString")
6 }
```

3.7 Deploy das Aplicações Mule para o Mule Runtime

Tal como já referido, o Mule Runtime está preparado para executar diferentes Aplicações, podendo assim ser visto como um servidor aplicacional.

Uma das diretorias que constitui a estrutura de ficheiros do *runtime* é a `$MULE_HOME/apps`. O *runtime* é responsável por verificar se existe alguma nova Aplicação nesta diretoria e, caso haja, de a iniciar. Para além disso, caso haja alguma alteração em algum dos ficheiros de qualquer Aplicação, a mesma é atualizada.

As Aplicações Mule podem ser movidas para a referida diretoria no formato *Java ARchive (JAR)* ou de forma descompactada (na prática, caso se faça *deploy* de uma Aplicação no formato JAR, o Mule Runtime trata de a descompactar). Também é possível manter as aplicações numa diretoria diferente, mas é sempre necessário criar um *symbolic link* (um "atalho") na diretoria `$MULE_HOME/apps`.

Tal como já referido anteriormente, uma Aplicação Mule é desenvolvida recorrendo ao Apache Maven. Para agilizar o *deploy* das aplicações para o Mule Runtime, a MuleSoft disponibiliza um *plugin*, o Mule Maven Plugin, declarado no ficheiro `pom.xml`, que permite fazer o *deploy* da Aplicação. Para tal, basta acrescentar uma configuração ao *plugin*, como a que se segue e utilizar o comando `mvn clean package deploy -DmuleDeploy`.

```
1 <configuration>
2     <standaloneDeployment>
3         <muleHome>/opt/mule-standalone</muleHome>
4         <muleVersion>4.1.1</muleVersion>
5     </standaloneDeployment>
6 </configuration>
```

Sempre que ocorre uma tentativa de *deploy* de uma Aplicação, o *runtime* disponibiliza informação sobre o resultado do mesmo como é possível verificar na Figura 24.

```
************************************************************************
*             - - + DOMAIN + - -              * - - + STATUS + - - *
************************************************************************
* default                                     * DEPLOYED           *
************************************************************************

************************************************************************************************************
*           - - + APPLICATION + - -           *       - - + DOMAIN + - -       * - - + STATUS + - - *
************************************************************************************************************
* proj1                                       * domain                        * DEPLOYED           *
************************************************************************************************************
```

Figura 24: *Logs* do Mule Runtime no *deploy* de uma Aplicação Mule

3.8 Desenvolvimento de novos Módulos e Conectores

De forma a permitir estender as funcionalidades do Mule Runtime, a MuleSoft criou dois *Software Development Kits (SDKs)* para o desenvolvimento de novos Módulos e Conectores: o XML SDK e o Java SDK, ou apenas Mule SDK.

3.8.1 XML SDK

O XML SDK trata-se de uma versão simplista para a criação de Módulos recorrendo a XML. O objetivo é a definição de um conjunto de Operações e dos respetivos parâmetros, sendo que em cada uma das Operações pode ser utilizada uma ou mais Operações de outros Módulos. Assim, as Operações podem ser vistas como um encapsulamento de outras e são capazes de ser reutilizadas em qualquer Fluxo.

De seguida encontra-se a declaração de um Módulo que expõe a Operação `sum`.

```
1  <module name="Math XML SDK">
2      <operation name="sum">
3          <parameters>
4              <parameter name="numberA" type="number"/>
5              <parameter name="numberB" type="number"/>
6          </parameters>
7          <body>
8              <mule:set-payload value="#[vars.numberA + vars.numberB]"/>
9          </body>
10         <output type="number"/>
11     </operation>
12 </module>
```

3.8.2 Java SDK

O Java SDK permite o desenvolvimento de Módulos ou Conectores para o Mule Runtime através de código Java e de uma API desenvolvida pela MuleSoft – a *Extensions API*. Assim como uma Aplicação Mule, no desenvolvimento de um Conector é utilizando o Apache Maven, sendo que, depois de desenvolvido, o Conector é adicionado às dependências de uma Aplicação Mule, por forma a poder ser utilizado.

Estrutura de um Módulo ou Conector

O desenvolvimento de um Módulo ou Conector para o Mule Runtime requer que seja feito um pequeno estudo a cada um dos componentes que compõe a sua estrutura (Figura 25).

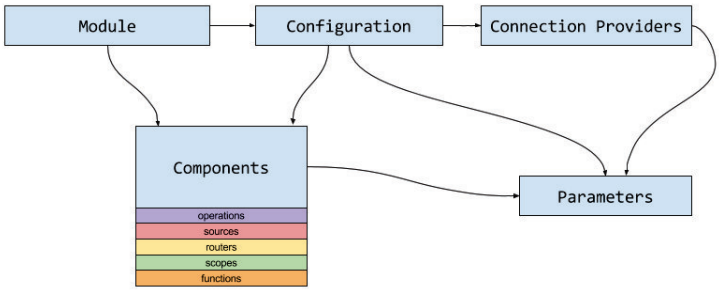


Figura 25: Estrutura básica de um Conector no Mule ESB (**Adoptado de:** *MuleSoft [21]*)

- **Módulo** – O ponto de entrada de um Módulo ou Conector é uma classe Java anotada com `@Extension`. Esta classe é utilizada para exportar toda a funcionalidade do Módulo e dos elementos associados a ele.
  No seguinte excerto, encontra-se um exemplo da declaração de um Conector. Todas as restantes anotações são referidas ao longo desta Secção, em cada um dos componentes na estrutura.

```
1 @Extension(name = "Foo")
2 @Configurations({FooConfig.class})
3 @Operations({FooOperations.class})
4 @ExpressionFunctions({FooFunction.class})
5 @Operations({FooRouter.class})
6 @ConnectionProviders({FooConnectionProvider.class})
7 public class FooConnector {
8 }
```

- **Configurações** – Uma Configuração corresponde a um conjunto de Parâmetros que afetam o comportamento do Módulo ou Conector como um todo. A forma mais indicada para a definição de uma Configuração, por questões de manutenibilidade e separação de responsabilidades, é a escrita de uma classe Java em que as variáveis de instância correspondem aos Parâmetros da Configuração. De referir que um mesmo Módulo pode ter mais do que uma Configuração.

```
1 @Configuration(name="config")
2 public class FooConfig {
3
4     @Parameter private int parameterA;
5
6     public int getParameterA() { return this.parameterA; }
7
8 }
```

- **Parâmetros** – Um Parâmetro é o componente mais granular da estrutura de um Módulo ou Conector e pode ser utilizado em Operações, Fontes, Configurações e *Connection Providers*. Um Parâmetro representa um argumento configurável e necessário para que um determinado componente possa executar de forma apropriada.
  Podem ser definidos como uma variável de instância de uma classe ou como um argumento de um método, dependendo do componente onde está a ser usado.

```
@Optional(defaultValue="80")
@Parameter
private String foo;
```

- **Connection Providers** – Um *Connection Provider* é um elemento que tem a responsabilidade de criar e terminar as Conexões (*Connections*) com o sistema externo. Cada Conexão é criada tendo em conta uma Configuração, sendo que um mesmo Módulo pode possuir mais do que um tipo de Conexão (por exemplo, a Conexão HTTP e uma Conexão *Hyper Text Transfer Protocol Secure (HTTPS)*). Para cada tipo de Conexão deve existir uma implementação de um *Connection Provider* e todas as implementações devem estar declaradas na extensão através da anotação `@ConnectionProviders`.

```
class FooConnectionProvider implements ConnectionProvider<FooConnection> {

    @Parameter
    private String password;

    public FooConnection connect() throws ConnectionException {
        // Estabelecimento de uma Conexão pronta a usar
        return new FooConnection(password);
    }

    public void disconnect(FooConnection connection) {
        connection.invalidate();
    }

    public ConnectionValidationResult validate(FooConnection connection) {
        // Validação da Conexão estabelecida. Se a validação falhar, uma
        // nova Conexão é estabelecida
  }
}
```

De notar que a classe `FooConnectionProvider` implementa a *interface* `ConnectionProvider<>`. Isto significa que sempre que uma Operação necessite de uma Conexão, é invocado o método `connect()`, e que quando a Operação estiver completa, a Conexão é destruída. Porém, existem duas outras estratégias para a obtenção de uma Conexão:

  - **Pooling** – é mantida uma *cache* de Conexões para que possam ser reutilizadas no futuro. Esta estratégia é útil nos casos em que a instanciação de uma Conexão possui um elevado custo computacional ou quando o acesso concorrente à mesma instância não é suportado. Para utilizar esta estratégia implementa-se a *interface* `PoolingConnectionProvider<>`;
  - **Cached** – é mantida uma instância para cada tipo de Conexão que exista e é sempre usada a mesma instância nas Operações que usem a mesma configuração (o mesmo tipo de Conexão). Esta estratégia pode ser utilizada através da *interface* `CachedConnectionProvider`.

- **Componentes** – A forma como é definida toda a funcionalidade que o Módulo expõe é através da declaração de diferentes tipos de Componentes.

  - **Operações** – Tal como já falado na Secção 3.5.5, as Operações representam as ações que um Módulo ou Conector pode executar num Fluxo. Com o Java SDK, cada uma das Operações faz correspondência com um método `public`, onde estão definidos os parâmetros de entrada e o tipo de retorno da Operação. Para que o compilador considere essas mesmas Operações, a classe onde estão definidas tem que ser declarada na extensão, utilizando a anotação `@Operations`.

    O método, para além dos parâmetros da Operação, pode receber dois tipos "especiais" de argumentos: uma configuração (`@Config`) ou uma Conexão (`@Connection`), quando a Operação necessita de invocar a API de um sistema externo, por exemplo.

```java
public class FooOperations {

    public int foo(@Config FooConfig config, int parameter) {
        return config.getParameterA() * parameter;
    }

}
```

  - **Fontes** – Na Secção 3.5.6 já foi referido o conceito de Fonte – são os componentes responsáveis por criar as Mensagens de um Fluxo. Em termos de implementação, a definição de uma Fonte é mais complexa do que a definição de uma Operação: é necessária a criação de uma classe que deve estender a classe genérica `Source<T,A>`, parametrizada com o tipo do corpo da mensagem (`T`) e o tipo dos atributos (`A`) das Mensagens que são criadas. Para além disso, devem ser implementados os métodos `onStart()` e `onStop()`.

    Existe um tipo "especial" de Fonte, a *Polling Source*, que é ativada por um *Scheduler*, i.e., periodicamente, e pode pode ser utilizado para fazer *poll* de dados de um sistema externo, com uma determinada frequência.

    No Anexo B.1 encontra-se um exemplo de implementação de uma Fonte, acompanhado de alguma explicação.

- **Funções** – A implementação de uma Função permite a contribuição de novas funcionalidades para a linguagem DataWeave e é feita de forma semelhante à implementação das Operações (sem a possibilidade de usar uma Conexão ou uma Configuração).
  Na classe de declaração do Conector (`FooConnector`) tem que ser declarada esta função através da anotação `@ExpressionFunctions`.

```
public classe FooFunction {
    public Object xpath(InputStream item, String expression) {
        return xPathFactory.newXPath()
            .evaluate(expression, documentBuilder.parse(item));
    }
}
```

  Esta Função pode ser usada no código DataWeave da seguinte forma:
  `<example value="#[Foo::xpath(''/books/book[price>35]/title'')]" />`

- **Encaminhadores** – Componentes que permitem o Encaminhamento de Mensagens num Fluxo, recebendo um conjunto de possíveis rotas de execução e uma expressão *booleana* passada como parâmetro.
  Para a implementação deste tipo de componente tem ser declarada, pelo menos, uma Rota, através da escrita de uma classe Java que estende a classe `Route`.

```
public class WhenRoute extends Route {

    @Parameter
    private boolean shouldExecute;

    public boolean shouldExecute() { return shouldExecute; }
}
```

  Depois de definidas as Rotas, é necessário criar o componente Encaminhador (*Router*), responsável por Encaminhar as Mensagens. Este componente pode ser declarado como uma das Operações do Módulo (na classe `FooOperations`).

```
public void choice(RouteA rA, RouteB rB, RouterCompletionCallback cb) {
    if (rA.shouldExecute()) {
        rA.getChain().process(cb::success, (e, p) -> cb.error(e));
    } else if (rB.shouldExecute()) {
        ...
    } else {
        cb.success(Result.builder().build());
    }
}
```

  Este componente pode ser utilizado uma aplicação Mule através do seguinte código XML:

```
<flow name="logDecoratorSampleFlow">
    <docs:choice>
        <docs:ra shouldExecute="#[payload != null]">
            <http:request config-ref="config" path="/" method="GET"/>
        </docs:ra>
        <docs:rb>
            <logger message="Payload was null"/>
        </docs:rb>
    </docs:choice>
</flow>
```

– **Scopes** – Componente que inclui a execução de um conjunto de outras Operações que recebe como argumento.

No seguinte exemplo, qualquer Operação declarada "dentro" do *Scope* `logDecorator` fica "envolvida" pela invocação das Operações de *logging*. O *Scope* pode ser declarado dentro de uma classe de Operações (`FooOperations`).

```
public void logDecorator(Chain operations, CompletionCallback cb) {
    LOGGER.debug("Invoking child operations")
    operations.process(
        result -> {
            LOGGER.debug(result.getOutput());
            cb.success(result);
        },
        (e, p) -> {
            LOGGER.error(e.getMessage());
            cb.error(e);
        }
    );
}
```

