INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Event-driven integrado com Enterprise Service Bus

CARLOS MIGUEL COSTA DA SILVA
Outubro de 2018

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Event-driven integrado com Enterprise Service Bus

Carlos Miguel Costa da Silva

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Engenharia de Software

Orientador: Paulo Maio

Supervisor: Pedro Pinto

Porto, Outubro 2018

Dedicatória

Aos meus pais. As duas pessoas mais importantes na minha vida. Sem eles, isto não era possível. Um obrigado do fundo do meu coração.

Resumo

Nos tempos que correm, torna-se fundamental para uma organização que desenvolve produtos de *software* manter o seu sistema atualizado e preparado para reagir ao mercado competitivo onde esta se insere. A constante mudança deste mercado faz com que as organizações necessitem de se adaptarem rapidamente a tudo o que vai sendo modificado, implementando soluções cada vez mais flexíveis, com capacidade de escalar significativamente e reduzindo os seus custos de manutenção e de desenvolvimento de novas funcionalidades.

A Glintt-HS é, portanto, uma organização que precisa de se adaptar às necessidades dos seus clientes e das constantes mudanças que vão surgindo no seu negócio, sendo por isso necessário estudar e desenvolver uma solução capaz de satisfazer as suas necessidades.

O objetivo deste trabalho é estudar e desenvolver uma solução capaz de satisfazer as necessidades da Glintt-HS e do mercado onde esta se insere, desenvolver uma solução de alto desempenho, com capacidades para crescer e ser modificada de uma maneira simples e com um baixo custo.

Palavras-chave: ESB, Eventos, Arquitetura, Integrações, Micro Serviços

Abstract

In these times, it becomes critical for an organization within the software area to keep its system up to date and prepared to react to the competitive marketplace where it fits. The constant change in this market means that organizations need to adapt quickly to everything that is being modified, implementing increasingly flexible solutions, with the capacity to scale significantly and reduce the costs of developing new functionalities and time spent in maintenance.

Glintt-HS is therefore an organization that needs to adapt to the needs of its customers and the constant changes that are emerging on the company business, so it is necessary to study and develop a solution that can meet the organization needs.

The objective of this work is to study and develop a solution capable of satisfying the needs of Glintt-HS and the market where it operates, developing a high performance solution with the ability to grow and be modified in a simple and low cost way.

Keywords: ESB, Events, Architecture, Integrations, Microservices

Agradecimentos

Em primeiro lugar, gostaria de prestar um agradecimento especial aos meus pais. O seu apoio foi incondicional e forneceram-me todas as condições necessárias para eu redigir este documento, completando assim todo o meu percurso académico.

Um obrigado ao meu orientador, o engenheiro Paulo Maio, por toda a ajuda prestada na elaboração deste documento, demonstrando-se sempre bastante disponível e com críticas muito construtivas para o desenvolvimento do trabalho.

Agradeço à Glintt por me ter ajudado a crescer como profissional e por me ter dado a oportunidade de analisar e aprender novas metodologias de desenvolvimento de *software*. Um especial obrigado aos engenheiros, Pedro Rocha, Pedro Pinto e José Melo por me terem ajudado no desenvolvimento desta dissertação, estando sempre disponíveis para debater novas ideias.

A todos os meus companheiros do ISEP, principalmente ao Paulo Pereira, Francisco Ramalho, José Cabeda e Sérgio Gomes, que desde o primeiro dia de estudos no ISEP até ao último foram uma grande força para conseguir escrever esta dissertação.

À minha namorada por todo o apoio e compreensão que demonstrou durante todo o meu percurso académico, estando sempre presente nas situações mais complicadas.

Por último, mas não menos importante, gostaria de agradecer à Instituição Superior de Engenharia do Porto (ISEP), especialmente ao Departamento de Engenharia Informática (DEI), pela qualidade de ensino, e também pelo apoio e disponibilidade do corpo docentes, que ajudaram na minha concretização profissional.

Um muito obrigado a todos!

Índice

Lista de Figuras

Lista de Tabelas

Acrónimos e Símbolos

Lista de Acrónimos

ESB	Enterprise Service Bus
CQRS	Command Query Responsibility Segregation
NCD	New Concept Development Model
REST	Representational State Transfer
XML	Extensible Markup Language
MB	Message Broker
IM	Identity Server
CEP	Complex Event Processor
DAS	Data Analytics Server
G-Reg	Governance Registry
JMS	Java Message Service
JSON	JavaScript Object Notation
AMQP	Advanced Message Queuing Protocol
RAML	RESTful API Modeling Language
JMX	Java Management Extensions
HTTP	Hypertext Transfer Protocol
API	Application Programming Interface
TCP	Transmission Control Protocol
SOA	Service Oriented Arquitechture
JDBC	Java Database Connectivity

1 Introdução

O trabalho desenvolvido nesta dissertação e no âmbito do Mestrado em Engenharia Informática (MEI-ISEP) assenta no estudo e desenvolvimento de uma nova arquitetura de software para tornar o sistema da Glintt-HS mais competitivo e atual. Neste capítulo é realizada uma contextualização do problema, seguida de uma sucinta descrição do problema e dos objetivos a alcançar, terminando com a respetiva abordagem a seguir para os atingir.

1.1 Contexto

Este trabalho apresenta uma solução implementada na *Glintt Healthcare Solutions* (Glintt-HS[1]), que faz parte do grupo *Global Intelligent Tecnologies* (Glintt[2]). Este é um grupo que representa uma empresa Portuguesa multinacional cotada na *Euronext* Lisboa. A Glintt-HS é uma empresa que opera na área da saúde e possui um grande leque de aplicações para responder às necessidades de todos os seus clientes. Possui uma plataforma denominada *GPlatform* que é onde se encontram grande parte dos serviços que a organização fornece. Esta plataforma possui uma arquitetura orientada a serviços (SOA, *Service-Oriented Architecture*) operando segundo o critério de pedido-resposta, ou seja, uma chamada a um serviço web desta plataforma é interpretado como um pedido, que por sua vez terá a respetiva resposta de acordo com a lógica associada a esse serviço.

Atualmente, é muito comum serem praticadas arquiteturas orientadas a serviços, caracterizadas pelo seu desempenho, escalabilidade e flexibilidade, pelo que a Glintt-HS conseguiu adaptar-se muito bem ao mercado através da sua arquitetura. Estas três características podem ser definidas da seguinte maneira:

- **Desempenho:** capacidade de o sistema responder às solicitações recebidas de uma forma rápida e consumindo poucos recursos do servidor (i.e., CPU, RAM, etc.) (Smith & Williams, 2001);

- **Escalabilidade:** capacidade de um sistema crescer e de gerir o aumento de carga. Pode ser descrita por um modelo de escalabilidade de 3 dimensões, conhecido por cubo de escalabilidade (Martin L. Abbott, 2015). Segundo o cubo de escalabilidade existem 3 eixos que representam as 3 dimensões: o eixo X, onde o sistema é duplicado horizontalmente, clonando o sistema; o eixo Y, onde o sistema é decomposto funcionalmente, dividindo o sistema através de conceitos distintos; o eixo Z, onde o sistema é divido através da partição de dados;

[1] https://www.glintt.com/pt/o-que-fazemos/mercados/healthcare/Paginas/Home.aspx
[2] https://www.glintt.com

- **Flexibilidade:** capacidade de um sistema evoluir facilmente devido a mudanças do negócio (internas ou externas) de modo a responder a essas mudanças com custos de tempo reduzidos (Truren, 2010);

Event-driven é um conceito que já é usado há algum tempo e pode ser observado ao longo da história da computação, desde a gestão de exceções em linguagens de programação, passando por conceitos e disciplinas como: sistemas de gestão, gestão de atividades de negócio, bases de dados ativas (Widom & Ceri, 1996) e sistemas de *publish & subscribe.* Recentemente tem existido um aumento do interesse por estes sistemas (Etzion, 2005). De todos os tipos de sistemas orientados a eventos salientam-se os sistemas de *publish & subscribe* que são arquiteturas orientadas, também a serviços (SOA), mas sendo caracterizadas principalmente pelo consumo e publicação de eventos. Estas mensagens podem ser definidas como os dados relacionados com evento espoletado por um produtor de eventos e que também será consumido por um consumidor de eventos. Arquiteturas orientadas a eventos, tornam uma chamada a um serviço web numa espécie de notificação que irá desencadear uma série de eventos, libertando a lógica dos serviços web, tornando os sistemas que praticam estas arquiteturas ainda mais escaláveis e flexíveis. Com este tipo de arquiteturas um sistema apenas se tem de preocupar com uma determinada tarefa e informar que tal tarefa ocorreu. A comunicação entre outros sistemas é realizada através destes eventos que podem ser geridos por um sistema de filas de mensagens, sendo os eventos consumidos pelos respetivos subscritores.

Acresce que cada vez mais existe a necessidade de integrar novos sistemas de uma forma regular para responder às necessidades do mercado. Na Glintt-HS, para além de integrações externas, existe a necessidade de integrar sistemas internos, sendo esta uma realidade em crescimento. O termo *Enterprise Service Bus* (ESB) refere-se a um padrão arquitetural que complementa o SOA, aumentando o poder de criação e orquestração de serviços (Pearlman, 2016), tornando os processos de integração mais simples, conseguindo interligar sistemas heterogêneos, funcionando como um intermediário entre eles. Além disso um ESB fornece uma camada de abstração, onde o processo de desenvolvimento é realizado através de componentes, sendo estes configurados tipicamente através de uma interface gráfica permitindo que pessoas sem grandes conhecimentos de codificação consigam desenvolver *software.*

1.2 Problema

A arquitetura praticada na Glintt-HS, embora atual, apresenta algumas debilidades em termos de desempenho, flexibilidade e escalabilidade. Operando num meio tão crítico como é a área da saúde, é necessário que os serviços prestados sejam altamente escaláveis, flexíveis, robustos e rápidos, sendo necessário tomar medidas que permitam responder às inúmeras alterações que vão sendo preciso realizar no seu negócio.

Embora seja praticada uma arquitetura orientada a serviços, estes começam a tornar-se demasiado extensos e difíceis de manter. Deixam de ser independentes entre si, tornando-se altamente acoplados, originando os seguintes problemas: a dificuldade em integrar novos sistemas e o decréscimo de escalabilidade, flexibilidade e do desempenho de todo o sistema.

Um exemplo atual, supondo que existe um serviço relacionado com marcações de consultas, o que se sucede é a existência de um serviço web que permite gerir marcações, onde, normalmente, são realizadas operações como criar, ler, editar e apagar, podendo estas operações ser descritas pelo acrónimo inglês CRUD. Este serviço opera segundo o estilo arquitetural REST (*Representational State Transfer*), onde assim que recebe um pedido HTTP (verbo POST para efetuar a criação de uma marcação), é espoletada uma série de ações associadas a esse pedido: é efetuada a criação da marcação, seguindo-se a publicação de uma notificação via email e a inserção de um requerimento associado à marcação. Caso se pretenda adicionar mais alguma lógica a este serviço web como, por exemplo, adicionar a marcação a um calendário, é necessário que seja adicionado mais código para esse efeito, obrigando a que todo o serviço web tenha de ser novamente implantado. Embora apenas seja preciso implantar o serviço que sofreu as alterações, este serviço não contém apenas uma responsabilidade como é desejado, ou seja, o serviço em questão não gere apenas marcações, mas também faz a gestão de outros conceitos, como por exemplo a gestão de pacientes. Qualquer alteração a um destes conceitos irá causar impacto no outro. Além disto, como se pode observar, um serviço que apenas deveria efetuar uma marcação, apenas dá por concluído esse processo, quando também é enviada a notificação, quando é adicionado o requerimento e também adicionada a marcação ao calendário, tornando o serviço muito mais lento do que aquilo que é suposto e com um acoplamento demasiado alto. Este alto acoplamento faz com que seja difícil de manter estes serviços.

Quanto à integração de novos sistemas praticada na Glintt-HS, a mesma é bastante custosa, não existindo um padrão a seguir nem um intermediário que permita orquestrar e centralizar todas estas integrações. Deste modo, sempre que se pretende integrar qualquer tipo de aplicação quer seja interna quer seja externa, esta é realizada através de uma comunicação Ponto-a-Ponto, gastando tempo desnecessário em configurações e repetição de código.

1.3 Objetivos

Tendo em consideração o problema descrito anteriormente, os objetivos do trabalho presente nesta dissertação passam pela elaboração de uma arquitetura, que comparada com a arquitetura atual, possua as seguintes características:

1. Maior desempenho;
2. Maior escalabilidade;
3. Maior flexibilidade;

4. Diminuição do tempo de manutenção e evolução do sistema.

Com a solução final espera-se que, não só os clientes que usufruem das soluções da Glintt-HS, como também os próprios colaboradores da organização se sintam mais satisfeitos com o nível da solução desenvolvida.

1.4 Abordagem

Para resolver o problema apresentado, existem algumas soluções que podem ser utilizadas, como é o caso da elaboração de uma arquitetura orientada a micro serviços com comunicação Ponto-a-Ponto, da utilização de micro serviços com um ESB para orquestrar todo o sistema ou da utilização de micro serviços com um ESB integrando um sistema de filas de mensagens, fazendo com que o sistema comunique entre os diferentes componentes através de eventos. Uma arquitetura orientada a eventos define uma metodologia para desenhar e implementar sistemas através da publicação de eventos entre componentes de *software* totalmente desacoplados entre si.

A abordagem deste projeto passa pela análise das referidas soluções e da implementação da solução que melhor se adequa ao problema em questão.

1.5 Estrutura do documento

A estrutura desta dissertação apresenta num primeiro capítulo uma introdução ao tema aqui documentado, onde é feita uma contextualização e dado a conhecer o problema a resolver, apresentando os objetivos, assim como a respetiva abordagem para os atingir.

No segundo capítulo pode-se observar um estudo sobre o Estado da Arte onde é feito um aprofundamento dos temas aqui discutidos e das tecnologias que podem ser utilizadas para resolver o problema apresentado. É apresentada a arquitetura atual da Glintt-HS e também é feita uma análise de valor da nova solução, onde são apresentados os benefícios, sacrifícios e a perceção de valor para o cliente. Também é apresentado o novo modelo de desenvolvimento de conceitos defendido por Peter Koen enquadrado com o problema a solucionar e o modelo de Canvas, para permitir uma melhor perceção do modelo de negócio a implementar.

No terceiro capítulo é realizada uma análise de soluções existentes que resolver o problema apresentado. Aqui também é feito o design do sistema que se pretende implementar.

No quarto capítulo é documentada toda a implementação que foi aqui idealizada, sendo que os resultados dos testes à implementação podem ser observados no sexto capítulo.

O quinte capítulo contém a identificação da abordagem a seguir para a avaliação de toda a solução, onde são apresentadas as grandezas a serem avaliadas, as hipóteses a serem testadas, as metodologias a seguir e as avaliações realizadas.

O sexto capítulo, contém as conclusões de todo o trabalho desenvolvido, onde são apresentados os objetivos atingidos e os não atingidos, assim como o trabalho que se pretende efetuar no futuro.

No último capítulo são apresentados os anexos.

2 Estado da Arte

Neste capítulo é apresentada uma visão geral do projeto, onde são discutidos conceitos e tecnologias importantes relacionados com o problema no contexto conceptual e tecnológico, sendo também realizada uma Análise de Valor do projeto aqui documentado.

2.1 Contexto Teórico

Nesta secção são abordados os conceitos principais desta dissertação, sendo eles os micro serviços, sistemas monolíticos, arquitetura orientada a eventos e o *Enterprise Service Bus*.

2.1.1 Monolítico vs SOA vs Micro Serviços

Estes três conceitos definem três arquiteturas distintas de *software.* Todas têm os pontos fortes e fracos sendo que cada uma desempenha um papel importante consoante as necessidades do sistema que se pretende construir.

Um sistema monolítico é definido por um sistema que está construído num componente de *software* único, e todo o código necessário para o correto funcionamento desse sistema está inteiramente num único artefacto (Zaymus, 2017). As vantagens destes sistemas são:

- Simples de desenvolver;
- Simples de implantar;
- Simples de escalar, permite duplicar o sistema.

Algumas desvantagens:

- Apenas permite escalar através da duplicação do sistema;
- Processos de *Continuous Integration*[3], são difíceis de manter pois todo o sistema é implantado;
- Difícil de manter dada a quantidade de funcionalidades que podem estar presentes numa única aplicação;

[3] https://martinfowler.com/articles/continuousIntegration.html

- Todo o desenvolvimento fica assente numa única tecnologia, não tirando partido de tecnologias que podem ser mais capazes para resolver uma determinada funcionalidade.

Uma arquitetura orientada a serviços (SOA) é um padrão arquitetural que contrariamente ao sistema monolítico, divide os seus serviços em diferentes componentes, permitindo que estes comuniquem entre si. Estes componentes são divididos com a ideia de reutilizar funcionalidades dentro do mesmo negócio (Richards, 2015). Algumas vantagens destes sistemas são:

- Reutilização de Serviços;
- Aumento da escalabilidade, permitindo escalar os serviços que dizem respeito ao mesmo negócio;
- Aumento da flexibilidade, permitindo que o sistema seja alterado facilmente.

Algumas desvantagens:

- Existe uma sobre carga de validações que são feitas quando diferentes serviços comunicam entre si;
- Pode tornar-se confusa a gestão de todos os serviços. O termo ESB desempenha aqui um papel importante (ver secção 2.1.3).

O termo micro serviço refere-se a um estilo arquitetural que estrutura um sistema baseado em serviços muito pequenos e completamente autónomos. A ideia aqui ao contrário do SOA é desacoplar ao máximo os serviços sendo que cada uma deve apenas conter funcionalidades relacionadas com um domínio específico (Newman, 2015). Algumas vantagens destes sistemas são:

- Melhor organização;
- Escalabilidade completamente independente, podendo escalar apenas um determinado domínio;
- Maior desacoplamento;
- Se um serviço falhar, não impacta nas restantes funcionalidades do sistema.

Algumas desvantagens:

- Obriga a criar processos de implantação para tornar o desenvolvimento eficiente;
- A sobrecarga de validações com as comunicações entre micro serviços ainda é maior do que no SOA;

- A utilização de transações para manter a consistência de dados exige um esforço adicional.

A Glintt-HS possui um vasto leque de sistemas, onde é possível observar dois dos conceitos referidos anteriormente, o monolítico e SOA. Os sistemas monolíticos dizem respeito a aplicações legadas que ainda têm de ser suportadas, enquanto que os sistemas orientados a eventos têm sido uma mais valia para o desenvolvimento dos serviços da Glintt-HS. Um dos produtos fornecidos pela organização é conhecido como *GPlatform* e segue este padrão arquitetural, fornecendo serviços para a grande parte do negócio que a empresa está envolvida. Contudo, um dos referidos problemas do SOA está bastante presente neste produto, a gestão de todos os serviços. A granularidade que estes serviços começam a atingir também se apresenta como um problema preocupante, tornando difícil a sua manutenção. É aqui que os micro serviços desempenham um papel importante, implementando serviços com uma granularidade muito fina capazes de ser facilmente modificados e escalados, replicando apenas quando necessário (Tilkov, 2012).

Para desenvolver arquiteturas orientadas a micro serviços existem bastantes padrões que devem ser seguidos para construir uma solução robusta e fiável, conforme se pode observar nas secções seguintes.

2.1.1.1 API Gateway

O API Gateway fornece um ponto de entrada para comunicar com todos os micro serviços existentes no sistema. Uma aplicação *mobile/desktop* que pretenda obter dados relacionados com uma determinada funcionalidade não precisa de saber com que micro serviço tem de comunicar, apenas precisa de saber que existe um *gateway* que irá fazer de ponte para os micro serviços pretendidos. É nesta camada que também é comum serem feitas autorizações, autenticações, logs de pedidos e respostas, etc. O roteamento para os micro serviços é feito através de um outro padrão, o *Service Discovery* que é abordado na secção seguinte.

2.1.1.2 Descoberta de Serviço

Tal como o nome indica este padrão permite realizar a descoberta de serviços. Uma aplicação que pretenda consumir um determinado serviço, não precisa de saber diretamente onde se encontra a instância do *gateway,* nem o *gateway* precisa de saber a localização de todos os serviços existentes. Para isso existe o Serviço de Descoberta, que permite que os diferentes serviços sejam registados e encontrados. Caso um serviço seja migrado para outro servidor apenas o registo no Serviço de Descoberta tem de ser alterado para a correta localização.

2.1.1.3 Base de Dados Partilhada

A utilização de uma base de dados partilhada por cada micro serviço é bastante simples de utilizar e permite que os micro serviços quando precisam de dados relacionados com outras entidades não necessitem de comunicar entre si. Estes dados podem ser obtidos diretamente na base de dados e o tratamento de erros é muito simples, podendo ser utilizadas transações simples para controlar estes erros.

Alguns problemas deste padrão são que proporcionam um alto acoplamento, fazendo com que as mudanças a uma entidade na base de dados possam quebrar todas as outras. Além disso a carga exercida numa única base de dados é bastante maior que se existissem base de dados por serviço, pois todas as funcionalidades estão a aceder à mesma base de dados.

2.1.1.4 Base de Dados por Serviço

Este padrão é muito utilizado em arquiteturas orientadas a micro serviços. Este padrão indica que cada micro serviço deve conter a sua base de dados e esta base de dados deve ser apenas acedida pelo seu respetivo micro serviço. Deste modo, o sistema torna-se mais desacoplado, pois uma mudança na base de dados de um serviço tem zero impacto na base de dados dos restantes serviços. Além disso cada serviço pode utilizar base de dados de diferentes tipos consoante o tipo que melhor se adequa às suas necessidades.

Contudo, este padrão cria alguns problemas no desenvolvimento de *software.* Voltando ao exemplo do cancelamento de uma consulta, se uma consulta for cancelada e o serviço que trata do envio da notificação lançar algum erro, existem duas soluções: ou o cancelamento da consulta fica sem efeito e tem de ser novamente cancelado, ou o cancelamento é efetuado e fica apenas pendente de notificação. Caso esta notificação fosse considerada fundamental e o processo tivesse de ser todo refeito, com este padrão teríamos um problema de inconsistência de dados, pois não poderíamos utilizar transações para controlar os dados que são alterados.

2.1.1.5 SAGA

Utilizado para manter consistência de dados entre diferentes serviços. SAGA é uma sequência de transações locais, em que cada transação altera a base de dados e informa a próxima transação local do sucedido. Se algum erro acontecer, são executados alguns mecanismos de compensação que revertem as alterações realizadas pelas transações anteriores (Richardson, 2018).

Este padrão visa resolver os problemas levantados pelo padrão referido anteriormente, base de dados por serviço.

2.1.2 Arquitetura Orientada a Eventos

Uma arquitetura orientada a eventos é uma metodologia de desenvolvimento de soluções de software baseada na geração, receção e processamento de eventos. Os eventos fluem entre componentes desacoplados entre si, sendo estes agnósticos entre si. Estes sistemas são usados para construir sistemas complexos e flexíveis, sendo controlados por eventos assíncronos, não existindo qualquer tipo de estrutura de controlo, substituindo os mecanismos de pedido e resposta que são tão usados atualmente (Baker, 2017).

O conceito principal destes sistemas é que tudo o que tem interesse num determinado negócio, deve subscrever os eventos relacionados com esse negócio, sendo que estes subscritores serão notificados assim que exista uma publicação relacionada com os eventos subscritos, surgindo daí o mecanismo de *publish & subscribe*. Este mecanismo é uma outra maneira de descrever sistemas orientados a eventos onde um determinado evento dá origem à publicação de uma notificação para todos os subscritores, informando a sua ocorrência. De seguida, apresentam-se algumas características de sistemas que operam segundo mecanismos de publicação e subscrição.

- ***Message Broker* (MB)*:*** fornece comunicação via publicação e subscrição. As publicações, são uma espécie de notificação que possui uma mensagem relacionada com a publicação e são enviadas para um MB que armazena as mensagens e as encaminha para os consumidores correspondentes de uma maneira confiável e garantida. Se um consumidor falhar antes de processar a mensagem, o MB fica responsável por reprocessar a mensagem, garantindo que esta é entregue com sucesso;

- **Produtor**: quem publica um determinado evento;

- **Consumidor**: também referidos como subscritores, é quem consome os eventos publicados pelos produtores;

- **Anonimato:** um produtor apenas precisa de saber que existem consumidores que querem receber uma notificação relacionada com o seu evento. Da mesma forma, um consumidor apenas precisa saber que o produtor é legítimo. As mensagens são idênticas para todos os consumidores, fazendo com que o produtor apenas se tenha de preocupar em publicar uma única mensagem, sem ter de ser adaptada aos subscritores, mesmo quando são adicionados novos;

- **Descoberta:** estes sistemas fornecem mecanismos para que os produtores possam descobrir novos consumidores e vice-versa;

- **Entrega garantida:** os consumidores precisam de estar seguros que todas as mensagens são recebidas na ordem correta dos eventos lançados pelos produtores;

- **Baixa latência:** os consumidores devem ser notificados o mais brevemente possível;

- **Assunto:** os consumidores podem subscrever e filtrar as informações a receber.

Martin Fowler, publicou recentemente um artigo que visa enumerar os principais padrões a serem respeitados quando se fala de construção de sistemas orientados a eventos, sendo eles: *event notification, event-carried state transfer, event-sourcing* e *CQRS* (Fowler, 2017). Estes padrões não devem ser obviamente utilizados em todas as ocasiões, sendo preciso avaliar o sistema que se pretende desenvolver e as funcionalidades que este deve conter. De seguida, pode-se observar uma breve descrição elaborada pelo Martin Fowler sobre os referidos padrões. Para além destes referidos padrões, existem outros que também podem ser utilizadas na construção de arquiteturas orientada a micro serviços, que logicamente também fazem sentido numa arquitetura orientada a eventos, como é o caso do SAGA e *Database Per Service*.

2.1.2.1 *E*vent Notification

"Acontece quando um sistema envia mensagens de eventos para notificar outros sistemas de uma mudança em seu domínio" (Fowler, 2017). Uma notificação de eventos tem como característica principal o sistema que publica não ter de se preocupar com qualquer tipo de resposta por parte do sistema que foi notificado. Pode existir casos em que seja expectável a obtenção de uma resposta, mas caso seja esse o caso, o sistema que publica não tem uma preocupação direta. Por exemplo, um sistema de cancelamento de consulta (produtor) apenas precisa de saber que tem de cancelar a consulta e enviar uma notificação. Caso exista uma funcionalidade que envie um email assim que é realizado o cancelamento, o sistema que irá enviar o email (consumidor), apenas recebe esta notificação e efetua envio de SMS ao paciente que cancelou a consulta, não tendo que se preocupar em informar o produtor do sucedido.

Este padrão é simples de configurar e implica um baixo acoplamento entre sistemas, podendo se tornar problemático caso exista uma grande afluência de notificações de eventos, tornando difícil a perceção do fluxo normal de um programa, dado que este não se encontra explícito no código. É preciso, portanto, um grande cuidado com a adoção deste tipo de sistemas, pois embora seja simples de configurar e de tornar os sistemas desacoplados, podem existir casos onde se torna difícil de os manter.

2.1.2.2 Event-Carried State Transfer

"Este padrão aparece quando se pretende atualizar um sistema de tal forma que este não precisa de entrar em contato com o sistema de origem para continuar a trabalhar" (Fowler, 2017). Um sistema de gestão de pacientes pode publicar eventos com informação que contêm os dados que sofreram alterações sempre que um paciente altera seus dados. Um consumidor pode atualizar sua própria cópia dos dados do paciente com as respetivas mudanças, de modo a que nunca precise falar com o sistema principal para fazer o seu trabalho no futuro.

Ganha-se uma maior resiliência, uma vez que os sistemas de consumidores podem funcionar se o sistema principal se tornar indisponível. Reduz-se a latência e não é preciso preocupações

com a carga no sistema principal para satisfazer as consultas de todos os sistemas de consumidores. Envolve mais complexidade nos consumidores, uma vez que tem que ordenar a manutenção de todo o estado, quando geralmente é mais fácil chamar o produtor para obter mais informações quando necessário.

2.1.2.3 Event-Sourcing

"A ideia central do *sourcing* de eventos é que sempre que se realiza uma mudança no estado de um sistema, regista-se essa mudança de estado como um evento e pode-se reconstruir com confiança o estado do sistema reprocessando os eventos a qualquer momento no futuro" (Fowler, 2017).

Deste modo os eventos são armazenados, criando uma fonte de eventos em que o estado do sistema depende diretamente dessa fonte. Com este padrão é possível obter um histórico de todos os eventos ocorridos permitindo também que seja feita a reposição de estados sempre que acontece algum erro.

2.1.2.4 CQRS

Segregação de Responsabilidade de Consulta de Comando (CQRS) é a noção de ter estruturas de dados separadas para ler e escrever informações.

CQRS deve ser aplicado em domínios complexos, sendo particularmente rentável quando se tem uma diferença nos padrões de acesso, como muitas leituras e muito poucas gravações. CQRS deve ser aplicado com cuidado e é muito comum a implementação errada deste padrão dada a complexidade de utilização e o risco em geral acrescentado ao projeto em questão.

Os padrões enumerados anteriormente foram descritos por Martin Fowler, onde ele indica que todos os padrões devem ser tidos em conta dependendo das situações e do problema em questão. Todos podem resolver bastantes problemas se usados corretamente, como originar novos, caso sejam usados erradamente. É muito comum a confusão destes padrões, e se não forem bem aplicados podem atrasar o desenvolvimento do sistema, ou mesmo tornar o sistema mais erro e suscetível a erros.

