USO DE MICRO SERVIÇO EM SISTEMA MONOLÍTICO: CASO WINKER S/A[1]

Kauê Rodrigues dos Santos

Resumo: A popularização da internet e dos dispositivos de comunicação demandam mais flexibilidade e escalabilidade dos sistemas operacionais. Este estudo de caso mostra como a empresa Winker S/A aplicou o conceito de micro serviços reaproveitando seu sistema até então monolítico (onde os módulos são executados em uma mesma máquina) e obteve mais flexibilidade no desenvolvimento, mais qualidade e maior desempenho.

Palavras-chave: Sistema monolítico. Micro serviço. Aplicativo. Escalabilidade.

1 INTRODUÇÃO

A exigência do usuário de tecnologia quanto a performance e relevância está cada vez maior. Seja por espaço em seu smartphone ou por ausência de recursos integrados, o volume de desinstalações é tema constante na mesa de empresários que querem que seu aplicativo seja um sucesso. Para enfrentar esse desafio, está ocorrendo o nascimento de muitas novas tecnologias, sejam elas em formas de linguagens, paradigmas, modelos, práticas ou até mesmo infraestrutura. Segundo Richards (2015), conhecer as vantagens e desvantagens de cada arquitetura visando a melhor escolha para um dado cenário é fundamental para o sucesso de um negócio.

Este trabalho apresentará conceitualmente a arquitetura de micro serviços, API Gateway, utilizando como fontes principais Lewis e Fowler, bem como Chris Richardson, entre outros estudiosos que servem como base para o entendimento das vantagens e desvantagens do sistema de micro serviços no estudo de caso, tema deste artigo.

A partir do levantamento teórico e análise das vantagens e desvantagens é apresentado como a Winker S/A aplicou estes conceitos na sua arquitetura atual e quais os resultados já esperados e obtidos.

---

[1] Artigo apresentado como trabalho de conclusão de curso de graduação da Universidade do Sul de Santa Catarina, como requisito parcial para obtenção do título de Bacharel em Ciências da Computação. Orientador: Prof. Maria Inês Castineira, Dr. Palhoça/SC, 2018.

2 ARQUITETURA DE SOFTWARE

Baseado em autores como Sommerville, Pressman, Fowler, Booch, Valipour, Garlan e Tanenbaum, Jesus (2013) pontua que a arquitetura de software pode ser considerada uma estrutura de componentes do sistema, com capacidade de se comunicarem entre si e, quando bem documentada, auxilia na comunicação profissional-cliente, bem como, permite melhor gerência do software, facilitando fatores como reuso, manutenibilidade, extensibilidade e escalabilidade.

No meio de tecnologia, é comum a afirmação de que “nenhum sistema nasce pronto”. Ou seja, por maior que seja a qualidade do código-fonte original, bem como a sua arquitetura, na medida em que o sistema vai crescendo em funcionalidades ou usuários, é comum a revisão de sua arquitetura para aumentar o desempenho e a flexibilidade do mesmo,

Segundo Newman (2015) as bases de códigos crescem à medida que se escreve código para adicionar novos recursos. Ao longo do tempo, pode ser difícil saber onde uma mudança precisa ser feita, pois, a base do código é muito grande.

A Arquitetura de Software possui estilos que funcionam como modelos para projetar o software, fornecendo informações superficiais sobre este, que é organizado através desses estilos. “Há vários tipos de estilos de arquitetura de software, como arquitetura em camadas, piper and filter, modelo de repositórios, arquitetura cliente-servidor, arquitetura orientada a objetos, entre outros”. (JESUS, 2013).

Há todo um legado de aplicações criadas que usam metodologias um pouco engessadas, ultrapassadas, mas que ao mesmo tempo não podem levar ao descarte da aplicação. Algumas destas apresentam estrutura monolítica, como será aprofundado a seguir.

2.1 INFRAESTRUTURA MONOLÍTICA

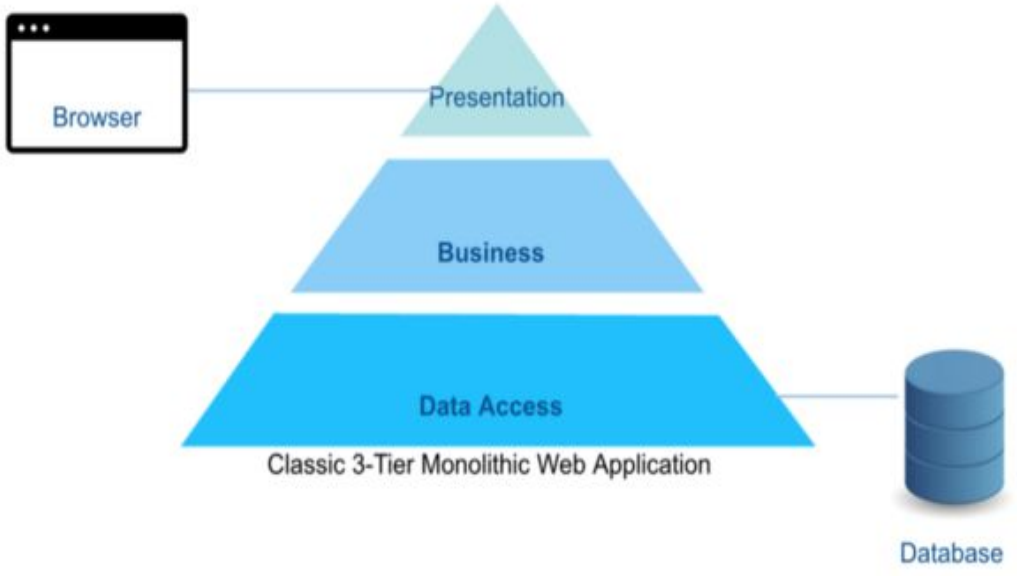
Como indica Santos (2017) “se você é um desenvolvedor há algum tempo e já trabalhou em alguma empresa relativamente grande, com certeza teve contato com os famosos "legados" ou então o grande monólito da aplicação”.

Uma aplicação monolítica descreve uma única aplicação de software em camadas onde a interface de usuário e código de acesso aos dados são combinados num único

programa partindo de uma única plataforma. Essa aplicação é autônoma e independente de outras aplicações de computação.

"Uma aplicação monolítica é aquele tipo de aplicação na qual toda a base de código está contida em um só lugar, ou seja, todas as funcionalidades estão definidas no mesmo bloco." (SANTOS, 2017). Sendo assim, em uma infraestrutura monolítica os serviços que compõem um sistema são organizados logicamente no mesmo código fonte e unidade de instalação. Isso permite a dependência entre os serviços que serão gerenciados dentro do mesmo ambiente de execução e também significa que modelos e recursos comuns podem ser compartilhados entre os componentes do sistema (WOODS, 2015). Ou seja, uma aplicação monolítica é quando toda a base de código está contida em um só lugar e todas as funcionalidades estão definidas no mesmo núcleo ou bloco. Geralmente um sistema monolítico é dividido em 3 partes conforme a Figura 1:

Figura 1 - Nível monolítico de aplicação web.



Fonte: Adaptado de Santos (2017).

Sobre a ilustração, Santos (2017) ainda discorre sobre a função de cada parte, a saber:

a) Apresentação (presentation): É onde o usuário visualiza o sistema. No caso de uma aplicação web, esta camada contém as páginas HTML com JS e CSS que serão renderizadas no navegador de internet, também conhecido como browser, de quem as acessar.

b) Negócio (business): É a camada que contém a lógica da aplicação. Nesta camada geralmente se encontram todas as bases de código, chamadas, API's e toda a inteligência do sistema em questão.

c) Dados (Data Access): Na cama de dados tem-se apenas as classes responsáveis pela conexão com o sistema de armazenamento de dados utilizado (geralmente um banco de dados relacional, como: Oracle, SQLServer entre outros).

2.1.1 Vantagens da estrutura monolítica

Muito da resistência dos empresários e programadores em mudar essa estrutura essencialmente monolítica deve-se ao fato de que a simplicidade inerente ao seu modelo traz vantagens importantes como coesão da equipe devido à padronização da tecnologia de desenvolvimento, agilidade na publicação e rápido desenvolvimento.

> Temos um sistema cujo o deploy é fácil de ser feito, já que o banco de dados facilmente evoluirá junto para todas as funcionalidades e há apenas um ponto onde o deploy precisa ser feito. Além disso, não há duplicidade de código e classes necessárias entre os diferentes módulos, já que todas elas fazem parte da mesma unidade. (ALMEIDA, 2015)

Dependendo do tamanho do sistema e da equipe, a estrutura monolítica é bem vantajosa, diante de outros modelos de estrutura.