```
<flow name="logDecoratorSampleFlow">
    <docs:log-decorator>
        <http:request config-ref="config" path="/" method="GET"/>
    </docs:log-decorator>
</flow>
```

3.9 Gestão de threads

O Mule Runtime faz uma gestão automática de *threads*, mantendo a cada momento diferentes *pools* que podem ou não ser partilhadas pelas Aplicações Mule em execução.

Cada uma das Aplicações Mule em execução obtém as *threads* a partir das *pools*, à medida que os Eventos passam pelos Processadores no Fluxo. Isto quer dizer que um mesmo Fluxo pode ser executado

em diferentes *threads*, embora o *runtime* faça uma gestão otimizada para evitar trocas desnecessárias entre *threads*.

Existem três *pools* específicas no *runtime* que são partilhadas pelas aplicações: `CPU_INTENSIVE`, `CPU_LITE` e `BLOCKING_IO`. A *Source* de um Fluxo e cada um dos Processadores seguintes são executados numa *thread* obtida a partir de uma destas *pools*, com a exceção da *Source HTTP Listener* e da Operação *HTTP Requester* do Conector HTTP, que utiliza a *framework* Grizzly *Non-blocking I/O (NIO)* e que mantém duas *thread pools* específicas: o *HTTP Listener* utiliza uma *pool* partilhada e o *HTTP Requester* utiliza uma *pool* dedicada a cada Aplicação Mule.

Cada uma das tarefas executadas por um Processador de Eventos pode ser classificada como 100% Não-bloqueante, Potencialmente bloqueante ou Bloqueante. Esta classificação faz correspondência com a responsabilidade de cada uma das *thread pools*, sendo que: a *pool* `CPU_LITE` é para as tarefas que são 100% Não-bloqueantes e cuja execução demora tipicamente menos do que $10ms$; a *pool* `CPU_INTENSIVE` é destinada às tarefas que tipicamente demoram mais do que $10ms$ e que são Potencialmente bloqueantes em menos do que 20% do tempo de relógio; a *pool* `BLOCKING_IO` é para as tarefas que são Bloqueantes na maior parte do tempo (Figura 26).

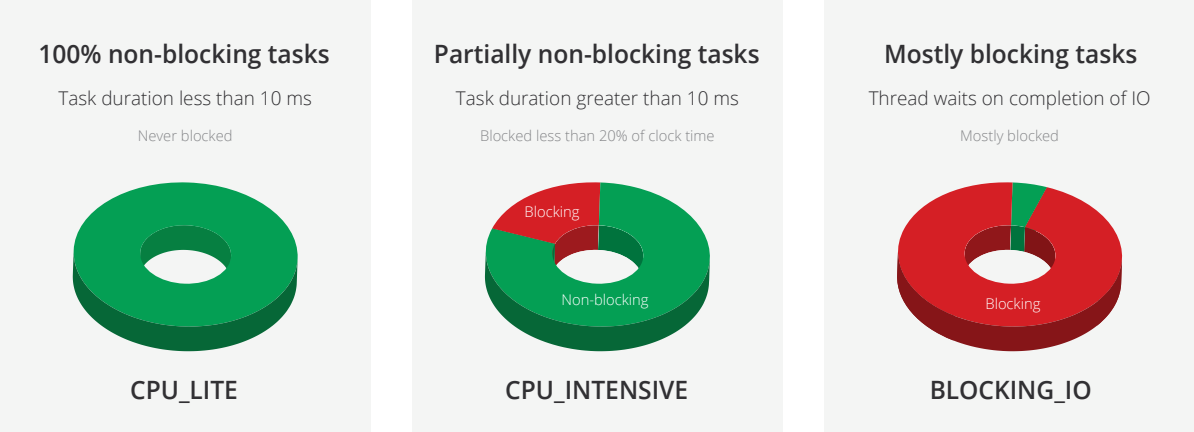


Figura 26: Responsabilidade de cada *thread pool* no Mule Runtime (**Adotado de:** *MuleSoft [21]*)

Relativamente ao tamanho mínimo e máximo de cada uma das *thread pools*, este depende do número de *cores* do *Central Processing Unit (CPU)* e da Memória disponíveis para o *runtime*. O tamanho mínimo de todas as *thread pools* é igual ao número de *cores* do CPU, como se pode ver pela Tabela 1, e este número incrementa até ao valor máximo com um fator igual a 1. De notar que para a *pool* `BLOCKING_IO` a fórmula apresentada é o resultado de testes de desempenho ao *runtime*, realizados pela MuleSoft (*mem* em *KB*).

Cada uma das *threads pools* é gerida por um componente distinto designado por *Scheduler*, que é responsável por extrair e repor as *threads* da respetiva *pool*. Sempre que é feito o *deploy* de uma Aplicação Mule para o *runtime*, a cada um dos Processadores de Eventos de todos os Fluxos é atribuído um *Scheduler*, seguindo os critérios definidos na seguinte Tabela.

Pool	Min	Max	Alocação
`CPU_LITE`	*#cores*	$2 \times$ *#cores*	No arranque do *runtime*
`CPU_INTENSIVE`	*#cores*	$2 \times$ *#cores*	No arranque do *runtime*
`BLOCKING_IO`	*#cores*	$\#cores + \frac{mem - 245760}{5120}$	No arranque do *runtime*
`SHARED_GRIZZLY`	*#cores*	$\#cores + 1$	No *deploy* da primeira aplicação com *HTTP Listener*
`DEDICATED_GRIZZLY`	*#cores*	$\#cores + 1$	No *deploy* de cada aplicação com *HTTP Requestor*

Tabela 1: Tamanho mínimo e máximo de cada *thread pool* do Mule Runtime

Scheduler/Pool	Processador de Eventos
`CPU_INTENSIVE`	DataWeave (Transformação de Mensagens) e Operações do Módulo *Scripting*
`BLOCKING_IO`	Operações de *I/O* com retorno (por exemplo, acesso a BDs) e Transações
`SHARED_GRIZZLY`	*HTTP Listener*
`DEDICATED_GRIZZLY`	*HTTP Requestor*
`CPU_LITE`	Restantes Processadores e Transições entre Processadores

Tabela 2: Critérios de atribuição de cada Processador a um *Scheduler*

Na Figura 27 encontra-se um exemplo de execução de um Fluxo, composto por diferentes tipos de Processadores. Nele é possível ver o processo de atribuição dos *Schedulers* a cada Processador, como definido nos critérios da Tabela 2. A descrição desse processo encontra-se a seguir:

1. A *thread* #10 `SHARED_GRIZZLY` recebe o pedido HTTP, da *Source HTTP Listener*.
2. A *thread* #8 `CPU_LITE` procede com a transição entre o *HTTP Listener* e a Operação *Database select*.
3. A *thread* #5 `BLOCKING_IO` procede com a chamada à BD e espera pelo resultado.
4. É necessária uma *thread* da *pool* `CPU_LITE` para a Operação *Logger* e a otimização do *Scheduler* permite que seja utilizada a *thread* #2 também para a transferência de Processador antes e depois desta Operação.

5. A *thread* #16 `CPU_INTENSIVE` executa a transformação DataWeave.

6. Ocorre novamente uma otimização para a segunda Operação *Logger*, com a *thread* #1 `CPU_LITE`. Esta *thread* procede também com a invocação do pedido HTTP, mas é a *thread* #2 `DEDICATED_GRIZZLY` que recebe a resposta.

7. Ocorre uma nova otimização na conclusão do Fluxo: a *thread* #7 `CPU_LITE` procede com a transição para o início do Fluxo (*HTTP Listener*) e também executa a resposta para o cliente.

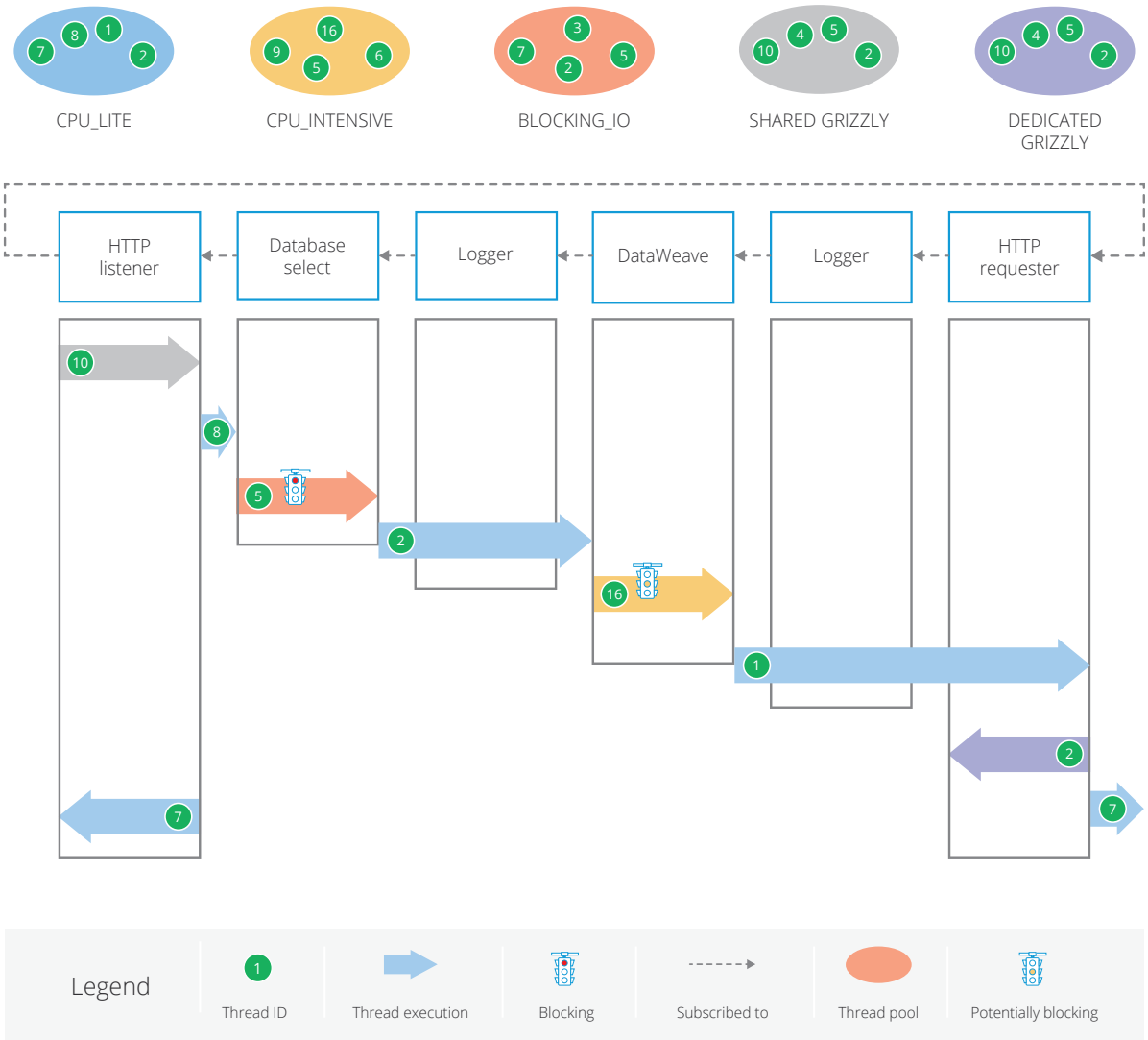


Figura 27: Gestão de *threads* na execução de um Fluxo de exemplo (**Adotado de:** *MuleSoft [21]*)

3.10 Concretização dos padrões de EAI no Mule ESB

Ao longo do Capítulo 2, particularmente nas Secções 2.2 e 2.3.2, foram apresentados diferentes padrões de EAI. Sendo o Mule ESB uma solução para a integração de aplicações num contexto empresarial, é pertinente encontrar de que forma a MuleSoft incluiu esses mesmos padrões na sua solução.

Em relação às quatro abordagens ou estilos de integração apresentadas na Secção 2.2.4 desta Dissertação, tem-se:

- **Partilha de ficheiros** – A MuleSoft disponibiliza três Conectores para lidar com ficheiros e diretorias: o Conector *File*, o Conector *FTP* e o Conector *FTPS*.
- **BD partilhada** – Existe um Conector que permite o estabelecimento de uma Conexão entre o *runtime* e uma BD relacional, para a invocação de diferentes operações, como *queries*, inserções, atualizações, remoção de registos, execução de procedimentos e utilização de transações.
- **Invocação de procedimentos remotos** – O Mule ESB está preparado para a invocação de métodos em sistemas remotos, através do Conector HTTP, por exemplo.
- **Sistema de Mensagens** – O próprio Mule ESB é baseado num Sistema de Mensagens e possui suporte nativo para a integração com um *Broker*.

Na Secção 2.2.7 foram referidos alguns dos elementos que caracterizam um Sistema de Mensagens, no contexto da Integração de Aplicações. É possível encontrar na solução da MuleSoft esses mesmos elementos, tais como:

- **Mensagem** – tal como já referido na Secção 3.2, a entidade Mensagem existe no Mule Runtime com uma estrutura bem definida;
- **Canal** – os processadores de mensagens são interligados num Fluxo através de um Canal;
- **Pipes and Filters** – Um Fluxo implementa uma arquitetura *Pipe and Filter*;
- **Encaminhamento de mensagens** – Existem diferentes componentes de Encaminhamento de Mensagens e de controlo de fluxo no Mule Runtime, que implementam os padrões apresentados na Secção 2.3.2:
  - *Content-Based Router* – Implementado pelo componente *Choice Router*, que encaminha as Mensagens no Fluxo tendo por base um conjunto de expressões DataWeave (uma por cada opção de encaminhamento).
  - Filtro de Mensagens – Implementado pelo *Validation Module*, que permite verificar se o conteúdo de cada Mensagem respeita uma dada condição.

  - *Scatter Gather* – Implementado pelo componente *Scatter-Gather Router*, já referido neste Capítulo, na Secção 3.5.3.
  - *Splitter* – Pode ser implementado num Fluxo utilizando os componentes *For Each Scope*, que divide uma Mensagem em vários elementos e permite o processamento individual de cada um, e o componente *Parallel For Each Scope* para o processamento paralelo dos elementos.
  - Agregador – Implementado pelo *Aggregators Module*, que permite fazer a recolha de vários elementos, até que seja satisfeita uma determinada condição, e o processamento do conjunto recolhido. A condição de agregação pode ter em conta o número máximo de elementos no conjunto ou podem ser agrupadas as mensagens por uma dada informação presente no conteúdo.

- **Transformação de Mensagens** – Pode ser feita recorrendo às operações *Set Payload* ou *Transform Message*.

- **Conector** – A noção de Conector está bem presente no Mule ESB.

4

Integração de aplicações num contexto empresarial

Tal como já referido no Capítulo 1, o principal objetivo desta Dissertação de Mestrado é a implementação de uma solução de Integração de Aplicações num ambiente empresarial.

Para o planeamento da solução de integração, é de extrema importância a perceção do modelo de negócio e a definição dos cenários de integração. Para tal, deve-se identificar quais as aplicações que estão em utilização no ambiente e qual é o fluxo de dados que deve ocorrer entre cada uma das aplicações.

4.1 Arquitetura de uma solução de integração

Ao longo desta Dissertação têm sido apresentados, de uma forma geral, os problemas que surgem no âmbito da integração de aplicações. No entanto, pretende-se que os conceitos estudados e apresentados anteriormente sejam aplicados num contexto concreto.

Tal como já referido, existem diferentes formas de estabelecer a comunicação entre duas ou mais aplicações. Tal depende essencialmente das necessidades da solução de integração e da forma com que cada aplicação permite a interação com sistemas externos.

Na Figura 28 encontra-se uma abstração de um cenário de integração, onde estão representados diferentes sistemas conectados entre si, através de um ESB e de uma camada intermédia.

Cada um destes sistemas pode corresponder a uma aplicação num contexto real, cada uma com uma responsabilidade específica. Por exemplo, poderemos ter um sistema para a gestão dos pedidos de clientes, que expõe uma API *REST*; um ERP que permite a invocação remota de procedimentos através de XML-*RPC*; um sistema *Lightweight Directory Access Protocol (LDAP)*; e um portal informativo onde é apresentada informação de outros sistemas.

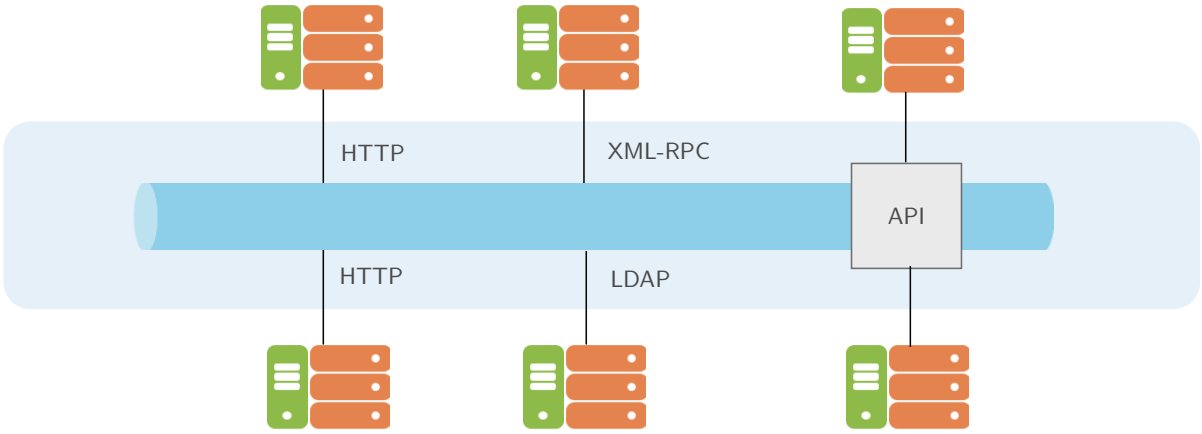


Figura 28: Abstração de um cenário de integração

Embora as aplicações possuam uma responsabilidade distinta, os dados com que lidam podem ser relacionados entre si, por forma a fornecer funcionalidades específicas de apoio ao negócio. Por exemplo, poderá existir a necessidade de fazer a sincronização de dados entre dois ou mais sistemas ou a agregação de informação com diferentes origens. Para além disso, um dos requisitos do portal informativo poderá ser a atualização em tempo da real da informação.

Para tal, é pertinente que a camada intermédia permita a comunicação entre os sistemas; a invocação remota de um conjunto de funcionalidades definidas numa API, para a obtenção de dados e execução de procedimentos; e também a subscrição de eventos para a atualização em tempo real de informação.

4.2 Apresentação do Caso de Estudo

A Eurotux Informática, S.A. é uma empresa especialista no planeamento, integração e implementação de sistemas informáticos, que oferece soluções de tecnologias de informação construídas à medida das necessidades dos clientes.

Um dos pontos de maior relevo no modelo de negócio da empresa está relacionado com o prestação de serviços de suporte aos seus clientes. Para além da gestão dos processos contratuais entre a empresa e os seus clientes, grande parte da informação com que os colaboradores necessitam de lidar tem a ver com a gestão dos pedidos de suporte dos clientes.

A cada um destes pedidos dá-se o nome de *Ticket*, que é registado num sistema informático concebido para o efeito. Nesse sistema, em cada *Ticket* é mantido um histórico das sucessivas interações entre os colaboradores e os clientes e também os vários registos de tempo gasto por parte dos colaboradores.

A par da prestação de serviços de suporte, a empresa possui também uma equipa cujo principal foco é a implementação de Projetos. Um Projeto, antes de ser iniciado, passa por um período de planeamento onde são feitas algumas especificações e definidas as tarefas a desempenhar.

Com o objetivo de prestar sempre um melhor serviço aos seus clientes, a empresa tem investido recursos na análise das interações dos *Tickets*, com vista a tirar conclusões pertinentes relacionadas com os tempos de resposta aos clientes. Para além disso, com o objetivo de agilizar o tratamento dos pedidos, a empresa tem procurado criar ferramentas que auxiliem os seus colaboradores na resolução dos mesmos.

Com esta preocupação e devido ao número de sistemas em utilização na empresa para a gestão dos seus processos internos, tornou-se importante a implementação de uma camada intermédia que permita o tratamento e a disponibilização da informação e que facilite a construção de ferramentas de apoio ao negócio.

De seguida apresenta-se uma descrição dos sistemas de *software* em utilização na empresa e que são de relevância para este Caso de Estudo. Para cada um deles, é feita uma descrição sobre a sua responsabilidade no cenário e sobre os dados com que lida.

- **Odoo** – Desenvolvido pela Odoo S.A., é um Sistemas Integrado de Gestão Empresarial (ERP). Os sistemas desta natureza são de vital importância para a gestão de uma empresa, uma vez que permitem o acesso direto a um fluxo de informação relativo a todos os aspetos do negócio da organização, dividido por várias áreas, como por exemplo, Logística, Recursos Humanos, Projetos e Serviços.

  Neste caso em concreto, este sistema disponibiliza dados sobre os clientes e sobre os colaboradores da empresa; informação relativa à marcação de férias; e ainda informação sobre os Projetos da empresa e as respetivas Tarefas.

- **Request Tracker** – Desenvolvido pela Best Practical, é um sistema de gestão de pedidos de clientes, que mantém um registo sobre todas as alterações num *Ticket*, com todas as interações internas e com clientes e também os registos de tempo gasto na sua execução, desde o estado inicial (Aberto) até ao seu estado final (Resolvido).

- **ETTempos** – Sistema interno da empresa que tem como responsabilidade a agregação dos registos de tempo gasto na execução de tarefas de um modo geral, quer sejam Tarefas de Projetos ou *Tickets* de suporte. Permite obter informação por diferentes parâmetros de pesquisa, quer a nível de clientes, de contratos, de colaboradores ou de equipas.

- **Dashboard** – Ferramenta interna que tem como objetivo agilizar a gestão da equipa de suporte, nomeadamente no processo de atribuição de *Tickets* aos colaboradores e na monitorização das chamadas telefónicas e das interações dos clientes nos *Tickets* abertos. Pretende-se que o Dashboard entre em cooperação com um sistema de *Machine Learning (ML)* que, tendo em conta o

histórico de resoluções de *Tickets* de cada colaborador, seja capaz de, à chegada de um novo *Ticket*, recomendar um colaborador a atribuir e prever o tempo que será necessário para a resolução do mesmo.

- **ERP PRIMAVERA** – ERP desenvolvido pela PRIMAVERA Business Software Solutions que apresenta como principais características a robustez, a fiabilidade, a integridade e a segurança, incluindo um conjunto de módulos totalmente interligados que permitem uma total fluidez de dados entre as áreas Financeira, Logística, Tesouraria, Recursos Humanos, CRM, Ativos, Projetos, entre outras. O ERP PRIMAVERA é um sistema certificado pela Autoridade Tributária e nele pode-se encontrar informação sobre a faturação da empresa.

- **FreeIPA** – Solução integrada e *open source* para a gestão de identidade e de acessos em ambientes Linux/UNIX. Um servidor FreeIPA fornece, de forma centralizada, autenticação, autorização e informações sobre contas de utilizador, grupos, instâncias e outras identidades necessárias à gestão da segurança de uma rede de computadores. É uma combinação de diferentes componentes, como: 389 Directory Server, que fornece uma infraestrutura de LDAP; MIT Kerberos KDC, um sistema de autenticação *Single Sign-On (SSO)*[1]; uma consola de gestão acessível através de uma *web interface*; e também uma ferramenta de Linha de comandos.
  No FreeIPA é possível encontrar informação sobre os utilizadores (colaboradores) que têm acesso às ferramentas da empresa e também sobre as equipas ou departamentos em que cada colaborador de insere.

Estando feito um enquadramento aos sistemas que constituem a infraestrutura de serviços a considerar neste Caso de Estudo e definida a informação que pode ser acedida em cada um deles, podem ser identificados os cenários de integração a implementar. De uma forma geral pretende-se que:

- O **Dashboard de Gestão** seja alimentado, em tempo real, com informação sobre a chegada de novos *Tickets* ao sistema de suporte (o **Request Tracker**), assim como de qualquer interação em cada um dos *Tickets*. Além disso, por forma a auxiliar na atribuição, pretende-se que este sistema apresente quais os colaboradores que estão de férias (**Odoo**) e também as equipas a que pertencem (**FreeIPA**);

- O **ETTempos** seja atualizado periodicamente com os tempos registados pelos colaboradores quer nos *Tickets* de suporte do *Request Tracker*, quer nas Tarefas dos Projetos do **Odoo**;

- Periodicamente sejam geradas no **ERP PRIMAVERA** as faturas relativas aos contratos entre a empresa e os seus clientes, que estão registados no **Odoo**;

1 SSO é um sistema que possibilita aos utilizadores que se autentiquem, de forma segura, em múltiplos sistemas e aplicações, fazendo autenticação apenas uma vez [67].

- Seja exposta uma API que disponibilize informação proveniente dos vários sistemas da empresa, por forma a possibilitar que ferramentas internas como o **Dashboard de Gestão** obtenha informação agregada quer dos *Tickets* de suporte, quer dos colaboradores da empresa.

4.3 Planeamento da solução

Por forma a cumprir com o que é pretendido e tendo em conta aquilo que o Mule Runtime e o Mule SDK fornecem para a integração de aplicações, apresentado no Capítulo 3, fez-se um plano com todos os componentes e configurações necessárias para a implementação da solução.

À primeira vista, tendo em conta o esquema da Figura 29, chega-se à conclusão que é necessário o desenvolvimento de 6 Conectores – um para cada um dos sistemas/serviços que compõe a infraestrutura: ETTempos, Dashboard, Odoo, ERP PRIMAVERA, FreeIPA e Request Tracker. Porém, é necessário ter em conta os Conectores e módulos desenvolvidos pela MuleSoft e pela comunidade e que estão prontos a usar.

O ETTempos, o ERP PRIMAVERA e o Request Tracker fornecem uma API *REST* que pode ser utilizada para a integração com outros sistemas. Para estes, deixa de ser estritamente necessário o desenvolvimento de um Conector, isto porque pode ser utilizado para o efeito o Conector HTTP disponibilizado pela MuleSoft. Como já referido anteriormente, este Conector permite a declaração de clientes HTTP para o consumo de uma API como a que é fornecida pelos referidos sistemas.

Porém, apesar de existência deste Conector, o desenvolvimento de um novo pode fazer sentido pelo facto de permitir Encapsulamento de Operações, reutilização de código e também uma maior facilidade de utilização. Com isto, decidiu-se que deveria ser feito o desenvolvimento de Conectores para o Request Tracker e para o ERP Primavera, uma vez que a informação que está presente nestes sistemas é de uma grande importância para a operação da empresa e uma vez que é grande a probabilidade de no futuro serem desenvolvidas ou adoptadas aplicações que interajam com este sistema, justificando o custo do desenvolvimento do Conector.

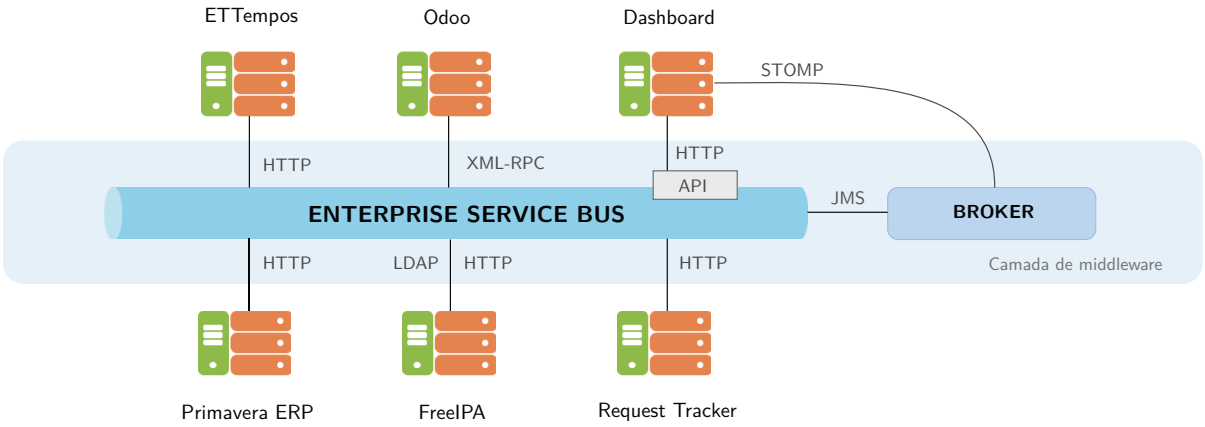


Figura 29: Proposta de solução da Integração

Relativamente ao ETTempos, dado que, no momento da escrita desta Dissertação, o sistema encontrava-se ainda em desenvolvimento, decidiu-se que o Conector poderia ser desenvolvido quando o sistema estiver numa versão estável.

Já no que diz respeito ao FreeIPA, constatou-se que o Mule Runtime está pronto para ser conectado a um sistema LDAP, pelo que, o desenvolvimento do Conector também não foi considerado relevante. Se, no futuro, surgirem novos requisitos que levem à utilização deste sistema para outras funcionalidades, o desenvolvimento do Conector deverá ser novamente considerado.

Em relação ao Odoo, este sistema expõe uma API disponível através de XML-*RPC* e acessível através de diferentes linguagens. Dado que não existe um Conector pronto a usar para fazer invocações a um sistema por XML-*RPC*, tornou-se necessário o desenvolvimento de um novo Conector.

Relativamente ao *Dashboard*, dado que este se trata se uma aplicação de *frontend* em que se pretende a listagem dos *Tickets* de suporte e a listagem dos colaboradores da empresa e das respetivas equipas, foram considerados dois cenários:

- o *Dashboard* consome diretamente as APIs do Request Tracker e do FreeIPA;
- o *Dashboard* consome de uma única API exposta pelo próprio Barramento, ficando da responsabilidade deste obter a informação do Request Tracker e do FreeIPA.

Por uma questão de eliminar as dependências entre as aplicações e de não contrariar os princípios de EAI, decidiu-se optar pela segunda opção. Esta escolha permite também que, por exemplo, a inclusão por parte da empresa de uma nova ferramenta de registo de *tickets* no conjunto de ferramentas utilizadas, não implique alterações no *Dashboard*, concentrando-se o processo de integração na camada intermédia.

Para além da API, um dos requisitos do *Dashboard* é a necessidade de este ser atualizado em tempo real com as alterações realizadas nos *tickets* de suporte, quer pelos colaboradores da empresa quer pelos clientes. Para tal, idealizou-se a utilização de um modelo *Publish-Subscribe*, que pode ser implementado no Mule Runtime utilizando o JMS Conector. Esta implementação é descrita nas próximas Secções deste Capítulo.

Depois de planeada a estrutura básica necessária para este Caso de Estudo e depois de desenvolvidos os Conectores, o próximo passo é a implementação de uma aplicação para o Mule Runtime com a definição de todos os Fluxos necessários para a integração.

4.4 Implementação da solução

Nesta Secção da Dissertação é apresentado o percurso realizado ao longo da implementação da solução de integração apresentada na Secção anterior.

4.4.1 Implementação dos Conectores

Na Secção 3.8 do Capítulo 3 foi já referida qual a estrutura básica de um Conector desenvolvido através do Mule SDK. Pela facilidade de implementação e pela grande liberdade de soluções que a utilização da linguagem Java permite, decidiu-se proceder à implementação dos Conectores utilizando o Java SDK, em detrimento do XML SDK.

Foi utilizado o seguinte comando Maven para gerar um projeto base para o desenvolvimento dos Conectores. O projeto que é gerado possui algumas classes básicas e uma implementação exemplo que segue a estrutura que foi definida anteriormente.