2.1.3 Enterprise Service Bus (ESB)

O termo *Enterprise Service Bus* foi inventado pela Gartner em 2002 e apresentado pelo analista Roy Schulte para descrever uma categoria de produtos de software que ele observou estarem disponíveis no mercado naquele momento. Muitos anos depois, ainda não existe uma definição universal do que é um ESB, sendo que estas variam consoante a fonte ou o negócio onde se inserem (Kress, et al., 2013). Contudo, a definição que melhor se enquadra no âmbito desta dissertação é a seguinte: "o ESB é o suporte principal de SOA baseado em padrões, capaz de

conectar aplicações através de interfaces de serviço. Ao combinar mensagens, serviços Web, XML e transformação / gestão de dados, um ESB pode conectar, mediar e controlar de forma confiável comunicações e interações entre serviços" (Burns, 2011).

Na Figura 1 pode-se observar um diagrama que representa a arquitetura simplificada da utilização de um ESB e da respetiva integração com outros sistemas ou aplicações.

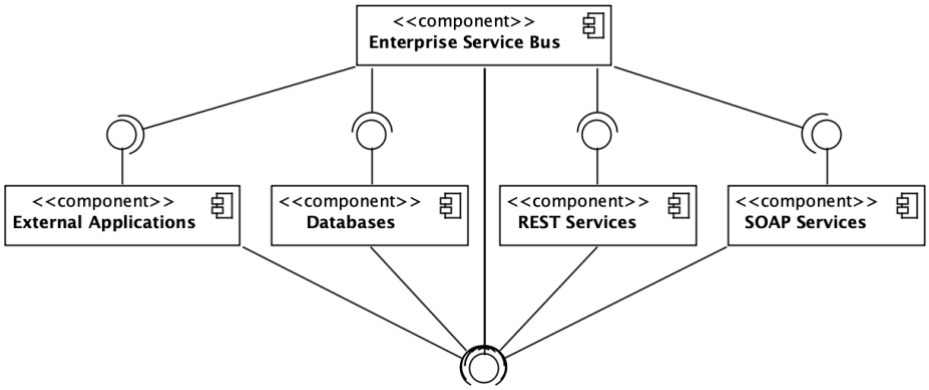


Figura 1 - Arquitetura simplificada de um sistema que utiliza um ESB

Um ESB é essencialmente uma arquitetura, definida por um conjunto de regras e princípios para integrar diferentes aplicações (What is an ESB, 2018). Como podemos observar na figura anterior o ESB é uma peça central de todo o sistema, que permite entre várias coisas integrar um sistema com outras aplicações, outros serviços diversos, comunicar com bases de dados de diferentes tipos, etc. O conceito principal é integrar diferentes sistemas, colocando um ESB entre eles e, permitindo que cada um comunique com o ESB. Deste modo, os sistemas ficam totalmente desacoplados entre si, permitindo que comuniquem sem dependências ou conhecimento dos outros sistemas que fazem parte do ESB. Este conceito surgiu devido à necessidade de se afastar da integração Ponto-a-Ponto, que se torna frágil e difícil de gerir ao longo do tempo. A integração Ponto-a-Ponto resulta num código de integração personalizado espalhado entre aplicações sem nenhuma maneira central de monitorizar ou solucionar problemas. Isso é muitas vezes referido como "s*paghetti code*", tornando difícil escalar o sistema dadas as dependências rígidas entre os diferentes componentes.

A principal razão para se utilizar um ESB é para aumentar a agilidade organizacional, reduzindo o tempo e facilitando a integração de novos sistemas. É bastante escalável e permite que as empresas melhorem os seus produtos com as vantagens de comunicação e de transformação de mensagens fornecidas pelos ESBs. Permite que as empresas sejam mais ágeis na entrega de novos produtos e serviços digitais, tanto internamente quanto em ecossistemas digitais. Esta agilidade é suportada pela capacidade da ESB de integrar perfeitamente aplicações, serviços, dados e processos em sistemas na nuvem, dispositivos móveis, etc. (What is an ESB, 2018).

Alguns princípios a serem seguidos ao implementar um ESB de modo a que este se mantenha ágil e escalável, agrupados pelos seguintes termos:

- **Orquestração:** criar um serviço composto por vários componentes existentes de granularidade fina. Existe um processo central que controla e coordena os restantes processos, ao contrário da coreografia onde não existe num processo central a coordenar, cada serviço envolvido necessita de interagir com outros serviços de forma ordenada. A orquestração promove a reutilização e a capacidade de gestão dos componentes subjacentes;

- **Transformação:** transformação de dados entre formatos de dados canônicos e formatos de dados específicos exigidos por cada conector ESB, como por exemplo uma simples transformação de XML para JSON. Os formatos de dados canônicos podem simplificar consideravelmente os requisitos de transformação associados a uma grande implementação ESB, onde há muitos consumidores e provedores, cada um com seus próprios formatos e definições de dados;

- **Transporte:** negociação do protocolo de transporte entre vários formatos (como HTTP, JMS, JDBC, etc.);

- **Mediação:** fornecer interfaces múltiplas com o objetivo de suportar várias versões de um serviço para compatibilidade com versões anteriores ou, alternativamente, permitir múltiplos canais para a mesma implementação de componente subjacente;

2.1.4 API-led Connectivity

API-led Connectivity é um padrão defendido pela Mulesoft, que é referenciado como o próximo passo na evolução de SOA (Mulesoft, 2018). Esta metodologia defende que uma arquitetura que possua necessidades complexas de ligação com outros sistemas, deve ser dividida em três camadas:

- ***Experience* ou experência*:*** responsável por adaptar os dados a ser consumidos ao respetivo interessado nesses dados. Uma prática muito comum é a existência de duas *experiences* distintas, uma para devolver dados para serem consumidos por aplicações *mobile*, outra para serem consumidos por aplicações web.

- ***Process* ou processo*:*** funciona como um orquestrador, é utilizado quando uma aplicação necessita de obter informações de diferentes aplicações (internas ou externas) para funcionar corretamente.

- ***System* ou sistema*:*** funcionalidade base de uma aplicação. Idealmente deve conter uma atividade atómica, como um acesso a uma base de dados para devolver ou inserir um registo por exemplo.

2.2 Arquitetura Atual

Nesta secção é dada a conhecer de uma forma mais detalhada a arquitetura praticada atualmente na Glintt-HS que, neste projeto, é alvo de reestruturação para resolver os problemas apresentados na secção 1.2.

Na Figura 2 é possível observar um diagrama de componentes de alto nível que representa a arquitetura da Glintt-HS que se pretende alterar.

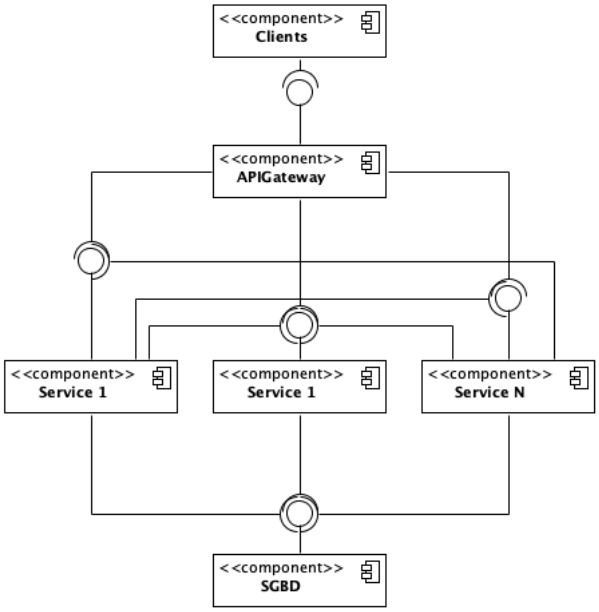


Figura 2 - Diagrama de Componentes - Arquitetura Atual, Glintt-HS

O presente diagrama de componentes pode ser interpretado da seguinte maneira:

- O componente *Clients* representa todos os clientes que podem ser utilizados para comunicar com todo o sistema. Este componente fornece uma camada de abstração que não necessita que sejam feitas chamadas HTTP para realizar a comunicação entre a aplicação que pretende utilizar este sistema, essa comunicação é feita através destes clientes que podem ser referenciados nas respetivas aplicações, comunicando com o restante sistema através de simples funções;

- O *APIGateway* representa o ponto de entrada para todos os serviços da plataforma e é um padrão muito conhecido em sistemas orientados a serviços. É aqui onde são feitas algumas operações de segurança e representa o ponto de comunicação com os serviços representados como os componentes *Service* 1, 2 e N*;*

- Os componentes *Service* representam então todos os serviços disponibilizados pela *GPlatform,* sendo que é possível observar que aqui são utilizados vários serviços que é onde está implementada a lógica de negócio do sistema. Estes serviços são acedidos

através do *APIGateway* mas é muito comum que comuniquem entre si, através de comunicações o Ponto-a-Ponto.

- O SGBD (Sistema de Gestão de Base de Dados), representa a base de dados que é acedida pelos repositórios recorrendo a *Entity Framework*. Aqui não é utilizado o padrão Base de Dados por Serviço, existindo apenas uma base de dados partilhada por todos os serviços existentes. O acesso à base de dados é conseguido através do padrão Descoberta de Serviço, onde através de uma chave previamente definida, os serviços consultam um outro serviço (de descoberta) para obterem as ligações para a base de dados associada à respetiva chave.

A arquitetura aqui apresentada, é representada por uma plataforma que fornece uma grande variedade de serviços, que tem como objetivo agregar todos os serviços disponibilizados pela organização. Esta plataforma, a GPlatform, é o produto da Glintt-HS que contém grande parte dos serviços que a empresa disponibiliza. Implementa uma arquitetura SOA, que é desenvolvida em .NET[4] e possuindo um sistema de gestão de bases de dados Oracle.

Esta arquitetura é bastante flexível e permite que os diferentes serviços estejam desacoplados entre si. Embora permita desenvolver novas funcionalidades ou efetuar correções sem existir necessidade de implantar novamente todos os serviços (apenas o serviço alterado necessita de ser implantado), todos os métodos fornecidos pelo serviço modificado, podem ficar suscetíveis a erros que tenham ocorrido aquando de novo desenvolvimento ou de uma correção. Estes erros podem ser facilmente detetados com os testes unitários e de integração existentes, contudo os erros podem acontecer, o que não é suposto e precisa de tempo para ser resolvido.

A lógica associada a cada serviço pode-se tornar demasiado extensa, tornando os serviços lentos e difíceis de manter. A granularidade dos serviços tornou-se bastante grossa, e quando existe uma alteração a realizar num serviço de cancelamento de consultas, para corrigir um bug detetado ou para permitir que agora esse serviço envie uma notificação a informar do sucedido, serviços relacionados com marcações, pacientes, recursos hospitalares, ficariam inativos quando se procedesse com as alterações e, mais uma vez, ficariam suscetíveis a erros dado que estão a ser publicados novamente.

A ideia inicial desta plataforma passava pelo desenvolvimento de micro serviços em vez de serviços web, ou seja, um micro serviço deveria conter apenas lógica relacionada com uma entidade de negócio, fazendo com que cada entidade seja completamente independente das outras. Tal deixou de acontecer, sendo que existe uma enorme quantidade de diferentes entidades em apenas um serviço. Os micro serviços que foram inicialmente pensados, tornaram-se rapidamente serviços muito extensos, sendo que neste momento existem cerca de quinze serviços, em que dois se estão a tornar demasiado extensos e difíceis de manter. Dada a grande centralização que se apoderou da plataforma, desenvolvendo quase todas as

[4] https://www.microsoft.com/net

novas funcionalidades inseridas nestes dois serviços, teve-se que se repensar todo o processo de desenvolvimento e a arquitetura de todo o sistema.

Nesta arquitetura é possível observar padrões como API *Gateway,* descoberta de serviço e base de dados partilhada. Este último padrão também não deveria ser utilizado, pois como a ideia seria a elaboração de micro serviços, deveria existir uma base de dados por cada serviço, contudo, o sistema da Glintt-HS é bastante antigo e possui uma grande quantidade de lógica inserida na base de dados. Esta lógica tem vindo aos poucos e poucos sendo retirada da base de dados, mas ainda está longe de ser finalizado, sendo que terá que ser utilizada uma base de dados partilhada por todos os serviços. Como os serviços são de granularidade grossa, não é utilizado SAGA para garantir consistência entre os dados, sendo assim utilizado transações simples para controlar esta consistência.

Pelas razões demonstradas anteriormente, sendo o maior problema referido a granularidade de cada serviço, este sistema é bastante difícil de escalar, de modificar e também apresenta tempos de resposta demasiados lentos, pelo que na secção seguinte é estudada uma solução existente para combater estes problemas.

2.3 Contexto Tecnológico

Nesta secção é realizado um estudo sobre arquiteturas e soluções que permitam resolver o problema mencionado na secção 1.2.

Existe um grande leque de soluções disponíveis no mercado com a finalidade de resolver problemas de escalabilidade e flexibilidade, como é o caso de soluções orientadas a micro serviços, orientadas a eventos e até mesmo soluções recorrendo ao uso de ESBs. Posto isto, decidiu-se estudar uma solução apresentada pelo WSO2, que é uma organização que oferece uma grande variedade de "produtos que suportam a agilidade e inovação para um negócio digital de sucesso" (WSO2 | The Open Source Technology for Digital Business, 2018), inclusive um dos produtos que é estudado nesta dissertação (ver secção 2.4.5). Esta solução apresenta-se como um misto de conceitos, onde é possível observar o uso de um ESB para integrar e orquestrar sistemas e também o uso de um *Message* Broker para a implementação de mecanismos de *publish & subscribe* de eventos de mensagens. Na Figura 3 pode-se observar a arquitetura que o WSO2 defende como o seu ideal de arquitetura orientada a eventos integrado com um *Enterprise Service Bus*.

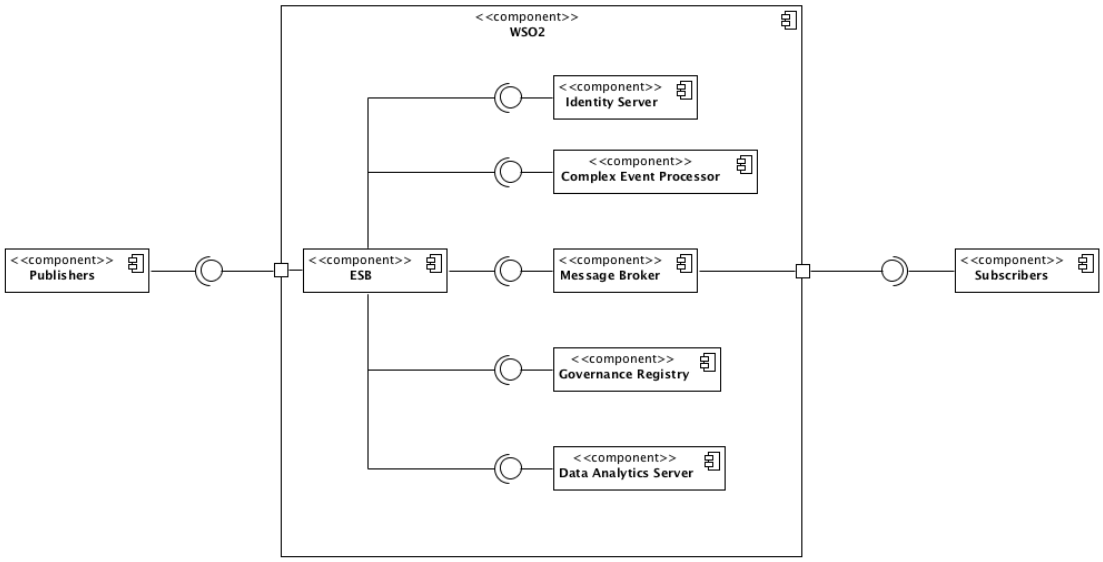


Figura 3 - Arquitetura event-driven integrada com um ESB

Como referido anteriormente, o WSO2 fornece um vasto leque de tecnologias/componentes, das quais se salientam o *Enterprise Service Bus, Message Broker, Identity Server, Complex Event Processor, Governance Registry* e *Data Analytics Server.* Para além dos conceitos ESB e *Message Broker* que foram apresentados na secção 2.1, é feita uma descrição de algumas tecnologias fornecidas pelo WSO2 para construir uma solução orientada a eventos:

- ***Identity Server* (IM):** permite a definição de padrões e controlar protocolos de autenticação. Possibilita a definição e implementação de políticas empresariais de segurança. São uma peça fundamental das arquiteturas modernas, fornecendo um ponto de aplicação de políticas e salvaguardando todo o sistema. Estes servidores devem ser altamente escaláveis.

- ***Complex Event Processor* (CEP):** permite reconhecer sequências de mensagens que podem sinalizar um evento importante, tais como ameaças de segurança, oportunidades de negócio, etc. Este conetor permite definir regras ou mecanismos, tornando o sistema mais inteligente.

- ***Governance Registry* (G-Reg)*:*** armazena informações específicas que as aplicações utilizam na inicialização ou em operação e mantêm a única resposta verdadeira para essas questões.

- ***Data Analytics Server* (DAS):** servidor onde são realizadas análises dos dados que transacionam no sistema. Pode servir para monitorizar o sistema.

Os componentes referidos anteriormente, são desenhados para otimizar a performance e a confiabilidade de um sistema, também permitindo que pessoas sem conhecimentos técnicos de codificação sejam capazes de definir operações de negócio. Estas ferramentas fazem a

arquitetura idealizada pela WSO2 bastante segura, robusta, escalável e resiliente quanto à falha (Abeysinghe, 2016).

Numa arquitetura como esta, o fluxo de execução normal passa pelo envio de uma mensagem por parte do produtor para o *Enterprise Service Bus*, que será responsável por receber a mensagem, realizar as devidas transformações caso assim seja necessário e prosseguir com o envio da mensagem para o *Message Broker*, onde irá ser, à posteriori, encaminhado para o respetivo consumidor da mensagem em trânsito. Esta mensagem durante todo o processo, pode passar por outros componentes abordados anteriormente, podendo ser alvo de autenticações, de análise de dados, etc.

Esta arquitetura defende que se o sistema requer um grande tráfego de mensagens o ESB deve ser o centro de toda a arquitetura, utilizando mesmo vários ESBs, salientando que devem ser tomados cuidados quando se pensa em implementar um sistema muito focado na confiabilidade e no reconhecimento total (i.e. produtores serem sempre informados de todas as mensagens entregues), dado que podem diminuir consideravelmente a performance do ESB (Abeysinghe, 2016).

Todos os componentes referidos são *open source*, sendo um dos grandes pontos fortes da solução fornecida pelo WSO2. Possuem software com muita qualidade totalmente *open source* e com uma grande comunidade que o suporta.

2.4 Tecnologias Existentes

Nesta secção é feito um estudo do estado da arte sobre tecnologias que podem ajudar a resolver o problema apresentado. Para a implementação de *event-driven* existem bastantes sistemas de filas de mensagens como é o caso do IBM MQ, RabbitMQ, Apache Kafka, *ActiveMQ*, etc. (Best Message Queue Solutions, s.d.). Nesta dissertação apenas vão ser estudados dois dos referidos sistemas, sendo eles o RabbitMQ e o Apache Kafka. A escolha destes dois sistemas recaiu principalmente pelo reconhecimento geral que existe sobre os dois sistemas, mas também pela facilidade de implementação e de monitorização do RabbitMQ e o do incrível poder do Kafka em termos de desempenho.

Embora a Glintt-HS já trabalhe com um ESB, desenvolvido pela Mulesoft, decidiu-se também estudar o WSO2 ESB para entender as principais diferenças entre estes dois sistemas, com o intuito de ficar a conhecer como o Mule ESB se compara com as restantes alternativas do mercado. O ESB terá a responsabilidade de comunicar com o sistema de fila de mensagens e de integrar sistemas / aplicações sempre que necessário. O WSO2 é uma tecnologia com um grande suporte da comunidade e encontra-se numa fase de grande crescimento, existindo mesmo um aumento da procura de pessoas com competências em WSO2 ESB em Portugal, nomeadamente no Porto (WSO2, 2018), enquanto que o Mule ESB que a Glintt-HS começou recentemente a utilizar, é dos ESBs mais conhecidos e no momento, sendo usado por nomes como *Spotify*, *Netflix*, *Mastercard*, *Salesforce*, etc.

2.4.1 RabbitMQ

RabbitMQ é um sistema de fila de mensagens distribuídas. É executado como um conjunto de nós onde as filas estão espalhadas pelos nós e, opcionalmente, replicadas para tolerância a falhas e alta disponibilidade. É "o intermediário de mensagens de código *open source* mais implantado" (RabbitMQ - Messaging that just works, 2018).

Implementa nativamente o AMQP (*Advanced Message Queuing Protocol*) 0.9.1, que é um protocolo de mensagens que permite que aplicações comuniquem com intermediários de mensagens. Oferece outros protocolos, como por exemplo o HTTP, via plug-ins.

Este sistema distingue-se pela sua capacidade de roteamento altamente flexível, permitindo construir um sistema de mensagens distribuídas rápido, escalável e confiável (Humphrey, 2017). Na Figura 4 pode-se observar uma arquitetura simplificada do RabbitMQ.

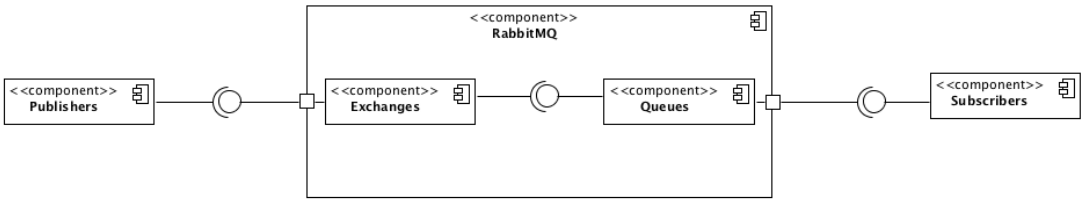


Figura 4 - Arquitetura simplificada – RabbitMQ

Na figura anterior podemos observar os produtores de mensagens que representam as aplicações que enviam mensagens para o RabbitMQ. O RabbitMQ possui *exchanges* que apenas recebem mensagens e tratam de as encaminhar para a respetiva fila de espera (*queues*). As mensagens que se encontram na fila de espera são, por fim, entregues aos consumidores.

Utiliza um modelo de intermediário inteligente (RabbitMQ) / consumidor "burro", focado na entrega consistente de mensagens aos consumidores que consomem a um ritmo aproximadamente similar ao que o RabbitMQ acompanha o estado do consumidor. O RabbitMQ processa todas as suas mensagens em memória, o que permite um desempenho muito bom quando o volume de dados não é muito elevado.

A visão geral deste sistema pode ser vista, muito resumidamente, do seguinte modo:

- Produtores enviam mensagens para *exchanges;*
- *Exchanges* roteiam mensagens para as *queues* e, caso necessário, outros *exchanges;*
- Os produtores recebem confirmações assim que um consumidor recebe as suas mensagens;

- Os consumidores mantêm conexões TCP (*Transmission Control Protocol*) persistentes com o RabbitMQ e indicam quais as filas que consomem;
- RabbitMQ envia mensagens aos consumidores;
- Os consumidores podem enviar reconhecimentos de sucesso / falha, caso sejam configurados para tal;
- As mensagens são removidas das filas quando consumidas com sucesso. Isto ocorre quando um dos consumidores informa que recebeu a mensagem.

2.4.2 Apache Kafka

Apache Kafka é sistema de fila de mensagens desenvolvido pelo LinkedIn e é escrito em Java. Em 2011 tornou-se *open source* e é considerado muito rápido e de custo extremamente baixo. É definido como "uma plataforma de transmissão distribuída" (Apache Kafka, 2017).

É desenhado para a publicação e subscrição de mensagens de alto volume, para ser durável, rápido e escalável (Humphrey, 2017). É executado num cluster de servidores, que armazena fluxos de registos, ou mensagens, em categorias identificadas por tópicos.

Este sistema pode ser usado como um sistema de mensagens, de armazenamento, ou processamento de dados, sendo que aqui será estudado como um sistema de mensagens.

O Kafka, possui tópicos que representam as filas de espera e podem ser subscritos por vários, ou mesmo nenhuns consumidores. São definidos por uma categoria para o qual as mensagens são publicadas, sendo estas mensagens todas processadas em disco, tornando-o bastante resiliente a falhas e não tendo um grande impacto no desempenho (Apache Kafka, 2017).

Na Figura 5 pode-se observar um exemplo de uma arquitetura simplificada que utiliza o Apache Kafka.

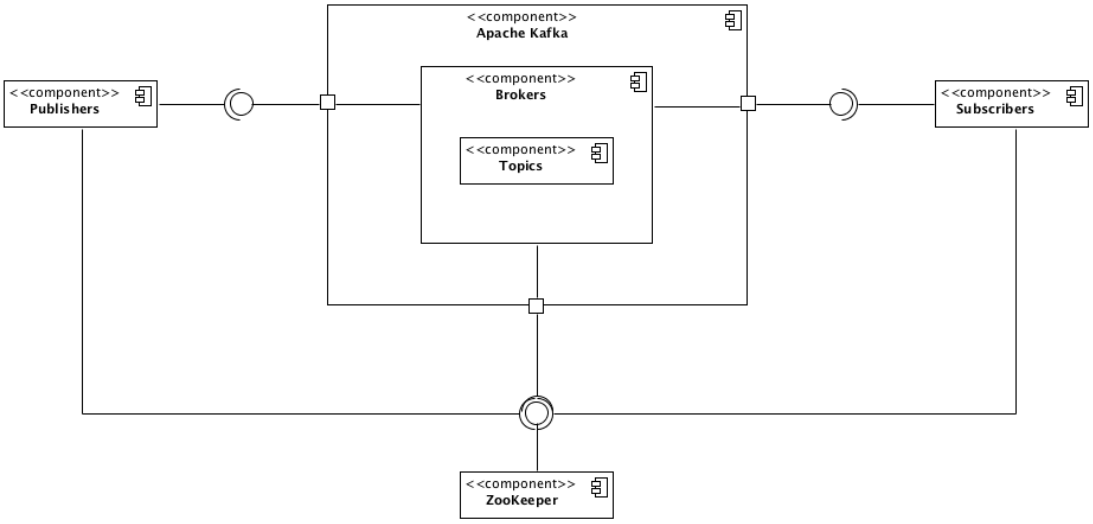


Figura 5 – Arquitetura simplificada - Apache Kafka

Utiliza um modelo intermediário "burro" (Kafka) / consumidor inteligente, não mantendo o controlo das mensagens que foram lidas, retendo todas as mensagens por um período de tempo determinado, sendo que os consumidores têm a responsabilidade de consumir as mensagens que estão nos tópicos a que estes subscreveram. Os produtores podem especificar que pretendem ser notificados quando as suas mensagens são colocadas em listas de espera, caso contrário podem correr o risco das mensagens não serem entregues aos respetivos consumidores.

O Kafka apresenta-se como um produto que não é autossuficiente, necessitando de software externo para funcionar corretamente. O *ZooKeeper* é a ferramenta que normalmente é utilizada como *scheduler,* ou mesmo como uma base de dados onde todos os intervenientes (i.e., produtores*,* consumidores*,* etc.) podem modificar e consultar o estado do sistema.

A visão geral deste sistema pode ser vista, muito resumidamente, do seguinte modo:

- Produtores publicam mensagens para o *broker*. Um produtor tem a responsabilidade de consultar o *ZooKeeper* para saber quais os *brokers* para os quais deve enviar as mensagens;

- A mensagem é recebida pelo *broker* e inserida no devido tópico que pode ser visto como a fila de espera (*queue*);

- Os consumidores têm a responsabilidade de ler as mensagens da fila de espera ao seu ritmo;

- Os *brokers*, produtores e consumidores comunicam com o *ZooKeeper* para gerir e partilhar o estado de todo o sistema;

2.4.3 Comparação entre RabbitMQ e Kafka

Na Tabela 1 podemos observar uma comparação entre o RabbitMQ e o Apache Kafka, que serão as duas ferramentas a estudar para responder à necessidade de processamento de mensagens para o desenvolvimento da arquitetura orientada a eventos.

Tabela 1 - Comparações de tecnologias – Sistemas de Filas de Mensagens

Características	**RabbitMQ**	**Apache Kafka**
Escalabilidade	Ampla	Ampla
Modelo de Operação	Intermediário inteligente / consumidor burro	Intermediário burro / consumidor inteligente
Clientes Suportados	Java, Spring, .NET, PHP, Python, Ruby, JavaScript, Go, Elixir, Objective-C, Swift, etc.	Java e alguns suportados pela comunidade
Armazenamento	Memória RAM. Pode também armazenar em disco se configurado para tal	Disco
Persistência de dados	Não persiste os dados. São apagados assim que as mensagens são consumidas	Os dados são sempre guardados até atingir o limite de tempo ou de tamanho
Características	RabbitMQ	Apache Kafka
SSL	Suporta	Não suporta
Autossuficiente	Sim	Não. Necessário o ZooKeeper
Permissões e controlos de acesso	Suporta	Não suporta
Histórico (Consulta de mensagens já processadas)	Não suporta. Pode fornecer esta característica com recursos a software adicional	Suporta

Características	RabbitMQ	Apache Kafka
Ideal para	Trabalhar com diferentes protocolos de transmissão de mensagens / roteamento complexo / integrar com infraestruturas já existentes	Fornecer eventos / processar fluxos / transmitir dados sem roteamento complexo
Desempenho	Com muita afluência de dados, torna-se significativamente mais lento	Muito boa
Documentação	Muito boa, com bastantes exemplos e muito intuitiva	Um pouco confusa e incompleta
Monitorização	Interface gráfica que corre no *browser* que permite monitorizar e gerir o RabbitMQ. Possui APIs para monitorizar e resolver problemas	Necessita de software externo

2.4.4 Mule ESB

Mule é um ESB *open source* desenvolvido sobre um *plugin* do Eclipse e é baseado em Java. Permite a integração de novos sistemas de uma forma muito rápida e intuitiva, podendo ser implantado em qualquer lugar. É muito fácil de configurar e instalar tornando-o muito simples de utilizar, "é altamente escalável, permitindo que se comece pequeno e conecte mais aplicações ao longo do tempo. O ESB gere todas as interações entre aplicativos e componentes de forma transparente, independentemente de existir na mesma máquina virtual ou através da Internet, e independentemente do protocolo de transporte subjacente utilizado" (What is an ESB, 2018).

