Pós-Graduação em Ciência da Computação

Bruno Chaves de Freitas

MODERNIZAÇÃO DE SISTEMAS LEGADOS PARA DISPONIBILIZAÇÃO EM DISPOSITIVOS MÓVEIS COM ARQUITETURA BASEADA EM *MICROSERVICES*

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
<www.cin.ufpe.br/~posgraduacao>

Recife
2017

Bruno Chaves de Freitas

MODERNIZAÇÃO DE SISTEMAS LEGADOS PARA DISPONIBILIZAÇÃO EM DISPOSITIVOS MÓVEIS COM ARQUITETURA BASEADA EM *MICROSERVICES*

Dissertação apresentada à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco - Cin/UFPE, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: *Roberto Souto Maior de Barros*

Recife
2017

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

F866m Freitas, Bruno Chaves de
Modernização de sistemas legados para disponibilização em dispositivos móveis com arquitetura baseada em *microservices* / Bruno Chaves de Freitas. – 2017.
100 f.: il., fig., tab.

Orientador: Roberto Souto Maior de Barros.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2017.
Inclui referências e apêndices.

1. Ciência da computação. 2. Dispositivos móveis. I. Barros, Roberto Souto Maior de (orientador). II. Título.

004 CDD (23. ed.) UFPE- MEI 2017-118

Bruno Chaves de Freitas

Modernização de Sistemas Legados para Disponibilização em Dispositivos Móveis com Arquitetura Baseada em microservices

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 23/02/2017.

BANCA EXAMINADORA

_____

Prof. Dr. Alexandre Marcos Lins de Vasconcelos
Centro de Informática / UFPE

_____

Prof. Dr. Ricardo André Cavalcante de Souza
Departamento de Estatística e Informática/UFRPE

_____

Prof. Dr. Roberto Souto Maior de Barros
Centro de Informática / UFPE
(**Orientador**)

AGRADECIMENTOS

Primeiramente agradeço a Deus, que me mostrou a sua existência e um caminho de luz para viver e guiar a minha família. Agradeço à minha esposa Elis, que se dedicou em diversos momentos a cuidar sozinha de nosso filho para que eu pudesse estudar e progredir com este trabalho. Agradeço a Tarcísio Coutinho, por todo o apoio prestado. Agradeço à minha sogra Mércia, que também se dedicou muito em cuidar de seu neto quando Elis e eu não podíamos fazê-lo. Agradeço ao meu filho Benício, pelas alegrias proporcionadas com as brincadeiras e seu lindo sorriso, em momentos de fuga do trabalho e estudo. Agradeço aos meus pais Ricardo e Telga, pela educação que me deram. Agradeço ao meu orientador Roberto, que foi paciente e incentivador. Agradeço a Mikko Kangasaho, um finlandês que generosamente cedeu sua dissertação sobre um tema correlato, cujo conteúdo contribuiu bastante com o meu trabalho.

"Não vos amoldeis às estruturas deste mundo,
mas transformai-vos pela renovação da mente,
a fim de distinguir qual é a vontade de Deus:
o que é bom, o que Lhe é agradável, o que é perfeito.
(Bíblia Sagrada, Romanos 12, 2)

RESUMO

O uso universal de dispositivos móveis computacionais, especialmente dos *smartphones*, é incontestável e um processo sem volta. Este fato impulsiona as organizações possuidoras de sistemas de informação a adaptá-los para um adequado acesso através deste veículo computacional, proporcionando uma boa experiência de uso além de aproveitar novas possibilidades inerentes a estes dispositivos. Os sistemas legados, no entanto, podem dificultar esta adaptação, seja por sua tecnologia, acoplamento de código ou arquitetura inapropriados, uma vez que a tecnologia à época de seu desenvolvimento ficou defasada com o passar do tempo, demandando uma modernização de sua arquitetura. Neste contexto, a arquitetura de *microservices* tem se destacado. Este trabalho propõe um processo de modernização de sistemas legados para uma arquitetura baseada em *microservices*, distribuindo o sistema em diversos serviços pequenos, independentes entre si, focados cada um em uma única tarefa e comunicando-se por mensagens. Esta distribuição e independência deixarão cada serviço livre para utilizar qualquer tecnologia, quebrando as amarras tecnológicas do sistema legado, além de facilitar futuras evoluções. Esta "quebra" do sistema, no entanto, pode ser demorada, em virtude da necessidade de entendimento das regras de negócio implementadas e dos refatoramentos necessários. Em virtude disto, para priorizar a disponibilização do acesso *mobile* a estes sistemas, o processo proposto prevê uma etapa intermediária de modernização utilizando a técnica de REST *Wrapping*.

Palavras-chave: *mobile*. dispositivos móveis. sistemas legados. *microservices*. REST *Wrapping*.

ABSTRACT

The universal use of mobile computing devices, especially smartphones, is undeniable and an irreversible process. This fact encourages the organizations using information systems to adapt them to provide an adequate access through this computational tool, providing a good experience of use besides taking advantage of new possibilities inherent to these devices. Legacy systems, however, can make this adaptation difficult, either because of its technology, inappropriate code coupling or architecture, since the technology at the time of its development lags behind with time, demanding the modernization of its architecture. In this context, the microservices architecture has been emerging. This work proposes a modernization process of legacy systems to a microservice-based architecture, distributing the system into several small independent services, each focused on a single task and communicating through messages. This distribution and independence will leave each service independent to use any technology, breaking the technological constraints of the legacy system in addition to facilitating future evolutions. However, this separation of the system, can be time-consuming because of the need to understand the business rules implemented and the necessary refactorings. In order to prioritize the availability of mobile access to these systems, the proposed process provides an intermediate step of modernization using the REST Wrapping technique.