```
$ mvn org.mule.extensions:mule-extensions-archetype-maven-plugin:1.2.0:generate
```

Conector Odoo

Relativamente ao Conector para o Odoo, como já referido, este sistema permite a integração através de uma API via XML-*RPC*. Por forma a facilitar a comunicação, foi utilizada uma biblioteca de Java que permite a invocação dos métodos expostos pelo Odoo. Esta biblioteca chama-se `odoo-java-api` e foi incluída no projeto através da seguinte dependência Maven:

```
<dependency>
    <groupId>com.odoojava</groupId>
    <artifactId>odoo-java-api</artifactId>
    <version>4.0.0</version>
</dependency>
```

Na Figura 41, Anexo A.1, encontra-se um Digrama de Classes, em *Unified Modeling Language (UML)*, que representa a estrutura deste Conector. Por uma questão de organização de código, foram criados três *packages* que não existem no projeto base, nomeadamente: o *package* `connection`, onde se podem encontrar as classes necessárias para a obtenção de um Conexão (`OdooConfiguration`, `OdooConnection` e `OdooConnectionProvider`); o *package* `operations`, onde estão as classes com Operações; e o *package* `sources` onde podem ser encontradas as Fontes do Conector.

Assim como já referido na Secção 3.8, o ponto de entrada de um Conector é a classe que possui a anotação com `@Extension`. No caso do Conector Odoo, a definição pode ser encontrada no Anexo B.3, na classe `OdooExtension`. Nesta classe, é possível encontrar a declaração de um *Connection Provider*, de duas classes de Operações e de uma Fonte.

A biblioteca `odoo-java-api` permite o estabelecimento de um Conexão com um servidor Odoo através da instanciação de um objeto `Session`, que é parametrizado com cinco variáveis: o *host*; a porta

onde o servidor recebe os pedidos XML-*RPC* (por defeito é a 8069); o *username*; a *password*; e o nome da BD da qual se pretende obter informação. Estas cinco variáveis correspondem aos parâmetros da classe de configuração do Conector: a `OdooConfiguration`.

A classe de configuração é usada no *Connection Provider*, mais especificamente na classe `OdooConnectionProvider`, que permite a instanciação de Conexões do tipo `OdooConnection`, e que implementa a *interface* `PoolingConnectionProvider`.

Por forma a encapsular as Operações específicas do Odoo, foi criada a classe `OdooClient` que, através de uma instância da classe `Session`, da biblioteca `odoo-java-api`, permite a invocação dos métodos da API.

Em relação à classe `OdooConnection`, esta recebe no seu construtor a configuração do Conector, que utiliza para instanciar um objeto do tipo `OdooClient`. Tal permite que em qualquer Operação ou Fonte implementada no Conector, seja possível a invocação dos métodos expostos pelo `OdooClient` e pela API do Odoo, através de uma mecanismo de Injeção de Dependências.

É também de referir que a classe `OdooConnectionProvider` implementa o método `validate()` para a validação de uma Conexão ao Odoo, que, neste caso, tem como principal objectivo verificar se:

- o servidor está à escuta de pedidos na porta que consta na configuração;
- as credenciais de acesso (*username* e *password*) são válidas;
- a BD existe.

No que diz respeito às Operações expostas pelo Conector, tal como já referido foi criada a classe `OdooOperations`, onde está declarado um conjunto de Operações, que fazem correspondência com os métodos da API do Odoo e que podem ser consultados de seguida:

- `getDatabaseList()` – para obter uma lista das BDs disponíveis;
- `serverVersion()` – para obter a versão do Odoo que está instalada no servidor;
- `getUserID()` – para obter o `Id` do utilizador autenticado;
- `searchRead()` – permite obter uma lista de registos de um determinado tipo (`objectName`), com possibilidade de pesquisa, ordenação e paginação;
- `read()` – dada uma lista de `Ids`, permite obter os respetivos registos;
- `countRecords()` – permite obter o número de registos que respeitam um critério de pesquisa;
- `create()` – para a criação de um novo registo;
- `write()` – para a alteração de um ou mais registos existentes, dada a lista de `Ids`;

- `unlink()` – para a remoção de registos, dada uma lista de `Ids`.

De salientar que o Odoo possui uma forte componente de extensibilidade, na medida em que permite a implementação de Módulos que introduzem no sistema novos modelos ou que alteram os modelos já existentes. Isto faz com que a API que é exposta pelo sistema tenha que ser genérica o suficiente para suportar as várias alterações que possam ser feitas. Para tal, os métodos da API que lidam com a leitura, criação, edição e remoção de registos são parametrizados com o nome do modelo a utilizar (o `objectName`) e, nos casos das Operações de leitura, com os campos do modelo (*fields*) que se pretende que sejam lidos. Todas essas questões foram refletidas nas Operações do Conector.

Conector RT

Para o sistema de *Tickets* da empresa, o Request Tracker, foi desenvolvido o Conector RT. Em relação à estrutura deste Conector, esta foi pensada para ser um "modelo" (ou *template*) para outros Conectores que a empresa possa desenvolver para a comunicação com uma *Representational State Transfer (REST)* API, sendo que parte dos componentes que foram desenvolvidos para serem reutilizados ou servir de base para a implementação de um novo Conector.

Na Figura 42, Anexo A.2, encontra-se um Diagrama de Classes, em UML, que representa essa estrutura, bem como as principais dependências à API do Mule Runtime que o Conector possui. Este diagrama serve de base para a análise que se segue.

Mais uma vez, na classe com a anotação `@Extension` – a classe `RTExtension` – estão declarados todos os componentes expostos pelos Conector, assim como se pode ver no Anexo B.4. Neste caso, existem os seguintes componentes:

- dois *Connection Providers* – `RTBasicConnectionProvider` e `RTTokenConnectionProvider`;
- uma Configuração – `RTConfig`;
- uma classe de Operações – `RTOperations`;
- uma Fonte – `UpdatedTicketsListener`;

Relativamente à instanciação de uma Conexão, este é um caso em que existem duas formas diferentes de o fazer, dependendo essencialmente do método de autenticação. A API do Request Tracker permite que seja feita a autenticação de duas formas distintas, nomeadamente:

- Através de *Basic Authentication*, onde os pedidos à API contêm no *header Authorization* as credenciais do utilizador (*username* e *password*), codificadas em Base64.

- Através de *Token Authentication*, onde os pedidos à API incluem no *header Authorization* um *token* previamente gerado, sem que haja a necessidade de enviar a *password* do utilizador nos pedidos HTTP.

Dado que existem parâmetros e lógica de negócio em comum entre ambos os tipos de Conexão, tal como é possível ver pelo Diagrama da Figura 42, Anexo A.2, foi implementada uma hierarquia de classes. A primeira, a classe `RTBaseConnectionProvider`, é uma classe abstrata (não podendo ser instanciada), que implementa a *interface* `PoolingConnectionProvider` e que contém o que é comum entre os dois *Connection Providers* deste Conector, nomeadamente:

- Uma Configuração, `RTConfig`, com três parâmetros: o endereço base da API (por exemplo, `https://requesttracker.etux.com/REST/2.0`); um inteiro `maxRetries`; e uma instância de `TlsContextFactory`, da API do Mule, necessária para a configuração do cliente HTTP.
- O método `disconnect()`, para terminar a Conexão;
- O método `validate()` para validar a Conexão, nomeadamente se o servidor onde se encontra a instalação do Request Tracker é acessível pelo Mule Runtime e se as credenciais são válidas.

Em relação à classe `RTTokenConnectionProvider`, que estende a classe `RTBaseConnectionProvider`, possui o *token* de autenticação como parâmetro e implementa o método `connect()`, da *interface* `PoolingConnectionProvider`. Para além disso, neste classe é feita a injeção de um serviço oferecido pelo Mule Runtime – o `HttpService` –, que permite a instanciação de um cliente HTTP.

A classe `RTBasicConnectionProvider`, que também estende a classe `RTBaseConnectionProvider`, possui dois campos adicionais, o *username* e a *password*, implementa o método `connect()` e possui uma instância do `HttpService`.

Anteriormente foi referida a existência de um parâmetro `maxRetries` na classe `RTConfig`. Este parâmetro está relacionado com um problema que foi identificado no Request Tracker, nomeadamente no processo de autenticação. Constatou-se que por vezes por vezes a resposta que é obtida dos pedidos HTTP possui o código de erro 401 ("`Unauthorized`"), mesmo que as credenciais utilizadas sejam válidas. Uma vez que não foi possível identificar a causa do erro, nem obter ajuda por parte da comunidade e dos responsáveis pelo desenvolvimento do Request Tracker, como solução temporária, decidiu-se que o Conector deve repetir todos os pedidos à API que falhem com o código de erro 401, até um número máximo de tentativas (`maxRetries`).

A necessidade da repetição dos pedidos HTTP constitui uma solução temporária, uma vez que diminui o desempenho das Operações do Conector. Porém, a existência deste erro tornou-se numa razão adicional para o desenvolvimento deste Conector e valorizou a sua implementação, na medida em que o Conector permite que o erro identificado seja "encapsulado" nas Operações do ESB, sem que as aplicações em integração tenham que lidar com o mesmo.

Por forma a conseguir uma melhor organização do código e a reutilização das funcionalidades, foi criada uma classe `RTClient`, onde estão implementados os métodos que fazem a invocação da API do Request Tracker e que podem ser reutilizados ao longo do Conector, quer em Operações, quer em Fontes. As invocações à API são feitas recorrendo às classes `RTRequestBuilderFactory` e `RTRequestBuilder` que permitem a construção e envio de pedidos HTTP.

Relativamente às Operações, como se pode ver pelo Diagrama de Classes da Figura 42, Anexo A.2, existe uma classe chamada `RTOperations`, onde se encontram alguns dos métodos de maior relevância e que foram necessários para a implementação do Caso de Estudo, como a obtenção de informação sobre um dado *Ticket*, a pesquisa de *Tickets* por diferentes critérios ou a obtenção do histórico de transações de um *Ticket*.

Em relação à Operação de obtenção do histórico, foram implementadas duas funcionalidades distintas. A primeira está relacionada com o facto de a API fornecer uma listagem paginada das transações do *Ticket* e com o facto de, no presente Caso de Estudo, serem sempre obtidas todas as transações do histórico e nunca apenas uma determinada página. Deste modo, foi criada uma Operação que permite a obtenção de todas as páginas do histórico de transações de um *Ticket*, através da invocação de um conjunto de pedidos.

Ainda sobre a obtenção do histórico de um *Ticket*, dada a necessidade de obter o tempo de trabalho registado pelos colaboradores da empresa, foi adicionada uma funcionalidade, que não é exposta pela API do Request Tracker, que é a obtenção de uma lista de transações de apenas registo de tempo. De notar que o sistema cria transações por diferentes motivos, não só pelo registo de tempo, mas por qualquer alteração que ocorra num *Ticket*. Deste modo, consegue-se evitar que a funcionalidade tenha que ser implementada por todas as aplicações que necessitem de obter esta informação.

Para cada um dos métodos das Operações foi criado um ficheiro com o *output* da Operação no formato *JavaScript Object Notation (JSON) Schema*, para que seja possível tirar partido da funcionalidade de pré-visualização dos resultados referida anteriormente na Secção 3.6 e que pode ser vista na Figura 23. A cada um dos métodos foi adicionada a seguinte anotação: `@OutputJsonType(schema = "metadata/schemas/ticket-schema.json")`.

Há ainda um componente relevante neste Conector que é a Fonte `UpdatedTicketsListener`. Este componente permite a execução de um Fluxo sempre que for detetada uma alteração em qualquer um dos *Tickets* do Request Tracker. A classe que implementa esta funcionalidade é uma *Polling Source* e nela podem ser encontrados os métodos `doStart()`, `doStop()` e `poll()`. No método `doStart()` é instanciada uma `RTConnection`, o que permite a invocação dos métodos implementados na classe `RTClient`.

A principal lógica de negócio desta Fonte encontra-se no método `doPoll()`. Esse método é invocado pelo próprio Mule Runtime, com uma frequência configurável, e tem como responsabilidade obter todos os *Tickets* que foram atualizados no Request Tracker desde a última vez em que o mesmo foi invocado. Para tal, no final de cada execução bem sucedida do método, é guardada numa *Object Store* fornecida pelo Mule Runtime a data da última atualização. Esta *Object Store* permite que o valor seja persistido

mesmo com a ocorrência de uma falha no *runtime* que provoque a sua interrupção. Tendo a lista dos *Tickets* atualizados, a Fonte `UpdatedTicketsListener` faz a "'publicação" de cada um dos *Tickets* no ESB, individualmente.

Todas estas classes dizem respeito à diretoria `main` da estrutura de ficheiros do projeto Maven. Porém, na diretoria `test` foram também implementados alguns casos de teste, utilizando a *framework* de testes para o Mule Runtime – a MUnit.

Esta *framework* permite a definição de testes automáticos para os componentes declarados no Conector, num ambiente semelhante ao de uma Aplicação Mule.

Para o teste, foi necessária a definição de um ficheiro XML idêntico àquele que é escrito numa Aplicação Mule, com a declaração das configurações e dos Fluxos de eventos. Como exemplo, tem-se de seguida o conteúdo do ficheiro que permite o teste da Operação `retrieveTicket`. Nesse ficheiro é possível encontrar a declaração de uma configuração do Conector (uma instância de `RTConfig`), para a criação de uma Conexão por *Basic Authentication* (uma instância da classe `RTBasicConnection`).