2.1.2 Desvantagens da estrutura monolítica

Já as desvantagens da estrutura monolítica, que estimularam o desenvolvimento desse artigo, deve-se basicamente a pouca flexibilidade apresentada. Segundo Santos (2017), as desvantagens de uma aplicação monolítica são:

a) Único ponto de falha: Problema no sistema de newsletters? Não conseguimos fazer o pagamento dos funcionários porque o sistema de folha não funciona.

b) Baixa escalabilidade: É necessário copiar TODA a *stack* para escalar horizontalmente.

c) Base de código gigante: Quanto maior o sistema, maior é a base de código, já que está tudo no mesmo lugar.

d) Agregação de tecnologia: Viu isso ali em cima né? Pois então, utilizar uma única tecnologia faz com que problemas que seriam resolvidos mais facilmente com outras tecnologias sejam ignorados (o clássico "Quando você tem um martelo, todos os problemas se tornam pregos").

e) Desperdício de esforço: Imagine que você tenha que mudar o texto de uma das telas de uma aplicação monolítica de 50.000 linhas, quando você for publicar isso, ela vai ter que ser totalmente recompilada, tudo isso por causa de um texto, será que valeu a pena?

f) Demora de aculturamento: Um novo funcionário vai demorar bastante para entender como tudo funciona.

O que fica claro entre os autores é que é fundamental analisar os pontos fortes e fracos de toda a estratégia antes de fazer a troca ou complemento da estrutura.

Muitos dos desafios enfrentados por quem desenvolve software cuja arquitetura é monolítica, podem ser solucionados com o uso de micro serviços, tema que será explorado no próximo tópico.

2.2 INFRAESTRUTURA EM MICRO SERVIÇO

O uso de micro serviços não é algo novo, suas raízes remetem no mínimo aos princípios de design do próprio Unix.

Segundo Newman (2015), micro serviços são serviços com poucas responsabilidades, pequenos e autônomos que podem trabalhar de forma independente ou em conjunto com outros serviços. Ao contrário de sistemas monolíticos, cada micro serviço é empacotado em um artefato separado e, portanto, pode-se realizar a implantação de cada independentemente.

A discussão entre os desenvolvedores ocorre sobre a interpretação do quão “pequeno” deve ser o micro serviço. Entende-se portanto que deve servir sozinho a alguma função. Se resolve algo sozinho, já pode ser um micro serviço.

A aplicação do uso de micro serviço em sistemas monolíticos permite gradativamente flexibilizar a estrutura, conforme explica Woods:

> Um sistema distribuído decompõe os componentes de um sistema monolítico em unidades individuais de distribuição, capazes de evoluir com as suas próprias exigências de escalabilidade e independente dos outros subsistemas. Isso significa que o impacto de um recurso no sistema como um todo pode ser gerenciado de forma mais eficiente e a

> conexão entre componentes pode compartilhar um contrato menos rígido, pois a dependência não é mais gerenciada através de um ambiente de execução. (WOODS, 2015)

2.2.1 Vantagens da arquitetura de micro serviços

A principal vantagem em micro serviços comparado a sistemas Monolíticos é a chamada "suíte de serviços". Além do fato de que os serviços são implantados e escalam de maneiras independentes, cada serviço também provê uma fronteira bem definida entre os módulos, permitindo até mesmo que diferentes serviços sejam escritos em diferentes linguagens de programação. Eles podem inclusive serem administrados por times diferentes.

Consideram-se vantagens do uso de micro serviços:

- Maior heterogeneidade tecnológica: Considerando o desenvolvimento usando micro serviços, você pode fazer uso de novas tecnologias sem medo. Isso significa que pode ter os mais diversos profissionais produzindo módulos para o sistema com o que há de mais atual e mais performático. Como afirma Newman (2015), isso permite escolher a ferramenta certa para cada trabalho.
- Maior Resiliência: Refere-se a habilidade de recuperação de falhas nos sistemas. Como é possível isolar a falha para correção, mantendo os demais serviços ativos normalmente. Para sistemas críticos, isso é fundamental.
- Maior escalabilidade: Com aplicações monolíticas, ou você escala tudo, ou não escala nada. Com micro serviços você opta por quais partes precisam ser atualizadas e foca nelas.
- Menor complexidade: De acordo com Almeida (2015) a base de código menor, facilita a barreira inicial na compreensão do projeto para um novo membro do time. Outro fator importante é o isolamento das funcionalidades do sistema, sendo assim, esse novo modelo arquitetural também aparta as falhas que possam ocorrer em uma determinada funcionalidade, permitindo que o restante das aplicações permaneça funcional.
- Facilidade de implantação: Na medida em que você pode atualizar apenas uma *feature*, você pode ter mais rodadas de atualizações, não sendo necessário que as outras *features* estejam concluídas para subir para produção.

2.2.2 Desvantagens da arquitetura de micro serviços

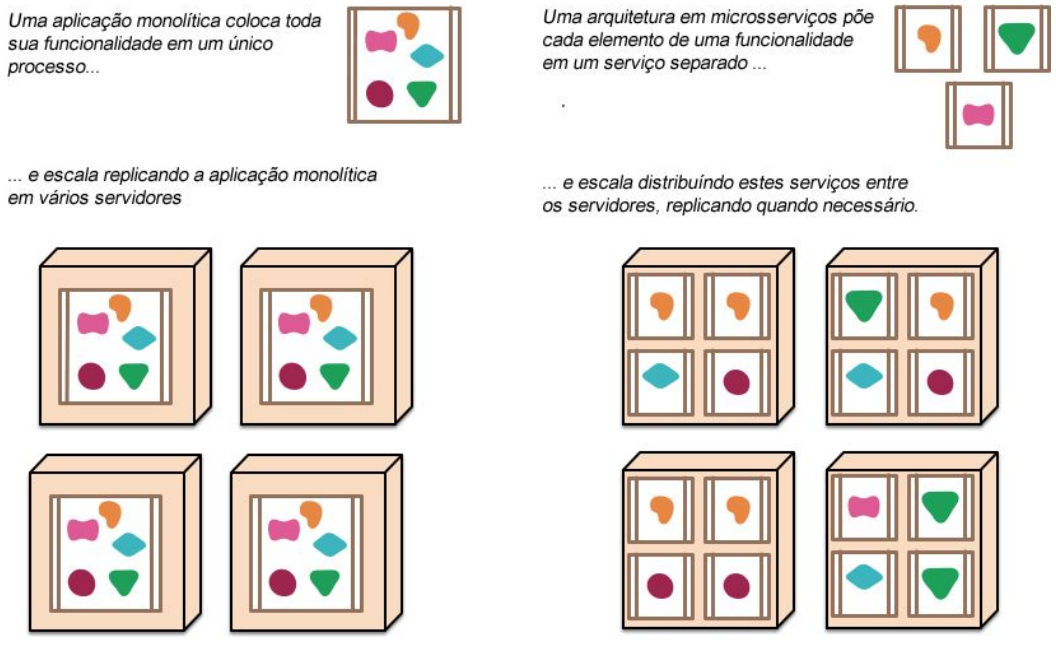
A arquitetura de micro serviços pode não ser a ideal em todas as situações, para isso nesta seção serão abordados alguns pontos onde os micro serviços assumem uma posição de desvantagens que, segundo Lewis e Fowler (2014), são os seguintes:

- Distribuição: os sistemas distribuídos são mais difíceis de programar, uma vez que as chamadas remotas são lentas e correm risco de falha.
- Consistência eventual: manter uma consistência forte é extremamente difícil para um sistema distribuído, o que significa que todos devem gerenciar a consistência eventual.
- Complexidade operacional: é necessária uma equipe de operações maduras para gerenciar muitos serviços, que estão sendo redistribuídos regularmente.

2.2.3 Diferenças entre uma estrutura monolítica e em micro serviço

A Figura 2 apresenta as diferenças entre uma estrutura monolítica e em micro serviço:

Figura 2 - Aplicações monolíticas e micro serviços.



Fonte: Adaptado de LEWIS e FOWLER (2014).

2.2.4 Características de uma arquitetura em micro serviços

Para classificar uma estrutura em micro serviços Lewis e Fowler (2014) descrevem algumas características, a saber:

a) Componentização via serviços

Componente é uma unidade de software que é substituída ou atualizada de maneira independente. Arquiteturas em micro serviços usam bibliotecas formadas por componentes, acionados por meio de chamadas de função diretamente em memória. Já serviços são componentes em processos diferentes que se comunicam através de mecanismos tais como requisições via web services ou chamadas de código remotas.