Figura 6 - Mulesoft logo (What is an ESB, 2018)

Como referido na secção 2.1.3, a principal vantagem de um ESB é que permite que diferentes aplicações comuniquem entre si, atuando como um sistema de trânsito para transportar dados entre várias aplicações.

Abaixo são apresentadas algumas características do Mule ESB:

- **Criação e hospedagem de serviços:** exposição e hospedagem de serviços reutilizáveis, utilizando o ESB como um *container* de serviços;
- **Mediação de serviços:** serviços independentes de formatos e protocolos de mensagens, lógica de negócios separada de mensagens e chamadas a serviços independentes da localização;
- **Roteamento de mensagens:** roteamento, filtração, agregação e reordenação de mensagens com base em conteúdo e regras;
- **Transformação de dados:** troca de dados em diferentes formatos (JSON para XML e vice-versa por exemplo).

Este ESB possui conetores, que representam o código do ESB de uma forma visual, que ajudam na integração de aplicações e na transformação de dados, existindo um leque muito vasto de

conetores pré-definidos e permitindo a criação de novos. Estes conetores podem depois ser reutilizados por todas as aplicações.

Algumas características interessantes:

- Mule possui um vasto leque de conetores pré-definidos, como por exemplo: HL7, *Salesforce*, *Facebook, Twitter, SAP, etc.* Isto permite que sejam feitas integrações sem ter de ser realizado qualquer tipo de instalação. Tudo está disponível com a instalação do Mule;

- Permitem a reutilização significativa de componentes. Os componentes não requerem nenhum código específico para ser executado e não é necessária nenhuma API programática. A lógica de negócios é mantida completamente separada da lógica de mensagens;

- O Mule pode ser implantado numa variedade de topologias, não apenas ESB. Pode diminuir drasticamente o tempo de mercado e aumentar a produtividade de projetos para fornecer aplicativos seguros e escaláveis que sejam adaptáveis às mudanças;

- Utiliza um conceito de *design-first* para a exposição de serviços web, recorrendo a RAML (*RESTful API Modeling Language*). RAML permite a definição de uma API completa de uma maneira muito simples baseado na linguagem YAML.

Na Figura 7 podemos observar como seria definida uma API utilizando RAML. Desta maneira é possível disponibilizar uma API funcional com dados de teste apenas utilizando YAML para descrever a API. Os dados apresentados a quem consulta a API são baseados nos exemplos fornecidos aquando da definição da API.

```
/songs:
  description: Collection of available songs in Jukebox
  get:
    description: Get a list of songs based on the song title.
    queryParameters:
      songTitle:
        description: "The title of the song to search (it is case insensitive and doesn't need to
match the whole title)"
        required: true
        minLength: 3
        type: string
        example: "Get L"
    responses:
      200:
        body:
          application/json:
            example: |
              {
                "songs": [
                  {
                    "songId": "550e8400-e29b-41d4-a716-446655440000",
                    "songTitle": "Get Lucky"
                  },
                  {
                    "songId": "550e8400-e29b-41d4-a716-446655440111",
                    "songTitle": "Loose yourself to dance"
                  },
                  {
                    "songId": "550e8400-e29b-41d4-a716-446655440222",
                    "songTitle": "Gio sorgio by Moroder"
                  }
                ]
              }
```

Figura 7 - Definição de uma API com RAML (RAML 200, 2018)

No exemplo anterior é possível observar o recurso *songs* definido, com uma breve descrição e um método GET. Este método também possui uma breve descrição e o *query parameter songTitle*. Apenas suporta um tipo de resposta com o código HTTP 200, que é do tipo JSON. Este serviço também apresenta um exemplo de resposta.

Como se pode observar o RAML permite a definição de serviços de uma forma bastante rápida e intuitiva. Permite a definição de recursos, métodos, *uri parameters, query parameters,* tipos de resposta, etc. É uma ferramenta muito útil que torna o desenvolvimento e exposição de APIs muito rápido.

2.4.5 WSO2 ESB

É um ESB *open source,* definido como um mecanismo de mensagens baseado em padrões que fornece o valor das mensagens sem escrita de código. Fornece recursos de integração de dados e além dos fluxos de integração de curta duração e sem estado, também pode ser usado para gerir processos de negócios de longa data controlando o seu estado. Fornece recursos para intermediários de mensagens que podem ser usados para transmitir mensagens de uma forma confiável, bem como recursos para executar micro serviços para fluxos de integração. "É uma das ferramentas mais fáceis para obter uma integração de aplicativos corporativos (EAI). Possui uma arquitetura que facilita a comunicação entre aplicações e ajuda a conectar sistemas de

software com diferentes formatos de dados numa arquitetura integrada unificada" (WSO2 | The Open Source Technology for Digital Business, 2018).

Figura 8 - WSO2 logo (WSO2 | The Open Source Technology for Digital Business, 2018)

Fornece um vasto leque de conetores pré-definidos e permite a comunicação via diferentes protocolos, assim como a transformação de mensagens de uma forma muito simples e intuitiva. Possui mecanismos de monitorização e análise de acessos, performance e estatísticas de vários tipos, permitindo identificar *bottlenecks* e a criação de *dashboards* com a informação recolhida.

Também como o Mule ESB apresenta características muito semelhantes, fornecendo capacidades de mediação de serviço, roteamento de mensagens, transformação de dados, hospedagem de serviços, etc., apresentando-se como um ESB muito capaz de responder às necessidades do mercado.

Uma das grandes vantagens deste sistema é que possui uma quantidade de *software* que opera dentro do ambiente WSO2, que ao serem integrados com o ESB permitem desenvolver soluções muito robustas, com capacidades de análise de dados, autenticação e autorização, tudo sem necessidade de utilização de *software* externo.

2.4.6 Comparação entre Mule ESB e WSO2 ESB

Na Tabela 2 podemos observar uma comparação entre o Mule ESB e o WSO2 ESB, que serão as duas ferramentas a estudar para responder à necessidade de integrar diferentes sistemas na arquitetura orientada a eventos.

Tabela 2 - Comparações de tecnologias – ESB

Características	Mule ESB	WSO2 ESB
Implantação	Muito fácil de implantar	Muito fácil de implantar
Definição de APIs	Utiliza RAML para definir uma API. Conceito de desenhar primeiro a API e só depois desenvolver	Utilizando o WSO2 API Management, também permite a definição de APIs baseada no swagger, via interface gráfica

Características	Mule ESB	WSO2 ESB
Conetores pré-definidos	Possui um grande leque de conetores pré-definidos: HL7, Salesforce, Slack, Amazon, Facebook, Magento, MongoDB, SAP, Twitter, entre outros, sendo que grande parte dests conetores são suportados pela própria Mulesoft	Também possui um grande leque de conetores pré-definidos suportados pela comunidade
Comunidade	Pequena. Em ascensão	Grande e bastante participativa
Utilizado por	Spotify, Netflix, Mastercard, Unicef, Cisco, Salesforce, Coca-Cola, etc.	Ebay, Motorola, verifone, subhub, etc.
Transformações e Filtragem	Conetores avançados de transformações, recorrendo a *dataweave* (linguagem baseada em *scala*) para transformar mensagens, que para além de converter mensagens entre diferentes formatos também permite a realização de operações de filtragem com o *dataweave*. Filtros e transformações realizados no mesmo conetor	Conetores pouco intuitivos e complexos para transformar mensagens. Normalmente são usados conetores *filter* para realizar filtros e de dados e conetores *DataMapper* para realizar mapeamentos entre diferentes formatos de dados. Os conetores *filters* são definidos em XML
Alta disponibilidade	Fornece ferramentas de alta disponibilidade	Fornece ferramentas de alta disponibilidade
Licenciamento	Solução *open source* é bastante limitada. Modelo de licenciamento bastante limitado.	Solução completamente *open source*.
Conetores	Bastante intuitivos e com uma aparência moderna	Menos intuitivos e com uma aparência mais antiga

2.5 Análise de Valor

Análise de valor "é uma abordagem sistemática que procura aumentar o valor de um projeto, produto ou serviço fornecendo as funções necessárias para que seja atingido o desempenho necessário através do menor custo possível" (Miles, 2011).

Valor, é um conceito subjetivo que é interpretado de diferentes maneiras pelos clientes, consoante o contexto inserido, podendo ser definido como uma necessidade, interesse ou preferência. Caracteriza-se pelo equilíbrio entre os benefícios e os sacrifícios da elaboração de um produto. Na ótica do cliente são estes benefícios e sacrifícios que influenciam o valor que um determinado produto produz, sendo que este pode variar consoante o cliente.

Para um cliente, a sua perceção de valor varia segundo as perceções do que o que recebe e do que dá, sendo que podem ser influenciadas por drivers de benefícios, tais como:

- Qualidade;
- Capacidade de resposta;
- Confiabilidade.

E alguns drivers de sacrifícios, tais como:

- Preço;
- Tempo;
- Esforço.

Quando existe uma quantidade considerável de benefícios em detrimento de sacrifícios e, obviamente, o consumidor percebe vantagem sobre a oferta de uma organização, é seguro afirmar que existe valor para o cliente.

Na Figura 9 podemos observar uma perspetiva longitudinal sobre a caracterização do valor para o cliente (VC), descrita por Tony Woodall.

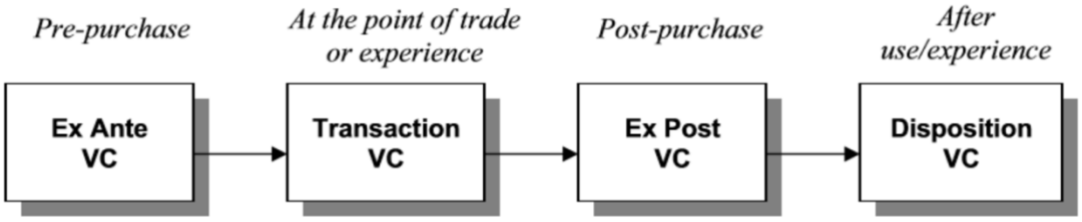


Figura 9 - Valor para o Cliente - perspetiva longitudinal (Woodal, 2003)

Woodal, divide o VC em quatro posições temporais:

- **Pré-compra:** fase onde se tenta prever como as pessoas entendem os serviços;
- **Ponto de negócio:** implica uma sensação de VC com experiência no ponto de negócio;
- **Pós-compra:** fase que fornece resultados de experiências com bases nas escolhas dos clientes ou dos fornecedores;
- **Após uso/experiência:** fase de reflexão sobre o ponto de disposição/venda.

A solução apresentada nesta dissertação retrata a melhoria da arquitetura de software praticada na Glintt-HS, podendo ser observado os seus benefícios e sacrifícios segundo as posições referidas por Woodal na Tabela 3.

Tabela 3 - Análise de Valor - benefícios e sacrifícios

Posição Temporal	Benefícios	Sacrifícios
Pré-compra	Soluções alternativas Flexibilidade Robustez	Esforço Custo temporal
Ponto de negócio	Confiabilidade	Custo temporal Custo monetário
Pós-compra	Apreciação dos utilizadores Suporte do serviço	Esforço Custo temporal Conflitos (reuniões)
Após uso/experiência	Qualidade do produto Poupança de tempo Aumento de performance	Custo temporal (suporte)

2.5.1 Proposta de Valor

A Proposta de Valor (VP) é essencial para alertar novos clientes sobre o valor dos produtos e serviços de uma organização e deve deixar claro o produto ou serviço que se fornece, os clientes-alvo, o valor inerente fornecido e seu valor exclusivo. O produto aqui apresentado tem

como o principal alvo a própria organização , mas também os utilizadores que irão usufruir dos serviços disponibilizados. Com este produto pretende-se aumentar melhorar o serviço prestado pela Glintt-HS aos seus clientes, apresentando uma solução inovadora que torna o sistema da Glintt-HS mais flexível, escalável e com mais desempenho. O processo de desenvolvimento de novas funcionalidades e de manutenção tornar-se-á mais simples, rápido e seguro, fornecendo vantagens tanto para os colaboradores da organização como para os utilizadores que consomem os produtos da organização.

2.5.2 Novo Modelo de Desenvolvimento de Conceitos

O novo modelo de desenvolvimento de conceitos, em inglês *New Concept Development Model* (NCD), é um modelo defendido por Peter Koen, "fornece uma linguagem comum e uma visão holística do *front end*. O modelo divide o *front end* em três áreas distintas. O primeiro é o motor, ou centro do modelo, que explica a visão, estratégia e cultura que impulsiona a FEI (*Front End of Innovation*). A segunda, ou parte interna, define os cinco elementos de atividade da frente. O terceiro elemento consiste nos fatores ambientais externos" (Front End Innovation - What is the New Concept Development (NCD) model?, 2018).

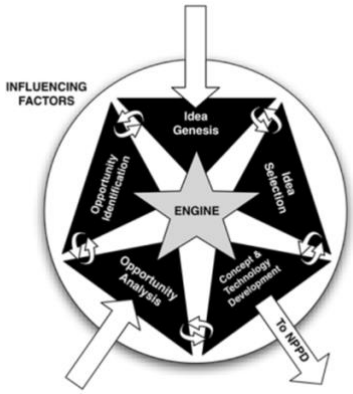


Figura 10 – NCD, por Koen (Dewulf, 2016)

O motor apresentado no centro da Figura 10, representa a liderança, cultura e estratégia de negócios da organização que conduz os cinco elementos chave, sendo eles a identificação da oportunidade, análise da oportunidade, gênese da ideia, seleção da ideia e desenvolvimento do conceito e da tecnologia.

2.5.2.1 Identificação da Oportunidade

Neste elemento, as oportunidades tecnológicas e de negócio são identificadas. Existem métodos que podem ser praticados por uma organização para identificar estas oportunidades, podendo estes métodos serem classificados em dois tipos:

- **Formais:** métodos de resolução de problemas, mapeamento de cenários futuros, *brainstorming*, etc.;

- **Informais:** simples conversa, a exposição de visões individuais, etc.

A solução apresentada neste documento surge da oportunidade de tornar o sistema da Glintt-HS mais moderno, escalável e flexível através do uso de tecnologias ou metodologias de ponta, como é o caso da arquitetura *event-driven* e dos ESBs.

2.5.2.2 Análise da Oportunidade

Aqui, deve ser reunida informação adicional que é fundamental para traduzir as oportunidades identificadas em oportunidades específicas de negócio e tecnologia para a organização. Alguns métodos de análise da oportunidade podem ser a criação de grupos focais, estudos de mercado, experiências científicas, etc.

Através da solução aqui estudada, após serem pesados os prós e contras da sua implementação através do estudo de mercado e da pesquisa intensiva sobre abordagens a seguir, pode-se concluir que esta é uma excelente oportunidade para revolucionar o sistema da Glintt-HS, fazendo-o corresponder às expectativas de todos os seus utilizadores e tornando-o uma presença ainda mais forte no mercado onde este se insere.

2.5.2.3 Gênese da Ideia

O terceiro elemento descreve-se como o nascimento e desenvolvimento da oportunidade até atingir o patamar de ideia concreta. É um processo evolutivo, onde podem ser usados métodos como *brainstorming* ou a criação de um armazenador de ideias, onde as ideias vão sofrendo alterações durante as sessões de *brainstorming* ou através das ideias que vão sendo armazenadas. Aqui conta muito a opinião interna e externa, podendo estas ideias surgir de um membro da organização ou até mesmo de um cliente ou fornecedor.

Aqui, espera-se que existam sessões de *brainstorming* constantes, para a ideia estar num constante processo de aperfeiçoamento, onde é expectável que todos os intervenientes no processo de desenvolvimento desta solução contribuam com as suas opiniões.

2.5.2.4 Seleção da Ideia

Este é um ponto fundamental, dada a dificuldade de selecionar a melhor ideia a ser implementada dada a quantidade de ideias, oportunidades que vão surgindo e o tempo e financiamento que a organização tem disponível. Dada a dificuldade deste processo, é vulgar aplicar o método de averiguar qual a ideia que pode trazer o maior valor comercial e de consumo.

Na seleção da ideia são pesadas várias variáveis que permitem identificar o valor comercial e de consumo que esta traz, bem como sendo sempre pesados os gastos financeiros e temporais para o desenvolvimento da mesma, identificando que o valor comercial e de consumo que esta ideia pode trazer é bastante superior ao gasto financeiro e/ou temporal.

2.5.2.5 Desenvolvimento do Conceito e da Tecnologia

O último elemento a ser considerado é onde o conceito de negócio é desenvolvido e é classificado como a fase final do FEI. Deve ser baseado em algumas variáveis como o potencial de mercado, necessidades dos clientes, requisitos de investimento, análise da concorrência, incerteza do sucesso do projeto, etc.

Aqui deve ser tomada uma especial atenção aos fatores influentes que se podem observar na Figura 10, podendo estes fatores ser definidos pelas leis, clientes, competidores, economia, etc., pois são estes fatores que influenciam as decisões a ser tomas no desenvolvimento do conceito.

2.5.3 Modelo de Canvas

O modelo de Canvas é uma ferramenta de gestão estratégica que permite desenvolver um novo ou existente modelo de negócio que foi proposta por Alexander Osterwalder. Apresenta-se como um mapa visual dividido em nove blocos. Na Figura 11 pode-se observar o modelo de Canvas para a solução que é apresentada nesta dissertação.

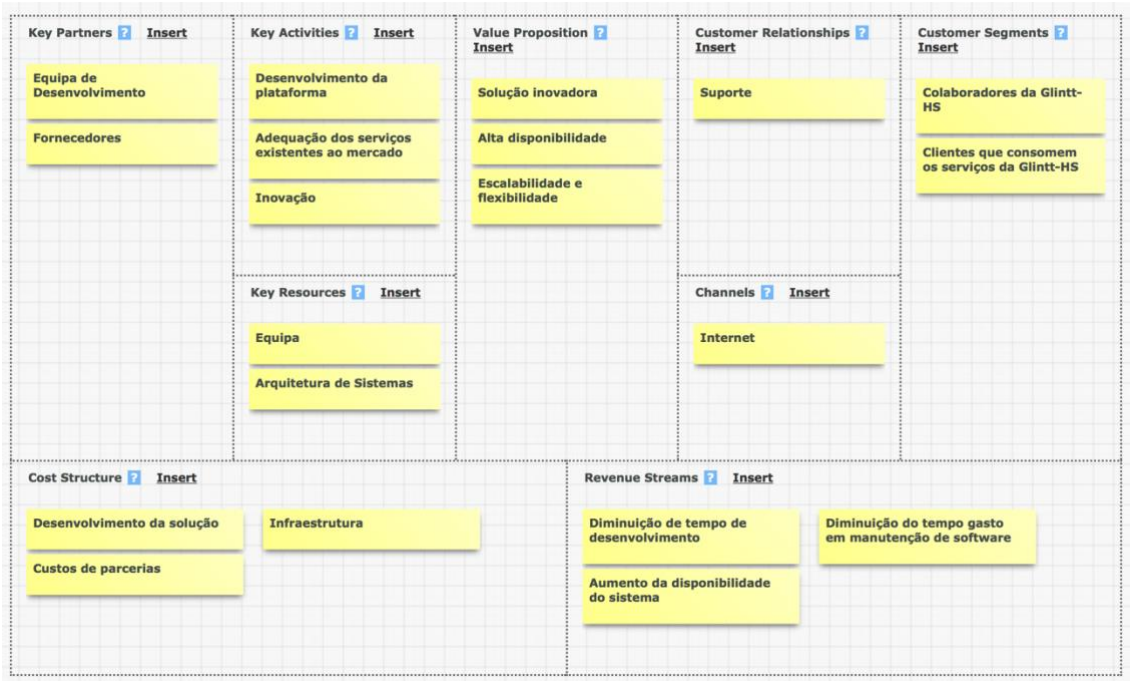


Figura 11 - Modelo de Canvas

De seguida pode-se observar uma análise mais aprofundada do modelo de Canvas apresentado anteriormente.

2.5.3.1 Parceiros Chave

Os parceiros chave identificam a rede de fornecedores e parceiros que se complementam e ajudam uma organização a criar valor.

Foram identificados dois parceiros chave:

- **Equipa de desenvolvimento:** responsável por fornecer o *feedback* relativo ao rumo que o desenvolvimento está a seguir;
- **Fornecedores:** responsáveis por fornecer o apoio necessário ao desenvolvimento da solução.

2.5.3.2 Atividades Chave

As atividades chave são as atividades consideradas fundamentais para a organização produzir valor. São as atividades que precisam de ser notadas para o modelo de negócio ser considerado de sucesso.

Foram identificadas as seguintes atividades chave:

- **Desenvolvimento da plataforma:** tornar a plataforma mais versátil, atual e capaz de ser alterada e desenvolvida sem grandes custos;
- **Adequação dos serviços existentes ao mercado:** é expectável que a plataforma se torne mais versátil e capaz de responder da melhor forma ás necessidades do mercado;

Inovação: inovar nos serviços e produtos que serão desenvolvidos utilizando tecnologia de ponta.

2.5.3.3 Recursos Chave

Os recursos chave são os bens fundamentais da organização para que esta forneça valor aos seus clientes e podem ser categorizados como humanos, financeiros, físicos e intelectuais.

Os recursos chave identificados são:

- **Equipa de desenvolvimento:** é quem é responsável por desenvolver e manter a solução a implementar;

Arquitetura de Sistemas: é uma arquitetura que visa revolucionar todo o sistema da Glintt-HS, tornando-o mais capaz e de acordo com os requisitos que o mercado sugere.

2.5.3.4 Proposta de Valor

A proposta de valor define-se pela combinação dos produtos e dos serviços que a organização fornece aos seus clientes, podendo ser divididos em duas categorias: quantitativos e qualitativos.

O valor que a organização pode trazer aos seus clientes com a solução a implementar:

- **Solução inovadora:** é uma solução inovadora no sentido que irá aplicar conceitos e tecnologias muito recentes;
- **Alta disponibilidade:** tornará o sistema muito mais disponível e isento de falhas;
- **Escalabilidade e flexibilidade:** permitirá que o sistema escale de uma maneira muito simples e com muito pouco custo. Aumentará a flexibilidade o sistema, permitindo que se adapte às constantes mudanças do negócio envolvente.

2.5.3.5 Relações com os Clientes

A relação com os clientes identifica o tipo de relação que a organização terá com os clientes sendo estabelecida uma relação de suporte total para perceber se o rumo tomado está a trazer o impacto desejado aos nossos clientes. Também é estabelecida uma relação de suporte com os colaboradores da Glintt-HS de modo à interiorização desta nova arquitetura a ser implementada.

2.5.3.6 Canais

Identificam o meio onde a organização irá fornecer o valor aos clientes, sendo nada mais nada menos que a internet. Todos os serviços serão expostos na internet podendo ser consumidos pelos nossos clientes.

2.5.3.7 Segmentos de Clientes

Os clientes devem ser divididos em segmentos baseados na maneira como o produto ou serviço da organização afeta as necessidades específicas de cada segmento.

Identificam-se os seguintes segmentos:

- **Colaboradores da Glintt-HS:** clientes indiretos que serão os grandes privilegiados da solução dado o aumento da facilidade de desenvolver e manter novas funcionalidades;

- **Clientes consumidores:** irão usufruir de um sistema muito mais disponível, confiável, com um grande desempenho e estando na vanguarda da tecnologia.

2.5.3.8 Estrutura de Custos

A estrutura de custos define os custos envolvidas para levar a cabo o modelo de negócio e foram identificados os seguidos custos:

- **Desenvolvimento da solução:** custos relacionados com o tempo de desenvolvimento da solução e de aprendizagem de novas metodologias de trabalho;

- **Infraestrutura:** necessário o investimento em infraestrutura para responder às necessidades da nova arquitetura. Investimento em filas de mensagens e intermediários;

- **Custos de parcerias:** licenciamento relacionado com o software de parceiros como é o caso dos ESBs.

2.5.3.9 Fontes de Receita

Fontes de receita são caracterizadas pelas metodologias que uma organização segue para os segmentos de clientes comprarem os seus produtos ou serviços. Este bloco não se caracteriza apenas pelos lucros ganhos com clientes, mas também com a redução de gastos que a Glintt-HS poderá atingir, o que leva às seguintes fontes de receita:

- **Diminuição do tempo de desenvolvimento:** o tempo gasto em desenvolvimento irá diminuir consideravelmente podendo esse tempo ser gasto em outros temas como a inovação;

- **Diminuição do tempo gasto em manutenção:** a manutenção do software pode ser diminuída significativamente, também permitindo que este tempo seja investido noutros temas;

- **Aumento da disponibilidade do sistema:** o sistema tornar-se-á mais disponível aumentando a satisfação dos clientes.

3 Análise e Design

Neste capítulo é feita uma análise de algumas soluções que visam ajudar a elaborar uma arquitetura que responda aos problemas apresentados na secção 1.2. São definidas algumas métricas para melhor perceber a diferença entre as duas arquiteturas.

Aqui também será apresentada a arquitetura que se pretende desenvolver segundo o modelo 4+1 proposto por Philippe Krutchen (Krutchen, 1995). Esta arquitetura tem como objetivo substituir na íntegra a arquitetura atual de modo a resolver os problemas apresentados na secção 1.2. Para exemplificar como esta será construída iremos apenas focar num caso de uso já existente que será apresentado na secção 3.2.1. Este caso de uso servirá como ponto de partida para todos os desenvolvimentos realizados segundo uma arquitetura orientada a eventos integrada com um ESB. Serão desenvolvidas duas soluções, utilizando os dois sistemas de filas de mensagens estudados no capítulo 2, o *RabbitMQ* e o *Apache Kafka.* O design tanto para uma solução como para outra é exatamente mesmo, sendo a única diferença o sistema de filas de mensagens que utiliza.

3.1 Soluções Alternativas

Aqui são discutidas as soluções alternativas que permitem resolver os problemas apresentados pela arquitetura atual da Glintt-HS e atingir os objetivos referidos na secção 1.3. Para avaliar qual será a melhor alternativa, as nossas métricas são definidas pelos objetivos que pretendemos atingir: o desempenho, a escalabilidade e a flexibilidade.

Quanto à primeira métrica, o desempenho em termos de tempo de resposta da solução atual pode-se tornar um fator muito crítico dado a quantidade de lógica que pode estar associada a um determinado serviço, tornando estes serviços lentos, suscetíveis a falhas, diminuindo a confiança que se tem. Neste aspeto a utilização de micro serviços pode ser uma grande vantagem, permitindo que os serviços que necessitam de mais poder de processamento sejam replicados de forma isolada, não necessitando de replicar um serviço de granularidade grossa como é o caso da arquitetura atual. Os micro serviços, também como possuem todas as funcionalidades separadas em diferentes serviços, consomem menos recursos, pois quando uma funcionalidade que consome muita RAM ou CPU a processar alguma coisa, este processamento fica apenas ligado a um serviço, sendo que é mesmo possível em casos extremos, que este seja migrado para uma outra máquina para não causar impacto nos restantes serviços. Para além dos micro serviços é aconselhável a utilização de sistemas de filas de mensagens para aumentar a velocidade de resposta do sistema. Isto é possível pois em muitos casos uma determinada funcionalidade não necessita de esperar que todas as funcionalidades associadas sejam processadas para o serviço responder. A utilização de um sistema de filas de mensagens permite que um serviço se preocupe apenas com a sua

funcionalidade e em notificar que tal evento aconteceu. As funcionalidades associadas são depois processadas quando uma notificação é enviada aos respetivos interessados nesse evento. O consumo de recursos destes sistemas aumenta, mas a sua capacidade também aumenta drasticamente permitindo que os sistemas sejam postos sob grande carga e continuem a funcionar corretamente.

Quanto à segunda métrica, o sistema da Glintt-HS embora permita alguma escalabilidade, não pratica a solução ideal. Mais uma vez a arquitetura orientada a micro serviços aumentaria significativamente a capacidade de escalar o sistema, tal como referido para a métrica anterior. Mais uma vez a introdução de eventos vem aumentar ainda mais a escalabilidade, permitindo que os micro serviços que operam como produtores e consumidores também sejam replicados, mas também todo o sistema de filas de mensagens. Com um sistema de filas de mensagens e micro serviços a escalabilidade é facilmente aumentada segundo os 3 diferentes eixos apresentados no capítulo 1, permitindo que todo o sistema de filas de mensagem seja replicado, clonando todo o sistema, permitindo que funcionalidades distintas sejam divididas e permitindo que o sistema seja dividido através da partição de dados, dividindo o sistema em conceitos semelhantes. A escalabilidade com micro serviços requer um outro conceito para funcionar corretamente, balanceamento de carga. É preciso utilizar uma ferramenta que efetue este balanceamento, para distribuir o tráfego por diferentes serviços, aumentando assim a escalabilidade. Por outro lado, os sistemas de filas de mensagens já fornecem estes mecanismos, sendo apenas necessário indicar as replicas existentes, sendo da responsabilidade do sistema de fila de mensagens o balanceamento da carga.