Keywords: mobile. mobile devices. legacy systems. microservices. REST Wrapping.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A popularidade atual dos dispositivos móveis, principalmente dos *smartphones*, é um fato incontestável (WROBLEWSKI, 2011). Como consequência desta ubiquidade, da evolução destes dispositivos e da conveniência proporcionada por sua mobilidade, a forma de acessar a internet vem sendo modificada: o tradicional computador de mesa ou *notebook* estão dando lugar a um acesso mais dinâmico, na realização de tarefas do dia a dia, a qualquer momento, proporcionado por dispositivos que estão ao alcance de sua mão.

Apesar das limitações dos dispositivos móveis, tais como tamanho de tela e poder de processamento, a referida conveniência se sobressai, contribuindo para um uso cada vez mais universal destes dispositivos. Isto impulsiona a indústria e as organizações a criarem soluções de tecnologia de informação específicas para este novo mercado.

Contudo, existe ainda todo um arcabouço de sistemas de informação que foram projetados em época passada, em que disponibilizar sistemas acessados por computadores pessoais era suficiente. Estas soluções, chamadas de sistemas legados, continuam em funcionamento e são vitais para as organizações. Elas precisam ser modernizadas para que sejam acessíveis de forma adequada em dispositivos móveis. Esta tarefa não é fácil, e sua complexidade pode ser ainda maior a depender de quão defasada é a tecnologia utilizada pelo o sistema.

No contexto de modernização de sistemas legados, a migração para uma arquitetura de *microservices* tem se destacado (FOWLER; LEWIS, 2014). Nesta arquitetura, que é vista como uma evolução de SOA - *Service Oriented Architecture* (SHARMA, 2016), o sistema é composto por diversos serviços independentes entre si, inclusive em nível de infraestrutura operacional, responsáveis por tarefas únicas, comunicando-se por mensagens para atingir objetivos de negócio. O processo de migração não é rápido, e pode ser mais difícil ainda se o sistema legado não estiver organizado de acordo com boas práticas de programação e princípios de engenharia de software, principalmente a alta coesão e o baixo acoplamento.

Este trabalho propõe um processo genérico de alto nível para modernizar sistemas legados de forma incremental com o objetivo de integrá-los com dispositivos computacionais móveis por meio de uma arquitetura baseada em *microservices*. Contudo, em virtude do tempo necessário para migrar partes do sistema para a nova arquitetura, que demanda entendimento de código e regras de negócio implementadas, o processo faz uso de uma etapa

intermediária com a técnica de REST *Wrapping*[1], de maneira a priorizar a disponibilização do sistema em dispositivos móveis.

O referido processo é aplicado em um estudo de caso, que consiste no sistema de gestão acadêmica SIGA da UFPE (Universidade Federal de Pernambuco), feito na plataforma J2EE [2] com mais de 15 anos de uso, para criação de uma aplicação de acesso a informações de discentes de pós-graduação.

1.1 Caracterização do Problema

Conforme introduzido, o problema abordado por este trabalho diz respeito às dificuldades para modernizar sistemas legados para uma nova arquitetura, especificamente a de *microservices*, com objetivo de tornar este sistema acessível por dispositivos computacionais móveis, além de obter os benefícios inerentes à nova arquitetura. A seguinte pergunta de pesquisa foi identificada:

a) Como modernizar sistemas legados de maneira a possibilitar acesso de usuários por meio de dispositivos computacionais móveis frente às dificuldades impostas por sua tecnologia legada e sua criticidade para a organização, que demanda continuidade de funcionamento?

1.2 Justificativa

Os sistemas legados por sua natureza são críticos para as organizações, no sentido de que são essenciais para a execução dos processos de negócio dando o apoio da tecnologia da informação às tarefas e atividades cotidianas. A interrupção no funcionamento destes sistemas gera prejuízos para a organização (BRODIE; STONEBRAKER, 1995).

Sob outra perspectiva, estes mesmos sistemas passam a oferecer dificuldades de modificação e evolução para atender os novos requisitos de negócio que surgem com o decorrer do tempo, seja por sua tecnologia defasada ou por sua alta complexidade de modificação, que ocorre normalmente em virtude de um acúmulo do débito técnico e sintomas de envelhecimento decorrentes de manutenções emergenciais inapropriadas realizadas no passado (KANGASAHO, 2016).

O acesso a sistemas por meio de dispositivos computacionais móveis, principalmente dos celulares, é uma necessidade contemporânea para acompanhar a tendência dos usuários, fazendo com que as organizações realizem esforços de modernização para prover este meio

[1] REST (*REpresentational State Transfer*) *Wrapping* é um caso especial de SOA *Wrapping*, em que funcionalidades ou dados de um sistema legado são envolvidos e expostos por uma API (*Application Program Interface*) REST (KANGASAHO, 2016).

[2] Java 2 Plataform Enterprise Edition. https://pt.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition

de acesso a seus sistemas (WROBLEWSKI, 2011). Contudo, esta não é uma tarefa simples, porque dentre outras razões os sistemas precisam continuar funcionais suportando os processos organizacionais e muitas das regras de negócio não são documentadas, residindo no próprio código do sistema ou em usuários que o utilizam.

Contemporaneamente há uma evidência para a arquitetura de *microservices* no campo de modernização de sistemas legados (FOWLER; LEWIS, 2014), sendo promovida como boa alternativa para reduzir grandes sistemas em interações de pequenos serviços independentes, facilmente escalonáveis e totalmente desacoplados de tecnologia, de maneira a tornar os sistemas aptos até para futuras linguagens de programação e tecnologias correlatas. Porém, por ser uma arquitetura recente e ainda não padronizada oficialmente por alguma entidade, não há muitos trabalhos acadêmicos relacionados, razão pela qual é abordada por este trabalho.