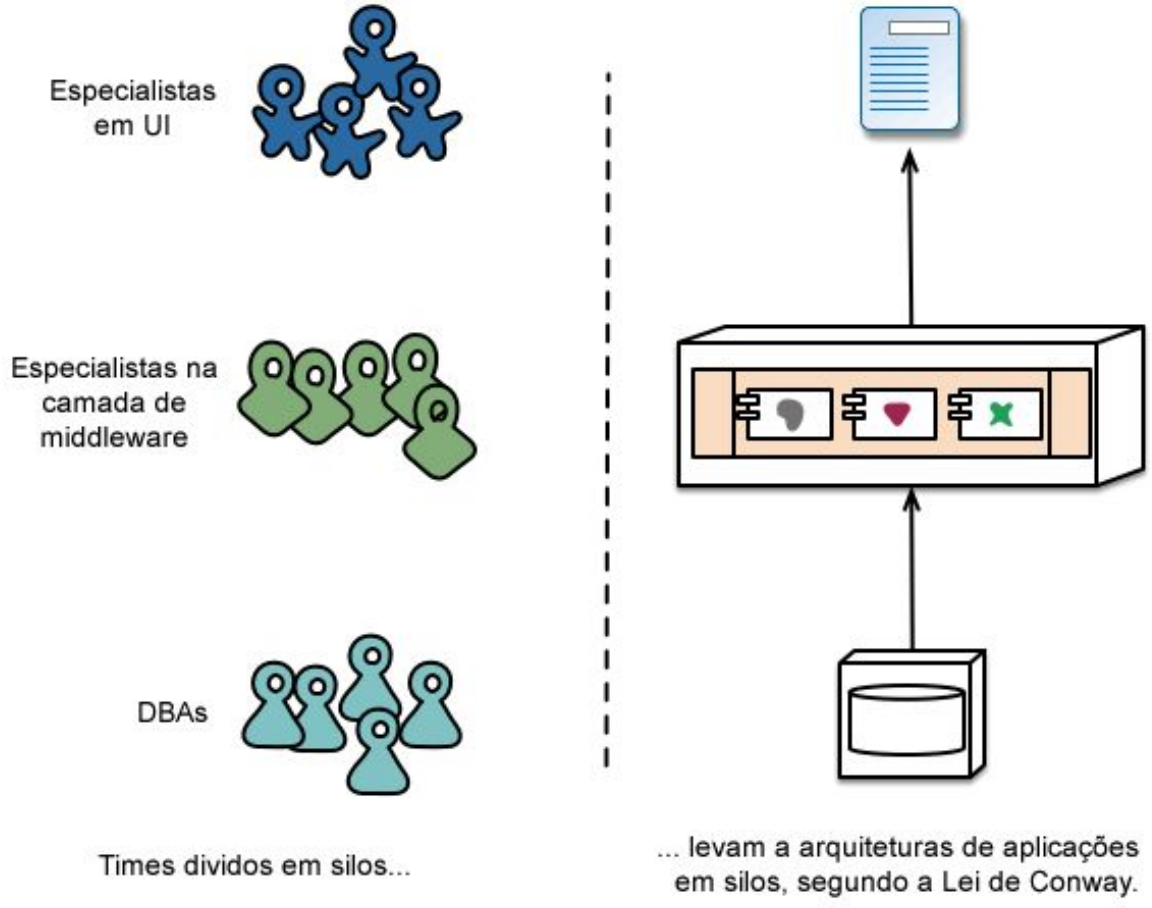
Usar serviços desta forma tem alguns efeitos. Chamadas remotas são mais custosas que chamadas dentro do mesmo processo e APIs remotas precisam ser granulares, o que torna ainda mais complicado para usar. Se você precisa mudar as responsabilidades entre os componentes, tais mudanças de comportamento são mais difíceis de fazer do que quando você consegue ultrapassar as fronteiras entre os processos.

b) Organizado através das áreas do negócio

Ao dividir uma grande aplicação em partes, o foco geralmente é na camada de tecnologia, levando os times a serem divididos entre aqueles que cuidam da interface, da lógica server-side e do banco de dados. Quando times são divididos desta forma, até mesmo mudanças simples podem exigir bastante tempo e aprovação financeira em projetos que envolvam diversos times. Ao fugir destes problemas, o time pode acabar trazendo a lógica para as aplicações que eles têm acesso. Em outras palavras, lógica em todos os lugares. Este é um exemplo da Lei de Conway em ação.

Como afirma Conway (1967 apud LEWIS, FOWLER, 2014), qualquer organização que desenha um sistema (definido de forma ampla) irá produzir um design cuja estrutura é uma cópia da estrutura de comunicação da própria organização, Figura 3.

Figura 3 - Lei de Conway em ação

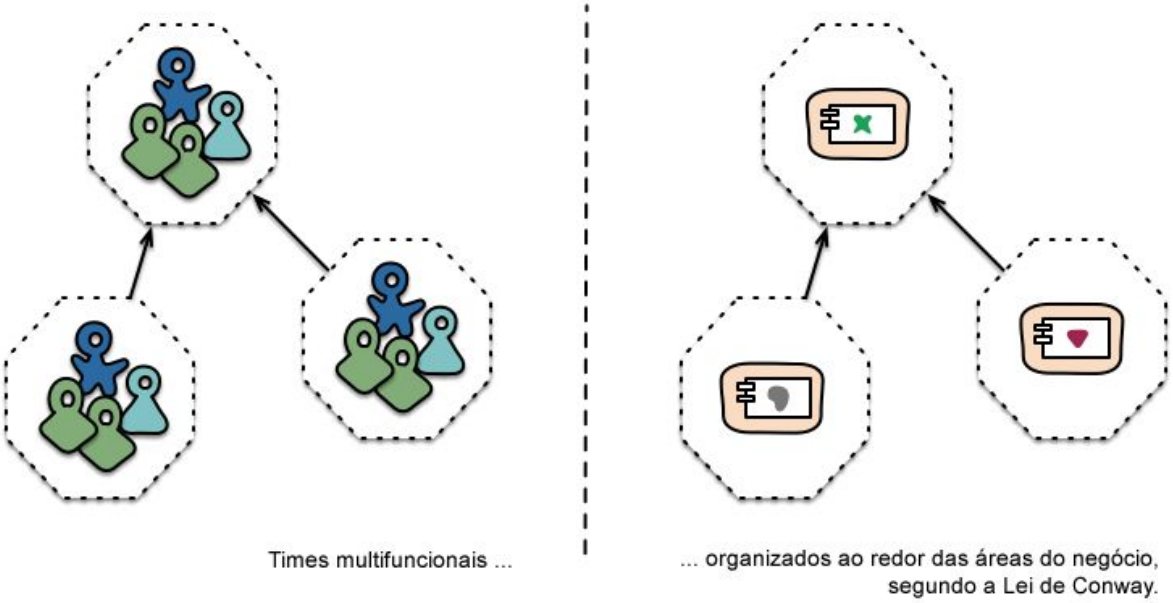


Fonte: Adaptado de LEWIS e FOWLER (2014).

Como explica Lewis e Fowler (2014), a abordagem proposta pelos micros-serviços para esta divisão é diferente, ao organizar os times ao redor das áreas do negócio. Assim os serviços possuirão uma implementação satisfatória para uma determinada área do negócio, incluindo uma interface com usuário, armazenamento de dados persistente e qualquer outra

necessidade externa. Consequentemente, os times se tornam multifuncionais, levando toda bagagem necessária para o desenvolvimento: interface com o usuário, banco de dados e o gerenciamento do projeto, Figura 4.

Figura 4 - Limites dos serviços reforçados pelos limites dos times.



Fonte: Adaptado de LEWIS e FOWLER (2014).

Micro serviços usam os princípios e protocolos que a Web usa (e em um contexto maior, o Unix também). Frequentemente os recursos usados podem ser cacheados com pouco esforço pelos desenvolvedores e o time que cuida da operação.

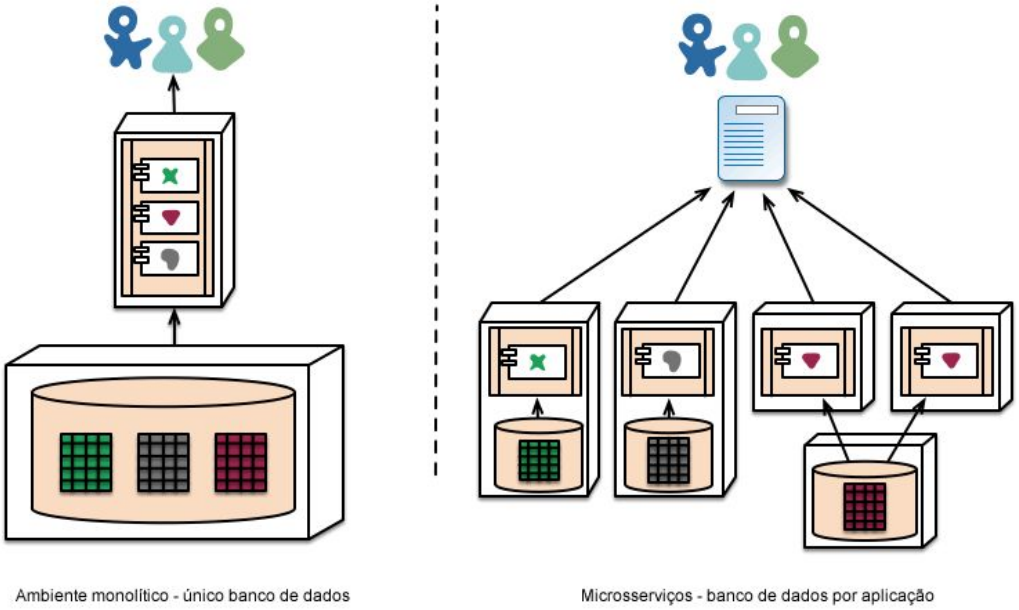
A segunda abordagem, segundo LEWIS e FOWLER (2014) geralmente usada é a comunicação através de um barramento de mensagens simples. A infraestrutura usada age como um roteador de mensagens, tais como o RabbitMQ ou ZeroMQ que proveem uma base de assincronismo confiável – a inteligência continua existindo nos endpoints que estão produzindo e consumindo mensagens nos serviços.

c) Governança descentralizada

Uma das consequências de governanças centralizadas é a tendência de padronizar tudo em uma única plataforma tecnológica. Mas é preciso ficar atento, pois nem todos os problemas são iguais, então, suas soluções também não devem ser.