Quanto à terceira métrica, o sistema da Glintt-HS também permite alguma flexibilidade, esta é ainda bastante limitada, pelo que uma solução orientada a micro serviços melhoraria bastante esta vertente. Uma solução orientada a micro serviços tornaria o sistema muito mais flexível, podendo ser alterada apenas uma funcionalidade específica sem causar impacto nas restantes funcionalidades. Contudo, os micro serviços ainda precisam de comunicar uns com os outros, sendo que a maneira tradicional é conseguida através de comunicação Ponto-a-Ponto, que implica que um determinado serviço conheça pelo menos os dados que os serviços com os quais pretende comunicar necessitam para funcionar corretamente. A utilização de uma arquitetura orientada a eventos torna o sistema muito flexível dado que muito facilmente permite a adição de novas funcionalidades sem correr riscos de danificar o que não devia ser modificado. É aqui que o ESB ganha um papel mais importante, trazendo vantagens para o controlo de todo o sistema, ganhando um intermediário que permite gerir todas as integrações necessárias e fornecendo uma camada de abstração, tornando mais simples qualquer alteração que pretenda ser feita ao sistema.

3.1.1 Tecnologias

Nesta secção, são apresentadas algumas conclusões relacionadas com as tecnologias apresentadas na secção 2.4. O *RabbitMQ* e *Apache Kafka* que foram estudados anteriormente apresentam-se então como os sistemas de filas de mensagens a ser comparados. Numa

primeira análise pode-se concluir que estas duas ferramentas são bastante utilizadas. O *RabbitMQ* apresenta-se muito mais maduro a nível de documentação e comunidade, tornando muito simples o processo de aprendizagem. A complexidade da solução do *Apache Kafka* associada ao facto de não ser autossuficiente torna-o mais difícil de aprender e implementar. Ambas as soluções conseguem melhorar o desempenho de todo o sistema. Nicolas Nannoni, publicou na sua tese um estudo aprofundado sobre estas duas tecnologias onde afirma que a principal razão para as pessoas escolherem *Kafka* em detrimento de *RabbitMQ* é o desempenho. Refere também que o *RabbitMQ* consegue superar o *Kafka* se não tiver a persistência de dados ativa, o que dependendo do contexto pode ser benéfico (Nannoni, 2015), contudo de uma forma geral, o *Kafka* parece ser o grande vencedor nesta métrica.

Um dos problemas levantados pelos sistemas orientados a eventos é o tratamento de erros. Como as aplicações comunicam via eventos, não é possível utilizar transações para controlar os erros ocorridos. Existem mecanismos para controlar este grande desafio, sendo que alguns tornam mais difícil o processo de desenvolvimento e outros exigem mais capacidade do sistema. O RabbitMQ, por exemplo, fornece um mecanismo conhecido por AMQP *Transactions*, que funciona como uma transação de base de dados, mas orientada às mensagens. Contudo este mecanismo diminui a taxa de transferência num fator de 250 segundo a Pivotal. Outro mecanismo muito menos custoso e que impacta no desenvolvimento é a utilização de mecanismo de confirmação de mensagens recebidas, ou seja, um nó do Rabbit retorna um reconhecimento para cada mensagem recebida, quer com erro, quer sem erro, é também preciso que do lado do produtor seja implementado uma fila para receber as mensagens de erro e reagir de acordo com o necessário (RabbitMQ by Pivotal, 2018). O *event-sourcing* é um padrão que também permite controlar estes erros, e proceder de acordo com o sucedido. Este padrão também é muito útil quando se pretende manter um histórico de todas as ações ocorridas no sistema. Ambos os sistemas fornecem mecanismos de tolerância falhas, permitindo replicar vários nós para que caso ocorra um erro inesperado num dos nós, automaticamente é destacado outro nó, também permitindo que sejam replicados vários produtores e consumidores, ou seja, escalar dividindo todo o sistema e em conceitos semelhantes.

Posto isto, decidiu-se implementar duas soluções, uma com *RabbitMQ* e outra com *Apache Kafka* de modo a comprovar o estudado, e conseguindo assim perceber qual o sistema que melhor se adapta às necessidades da Glintt-HS.

Quanto aos ESBs, o Mule ESB é uma ferramenta que tem vindo a crescer imenso, apresentando uma documentação muito completa e intuitiva. O WSO2 ESB também se encontra num grande processo de crescimento, onde apresentam documentação muito completa e intuitiva, contendo bastantes exemplos de como construir uma aplicação utilizando o seu ESB. A documentação do WSO2 apresenta, contudo, uma interface menos bonita do que a prestada pelo Mule ESB, parecendo esta última uma documentação mais atual em termos de apresentação. Ambos os ESBs pecam na vertente de suporte quando é necessário encontrar informação e apoio para a resolução de problemas que vão surgindo. O *dataweave* do Mule,

também torna este sistema mais simples de manipular os dados que transacionam nos fluxos de mensagens do ESB, permitindo que com um simples conetor sejam feitas transformações e filtragens de dados.

Em suma, as soluções apresentadas pelo WSO2 são bastante boas, sendo a sua maior força estarem todos dentro do mesmo ambiente WSO2, removendo a probabilidade de erros relacionados com compatibilidades, por exemplo.

Tanto o WSO2 como o Mule ESB apresentam-se como soluções bastante capazes de satisfazer as necessidades da Glintt-HS, mas dado que a na Glintt-HS já se encontra a trabalhar com o Mule ESB, será este o ESB a utilizar neste trabalho.

3.2 Modelo 4+1

É aqui que é descrita com mais pormenor a arquitetura que se pretende desenvolver nesta dissertação. São apresentadas as diferentes vistas do modelo 4+1, a vista de cenários, vista lógica, vista de processos e a vista de implantação, excluindo apenas a vista de implementação, pois esta apresenta uma grande importância para esta arquitetura.

3.2.1 Vista de Cenários

Nesta secção é apresentada a vista de cenários onde se pode observar o diagrama de casos de uso que foi escolhido para representar esta solução arquiteturalmente. Na Figura 12 podemos observar o referido diagrama.

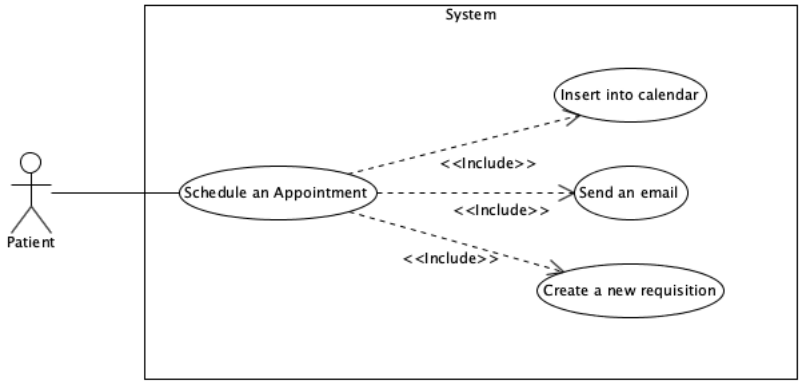


Figura 12 - Diagrama de Casos de Uso

Como é possível observar, iremos prestar atenção apenas ao caso de uso que se tem vindo a falar ao longo desta dissertação. A marcação de uma consulta. Aqui podemos observar que quando o paciente efetua a marcação, é suposto então inserir no calendário, enviar um email e criar um requerimento. Neste momento, existe um único serviço que possui toda esta lógica, sendo por isso impossível que os objetivos definidos na secção 1.3 sejam atingidos.

Para compreender melhor as entidades que foram abordadas anteriormente, podemos observar na Figura 13 como estas se relacionam. Estas são principais entidades de negócio que são utilizadas no âmbito do caso de uso apresentado anteriormente.

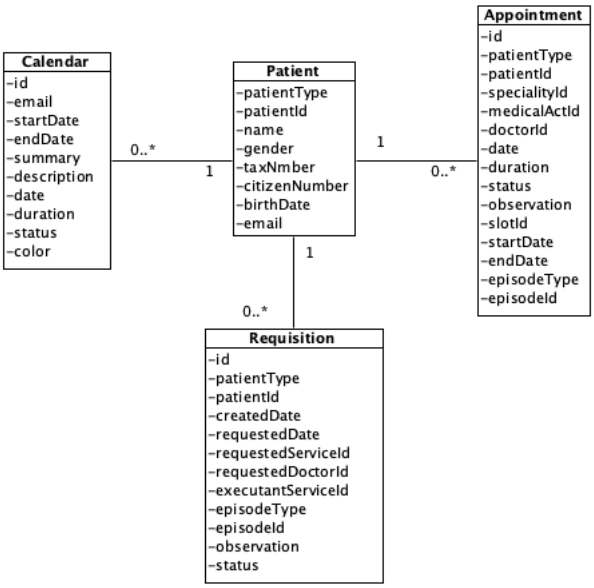


Figura 13 - Diagrama de Classes

Estas entidades têm como entidade principal o paciente, sendo que um paciente pode ter vários registos no calendário, várias marcações e vários requerimentos. Um requerimento representa um exame que um médico pode pedir para o paciente realizar. O calendário representa um calendário interno que pode mais tarde ser utilizado para integrar com uma aplicação mobile, adicionando estes registos no calendário ao Google Calendar ou ao Outlook por exemplo. A marcação em si, consiste numa marcação de consultas ou exames, que neste caso em particular apenas será utilizado o conceito de marcação de consulta. Estas marcações são sempre associadas a um paciente, mas possuem relações com outras entidades que não estão aqui representadas, como é o caso de um ato médico, de uma especialidade e de um médico. Uma marcação, por exemplo, é feita para um determinado ato médico, Consulta de Urologia, associado à especialidade de Urologia e associada também a um determinado médico que tem disponibilidade para efetuar a consulta de urologia na data e local pretendido.

Para compreender como este caso de uso irá desempenhar as suas funcionalidades é possível observar um diagrama de sequência que ilustra essas funcionalidades e como todo o sistema irá comunicar entre si.

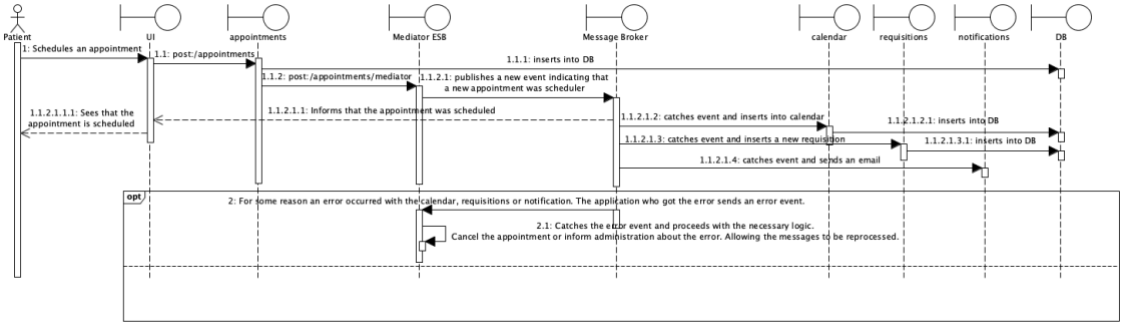


Figura 14 - Diagrama de Sequência

Na Figura 14 podemos observar o referido diagrama e interpretá-lo da seguinte maneira:

1. Um utilizador efetua uma marcação de uma consulta;
2. É enviado um pedido HTTP para o serviço de marcações;
3. O serviço de marcações insere a marcação e efetua uma chamada HTTP para o mediador;
4. O mediador publica um evento a informar que foi marcada uma consulta e envia uma resposta de sucesso;
5. O evento é publicado para o *Message Broker* e será entregue aos respetivos consumidores através de uma notificação sem ordenação, dado que neste caso não é importante;
6. Os consumidores consomem a notificação por uma ordem aleatória e de forma assíncrona;
7. O serviço do calendário insere um novo registo na base dados; O serviço de notificações envia um email; O serviço de requerimentos insere um novo registo na base de dados;
8. Se tudo correr bem, o caso de uso termina.

*. A qualquer momento ocorre um erro:

a. Caso o erro aconteça nos consumidores ou no mediador, é enviado um evento informar que aconteceu um erro, e o mediador recebe esse erro e age de acordo com o necessário;

b. Caso o erro aconteça no serviço de marcações a inserção da consulta é cancelada e o caso de uso termina.

3.2.2 Vista Lógica

Aqui serão apresentados alguns diagramas de componentes com uma granularidade mais grossa e um diagrama de granularidade mais fina, que permitem compreender o processo e raciocínio para construir a solução aqui documentada. Na Figura 15 podemos observar um diagrama de componentes que representa o primeiro passo que foi dado para construir o sistema, tornando-o mais escalável. Em vez de existir apenas um único serviço com toda a lógica, dividiu-se esse serviço em 4 serviços distintos e cada um possui apenas uma responsabilidade. O primeiro, *appointments*, irá gerir dados relacionados com marcações, o segundo, *requisitions,* irá tratar apenas de requerimentos, o terceiro, *notifications,* é responsável pelo envio de notificações e o quarto, *calendar* é responsável pela gestão do calendário.

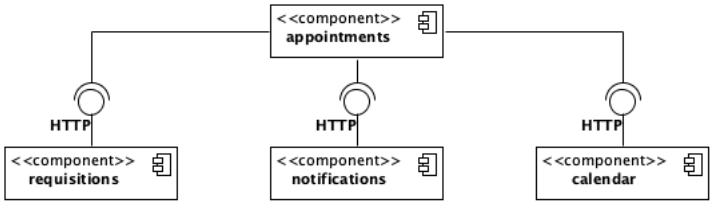


Figura 15 - Diagrama de Componentes, Ponto-a-Ponto

Todos os serviços aqui apresentados são desenvolvidos em Mule. É importante referir que todos os serviços da Glintt-HS, são agora desenvolvidos em Mule, mesmo quando não é necessário o conceito de ESB.

A comunicação entre estes serviços aqui, é do tipo Ponto-a-Ponto, o que implica que o serviço de marcações seja alterado para se adaptar às mudanças que surgem nos outros serviços, ou mesmo quando se pretender adicionar uma funcionalidade nova, este serviço tem de ser alterado. Contudo, caso se tencione escalar um destes serviços, é agora muito mais simples, podendo ser facilmente replicados.

Para que o serviço de marcações não tenha de se preocupar com as mudanças que vão existindo aos restantes serviços, foi então introduzido o conceito do ESB, conforme se pode observar na Figura 16.

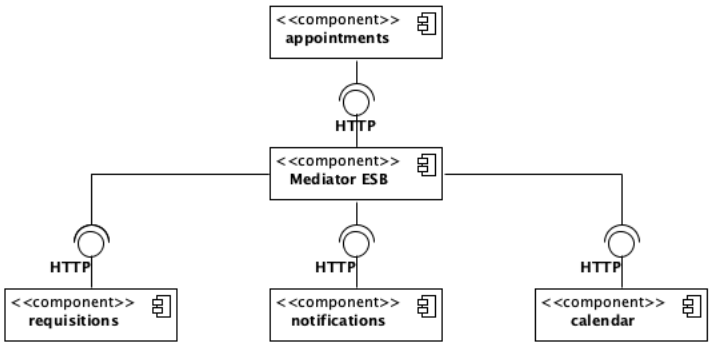


Figura 16 - Diagrama de Componentes, ESB

Na figura anterior, podemos observar então que foi criado um mediador, também desenvolvido em Mule, mas desta vez está a servir como um ESB, a orquestrar as integrações com os restantes serviços. Agora temos flexibilidade, pois podemos adicionar novas funcionalidades ou modificar sem qualquer problema. O serviço de marcações não irá ser alterado, sendo apenas necessário atualizar o ESB. Para finalizar, foi introduzido o conceito do sistema de filas de mensagens, de modo a aumentar e facilitar a escalabilidade, mas também tornando todo o sistema mais agnóstico entre si, abandonando assim a comunicação Ponto-a-Ponto. Na Figura 17 podemos observar o diagrama representativo desta arquitetura.

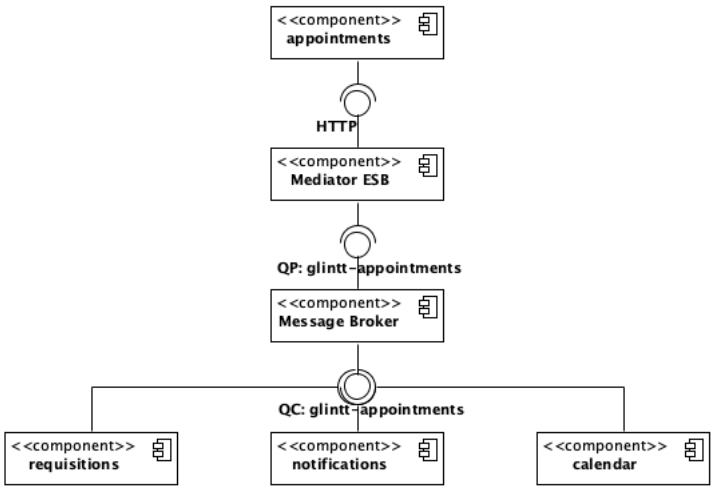


Figura 17 - Diagrama de Componentes, Message Broker

O diagrama anterior, representa então a arquitetura que foi desenhada para resolver os problemas apresentados nesta dissertação. Este é um diagrama com uma granularidade mais grossa, onde podemos observar que agora, para além do ESB, foi introduzido um *Message Broker*, que representa o sistema de filas de mensagens. Este sistema terá a responsabilidade de gerir os produtores e consumidores de eventos, bem como entregar os respetivos eventos aos respetivos consumidores. Com esta arquitetura podemos então escalar o nosso sistema, podemos modificar sem causar erros inesperados e tempo de resposta aumenta consideravelmente, dado que uma marcação agora apenas se preocupa em efetuar a marcação e publicar um evento. O papel do mediador, passa pela abstração de configuração do sistema de mensagens, permitindo que este sistema seja trocado com apenas a substituição de um componente. Também caso exista necessidade de uma marcação passar a publicar para mais alguma fila de mensagens, é muito fácil de o fazer, bastando adicionar no mediador uma publicação nova.

Para compreender melhor como esta arquitetura se irá comportar, é apresentado um diagrama de componentes com uma granularidade mais fina na Figura 18.

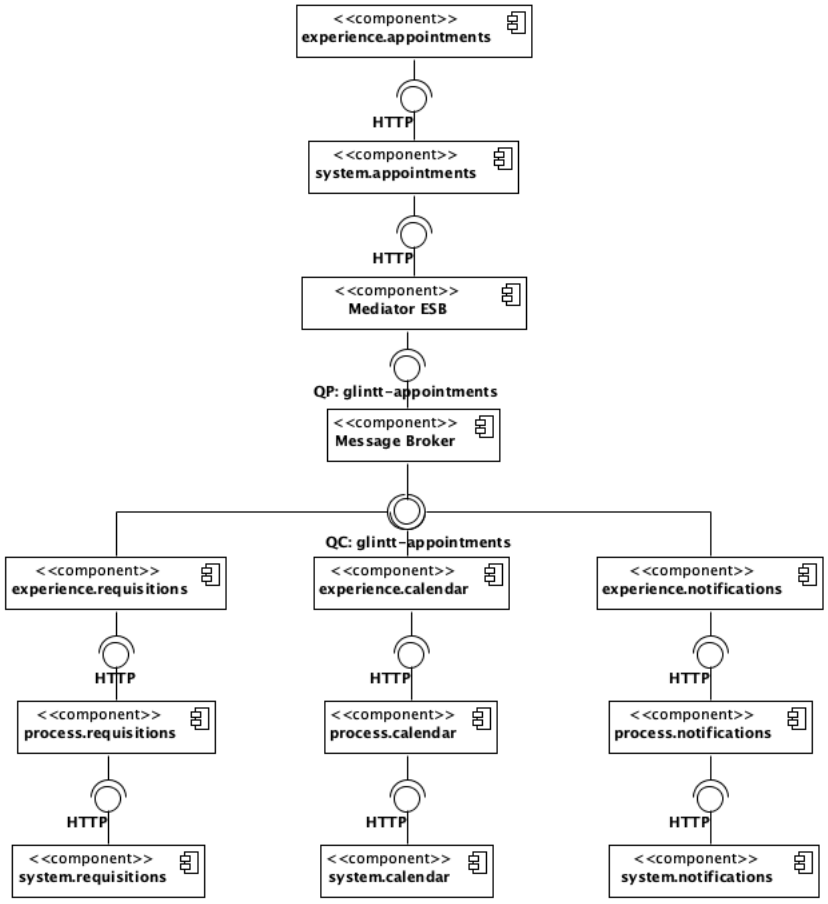


Figura 18 - Diagrama de Componentes, detalhado

Na figura anterior podemos então observar um diagrama mais detalhado de como este sistema será construído. Os componentes apresentados representam diferentes aplicações Mule que seguem uma metodologia descrita na secção 2.1.4, API-led Connectivity. As aplicações podem ser agrupadas em 3 grupos: *appointments* com 2 camadas, experiência e sistema; *calendar, notifications e requisitions* com 3 camadas, experiência, processo e sistema; *appointments.mediator* com apenas 1 camada. Estas camadas representam diferentes responsabilidades. A camada de experiência é normalmente utilizada para definir regras aplicadas aos diferentes consumidores da API, sendo comum fornecer duas camadas de experiência, por exemplo, uma para apresentar resultados para aplicações *mobile,* e outra para apresentar resultados para aplicações *desktop.* A camada de experiência da aplicação relacionada com marcações possui esta referida lógica. As restantes camadas de experiência são aqui usadas de uma maneira distinta, adaptadas para funcionarem orientadas a eventos. Estas camadas têm a responsabilidade de receber eventos de um determinado tipo, transformar a mensagem de acordo com o seu negócio, e implementar sua funcionalidade nas camadas de processo e sistema. As camadas de processo são aqui utilizadas para comunicar com diferentes camadas de sistema, quando necessário. Por exemplo, caso um evento para inserir no calendário não possua um assunto definido, pode ser consultada uma camada de sistema para obter uma configuração que indica qual a o assunto que deve ser utilizado por

defeito. Na camada de sistema é onde está assente a lógica específica de uma entidade. Neste caso as camadas de sistema têm a única responsabilidade de receber a entidade que são responsáveis e inserir a mesma na base de dados, com exceção da camada de sistema da marcação, que também precisa comunicar com o mediador para informar que efetuou uma marcação.

A comunicação entre diferentes grupos de aplicações é realizada através de eventos, tendo sido configurada apenas uma fila de mensagens (*glintt-appointments*) como se pode observar na Figura 18. Esta fila de mensagens é subscrita pelos consumidores representados pelos componentes *experience.calendar, experience.requisitions* e *experience.notifications.*

Esta arquitetura permite resolver os vários problemas apresentados no capítulo 1, fornecendo capacidades de escalabilidade enormes, através dos sistemas de filas de mensagens que podem ser replicados em várias máquinas, bem como os respetivos consumidores de eventos. A facilidade com que este sistema pode ser alterado ou adicionadas novas funcionalidades é notável, não só pelo sistema ser orientado a eventos, mas também pelo mecanismo adotado, o *API-led Connectivity,* promovendo a orquestração e modularização, sendo cada aplicação segmentada por camadas com diferentes responsabilidades.

Para o tratamento de erros do sistema optou-se pela utilização do padrão descrito na secção 3.1, sendo criada uma fila de mensagens no mediador, que irá receber todos os erros lançados pelos consumidores e agir conforme necessário. Por exemplo, ocorre um erro a inserir os dados no calendário. É na aplicação relacionada com o calendário que tem de ser despoletado um evento a informar do ocorrido. Do lado do mediador, tem de existir um consumidor para tratar estes erros ocorridos, que pode ou cancelar a marcação ou mudar o estado da mesma para ser processada mais tarde. Este mecanismo aumenta a complexidade de desenvolvimento do sistema.

3.2.3 Vista de Processos

Aqui é observada uma vista dos processos existentes e como estes podem comunicar entre si. Um processo representa uma instância de memória partilhada pelos respetivos *threads.* Na Figura 19 podemos observar os referidos processos.

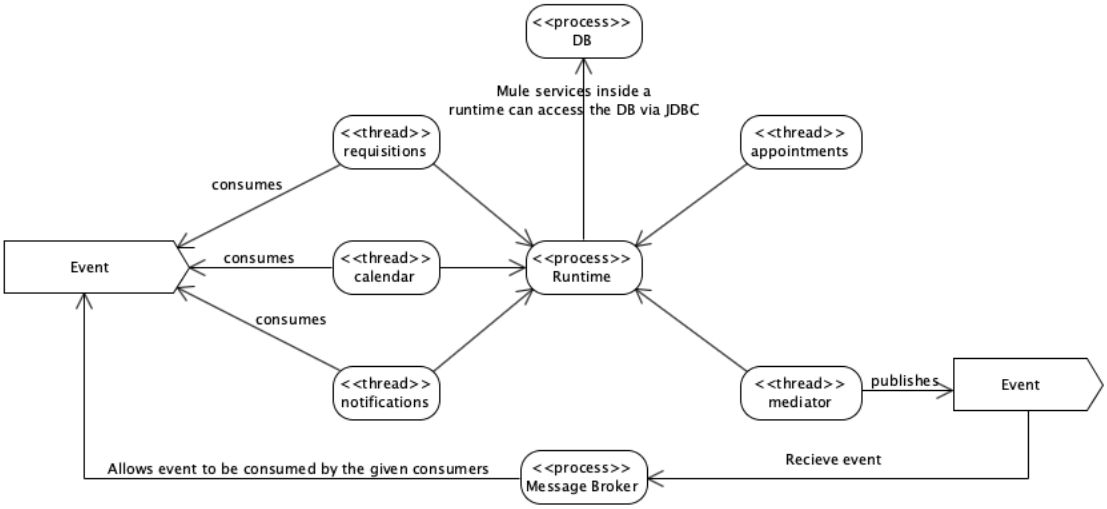


Figura 19 - Diagrama de Processos

Como podemos observar, existem 3 processos principais, o *runtime*, onde estarão todos os micro serviços, o Message Broker, que representa o sistema de filas de mensagens e a base de dados que será acedida pelos micro serviços. Como podemos ver o sistema de filas de mensagens é completamente independente do *runtime* que está a ser utilizado, podendo ele ser um servidor de IIS ou um Mule *Runtime*.

3.2.4 Vista de Implantação

Nesta secção será dado a conhecer o diagrama de implantação que representa uma das muitas maneiras que esta solução poderia ser implantada. Na Figura 20 podemos observar o referido diagrama.

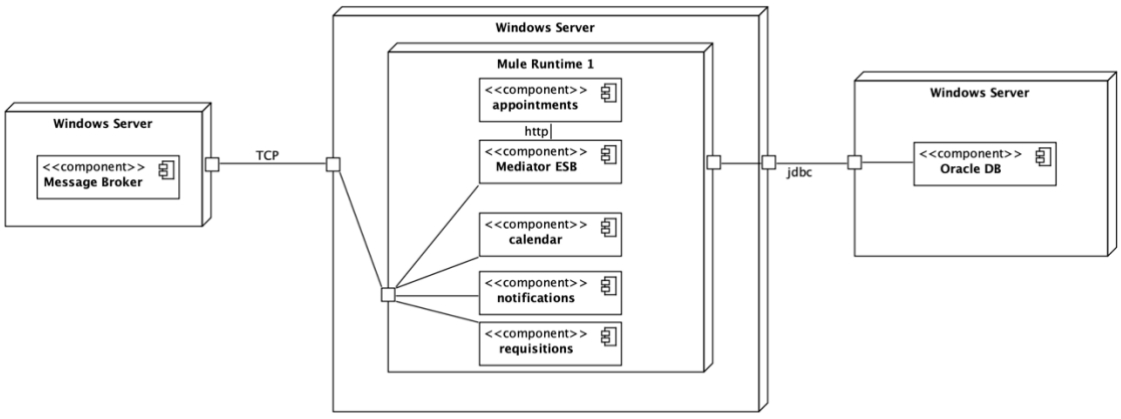


Figura 20 - Diagrama de Implantação

No diagrama anterior podemos observar que existem 3 servidores com diferentes responsabilidades. No primeiro, temos o nó do Sistema de Fila de mensagens, completamente independente dos seus produtores e consumidores de eventos, no segundo temos o *runtime*,

onde estão a correr os micro serviços. Os produtores e consumidores comunicam com o sistema de filas de mensagens através de um protocolo TCP ou uma variante do mesmo, o AMQP consoante o sistema de filas de mensagens utilizado. O terceiro servidor contém o sistema de base de dados Oracle. Os serviços podem aceder a esta base de dados via JDBC (*Java Database Connectivity*)[5].

Supondo que existia a necessidade de escalar este sistema e de aumentar a capacidade de consumir mais rapidamente os eventos publicados, poderíamos facilmente escalar um sistema de um ponto de vista funcional, onde seria possível replicar os consumidores numa máquina diferente, conforme se pode observar na Figura 21.