1.3 Objetivo Geral

O objetivo geral desta pesquisa é especificar e avaliar um conjunto de atividades para modernização de sistemas legados possibilitando o acesso por dispositivos computacionais móveis, utilizando uma arquitetura baseada em *microservices*.

1.4 Objetivos Específicos

Para alcançar o objetivo geral, os seguintes objetivos específicos foram elencados:

1. Avaliar técnicas de modernização de sistemas legados;
2. Contextualizar a arquitetura de *microservices* em relação a SOA e serviços de *software*.
3. Analisar tipos de solução de *software* para dispositivos móveis;
4. Analisar atividades e boas práticas relacionadas à arquitetura de *microservices* na literatura, para formar um processo bem definido de modernização.

1.5 Metodologia

Em relação ao tipo de pesquisa realizada neste trabalho pode ser caracterizada: (1) quanto à finalidade, como Aplicada, pois tem por objetivo desenvolver um processo contribuindo para resolver um problema concreto e imediato da sociedade (APPOLINÁRIO, 2012); (2) quanto ao caráter, Exploratória, porque procura proporcionar maior familiaridade com o problema, uma vez que a temática não foi suficientemente discutida e não há uma

necessária adoção de hipótese que a norteie (PESSOA, 2005); (3) quanto aos procedimentos técnicos, Estudo de Caso, porque concentra a pesquisa numa situação ou local específico (PRODANOV; FREITAS, 2013; PESSOA, 2005).

Para identificar as ações necessárias para atingir o objetivo da pesquisa e buscar familiaridade com o tema, foram realizadas pesquisas bibliográficas em bibliotecas físicas, virtuais e em portais de artigos e periódicos pelos assuntos: dispositivos computacionais móveis, *mobile computing*, *mobile first*, sistemas legados, técnicas de modernização de sistemas, SOA (*Service Oriented Architecture*), *Web services*, serviços computacionais, *microservices*, REST (Representational State Transfer), APIs (Application Program Interfaces) e Segurança de Web APIs.

Com o objetivo de avaliar o conjunto de atividades de modernização especificado, este foi aplicado em um sistema legado de Gestão Acadêmica que atende a uma Universidade Federal, de maneira a avaliar a aplicabilidade em um caso real identificando a eficiência do processo. Para isso, fez-se necessário projetar e implementar as alterações arquiteturais no sistema e desenvolver um aplicativo para dispositivo móvel no sistema operacional *Android*, para atuar como consumidor dos dados do sistema modificado.

1.6 Escopo da Pesquisa

A definição de sistemas legados compreende uma grande quantidade de possibilidades de sistemas, desde os mais antigos, feitos em linguagens de programação que não são utilizadas há muito tempo; aos mais modernos, que até mesmo proveem meios de acesso *web* a seus usuários. O escopo utilizado neste trabalho para o qual o processo proposto se aplica é de sistemas monolíticos com arquitetura em camadas. Por sistema monolítico entende-se aquele composto por módulos que não são independentes da aplicação a que pertencem (DRAGONI et al., 2016).

Em virtude disto, o conjunto de atividades de modernização avaliado neste trabalho não se aplica a sistemas que já utilizam a consagrada arquitetura SOA (*Service Oriented Architecture*), que já possibilitam a integração com dispositivos computacionais móveis e outros sistemas.

Apesar de sistemas legados monolíticos apresentarem normalmente características similares, a pesquisa utilizou apenas um estudo de caso, o que limita as generalizações realizadas. Contudo, conforme a literatura estudada, as dificuldades e problemas encontrados no estudo de caso são de fato comuns e compartilhados com a maioria dos sistemas legados, fazendo com que os aspectos tratados neste trabalho possam ser aplicados a outros sistemas.

Há de se observar também que apenas uma parte do sistema do estudo de caso

será modernizada, e não todo o sistema, uma vez que este projeto não recebe apoio de uma equipe da organização que autorizou o uso do sistema, tratando-se de uma iniciativa individual. A modernização completa demandaria muito tempo e uma equipe com várias pessoas envolvidas.

1.7 Estrutura da dissertação

Este trabalho está organizado da seguinte maneira:

- Capítulo 1 - Introdução do trabalho, contendo a caracterização do problema e os objetivos.
- Capítulo 2 - Referencial teórico, abordando os fundamentos sobre dispositivos computacionais móveis, sistemas legados, computação orientada a serviço, *microservices* e REST *Web APIs*. Inclui também os trabalhos relacionados.
- Capítulo 3 - Abordagem proposta. Apresenta uma visão geral do processo proposto e detalha as suas etapas.
- Capítulo 4 - Estudo de caso. Aplica o processo a um sistema de gestão acadêmica com mais de 15 anos de uso.
- Capítulo 5 - Conclusões. Contempla considerações finais, limitações do trabalho e possibilidades de trabalhos futuros.

2 REFERENCIAL TEÓRICO

2.1 Dispositivos Computacionais Móveis

É notória a adoção universal de dispositivos móveis de telefonia, ou telefones celulares, pela população mundial (WROBLEWSKI, 2011). Com o avanço da tecnologia, os celulares expandiram suas funcionalidades para muito além de realizar e receber ligações telefônicas; tornaram-se verdadeiros dispositivos computacionais oferecendo funcionalidades aos usuários que normalmente eram disponibilizadas pelos tradicionais computadores pessoais. A ideia inicial associada ao termo "mobilidade" remete à imagem de uma pessoa em movimento, portando seu telefone celular que lhe proporciona todo um conjunto de funcionalidades, tais como ligações de telefone, navegação na internet, câmera, GPS, e-mail, dentre outros, aliadas a fortes mecanismos de segurança (DELGADO, 2011).