Assim como micro serviços descentralizam decisões sobre os modelos conceituais, eles também descentralizam decisões sobre o armazenamento de dados. Enquanto aplicações monolíticas preferem uma única base de dados lógica para a persistência de dados, empresas frequentemente preferem uma única base de dados para uma variedade de aplicações – e muitas destas decisões são dirigidas por modelos comerciais vendidos através de licenças. Micro serviços preferem permitir que cada serviço gerencie sua própria base de dados, quer através de diferentes instâncias usando a mesma tecnologia de banco de dados, ou até mesmo usando diferentes sistemas de banco de dados – uma abordagem chamada Polyglot Persistence, feita por programadores que entendem pelo menos 4 linguagens diferentes de programação. “Pode-se usar uma persistência poliglota em uma aplicação monolítica, mas isso aparece com mais frequência com micro serviços.” (LEWIS, FOWLER, 2014), Figura 5.

Figura 5 - Banco de dados em ambiente monolítico e em micro serviços



Fonte: Adaptado de LEWIS e FOWLER (2014).

d) Padrões amplamente testados e padrões impostos

Os times que usam micro serviços tendem a desestimular o uso dos padrões simplesmente impostos pelos grupos de arquiteturas enterprise, e tendem a usar alegremente

(e até evangelizar) o uso de padrões abertos tais como HTTP, ATOM e outros micro formatos.

A diferença chave é como os padrões são criados e como eles são propostos. Padrões gerenciados por grupos como o IETF somente se tornam padrões quando existirem diversas implementações no ar e projetos open-source bem-sucedidos.

Estes padrões formam um mundo a parte de muitos no ambiente corporativo, que são frequentemente desenvolvidos por grupos que tem pouca experiência em programação, ou são fortemente influenciados por vendedores.

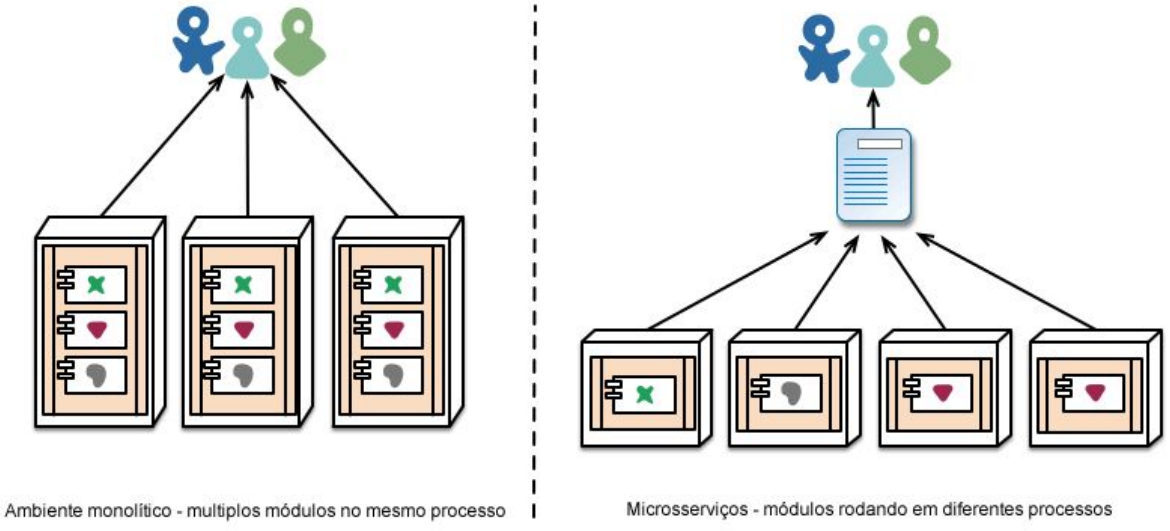
e) Automação da Infraestrutura

Técnicas para automação da infra-estrutura tiveram grande evolução nos últimos anos – como a computação na nuvem e a própria AWS em particular, que permite reduzir a complexidade de construir, publicar e operar micro serviços.

> Muitos dos produtos ou sistemas construídos com micro serviços têm sido feitos por times com uma extensa experiência em Continuos Delivery e seu precursor, a Integração Contínua. Times que constroem software desta forma fazem uso intenso das técnicas de automação de infra-estrutura. (LEWIS, FOWLER, 2014).

A ilustração a seguir apresenta as diferenças na forma de publicação em ambiente monolítico e em micro serviços.

Figura 6 - A forma de publicação frequentemente é diferente.



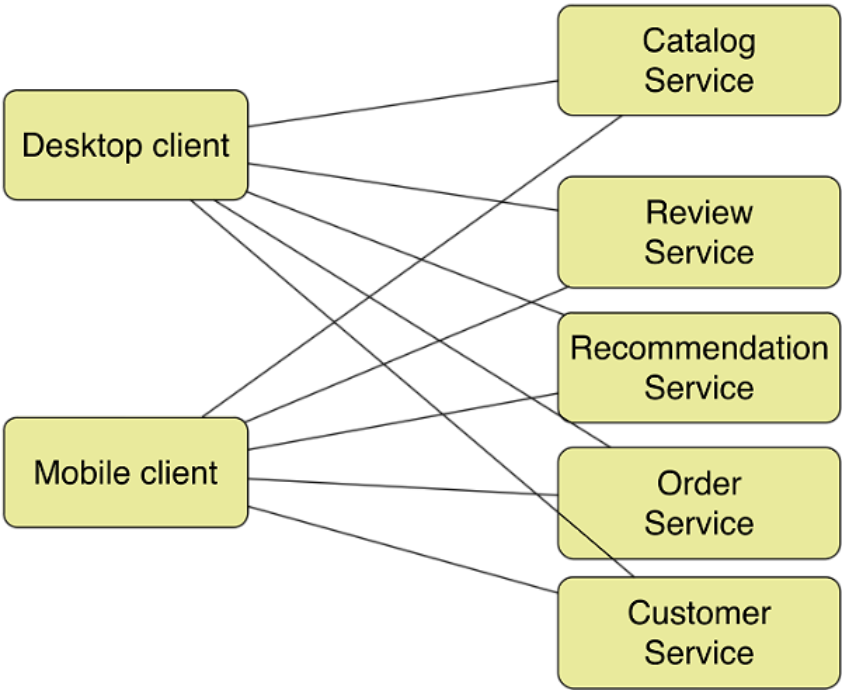
Fonte: Adaptado de LEWIS e FOWLER (2014).

2.3 PADRÃO API GATEWAY

Como afirma Richardson (2014), em uma arquitetura monolítica, os clientes da aplicação (navegadores web e aplicações nativas), realizam requisições HTTP para um balanceador de carga que direciona a requisição para uma das N instâncias idênticas da aplicação. Porém, em uma arquitetura de micro serviços, a aplicação monolítica foi substituída por uma série de serviços. Em razão disso, a questão a ser respondida é: Como os clientes interagem com os micro serviços?

Uma aplicação cliente móvel nativa, por exemplo, pode realizar requisições HTTP RESTful diretamente para os serviços como ilustrado pela figura a seguir.

Figura 7 - Chamada direta de serviços



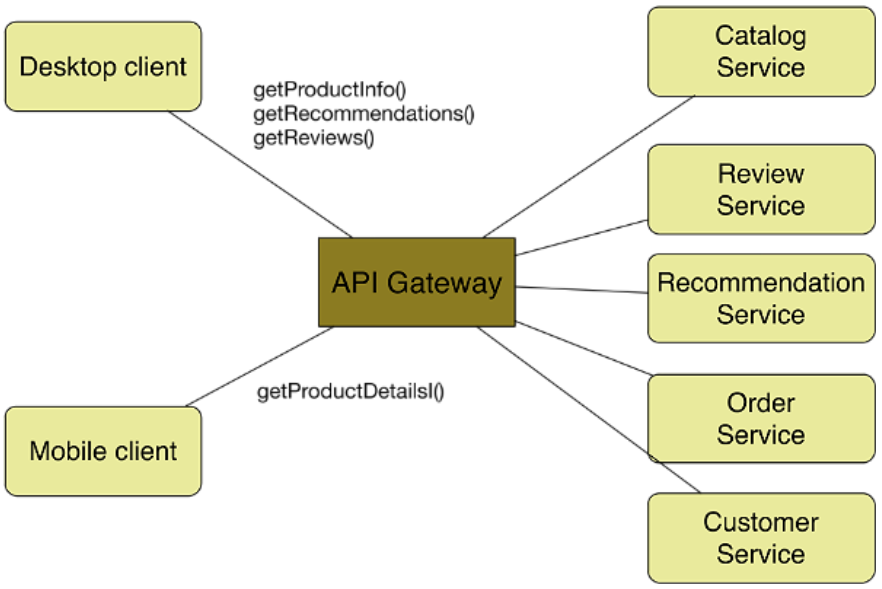
Fonte: RICHARDSON, 2014

Vale ressaltar que existe uma incompatibilidade significativa de granularidade entre APIs compostas por serviços individuais e a exigência de dados dos clientes. Por exemplo, exibir uma página web pode exigir chamadas para diversos serviços. Realizar todas essas

requisições, mesmo com internet de alta velocidade, seria muito ineficaz, resultando em uma experiência ruim ao usuário. (RICHARDSON, 2014).