```
<mule>
    <rt:config name="rt-config">
        <rt:basic-connection
                username="username"
                password="password"
                apiUrl="https://rt.etux/REST/2.0">
        </rt:basic-connection>
    </rt:config>

    <flow name="retrieveTicketFlow">
        <rt:retrieve-ticket config-ref="rt-config" ticketId="165324"/>
    </flow>
</mule>
```

O próximo passo foi a implementação de uma classe `RTTestCases`, que estende a classe `MuleArtifactFunctionalTestCase` da *framework* de testes, como se pode ver no exemplo:

```
public class RTTestCases extends MuleArtifactFunctionalTestCase {

    @Test
    public void testRetrieveTicket() {
        Event event = flowRunner("retrieveTicketFlow").run();
        Object payloadValue = event.getMessage()
                .getPayload()
                .getValue();
        Assert.assertNotNull(payloadValue);
    }
}
```

Conector ERP Primavera

A estrutura do Conector ERP Primavera segue o modelo que foi implementado no Conector RT. Foi criado um *package* `connection` com as classes `ERPPrimaveraConnection`, `ERPPrimaveraConnectionConfig` e `ERPPrimaveraConnectionProvider`, sendo que na classe de configuração estão declarados os parâmetros que são necessários para a utilização da REST API do ERP Primavera, nomeadamente: a `apiURL` (que corresponde ao *URL* para onde devem ser enviados os pedidos HTTP), o `username`, a `password` e a `company`.

Relativamente às Operações, dada a complexidade do modelo de dados do ERP Primavera, decidiu-se fazer a separação das Operações por diferentes Módulos. Como exemplo, foram implementadas duas classes com Operações num *package* `operations.base`, nomeadamente as classes `ArtigosOperations` e `ClientesOperations`, onde se encontram Operações relativas a Artigos e Clientes, respetivamente.

Tal como no Conector RT, foram criadas as classes `ERPPrimaveraClient`, `ERPPrimaveraRequestBuilderFactory` e `ERPPrimaveraRequestBuilder` para o envio de pedidos HTTP à API do ERP Primavera.

4.4.2 Implementação da Aplicação Mule

Uma Aplicação Mule é o que define os cenários de integração, ou seja, onde são utilizados os componentes oferecidos pelos Conectores para definir toda a lógica de integração, sob a forma de um conjunto de Fluxos.

Tal como referido anteriormente, a implementação de uma Aplicação Mule pode ser feita recorrendo ao IDE Anypoint Studio, onde é possível ter uma visão gráfica das configurações e dos Fluxos definidos.

Assim que se adicionam os Conectores desenvolvidos às dependências do projeto Maven, os mesmos podem ser encontrados numa área designada por *Mule Palette* (Figura 30).

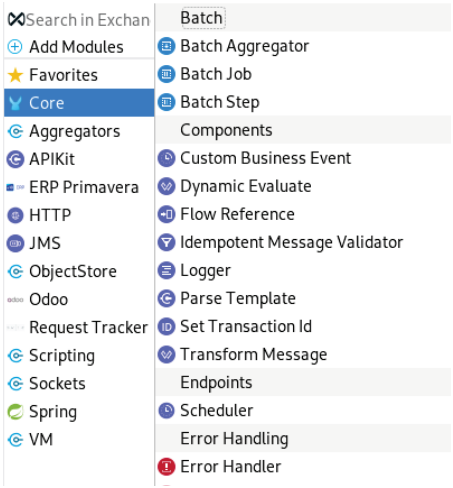


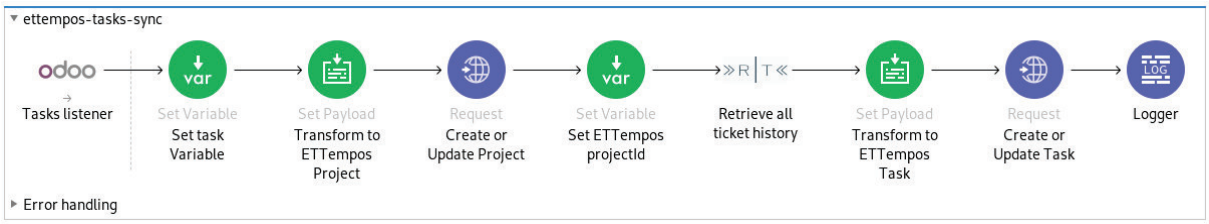
Figura 30: *Mule Palette*

Ao longo desta Secção é apresentada a implementação dos diferentes Fluxos e das configurações necessárias à concretização deste Caso de Estudo.

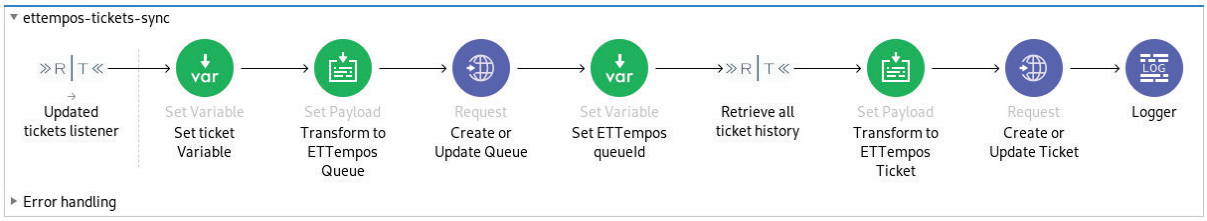
ETTempos e o cálculo do tempo gasto pelos colaboradores

Tal como referido na Secção 4.3, pretende-se que o sistema ETTempos seja alimentado com os registos de tempo gasto pelos colaboradores na execução de Tarefas de projetos e também em *Tickets* de suporte.

Relativamente aos Projetos de Infraestruturas, acontece que, embora as tarefas sejam criadas pelo Gestor de Projetos no sistema Odoo, o registo de tempo gasto por parte colaboradores é feito no Request Tracker, num *Ticket* criado para o efeito. Deste modo, para o cálculo do tempo, não é suficiente a consulta da informação presente no Odoo, mas também no Request Tracker.



(a) Tarefas dos Projetos de Infraestrutura



(b) *Tickets* de suporte

Figura 31: Fluxos de integração do ETTempos com o Request Tracker e o Odoo

Como é possível ver pelo Fluxo da Figura 31a e pelo Anexo B.10, a origem dos eventos do Fluxo é a Fonte `<odoo:tasks-listener />` do Conector do Odoo. Esta Fonte, repetidamente, de x em x tempo (com a frequência é configurável), estabelece uma Conexão com o Odoo e publica um conjunto de t tarefas no ESB.

A cada tarefa publicada (que possui a estrutura apresentada no Anexo B.16), o ESB executa um conjunto de Operações definidas no Fluxo, nomeadamente:

1. Transformar a entidade Projeto num formato compatível com o ETTempos.
2. Invocar o método `POST` em `/projects/project` da API do ETTempos, para a criação ou atualização do Projeto;
3. Obter a partir da Operação `<rt:retrieve-all-ticket-history/>` do Conector RT o histórico de transações do *Ticket*;
4. Transformar a entidade Tarefa num objeto compatível com o modelo de dados do ETTempos;
5. Invocar o método `POST` em `/projects/task` da API do ETTempos, para a criação ou atualização da Tarefa e dos registos de tempo;
6. Imprimir no ficheiro de *log* uma Mensagem de sucesso.

Relativamente aos *Tickets* de suporte presentes no Request Tracker, o Fluxo que permite a integração deste sistema com o ETTempos pode ser visto na Figura 31b e consultado no Anexo B.11. A fonte de Mensagens deste Fluxo é a Fonte `<rt:updated-tickets-listener />` do Conector do RT, que periodicamente obtém a lista dos *Tickets* atualizados e publica-os no ESB.

Para cada um dos *Tickets* (que possuem uma estrutura igual à que pode ser consultada no Anexo B.17), o ESB executa um conjunto de Operações como:

1. Transformar a *Queue* num formato compatível com o ETTempos.
2. Invocar o método `POST` em `/support/queue` da API do ETTempos, para a criação ou atualização da *Queue*;
3. Obter a partir da Operação `<rt:retrieve-all-ticket-history />` do Conector RT o histórico de transações do Ticket;
4. Transformar o *Ticket* num objeto compatível com o formato do ETTempos;
5. Invocar o método `POST` em `/support/ticket` da API do ETTempos, para a criação ou atualização do *Ticket* e dos registos de tempo;
6. Imprimir no ficheiro de *log* uma Mensagem de sucesso.

Definição de uma REST API

Uma das necessidades encontradas para a implementação do Caso de Estudo foi a implementação de uma REST API para facilitar a integração com sistemas que necessitam de consultar e manipular a informação de outros sistemas da empresa, como é o caso do Dashboard.

Para tal, o ESB da MuleSoft possui um Módulo específico – o APIKit. Este Módulo permite que, a partir da definição de um ficheiro *RESTful API Modeling Language (RAML)*, sejam gerados um conjunto de Fluxos para que, através do Conector HTTP, seja possível a comunicação com o ESB por uma API bem definida.

A linguagem RAML [61], baseada na linguagem YAML [68], foi inicialmente proposta em 2013 e teve o apoio da própria MuleSoft e de outras empresas tecnológicas, como a Cisco, a VMWare e o PayPal. Esta linguagem permite uma especificação a diferentes níveis de uma API, como: o protocolo utilizado (HTTP ou HTTPS); o *Media Type*; os mecanismos de segurança; a descrição de todos os métodos da API, bem como os parâmetros e tipos de dados que aceitam e que devolvem na resposta; os possíveis códigos de erro; entre outros aspectos [69].

Neste Caso de Estudo, para a definição da API foram criados quatro ficheiros RAML:

- O ficheiro `esb-api.raml` (Anexo B.18), onde está declarada o nome da API ("Eurotux ESB API), o método de autenticação (*Basic Authentication*) e todos os métodos suportados.

- O ficheiro `data-types.raml` (Anexo B.19), onde estão definidos os tipos de dados utilizados pela API.
- O ficheiro `traits.raml` (Anexo B.20), onde estão declarados os *Traits*, que permitem definir atributos comuns aos métodos HTTP, como se são ou não filtráveis (`filterable`), ordenáveis (`sortable`) e paginàveis (`pageable`).
- O ficheiro `resource-types.raml` (Anexo B.21), onde estão declarados os *Resource Types*, que permitem especificar descrições, métodos e parâmetros genéricos, por forma a facilitar a reutilização de código. Neste caso foi declarado um *Resource Type* `list` a ser aplicado nos métodos que retornam uma lista de um qualquer tipo de dados.

Depois de realizada a especificação da API, foi feita a configuração do Encaminhador do Módulo APIKit. Essa configuração baseia-se na referência do ficheiro base de especificação da API, que neste caso é o ficheiro `esb-api.raml`, e no mapeamento entre cada um dos métodos declarados nessa especificação e o Fluxo que deve ser executado para dar resposta ao pedido HTTP. Esse mapeamento pode ser visto de seguida:

```
<apikit:config name="esb-api-config" raml="esb-api.raml">
   <apikit:flow-mappings >
      <apikit:flow-mapping resource="/tickets/resume" action="get"
                           flow-ref="get:\tickets\resume" />
      <apikit:flow-mapping resource="/tickets" action="get"
                           flow-ref="get:\tickets"/>
      <apikit:flow-mapping resource="/tickets/interactions" action="get"
                           flow-ref="get:\tickets\interactions" />
      <apikit:flow-mapping resource="/Broker/tickets/interactions" action="post"
                           flow-ref="post:\Broker\tickets\interactions" />
      <apikit:flow-mapping resource="/Broker/tickets/interactions" action="get"
                           flow-ref="get:\Broker\tickets\interactions" />
      <apikit:flow-mapping resource="/employees" action="get"
                           flow-ref="get:\employees" />
    </apikit:flow-mappings>
</apikit:config>
```

Esta configuração foi atribuída à Operação APIKit *Router*, que pode ser encontrada no Fluxo da Figura 32a. Dado que a Mensagem de entrada do Fluxo possui os atributos de uma Mensagem HTTP, esta Operação do Módulo APIKit tem como responsabilidade utilizar esses mesmos atributos e o mapeamento anterior para encaminhar a Mensagem para o Fluxo correto.

No mesmo Fluxo da Figura 32a é possível encontrar uma Operação do Conector HTTP para a filtragem das Mensagens através de um mecanismo de segurança. Neste caso, esse mecanismo corresponde à verificação da autenticação dos pedidos HTTP. Dado que a empresa possui um serviço de LDAP, decidiu-se que esse serviço poderia ser utilizado para a autenticação dos pedidos à API do ESB.

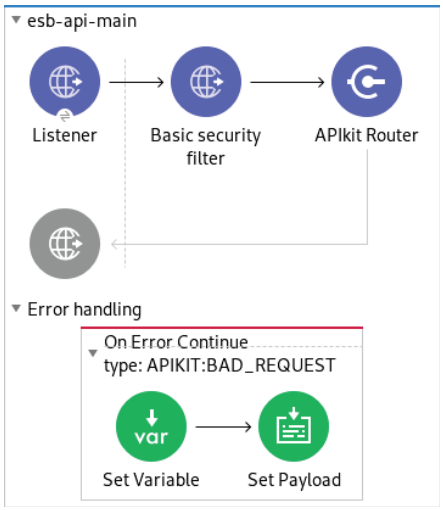
Na solução da MuleSoft, a autenticação dos pedidos HTTP pelo protocolo LDAP pode ser conseguida recorrendo à Spring Framework. Para tal, foi criado um ficheiro `beans.xml` onde pode ser encontrada a declaração de um *Authentication Manager* da Spring Security, que é o responsável por processar um pedido de autenticação através de um ou mais *Authentication Providers*.

Neste caso, foram declarados dois *Authentication Providers*: um que utiliza o LDAP, cujo servidor foi configurado no *Bean* `ldapContextSource`; e um do tipo *User Service*, onde é feita a declaração *inline* dos utilizadores e que tem o objetivo de evitar que a API se torne totalmente dependente do funcionamento do servidor de LDAP.

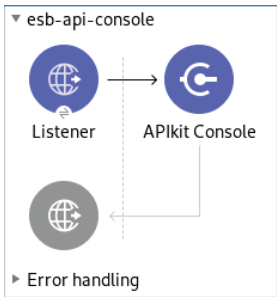
```
<bean id="ldapContextSource"
   class="org.springframework.security.ldap.DefaultSpringSecurityContext">
    <constructor-arg value="ldap://freeipa.etux.pt" />
    <property name="userDn" value="uid=${freeipa.user},cn=users,dc=etux,dc=pt"/>
    <property name="password" value="${freeipa.password}" />
</bean>
```