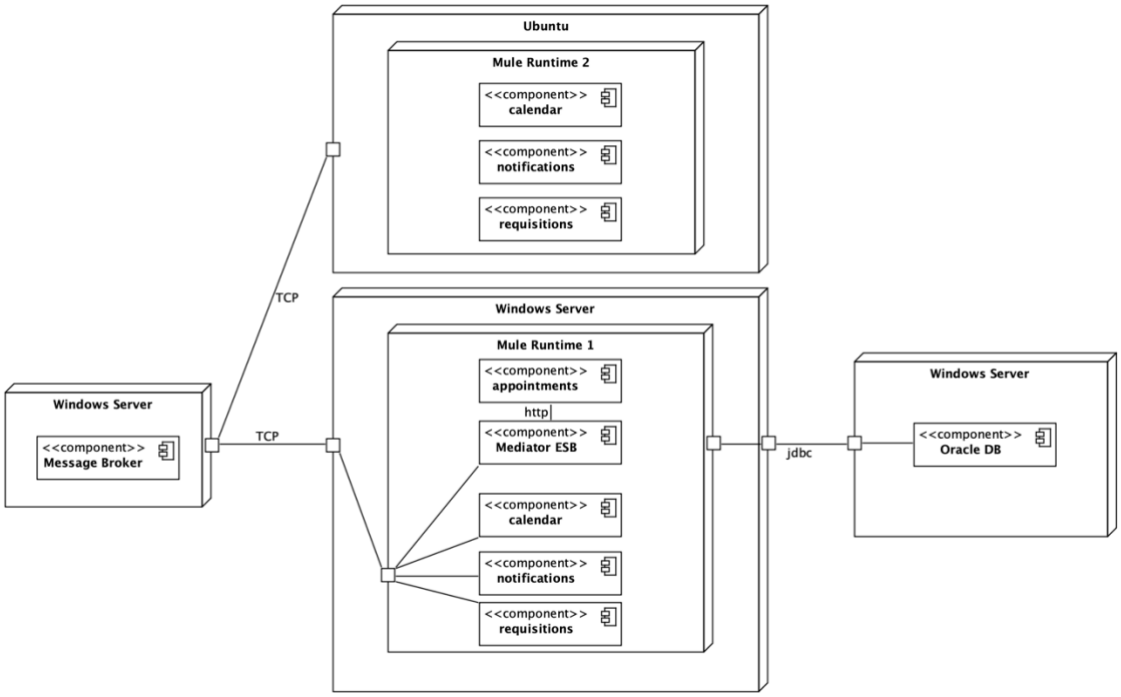


Figura 21 - Diagrama de Implantação, Escalabilidade

Na figura anterior podemos observar dois *runtimes* distintos, tendo estes sido duplicados por funcionalidade. De salientar que estes *runtimes* podem ser instalados tanto em servidores Windows como Linux. É muito comum a necessidade de aumentar a capacidade de consumir os eventos, não só para aumentar a velocidade com que todos os eventos são processados, tirando partido de paralelismo, mas também para reduzir a carga exercido num único consumidor. Este é apenas um exemplo de como esta arquitetura poderia ser facilmente escalada. Além disto, ainda é possível que apenas um nó do sistema de filas de mensagens não consiga distribuir as mensagens de uma forma eficiente, armazenando bastantes mensagens na fila de espera, antes de estas ser entregues aos respetivos consumidores. Para combater este problema também é possível criar mais nós para o sistema de filas de mensagens, criando

[5] https://www.oracle.com/technetwork/java/javase/jdbc/index.html

assim um *cluster* (conjunto de nós). Na Figura 22 podemos observar um exemplo de uma arquitetura que possui os consumidores replicadas e também dois nós.

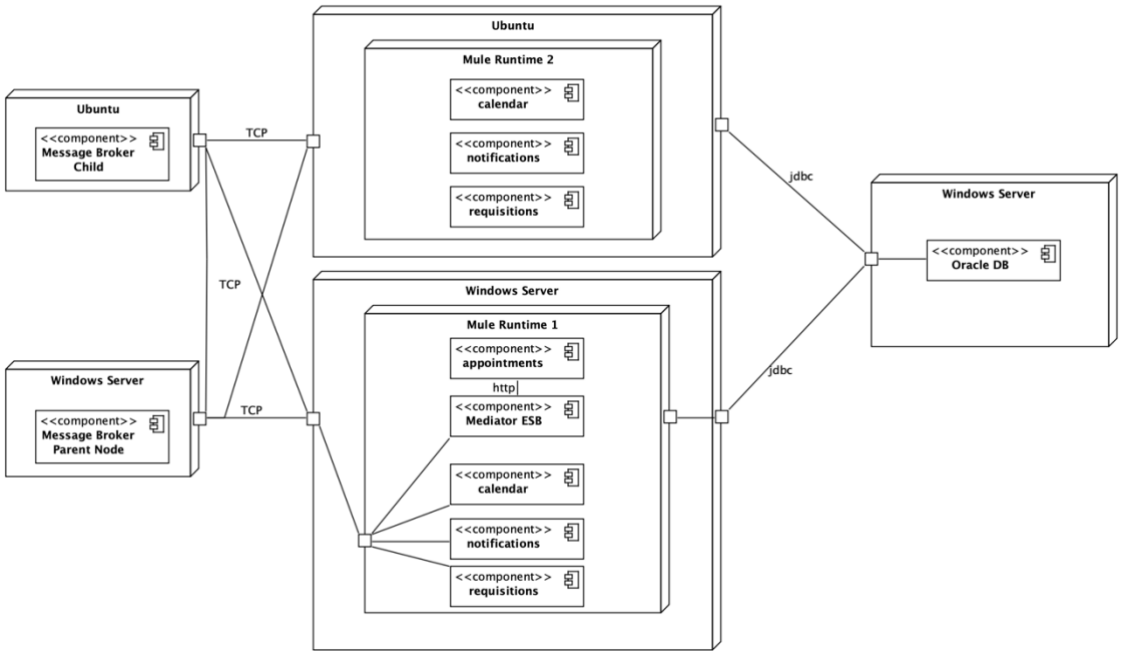


Figura 22 - Diagrama de Implantação, distribuído

Na figura anterior podemos observar uma arquitetura orientada a eventos integrada com um ESB bastante distribuída. Aqui existe mais um nó do sistema de filas de mensagens, aumentando a capacidade de processamento de eventos. Estes dois nós comunicam entre para realizar a distribuição dos eventos. É muito vantajoso a existência de um *cluster* pois o que acontece muitas vezes é que o Sistema de Filas de Mensagens está a receber uma determinada quantidade de eventos, mas não os consegue processar todos ao mesmo tempo que vai recebendo, armazenando as mensagens na fila de espera. Com dois nós é feita uma distribuição dessas mensagens, sendo esta sempre coordenada pelo nó principal.

4 Implementação

Neste capítulo é abordado todo o processo relacionado com a implementação da nova arquitetura. São abordados os dois sistemas de mensagens estudados e também o Mule ESB, sendo também especificado o processo de criação de micro serviços a partir de um serviço de granularidade mais grossa. Foi realizada a implementação de duas soluções idênticas que serão testadas no capítulo seguinte.

As duas soluções foram implementadas recorrendo ao Anypoint Studio como IDE. É um IDE gráfico baseado no eclipse desenvolvido pela Mulesoft que permite desenvolver e testar fluxos Mule.

Uma das grandes vantagens fornecidas pelos sistemas de filas de mensagens é a escalabilidade, mas para esta implementação não foi tirado partido desta vertente devido à falta de recursos para o realizar.

4.1 RabbitMQ

Conforme estudado na secção 2.4.1 o RabbitMQ apresenta-se como uma solução bastante simples de configurar e utilizar. Para a arquitetura em questão foram abordadas as diferentes funcionalidades que o RabbitMQ fornece, com a exceção da replicação de consumidores.

Este sistema apresenta algumas configurações que são necessárias definir para o correto funcionamento de todo o sistema. A configuração base passa por definir um *Exchange* que terá a responsabilidade de enviar as mensagens recebidas para as respetivas filas de mensagens que também são definidas para responder às necessidades da arquitetura.

A solução aqui apresentada possui dois *exchanges* distintos. Ambos são relacionados com marcações, mas de tipos diferentes. O RabbitMQ fornece quatro tipos de *exchanges* (Tennakoon, 2017):

- ***Fanout:*** envia mensagens para todas as filas de mensagens que estão à escuta no *Exchange;*

- ***Direct:*** apenas envia mensagens para as filas de mensagens que foram especificadas através de uma chave de roteamento;

- ***Topic:*** semelhante ao direct, mas permite que a chave de roteamento utilize os *wildcards* "#" e "*" para encontrar as filas de mensagens. O "#" representa zero ou mais palavras e o "*" representa apenas uma palavra;

- ***Headers:*** também semelhante ao direct Exchange, mas desta vez de uma chave de roteamento, utiliza um conjunto de chave-valores para decidir quais as filas de mensagens que vão receber as mensagens. Permite que um o envio de mensagens para uma fila seja mais filtrado.

Foi configurado um *exchange* do tipo *fanout* que irá receber os eventos relacionados com a marcação de consulta. A publicação para este *exchange* é realizada pelo mediador, e estes serão consumidos por todos os consumidores definidos. Sendo este *exchange* do tipo *fanout,* o evento é enviado para todos os consumidores, sendo processado por todos eles. Se por ventura, fosse necessário efetuar um log de todas as ações, ou mesmo implementar o padrão event-sourcing, poderia ser utilizado um *exchange* do tipo *direct*, onde a publicação seria feita apenas para a fila de mensagem especificada.

Além destas configurações é possível especificar se o *exchange* e a fila de mensagens devem sobreviver quando um nó do RabbitMQ reinicia. Nesta implementação foi considerado que todos os *exchanges* não se devem perder quando o sistema é reiniciado e as filas de mensagens são apagadas assim que o último (neste caso único) consumidor é desligado. Num ambiente real estas filas de mensagens deveriam ser apagadas apenas ao fim de algum tempo sem ser utilizadas, sendo possível através da configuração *exclusive* ao invés de *auto-delete.*

Também como foi referido na secção 2.4.1 o RabbitMQ possui uma interface gráfica que foi bastante utilizada para validar todos os *exchanges* e filas de mensagens que foram criados, bem como monitorizar todo o sistema.

4.2 Apache Kafka

O Apache Kafka apresentou-se como uma solução mais simples de configurar do que aquilo que foi estudado nas secções anteriores. O único entrave na configuração inicial deste sistema de mensagens foi a necessidade de instalar o *ZooKeper* (ver secção 2.4.2) e de configurar o Kafka para comunicar com esta ferramenta. Como o Kafka mantém o controlo das mensagens por um determinado período de tempo ou até atingir um limite não são precisas tantas configurações a nível de consumidores e produtores. Apenas três configurações foram necessárias para o correto funcionamento deste sistema:

- **Tipo de operação:** se funciona como consumidor ou produtor;
- **Tópico:** a fila de mensagens em questão, para onde serão publicadas ou lidas as mensagens;
- **Chave:** meramente indicativa para associar a mensagem enviada (apenas configurada nos produtores);
- **Partições:** as partições do tópico. Permite que várias partições processem as mensagens em paralelo.

Na arquitetura implementada com o Kafka seguiu-se a mesma lógica que a implementação com RabbitMQ. Existe um mediador que publica o evento relacionado com marcação de consultas. Este mediador publica as mensagens para um tópico denominado por g*lintt-appointments* que será consumido pelos respetivos consumidores, conforme realizado pela implementação com *RabbitMQ.* Todos os consumidores aqui criados têm de pertencer a um grupo de consumidores diferente para consumirem a mesma mensagem, caso contrário, se pertencessem ao mesmo grupo, existiria uma espécie de disputa para a obtenção da mensagem. É com este mecanismo que pode ser aplicado o paralelismo.

O Kafka obriga a que os produtores e consumidores possuam um ficheiro de configuração com certas propriedades. Nesta implementação foram utilizadas todas propriedades que o Kafka fornece por defeito.

4.3 Micro Serviços

Para a implementação do sistema aqui documentado, foi preciso efetuar um processo de decomposição de um serviço com uma granularidade muito grossa, em vários serviços com uma granularidade mais fina que dizem respeito a um domínio completamente independente entre si. Este processo foi relativamente simples, tendo sido apenas considerado para cada micro serviço as ações que pretendemos demonstrar neste trabalho. Foram então criados 4 micro serviços que irão conter a lógica necessário para responder ao caso de uso descrito na secção 3.2.1. Foi então criado um serviço relacionado com marcações de consulta, onde foi implementada a lógica de inserir uma nova marcação de consulta, outro serviço relacionado com calendário, onde é adicionado um determinado acontecimento a um calendário, um outro serviço que tem a responsabilidade de enviar notificações, neste caso, um email, e ainda um outro serviço que trata de requerimentos hospitalares, sendo aqui inserido apenas um novo requerimento. Deste modo, em vez de um único serviço com toda a lógica lá estipulada, agora existem 4 serviços, completamente independentes que apenas se preocupam com o seu negócio, não necessitando de conhecer o negócio de outros serviços.

Este micro serviços são desenvolvidos em Mule e segundo o padrão API-led *Connectivity* conforme descrito na secção 3.2.2. Na secção seguinte é possível observar alguns exemplos de como estes micro serviços foram construídos.

4.4 Mule ESB

Como referido anteriormente, este foi o ESB escolhido para implementar a arquitetura apresenta nesta dissertação.

As duas alternativas implementadas apresentam apenas uma diferença, o sistema de mensagens que utilizam, sendo que uma utiliza RabbitMQ e outra Apache Kafka. O Mule ESB apresenta conetores pré-definidos para ambas os sistemas, tornando esta integração bastante

simples e intuitiva. As dependências e configurações dos projetos Mule são baseados no *Maven*, sendo definidas através de um ficheiro XML, conhecido por "POM.XML". O *Mule Exchange* é um repositório de conetores para o *Anypoint Studio*, de onde foram instalados os conetores do RabbitMQ e do Apache Kafka. Ao instalar estes conetores ainda é necessário adicionar as dependências ao ficheiro POM de cada um dos projetos, conforme se pode observar na Figura 23 e Figura 24.

```
<dependency>
    <groupId>org.mule.transports</groupId>
    <artifactId>mule-transport-amqp</artifactId>
    <version>3.7.8</version>
</dependency>
```

Figura 23 - Dependência RabbitMQ

```
<dependency>
    <groupId>org.mule.modules</groupId>
    <artifactId>mule-module-kafka</artifactId>
    <version>2.1.0</version>
</dependency>
```

Figura 24 - Dependência Apache Kafka

Como se pode observar nas figuras anteriores, foram utilizadas as versões 3.7.8 do RabbitMQ e 2.1.0 do Apache Kafka.

Num mundo orientado a eventos as aplicações retratadas na Figura 15 podem ser caracterizadas da seguinte forma:

- **Produtores:** a aplicação *system.appointments*, representa um produtor que irá publicar mensagens relacionadas com marcações;

- **Consumidores:** as aplicações *experience.calendar, experience.notifications e experience.requisitions,* representam três consumidores com responsabilidades semelhantes, apenas recebem o evento de uma marcação, e transformam a mensagem de acordo com a funcionalidade que implementam, esta funcionalidade é apenas implementada na camada de sistema das referidas aplicações;

- **Produtor e Consumidor***:* a aplicação *appointments.mediator* funciona como consumidor e produtor, irá consumir os eventos relacionados com marcações de consulta e é onde será feita a publicação para as filas relacionadas com o calendário, notificações e requisições; esta aplicação faz com que caso se pretenda adicionar mais alguma lógica á marcação de uma consulta, apenas é preciso implementar uma nova aplicação, e adicionar a sua publicação nesta aplicação, não causando impacto nos serviços que já estão a funcionar.

4.4.1 RAML

Como referido na secção 2.4.4, foi utilizado RAML para a elaboração das APIs, adotando uma abordagem "desenho primeiro". A ideia consiste em desenhar toda a API especificando os recursos disponíveis, parâmetros, respostas, etc.

Nesta secção será apresentado o RAML utilizado para a definição da API relacionado com as marcações, onde é demonstrado o RAML base e como pode ser utilizado da maneira mais eficaz, tirando partido das capacidades de definição de bibliotecas e de tipos de recurso.

Na Figura 25 podemos observar a definição da API *Appointments*:

```
#%RAML 1.0
title: Thesis Appointments API
version: v1.0
baseUri: http://localhost:8086/api/system/rabbit
mediaType: application/json
description: Thesis appointments API to be used as an event driven system

uses:
  library: libraries/datatypes.raml

resourceTypes:
  collection: !include resourcetypes/collection.raml
  collection_item: !include resourcetypes/collection_item.raml

/appointments:
  type: collection
  post:
```

Figura 25 – RAML base, Appointments

Ao observar a figura anterior podemos concluir que a API especificada tem os seguintes atributos:

- ***Title*:** O título que ficará visível na documentação;
- ***Version*:** A versão da API;
- ***Base URI*:** O URL base que será o ponto de entrada para a API;
- ***Media Type*:** Indica que API opera com JSON;
- ***Description*:** A descrição que também estará visível na documentação;
- ***Uses*:** Declaração de variáveis ou de ficheiros importados. Neste caso indica que a API utiliza a biblioteca "*datatypes.raml*";
- ***Resource Types*:** Importação de ficheiros classificados como tipo de recursos que funcionam como template para todos os recursos da API.

A API apresentada possui apenas um recurso que é o "*appointments*", onde apenas está exposto o método POST, responsável pela criação de uma marcação.

Como podemos observar, não existe qualquer descrição do que este método faz, contudo, como este recurso utiliza um template "*collection*" será apresentada uma descrição baseada no conteúdo da Figura 26.

```
#%RAML 1.0 ResourceType
uses:
  library: ../libraries/datatypes.raml

description: Resource to manage <<resourcePathName>>
get?:
  description: Retrieve a list of <<resourcePathName>>
  displayName: Get all <<resourcePathName>>
  responses:
    200:
      body:
        application/json:
          type: array
          items: library.<<resourcePathName | !singularize>>
post?:
  description: Add a new <<resourcePathName | !singularize>>
  displayName: Add new <<resourcePathName | !singularize>>
  body:
    application/json:
      type: library.<<resourcePathName | !singularize>>
  responses:
    201:
      headers:
        Location:
      body:
        application/json:
```

Figura 26 - RAML, Resource Type

Este template através da tag "*resourcePathName*" obtém o nome do recurso onde o template está a ser utilizado e disponibiliza algumas funções como singularize, pluralize, camelize, etc., que tal como o nome indica, permite passar o nome para o singular, plural e fazer com que comece com letra maiúscula. Deste modo podemos observar que para o exemplo presente na Figura 25 que utiliza o template da figura anterior irá originar uma documentação conforme a que se encontra no Anexo 1:

De um modo bastante simples, estas foram algumas das funcionalidades fornecidas pelo RAML utilizadas para definir as APIs. A documentação gerada ainda permite que a API seja testada. No Anexo 1 podemos observar que ao clicar no botão verde, irá ser feita uma chamada ao serviço para criação de uma nova marcação, onde o conteúdo que será enviado foi automaticamente preenchido, baseado no exemplo fornecido na documentação.

4.4.2 Conetores

Como referido na secção 2.4.4 o Mule ESB fornece um vasto leque de conetores que permitem a integração com outros sistemas e manipulação de mensagens seja feita de uma forma

bastante simples e intuitiva. Para o problema em questão, apenas serão aprofundados os mais importantes no âmbito do problema apresentado.

- **Flow –** Numa linguagem de programação tradicional um Flow representa um método e é onde é aplicada a lógica no ESB. Podemos observar na Figura 27:

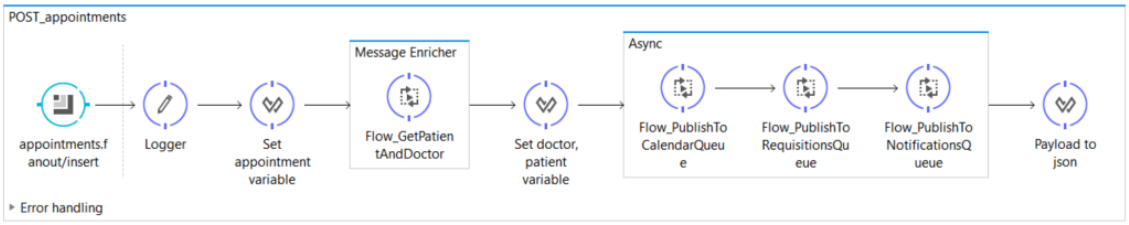


Figura 27 - Flow, exemplo

Na figura anterior podemos observar o Flow com o nome "POST_appointments". Este Flow é a implementação de um consumer de uma queue de RabbitMQ. Neste exemplo podemos observar outros conetores utilizados como o AMQP, *Logger*, *Transform Message*, *Message Enricher*, *Flow Reference* e *Async*, que serão abordados nos pontos seguintes.

- ***Transform Message*:** este é um dos conetores mais utilizados no Mule ESB, e permite que uma mensagem seja facilmente modificada, permitindo converter o seu conteúdo para JSON, XML ou HashMap. Também permite que os vários atributos da mensagem sejam alterados, removidos ou adicionados. Este conetor fornece uma linguagem de transformação conhecida por *dataweave.* Na Figura 28 podemos observar um exemplo da utilização deste conetor:

Output Payload Preview

```
%dw 1.0
%output application/json
---
{
    patient_type: flowVars.appointment.patient_type,
    patient_id: flowVars.appointment.patient_id,
    requisition_date: now as :string {format: "yyyy-MM-dd'T'HH:mm:ss"},
    requested_date: flowVars.appointment.date,
    requested_service_code: flowVars.appointment.speciality_id,
    requested_doctor_code: flowVars.appointment.doctor_id,
    executant_service_code: null,
    episode_type: p("requisition.episode_type"),
    episode_id: flowVars.appointment.patient_id,
    exam_type: null,
    document_id: null,
    observation: flowVars.appointment.observation,
    status: null
}
```

Figura 28 - Dataweave, exemplo

- **HTTP:** conetor que serve como ponto de entrada e de saída da API onde é utilizado. Expõe a API segundo uma configuração HTTP e permite que sejam feitas chamadas a outras APIs;

- ***Database*:** fornece uma interface de ligação a uma base de dados. Permite efetuar chamada de procedimentos, inserir, ler, apagar e editar tabelas da base dados:

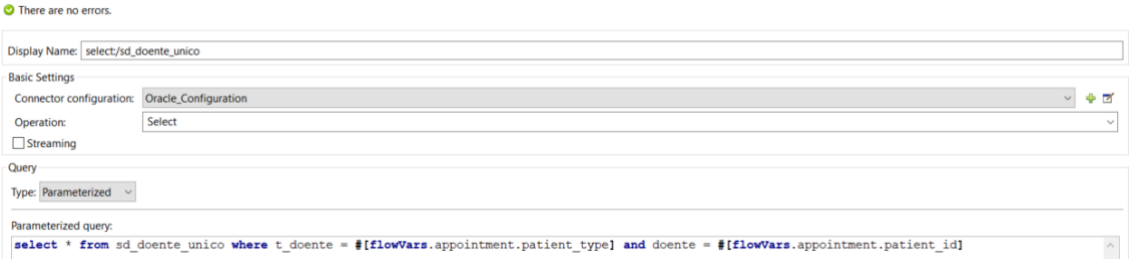


Figura 29 - Base de Dados, exemplo

Na Figura 29 está presente um exemplo de como ler lados de uma tabela da base dados utilizando uma configuração de Oracle. O Mule ESB permite de uma forma muito simples conectar com bases de dados de outros tipos, bastando para isso utilizar um driver de base de dados diferente.

No projeto aqui documentado é utilizado o protocolo *Service Discovery* para realizar a descoberta de serviços e de base de dados. Através de uma determinada chave e, opcionalmente, alguns filtros extra, é possível obter onde determinado serviço e base de dados se encontra disponível. Numa arquitetura orientada a eventos a descoberta de serviços não é tão comum pois a comunicação com esses serviços é realizada através dos sistemas de filas de mensagens, mas o acesso à base de dados é uma prática comum nas camadas de sistema. A configuração "*Oracle_Configuration*" que pode ser observada na figura anterior representa a utilização de uma configuração que funciona através do *Service Discovery.* A aplicação que vai aceder à base de dados só precisa de importar um *Java Bean,* que são componentes reutilizáveis de software escritos em Java (Sun Microsystems, 1997), que possui a lógica de consulta ao *Service Discovery.* A importação do *Java Bean* é feita conforme demonstrado na figura seguinte:

```
<spring:beans>
    <spring:import resource="classpath*:gplatform.infrastructure.utils.xml" />
</spring:beans>
```

Esta consulta a este serviço é feita baseado numa chave definida como propriedade da aplicação, como se pode observar na Figura 30:

```
database.set_basecalldata_enable = false
database.serviceKey = DBGPLATFORM
```

Figura 30 - Propriedades de uma aplicação MULE

Através da chave “DBGPLATFORM”, será obtida a ligação para a base de dados e o nosso serviço conseguirá assim aceder à base de dados que deseja para realizar as operações normais de gestão de dados.

- ***Transactional*:** utilização de transações nas ligações à base dados. Muito prático quando é necessário inserir dados com dependências e é necessário reverter os dados inseridos quando alguma coisa corre mal. Um exemplo da utilização de um componente transacional pode ser observado na Figura 31:

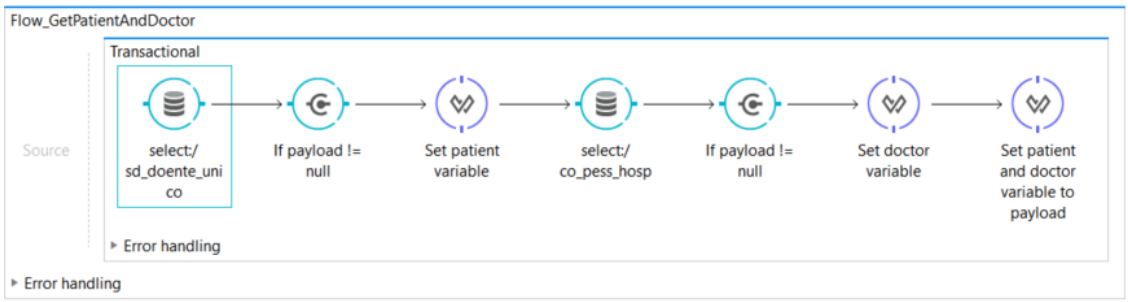


Figura 31 - Acesso à base de dados, exemplo

Na figura anterior, podemos observar a utilização de uma transação. Embora as duas operações na base de dados sejam de consulta, apenas é utilizada uma ligação em vez de duas. Caso a operação em questão se tratasse de uma inserção de dados e se algum dos componentes de validação ou um erro inesperado ocorresse, todo o processo já realizado na base de dados seria revertido.

- ***Validation*:** utilizado para efetuar validações e lançar exceções. Pode ser configurado como na figura seguinte:

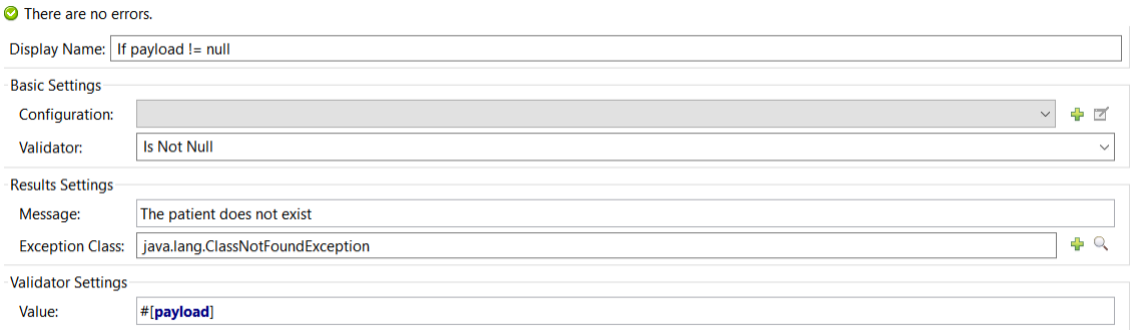


Figura 32 - Componente de validações, exemplo

Na figura anterior observamos um exemplo de como validar se o *payload* não está vazio, e caso este se encontre vazio é lançada uma exceção com a mensagem “The patient does not exist”.