Desta maneira, tarefas que os usuários anteriormente realizavam sentados à mesa em frente a um computador passaram a ter a dinamicidade proporcionada por um dispositivo portátil. Estes celulares passaram a se chamar smartphones, e em 2012 já havia previsões de que suas vendas ultrapassariam as vendas de computadores pessoais (WROBLEWSKI, 2011).

Um dispositivo móvel pode ser entendido como um dispositivo que os usuários carregam consigo a qualquer tempo, podendo ser utilizado de forma relativamente instantânea, possuindo uma conexão com uma rede. Os smartphones e tablets atendem a esta definição (ESPOSITO, 2012). Para Lee, Schneider e Schell (2004), mobilidade no contexto da computação se refere ao uso de poderosos dispositivos portáteis e funcionais que provêm a habilidade de executar um conjunto de funções de aplicativos, capazes de conectar a outros usuários, aplicações ou sistemas para obter ou prover dados. Segundo os autores, os dispositivos móveis possuem as seguintes características:

- Portabilidade - capacidade de ser facilmente carregado pelo usuário;
- Usabilidade - pode ser utilizado por diferentes tipos de pessoas em diferentes ambientes.
- Funcionalidade - pode servir a múltiplos propósitos.
- Conectividade - a capacidade de se conectar a outras pessoas ou sistemas para transmitir e receber informações.

Esposito (2012) considera que o sucesso obtido pelo dispositivo iPhone da Apple em 2007 desencadeou uma nova era de desenvolvimento de software, um novo paradigma, em que o foco de desenvolvimento são os dispositivos móveis. Ainda de acordo com o autor, as previsões de que isso aconteceria remontam ao ano de 1999, sendo a demora para concretização justificada pelo fato de que os softwares para celulares ainda não haviam alcançado uma massa crítica de usuários. Esta massa crítica foi alcançada pelo dispositivo da Apple, reorientando o mercado de desenvolvimento de software.

2.1.1 Tipos de Solução *Mobile*

Para Esposito (2012) uma solução *mobile* não diz respeito apenas à criação de um aplicativo para celular, mas sim uma combinação de *websites* clássicos, *websites* específicos para dispositivos móveis, ou *m-sites*, e aplicativos para plataformas específicas de dispositivos móveis. Contudo, dada a evolução constante da indústria de dispositivos móveis, esta definição pode mudar em um curto espaço de tempo. O autor defende que uma organização que pretende desenvolver soluções móveis deve estabelecer um planejamento estratégico, identificando o futuro de seu negócio frente a alguns axiomas mobile. Estes axiomas devem ser bem entendidos antes da elaboração do planejamento:

- Prover seus serviços através de múltiplos canais;
- Buscar por novas oportunidades e novas maneiras de prover seus serviços;
- Ter como objetivo tornar a vida de seu cliente mais fácil.

Com os axiomas em consideração, a organização deve se preocupar em alcançar os clientes *mobile*. Para isso, dois questionamentos devem ser realizados:

- Quais dispositivos móveis os clientes estão usando?
- Como tornar a aplicação da empresa disponível em todos eles?

Para Wisniewski (2011) a escolha da estratégia e tipo de solução *mobile* que será utilizada deve começar pela análise dos usuários. O autor sugere que seja realizada uma pesquisa para buscar as necessidades e desejos dos usuários, além de informações sobre os tipos de dispositivos móveis que eles usam.

Quando há o objetivo de alcançar a maior audiência possível para uma solução mobile, investir no desenvolvimento de um *website* otimizado para dispositivos móveis, ou *m-site*, pode ser uma alternativa indicada. Contudo, esta não é uma solução livre de defeitos e problemas, em virtude da variedade de dispositivos móveis e browsers utilizados. Para contornar esta questão, Esposito (2012), sugere o que chama de *multiserving* para

tornar um *m-site* compatível com o máximo de dispositivos, que consiste em três pontos basicamente:

- Agrupar os dispositivos em classes, baseadas em suas capacidades;
- Construir uma versão do site para cada classe de dispositivos;
- Definir uma estratégia para servir o site correto para cada dispositivo.

Para apoiar o *multiservi*ng, o autor ressalta a importância da iniciativa WURFL – *Wireless Universal Resource File*, que provê uma base de dados com informações detalhadas de diversos dispositivos móveis e navegadores nativos utilizados.

Os *m-sites* representam uma solução distinta de aplicações nativas. É necessário entender as vantagens e desvantagens de cada uma das soluções para que, aplicando ao contexto da organização, seja possível optar por uma das soluções ou um misto delas. Esposito (2012) sugere que a organização deve focar nos produtos, serviços e audiência da solução, para em seguida detalhar os objetivos, orçamento, recursos e tempo. Com isto a escolha do tipo de solução é facilitada.

De forma geral, os tipos de solução mobile são classificados em aplicações nativas, mobile websites ou m-sites e soluções híbridas (WISNIEWSKI, 2011).

Aplicações Nativas

As aplicações nativas estão vinculadas ao sistema operacional dos dispositivos móveis a que pertencem, isto implica que estão sujeitas às limitações do dispositivo bem como podem usufruir dos recursos disponibilizados pelos dispositivos. Esposito (2012) elenca algumas das principais características das aplicações nativas:

- Maior possibilidade de uma boa integração com o dispositivo, uso do hardware e serviços nativos;
- Menor influência de latência de rede;
- Possibilidade de funcionamento sem rede;
- Não necessidade de utilizar URLs;
- Experiência de usuário nativa do dispositivo, que promove maior responsividade da aplicação;
- Usuários fazem o download de aplicações nativas como um pacote único com uma única requisição;

- Uma aplicação nativa precisa ser criada para cada plataforma mobile que se pretende suportar.