```
<ss:authentication-manager alias="authManager">
    <ss:ldap-authentication-provider server-ref="ldapContextSource"
        user-search-base="cn=users,dc=etux,dc=pt"
        user-search-filter="(uid={0})" group-search-base="cn=groups,dc=etux,dc=pt"
        group-search-filter="(member={0})" group-role-attribute="cn" />
    <ss:authentication-provider>
        <ss:user-service id="userService">
            <ss:user name="admin" password="{noop}${esb.adminPass} />
        </ss:user-service>
    </ss:authentication-provider>
</ss:authentication-manager>
```

Esta configuração pode ser referida na Operação *Basic Security Filter* (Figura 32a).



(a) Encaminhador



(b) *Console*

Figura 32: Fluxos da REST API do ESB

Na Figura 32b é possível encontrar a declaração de um *endpoint* específico da API. Esse *endpoint* está relacionado com a disponibilização de uma consola interativa que permite a exploração de todos os métodos da REST API e a consulta da documentação criada a partir da linguagem RAML, como os parâmetros necessários em cada método e o tipo de dados que é devolvido nas respostas. Na Figura 33 é possível ver uma captura de ecrã da referida consola.

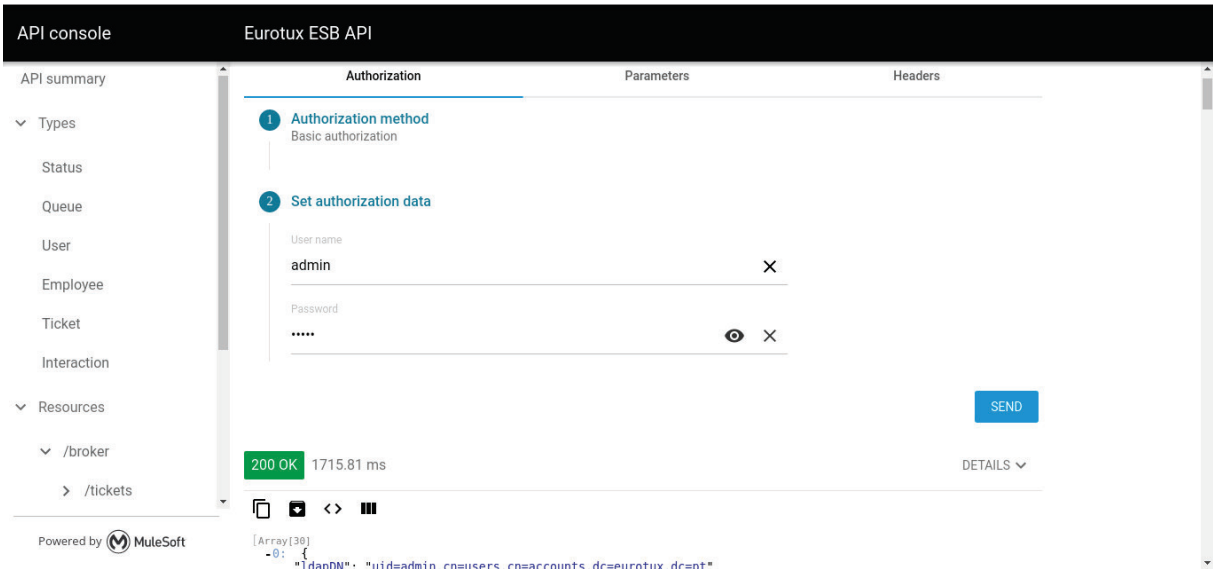


Figura 33: API *Console*

Exemplo de implementação de um método da API

Como exemplo de implementação de um dos métodos da API do ESB, tem-se o método `/tickets/resume`, que permite a obtenção do número de *Tickets* que existem consoante o seu estado (*New*, *Open*, *Stalled*, *Resolved*, *Cancelled*). Esta funcionalidade foi pensada no contexto da implementação do Dashboard, onde se pretende ter uma visão geral sobre indicadores relacionados com os *Tickets*, mas poderá ser utilizada noutro contexto.

Na Figura 34 e no Anexo B.12 encontra-se o Fluxo que permite dar resposta a este método. Dado que é necessário obter o número de *Tickets* em cada um dos estados, a implementação desta funcionalidade envolve a invocação de vários pedidos HTTP ao Request Tracker (um pedido por cada estado).

Por uma questão de generalização e reutilização, decidiu-se que a lista de estados deveria ser enviada como parâmetro no pedido HTTP à API do ESB. A partir dessa lista de estados, o Fluxo passa pela execução de um ciclo, através do componente *For Each Scope*.

Para cada uma dos estados, é utilizada a Operação *Search tickets* do Conector RT, que, através de uma condição de pesquisa (por exemplo, `"Status=New"`), devolve uma estrutura que contém o número total de registos que satisfazem a condição (entre outras coisas). Esse valor é utilizado para construir um novo objeto, como o que se segue: `{ "Status": "New", "Total": 41401 }`.

Neste caso, cada uma das iterações do elemento *For Each* é executada de forma assíncrona, pela utilização do componente *Async*, querendo isto dizer que os pedidos à API do Request Tracker são feitos em paralelo.

A execução assíncrona de cada uma das iterações do ciclo leva à existência de várias *threads* – uma para cada um dos estados enviados como parâmetro. Isto leva a que se ponha em questão a forma como o ESB trata de fazer a agregação de cada um dos resultados, por forma a retornar uma única Mensagem de resposta ao pedido HTTP.

A agregação dos resultados foi conseguida com a utilização do componente *Group-Based Aggregator*, já referido no contexto dos padrões de EAI. Este componente permite a agregação de Mensagens tendo em conta um identificador (`groupId`) e um `groupSize`, que indica o número máximo de elementos que podem ser agrupados no mesmo grupo. Neste caso, esse valor corresponde ao número de elementos no Vetor de estados.

Em relação ao mecanismo de Agregação, este é composto por duas fases: a fase incremental, que é executada sempre que é agrupada uma nova Mensagem; e a fase final que é executada quando o processo de agregação é dado como completo (quando atinge o número máximo de elementos, por exemplo).

Neste caso, uma vez o que agregador é executado no contexto de um *For each*, o próprio *runtime* constrói uma lista com todas as Mensagens que foram processadas pelo agregador, por ordem de chegada. Deste modo, na fase final do processo de agregação, o *runtime* terá construído uma lista em que cada elemento corresponde a um objeto com os campos `"Status"` e `"Total"`.

Tendo já a resposta desejada, é necessário resolver um problema: uma vez que o processo de agregação é assíncrono, o *runtime* continua com a execução do Fluxo sem esperar que a agregação termine. O que, neste caso, não pode acontecer, uma vez que é necessário devolver uma resposta ao pedido HTTP.

A implementação desse mecanismo foi conseguida através da utilização de uma fila de Mensagens (*queue*). Na fase final da agregação, o resultado é publicado numa *queue* chamada `allStatusRetrieved`, através da Operação `<vm:publish />`. Por outro lado, a Operação que o *runtime* executa no fluxo principal, imediatamente a seguir ao ciclo assíncrono, é a Operação `<vm:consume />`. Esta Operação faz com que o *runtime* "tente" ler alguma Mensagem de uma *queue*, até um determinado *timeout*.

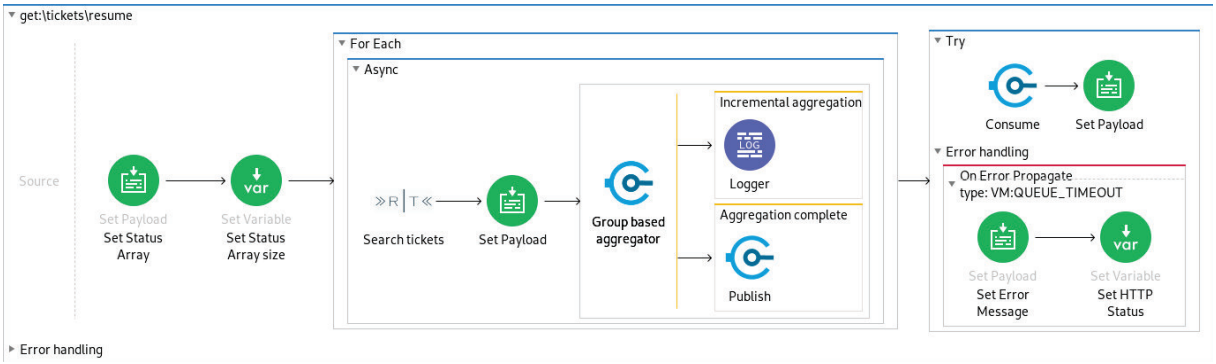


Figura 34: Método `/tickets/resume` da REST API do ESB

Neste Fluxo, tal como é possível verificar no Anexo B.12, foi atribuído um *timeout* de 20 segundos à Operação `<vm:consume />`. Assim, na fase final do processo de agregação, quando é publicada a Mensagem na *queue*, o *runtime* imediatamente lê essa Mensagem e devolve-a na resposta HTTP. Por outro lado, caso o *timeout* seja excedido (o que, em condições normais, não deve acontecer), é propagado um erro do tipo `VM:QUEUE_TIMEOUT`, que faz com que seja devolvida uma Mensagem de erro na resposta HTTP.

Como já referido, a implementação deste método envolve a invocação de vários *endpoints* da API do Request Tracker. Além disso, se, no futuro, a empresa decidir adotar mais uma ferramenta onde seja feito o registo de *Tickets* e se pretender que as referidas métricas incluam a informação dessa nova ferramenta, a implementação do método terá que ser alterada para incluir a nova informação.

Num cenário em que não exista uma camada intermédia como a que se está a criar, ter-se-ia que replicar a implementação desta funcionalidade por todas as aplicações que necessitassem de a usar. Além disso, a adoção de uma nova ferramenta implicaria a alteração de cada uma das aplicações existentes, reforçando assim a importância da existência do ESB.

Utilização do padrão Publish-Subscribe na solução

Tal como referido na Secção 4.3, na Apresentação do Caso de Estudo, um dos pontos principais do *Dashboard* é a apresentação em tempo real das alterações nos *Tickets*.

Uma das possíveis implementações seria a definição de um método na API do ESB que permitisse a listagem das últimas interações, e passar ao Dashboard a responsabilidade de invocar este método com uma frequência suficientemente alta para "simular" uma atualização em tempo real.

Uma solução alternativa e que foi cuidadosamente estudada foi a de utilizar uma abordagem orientada a eventos, seguindo o padrão *Publish-Subscribe*. Neste padrão, com base nos Sistemas de Mensagens e no que foi definido no Estado da Arte, existem três principais entidades: um Produtor, um canal de Mensagens e um ou mais Consumidores. Como é possível ver na Figura 35, o Produtor publica as Mensagens no Canal de Mensagens para que cada um dos Consumidores as possa obter.

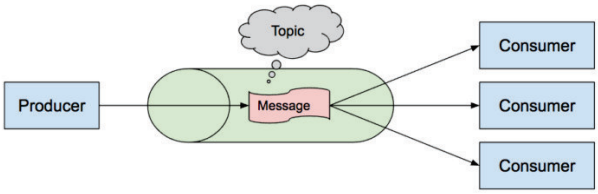


Figura 35: Representação do padrão *Publish-Subscribe* (**Adotado de:** *Uhrig [70]*)

A MuleSoft incorporou no Mule Runtime um Conector que permite a Conexão a um *Broker*, através de JMS. Um dos sistemas que pode ser utilizado para o efeito é o Apache ActiveMQ. No contexto deste *Broker* o Canal de Mensagens é representado por um Tópico, que pode ser subscrito por vários Consumidores.

Posto isto, há duas questões a ter em mente: de que forma é que as interações são publicadas no Tópico e de que forma é que o *Dashboard* as pode receber.

Relativamente à primeira questão, decidiu-se criar um método na API do ESB para permitir a publicação de uma interação num Tópico designado `tickets-interactions`. A ideia é que esse método seja invocado pelo Request Tracker, sempre que nele for registada uma nova interação num *Ticket* (seguindo a ideia do que é um *Webhook*[2]).

A forma como se conseguiu implementar este mecanismo no Request Tracker, foi através do que no sistema se designa por *Scrip*. Uma *Scrip* permite fazer alterações ao comportamento do Request Tracker, através da execução de uma ação definida pelo utilizador, quando uma determinada condição é satisfeita. Neste caso, a condição utilizada foi *On Transaction*, i.e., quando um *Ticket* é atualizado ou alterado de qualquer forma e a ação consiste na invocação do método HTTP da API do ESB.

Com isto, consegue-se que o ESB receba um evento sempre que é feita alguma alteração num *Ticket* no Request Tracker. Para completar, falta apenas criar um Fluxo no ESB que faça a publicação da interação no *Broker*. A implementação desse Fluxo pode ser encontrada no Anexo B.14, onde foi utilizada a Operação `<jms:publish />`, que permite a publicação de uma Mensagem no ActiveMQ.

Estando os eventos a dar entrada no *Broker*, falta conseguir que o *Dashboard* (ou outro consumidor qualquer) consiga obter imediatamente esses eventos ou, neste caso, a interações dos *Tickets*.

O Apache ActiveMQ, assim como outros *Brokers*, permite que seja feita a subscrição de um Tópico a partir de diferentes protocolos, como *Advanced Message Queuing Protocol (AMQP)* [72], *Message Queuing Telemetry Transport (MQTT)* [73], *Streaming Text Oriented Messaging Protocol (STOMP)* [74], entre outros. Dado que o Dashboard é uma aplicação *Web* (desenvolvida na linguagem JavaScript [75], com a *framework* React [76]), decidiu-se utilizar uma biblioteca que permite a utilização do protocolo STOMP através de *WebSockets* [77]. Com essa biblioteca, o *Dashboard* subscreve o Tópico `tickets-interactions` e recebe, em tempo real, as atualizações nos *Tickets*.

O Dashboard, sendo uma aplicação *web*, *client-side*, recebe os eventos do *Broker* apenas quando se encontra em execução no *browser*. Para que cada utilizador consiga visualizar uma lista das últimas interações publicadas, decidiu-se criar um método na API do ESB que disponibilizasse essa informação.

Para tal, foi necessário implementar no ESB um mecanismo de armazenamento das interações, para que possam ser disponibilizadas pelo método da API. Para tal, utilizou-se a *Object Store* fornecida pelo Mule Runtime e a Operação `<os:store />`. Este mecanismo, em vez de ser feito no mesmo Fluxo em que é feita a publicação da interação no *Broker*, foi criado noutro que utiliza a Fonte `<jms:listener />`. Esta Fonte, que permite "ficar à escuta" de eventos que sejam publicados no *Broker*, foi utilizada para despoletar a escrita de cada interação para a *Object Store*. Deste modo, o método HTTP que permite a publicação de uma interação no *Broker* não é "atrasado" por esta escrita. O ESB tornou-se, assim, simultaneamente Produtor e Consumidor do Tópico `tickets-interactions`.

2 Um *Webhook* é um conceito de API em que uma aplicação fornece informação em tempo real a outras aplicações, através de uma *callback* em HTTP. Na prática, a aplicação consumidora regista-se na aplicação produtora com um *Uniform Resource Locator (URL)* para onde é feito um pedido POST sempre que há uma atualização [71].

4.4.3 Monitorização do ESB

A Monitorização é um componente essencial de qualquer arquitetura, para garantir o correto funcionamento dos sistemas. No caso do Mule Runtime, a monitorização, a nível aplicacional, pode ser feita recorrendo a JMX [65] – uma tecnologia Java que fornece ferramentas para o controlo e monitorização de aplicações, objetos de sistema e dispositivos.

Os recursos que podem ser monitorizados por JMX designam-se *Managed Beans*, ou simplesmente *MBeans*, e existe um conjunto diverso desses recursos fornecidos pela própria JVM. Cada um dos *MBeans* fornece diferentes atributos, que podem ser usados para tirar conclusões sobre desempenho, disponibilidade e utilização [65],.

No caso do Mule Runtime, a monitorização JMX pode ser ativada através da configuração do *Wrapper*, no ficheiro `$MULE_HOME/conf/wrapper.conf`. Após a ativação, podem ser feitas *queries* aos diferentes *MBeans* presentes na JVM.

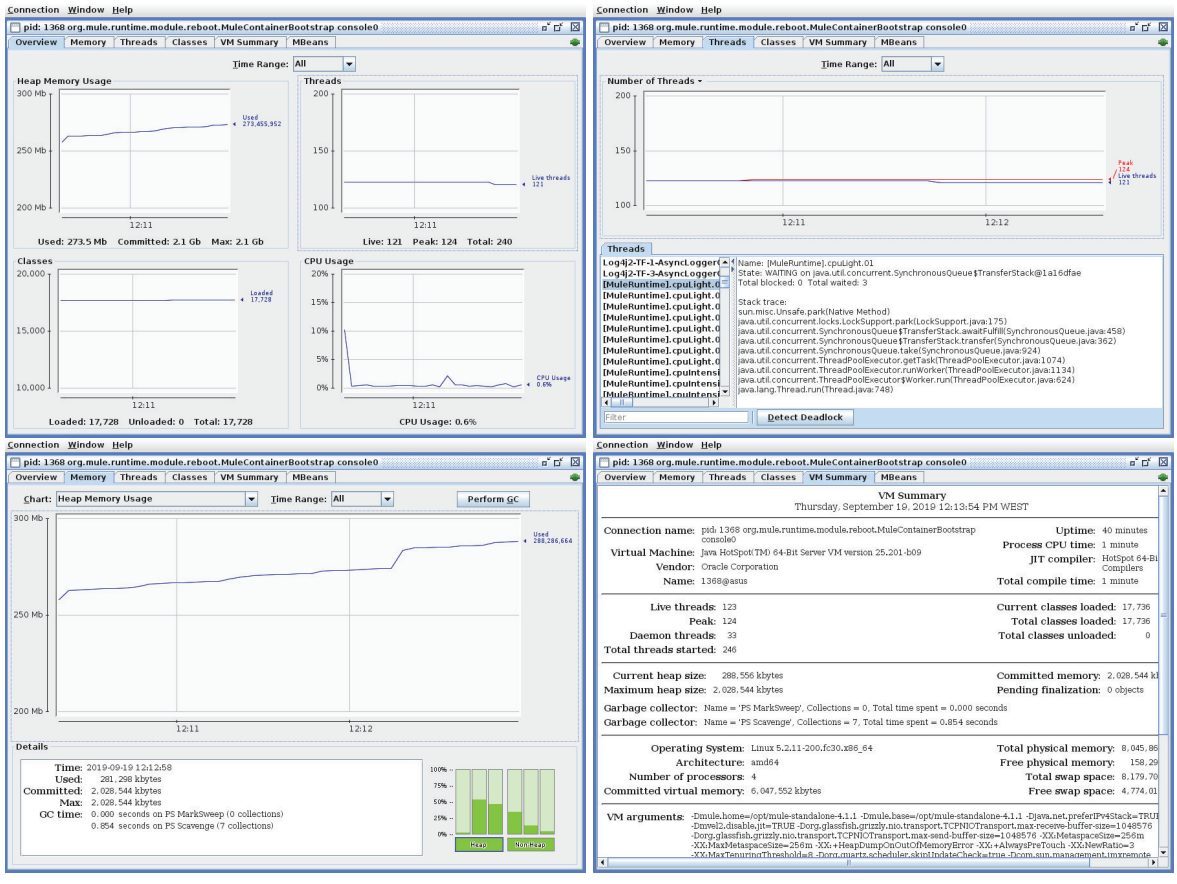


Figura 36: Visualização das métricas JMX no JConsole

Existem diversas ferramentas disponíveis para a visualização dessas métricas, como a JConsole [78] (Figura 36), mas também podem ser utilizadas ferramentas de maior complexidade para a recolha, análise e construção de visualizações, como o conjunto Logstash, ElasticSearch e Kibana [64].

Relativamente à utilização da Elastic Stack neste Caso de Estudo, a mesma foi conseguida através da utilização de um *plugin* para o Logstash, que permite obter métricas JMX de uma aplicação Java remota, com uma determinada frequência (`polling_frequency`).

No Anexo B.23 encontra-se a *Pipeline* do Logstash que foi utilizada e no Anexo B.24 encontra-se o ficheiro de configuração do *plugin* de JMX, onde estão declarados os *MBeans* e respetivos atributos que se pretendem obter da instância do Mule Runtime.

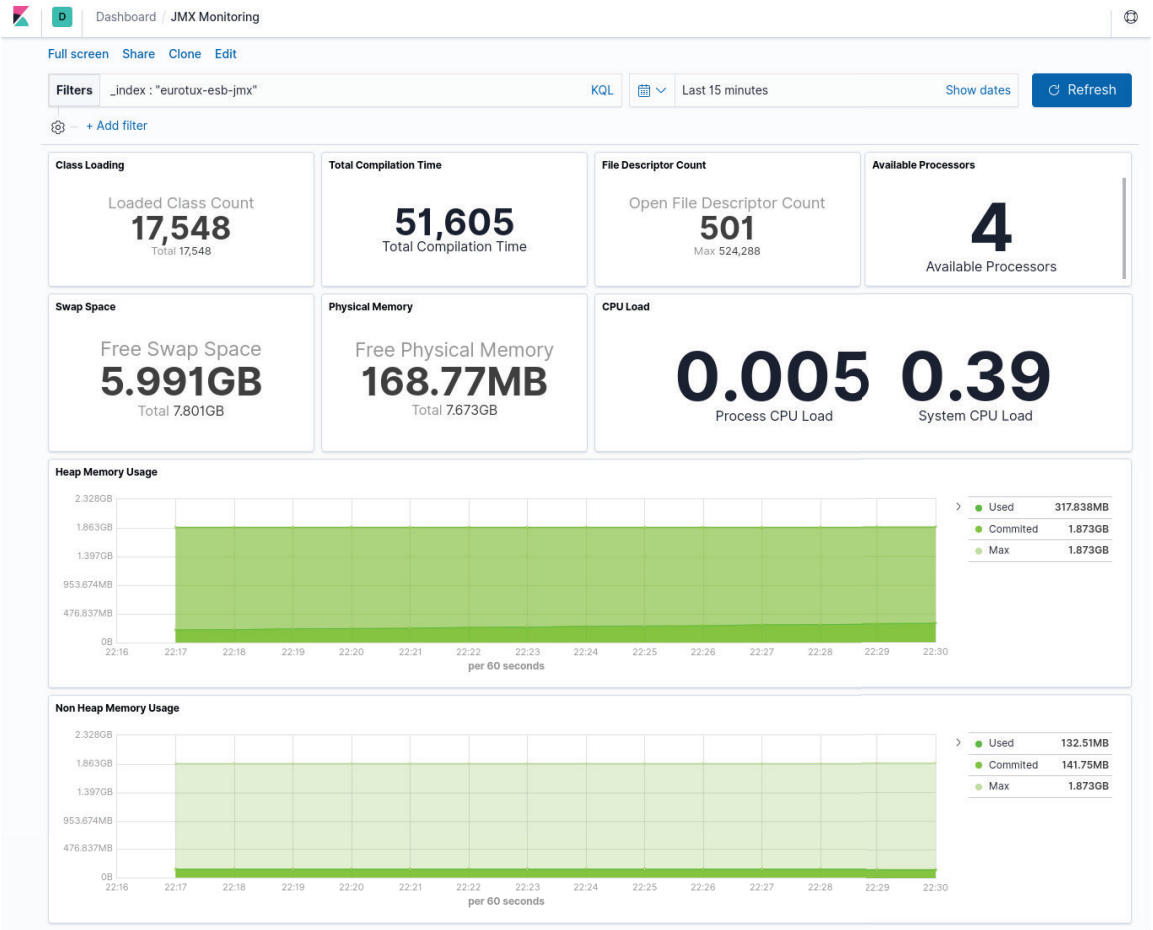


Figura 37: Kibana – *Dashboard* de Visualização com métricas dos *MBeans*

Com a recolha das métricas por parte do Logstash e a respetiva inserção no índice `eurotux-esb-jmx` do ElasticSearch, foi possível proceder à criação de Visualizações e de *Dashboards* no Kibana, para a análise dos diferentes atributos definidos no ficheiro `jmx.conf` (Anexo B.24). Algumas dessas Visualizações podem ser vistas na Figura 37.

Figura 38: Esquema representativo do fluxo de dados da Elastic Stack

Para além da monitorização JMX, a utilização de um conjunto de ferramentas com as capacidades da Elastic Stack permite que sejam recolhidos diferentes tipos de informação, como é o caso dos *Logs* de uma aplicação.

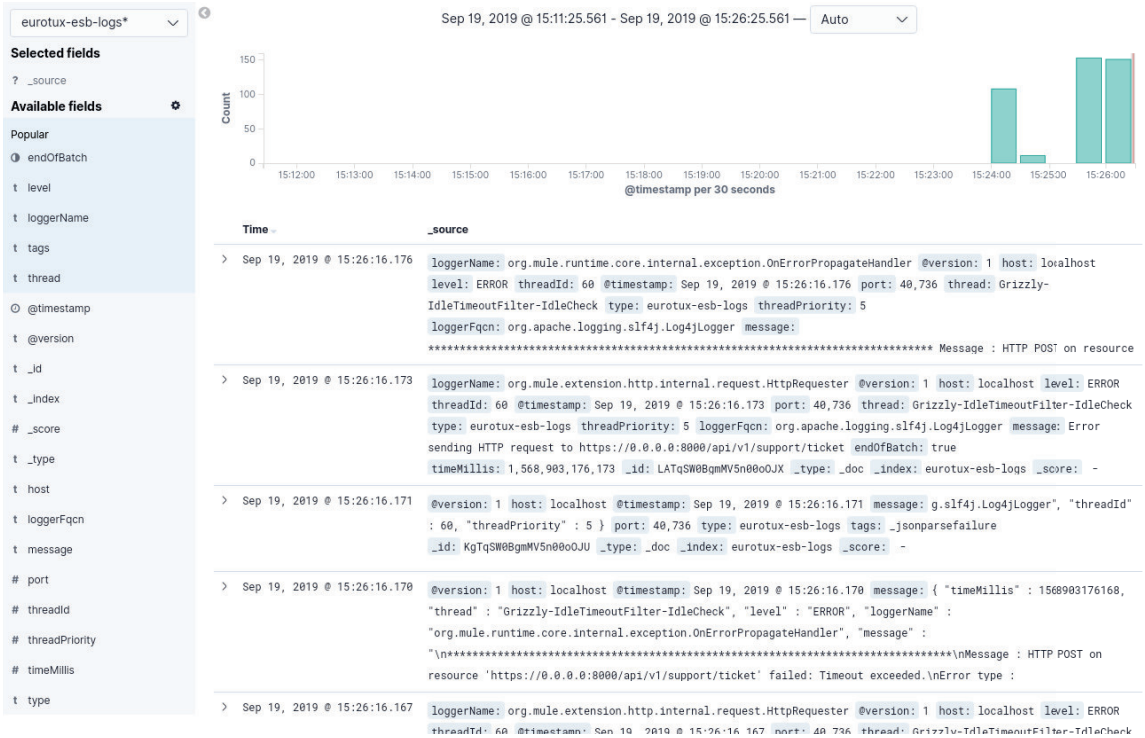


Figura 39: Kibana – Exploração dos últimos registos de *Log*

Nesse sentido, foi também utilizado o *input* `tcp`, que permite a leitura de eventos através de um *Socket* e a respectiva inserção no ElasticSearch. Neste caso, o envio dos eventos para o *Socket* é feito pelo próprio Mule Runtime, através da ferramenta Log4j, e os eventos foram inseridos no índice `eurotux-esb-logs` do ElasticSearch.

Esta implementação torna possível encontrar todos os registos de *Log* do *runtime* no próprio Kibana e fazer Visualizações sobre os mesmos. Além disso, torna possível definir um conjunto de Alertas que podem ser despoletados na ocorrência de erros, por exemplo.

4.4.4 Ambiente de Desenvolvimento e Instalação da solução

Nesta Secção pretende-se apresentar qual foi o Ambiente de Desenvolvimento e de Instalação em que se concebeu para este Caso de Estudo.

Em primeiro lugar, foi utilizado o GitLab como ferramenta de Controlo de Versões, para o desenvolvimento quer dos Conectores como da Aplicação Mule. O GitLab, para além do Controlo de Versões, permite a gestão de *issues*, para o desenvolvimento colaborativo de ideias, resolução de problemas e planeamento de trabalho.

Relativamente ao desenvolvimento do *software* (Conectores e Aplicação Mule), tal como já referido anteriormente, recorreu-se ao Anypoint Studio. Em paralelo com o desenvolvimento, estão os processos de *design* e conceção da solução, onde foram utilizadas as linguagens RAML, para a especificação da API, e a linguagem UML, para a especificação da estrutura dos Conectores.

Como já referido, os Conectores são "importados" para uma Aplicação Mule recorrendo à gestão de dependências do Apache Maven. No que diz respeito aos Conectores e Módulos desenvolvidos pela comunidade e pela MuleSoft, os mesmos encontram-se disponíveis num repositório remoto da MuleSoft, ao qual qualquer programador tem acesso. Porém, relativamente aos Conectores desenvolvidos no contexto deste Caso de Estudo, numa fase inicial, quando se pretendia utilizar o Conector numa Aplicação Mule era feita a instalação para o Repositório Maven local [3].

A dada altura, percebeu-se que a utilização de um Repositório Maven local não se adequa a um contexto de desenvolvimento em equipa, nem facilita a instalação do *runtime* no seu ambiente de execução. Nesse sentido, decidiu-se proceder à instalação de um gestor de Repositórios Maven remoto (o Sonatype Nexus), onde passaram a ser armazenados os artefactos de todos as dependências internas. Tal permite que cada programador participe no desenvolvimento da solução e particularmente num dos componentes (por exemplo, num determinado Conector), sem ter que se preocupar com instalação local de cada uma das dependências.