Para evitar instabilidade, o ideal é realizar poucas requisições por página, preferencialmente uma, através de um servidor de front end denominado API gateway, como mostra a figura a seguir.

Figura 8 - API gateway



Fonte: RICHARDSON, 2014

Observa-se pela ilustração que a API gateway localiza-se entre os clientes (desktop ou mobile) e os serviços, ou seja, ela irá acionar os micro serviços de acordo com a funcionalidade solicitada pelo cliente.

> API gateway fornece uma API de granularidade grossa para clientes móveis e uma API com granularidade fina para clientes desktop que utilizam uma rede de alta velocidade. Neste exemplo, os clientes desktop realizam várias requisições para recuperar informações sobre um produto, porém, um cliente móvel realiza uma única requisição. (RICHARDSON, 2014)

Ao interpretar a ilustração também é possível afirmar que com a API gateway é possível a evolução dos serviços sem causar impacto aos clientes.

Até este ponto, foi apresentado o papel da API gateway como mediador entre a aplicação e seus clientes. A seguir, é apresentado como implementar a comunicação entre micro serviços.

2.3.1 Mecanismos de comunicação inter-service

Outra grande diferença da arquitetura de micro serviços é a forma de interação entre os diferentes componentes da aplicação.

> Em uma aplicação monolítica, componentes comunicam-se através de chamadas de métodos. Mas em uma arquitetura de micro serviços, diferentes serviços são executados em diferentes processos. Consequentemente, os serviços devem utilizar uma comunicação entre processos (IPC - inter-process communication). (RICHARDSON, 2014)

Existem duas principais abordagens para a comunicação entre processos em uma arquitetura de micro serviços.

Os dois métodos dentro da comunicação inter-service são:

a) HTTP síncrono

O mecanismo baseados em HTTP síncrono como REST ou SOAP é uma opção simples, conhecida e facilmente gerenciável por firewalls, ou seja, permite a comunicação através da Internet, facilitando a comunicação no formato request/reply. Richardson (2014) ressalta que a desvantagem do HTTP é a falta de suporte a outros padrões de comunicação, como publish/subscribe. Outra limitação é que cliente e servidor devem estar disponíveis

simultaneamente, situação difícil de ser garantida, pois os sistemas distribuídos são propensos a falhas parciais.

b) Mensagens Assíncronas

O mecanismo assíncrono é baseado em mensagens como um broker de mensagens AMQP.

> Esta abordagem tem muitos benefícios, pois desacopla os produtores de mensagens dos consumidores. O broker é capaz de enfileirar e manter as mensagens até serem processadas por consumidores. O produtor simplesmente envia a mensagem para o broker, sem a necessidade de um mecanismo de descoberta de serviços. A comunicação baseada em mensagens também suporta uma variedade de padrões de comunicação, incluindo os pedidos de caminho único (one-way requests) e publish-subscribe. (RICHARDSON, 2014)

Richarson (2018) ainda afirma que a desvantagem do uso de mensagens é a necessidade de um broker, que é mais um componente que aumenta a complexidade do sistema. Outra desvantagem é que a comunicação síncrona (request/reply) não se ajusta naturalmente a esse estilo de comunicação.

2.3.2 Gerenciamento de dados descentralizados

Uma consequência da decomposição da aplicação em serviços é o particionamento da base de dados.

> Para garantir o baixo acoplamento, cada serviço tem sua própria base de dados (schema). Além disso, diferentes serviços podem usar diferentes tipos de base de dados, conhecido como arquitetura de persistência poliglota. Por exemplo, um serviço que necessita de transações ACID pode utilizar um banco de dados relacional, enquanto um serviço que manipula uma rede social pode utilizar um bando de dados baseado em grafos. Particionar a base de dados é essencial, mas isso resulta em um novo problema: como lidar com as solicitações que acessam dados pertencentes a vários serviços? (RICHARDSON, 2014)

A seguir é abordado como manipular as solicitações de leitura e atualização.

2.3.3 Manipulação de requisições de leitura e de atualização

Entende-se por requisição de leitura cada acionamento que o usuário faz no sistema (consulta, cadastro, exclusão, entre outros) e para que haja performance no sistema é necessário haver um limite para a quantidade de requisições.

> A implementação dessa regra de negócio é trivial em uma aplicação monolítica. Entretanto, é muito mais difícil de implementar essa verificação em um sistema onde os clientes (customers) são gerenciados pelo CustomerService e os pedidos são gerenciados por OrderService. De alguma forma, OrderService deve acessar o limite de crédito mantido pelo CustomerService. (RICHARDSON, 2014)

Richardson (2014) cita duas soluções para esta implementação:

a) o uso de OrderService para recuperar o limite de crédito através de chamadas remotas RPC (Remote Procedure Call) ao CustomerService. A desvantagem é que ele reduz a disponibilidade, pois CustomerService deve estar em execução para que OrderService realize um pedido, além de aumentar o tempo de resposta, devido a chamada remota adicional.

b) armazenar uma cópia do limite de crédito em OrderService. Esta abordagem elimina a necessidade de uma requisição ao CustomerService, aumenta a disponibilidade e reduz o tempo de resposta. Porém, implica na implementação de um mecanismo de atualização da cópia do limite de crédito quando ocorrer alguma modificação em CustomerService.

Manter o limite de crédito atualizado no OrderService é um exemplo comum de problema resultante da manipulação de requisições de atualização de dados de diferentes serviços. Existem duas formas de atualização de requisições:

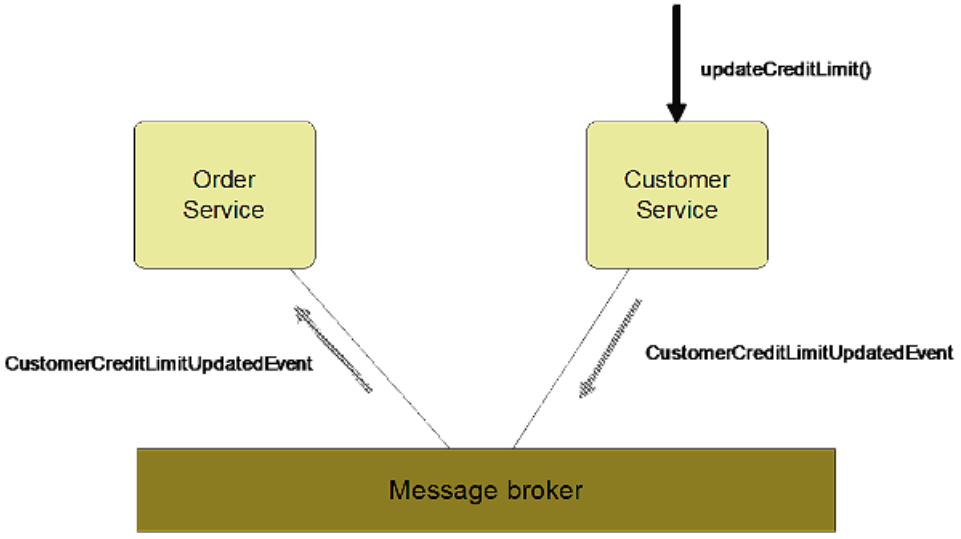
a) Transações distribuídas: consiste em utilizar em diversos serviços da aplicação para atualizar os valores.
b) Transações assíncronas baseadas em eventos: consiste em utilizar eventos emitidos pela aplicação que ativam serviços pré-determinados.

A seguir será aprofundado o método de transação assíncrona baseada em eventos.

2.3.6 Atualizações assíncronas baseadas em eventos

Richardson (2014) afirma que atualizações assíncronas baseadas em eventos são serviços publicam eventos anunciando que alguns dados foram alterados. Outros serviços que estão inscritos para receber esses eventos, atualizam seus dados. Por exemplo, quando CustomerService atualiza o limite de crédito de um cliente, ele publica um CustomerCreditLimitUpdatedEvent, que contém o ID do cliente e o novo limite de crédito. OrderService recebe esses eventos e atualiza sua cópia do limite de crédito. O fluxo de eventos é mostrado na figura a seguir.

Figura 9 - Replicação do limite de crédito através de eventos



Fonte: RICHARDSON, 2014

O grande benefício desta abordagem é o desacoplamento entre produtores e consumidores de eventos. O desacoplamento simplifica o desenvolvimento e aumenta a disponibilidade, em comparação ao uso de transações distribuídas.