Este conetor fornece um grande número de validações pré-definidas como se pode observar na Figura 33:

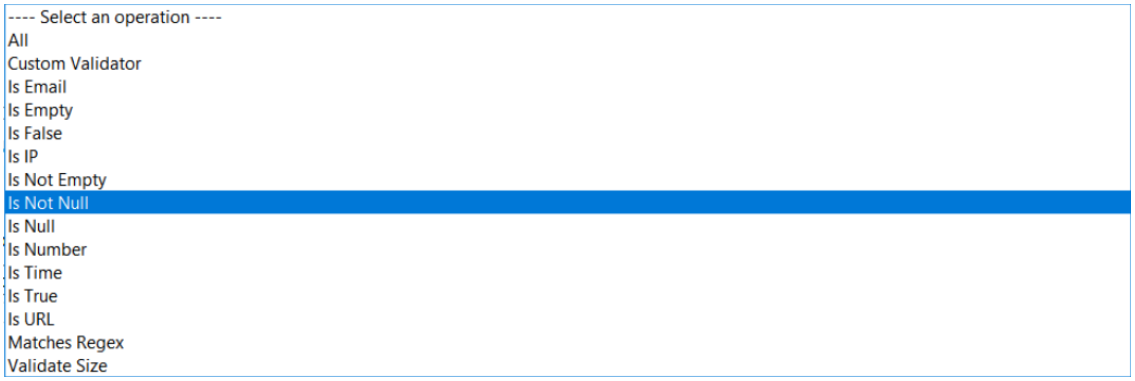


Figura 33 - Opções, conetor de validação

- **SMTP:** utilizado para o envio de emails. Este conetor permite definir como configuração básica um endereço e a porta do servidor de SMTP, bem como um utilizador e password de acesso. Além destas configurações básicas relacionadas com a configuração SMTP, permite, obviamente a definição de dados relacionados com o email como a origem, destino, assunto, CC, etc. O conteúdo da mensagem é o definido no *payload* da mensagem. Na Figura 34 podemos observar esta configuração:

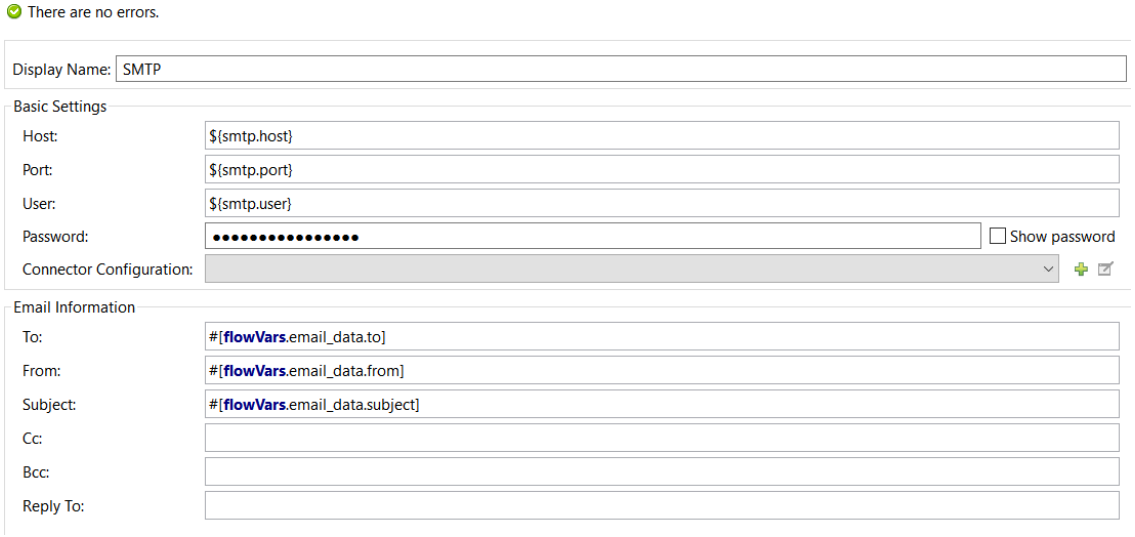


Figura 34 - Configuração SMTP, exemplo

- ***Logger*:** utilizado para escrever mensagens para a consola ou para um ficheiro de logs;
- ***Flow Reference*:** usado para invocar outros Flows;

- **AMQP:** Interface de integração com o RabbitMQ. Permite a definição de produtores e consumidores de eventos. Na Figura 35 podemos observar um exemplo de configuração:

Figura 35 - Configuração AMQP, exemplo

Na figura anterior podemos observar alguns atributos importantes como o nome do Exchange, da queue, o identificador do consumidor e o tipo de queue. É também possível definir algumas características do consumidor como se a queue e o Exchange deve ser apagada ou não, assim que o consumidor é desligado.

- ***Apache Kafka*:** define uma interface e integração com o Kafka mas apresenta-se com uma configuração mais simples que o AMQP, como podemos observar na Figura 36:

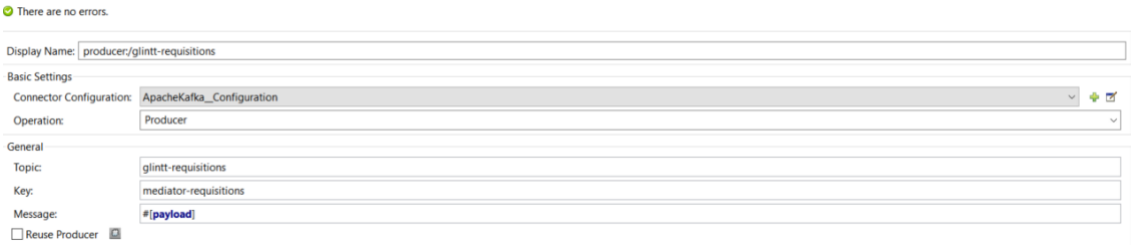


Figura 36 - Configuração Kafka, exemplo

Neste caso apenas é referido o tipo de operação, se é consumidor ou produtor e qual o tópico para o qual vai produzir ou ler consoante o tipo de operação. Também se define uma chave meramente indicativa e qual a mensagem a enviar para o tópico.

- ***Message Enricher*:** utilizado quando se pretende fazer operações que podem alterar o fluxo normal de uma mensagem no ESB, mas não se pretenda que essas alterações tenham efeito. Tudo o que é processado dentro de um *Enricher* é perdido, a não ser que seja configurado para que tal não aconteça. Fornece dois atributos, a fonte e o destino. A fonte é de onde será retirado os dados provenientes do conetor e o destino é onde dados presentes na fonte serão guardados no fluxo normal;

- ***Async*:** todos os fluxos referenciados dentro deste conetor, correm de forma assíncrona sem qualquer ordem pré-definida. Muito útil quando se pretende processar dados que não têm dependência entre si.

4.4.3 Segurança

Todos os serviços Mule utilizam *OAuth* 2.0 como mecanismo de autorização. Foi desenvolvido um conetor capaz de produzir e validar *tokens* JWT para controlar os acessos aos serviços. Para além deste conetor existe um servidor de *OAuth* que pode ser utilizado para gerar, validar e editar estes *tokens*. Um *token* JWT depois de assinado com uma chave secreta é validado através de um id de cliente e um segredo. Para além destas duas variáveis existem alguns serviços que precisam de um utilizador e de uma password que também são enviados aquando do pedido de *token*.

4.5 Diagrama de Implantação

Nesta secção é demonstrado o diagrama de implantação que representa como o sistema foi implantado, conforme se pode observar na Figura 37.

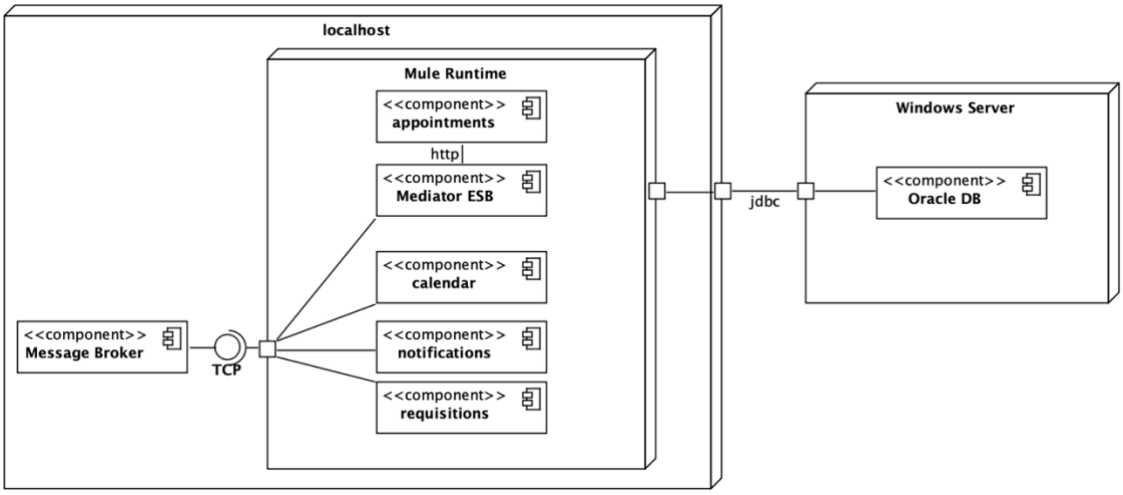


Figura 37 - Diagrama de Implantação, implementado

Esta implantação não utiliza uma das grandes funcionalidades dos Sistemas de Filas de Mensagens, a replicação dos consumidores e do *broker* por diferentes nós, tirando partido de paralelismo e tolerância a falhas. Tal não foi utilizado devido à falta de recursos físicos para esta implementação, sendo por isso apenas apresentada esta solução implantada localmente. Como se pode observar na figura anterior, os serviços Mule estarão a correr na mesma máquina que o sistema de filas de mensagens, sendo que estes serviços comunicarão com o *broker* via TCP e com uma base de dados Oracle que se encontra num servidor Windows externo via JDBC.

4.6 Aplicações Legadas

Em grande parte das empresas é muito comum ficarem agarradas a certos sistemas que pelas mais variadas razões não se conseguem adaptar ás mudanças tecnológicas que vão surgindo ao longo do tempo. A Glintt-HS não é exceção e também possui algumas aplicações legadas. Para contrariar este problema foi necessário averiguar qual o melhor procedimento, se possível, para adaptar uma arquitetura orientada a eventos a estas aplicações. Para além da arquitetura apresentada na secção 2.2 a Glintt-HS possui algumas aplicações desenvolvidas em Oracle *Forms.* Tendo, por isso, muita lógica definida na base de dados. Contudo, é possível comunicar através de eventos na base dados Oracle, através de um mecanismo chamado de *Database Change Notification*[6]. Um exemplo simples que foi implementado para testar este mecanismo pode ser observado na Figura 38:

```
PROCEDURE        calendar_legacy
IS
    v_regds           SYS.chnf$_reg_info;
    v_regid           NUMBER;
    v_calendar_id     NUMBER;
BEGIN
    v_regds :=
        sys.chnf$_reg_info ('glintt.calendar_legacy_callback',
                            DBMS_CHANGE_NOTIFICATION.qos_rowids,
                            0,
                            0,
                            0);
    v_regid := DBMS_CHANGE_NOTIFICATION.new_reg_start (v_regds);

    SELECT id
      INTO v_calendar_id
      FROM xxtese.calendar                    -- register appointments object
     WHERE ROWNUM < 2;     -- return a single row to avoid multiple fetch error

    DBMS_CHANGE_NOTIFICATION.reg_end;
END;
```

Figura 38 - Aplicação Legado, subscrição de evento

O procedimento observado na figura anterior é um exemplo de como uma aplicação legada pode indicar que quando determinado objeto é modificado deve enviar uma notificação. O registo do objeto que pretende ser monitorizado é feito através do comando SQL *select*. O *package* DBMS_CHANGE_NOTIFICATION, automaticamente regista o objeto, neste caso a tabela que está a ser consultada no comando *select.* O objeto CHNF$_REG_INFO, é o objeto que permite que seja indicado qual o procedimento responsável por realizar a notificação, neste

[6]https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_dcn.htm

caso é o "*glintt.calendar_legacy_callback*". Esta notificação é, normalmente, um procedimento em base de dados que pode efetuar uma chamada HTTP para enviar uma notificação, ou apenas inserir um novo registo numa outra tabela, como podemos observar na Figura 39:

```
IF (event_type = DBMS_CHANGE_NOTIFICATION.event_objchange)
THEN
    FOR i IN 1 .. numtables
    LOOP
        tbname := ntfnds.table_desc_array (i).table_name;
        operation_type := ntfnds.table_desc_array (i).opflags;

        INSERT INTO nftablechanges
             VALUES (regid, tbname, operation_type);

        /* Send the table name and operation_type to  client side listener
           using UTL_HTTP */
        /* If interested in the rowids, obtain them as follows */
        IF (BITAND (operation_type, DBMS_CHANGE_NOTIFICATION.all_rows) =
                0)
        THEN
            numrows := ntfnds.table_desc_array (i).numrows;
        ELSE
            numrows := 0;                    /* ROWID INFO NOT AVAILABLE */
        END IF;

        /* The body of the loop is not executed when numrows is ZERO */
        FOR j IN 1 .. numrows
        LOOP
            row_id :=
                ntfnds.table_desc_array (i).row_desc_array (j).row_id;

            INSERT INTO nfrowchanges
                 VALUES (regid, tbname, row_id);
        /* optionally Send out row_ids to client side listener using
           UTL_HTTP */
        END LOOP;
    END LOOP;
END IF;
```

Figura 39 - Excerto de código de notificação

O código apresentado em cima faz parte do procedimento "*glintt.calendar_legacy_callback*". Neste caso, quando recebe um objeto CHNF$_DESC que contém toda a informação relativa ao objeto que está a ser alterado e regista uma nova linha numa tabela a indicar o ocorrido. Caso o pretendido fosse efetuar uma chamada HTTP também tal seria possível utilizando o pacote UTL_HTTP fornecido pela Oracle.

5 Experiências e Avaliação

Neste capítulo são descritos os processos de experimentação e avaliação usados para avaliar a solução desenvolvida. Aqui será definida a abordagem a seguir para avaliar a solução final e com base nessa abordagem será possível avaliar a qualidade de todo o trabalho desenvolvido.

5.1 Abordagem

Para realizar o processo de avaliação foram definidas algumas grandezas que irão ajudar a avaliar a solução aqui documentada, sendo elas:

- **Satisfação dos colaboradores:** utilizada para averiguar se a solução está a ir de encontro aos objetivos definidos e se os colaboradores se sentem bem com a nova arquitetura praticada;

- **Desempenho:** utilizada para medir o desempenho da nova arquitetura em comparação com a praticada anteriormente. É esperada uma diminuição considerável do tempo de resposta desta nova solução, mas por outro lado é esperado um maior consumo de recursos;

- **Escalabilidade**: utilizada para avaliar a capacidade de o sistema crescer e gerir um aumento da carga em todo o sistema.

Além das referidas grandezas, devem ser testadas hipóteses para suportar a implementação da solução, identificando-se as seguintes:

- **Aumento da satisfação dos colaboradores:** é expectável que a satisfação dos colaboradores aumente consideravelmente, dado que é dada a oportunidade de aprender novas metodologias de desenvolvimento de software e também é expectável observar o tempo gasto pelos colaboradores em desenvolvimento e manutenção ser reduzido;

- **Melhoria do desempenho da nova arquitetura:** prevê-se um aumento considerável do desempenho das soluções fornecidas pela Glintt-HS;

- **Aumento da escalabilidade da nova arquitetura:** é previsto um aumento da capacidade de escalar o sistema, tornando este processo mais simples e mais independente. O sistema também deve suportar uma quantidade de carga superior.

Para avaliar as grandezas e testar as hipóteses referidas anteriormente é necessária a definição de metodologias de avaliação, salientando-se as seguintes:

- **Inquéritos:** definição de inquéritos de resposta por nível (0 a 5), para avaliar se os objetivos definidos estão a ser cumpridos, quer quanto ao nível de satisfação dos colaboradores, como a opinião dos mesmos quanto ao desempenho de tempo de desenvolvimento;

- **Monitorização do desempenho da nova arquitetura:** definição de rotinas ou planos de monitorização para averiguar se o desempenho da nova arquitetura vai de encontra ao esperado. Existem várias ferramentas que permitem avaliar o desempenho de sistemas;

- **Monitorização da escalabilidade da nova arquitetura:** tal como na monitorização do desempenho, devem ser definidas rotinas para monitorizar como o sistema se comporta quando é escalado, observando que este passou a suportar mais carga;

Por fim, é necessário realizar testes para averiguar o cumprimento das hipóteses definidas. Existem vários testes estatísticos que podem ser utilizados para averiguar o cumprimento das referidas hipóteses. Estes testes podem ser divididos em dois tipos: os testes paramétricos e os testes não paramétricos.

Para avaliar esta solução vão ser utilizados estes dois tipos. Os testes paramétricos são mais rigorosos e exigem mais pressupostos do que os testes não paramétricos (Reis & Júnior, 2007). Aqui será sempre utilizado um t-Test (paramétrico) quando os seus pressupostos são atingidos, caso contrário será aplicado um teste de Wilcoxon (não paramétrico) (Martinez & Ferreira, 2010). Ambos os testes fornecem a possibilidade de avaliar apenas uma determinada amostra, ou avaliar um conjunto de amostrar. Estes tipos de testes são muito usados para avaliar se um novo processo é melhor ou não que o processo atual, que é exatamente o que se pretende avaliar nesta dissertação.

Como referido, os testes paramétricos requerem alguns pressupostos, tais como:

- As amostras devem ser quantitativas;

- Os dois grupos de amostras são independentes entre si;

- Cada grupo de amostras deve ser uma distribuição normal. Para avaliar se a distribuição é normal, caso a amostra seja inferior a 30 observações, deve ser utilizado o teste de Shapiro-Wilk para averiguar se podemos assumir uma distribuição normal para efetuar o t-Test;

- Cada grupo de amostras não deve conter *outliers.* Para avaliar a existência de *outliers* será utilizado um *box plot*, que permitirá observar se estes *outliers* existem. Caso existam terá de ser utilizado outro teste, normalmente o Wilcoxon s*ignal rank test;*

O t-Test com amostras emparelhadas indica que devem ser definidas duas hipóteses, sendo a primeira a hipótese nula e a hipótese alternativa, sendo que a última pode tomar três opções:

- A média da população das diferenças não é igual à media hipotética das diferenças;
- A média da população das diferenças é maior que a média da hipótese das diferenças;
- A média da população das diferenças é menor que a média da hipótese das diferenças.

É necessária a definição de um grau de confiança para a aplicação deste tipo de testes sendo comum a definição de um intervalo de confiança de 95%, que reflete um nível de significância de 0.05 (α).

Para interpretar os resultados, deve-se então indicar que a hipótese nula é a hipótese que é assumida como verdade, enquanto que a alternativa é assumida como verdadeira quando a hipótese nula não tem evidência estatística que a defenda. Para saber se a hipótese nula é rejeitada ou não, observa-se um valor retornado pelo teste aplicado, o *p-value*, que é uma medida de credibilidade, e caso este valor seja inferior ao nível de significância, calculado através da expressão, 1 – (grau de confiança), podemos concluir que a hipótese é rejeitada, caso contrário podemos concluir que a hipótese não é rejeitada.

Para fazer uma análise estatística sobre os resultados obtidos e de modo a retirar as conclusões sobre este questionário foi utilizada uma ferramenta para auxiliar na análise dos resultados, o RStudio[9].

Como referido anteriormente, para avaliar os *outliers* das amostras será utilizado um *box*. Para criar um *box plot* através do RStudio é utilizada a função *boxplot*. Esta função recebe um parâmetro que é a diferença das duas amostras que se pretendem avaliar e uma descrição do *box plot*. Na Figura 40 podemos observar um exemplo do resultado desta função.

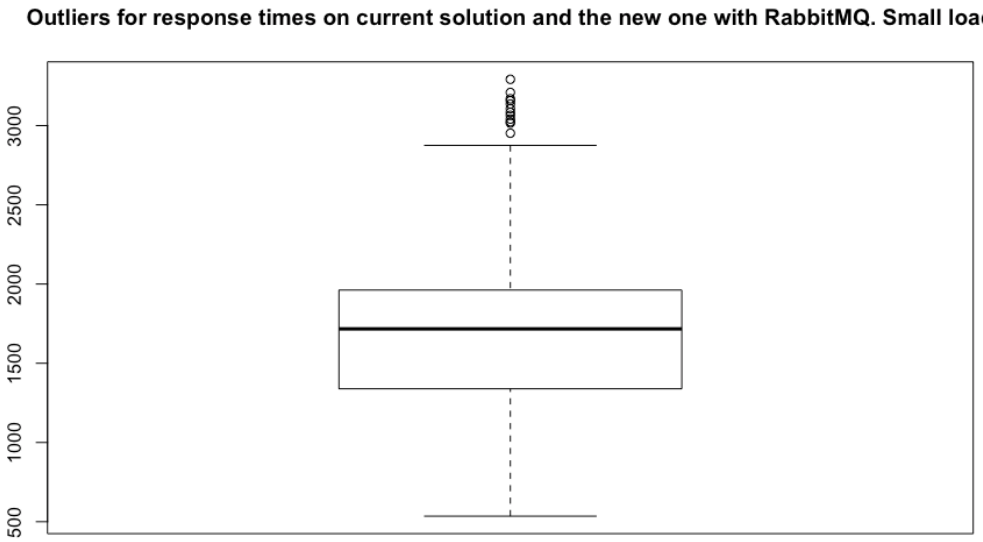


Figura 40 - Boxplot, exemplo de análise de outliers

[9] https://www.rstudio.com

Ao analisar a figura anterior podemos observar que existem *outliers*, que são aqueles pontos no topo da imagem. Posto isto, reunimos agora todas as condições para avaliar todo o trabalho aqui desenvolvido. Os resultados e as análises dos mesmos serão abordados nas secções seguintes.

5.2 Resultados e Análise Estatística dos Inquéritos

Para avaliar a satisfação dos colaboradores da Glintt-HS foi criado um inquérito para ser respondido por uma equipa de desenvolvimento. Este inquérito pode ser observado no Anexo 2.

Foram criadas 5 perguntas:

1. "O que pensas sobre a utilização de uma arquitetura orientada a eventos, integrada com o Mule ESB, em vez da arquitetura atual da Glintt?"

2. "Quando é necessário alterar ou adicionar uma nova funcionalidade a um serviço já existente é necessário que este seja alterado. Atualmente, esta alteração implica que seja feita uma publicação do Web Service onde o serviço alterado está hospedado, causando impacto em todos os serviços hospedados no mesmo Web Service. Além disso, o acréscimo de novas funcionalidades faz com que os serviços contenham demasiada lógica e demasiada responsabilidade. Este comportamento está correto?"

3. "Concordas que a arquitetura orientada a eventos aumenta o desempenho de todo o sistema?"

4. "Concordas que a arquitetura orientada a eventos torna o sistema mais escalável e flexível?"

5. "Concordas que o custo de desenvolvimento de novas funcionalidades e de manutenção de um sistema orientado a eventos é menor do que o custo do sistema atual?"

A avaliação das perguntas anteriores é numerada de 1 a 5. Sendo que o 1 indica que o inquirido discorda totalmente com o que foi afirmado ou questionado e o 5 indica que o inquirido concorda totalmente.

Os resultados podem ser observados na Tabela 4, onde é possível observar que 10 pessoas responderam ao questionário.

Tabela 4 - Resultados dos Inquéritos

Questões	**Respostas**				
	1	2	3	4	5
1	0	0	1	3	6
2	6	2	0	2	0
3	0	0	1	3	6
4	0	0	0	4	6
5	0	0	1	5	4

Ao observar a tabela anterior podemos observar que todos os inqueridos que responderam ao questionário, responderam à totalidade das perguntas. Como apenas 10 pessoas responderam ao questionário, não podemos assumir uma distribuição normal, pelo que será efetuado um teste de Shapiro-Wilk para avaliar se a distribuição é normal. Aqui cada resposta será avaliada individualmente nas próximas secções, com a utilização do RStudio.

5.2.1 Pergunta 1

A primeira pergunta pretende avaliar se os inquiridos pensam que uma solução orientada a eventos integrada com um ESB é melhor que a solução atual da Glintt-HS. Na Figura 41 podemos observar o teste de Shapiro-Wilk.

```
> q1<-c(3,4,4,4,5,5,5,5,5,5)
> shapiro.test(q1)

        Shapiro-Wilk normality test

data:  q1
W = 0.73087, p-value = 0.002088
```

Figura 41 - Shapiro-Wilk teste, pergunta 1

Podemos concluir ao observar o p-value, que este valor é inferior ao nível de significância, pelo que é considerada que a amostra não segue uma distribuição normal. Como não segue uma distribuição normal será então aplicado o teste de Wilcoxon.

Para a primeira pergunta é definido que a média mínima das respostas deve ser 4.5, originando então a seguinte hipótese alternativa: média > 4.5. Na Figura 42 podemos observar o resultado do teste de Wilcoxon para as referidas hipóteses. (RabbitMQ - Messaging that just works, 2018)

```
> wilcox.test(q1, mu = 4.5, alternative = "less")

        Wilcoxon signed rank test with continuity correction

data:  q1
V = 30, p-value = 0.6304
alternative hypothesis: true location is less than 4.5
```

Figura 42 - Wilcoxon teste, pergunta 1

Como o p-value é superior a 0.05, não se rejeita a hipótese nula, pelo que podemos concluir com 95% de confiança que para os colaboradores da Glintt-HS, uma arquitetura orientada a eventos integrada com o Mule ESB é preferível à arquitetura atual.

5.2.2 Pergunta 2

A segunda pergunta pretende averiguar se os inquiridos compreendem o problema que a solução atual apresenta. Na Figura 43 podemos observar o teste de Shapiro-Wilk.

```
> q2<-c(1,1,1,1,1,1,2,2,4,4)
> shapiro.test(q2)

        Shapiro-Wilk normality test

data:  q2
W = 0.68185, p-value = 0.0005313
```

Figura 43 - Shapiro-Wilk teste, pergunta 2

Podemos concluir ao observar o p-value, que este valor é inferior ao nível de significância, pelo que é considerada que a amostra não segue uma distribuição normal. Como não segue uma distribuição normal será então aplicado o teste de Wilcoxon.

Para a segunda pergunta é definido que a média máxima das respostas deve ser inferior a 1.5, originando então a seguinte hipótese alternativa: média < 1.5. Na Figura 44 podemos observar o resultado do teste de Wilcoxon para as referidas hipóteses.

```
> wilcox.test(q2, mu = 1.5, alternative = "greater")

        Wilcoxon signed rank test with continuity correction

data:  q2
V = 28, p-value = 0.5
alternative hypothesis: true location is greater than 1.5
```

Figura 44 - Teste de Wilcoxon, pergunta 2

Como o p-value é igual a 0.05, não se rejeita a hipótese nula, pelo que podemos concluir com 95% de confiança que os colaboradores da Glintt-HS compreendem o problema da arquitetura atual.

5.2.3 Pergunta 3

A terceira pergunta pretende averiguar se os inquiridos pensam que solução implementada irá aumentar o desempenho de todo o sistema. Na Figura 45 podemos observar o teste de Shapiro-Wilk.

```
> q3<-c(3,4,4,5,5,5,5,5,5,5)
> shapiro.test(q3)

        Shapiro-Wilk normality test

data:  q3
W = 0.64969, p-value = 0.0002176
```

Figura 45 - Teste de Shapiro-Wilk, pergunta 3

Podemos concluir ao observar o p-value, que este valor é inferior ao nível de significância, pelo que é considerada que a amostra não segue uma distribuição normal. Como não segue uma distribuição normal será então aplicado o teste de Wilcoxon.

Para a terceira pergunta é definido que a média mínima das respostas deve ser 4.5, originando então a seguinte hipótese alternativa: média > 4.5. Na Figura 46 podemos observar o resultado do teste de Wilcoxon para as referidas hipóteses.

```
> wilcox.test(q3, mu = 4.5, alternative = "less")

        Wilcoxon signed rank test with continuity correction

data:  q3
V = 35, p-value = 0.8126
alternative hypothesis: true location is less than 4.5
```

Figura 46 - Teste de Wilcoxon, pergunta 3

Como o p-value é superior a 0.05, não se rejeita a hipótese nula, pelo que podemos concluir com 95% de confiança que para os colaboradores da Glintt-HS, a solução implementada irá aumentar o desempenho do sistema.

5.2.4 Pergunta 4

A quarta pergunta pretende averiguar se os inquiridos pensar que a solução desenvolvida aumenta a escalabilidade e flexibilidade do sistema. Na Figura 47 podemos observar o teste de Shapiro-Wilk.

```
> q4<-c(4,4,4,4,5,5,5,5,5,5)
> shapiro.test(q4)

        Shapiro-Wilk normality test

data:  q4
W = 0.64049, p-value = 0.0001687
```

Figura 47 - Teste de Shapiro-Wilk, pergunta 4

Podemos concluir ao observar o p-value, que este valor é inferior ao nível de significância, pelo que é considerada que a amostra não segue uma distribuição normal. Como não segue uma distribuição normal será então aplicado o teste de Wilcoxon.

Para a quarta pergunta é definido que a média mínima das respostas deve ser 4.5, originando então a seguinte hipótese alternativa: média > 4.5. Na Figura 48 podemos observar o resultado do teste de Wilcoxon para as referidas hipóteses.

```
> wilcox.test(q4, mu = 4.5, alternative = "less")

        Wilcoxon signed rank test with continuity correction

data:  q4
V = 33, p-value = 0.7549
alternative hypothesis: true location is less than 4.5
```

Figura 48 - Teste de Wilcoxon, pergunta 4

Como o p-value é superior a 0.05, não se rejeita a hipótese nula, pelo que podemos concluir com 95% de confiança que para os colaboradores da Glintt-HS, a solução implementada irá aumentar a escalabilidade e flexibilidade do sistema.

5.2.5 Pergunta 5

A quinta pergunta pretende averiguar se os inquiridos pensam que a nova arquitetura irá reduzir os custos de desenvolvimento e de manutenção em relação à arquitetura atual. Na Figura 49 podemos observar o teste de Shapiro-Wilk.

```
> q5<-c(3,4,4,4,4,4,5,5,5,5)
> shapiro.test(q5)

        Shapiro-Wilk normality test

data:  q5
W = 0.80218, p-value = 0.01541
```

Figura 49 - Teste de Shapiro-Wilk, pergunta 5

Podemos concluir ao observar o p-value, que este valor é inferior ao nível de significância, pelo que é considerada que a amostra não segue uma distribuição normal. Como não segue uma distribuição normal será então aplicado o teste de Wilcoxon.

Para a quinta pergunta é definido que a média mínima das respostas deve ser 4.5, originando então a seguinte hipótese alternativa: média > 4.5. Na Figura 50 podemos observar o resultado do teste de Wilcoxon para as referidas hipóteses.

```
> wilcox.test(q5, mu = 4.5, alternative = "less")

        Wilcoxon signed rank test with continuity correction

data:  q5
V = 20, p-value = 0.2187
alternative hypothesis: true location is less than 4.5
```

Figura 50 - Teste de Wilcoxon, pergunta 5

Como o p-value é inferior a 0.05, rejeita-se a hipótese nula, pelo que não podemos afirmar que os colaboradores da Glintt-HS acreditam que a solução implementada irá reduzir os custos de desenvolvimento e manutenção.

Este resultado é algo esperado, pois o custo inicial de desenvolvimento será bastante maior. Os colaborares estão habituados a desenvolver de uma determinada maneira, pelo que existirá um processo de adaptação a esta nova maneira de desenvolver. Quanto ao tema de

manutenção, esta ficará mais fácil em termos de correção de erros, mas também será preciso gastar mais tempo num processo inicial de manutenção para conseguir controlar e monitorizar todo o sistema corretamente, precavendo qualquer tipo de erros que possa acontecer.