Depois de instalado o gestor de repositórios, foi feita a configuração de cada uma dos Conectores e da Aplicação Mule para utilizar os repositórios internos, quer para obter as dependências como para fazer *deploy* dos artefactos (através do comando `mvn clean deploy`).

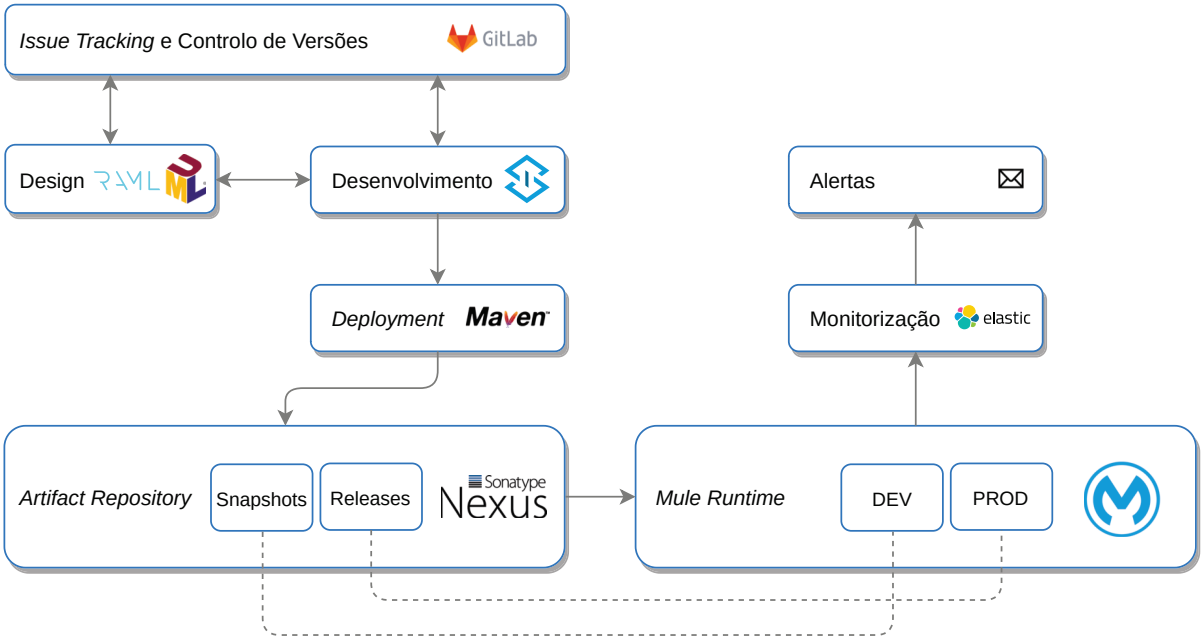


Figura 40: Ambiente de Desenvolvimento e de Instalação

3 O Repositório Maven local é uma diretoria onde todos os artefactos dos projetos são mantidos automaticamente pelo Maven. Geralmente esta diretoria encontra-se em `/.m2` nos sistemas Unix.

5

Conclusão

Neste último Capítulo da Dissertação são apresentadas as considerações finais de todo o trabalho desenvolvido e também sobre os objetivos que foram alcançados. Além disso são definidas algumas melhorias que se consideram importantes para o futuro do projeto.

5.1 Considerações finais

O principal objetivo desta Dissertação de Mestrado foi o estudo dos padrões de EAI e da forma como esses padrões podem ser utilizados para resolver os problemas da existência de diferentes interdependências entre aplicações, num contexto empresarial.

Assim como definido na metodologia DSR, o primeiro passo foi o estudo do Estado da Arte, nomeadamente dos conceitos de Service-Oriented Architecture (SOA), que esteve na origem das primeiras noções de integração de Serviços, de Enterprise Application Integration (EAI) e de Enterprise Service Bus (ESB).

No contexto de uma empresa é comum a utilização de diferentes ferramentas, não só para o fornecimento de serviços aos clientes como também para a gestão dos processos internos. O problema surge quando algumas dessas ferramentas, ou mesmo todas, passam a ter que comunicar entre si para dar resposta às necessidades da empresa ou dos clientes.

As noções de EAI surgiram para dar resposta a esse problema e, ao longo das últimas décadas, tem-se notado uma evolução do mercado com uma crescente disponibilização de soluções de integração, cada vez mais voltadas para a *Cloud*.

Nesta Dissertação e no contexto da Eurotux Informática, S.A., foi adotada uma solução baseada num ESB para implementar a integração de um conjunto de aplicações em utilização na empresa.

Depois de feita uma análise a algumas das soluções disponíveis no mercado, decidiu-se adotar a solução *open source* da MuleSoft, o Mule ESB. Considerou-se esta a escolha acertada, uma vez que permitiu uma implementação relativamente rápida e simples de um conjunto de conectores para as principais ferramentas em utilização na empresa.

Para além da implementação dos conectores, considerou-se o Mule ESB uma solução intuitiva na forma como é desenhada a solução de integração, nomeadamente na utilização dos conectores e das operações para a construção dos Fluxos.

Também o facto de terem sido encontrados vários conectores que permitem a integração do ESB com sistemas bastante utilizados no mercado, como Slack [59], Redis [57], MongoDB [55], Neo4j [56], ElasticSearch [64] e diferentes serviços da Amazon (S3 [52], RDS [79], SQS [80], SNS [54], EC2 [53] e Glacier [81]), trouxe alguma confiança para o potencial da solução e para as vantagens que esta poderá trazer para o futuro da empresa.

Em termos de implementação, foram criados conectores para três sistemas em utilização na empresa: o Request Tracker, que serve de base ao trabalho de toda a equipa de Suporte e que é uma das ferramentas de maior relevância na empresa; o Odoo, sistema onde é feita a gestão da maioria dos processos internos, desde a gestão de Recursos Humanos, processos Comerciais (como propostas e contratos com clientes) e gestão de projetos; e o ERP Primavera, que é utilizado para a gestão da área Financeira e Logística da empresa.

Com o desenvolvimento destes conectores e com a utilização dos Módulos já disponibilizados no Mule ESB, conseguiu-se fazer a integração de três das ferramentas utilizadas na empresa: o ETTempos, uma ferramenta interna para a análise do tempo gasto pelos colaboradores; o Request Tracker, sistema de *Tickets*; e o Odoo. Além disso, o desenvolvimento do conector ERP Primavera, permite que, num futuro próximo, seja feita a integração deste sistema com o Odoo, para a sincronização entre os dois sistemas das faturas emitidas pela empresa.

Foi também implementada uma REST API para a disponibilização de funcionalidades no contexto da empresa, que podem ser utilizadas por outras ferramentas, como foi o caso do Dashboard.

Para além disso, foi feita a integração do ESB da MuleSoft com o Apache ActiveMQ e com o Request Tracker para a disponibilização, em tempo real, das alterações nos *Tickets* de suporte, seguindo um abordagem *Publish-Subscribe*. Esta implementação teve em conta os requisitos do Dashboard, mas poderá ser reutilizada e evoluída para dar resposta a novos requisitos de outras ferramentas da empresa.

Em suma, tendo em conta todo o trabalho desenvolvido, considera-se que foi cumprido o objetivo de fazer abandonar as implementações da topologia Ponto a Ponto para a integração das aplicações.

5.2 Trabalho futuro

Dado o contexto em que esta Dissertação se insere, tendencialmente surgirão novas situações em que se poderá utilizar o Barramento para a integração de diferentes aplicações.

Ainda no decorrer desta fase de implementação, foram surgindo funcionalidades que poderão (e deverão) ser implementadas recorrendo à solução desenvolvida. Exemplo disso é a integração do sistema Odoo com o ERP Primavera, para a sincronização das faturas emitidas pela empresa. Uma vez que foram implementados, no contexto desta Dissertação, conectores para ambos os sistemas, o próximo passo será a implementação de novas Operações (se necessário) e a definição de novos Fluxos.

Apesar de se ter considerado que a solução *open source* do Mule ESB é completa o suficiente para permitir a implementação de uma solução de integração de aplicações, poderia ser interessante para a empresa utilizar a versão *Enterprise* da MuleSoft (a Anypoint Platform), afim de construir uma solução mais avançada.

Neste sentido, considera-se que a maior limitação da versão *Community* (*open source*), comparativamente com a versão *Enterprise*, é a incapacidade de se poder definir um *cluster* de instâncias do Mule Runtime e, assim, construir uma infraestrutura mais resiliente a falhas. Como trabalho futuro, espera-se que seja possível fazer um *upgrade* da solução e assim combater alguns destes problemas.

Bibliografia

[1] Dirk Krafzig, Karl Banke, and Dirk Slama. *Enterprise SOA: Service-Oriented Architecture Best Practices*. Prentice Hall Ptr, 2004. ISBN 978-0131465756.

[2] Sixto Ortiz. Getting on Board the Enterprise Service Bus. *Computer*, 40:15–17, 2007.

[3] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall Professional Technical Reference, Upper Saddle River, NJ, USA, 2005. ISBN 978-0134524450.

[4] Roy Schulte. Predicts 2003: Enterprise Service Buses Emerge. *Gartner Research*, pages 1–6, 2002.

[5] David Chappell. *Enterprise Service Bus*. O'Reilly Media, Inc., 2004. ISBN 0596006756.

[6] Ken Peffers, Tuure Tuunanen, Marcus Rothenberger, and Samir Chatterjee. A Design Science Research Methodology for Information Systems Research. *J. Manage. Inf. Syst.*, 24(3):45–77, Dezembro 2007.

[7] Salvatore T. March and Gerald F. Smith. Design and natural science research on information technology. *Decision Support Systems*, 15(4):251 – 266, 1995.

[8] Nicolai M. Josuttis. *Soa in Practice: The Art of Distributed System Design*. O'Reilly Media, Inc., 1 edition, 2007. ISBN 978-0-596-52955-0.

[9] David S Linthicum. *Enterprise Application Integration*. Addison-Wesley Professional, 2000.

[10] Hao He. What Is Service-Oriented Architecture. https://www.xml.com/pub/a/ws/2003/09/30/soa.html, 2003. [Acedido em 11-12-2018].

[11] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. Web services description language (WSDL) 2. https://www.w3.org/TR/wsdl, 2001.

[12] Yefim V. Natis. Service-Oriented Architecture Scenario. *Gartner Research*, pages 1–6, 2003.

[13] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, Dezembro 1983.

[14] Thomas Erl. *SOA Principles of Service Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007. ISBN 978-0132344821.

[15] Zaigham Mahmood. Service Oriented Architecture: Potential Benefits and Challenges. In *11th WSEAS International Conference on COMPUTERS*, pages 497–501, Agios Nikolaos, Crete Island, Greece, 2007. ISBN 978-960-8457-95-9.

[16] MuleSoft. Understanding Enterprise Application Integration - The Benefits of ESB for EAI. http://www.mulesoft.com/resources/esb/enterprise-application-integration-eai-and-esb, 2015. [Acedido em 02-01-2019].

[17] Gregor Hohpe and Bobby Woolf. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. *Channels*, 4(3):683, Outubro 2004.

[18] Opensource.org. The Open Source Definition (Annotated). http://www.opensource.org/osd-annotated, 2019. [Acedido em 24-07-2019].

[19] NeuronESB. Bus Architecture. https://www.neuronesb.com/neuron/Help3/Architecture/bus_architecture.htm, 2018. [Acedido em 31-12-2018].

[20] Anurag Goel. Enterprise Integration EAI vs. SOA vs. ESB. *Infosys Technologies White Paper*, 87:1–6, 2006.

[21] MuleSoft. MuleSoft Documentation. https://docs.mulesoft.com, 2019. [Acedido em 14-08-2019].

[22] Dejan Risimic. An Integration Strategy for Large Enterprises. *Yugoslav Journal of Operations Research*, 17(2):209–222, 2007.

[23] Ross Mason. Choosing the right integration and ESB platform. https://blogs.mulesoft.com/dev/mule-dev/choosing-the-right-integration-esb-platform, 2011. [Acedido em 25-09-2019].

[24] Chrissy Kidd. What is a Canonical Data Model? CDMs Explained. https://www.bmc.com/blogs/canonical-data-model, Março 2018. [Acedido em 25-09-2019].

[25] Thomas Gulledge. What is integration? *Industrial Management & Data Systems*, 106(1):5–20, Janeiro 2006.

[26] Tariq Rahimsoomro and Abrar Hasnain Awan. Challenges and Future of Enterprise Application Integration. *International Journal of Computer Applications*, 42(7):42–45, 2012.

[27] Edward Curry. Message-Oriented Middleware. In *Middleware for Communications*, pages 1–28. John Wiley & Sons, Ltd, Julho 2005.

[28] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 186–189. ACM, 2002.

[29] Masashi Narumoto, Mike Wasson, and Christopher Bennage. Pipes and Filters pattern. https://docs.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters, 2017. [Acedido em 31-12-2018].

[30] IBM. IBM Integration Bus. https://www.ibm.com/support/knowledgecenter/SSMKHH_10.0.0/com.ibm.etools.msgbroker.helphome.doc/help_home_msgbroker.htm, 2019. [Acedido em 10-01-2019].

[31] Oracle. Oracle Service Bus. https://www.oracle.com/middleware/technologies/service-bus.html, 2018. [Acedido em 10-01-2019].

[32] Microsoft. Microsoft BizTalk ESB Toolkit. https://docs.microsoft.com/en-us/biztalk/esb-toolkit/microsoft-biztalk-esb-toolkit, 2017. [Acedido em 10-01-2019].

[33] WSO2. WSO2 Enterprise Integrator. https://docs.wso2.com/display/EI640, 2018. [Acedido em 10-01-2019].

[34] Red Hat. Red Hat Fuse. https://www.redhat.com/en/technologies/jboss-middleware/fuse, 2019. [Acedido em 10-01-2019].

[35] Falko Menge. Enterprise Service Bus. In *Free and Open Source Software Conference*, pages 1–6, 2007.

[36] Pierre De Leusse, Panos Periorellis, and Paul Watson. Enterprise Service Bus: An overview. Technical report, University of Newcastle upon Tyne, 2007.

[37] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP), 2000.

[38] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL). https://www.w3.org/TR/2001/NOTE-wsdl-20010315, 2001.

[39] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. Java Message Service (JMS). *Sun Microsystems Inc., Santa Clara, CA*, 9, 2002.

[40] Vinton G. Cerf and Robert E. Kahn. A Protocol for Packet Network Intercommunication. *IEEE Transactions on Communications*, 22(5):637–648, Maio 1974.

[41] Jon Postel. User Datagram Protocol. *Isi*, 1980.

[42] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. HyperText Transfer Protocol (HTTP), 1999.

[43] Tim Dierks and Eric Rescorla. RFC 5246 - The Transport Layer Security (TLS) protocol. *Internet Engineering Task Force*, 2008.

[44] Oracle. Java Remote Method Invocation - Distributed Computing for Java. https://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html, 2019. [Acedido em 10-01-2019].

[45] Bruce Jay Nelson. Remote Procedure Call (RPC). 1981.

[46] Oracle. The Java Database Connectivity (JDBC). https://www.oracle.com/technetwork/java/javase/jdbc/index.html, 2018. [Acedido em 10-01-2019].

[47] Jonathan B Postel. Simple Mail Transfer Protocol (SMTP). *Internet Engineering Task Force Request for Comments*, 821, 1982.

[48] John Myers and Marshal Rose. Post Office Protocol (POP3). Technical report, STD 53, RFC 1939, May, 1996.

[49] Jon Postel and Joyce Reynolds. RFC 959: File Transfer Protocol (FTP). *InterNet Network Working Group*, 8:143–157, 1985.

[50] Charles Smulders. Magic Quadrants and MarketScopes: How Gartner Evaluates Vendors Within a Market. http://portal.acm.org/citation.cfm?id=944221, 2011.

[51] Salesforce. CRM Software & Cloud Computing Solutions. https://www.salesforce.com, 2019. [Acedido em 20-10-2019].

[52] Amazon. Cloud Object Storage. https://aws.amazon.com/s3, 2019. [Acedido em 19-10-2019].

[53] Amazon. Amazon EC2. https://aws.amazon.com/ec2, 2019. [Acedido em 19-10-2019].

[54] Amazon. Amazon Simple Notification Service. https://aws.amazon.com/sns, 2019. [Acedido em 19-10-2019].

[55] MongoDB, Inc. MongoDB. https://www.mongodb.com, 2019. [Acedido em 19-10-2019].

[56] Neo4j, Inc. Neo4j Graph Platform. https://neo4j.com, 2019. [Acedido em 19-10-2019].

[57] RedisLabs. Redis. https://redis.io, 2019. [Acedido em 19-10-2019].

[58] ServiceNow. Digital Workflows for Enterprise. https://www.servicenow.com, 2019. [Acedido em 20-10-2019].

[59] Slack. Slack - Where work happens. https://slack.com/intl/en-pt/?eu_nc=1, 2019. [Acedido em 19-10-2019].

[60] Apache. Kafka. https://kafka.apache.org, 2019. [Acedido em 20-10-2019].

[61] RAML. RESTful API Modeling Language. https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md, 2019. [Acedido em 19-10-2019].

[62] Google. Firebase. https://firebase.google.com, 2019. [Acedido em 20-10-2019].

[63] Google. Google Cloud Pub/Sub. https://cloud.google.com/pubsub, 2019. [Acedido em 20-10-2019].

[64] Open Source Search & Analytics - ElasticSearch. https://www.elastic.co, 2019. [Acedido em 26-09-2019].

[65] Oracle. Java Management Extensions (JMX). https://docs.oracle.com/javase/tutorial/jmx/index.html, 2019. [Acedido em 18-09-2019].

[66] Profit. Enrichment Pattern in Mule 4. https://profit-online.pl/2018/07/message-enricher-in-mule-4, 2019. [Acedido em 26-09-2018].

[67] Ivan Milenković, Olja Latinović, and Dejan Simić. Using Kerberos protocol for Single Sign-On in Identity Management Systems. *JITA - Journal of Information Technology and Applications (Banja Luka) - APEIRON*, 5, Junho 2013.

[68] YAML. YAML Ain't Markup Language. https://yaml.org, 2019. [Acedido em 19-10-2019].

[69] Baeldung. Intro to RAML - The RESTful API Modeling Language. https://www.baeldung.com/raml-restful-api-modeling-language-tutorial, 2019. [Acedido em 13-09-2019].

[70] Thomas Uhrig. Queues vs. Topics vs. Virtual Topics (in ActiveMQ). https://tuhrig.de/queues-vs-topics-vs-virtual-topics-in-activemq, 2017. [Acedido em 16-09-2019].

[71] Sengrid. What's a Webhook? https://sendgrid.com/blog/whats-webhook, Junho 2014. [Acedido em 17-09-2019].

[72] S. Vinoski. Advanced Message Queuing Protocol. *IEEE Internet Computing*, 10(6):87–89, Novembro 2006.

[73] Andrew Banks and Rahul Gupta. MQTT. *OASIS standard*, 29:89, 2014.

[74] STOMP. STOMP Protocol Specification. http://stomp.github.io/stomp-specification-1.2.html, 2016. [Acedido em 19-10-2019].

[75] ECMA International. Standard ECMA-262. http://www.ecma-international.org/publications/standards/Ecma-262.htm, 2019. [Acedido em 19-10-2019].

[76] Facebook Open Source. React – A JavaScript library for building user interfaces. https://reactjs.org, 2019. [Acedido em 19-10-2019].

[77] IETF HyBi Working Group. RFC 6455 - The WebSocket Protocol. https://tools.ietf.org/html/rfc6455, 2011. [Acedido em 19-10-2019].

[78] Oracle. Using JConsole - Java SE Monitoring and Management Guide. https://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html, 2018. [Acedido em 18-09-2019].

[79] Amazon. Amazon Relational Database Service. https://aws.amazon.com/rds, 2019. [Acedido em 19-10-2019].

[80] Amazon. Amazon Simple Queue Service. https://aws.amazon.com/sqs, 2019. [Acedido em 19-10-2019].

[81] Amazon. Amazon S3 Glacier & S3 Glacier Deep Archive. https://aws.amazon.com/glacier, 2019. [Acedido em 19-10-2019].

A

Diagramas

A.1 Conector Odoo

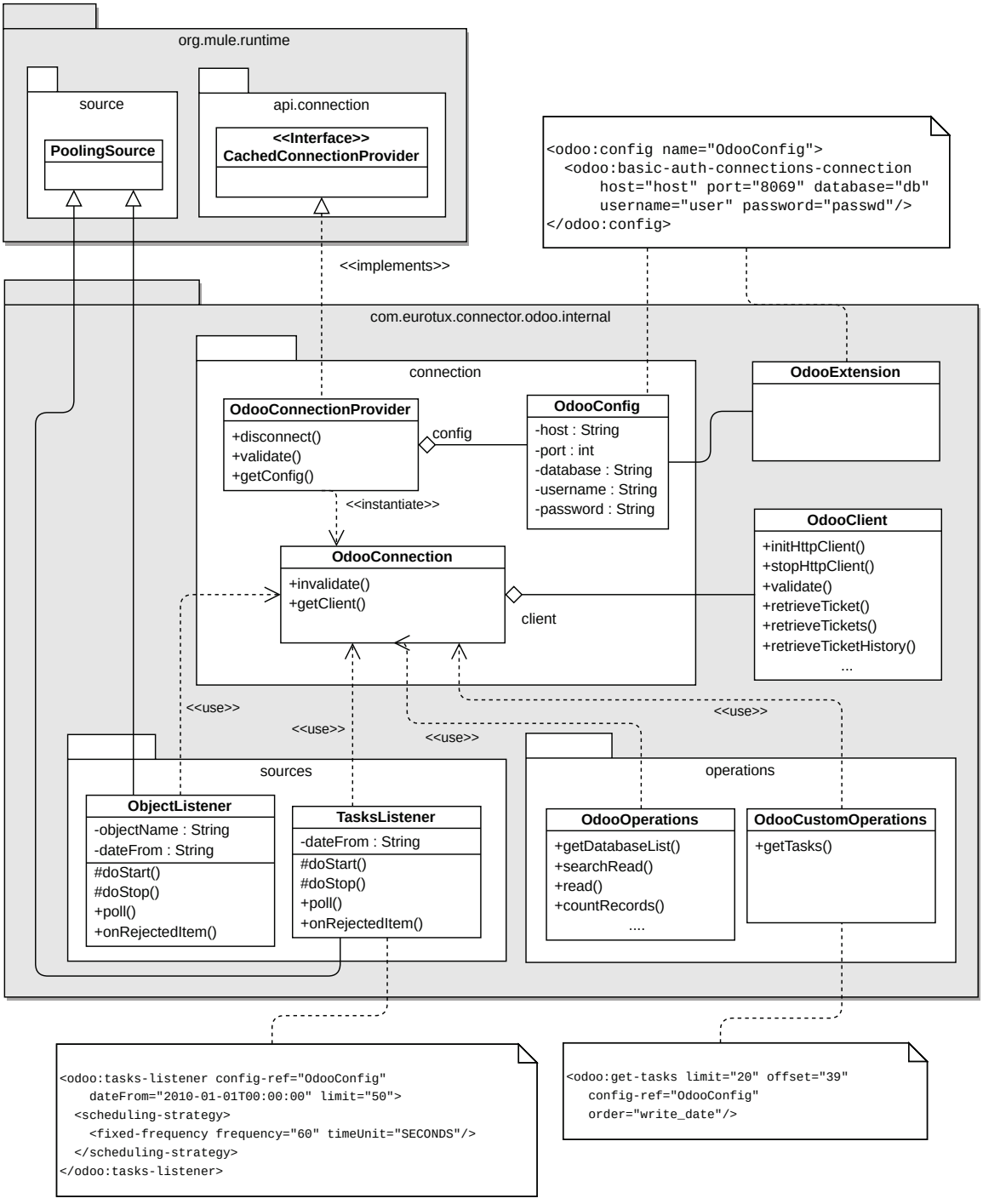


Figura 41: Diagrama de classes do Conector Odoo

A.2 Conector RT

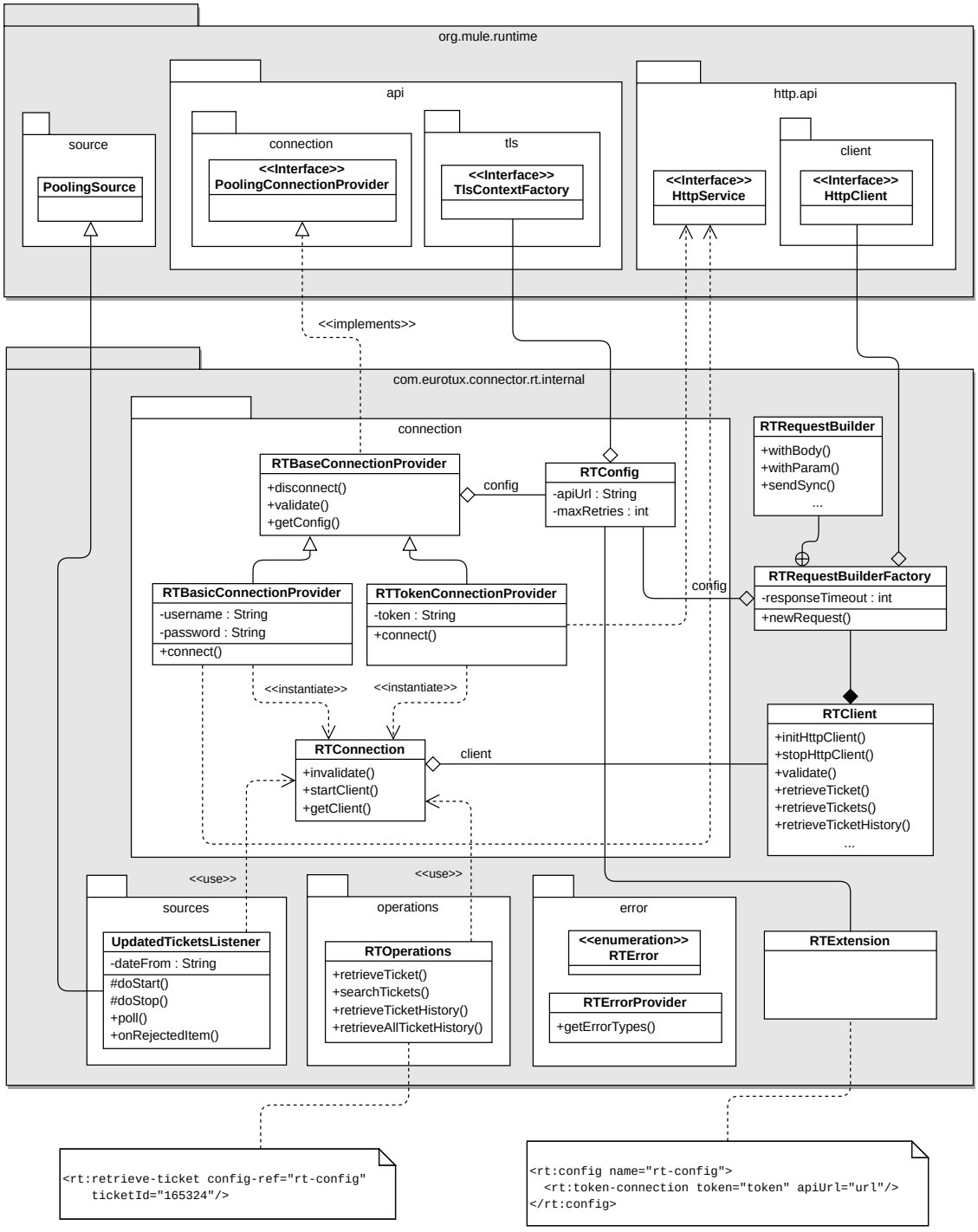


Figura 42: Diagrama de classes do Conector RT

B

Código Fonte

B.1 Exemplo de implementação de uma Fonte

Código Fonte B.1: Implementação da Fonte `HttpListener` no *HTTP Conector*