Alguns exemplos de recursos disponíveis para as aplicações nativas são câmera, notificações ao usuário, armazenamento local, localização GPS, telefonia, contatos e mensagens SMS. Alguns destes recursos podem ser utilizados de maneira diferente por m-sites que utilizam HTML5.

Para desenvolver aplicações nativas para dispositivos móveis normalmente é necessário utilizar kits de desenvolvimento ou SDKs providos pelos fabricantes dos dispositivos. Wisniewski (2011) ressalta que aplicações nativas levam enorme vantagem em relação aos *m-sites* no que diz respeito à experiência de usuário: elas provêm acesso com um clique, por meio de ícones disponibilizados no aparelho, enquanto que os *m-sites* dependem de diversas ações do usuário. Outras vantagens apontadas pelo autor são:

- Aplicações nativas (*apps*) são muito mais responsivas, uma vez que os dados utilizados fazem parte do próprio *app*, proporcionando menos retardo e latência do que os *m-sites*.
- *apps* são encontrados pelos usuários normalmente por lojas online dos fabricantes, como *App Store* da Apple[1] ou *Play Store* da Google[2].

O argumento mais favorável dos *apps* é o fato de que podem ser utilizados *offline*, sem acesso à rede. A principal desvantagem apontada por Wisniewski (2011) é que são específicos por plataforma. A não ser que os usuários da aplicação sejam muito homogêneos no tocante aos tipos de dispositivos utilizados, será necessário desenvolver a solução para duas plataformas ao menos. Mesmo em uma única plataforma, deve-se considerar as diferenças entre *smartphones* e *tablets*, essencialmente o tamanho da tela, demandando ajustes da solução.

Mobile Sites

Os *m-sites*, como dito anteriormente, são *websites* otimizados para dispositivos móveis, considerando essencialmente suas características de tela e interação do usuário. Um *website* comum pode ser visualizado por um dispositivo móvel, contudo em virtude do tamanho da tela a experiência do usuário não será muito agradável. O usuário terá que realizar ampliações para visualizar os trechos desejados do site, o que pode ser incômodo. Esposito (2012) cita algumas das principais características dos *m-sites*:

[1] https://support.apple.com/pt-br/HT204266
[2] https://play.google.com/store

- Não podem acessar recursos nativos do dispositivo como acelerômetro, câmera, áudio e retorno tátil;
- Sujeitos à latência de rede em virtude da comunicação com o servidor do site;
- Podem não estar disponíveis sem rede;
- Acessíveis apenas por digitação de URL ou utilização de atalhos e registros de sites favoritos;
- Baseados em HTML;
- Sujeitos às capacidades de cache e renderização dos *browsers*;
- São inerentemente multi-plataforma;
- Oferecem aproximadamente as mesmas facilidades de mecanismos de busca dos *websites* convencionais.

Utilizar *m-sites* como solução *mobile* implica que a equipe de desenvolvimento não precisará ter conhecimentos de plataformas específicas, uma vez que é um tipo de solução *server-side*. Contudo, além das desvantagens já citadas, o *m-site* muito provavelmente terá que possuir interfaces apropriadas para diferentes classes de dispositivos móveis em virtude da fragmentação de capacidades destes dispositivos (ESPOSITO, 2012). Para atender a esta dinamicidade de tamanhos de telas de dispositivos, os *m-sites* devem ser desenvolvidos considerando os princípios de *design web* responsivo, cujo principal objetivo é manter o visual de uma página agradável independente do tamanho da tela do dispositivo (BOHIUM, 2013).

Wisniewski (2011) considera como principais argumentos em favor de *m-sites*: 1. A possibilidade de desenvolver uma única vez com linguagens em que a equipe de desenvolvimento é familiar; 2. Usuários de qualquer plataforma terão acesso ao conteúdo, desde que tenham um *browser* e acesso à internet. Como desvantagem, o autor ressalta a fraca usabilidade e experiência de usuário: os usuários têm que abrir um navegador, digitar um endereço, esperar para carregar e em seguida realizar ações.

Em resumo, a escolha do tipo de solução que será utilizada depende dos objetivos organizacionais. Deve-se ponderar os objetivos frente às características de cada tipo de solução. Wroblewski (2011) entende que há razões para utilizar os dois tipos de solução.

Solução Híbrida

Soluções híbridas ou *apps* híbridos são um misto de aplicação nativa e *m-sites*: são instalados no dispositivo como um *app*, mas não foram desenvolvidos com a linguagem específica da plataforma, e sim com linguagens *web* tradicionais de *websites* como HTML,

CSS e Javascript, empacotados por um aplicativo nativo específico de uma ou mais plataformas (WISNIEWSKI, 2011). Existem ferramentas apropriadas para criar estas soluções híbridas, tais como PhoneGap[3] e Titanium[4].

Wisniewski (2011) elenca como vantagem das soluções híbridas a persistência de um *app* nativo, em que é disponibilizado para o usuário um ícone como qualquer outro *app*; além da facilidade de desenvolvimento em virtude de um único conjunto de linguagens utilizado para desenvolver *websites*. Como desvantagem, o autor cita o fato de continuar sendo uma solução específica por plataforma. As ferramentas utilizadas precisam dar suporte para distribuir o aplicativo híbrido nas plataformas desejadas pela organização.

2.1.2 Implicações arquiteturais

Além das diferenças já discutidas entre o acesso à internet por parte de dispositivos móveis e computadores pessoais, relativas ao contexto de utilização, cabe ressaltar também que, durante a execução de uma tarefa ou ação utilizando um dispositivo móvel na internet, o usuário utiliza normalmente uma sessão com tempo médio muito inferior em relação ao tempo médio de usuários que utilizam computadores pessoais tradicionais, além de fazer uso apenas de um pequeno conjunto de informações ou dados relacionados estritamente a sua tarefa (ZHANG et al., 2015).