5.2.6 Sumário

Ao observar os resultados da análise ao inquérito realizado a uma equipa de colaboradores da Glintt-HS, podemos concluir que de um modo geral todos concordam que a solução atual não está de acordo com o pretendido e que as soluções aqui desenvolvidas conseguem dar resposta aos problemas apresentados pela solução atual.

A pergunta 5, foi a única que não teve os resultados esperados, o que é facilmente justificável pois as soluções desenvolvidas requerem uma curva de aprendizagem inicial e também é necessário investir tempo e preparar a base da solução para que esta funcione corretamente, seja facilmente monitorizada e alterada.

De um modo geral, podemos concluir que os colaboradores da Glintt-HS estão satisfeitos com o trabalho aqui desenvolvido, acreditando que estas soluções são uma mais valia para o processo de desenvolvimento da Glintt-HS.

5.3 Resultados de Desempenho

É nesta secção onde se começa a perceber os resultados de todo o trabalho desenvolvido. Para testar a arquitetura e compreender se este satisfaz os objetivos que se pretende atingir, foram definidas as métricas para avaliar o seu desempenho:

- Consumo médio de CPU e Memória;
- Número médio de mensagens que são processadas por segundo (publicação e consumo);
- Tempo médio de resposta ao utilizador.

Além destas métricas, foram definidas algumas variantes para estes testes:

- Número de filas em utilização;
- Quantidade de pedidos efetuados num espaço pré-definido de 10 segundos. Para a arquitetura atual foram realizados testes com, 50, 100 e 1000 pedidos, enquanto que para os sistemas com filas de mensagens foram realizados testes com 50, 100, 1000, 5000 e 10000 pedidos.

Os testes aqui apresentados foram aplicados às duas soluções implementadas com os sistemas de filas de mensagens e à arquitetura demonstrada na secção 2.2, sendo que para esta última

não foi avaliado o número de mensagens a ser processadas, nem foi variado o número de filas em utilização, pois esta não possui um sistema de filas de mensagens.

De modo a perceber o impacto que a quantidade de filas de mensagens implica numa arquitetura orientada a eventos, procedeu-se com algumas alterações na arquitetura apenas para averiguar este impacto. Estas alterações apenas foram feitas para conseguir testar da melhor maneira todo o sistema. Na Figura 51 podemos observar as diferenças.

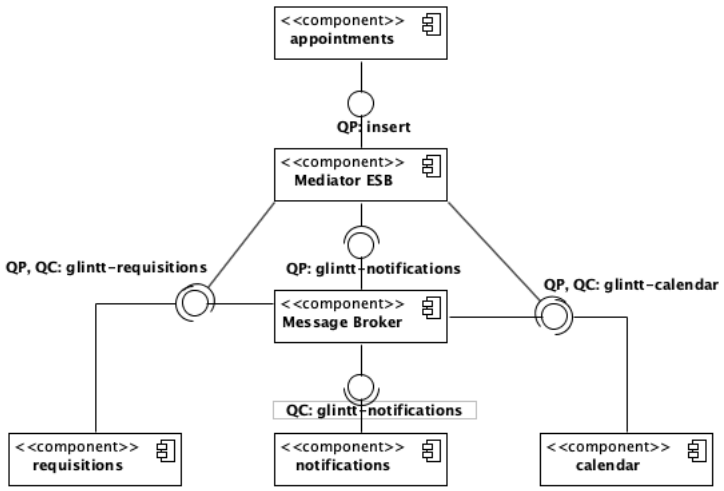


Figura 51 - Diagrama de Componentes, testes

As únicas diferenças que se podem observar na figura anterior em comparação com o diagrama da Figura 17 é a quantidade de filas de mensagens que existem. Para estes testes foram criadas 4 filas de mensagens: *glintt-requisitions, glintt-calendar, glintt-notifications* e *insert.* O modo de funcionamento aqui também é simples e tirou-se partido do ESB para publicar para estas filas. Nestes testes o ESB passou a funcionar como consumidor e produtor de eventos, recebendo um evento de inserção de marcações, mas agora em vez de publicar para uma única fila de mensagens e os consumidores interessados processarem todos o mesmo evento, agora o ESB publica para 3 filas distintas, sendo que os consumidores em vez de estarem subscritos a uma fila, cada consumidor está subscrito à sua respetiva fila. Deste modo é possível observar o impacto que a quantidade de filas tem, sendo que também foi possível observar como foi simples de efetuar esta alteração através do ESB.

Para avaliar as referidas métricas foram efetuados testes de carga em todo o sistema. Para isso foram avaliadas algumas ferramentas:

- ***Tsung*:** pouco suporte, última atualização em Agosto de 2017.
- ***HttpPerf*:** bastante simples. Resultados obtidos através de linha de comandos;
- ***SmartMeter*:** apenas disponibiliza versão trial;

- ***Apache JMeter*:** bastante utilizado e desenvolvido pela Apache. Fornece uma interface gráfica para desenhar os testes e gera gráficos e tabelas com os resultados obtidos;

O *JMeter* foi a ferramenta definida para efetuar estes testes de carga e também avaliar algumas das métricas definidas acima.

Um desafio levantado por estes testes é como avaliar o sistema e avaliar as métricas referidas. Para isso também foram estudadas algumas ferramentas de monitorização:

- ***Datadog*:** uma das ferramentas mais utilizadas para monitorizar os dois sistemas em questão. Disponibiliza uma versão trial de 30 dias. Possui agentes que recolhem dados do RabbitMQ e do Kafka e enviam para o *Datadog*. Uma das dificuldades encontradas foi a instalação do agente para monitorizar o *RabbitMQ*;

- **Collectd:** apenas funciona em sistemas UNIX. Aconselha a utilização do SSC Serv para monitorizar sistemas em Windows, sendo que este não foi sequer equacionado;

- ***Kafka Manager*:** apenas funciona em UNIX. Dificuldade em adicionar o nó do Kafka para ser monitorizado. Apenas permite monitorizar o *Kafka*;

- ***JConsole*:** ferramenta fornecida pelo Java, que permite retirar informações de monitorização através de recursos JMX (*Java Management Extensions*). Apenas permite monitorizar o Kafka;

- ***RabbitMQ* API *Management*:** ferramenta disponibilizada pelo RabbitMQ. Fornece uma interface gráfica que permite retirar métricas relacionadas com as filas de mensagens do RabbitMQ.

Devido à dificuldade de encontrar uma ferramenta que permitisse monitorizar tanto o RabbitMQ como o Kafka, admitindo que estas ferramentas não possuem grandes desvios nos métodos de cálculo de mensagens processadas por segundo, optou-se por utilizar a ferramenta disponibilizada pelo *RabbitMQ* para monitorizar as suas filas de mensagens, enquanto que para monitorizar o *Kafka*, como não fornecia nenhuma ferramenta que permitisse esta monitorização, optou-se pela utilização do *JConsole*, dada a sua facilidade de configuração e de interpretação dos dados levantados.

O *JMeter* permite a definição de planos de teste através da sua interface gráfica, onde foi especificado um período de tempo 10 segundos para efetuar pedidos ao sistema desenvolvido, sendo este o período definido para todos os testes aqui apresentados. Além deste período de tempo é definido o número de processos que vão ser corridos, ou seja, o número de pedidos que vão ser efetuados para o período definido. Através da interface gráfica também foi definido o que monitorizar e algumas variáveis necessárias para efetuar as chamadas aos serviços em alvo de teste, como é o caso do *token OAuth*, referido na secção 4.4.3. Na Figura 52 podemos observar um exemplo da definição de uma chamada a um serviço:

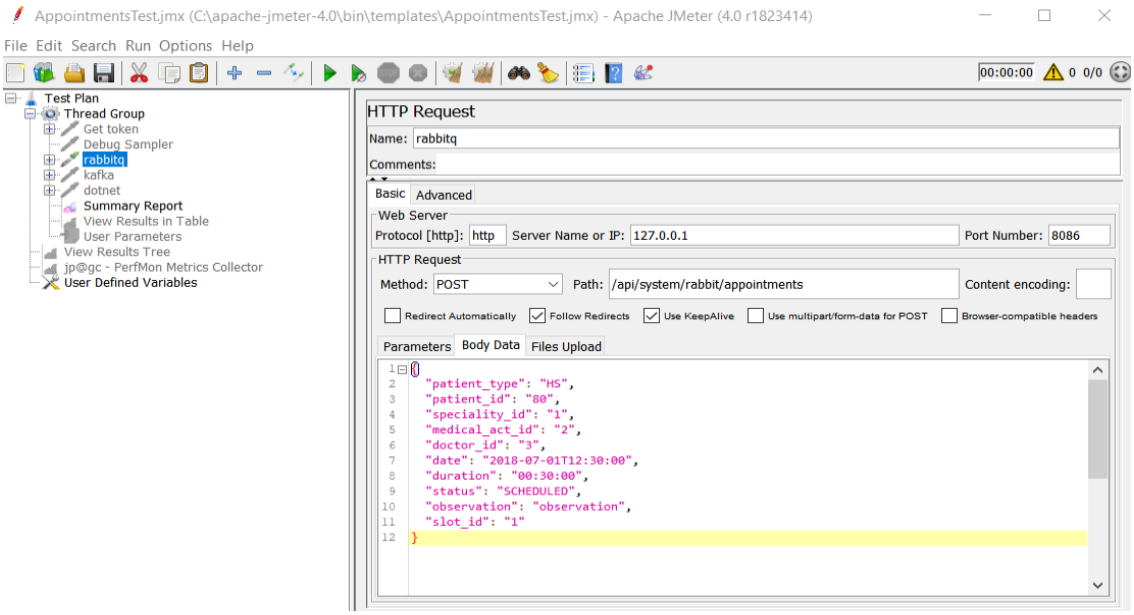


Figura 52 - JMeter, exemplo

Os resultados dos testes efetuados poderiam ser observados na interface gráfica fornecida pelo *JMeter*, mas é aconselhável pelos próprios a que apenas seja utilizada a interface gráfica para a elaboração do teste (Apache, 2018), sendo o teste executado através da linha de comandos. Para execução destes testes foi desenvolvido um script escrito em *bash* para tornar a execução dos testes mais simples e rápida. O script de execução pode ser observado na Figura 53:

```
set folder=C:\apache-jmeter-4.0
set n_threads=5000
set period=10
set token="Bearer
ey3hbGciOiJIUzI1NiJ9.eyJzdWIiOiJHUEFUSUVOVFBPUlRBTCIsImF1ZCI6IkdQTEFURk9STSIsInVuaXF1ZV9uYW1lIjoiSk9TRUZFUk5BTkRFU0BHTElOVFQuQ09NIiwibmJmIjoxNTM3Mzk3NzgyLCJpc3MiOiJHUExB
VEZPUk0iLCJuYW1laWQiOiIzX184MCIsImNsaWVudEluZm8iOiJ7XCJGYWNpbGl0eUlEXCI6XCJcIixcIlRlbmFudElEXCI6XCI2NEQ5RDAxQS0yQjU0LUQxMTQtRTA1My0xMzI3RkEwQUEzRTFcIixcIkFwcGxpY2F0aW9uV
mVyc2lvblwiOlwiXCIsXCJVc2VybmFtZVwiOlwiR1BBVElFTlRQT1JUQUxcIixcIkV4dGVybmFsUHJvdmlkZXJVc2VySURcIjpcIjNfXzgwXCIsXCJVc2VySWRcIjpcIjNfXzgwXCIsXCJFeHRlcm5hbFByb3ZpZGVyVXNlcm
5hbWVcIjpcIkpPU0VGRVJOQU5ERVNAR0xJTlRULkNPTVwiLFwiUm9sZXNcIjpbXSxcIkFwcGxpY2F0aW9uSURcIjpcIkdQQVRJRU5UUE9SVEFMXCIsXCJDdWx0dXJlXCI6XCJwdC1QVFwifSIsImV4cCI6MTUzNzQ4NDE4Miw
iY2xpZW50X2lkIjoiR1BBVElFTlQifQ.8cp0fPj2VfDS2iH-nZAMUZ4bhzaXfkhnHDMzuZvFeig"

cd %folder%\bin
rmdir /S /Q %folder%\reports
rmdir /S /Q %folder%\logs
CALL jmeter -n -Jthreads=%n_threads% -Jperiod=%period% -Jtoken=%token% -t .\templates\AppointmentsTest.jmx -l ..\logs\appointments_log.csv -e -o ..\reports
```

Figura 53 - Script JMeter

Como já referido este script, permite a definição do número de pedidos que serão realizados para um determinado espaço de tempo e o *token* que é necessário para efetuar a chamada ao serviço a ser testado. O resultado final pode ser observado em dois ficheiros criados pelo *JMeter*, um ficheiro HTML, com gráficos e tabelas e outro ficheiro CSV. Esses dados vão ser demonstrados nos capítulos seguintes.

Os testes aqui apresentados foram efetuados em apenas uma máquina com a seguintes características:

- **Processador:** Intel Core i7-7600, 2.8 GHz
- **RAM:** 16 GB

- **Sistema Operativo:** Windows 10
- **Disco**: SSD

Para organizar a demonstração destes resultados, dividiu-se este capítulo em duas secções. A primeira onde são apresentados os resultados relacionados com a antiga solução da Glintt-HS, desenvolvida em .NET e sem qualquer tipo de sistema de fila de mensagens, enquanto que a segunda apresenta os resultados relacionados com as implementações com uso do sistema de fila de mensagens. Para cada uma destas secções será apresentado os resultados relacionados com o tempo de resposta ao utilizador, o consumo de CPU e RAM.

5.3.1 Resultados Arquitetura Atual

Para avaliar os referidos problemas do sistema atual da Glintt-HS, este também foi alvo de monitorização e de testes. Aqui serão apresentados os resultados obtidos.

Estes resultados estão agrupados pelas três métricas definidas para serem avaliadas, conforme se pode observar na Tabela 5.

Tabela 5 - Resultados Solução Atual

Nº de Pedidos / 10 s	Resposta (ms)	CPU (%)	RAM (%)
50	1474.14 ms	13.614 %	45.238 %
100	1920.61 ms	17.317 %	44.094 %
1000	41167.79 ms	23.741 %	46.542 %

O serviço que aqui está a ser testado demora em média cerca de 600 ms a responder quando é efetuado apenas um único pedido por segundo. Quando é efetuado um teste de carga ao mesmo os tempos de resposta vão aumentar, derivado da carga exercida.

Como se pode observar na tabela anterior, os gastos de recursos deste sistema são reduzidos. O mais preocupante é o tempo de resposta, que para um serviço de marcação de consulta, algo que se pretende que seja bastante rápido, caso sejam feitos 50 pedidos num espaço de tempo de 10 segundos, o serviço demora, em média, perto de 1 segundo e meio a responder. Para 100 pedidos, as diferenças são poucas, aumento ligeiramente o consumo de CPU em 4 %. O último teste aqui realizado foram cerca de 1000 pedidos. Neste teste os tempos de respostas aumentaram cerca de 21 vezes. Em média os tempos de resposta passaram a ser de 41 segundos, também aumentando o consumo de CPU em 6%.

Foi realizado um outro teste, com 2500 pedidos em 10 segundos, mas este último fez com que o serviço ficasse indisponível. A arquitetura aqui testada está hospedada no IIS (*Internet*

Information Services), tendo os seus serviços divididos em diferentes *Application Pools*, ou seja, embora os serviços utilizados estejam hospedados no mesmo servidor IIS, estão completamente isolados entre si. Ao observar o *Event Viewer* do Windows, observou-se que uma das *Applications Pools* utilizadas vai abaixo, dado a quantidade de pedidos que estão a ser processados, originando um grande número de erros HTTP 503, Serviço Indisponível.

5.3.2 Resultados com Sistema de Fila de Mensagens

Nesta secção serão apresentados os resultados obtidos relacionados com as duas soluções desenvolvidos, uma utilizando *RabbitMQ* e outra utilizando *Apache Kafka*. Devido à falta de recursos não foi possível testar estas duas soluções da forma desejada, tirando partido da escalabilidade fornecida por estes sistemas, sendo, por isso, apenas utilizado um nó de RabbitMQ e outro de Kafka. Ou seja, cada produtor ou consumidor apenas existe numa máquina, não estando replicados em diferentes máquinas com características idênticas como desejado. De seguida podemos observar os resultados obtidos para os tempos de resposta, consumo de CPU e RAM, seguindo-se os resultados das taxas de transferências de mensagens.

5.3.2.1 Tempo de Resposta, CPU e RAM

Os testes apresentados aqui demonstram os resultados obtidos relacionado com as métricas estipuladas. Na Tabela 6 são apresentados os resultados onde todas as filas de mensagens estão a ser utilizadas.

Tabela 6 - Resultados com Sistemas de Fila de Mensagens

Sistema de Filas de Mensagens	Nº de Pedidos / 10 s	Resposta (ms)	CPU (%)	RAM (%)
RabbitMQ	50	45.98 ms	21.386 %	60.822 %
	100	54.81 ms	34.6 %	59.627 %
	1000	166.77 ms	79.411 %	59.912 %
	5000 [10]	35950.66 ms	91.976 %	63.245 %

[10] RabbitMQ para 5000 pedidos conseguiu publicar todos os eventos relacionados com a inserção de uma marcação. Contudo, apenas com 5000 pedidos, os consumidores não processaram as mesmas em nenhum dos testes realizados. Apenas cerca de 10 % das mensagens eram processadas. As filas de mensagem eram desligadas devido à carga.

Sistema de Filas de Mensagens	Nº de Pedidos / 10 s	Resposta (ms)	CPU (%)	RAM (%)
RabbitMQ Durable Queues	5000 [11]	41349.28 ms	89.453 %	65.745 %
Apache Kafka	50	93.48 ms	24.558 %	63.556 %
	100	128.51 ms	35.062 %	63.378 %
	1000	1472.64 ms	79.802 %	63.888 %
	5000	32314.34 ms	92.804 %	65.714 %
	10000	138672.36 ms	95.839 %	66.118 %

Observando a tabela anterior é possível retirar as seguintes conclusões:

- Em termos de consumo de CPU e de RAM não existem diferenças muito significativas; O *Rabbit,* contudo, consome ligeiramente menos CPU e RAM;
- O *Rabbit* também se apresenta como um sistema mais rápido em termos de resposta ao utilizador com menos carga, para os 50, 100 e 1000 pedidos;
- O *Kafka* apresenta tempos de resposta ao utilizador mais rápidos para situações com mais carga, 1000, 5000 e 10000 pedidos;
- O *Rabbit* com 5000 pedidos não conseguiu processar todas as mensagens, sendo que por isso nem foi testado com 10000 mensagens;
- Se o *Rabbit* funcionar como o *Kafka,* escrevendo as mensagens para disco, sendo impossível perder as mensagens, não existem grandes alterações em termos de tempos de resposta, mas os seus consumidores conseguem processar bastantes mais mensagens do que sem escrever as mensagens para o disco;
- O *Kafka* mesmo com 10000 mensagens conseguiu processar todas as mensagens corretamente;

Posto isto, vamos averiguar a influência que a quantidade de filas de mensagens tem em todo o sistema. Para estes testes duas filas de mensagens foram desativadas, ou seja, o sistema agora

[11] Para os consumidores não desligarem as filas de mensagem, foi realizado um teste com uma opção que obriga a que as filas de mensagem não sejam destruídas (*Durable Queues*). O consumo de mensagens aumentou para os 60%, mas o tempo de resposta também aumentou. Esta opção faz com que o RabbitMQ opere como o Kafka, escrevendo as mensagens para o disco em vez de as guardar em memória.

apenas insere uma marcação e regista essa marcação no calendário, removendo assim a lógica de envio de uma notificação e a inserção de uma requisição para a marcação.

De acordo com os dados obtidos anteriormente, como o *Rabbit* com 5000 pedidos apresentava-se com dificuldades em processar todas as mensagens, decidiu-se aplicar este teste para 1000 pedidos, com o intuito de verificar se o *Rabbit* continua com melhor desempenho que o *Kafka* quando existem menos filas de mensagens em utilização. Na Tabela 7 é possível observar os resultados obtidos.

Tabela 7 - Resultados com Sistema de Fila de Mensagens com menos duas filas

Sistema de Filas de Mensagens	**Nº de Pedidos / 10 s**	**Resposta (ms)**	**CPU (%)**	**RAM (%)**
RabbitMQ	1000	164.18 ms	79.142 %	55.437 %
Apache Kafka	1000	1240.36 ms	79.174 %	58.985 %

Ao observar os resultados da tabela anterior, podemos concluir que não existem diferenças muito significativas consoante o número de filas, com tudo, existe uma pequena diminuição no tempo de resposta e também no consumo de RAM.

Relativamente às métricas aqui testadas, para apenas um nó, tanto de *Rabbit* como de *Kafka,* podemos concluir que o *Rabbit* se apresenta como uma solução melhor quando o nível de carga é mais baixo, mas quando se passa para níveis de carga mais elevados o *Kafka* apresenta-se como a melhor solução. A nível de recursos ambas as soluções se apresentam sem grandes diferenças, sendo que o *Rabbit* consome menos recursos, mas sendo esta diferença pouco significativa.

Tanto o *Rabbit* como o *Kafka* fornecem mecanismos de escalabilidade, aumentando o paralelismo e aumentando ainda mais a capacidade de publicar e receber mensagens. Estes mecanismos não foram aqui tidos em conta pois não existiam recursos físicos para realizar estes testes. Estes mecanismos passam pela replicação dos nós, dos consumidores e da utilização de partições no caso do *Kafka.* Embora sejam dispendiosos em termos de recursos, estes mecanismos aumentam a capacidade dos dois sistemas consideravelmente. Normalmente, e também pelos resultados obtidos nos testes aqui realizados, o *Kafka* é mais aconselhado quando existe uma necessidade de processar grandes quantidades de dados, mas em 2014, a Google e a Pivotal conseguiram processar cerca de 1 milhão de mensagens por segundo (Kuch, 2014). De seguida, vamos observar os resultados obtidos relacionados com as taxas de transferências de mensagens.

Os testes aqui realizados estão, portanto, de acordo com o estudado no capítulo 2.

5.3.2.2 Taxas de Transferência de Mensagens

As taxas de transferências de mensagens, indicam-nos a velocidade com que as mensagens estão a ser processadas, sendo definidas pelo número de mensagens que estão a ser processadas por segundo. O *Rabbit,* como já referido, fornece uma interface gráfica para analisar estes dados, fornecendo métricas por fila de mensagens, por *exchange* e pelo nó global do *Rabbit.* Por outro lado, para monitorizar o *Kafka* foi utilizado o *JConsole*, lendo as métricas fornecidas pelo *Kafka* via JMX, onde são observados dados relacionados com os tópicos e também com o nó. Para estes testes, mais uma vez foram considerados os 1000 pedidos num período de 10 segundos. Na Tabela 8 podemos observar as taxas médias de um nó, de um produtor e de um consumidor.

Tabela 8 - Taxas de Transferência de Mensagens

Sistema de Filas de Mensagens	Nº de filas	Taxas do Nó	Taxas do produtor	Taxas do Consumidor
RabbitMQ	4	39 / s	17 / s	9.2 / s
RabbitMQ Durable Queues	4	35 / s	17 / s	8.6 / s
Apache Kafka	4	18.1 / s	9.3 / s	4.14 / s

Na tabela anterior é possível observar os resultados obtidos relacionados com as taxas de transferência de mensagens.

Este teste foi mais uma vez baseado em 1000 pedidos para um espaço de tempo de 10 segundos. Como se pode observar, neste caso em particular, o *Rabbit* apresenta taxas de transferência bastante superiores. Mesmo com as filas de mensagem declaradas como duradoras, ou seja, o *Rabbit* passa a escrever as mensagens para o disco, o sistema não apresentou grandes diferenças diminuindo muito ligeiramente as taxas. Mais uma vez o *Kafka* para 1000 pedidos possui taxas mais baixas.

De seguida são demonstrados alguns gráficos gerados pelo sistema de monitorização do *Rabbit* que facilitam bastante a análise do sistema.

Na Figura 54 é possível observar um gráfico que representa as taxas de mensagens de todo o nó, onde se pode observar um pico de cerca de 90 mensagens processadas por segundo num determinado momento, sendo que quando o processamento estabilizou, o número de mensagens processadas por segundo rondava as 30.

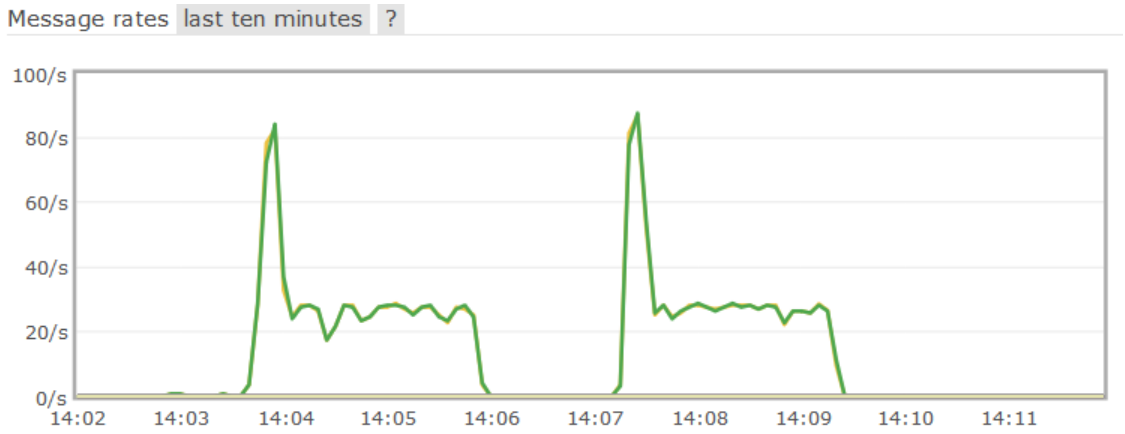


Figura 54 - Monitorização Taxas do Nó, Rabbit

Na Figura 55 podemos observar as taxas dos *exchanges* que fazem o roteamento das mensagens para as respetivas filas. Como já referido foram criados dois *exchanges*, um *appointments.direct* que encaminha as mensagens para as filas especificadas com uma determinada chave, outro *appointments.fanout* que transmite todas as mensagens para todas as filas que estão mapeadas para este *exchange.* Na figura, para além das taxas por *exchange,* é possível observar todos os *exchanges*, bem como o seu tipo e as suas configurações (i.e. duradouro, deve ser apagado automaticamente, etc.).

Name	Type	Features	Message rate in	Message rate out
(AMQP default)	direct	D		
amq.direct	direct	D		
amq.fanout	fanout	D		
amq.headers	headers	D		
amq.match	headers	D		
amq.rabbitmq.trace	topic	D I		
amq.topic	topic	D		
appointments.direct	direct	D	43/s	43/s
appointments.fanout	fanout	D	89/s	89/s

Figura 55 - Monitorização Exchanges, Rabbit

Na Figura 56 é possível observar o estado de todas as filas de mensagens, as suas configurações, as taxas de transferência de mensagens, o estado, etc.

Overview				Messages				Message rates		
Name	Features	**Consumer utilisation**	**State**	**Ready**	**Unacked**	**In Memory**	**Total**	**incoming**	**deliver / get**	**ack**
glintt-calendar	AD	100%	running	0	0	0	0	11/s	11/s	0.00/s
glintt-notifications	AD	100%	running	0	0	0	0	11/s	11/s	0.00/s
glintt-requisitions	AD	100%	running	0	0	0	0	11/s	11/s	0.00/s
insert	AD	100%	running	0	0	0	0	95/s	96/s	0.00/s

Figura 56 - Monitorização Filas de Mensagens, Rabbit

As imagens apresentadas anteriormente demonstram a utilidade de monitorização do sistema, permitindo entender se os testes realizados estão a cumprir o desejado e perceber como o sistema se comporta.

5.3.3 Análise Estatística

Na secção anterior foi possível observar os resultados obtidos quanto ao desempenho dos dois sistemas implementados e do sistema atual. Nesta secção será feita uma análise estatística destes resultados de modo a comprovar as questões levantas com os resultados obtidos.

Efetuaram-se testes estatísticos para os tempos de resposta e consumos de CPU obtidos. O consumo de memória não é aqui analisado pois não apresenta grandes diferenças entre as soluções implementadas, sendo apenas observada um aumento do consumo de memória em comparação com o sistema atual. Este aumento é perfeitamente normal, pois passou a existir um sistema de filas de mensagens implantado na máquina onde foram realizados os testes e os micro serviços implementados não foram escalados nem migrados para máquinas diferentes. O consumo de CPU apenas será avaliado entre as soluções implementadas, pois a solução atual demora muito mais tempo (mais do que os 10 segundos definidos) a efetuar o processamento de todos os pedidos, sendo que para um teste de 1000 pedidos a amostra da solução atual é cerca de 73 observações, enquanto que para os sistemas com filas de mensagens é de apenas 10.

Os dados que serão aqui utilizados são os mesmos que foram apresentados na secção 5.3.2, tendo sido recolhidos através do *JMeter.* Como observado em alguns resultados já apresentados anteriormente, podemos concluir que a nível de desempenho houve um aumento considerável, mas o consumo de recursos também aumentou. Posto isto, vamos então realizar uma análise estatística de modo a comprovar os resultados obtidos.

Para os testes aqui apresentados será sempre considerada uma distribuição normal dado que a amostra é superior a 30 observações.