```
class HttpListener extends Source<InputStream, HttpRequestAttributes> {

    @Config
    private HttpListenerConfig config;

    @Connection
    private ConnectionProvider<HttpServer> serverProvider;

    @Parameter @Optional(defaultValue = "/")
    private String path;

    private HttpServer httpServer;

    @Override
    public void onStart(
        SourceCallback<InputStream, HttpRequestAttributes> sourceCallback
        // Objeto utilizado pela Source para comunicar com o runtime
        // para a publicação de mensagens no Fluxo
    ) throws MuleException {
        // Inicialização dos recursos necessários para a
        // produção de mensagens (como as conexões)
        httpServer = serverProvider.connect();
        server.listen(path).onRequest(request -> {
            processRequest(request, sourceCallback);
        });
    }
```

```
27      @Override
28      public void onStop() {
29          // Lógica necessária para que a Source deixe de produzir
30          // mensagens e para que os recursos alocados sejam libertados.
31          if (httpServer != null) {
32              serverProvider.disconnect(httpServer);
33          }
34      }
35  }
```

B.2 Exemplo de implementação de uma Polling Source

Código Fonte B.2: Implementação da *Polling Source* `FtpDirectoryListener` no *FTP Conector*

```
1   class FtpDirectoryListener extends PollingSource<InputStream,FtpFileAttributes> {
2
3       @Parameter
4       @Optional(defaultValue = "false")
5       private boolean watermarkEnabled = false;
6
7       @Override
8       public void poll(PollContext<InputStream, FtpFileAttributes> pollContext) {
9           if (pollContext.isSourceStopping()) {
10              return;
11          }
12          List<FtpFileAttributes> attributesList = listFilesAttributes();
13          for (FtpFileAttributes attributes : attributesList) {
14              if (pollContext.isSourceStopping()){
15                  break;
16              }
17
18              // O método accept() publica o item no Fluxo
19              PollItemStatus status = pollContext.accept(item -> {
20                  Result<InputStream, FtpFileAttributes> result =
21                      read(attributes.getPath());
22                  // O método setResult() define o payload da mensagem
23                  item.setResult(result);
24                  // O método setId() previne que sejam processados dois
25                  // itens iguais (idempotência)
26                  item.setId(attributes.getPath());
27                  if (watermarkEnabled) {
28                      // O método setWatermark() recebe uma propriedade incremental
29                      // que permite estabelecer ordem nos items, de forma a
30                      // que o runtime descarte os items que tenham uma watermark
31                      // inferior à atual
32                      item.setWatermark(attributes.getTimestamp());
33                  }
34              });
35          }
36      }
37
```

```
38      @Override
39      public void onRejectedItem(
40          Result<InputStream, FtpFileAttributes> result,
41          SourceCallbackContext callbackContext
42      ) {
43          // Executado quando um item é rejeitado (pelo watermark, por exemplo)
44          // Devem ser libertados os recursos associados ao PollItem
45          closeFileStream(result);
46      }
47  }
```

B.3 Conector Odoo

Código Fonte B.3: Classe `OdooExtension`

```
1  @Xml(prefix = "odoo")
2  @Extension(
3      name = "Odoo", vendor = "Eurotux Informática, S.A.", category = COMMUNITY
4  )
5  @ConnectionProviders({OdooConnectionProvider.class})
6  @Operations({OdooOperations.class, OdooCustomOperations.class})
7  @Sources({ObjectListener.class, TasksListener.class})
8  public class OdooExtension {
9  }
```

Código Fonte B.4: Classe `OdooConfiguration`

```
1  public class OdooConfiguration {
2
3      @Parameter
4      private String host;
5
6      @Parameter
7      private int port;
8
9      @Parameter
10     private String database;
11
12     @Parameter
13     private String username;
14
15     @Parameter
16     @Password
17     private String password;
18
19 }
```

Código Fonte B.5: Classe `OdooConnectionProvider`

```
@DisplayName("Basic Authentication Connection")
@Alias("basic-auth-connections")
class OdooConnectionProvider implements PoolingConnectionProvider<OdooConnection> {

    @ParameterGroup(name = "Connection")
    OdooConfiguration config;


    @Override
    public OdooConnection connect() {
        return new OdooConnection(config);
    }

    @Override
    public void disconnect(OdooConnection odooConnection) {
        odooConnection.invalidate();
    }

    @Override
    public ConnectionValidationResult validate(OdooConnection odooConnection) {
        return odooConnection.isConnected() ?
                ConnectionValidationResult.success() :
                ConnectionValidationResult.failure(
                    "Connection Failed", odooConnection.getException()
                );
    }
}
```

B.4 RT Connector

Código Fonte B.6: Classe `RTExtension`

```
@Xml(prefix = "rt")
@Extension(
    name = "Request Tracker", vendor = "Eurotux Informática, S.A.",
    category = COMMUNITY
)
@ConnectionProviders({
    RTBasicConnectionProvider.class, RTTokenConnectionProvider.class
})
@Operations({RTOperations.class})
@Sources({UpdatedTicketsListener.class})
@ErrorTypes(RTError.class)
public class RTExtension {
}
```

Código Fonte B.7: Classe `RTBaseConnectionProvider`

```
abstract class RTBaseConnectionProvider implements PoolingConnectionProvider<RTConnection> {

    @ParameterGroup(name = "Connection")
    private RTConfig config;


    @Override
    public void disconnect(RTConnection connection) { connection.invalidate(); }