Em conjunto com estas características, as limitações naturais dos dispositivos móveis, principalmente no tocante ao armazenamento de dados e instabilidade da qualidade de conexão à internet, impulsionam para que as soluções voltadas aos dispositivos móveis sejam de certa forma mais leves, consumindo menos dados do que soluções voltadas a computadores tradicionais. De acordo com Zhang et al. (2015), acessar sistemas legados a partir de dispositivos móveis normalmente envolve a criação de uma camada de integração para fins de provimento de segurança e transformação dos dados do sistema legado, reduzindo os dados para se adequar às necessidades das soluções para dispositivos móveis. Ainda segundo o autor, esta integração é tipicamente realizada com RESTful *web services*[5] para comunicação e JSON[6] como formato de dados, por ser mais leve do que outras alternativas como o XML.

Desta forma, sistemas de informação antigos que não estão devidamente habilitados ao acesso por meio de dispositivos computacionais móveis podem ser modernizados para prover informações também por meio deste canal.

[3] http://phonegap.com/
[4] http://www.appcelerator.com/
[5] http://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html
[6] http://www.json.org/json-pt.html

2.2 Sistemas legados

O termo "legado" comumente remete ao que é passado de uma geração para outra. Em geral, entende-se por sistemas legados aqueles que foram desenvolvidos há muito tempo com uso de tecnologias e metodologias defasadas, que são difíceis de modificar ou evoluir atualmente. Brodie e Stonebraker (1995) os definem como "qualquer sistema de informação que resiste significativamente a mudanças e evoluções para atender novas e constantes mudanças nos requisitos de negócio". Os autores ressaltam que estes sistemas são críticos para as organizações, pois precisam estar operacionais o tempo todo, o que dificulta esforços de migração para outros sistemas. Listam ainda algumas características comuns destes sistemas:

- Grandes: possuem normalmente milhões de linhas de código;
- Antigos: com mais de uma década de uso;
- Utilizam linguagens de programação legadas;
- Utilizam bancos de dados legados;
- Autônomos: operam de forma independente, com pouca ou nenhuma interface com outros sistemas.

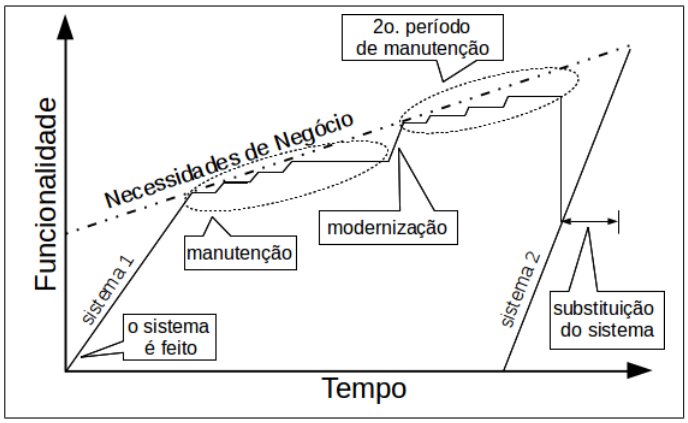
Fatores como a tecnologia defasada, falta de documentação, dificuldade de entendimento da implementação e estrutura degradada do sistema dificultam a manutenção e evolução dos sistemas legados (NILSSON, 2015; KANGASAHO, 2016). Sobre a degradação do sistema com o passar do tempo, Kangasaho (2016) ressalta que pode ser resultado de três fatores relacionados: (1) a segunda lei de Lehman[7], em que a complexidade do sistema cresce à medida que ele é evoluído; (2) o débito técnico, entendido como a decadência de qualidade das funcionalidades do sistema ou interação entre elas em virtude do atendimento a um nível mínimo de satisfação por seus usuários (STERLING; BARTON, 2010); e (3) sintomas de envelhecimento, relacionados à poluição ocasionada por código desnecessário, conhecimento de negócio embutido em código e não em documentação, nomenclatura inadequada de componentes, acoplamento indevido de código e violações a arquiteturas em camadas (VISAGGIO, 2001).

A manutenção dos sistemas de informação é uma das fases de seu ciclo de vida, que contempla ainda a modernização e a substituição, conforme classificação de Comella-Dorda et al. (2000). A manutenção consiste nas pequenas mudanças que ocorrem de forma incremental com o passar do tempo, tais como correções de erro ou melhorias. A norma internacional ISO14764[8] classifica a manutenção em quatro tipos: adaptativa, corretiva,

[7] http://users.ece.utexas.edu/ perry/work/papers/feast1.pdf
[8] https://www.iso.org/standard/39064.html

preventiva e perfectiva. A modernização por sua vez é mais extensa do que a manutenção, envolvendo normalmente uma reestruturação do sistema. Por fim, a substituição consiste na retirada do sistema para a utilização de um novo. A Figura 1 ilustra estas fases no decorrer do tempo frente às demandas de negócio.

Figura 1 – Ciclo de vida dos sistemas de informação.



Fonte: Adaptado de Comella-Dorda et al. (2000)

2.2.1 Técnicas de Modernização

Para Weiderman et al. (1997) a modernização pode ser entendida como uma evolução do sistema, considerando que é uma mudança de nível mais alto, tornando o sistema qualitativamente mais fácil de manter e acompanhar a demanda por novos requisitos de negócio. Os referidos autores classificam as atividades de modernização com base no nível de conhecimento do sistema legado exigido: *top-down* ou *black-box*, e *bottom-up* ou *white-box*. No primeiro tipo, não é necessário ter o conhecimento das regras de negócio implementadas ou dos detalhes técnicos de implementação, pois o sistema não será modificado neste aspecto, apenas receberá uma nova interface ou um encapsulamento para interagir com outros sistemas ou plataformas. No tipo *white-box*, no entanto, o conhecimento das regras de negócio é necessário, usando para isso técnicas como análise de código ou entrevistas, por exemplo.