> Caso nenhum consumidor esteja disponível para processar um evento, o broker de mensagens irá enfileirar o evento até que a mensagem possa ser processada. A grande desvantagem desta abordagem é troca de consistência em favor de disponibilidade. A aplicação deve ser escrita de forma a tolerar eventuais inconsistências de dados. Desenvolvedores podem ter que implementar mecanismos de transação com suporte à rollbacks por compensação. (RICHARDSON, 2014)

Apesar desses inconvenientes, esta é a abordagem preferida para muitas aplicações.

3 ESTUDO DE CASO WINKER S/A

A Winker nasceu em 2013 sob o nome de Rede Domus. Seu principal produto é uma plataforma para condomínios, onde moradores, síndicos e prestadores de serviço interagem para simplificar a vida em condomínio.

Hoje, as rotinas dos condomínios são percebidas pela maioria dos condôminos como um problema, fazendo com que as pessoas sintam desinteresse e apatia pelos assuntos relacionados ao condomínio. Participar de uma assembleia, nem sempre é motivo de alegria, pois na sua maioria, não passam de reuniões cansativas e sujeitas a desentendimentos. As pessoas passam a coabitar em vez de se relacionarem umas com as outras. Esquecem muitas vezes de que juntas podem ter mais força e poder de influência nas decisões.

A Winker surgiu com a intenção de mudar esse paradigma. Quando o sócio, fundador e atual Presidente, Henrique Melhado, foi morar em um condomínio, ele viu a necessidade de criar um sistema para resolver os problemas que lhe incomodavam. “O coletivo é formado por pessoas e um condomínio é a situação ideal para promover essa integração. Faltava apenas uma força que unisse os moradores e os fizessem enxergar que juntos podem ganhar mais”, afirma Henrique[2].

Nascida na cidade de Florianópolis, atualmente, a Winker engloba uma série de outras experiências e melhorias na vida em condomínio, que foram coletadas ao longo dos seus anos de estrada.

Henrique afirma ainda que “o mercado atual contribuiu muito para o negócio que criamos, pois com a necessidade crescente de transparência ao qual estamos acostumados, a função de síndico de condomínio foi evidenciada”.

Um dos grandes desafios da Winker hoje é atender seus mais de 5 mil condomínios espalhados pelo Brasil com a qualidade de quanto tinha apenas 40 clientes, no ano de sua fundação. Automatizar o processo de implementação é um dos pontos estratégicos da área de desenvolvimento e anda em paralelo com o desafio de crescer a plataforma incrementando novas tecnologias e integrações, sem perder o desempenho.

[2] Comunicação pessoal em 13 de novembro de 2018.

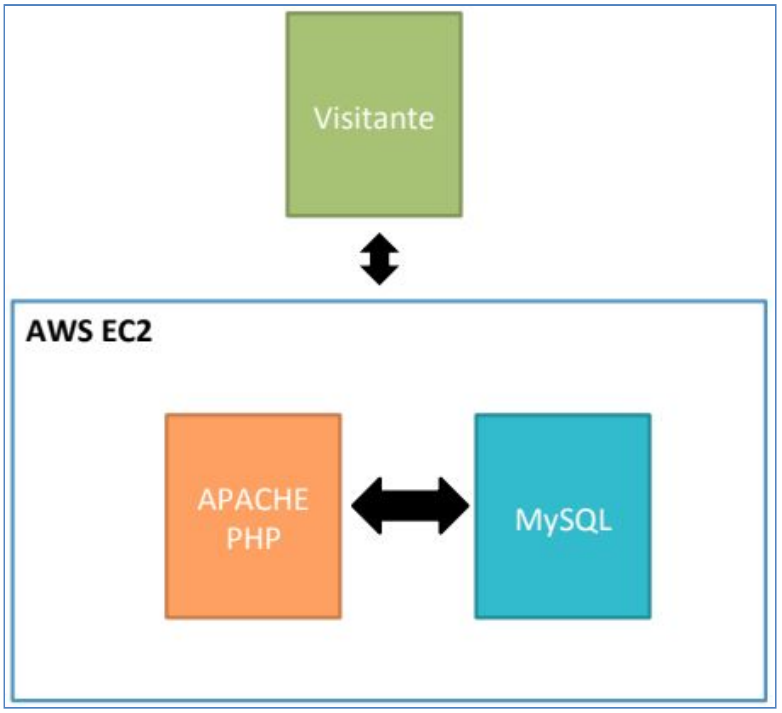
3.1 SISTEMA E ARQUITETURA INICIAL

Neste cenário, a escolha por utilizar micro serviços não foi tão simples, o sistema orginalmente foi desenvolvido como monolítico visto que era administrado por apenas dois programadores. Hoje, com mais de 8 programadores, alguns deles trabalhando em home-office esporadicamente, seria impossível manter a produtividade dos diversos profissionais sem que fosse optado pela estrutura de micro serviços.

A ideia da empresa é também permitir que empresas do segmento possam integrar com a plataforma da Winker sem demandar tanto dos seus colaboradores, por meio de uso de APIs e documentação.

A arquitetura inicial era monolítica e como descreve SANTOS (2017), possuía seus códigos basicamente em uma única plataforma, isso trazia para empresa diversos problemas, como baixa performance, difícil escalabilidade e principalmente a dificuldade de integração, a necessidade de evoluir comercialmente e produtivamente impunha grande capacidade de poder integrar com fácil curva de aprendizagem e alto nível de desacoplamento, Figura 10.

Figura 10 - Arquitetura monolítica do sistema Winker



Fonte: O autor (2018)

O sistema monolítico atendeu as necessidades de crescimento do sistema Winker enquanto a equipe era restrita a 3 desenvolvedores. O sistema foi inicialmente desenvolvido em PHP com framework yii, utilizando banco de dados MySQL. Devido a limitações do framework era difícil utilizar outras bibliotecas. Na medida que houve necessidade de crescimento da equipe, a estrutura monolítica não trazia produtividade. Neste momento da empresa, a implementação da estrutura de micro serviços começou a ser estudada.

3.2 USANDO MICRO SERVIÇOS

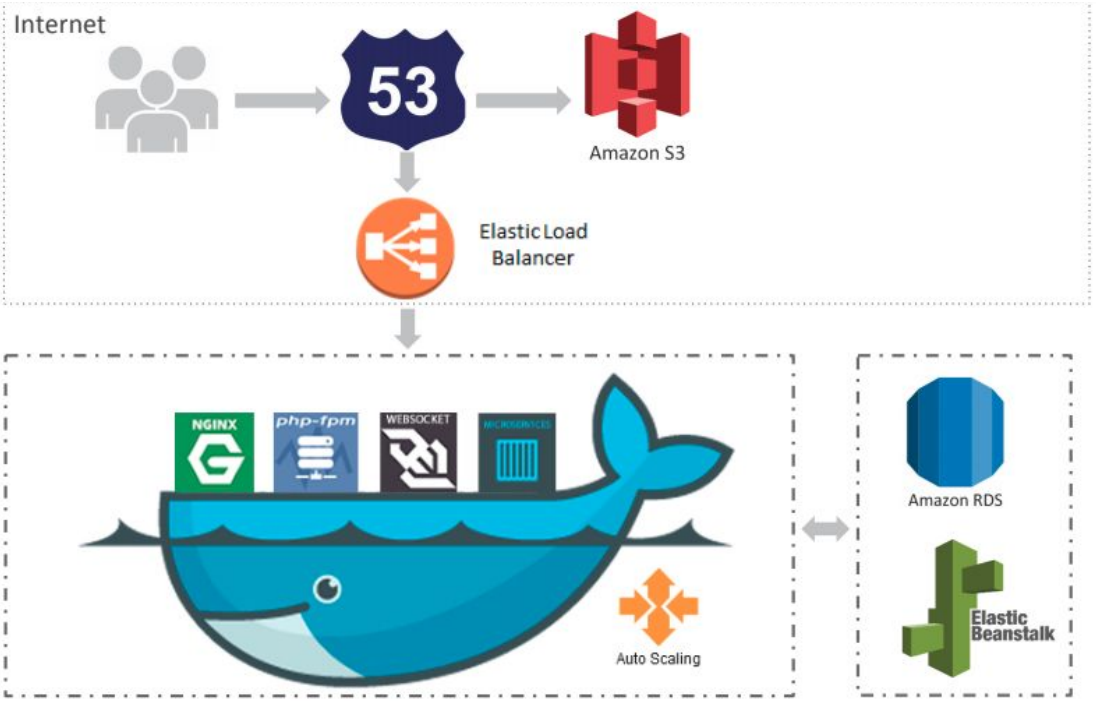
De acordo com Newman (2015) e Woods (2015) as bases de códigos crescem a medida que novos recursos são implementados e as mudanças de códigos acabam se dificultando, em função do alto acoplamento de código. Dessa forma em julho de 2017, optou-se por uma saída através de micros serviços, também levando em consideração a possibilidade de flexibilizar a estrutura.

Foram consideradas as seguintes vantagens: Heterogeneidade tecnológica, maior resiliência, escalabilidade, complexidade e implantação.