5.3.3.1 Solução Atual vs Solução com RabbitMQ

Aqui é comparada a solução atual com a solução que utiliza o RabbitMQ quando o sistema se encontra com pouca carga, sendo avaliada a seguinte hipótese alternativa: para 150 pedidos, a média de tempo de resposta da solução atual é maior do que a média de tempo de resposta da

solução com *RabbitMQ*. Ao observar o *box plot,* podemos concluir que as amostras recolhidas contêm *outliers,* sendo por isso aplicado o teste de *Wilcoxon*. Na Figura 57 podemos observar o resultado obtido, onde é avaliada a seguinte hipótese alternativa: média de tempo de resposta da solução atual é maior do que a média de tempo de resposta da solução com *RabbitMQ*.

```
> wilcox.test(dotnet_low$elapsed, rabbit_low$elapsed, alternative = "greater")

        Wilcoxon rank sum test with continuity correction

data:  dotnet_low$elapsed and rabbit_low$elapsed
W = 22500, p-value < 2.2e-16
alternative hypothesis: true location shift is greater than 0
```

Figura 57 - Teste de Wilcoxon, Solução Atual vs RabbitMQ, 150 pedidos

Como o p-value é inferior a 0.05, a hipótese nula é rejeitada, pelo que podemos concluir com 95% de confiança que a solução atual apresenta um tempo médio de resposta superior ao da solução que utiliza *RabbitMQ* para cargas mais pequenas.

De seguida é analisada outra hipótese alternativa: para 1000 pedidos, a média de tempo de resposta da solução atual é maior do que a média de tempo de resposta da solução com *RabbitMQ*. Ao observar o *box plot,* podemos concluir que as amostras recolhidas não possuem *outliers,* sendo por isso aplicado o t-Test. Na Figura 58 podemos observar o resultado obtido.

```
> t.test(dotnet_mid$elapsed, rabbit_mid$elapsed, alternative = "greater")

        Welch Two Sample t-test

data:  dotnet_mid$elapsed and rabbit_mid$elapsed
t = 61.824, df = 999.04, p-value < 2.2e-16
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
 39909.16      Inf
sample estimates:
mean of x mean of y
41167.792   166.767
```

Figura 58 - T-test, Solução Atual vs RabbitMQ, 1000 pedidos

Como o p-value é inferior a 0.05, a hipótese nula é rejeitada, pelo que podemos concluir com 95% de confiança que a solução atual apresenta um tempo médio de resposta superior ao da solução que utiliza *RabbitMQ* para 1000 pedidos. Além disso podemos observar a diferença de médias e que estas estão de acordo com o apresentado na secção 5.3.

Observando os resultados, existem dados estatísticos para concluir que o sistema desenvolvido que utiliza RabbitMQ melhorou consideravelmente o desempenho do sistema em termos de tempos de resposta.

5.3.3.2 Solução Atual vs Solução com Kafka

Aqui é comparada a solução atual com a solução que utiliza o *Kafka* quando o sistema se encontra com pouca carga, sendo avaliada a seguinte hipótese alternativa: para 150 pedidos, a média de tempo de resposta da solução atual é maior do que a média de tempo de resposta da solução com *Kafka*. Ao observar o *box plot,* podemos concluir que as amostras recolhidas contêm *outliers,* sendo por isso aplicado o teste de *Wilcoxon*. Na Figura 59 podemos observar o resultado obtido.

```
> wilcox.test(dotnet_low$elapsed, kafka_low$elapsed, alternative = "greater")

        Wilcoxon rank sum test with continuity correction

data:  dotnet_low$elapsed and kafka_low$elapsed
W = 22499, p-value < 2.2e-16
alternative hypothesis: true location shift is greater than 0
```

Figura 59 - Teste de Wilcoxon, Solução Atual vs Kafka, 150 pedidos

Como o p-value é inferior a 0.05, a hipótese nula é rejeitada, pelo que podemos concluir com 95% de confiança que a solução atual apresenta um tempo médio de resposta superior ao da solução que utiliza *Kafka* para cargas mais pequenas.

De seguida é analisada outra hipótese alternativa: para 1000 pedidos, a média de tempo de resposta da solução atual é maior do que a média de tempo de resposta da solução com *Kafka*. Ao observar o *box plot,* podemos concluir que as amostras recolhidas não possuem *outliers,* sendo por isso aplicado o t-Test. Na Figura 60 podemos observar o resultado obtido.

```
> t.test(dotnet_mid$elapsed, kafka_mid$elapsed, alternative = "greater")

        Welch Two Sample t-test

data:  dotnet_mid$elapsed and kafka_mid$elapsed
t = 59.781, df = 1004, p-value < 2.2e-16
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
 38601.94      Inf
sample estimates:
mean of x mean of y
41167.792  1472.636
```

Figura 60 - T-test, Solução Atual vs Kafka, 1000 pedidos

Como o p-value é inferior a 0.05, a hipótese nula é rejeitada, pelo que podemos concluir com 95% de confiança que a solução atual apresenta um tempo médio de resposta superior ao da solução que utiliza *Kafka* para 1000 pedidos. Além disso podemos observar a diferença de médias e que estas estão de acordo com o apresentado na secção 5.3.

Observando os resultados, existem dados estatísticos para concluir que o sistema desenvolvido que utiliza *Kafka* melhorou consideravelmente o desempenho do sistema em termos de tempos de resposta. Mais uma vez, é possível observar as médias das diferentes amostras e que estas também estão de acordo com o apresentado na secção 5.3.

5.3.3.3 Solução com RabbitMQ vs Solução com Kafka

Aqui é onde serão apresentados os resultados relacionados com os dois sistemas implementados, onde poderá ser comprovado com dados estatísticos em que situação uma solução é melhor que a outra. Aqui também são apresentados os resultados relacionados com o consumo de CPU.

Ao observar os *box plots* para 150, 1000 e 5000 pedidos, podemos observar que em todas as amostras existem outliers, pelo que terá de ser utilizado um teste de Wilcoxon para analisar os sistemas. Na Figura 61 podemos observar o resultado do teste, onde é avaliada a seguinte hipótese alternativa: para 150 pedidos, a média de tempo de resposta da solução com *RabbitMQ* é inferior à média de tempo de resposta da solução com *Kafka*.

```
> wilcox.test(rabbit_low$elapsed, kafka_low$elapsed, alternative = "less")

        Wilcoxon rank sum test with continuity correction

data:  rabbit_low$elapsed and kafka_low$elapsed
W = 1369, p-value < 2.2e-16
alternative hypothesis: true location shift is less than 0
```

Figura 61 - Teste de Wilcoxon, RabbitMQ vs Kafka, 150 pedidos

Como o p-value é inferior a 0.05, a hipótese nula é rejeitada, pelo que podemos concluir com 95% de confiança que a solução com *RabbitMQ* apresenta um tempo médio de resposta inferior ao da solução que utiliza *Kafka* para 150 pedidos.

De seguida, na Figura 62 podemos observar o resultado do, onde é avaliada a seguinte hipótese alternativa: para 1000 pedidos, a média de tempo de resposta da solução com *RabbitMQ* é inferior à média de tempo de resposta da solução com *Kafka*.

```
> wilcox.test(rabbit_mid$elapsed, kafka_mid$elapsed, alternative = "less")

        Wilcoxon rank sum test with continuity correction

data:  rabbit_mid$elapsed and kafka_mid$elapsed
W = 56848, p-value < 2.2e-16
alternative hypothesis: true location shift is less than 0
```

Figura 62 - Teste de Wilcoxon, RabbitMQ vs Kafka, 1000 pedidos

Como o p-value é inferior a 0.05, a hipótese nula é rejeitada, pelo que podemos concluir com 95% de confiança que a solução com *RabbitMQ* apresenta um tempo médio de resposta inferior ao da solução que utiliza *Kafka* para 1000 pedidos.

Para termina análise dos tempos de resposta, na Figura 63 podemos observar o resultado do teste, onde é avaliada a seguinte hipótese alternativa: para 5000 pedidos, a média de tempo de resposta da solução com *RabbitMQ* é superior à média de tempo de resposta da solução com *Kafka*.

```
> wilcox.test(rabbit_high$elapsed, kafka_high$elapsed, alternative = "greater")

        Wilcoxon rank sum test with continuity correction

data:  rabbit_high$elapsed and kafka_high$elapsed
W = 19457000, p-value < 2.2e-16
alternative hypothesis: true location shift is greater than 0
```

Figura 63 - Teste de Wilcoxon, RabbitMQ vs Kafka, 5000 pedidos

Como o p-value é inferior a 0.05, a hipótese nula é rejeitada, pelo que podemos concluir com 95% de confiança que a solução com *RabbitMQ* apresenta um tempo médio de resposta superior ao da solução que utiliza *Kafka* para 5000 pedidos.

Por fim, é apresentado uma análise dos consumos de CPU para 1000 pedidos. Ao observar os resultados da secção 5.3.2, podemos observar que não existem grandes diferenças em termos de consumo de CPU e também de memória entre as duas soluções, contudo iremos aplicar um teste de Wicoxon para provar que mesmo sendo uma diferença muito reduzida, o consumo de CPU do *Kafka*, é maior do que o do *RabbitMQ,* sendo esta a nossa hipótese alternativa, conforme se pode observar na Figura 64.

```
> wilcox.test(kafka_cpu$elapsed, rabbit_cpu$elapsed, alternative = "greater")

        Wilcoxon rank sum test

data:  kafka_cpu$elapsed and rabbit_cpu$elapsed
W = 64, p-value = 0.01999
alternative hypothesis: true location shift is greater than 0
```

Figura 64 - Teste de Wilcoxon, Kafka vs RabbitMQ, consumo de CPU

Como o p-value é inferior a 0.05, a hipótese nula é rejeitada, pelo que poemos concluir com 95% de confiança que a solução que utiliza o *Kafka,* consome mais CPU do que a solução que utiliza o *RabbitMQ.* Aqui também podemos observar que o p-value está mais próximo de 0.05, o que quer dizer que neste caso embora se possa afirmar com 95% de confiança, esta métrica não se apresenta tão clara como as restantes métricas que foram avaliadas anteriormente.

5.3.4 Sumário

Ao observar todos os resultados presentes nas secções anteriores, podemos afirmar com certezas que as duas soluções implementadas melhoraram o sistema a nível de tempo de resposta. Com uma arquitetura orientada a eventos podemos desenvolver serviços que se preocupam apenas com a sua responsabilidade, diminuindo consideravelmente os tempos de resposta. Além disto, é possível concluir que o *RabbitMQ* com menos carga apresenta tempos de resposta bastante mais rápidos, mas quando o sistema fica com demasiada carga, o *Kafka* comporta-se melhor, apresentando tempos de resposta mais rápidos conforme estudado no capítulo 2.

Em termos de consumo de CPU, podemos concluir que o *Kafka,* de um modo geral*,* consome ligeiramente mais do que o *RabbitMQ.*

Os dados estáticos apresentados anteriormente, apresentam quase sempre um p-value de 2.2e-16, com exceção da análise de CPU, o que poderia ser interpretado como um erro. Contudo, este valor é apresentado devido a um arredondamento que o *RStudio* efetua quando o p-value é muito pequeno. Na análise de CPU, como o p-value está mais próximo de 0.05, podemos concluir que é a métrica que não apresenta tanta evidência estatística do que se tentou provar.

5.4 Resultados de Escalabilidade

Como referido na abordagem a seguir para avaliar esta solução a escalabilidade seria uma das métricas em questão. Contudo, como referido na secção 4.5, não foi possível escalar a solução, pelo que esta métrica não será aqui avaliada. Embora não se tenha conseguido escalar o sistema, nos resultados apresentados na secção 5.3.2.1, podemos observar que o sistema passou a suportar uma carga bastante superior ao que a solução atual suportava (secção 5.3.1). Esta diferença deve-se essencialmente ao facto da utilização de um sistema de filas de mensagens e da respetiva utilização de micro serviços com responsabilidades independentes.

6 Conclusões

Este capítulo apresenta as conclusões finais de todo o trabalho aqui desenvolvido, onde são referidos os objetivos alcançados e os não alcançados, também sendo dado a conhecer o trabalho futuro que será possível e que se espera realizar.

6.1 Objetivos Alcançados

Os objetivos do trabalho desenvolvido passavam por construir uma solução capaz de melhorar o desempenho, a escalabilidade e flexibilidade dos serviços fornecidos pela Glintt-HS, também reduzindo o tempo de manutenção desses serviços. De um modo geral, todos os objetivos foram alcançados, melhorando o desempenho em termos de tempo de resposta e aumentando a escalabilidade, fornecendo a possibilidade de escalar os serviços através da duplicação de todo o sistema ou mesmo através de replicação de uma determinada funcionalidade. O aumento da flexibilidade também foi algo observado através da introdução do Sistema de Filas de Mensagens e do ESB, permitindo integrar sistemas de uma forma mais abstrata e mais rápida. O ESB a funcionar como orquestrador e a adoção de micro serviços que comunicam através de eventos, permite que facilmente sejam adicionadas novas funcionalidades, também facilitando alteração de funcionalidades já existentes (Schabowsky, 2018). Quanto à redução do tempo/custo de manutenção, esta arquitetura também se apresenta como uma boa solução dado que o sistema se tornou muito mais independente entre si, sendo que cada micro serviço diz respeito apenas a um único domínio, tornando mais fácil o processo de manutenção na eventualidade de ocorrer um erro. O custo de manutenção por outro lado, tornou-se mais complexo, dado que existe mais *software* que necessita de ser monitorizado para garantir o correto funcionamento de todo o sistema.

6.2 Objetivos Não Alcançados

Embora de um ponto de vista conceptual todos os objetivos tenham sido alcançados, também era objetivo desta dissertação comprovar estatisticamente que esses objetivos foram realmente atingidos. Para isso seria necessário implementar uma solução que tirasse partido das funcionalidades de escalabilidade fornecidas por soluções deste tipo, sendo também necessário que esta solução seja adotada pela Glintt-HS para avaliar se realmente existe uma redução do tempo de manutenção. Contudo, tal não foi possível, podendo-se afirmar que alguns dos objetivos foram atingidos, mas apenas de um ponto de vista conceptual, como é o caso da escalabilidade e da redução do tempo de manutenção.

6.3 Trabalho Futuro

Como trabalho futuro, seria interessante implementar este sistema o mais distribuído possível, conforme apresentado na Figura 21, tirando o máximo partido das capacidades de escalabilidade que estes sistemas fornecem.

É expectável que esta seja a arquitetura adotada pela Glintt-HS para desenvolver e entregar *software* com mais qualidade, também permitindo que se avalie se os tempos de manutenção foram realmente reduzidos.

Espera-se que num futuro, os serviços da Glintt-HS, sejam completamente independentes entre si, rápidos e que permitam ser facilmente modificados.

Referências

Abeysinghe, A. (2016). *Event-Driven Arquitechture: The Path to Increased Agility and High Expandability.* Asanka Abeysinghe.

Apache. (2018, September). *Apache JMeter*. Retrieved from Apache JMeter: Getting Started: https://jmeter.apache.org/usermanual/get-started.html

Apache Kafka. (2017, February). Retrieved from Apache Kafka: https://kafka.apache.org/documentation/#introduction

Baker, J. (2017, Outubro). *By 2020, 50% of Managed APIs Projected to be Event-Driven*. Retrieved from realtimeapi.io: https://realtimeapi.io/2020-50-percent-managed-apis-projected-event-driven/

Best Message Queue Solutions. (n.d.). Retrieved from itcentralstation.com: https://www.itcentralstation.com/categories/message-queue

Burns, J. (2011). *When Is an Enterprise Service Bus (Esb) the Right Choice for an Integrated Technology Solution?*

Business Model Canvas: A Complete Guide. (2015). Retrieved from Cleverism: https://www.cleverism.com/business-model-canvas-complete-guide/

Cowan, A. (2018). *The 20 Minute Business Plan: Business Model Canvas Made Easy*. Retrieved from AlexCowan: https://www.alexandercowan.com/business-model-canvas-templates/

Dewulf, K. (2016). *Importance of the Front- End Stage in the Innovation Process.* Kristel Dewulf.

Etzion, O. (2005). *Towards an Event-Driven Architecture: An Infrastructure for Event Processing Position Paper.*

Fowler, M. (2017). *What do you mean by "Event-Driven"?* Martin Fowler.

Front End Innovation - What is the New Concept Development (NCD) model? (2018). Retrieved from Frontendinnovation.com: http://frontendinnovation.com/fei/what-is-the-new-concept-development-ncd-model

Humphrey, P. (2017). *Understanding When to use RabbitMQ or Apache Kafka*. Retrieved from Content.pivotal.io: https://content.pivotal.io/rabbitmq/understanding-when-to-use-rabbitmq-or-apache-kafka

Kress, J., Maier, B., Normann, H., Schmeidel, D., Schmutz, G., Trops, B., . . . Winterberg, T. (2013). *Enterprise Service Bus*. Retrieved from Oracle.com: http://www.oracle.com/technetwork/articles/soa/ind-soa-esb-1967705.html

Krutchen, P. (1995). *Architectural Blueprints—The "4+1" View Model of Software Architecture.*

Kuch, J. (2014). *RabbitMQ Hits One Million Messages Per Second on Google Compute Engine.* Pivotal.

Maréchaux, J.-L. (2006). *Combining Service-Oriented Architecture and Event-Driven Architecture using an Enterprise Service Bus*. Retrieved from Ibm.com: https://www.ibm.com/developerworks/library/ws-soa-eda-esb/

Martin L. Abbott, M. T. (2015). *The Art of Scalability.* Addison-Wesley Professional.

Martinez, L. F., & Ferreira, A. I. (2010). *Análise de dados com SPSS: Primeiros Passos.* Escolar Editora.

Miles, L. (2011). *Value Engineering | Value Analysis | Value Methodology | Value Management*. Retrieved from Valuefoundation.org: http://www.valuefoundation.org

Mulesoft. (2018). *API-led Connectivity: The next step in the evolution of SOA.* Mulesoft.

Nannoni, N. (2015). *Message-oriented Middleware for Scalable Data Analytics Architectures.* Nicolas Nannoni.

Newman, S. (2015). *Building Microservices.* O'Reilly Media.

Paired Sample T-Test - Statistics Solutions. (2018). Retrieved from Statistics Solutions: http://www.statisticssolutions.com/manova-analysis-paired-sample-t-test/

Pearlman, S. (2016, October). *Enterprise Service Bus vs Traditional SOA*. Retrieved from blogs.mulesoft.com: https://blogs.mulesoft.com/dev/connectivity-dev/esb-vs-soa/

RabbitMQ - Messaging that just works. (2018). Retrieved from RabbitMQ.com: https://www.rabbitmq.com

RabbitMQ by Pivotal. (2018). *Consumer Acknowledgements and Publisher Confirms.* Retrieved from rabbitmq: https://www.rabbitmq.com/confirms.html

RAML 200. (2018). Retrieved from raml.org: https://raml.org/developers/raml-200-tutorial

Reis, G. M., & Júnior, J. I. (2007). *Comparação de testes paramétricos e não paramétricos aplicados em delineamentos experimentais.*

Richards, M. (2015). *Microservices vs. Service-Oriented Arquitechture.* O'Reilly Media.

Richardson, C. (2018). *Microservices Patterns.*

Schabowsky, J. (2018). *Event-Driven Microservices: What Enterprise Architects Need To Know.*

Smith, C. U., & Williams, L. G. (2001). *Introduction to Software Performance Engineering.*

Sun Microsystems. (1997). *JavaBeans.* Sun Microsystems.

Tennakoon, C. (2017, November 12). *Spring AMQP / RabbitMQ Topic Exchange* . Retrieved from springbootdev: https://springbootdev.com/2017/11/12/spring-amqp-rabbitmq-topic-exchange-example-part-1-producer-application/

Tilkov, S. (2012). *Breaking the Monolith.*

Truren, B. (2010). *Improving software flexibility in a smart business network.*

Vanlightly, J. (2017). *RabbitMQ vs Kafka Part 1 - Two Different Takes on Messaging*. Retrieved from Jack Vanlightly: https://jack-vanlightly.com/blog/2017/12/4/rabbitmq-vs-kafka-part-1-messaging-topologies

What is an ESB. (2018). Retrieved from Mulesoft: https://www.mulesoft.com/resources/esb/what-esb

Why should I use a paired t test? (2018). Retrieved from Minitab: http://support.minitab.com/en-us/minitab/17/topic-library/basic-statistics-and-graphs/hypothesis-tests/tests-of-means/why-use-paired-t/

Widom, J., & Ceri, S. (1996). *Active Database Systems: Triggers and Rules For Advanced Database Processing.* San Francisco: Morgan Kaufmann.

Woodal, T. (2003). *Conceptualising 'Value for the Customer': An Attributional, Struc- tural and Dispositional Analysis.* Tony Woodal.

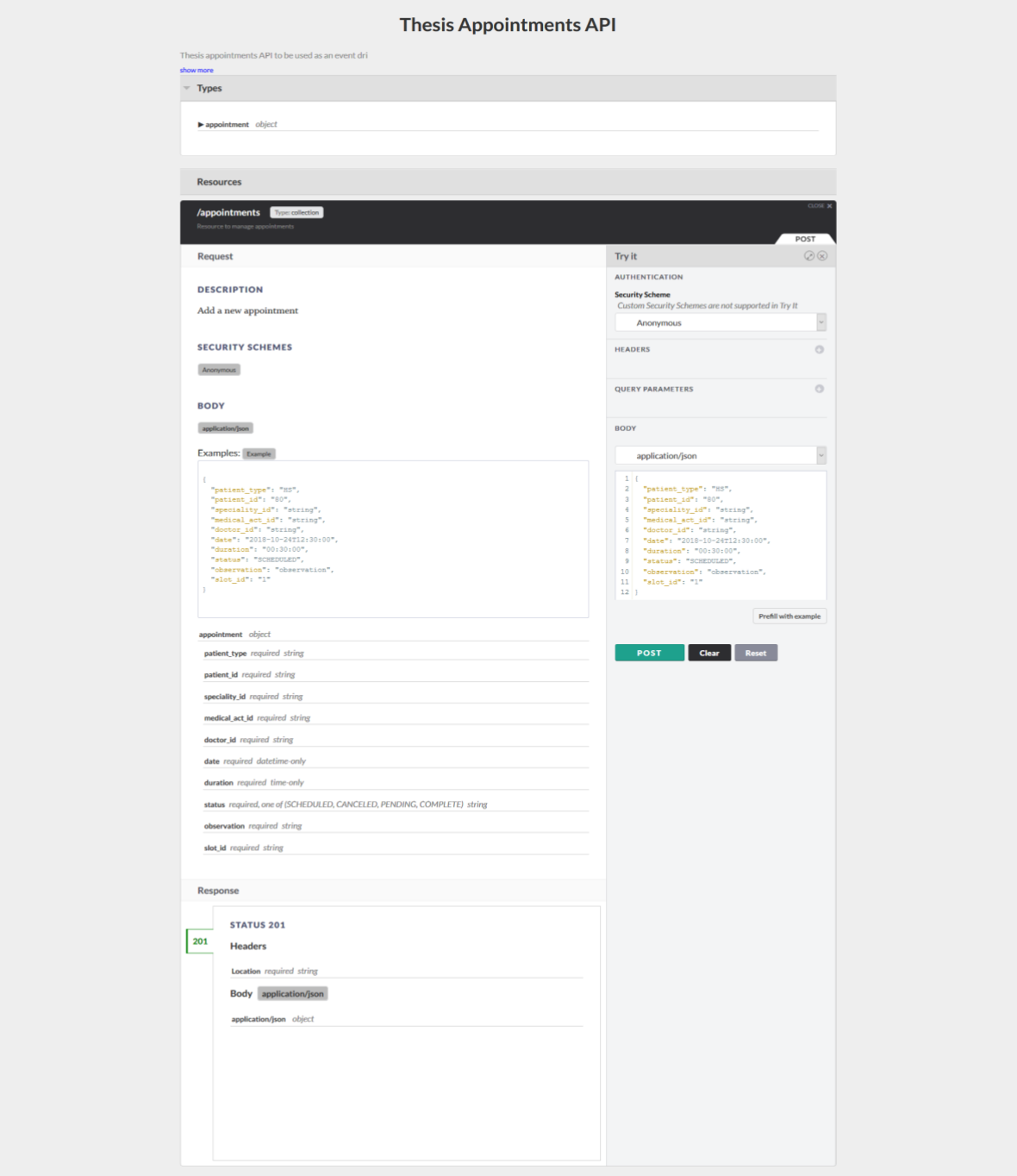
WSO2 | The Open Source Technology for Digital Business. (2018). Retrieved from Wso2.com: https://wso2.com

WSO2. (2018). *Bringing Innovation to Your Enterprise*. Retrieved from wso2.com: https://wso2.com/library/conference/2017/11/wso2con-eu-2017-panel-discussion-bringing-innovation-to-your-enterprise-use-cases-with-wso2/

Zaymus, M. (2017). *Decomposition of monolithic web application to microservices.*

7 Anexos

7.1 Anexo 1 – Documentação gerada pelo RAML



7.2 Anexo 2 – Inquérito

Arquitetura Orientada a Eventos integrada com um ESB

Este formulário pretende avaliar se a arquitetura orientada eventos que foi desenhada apresentou os resultados esperados, melhorando o sistema quanto ao nível de desempenho, flexibilidade, escalabilidade e reduzindo custos de manutenção e de desenvolvimento.

O que pensas sobre a utilização de uma arquitetura orientada a eventos, integrada com o Mule ESB, em vez da arquitetura atual da Glintt?

	1	2	3	4	5	
Muito mau	◯	◯	◯	◯	◯	Muito bom

Quando é necessário alterar ou adicionar uma nova funcionalidade a um serviço já existente é necessário que este seja alterado. Atualmente, esta alteração implica que seja feita uma publicação do Web Service onde o serviço alterado está hospedado, causando impacto em todos os serviços hospedados no mesmo Web Service. Além disso, o acréscimo de novas funcionalidades fazem com que os serviços contenham demasiada lógica e demasiada responsabilidade. Este comportamento está correto?

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

Concordas que a arquitetura orientada a eventos aumenta o desempenho de todo o sistema?

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

Concordas que a arquitetura orientada a eventos torna o sistema mais escalável e fléxivel?

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

Concordas que o custo de desenvolvimento de novas funcionalidades e de manutenção de um sistema orientado a eventos é menor do que o custo do sistema atual?

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

7.3 Anexo 3 – Script RStudio, Análise do Inquérito

```
q1<-c(3,4,4,4,5,5,5,5,5,5)
shapiro.test(q1)
wilcox.test(q1, mu = 4.5, alternative = "less")

q2<-c(1,1,1,1,1,1,2,2,4,4)
shapiro.test(q2)
wilcox.test(q2, mu = 1.5, alternative = "greater")

q3<-c(3,4,4,5,5,5,5,5,5,5)
shapiro.test(q3)
wilcox.test(q3, mu = 4.5, alternative = "less")

q4<-c(4,4,4,4,5,5,5,5,5,5)
shapiro.test(q4)
wilcox.test(q4, mu = 4.5, alternative = "less")

q5<-c(3,4,4,4,4,4,5,5,5,5)
shapiro.test(q5)
wilcox.test(q5, mu = 4.5, alternative = "less")
```

7.4 Anexo 4 – Script RStudio, Análise de Tempos de Resposta e Consumos de CPU

```
#Atual
boxplot(dotnet_low$elapsed - rabbit_low$elapsed, main="Outliers for response times on current solution and the new one with RabbitMQ. Small load")
wilcox.test(dotnet_low$elapsed, rabbit_low$elapsed, alternative = "greater")

boxplot(dotnet_mid$elapsed - rabbit_mid$elapsed, main="Outliers for response times on current solution and the new one with RabbitMQ. Medium load")
t.test(dotnet_mid$elapsed, rabbit_mid$elapsed, alternative = "greater")

boxplot(dotnet_low$elapsed - kafka_low$elapsed, main="Outliers for response times on current solution and the new one with Kafka. Small load")
wilcox.test(dotnet_low$elapsed, kafka_low$elapsed, alternative = "greater")

boxplot(dotnet_mid$elapsed - kafka_mid$elapsed, main="Outliers for response times on current solution and the new one with Kafka. Medium load")
t.test(dotnet_mid$elapsed, kafka_mid$elapsed, alternative = "greater")

#Brokers
boxplot(kafka_low$elapsed - rabbit_low$elapsed, main="Outliers for response times on new solutions with RabbitMQ and Kafka. Small load")
wilcox.test(rabbit_low$elapsed, kafka_low$elapsed, alternative = "less")

boxplot(rabbit_mid$elapsed - kafka_mid$elapsed, main="Outliers for response times on new solutions with RabbitMQ and Kafka. Medium load")
wilcox.test(rabbit_mid$elapsed, kafka_mid$elapsed, alternative = "less")

boxplot(rabbit_high$elapsed - kafka_high$elapsed, main="Outliers for response times on new solutions with RabbitMQ and Kafka. High load")
wilcox.test(rabbit_high$elapsed, kafka_high$elapsed, alternative = "greater")

#CPU
rabbit_cpu<-(rabbit_resources[grep("localhost CPU", rabbit_resources$label, ignore.case=T),])
kafka_cpu<-(kafka_resources[grep("localhost CPU", kafka_resources$label, ignore.case=T),])
dotnet_cpu<-(dotnet_resources[grep("localhost CPU", dotnet_resources$label, ignore.case=T),])

kafka_cpu<-(kafka_cpu[order(kafka_cpu$elapsed),])
rabbit_cpu<-(rabbit_cpu[order(rabbit_cpu$elapsed),])
dotnet_cpu<-(dotnet_cpu[order(dotnet_cpu$elapsed),])

boxplot(kafka_cpu$elapsed)
boxplot(rabbit_cpu$elapsed)
boxplot(dotnet_cpu$elapsed)

boxplot(kafka_cpu$elapsed - rabbit_cpu$elapsed, main="Outliers for CPU consumption on new solutions with RabbitMQ and Kafka. High load")
wilcox.test(kafka_cpu$elapsed, rabbit_cpu$elapsed, alternative = "greater")
```