    @Override
    public ConnectionValidationResult validate(RTConnection rtConnection) {
        return rtConnection.getClient().validate() ?
                ConnectionValidationResult.success() :
                ConnectionValidationResult.failure(
                    "Connection Failed",
                    new ConnectionException("Invalid Credentials")
                );
    }

}
```

Código Fonte B.8: Classe `RTTokenConnectionProvider`

```
@DisplayName("1 - Token Auth Connection")
@Alias("token-connection")
class RTTokenConnectionProvider extends RTBaseConnectionProvider {

    @Parameter
    private String token;

    @Inject
    private HttpService httpService;


    @Override
    public RTConnection connect() throws ConnectionException {
        RTConnection connection = new RTConnection(
            httpService, getConfig(), token
        );
        connection.startClient();
        return connection;
    }

}
```

Código Fonte B.9: Classe `RTBasicConnectionProvider`

```
@DisplayName("2 - Basic Auth Connection")
@Alias("basic-connection")
class RTBasicConnectionProvider extends RTBaseConnectionProvider {

    @Inject
    private HttpService httpService;

    @Parameter
    private String username;

    @Parameter @Password
    private String password;

    @Override
    public RTConnection connect() throws ConnectionException {
        RTConnection connection = new RTConnection(
            httpService, getConfig(), username, password
        );
        connection.startClient();
        return connection;
    }
}
```

B.5 Fluxos da Aplicação Mule

Código Fonte B.10: Implementação do Fluxo `ettempos-tasks-sync`

```
<flow name="ettempos-tasks-sync">
    <odoo:tasks-listener config-ref="odoo-config"
            dateFrom="2010-01-01T00:00:00" limit="50">
        <scheduling-strategy >
            <fixed-frequency frequency="60" timeUnit="SECONDS"/>
        </scheduling-strategy>
    </odoo:tasks-listener>
    <set-variable value="#[payload]" variableName="task"/>
    <set-payload doc:name="Transform to ETTempos Project" value='#[{
        name: payload.project.name,
        (description: payload.project.description)
            if (payload.project.description != false),
        create_date: payload.project.create_date,
        (client: {
            source: "Odoo",
            external_id: payload.project.partner.id,
            name: payload.project.partner.name
        }) if (payload.project.partner != null),
        (manager: { short_name: payload.project.user.login })
            if (payload.project.user != null),
```

```
        source: "Odoo",
        external_id: payload.project.id,
        company: { name: "Eurotux Informática, S.A." }
    }]'/>
    <http:request method="POST" doc:name="Create or Update Project"
            config-ref="ettempos-config" path="/projects/project"/>
    <set-variable value="#[payload.id as Number]" variableName="projectId"/>
    <rt:retrieve-all-ticket-history config-ref="rt-config"
            ticketId="#[vars.task.ticket_id]" onlyWithTimeTaken="true"
            types='#[["Comment", "Correspond", "Set"]]'/>
    <set-payload doc:name="Transform to ETTempos Task" value='#[{
        subject: vars.task.name,
        description: vars.task.description,
        project: vars.projectId,
        create_date: vars.task.create_date,
        source: "RT",
        external_id: vars.task.ticket_id,
        time_spend_records: payload map ( transaction , index ) -> {
            source: "RT",
            external_id: transaction.id,
            external_url: transaction."_url",
            date: transaction.Created,
            time_worked: transaction.TimeTaken * 60,
            internal_time: transaction."CF.{tempo interno}" * 60,
            employee: { short_name: transaction.Creator.id }
        },
        (assignee: { "short_name": vars.task.user.login })
            if (vars.task.user != null),
    }]'/>
    <http:request method="POST" doc:name="Create or Update Task"
            config-ref="ettempos-config" path="/projects/task"/>
    <logger level="INFO" message="#[payload]"/>
</flow>
```

Código Fonte B.11: Implementação do Fluxo `ettempos-tickets-sync`

```
<flow name="ettempos-tickets-sync">
    <rt:updated-tickets-listener config-ref="rt-config" dateFrom="2010-01-01">
        <scheduling-strategy >
            <fixed-frequency frequency="1" timeUnit="MINUTES" />
        </scheduling-strategy>
    </rt:updated-tickets-listener>
    <set-variable value="#[payload]" variableName="ticket"/>
    <set-payload doc:name="Transform to ETTempos Queue" value='#[{
        name: payload.Queue.Name,
        source: "RT",
        external_id: payload.Queue.id,
        company: { name: "Eurotux Informática, S.A." }
    }]' />
    <http:request method="POST" doc:name="Create or Update Queue"
            config-ref="ettempos-config" path="/support/queue" />
    <set-variable value="#[payload.id as Number]" variableName="queueId"/>
    <rt:retrieve-all-ticket-history config-ref="rt-config"
```

```
            types='#[["Comment", "Correspond", "Set"]]'
            ticketId="#[vars.ticket.id]" onlyWithTimeTaken="true"/>
    <set-payload doc:name="Transform to ETTempos Ticket" value='#[{
        subject: vars.ticket.Subject,
        queue: vars.queueId,
        create_date: vars.ticket.Created,
        source: "RT",
        external_id: vars.ticket.id,
        time_spend_records: payload map ( transaction , index ) -> {
            source: "RT",
            external_id: transaction.id,
            external_url: transaction."_url",
            date: transaction.Created,
            time_worked: transaction.TimeTaken * 60,
            internal_time: transaction."CF.{tempo interno}" * 60,
            employee: { short_name: transaction.Creator.id }
        },
        (assignee: { "short_name": vars.ticket.Owner.id })
            if (vars.ticket.Owner != null),
    }]'/>
    <http:request method="POST" config-ref="ettempos-config"
            path="/support/ticket"/>
    <logger level="INFO" message="#[payload]" />
</flow>
```

Código Fonte B.12: Implementação do Fluxo `get:/tickets/resume`

```
<flow name="get:/tickets/resume">
    <set-payload value="#[(attributes.queryParams.status splitBy ',')]"/>
    <set-variable value="#[sizeOf(payload)]" variableName="arraySize"/>
    <foreach collection="#[payload]">
        <async>
            <rt:search-tickets config-ref="rt-config" page="1" perPage="1"
                    target="rtResponse" ticketSQL="#[''Status='' ++ payload]">

            <set-payload value="#[
                { Status: payload, total: vars.rtResponse.total }
            ]"/>
            <aggregators:group-based-aggregator groupSize="#[vars.arraySize]">
                <aggregators:incremental-aggregation>
                    <logger level="DEBUG" message="payload"/>
                </aggregators:incremental-aggregation>
                <aggregators:aggregation-complete>
                    <vm:publish config-ref="vm-config" queue="all-retrieved"/>
                </aggregators:aggregation-complete>
            </aggregators:group-based-aggregator>
        </async>
    </foreach>
    <try>
        <vm:consume queue="all-retrieved" config-ref="vm-config" timeout="20"/>
        <set-payload value="#[output application/json --- payload]"/>
        <error-handler>
            <on-error-propagate type="VM:QUEUE_TIMEOUT">
```

```
                <set-payload value="#[''An error has occurred.'']" />
                <set-variable value="400" variableName="httpStatus"/>
            </on-error-propagate>
        </error-handler>
    </try>
</flow>
```

Código Fonte B.13: Implementação do Fluxo `get:/employees`

```
<flow name="get:/employees">
    <http:request method="POST" doc:name="FreeIPA Login"
            config-ref="freeipa-http-config" path="/login_password">
        <http:body>#[
            output application/x-www-form-urlencoded
            ---
            { user: p("auth.freeipa.login"), password: p("auth.freeipa.password") }
        ]</http:body>
    </http:request>
    <http:request method="POST" doc:name="FreeIPA user_find"
            config-ref="freeipa-config" path="/json">
        <http:body>#[{
            method: "user_find", params: [ [""], { pkey_only: true, sizelimit: 0 } ]
        }]</http:body>
    </http:request>
    <http:request method="POST" doc:name="FreeIPA batch user_show"
            config-ref="freeipa-http-config" path="/json">
        <http:body>#[{
            method: "batch",
            params: [
                payload.result.result map ((item, index) -> {
                    method: "user_show", params: [ [ item.uid[0] ], { } ]
                })
            ]
        }]></http:body>
    </http:request>
    <set-payload value="#[
        output application/json
        import freeipaUserToEmployee from modules::freeipaTransformers
        ---
        payload.result.results map (
            (result, index) -> freeipaUserToEmployee(result.result)
        )
    ]"/>
</flow>
```

Código Fonte B.14: Implementação do Fluxo `post:/broker/tickets/interactions`

```
<flow name="post:/broker/tickets/interactions">
    <jms:publish doc:name="Publish" config-ref="broker-jms-config"
            destination="tickets-interactions" destinationType="TOPIC"/>
</flow>
```

Código Fonte B.15: Implementação do módulo DataWeave `freeipaUserToEmployee`

```
fun freeipaUserToEmployee
(freeipaUser) = {
    ldapDN: freeipaUser.dn,
    id: freeipaUser.uid[0],
    RealName: trim ((freeipaUser.givenname[0] default "") ++ " " ++
            (freeipaUser.sn[0] default "")),
    EmailAddress: freeipaUser.mail[0],
    Teams: freeipaUser.memberof_group default [],
    IndirectTeams: freeipaUser.memberofindirect_group default []
}
```

B.6 Modelos de dados

Código Fonte B.16: Estrutura de uma *task* do Odoo

```
{
    "id": 835,
    "ticket_id": "134547",
    "create_date": "2019-02-02 11:49:24",
    "write_date": "2019-02-02 12:30:11",
    "name": "Instalação de Etvoip",
    "description": "Instalação e parametrização do sistema operativo...",
    "project": {
        "id": 1,
        "name": "Cliente X: Gravação chamadas",
        "description": "Solução de gravação de chamadas telefónicas...",
        "create_date": "2019-02-02 11:29:32",
        "partner": { "id": 3, "name": "Cliente", "email": "geral@cli.com"},
        "user": { "id": 28, "login": "cba" }
    },
    "user": { "id": 13, "login": "abc@tux.com" }
}
```

Código Fonte B.17: Estrutura de um *Ticket* do Request Tracker

```
{
    "id": 20002,
    "TimeEstimated": 200,
    "Status": "open",
    "Queue": { "id": 20, "Name": "eurotux" },
    "Started": "2019-09-04T09:35:44Z",
    "InitialPriority": 8,
    "Starts": "1970-01-01T00:00:00Z",
    "TimeWorked": 40,
```

```
    "LastUpdated": "2019-09-04T09:35:55Z",
    "Requestor": { "id": "abc", "Name": "Ana Costa", "EmailAddress": "abc@tux.com" },
    "Subject": "Cliente x: Ativação da monitorização 24x7",
    "FinalPriority": 99,
    "TimeLeft": 160,
    "Creator": { "id": "abc", "Name": "Ana Costa", "EmailAddress": "abc@tux.com" },
    "Owner": { "id": "abc", "Name": "Ana Costa", "EmailAddress": "abc@tux.com" },
    "LastUpdatedBy": { "id": "abc", "Name": "João", "EmailAddress": "jc@tux.com" },
    "Resolved": "1970-01-01T00:00:00Z",
    "Created": "2019-08-23T08:44:32Z",
    "Priority": 85,
    "Due": "1970-01-01T00:00:00Z"
}
```

B.7 Definição da API do ESB na linguagem RAML

Código Fonte B.18: Ficheiro `esb-api.raml`

```
#%RAML 1.0
title: Eurotux ESB API
version: 1.0
mediaType: application/json
securitySchemes:
    basic:
        description: This API supports Basic Authentication
    type: Basic Authentication
securedBy: [basic]
uses:
    esbResourceTypes: ./resource-types.raml
    esbDataTypes: ./data-types.raml
    esbTraits: ./traits.raml
/broker:
    /tickets:
        /interactions:
            get:
                responses:
                    200:
                        body:
                            type: esbDataTypes.Interaction[]
            post:
                body:
                    application/json:
                        type: esbDataTypes.Interaction
                responses:
                    200:
                        body:
                            type: esbDataTypes.Interaction
/tickets:
    type:
        esbResourceTypes.list:
            response-type: esbDataTypes.Ticket
```

```
    get:
        is: [esbTraits.sortable, esbTraits.filterable]
        queryParameters:
            filter:
                example: "'Queue'='eurotux' AND 'Owner'='Nobody'"
    /interactions:
        type:
            esbResourceTypes.list:
                response-type: esbDataTypes.Interaction
    /resume:
        get:
            queryParameters:
                status:
                    type: string
                    required: true
            responses:
                200:
                    body:
                        type: object
                        properties:
                            //: number
/employees:
    get:
        responses:
            200:
                body:
                    type: esbDataTypes.Employee
```

Código Fonte B.19: Ficheiro `data-types.raml`

```
#%RAML 1.0 Library
types:
    Status:
        type: string
        enum: ["New", "Open", "Stalled", "Resolved", "Rejected"]
    Queue:
        type: object
        properties:
            id: string
            Name: string
            url: string
    User:
        type: object
        properties:
            id: string
            RealName: string
            EmailAddress: string
    Employee:
        type: User
        properties:
            Team: string
    Ticket:
        type: object
```

```
        properties:
            id: number
            TimeEstimated: number
            Status: Status
            Queue: Queue
            Started: string
            TimeWorked: number
            LastUpdated: string
            Subject: string
            Creator: User
            url: string
            Owner: User
            LastUpdatedBy: User
            Resolved: string
            Created: string
            Priority: number
    Interaction:
        type: object
        properties:
            id: number
            Type: string
            ObjectId: number
            Creator: User
            TimeTaken: number
            Created: string
            OldValue: string
            NewValue: string
            Field: string
            Subject:
                type: string
                required: false
            Description:
                type: string
                required: false
            Ticket:
                type: Ticket
                required: false
```

Código Fonte B.20: Ficheiro `traits.raml`

```
#%RAML 1.0 Library
traits:
    pageable:
        queryParameters:
            page:
                type: integer
                required: false
                default: 1
            per_page:
                type: integer
                required: false
                minimum: 1
                maximum: 100
```

```
                default: 20
    filterable:
        queryParameters:
            filter:
                type: string
                required: true
    sortable:
        queryParameters:
            orderby:
                type: string
                required: false
                example: "Started,id"
            order:
                type: string
                required: false
                example: "ASC,DESC"
```

Código Fonte B.21: Ficheiro `resource-types.raml`

```
#%RAML 1.0 Library
uses:
    esbTraits: ./traits.raml
resourceTypes:
    list:
        usage: Apply this to any resource that returns a collection of members.
        get:
            is: [ esbTraits.pageable ]
            responses:
                200:
                    body:
                        type: object
                        properties:
                            count: number
                            per_page: number
                            page: number
                            items: <<response-type>>[]
```

B.8 Configuração do Repositório Maven remoto

Código Fonte B.22: Configuração dos Repositórios Maven remotos no ficheiro `pom.xml`

```
<pom>
    <repositories>
        <repository>
            <id>nexus</id>
            <name>Eurotux Maven Repository</name>
            <url>${eurotux.maven.repo}/repository/maven-public</url>
        </repository>
    </repositories>
```

```
    <build>
        <pluginManagement>
            <plugins>
                <plugin>
                    <groupId>org.sonatype.plugins</groupId>
                    <artifactId>nexus-staging-maven-plugin</artifactId>
                    <executions>
                        <execution>
                            <id>default-deploy</id>
                            <phase>deploy</phase>
                            <goals><goal>deploy</goal></goals>
                        </execution>
                    </executions>
                    <configuration>
                        <serverId>nexus</serverId>
                        <nexusUrl>maven.etux.com/nexus/</nexusUrl>
                        <skipStaging>true</skipStaging>
                    </configuration>
                </plugin>
            </plugins>
        </pluginManagement>
    </build>
    <distributionManagement>
        <repository>
            <id>nexus</id>
            <name>Releases</name>
            <url>maven.etux.com/repository/maven-releases</url>
        </repository>
        <snapshotRepository>
            <id>nexus</id>
            <name>Snapshot</name>
            <url>maven.etux.com/repository/maven-snapshots</url>
        </snapshotRepository>
    </distributionManagement>
</pom>
```

B.9 Configuração da Monitorização

Código Fonte B.23: *Pipeline* do Logstash para recolha de atributos JMX e de *Logs*

```
input {
    tcp {
        port => 5000
        codec => json
        type => "eurotux-esb-logs"
    }
    jmx {
        path => "/monitor/jmx"
        polling_frequency => 30
        type => "eurotux-esb-jmx"
        nb_thread => 3
```

```
    }
}
filter {
    if [type] == "eurotux-esb-jmx" {
        ruby {
            code => '
                p = event.get("metric_path")
                vn = event.get("metric_value_number")
                vs = event.get("metric_value_string")
                if vn
                    event.set(p, vn)
                else
                    event.set(p, vs)
                end
            '
        }
        de_dot {
            nested => true
        }
        aggregate {
            task_id => "%{path}"
            code => "
                class ::Hash
                    def deep_merge(second)
                        merger = proc { |key, v1, v2| Hash === v1 && Hash === v2 ?
                            v1.merge(v2, &merger) : v2
                        }
                        self.merge(second, &merger)
                    end
                end
                map[:jmx] ||= Hash.new
                map[:jmx] = map[:jmx].deep_merge(event.get('jmx'))
                map[:type] = event.get('type')
                event.cancel()
            "
            push_map_as_event_on_timeout => true
            timeout => 29
        }
        mutate {
            remove_field => [
                "metric_path", "metric_value_number",
                "metric_value_string", "path"
            ]
        }
    }
}
output {
    elasticsearch {
        hosts => "localhost:9200"
        user => "elastic"
        password => "changeme"
        index => "%{type}"
    }
    stdout { codec => rubydebug }
}
```

Código Fonte B.24: Ficheiro de configuração do *plugin* JMX do Logstash

```
{
    "host": "localhost",
    "port": 1096,
    "alias": "jmx",
    "queries": [{
        "object_name": "java.lang:type=ClassLoading",
        "object_alias": "ClassLoading",
        "attributes": [
            "LoadedClassCount", "TotalLoadedClassCount",
            "UnloadedClassCount"
        ]
    }, {
        "object_name": "java.lang:type=Memory",
        "object_alias": "Memory",
        "attributes" : [ "HeapMemoryUsage", "NonHeapMemoryUsage" ]
    }, {
        "object_name": "java.lang:type=Compilation",
        "object_alias": "Compilation",
        "attributes": [ "TotalCompilationTime" ]
    }, {
        "object_name": "java.lang:type=OperatingSystem",
        "object_alias": "OperatingSystem",
        "attributes": [
            "SystemLoadAverage", "ProcessCpuTime",
            "AvailableProcessors", "SystemCpuLoad",
            "ProcessCpuLoad", "CommittedVirtualMemorySize",
            "FreeSwapSpaceSize", "TotalSwapSpaceSize",
            "FreePhysicalMemorySize", "TotalPhysicalMemorySize",
            "MaxFileDescriptorCount", "OpenFileDescriptorCount"
        ]
    }, {
        "object_name": "java.lang:type=Runtime",
        "object_alias": "Runtime",
        "attributes": [ "StartTime", "Uptime" ]
    }, {
        "object_name": "java.lang:type=Threading",
        "object_alias": "Threading",
        "attributes": [
            "TotalStartedThreadCount", "ThreadCount",
            "PeakThreadCount", "DaemonThreadCount",
            "CurrentThreadUserTime", "CurrentThreadCpuTime"
        ]
    }]
}
```