O processo do novo modelo se inicia pela Infraestrutura a fim de buscar a fácil distribuição do sistema e alto nível de escalabilidade. Como demonstra Lewis e Fowler (2014) a grande evolução da computação na nuvem trouxe grandes benefícios para a área de infraestrutura, em especial a AWS Amazon.

Observa-se na figura a seguir a arquitetura da Winker já usando o conceito de micro serviços.

Figura 11 - Arquitetura Winker usando micro serviços



Fonte: O autor (2018)

A estrutura inicia-se por um *load balancer* que distribui cargas de trabalho para várias instâncias do micro serviço, o micro serviço está contido em um *docker* que já acopla o *Nginx*, php-fpm, *websocket* e facilita o auto escalonamento, através do *Elastic Beanstalk* que pode-se aumentar ou diminuir a quantidade de máquinas dedicadas a cada micro serviço.

Os sistemas de cache, fila, banco de dados e armazenamento também ficam na nuvem.

3.3 ELABORANDO O MICRO SERVIÇO PRINCIPAL

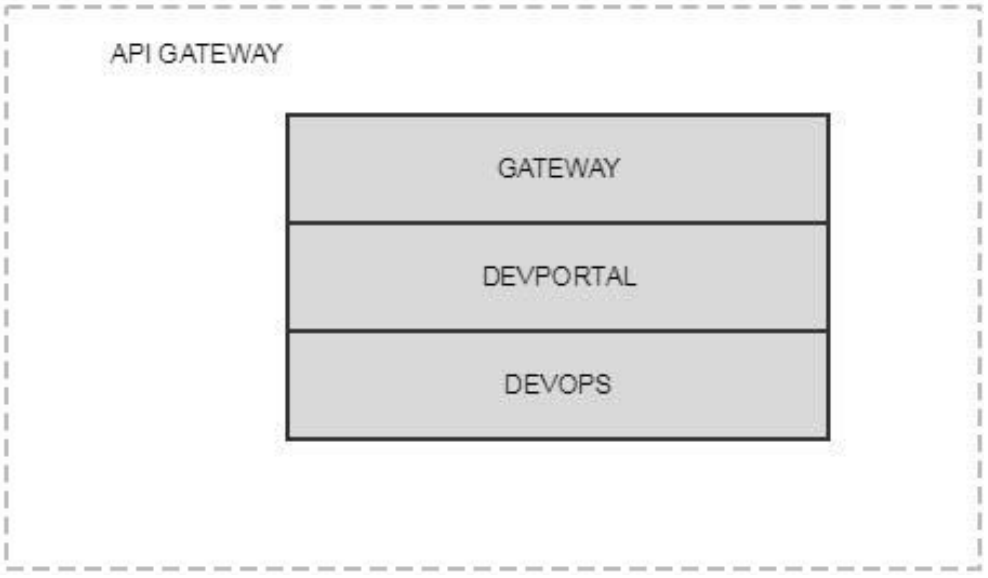
Elaborou-se a componentização da arquitetura, ficando uma maior componentização de serviços organizados através de áreas de negócios, governanças separadas e automação da estrutura.

O novo modelo se adequa ao padrão Api Gateway no qual uma estrutura que reúne diversos serviços consegue administra-los de modo que seus desacoplamentos não interfiram no funcionamento do conjunto da aplicação.

Buscou-se aplicar o mesmo modelo de comunicação adotado de modo que nossa comunicação inter-service fosse executada em diferentes processos, são eles Http Síncrono e Mensagens Assíncronas.

Ficando assim a arquitetura principal do projeto, Figura 12:

Figura 12 - Arquitetura principal do projeto Winker



Fonte: O autor (2018)

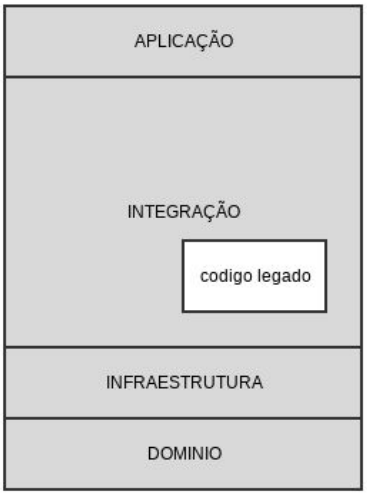
O micro serviço possui 3 grandes componentes.

a) *Gateway*: nesta camada possui-se todas rotas, logica de funcionamento utilizados pelo micro serviço através de Http Síncrono.
b) *Devportal*: nesta camada encontra-se toda logica de funcionamento utilizada para Mensagens Assíncronas.
c) *Devops*: Nesta camada encontra-se toda a estrutura de desenvolvimento e operações da aplicação, como gerenciamento de servidor e filas de processamentos.

3.3.1 Gateway

O componente Gateway possui 4 grandes módulos, como demonstra a figura:

Figura 13 - Módulos do componente Gateway da Winker



Fonte: O autor (2018)

a) Aplicação

Esse módulo é responsável por tudo que se diz relacionado ao funcionamento lógico da aplicação.

No modelo adotado implementou-se a utilização de perfis de acesso na aplicação através de instalações de aplicativos. Esses aplicativos seriam instalações que clientes externos integradores podem configurar e utilizar como plug-ins instaláveis e desinstaláveis em seus condomínios.

Também está incluso, além do serviço de autenticação do micro serviço em si, o acesso através do método de autenticação do sistema legado, unificando o sistema legado com o novo micro serviço.

b) Integração

A camada de integração é sem dúvida a camada de maior importância do novo micro serviço. Como foi visto em capítulos anteriores é grande a dificuldade de migração de sistemas legados e suas regras complexas, tendo isso em mente o desafio foi trazer as regras de negócios para a nova plataforma de forma e tempo viável, a alternativa a esse desafio foi utilizar um compartilhamento de código.

Foi criado um sistema de inicialização da aplicação legada no qual ele executa a inicialização da aplicação através do código compartilhado e disponibiliza algumas camadas estratégicas em forma de serviços, são elas: *model*, *service* e *controller*.

c) Infraestrutura

Nessa camada foi definido tudo que se utiliza no funcionamento básico de todo micro serviços, como: gestão de cache, gestão de requisições, banco de dados, autenticação, *models*, *migrates*.

d) Domínio

Na camada de domínio criou-se as rotas que executam todas chamadas de serviços que irá ser decomposto do sistema monolítico conforme o tempo.

Através dos serviços que a camada de integração nos oferta, é possível elaborar novas regras no novo sistema, ou mantê-las no sistema legado e apenas buscar essas informações através dos serviços antigos ou, até mesmo, redirecionar para um novo micro serviço componentizando mais ainda a aplicação.

Como define Richardson (2014) este componente funciona no modo Http Síncrono, as requisições de consulta de informações chegam através de requisições que a retornam após o processamento.

3.3.2 Devportal

Esse componente possui o sistema de mensagens assíncronas que, como ressalta Richardson (2014), devido ao grande fluxo de integração com diversos parceiros, há necessidade de aceitação de um gigante fluxo de informação entrando e saindo no sistema.

Foi criado um sistema de fila de mensagens assíncronas, no qual armazena-se e processa toda a informação de acordo com a demanda necessária.

Ao chegar uma requisição assíncrona gera-se um evento e envia-se para ser processada posteriormente.

Foi criada uma biblioteca de eventos a qual compartilha-se em todos micros serviços e sistemas legados, essa biblioteca recebe os eventos e encaminha para fila, unificando assim toda a informação espalhada nos diversos sistemas em um único lugar para ser processado.

O funcionamento básico do evento consiste em executar comandos pré-definidos pelo aplicativo criado. Como exemplo pode-se citar os principais, como: envio de e-mail, *pusher*[3]. *Webhook*[4], Http assíncrono.

Como forma de exemplificar serão apresentados dois possíveis cenários e casos de uso.

a) Caso de uso 1

Um condomínio possui um aplicativo de integração que assina o envio de e-mail e pusher ao se cadastrar um morador novo.

Ao sistema legado processar o cadastro desse morador, será gerado um evento alertando esse cadastro com os dados respectivos e adicionado na fila, a fila por sua vez ira receber esse evento e armazená-lo.

Em outro micro serviço, o qual chama-se de *worker*[5], esse evento na fila será processado e o sistema ira ativar os comandos de e-mail e *pusher*, notificando a todos desse aplicativo que assinam esse evento.

b) Caso de uso 2

Um cliente integrador cria um aplicativo de integração no qual ele gostaria que ao se cadastrar um usuário a um condomínio que ele possua, possa se receber em seu sistema de portaria remota esse usuário, gerando uma credencial de acesso.

Ao sistema legado processar o cadastro desse morador, será gerado um evento alertando esse cadastro com os dados respectivos e adicionado na fila, a fila por sua vez irá receber esse evento e armazená-lo.

O *worker* irá processar esse evento e ativará a execução do comando de webhook no qual o cliente se cadastrou ao configurar o aplicativo. Ao se disparar esse *webhook* ele estará enviando o usuário para o sistema parceiro efetuando a integração de modo automático.

3.3.3 Devops

[3] pusher: recurso que envia notificação para o smartphone do usuário quando um evento ocorre na aplicação.

[4] webhook: método de ampliar ou alterar o comportamento de uma página da Web.

[5] worker: é um programa que é executado continuamente e existe com a finalidade de lidar com solicitações de serviço periódicas que um sistema de computador espera receber.

Encontra-se nessa parte do projeto algumas ferramentas essenciais para o funcionamento.

Supervisor para gerenciar as filas de Mensagens, nele está contido os *daemons*[6] responsáveis por executar os *workers* e filas respectivamente.

Também encontra-se configurações de ambiente como *dockers*[7] e *nginx*[8], toda a infraestrutura está montada em cima de *dockers* os quais utiliza-se em todos os sistemas a fim de facilitar o *deploy* e manutenção.

3.4 BANCO DE DADOS

Foi utilizado a governança descentralizada para reduzir a carga nos bancos de dados.

O banco de dados do sistema legado permaneceu original, apenas foi implementado um modelo de replicação de banco de dados de modo separado a escrita da leitura, podendo assim escalar o banco com quantas máquinas forem necessárias para execução do micro serviço.

Já o sistema de aplicativos, eventos e *webhooks* ganharam um novo banco de dados, separando todo funcionamento da api gateway da aplicação legada. Esse banco também possui modelo de leitura e replicação instantânea.

3.5 FUNCIONAMENTO DO PRODUTO FINAL

O sistema Winker, como demonstrado nos tópicos anteriores, ficou separado em duas frentes de processamento.

A primeira frente corresponde ao recebimento de dados, podendo essas requisições virem de clientes parceiros, usuários ou integrações. A segunda frente ficou responsável por executar em *back-end* todo processamento assíncrono, eventos, comandos.

---

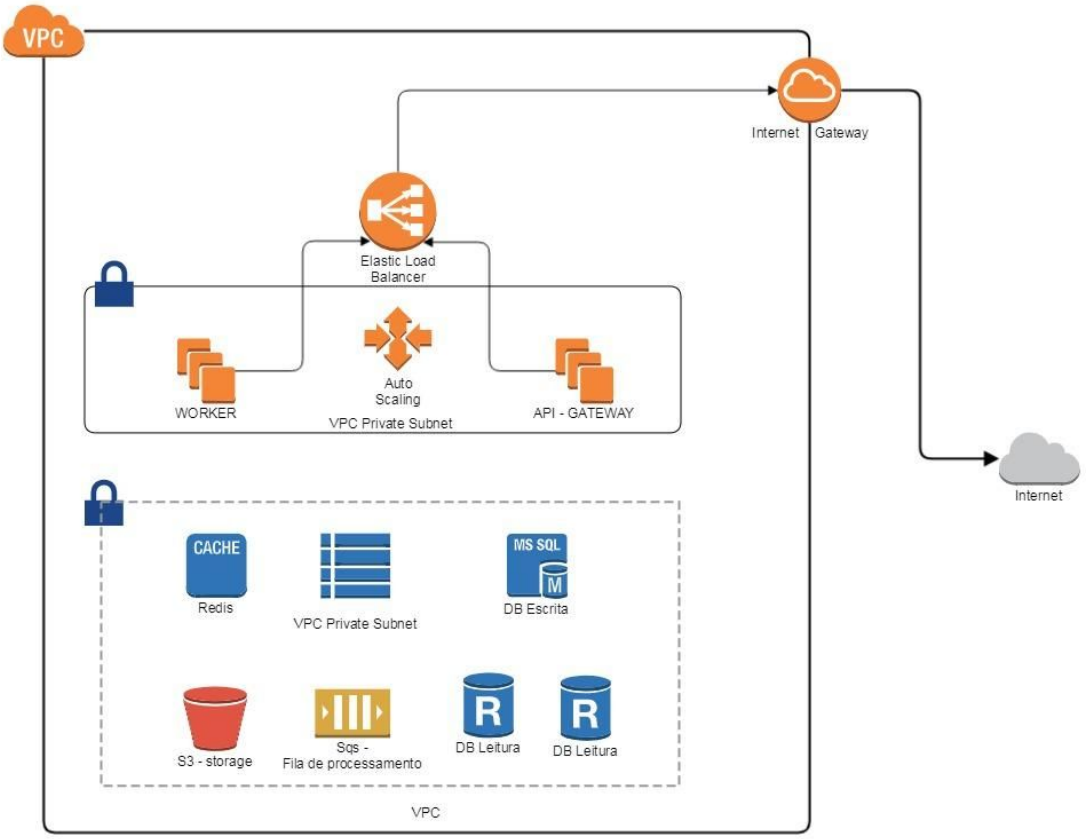
[6] daemons: é um programa de computador que executa como um processo em plano de fundo, em vez de estai r sob o controle direto de um usuário interativo.

[7] dockers: são plataformas de alto desempenho que facilitam a criação e administração de ambientes isolados.

[8] nginx: é um servidor HTTP e proxy reverso, bem como, um servidor para proxy de e-mail IMAP/POP3.

Na figura a seguir é apresentada a estrutura da API Gateway Winker.

Figura 14 - Estrutura API Gateway Winker



Fonte: O autor (2018)

Como é possível observar na figura, existem duas formas de entrada de requisições: uma de forma síncrona usando o API Gateway e outra de forma assíncrona por meio do *worker*. Além disso, em caso de muitas requisições em paralelo o sistema de *Elast Load Balancer* faz o *auto scaling* para manter a performance.

4 CONCLUSÃO

A busca por relevância e desempenho para o usuário final demandou à Winker S/A maior flexibilidade no processo de desenvolvimento de seu sistema principal que leva o mesmo nome.

A Winker avaliou diversos modelos de projetos disponíveis, selecionando a arquitetura de micro serviços com ênfase no padrão API gateway e utilizando métodos síncronos e assíncronos de comunicação.

As publicações dos autores Chris Richardson bem como Lewis e Fowler foram fundamentais para a o embasamento científico do estudo de caso na Winker, porém, observou-se ainda uma variedade pequena de autores que se aprofundaram nos conceitos aqui citados, impossibilitando uma maior riqueza de visões sobre tais conceitos. Diante dessa constatação e do sucesso da implementação dos micro serviços na Winker, o presidente da Winker autorizou a publicação deste estudo com vistas a subsidiar estudantes e profissionais da área nos desafios associados a transformação de estruturas monolíticas para micro serviços.

Ficou evidente que o processo de evolução de qualquer sistema demanda a evolução dos processos internos de desenvolvimento bem como a avaliação das novas tecnologias com o intuito de melhoria contínua, especialmente em se tratando de desempenho.

No papel de desenvolvedor na Winker e estudante da área de tecnologia, foi possível acompanhar e participar não somente dos aspectos técnicos do processo de construção da nova infraestrutura na Winker, mas também acompanhar o processo decisório. Ficou claro que decidir por mudar a infraestrutura base de um sistema é algo que carece de uma análise profunda e detalhada dos impactos, vantagens e desvantagens bem como do custo direto e indireto para fazê-lo.

Nessa linha, vejo como uma oportunidade de estudo futuro a evolução do estudo levantando custos bem como documentação do processo de mudança da infraestrutura.

REFERÊNCIAS

ALMEIDA, Adriano. **Arquitetura de microserviços ou monolítica?**. Disponível em <http://blog.caelum.com.br/arquitetura-de-microservicos-ou-monolitica/>. Acesso em: 13 nov. 2018.

JESUS, Vinícius Fernandes de. **Arquitetura de Software**: uma proposta para a primeira aplicação. 2013. 45 f. TCC (Graduação) - Curso de Ciência da Computação, Universidade Estadual de Londrina, Londrina, 2013.

LEWIS, James; FOWLER, Martin. **Microservices:** a definition of this new architectural term. 2014. Traduzido por Pedro Mendes. Disponível em: <https://martinfowler.com/articles/microservices.html>. Acesso em: 19 set. 2018.

NEWMAN, Sam. Building **Microservices**: Designing fine-grained systems. 1. ed. Califórnia: O'Reilly Media, 2015.

RICHARDS, M. **Software Architecture Patterns**. Sebastopol: O'Reilly Media, 2015.

RICHARDSON, Chris. **Microservices**: decomposing applications for deployability and scalability. 2014. Disponível em: <https://www.infoq.com/articles/microservices-intro#anch150992>. Acesso em: 19 set. 2018.

SANTOS, Lucas. **Microserviços**: dos grandes monólitos às pequenas rotas. 2017. Disponível em: <https://medium.com/trainingcenter/microservi%C3%A7os-dos-grandes-mon%C3%B3litos-%C3%A0s-pequenas-rotas-adb70303b6a3>. Acesso em: 19 set. 2018.

WOODS, 2015. Dan. **Building Microservices with Spring Boot**. Disponível em < https://www.infoq.com/articles/boot-microservices?utm_source=infoq&utm_campaign=user_page&utm_medium=link>. Acesso em:30 out.

AGRADECIMENTOS

Agradeço a minha família e meu orientador.

“O sucesso é um professor perverso. Ele seduz as pessoas inteligentes e as faz pensar que jamais vão cair” Bill Gates.