Bisbal et al. (1999), por sua vez, com base em um conjunto de soluções propostas para modernizar os sistemas legados, sumariza as técnicas de modernização em três categorias distintas:

- Redesenvolvimento: o sistema é totalmente refeito, desprendendo-se de sua tecnologia defasada. Esta categoria é a mais custosa e arriscada para a organização;
- *Wrapping*: provê uma nova interface para o sistema ou parte dele, possibilitando novas integrações com outros sistemas ou plataformas;

- Migração: a mudança incremental do sistema de um ambiente ou plataforma para outro. Esta categoria é mais complexa do que o *wrapping*, mas provê maiores benefícios de longo prazo, como a manutenção do sistema, por exemplo.

Realizando um cruzamento entre as classificações de Bisbal et al. (1999) e Weiderman et al. (1997), pode-se observar que o redesenvolvimento é *white-box*, o encapsulamento (*wrapping*) é *black-box*, e a migração é mista, uma vez que para migrar uma parte do sistema, será necessário o entendimento de suas regras de negócio e implementação; por outro lado, o que foi migrado precisará interagir com o sistema legado por meio do *wrapping*.

Kangasaho (2016) define REST *wrapping* como um caso especial de SOA *wrapping*, em que funcionalidades ou dados de um sistema legado são envolvidos e expostos por uma API REST. Esta tecnologia que envolve o sistema legado tem como responsabilidade receber as requisições dos clientes, comunicar-se com o sistema legado para obter os dados e encaminhá-los para o cliente, utilizando serviços de software.

2.3 Serviços de Software

De acordo com Di Nitto et al. (2008), os sistemas que apoiam os negócios de organizações e empresas têm evoluído de uma forma centralizada e monolítica para uma abordagem mais distribuída, para atender a uma demanda cada vez maior por flexibilidade, dinamismo e adaptatividade. Esta evolução acompanha os negócios das próprias organizações, que passaram de estruturas hierárquicas estáveis, monolíticas e centralizadas para entidades dinamicamente federadas, formadas pela integração e composição de serviços individualmente oferecidos pela organização. A alta dinamicidade destas composições de serviços visa suportar de forma flexível e rápida a evolução dos requisitos e objetivos de negócio (Di Nitto et al., 2008). Dragoni et al. (2016) definem uma aplicação monolítica como um sistema composto de módulos que não são independentes da aplicação a que pertencem.

Para atender a estes requisitos de distribuição, flexibilidade e adaptabilidade surgiu uma abstração chamada de serviço de software, diferenciando-se de *softwares* tradicionais porque representam a funcionalidade oferecida por eles, permitindo a agregação individual de serviços em composições. Esta nova abordagem possibilitou o surgimento das aplicações baseadas em serviços (Di Nitto et al., 2008). A Figura 2 ilustra um exemplo de representação de um serviço e suas capacidades.

Papazoglou et al. (2008) definem serviços como componentes abertos e autodescritivos que suportam composições de sistemas distribuídos de forma rápida e com baixo custo. De acordo com Hwang, Fox e Dongarra (2013), um sistema distribuído é formado por computadores autônomos, cada um com sua própria memória, utilizando comunicação

Figura 2 – Representação de um serviço e suas capacidades

Fonte: adaptado de Erl (2005)

via rede para intercâmbio de dados por um mecanismo de passagem de mensagem.

Neste contexto, os serviços são os elementos fundamentais do paradigma de Computação Orientada a Serviço para desenvolvimento de aplicações (PAPAZOGLOU et al., 2008). Ramanathan e Raja (2013) utilizam como sinônimo do termo “Computação Orientada a Serviço” o termo "*Service-Driven* Computing".

2.3.1 Computação Orientada a Serviço

Computação orientada a serviço, como um paradigma para desenvolvimento de aplicações, é também um termo geral que representa uma nova geração de plataforma de computação distribuída. A orientação a serviço preconiza que unidades lógicas de solução sejam individualmente modeladas para que, coletivamente e repetidamente, sejam utilizadas dando apoio à realização de objetivos estratégicos específicos (ERL, 2005).

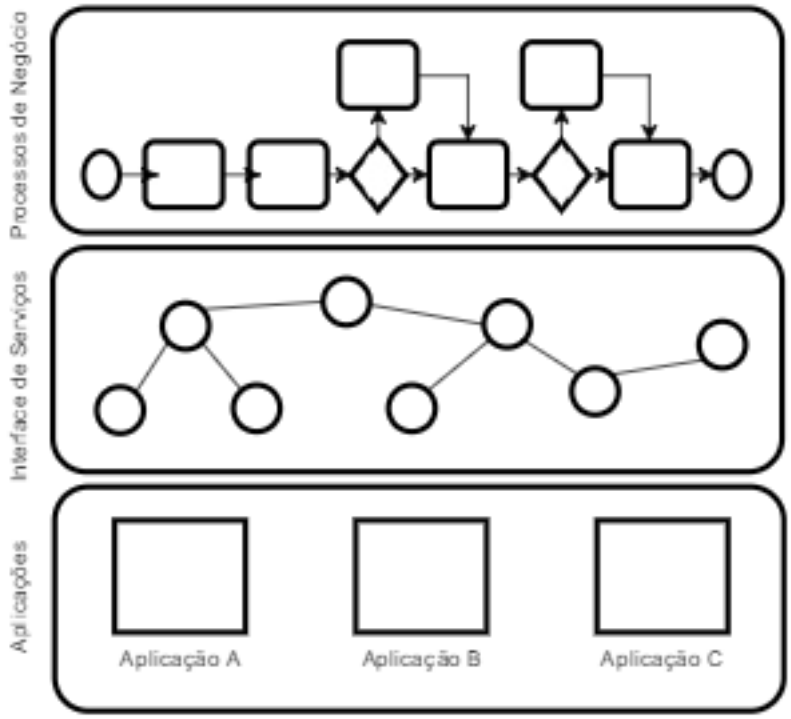
De acordo com Erl (2005), a orientação a serviço tem suas raízes na teoria da separação de interesses, em que existe a noção de que é benéfico quebrar um problema grande em partes pequenas, de maneira que a solução para o problema maior é alcançada pela solução de suas partes menores. O autor elenca ainda um conjunto comum de princípios associados à orientação a serviço:

- Serviços são reusáveis – são projetados para apoiar reuso em potencial;
- Serviços compartilham um contrato formal – descreve cada serviço e define os termos utilizados para troca de informações;
- Serviços são fracamente acoplados – são projetados para interagir sem dependência de tecnologia entre serviços;
- Serviços abstraem a lógica utilizada – a única parte visível do serviço é o contrato formal, a sua lógica é invisível para os requisitantes;

- Serviços podem formar composições – serviços podem ser compostos de outros serviços, promovendo reusabilidade e camadas de abstração;
- Serviços são autônomos – a lógica de um serviço reside em uma fronteira explícita. O serviço tem controle desta fronteira e não depende de outros serviços para exercer este controle;
- Serviços não têm estado – não devem gerenciar informação de estado porque dificultaria o princípio de fraco acoplamento tecnológico;
- Serviços podem ser descobertos – devem permitir que suas descrições sejam descobertas e entendidas por humanos e máquinas que podem fazer uso de sua lógica.

Para Erl (2005), a lógica organizacional é dividida nos domínios de lógica de negócio e de lógica de aplicação. A computação orientada a serviços introduz uma camada de interface de serviços, que representa a lógica de negócio e abstrai a lógica de aplicação, conforme exposto na Figura 3. Papazoglou et al. (2008) defende que a abordagem orientada a serviços tem como grande vantagem a sua independência de linguagens de programação específicas e de sistemas operacionais, permitindo que as organizações foquem em suas principais competências.

Figura 3 – Arquitetura básica de orientação a serviço



Fonte: Adaptado de Erl (2005)

Para aplicar a computação orientada a serviços, a utilização conjunta das especificações de *Web Services* e dos princípios de SOA (*Service Oriented Architecture*) é vista como uma das técnicas contemporâneas de modernização de sistemas (BAGHDADI; AL-BULUSHI, 2015).

2.3.2 Web services

Erl et al. (2014) defendem que o paradigma de programação orientada a objetos teve um papel crucial no surgimento da orientação a serviço, e que as limitações na execução da lógica de negócio em sistemas dinamicamente ligados a ambientes legados proporcionaram o surgimento dos *web services*. A definição formal dada pela W3C – *World Wide Web Consortium* – para o termo *web service*[9] é:

> *"Um Web service é um sistema de software projetado para suportar a interação entre máquinas interoperável em uma rede. Ele tem uma interface descrita em um formato processável por máquina (especificamente WSDL). Outros sistemas interagem com Web services de uma forma prescrita por sua descrição usando mensagens SOAP, normalmente transmitidas com uma serialização XML em conjunto com outros padrões web relacionados."*

Esta definição ressalta a função principal genérica dos web services como meio de suporte à interação entre máquinas em uma rede, mas associa o termo a uma de suas opções de implementação utilizando o protocolo SOAP – *Simple Object Access Protocol* e a descrição com WSDL – *Web Services Description Language*. Haas (2005) reconhece que existem diferentes definições, mas coloca que de maneira geral *web services* são serviços disponibilizados na *web*. Contudo, o termo *web service* comumente é usado para fazer referência ao que preconiza a definição formal, ou seja, SOAP-*based Web services*.

Em resumo, o termo "web services" tem dois significados distintos: um mais genérico e abrangente, relacionado com qualquer serviço de software implementado e disponibilizado pela web; e outro mais específico, relacionado com a definição formal e o conjunto de especificações correlatas. Bloomberg (2013) ressalta que *Web Services*, com iniciais maiúsculas, não representam uma abordagem arquitetural, e sim um conjunto de padrões instruindo sobre como definir serviços (*WSDL - Web Services Description Language*), realizar comunicação com serviços (*SOAP - Simple Object Access Protocol*), descobrir sua disponibilidade (*UDDI - Universal Description, Discovery and Integration*), e realizar outras tarefas como composição, segurança e gerenciamento de serviços.

[9] https://www.w3.org/TR/ws-arch/#whatis

2.3.3 SOA - Service Oriented Architecture

De acordo com a IEEE 42010 [10], a arquitetura de um sistema é definida por conceitos ou propriedades fundamentais de um sistema em um ambiente formado por seus elementos, relações e por seus princípios de projeto e evolução. O termo SOA, acrônimo para Service Oriented Architecture, surgiu em 1996, segundo Erl et al. (2014), quando um analista da Gartner chamado Yefim V. Natis definiu a arquitetura orientada a serviço como:

> *"Uma arquitetura que se inicia com a definição de uma interface e constrói uma topologia inteira de aplicação como uma topologia de interfaces, implementações de interfaces e chamadas de interfaces."*

No ano de 2000, junto com a adoção das tecnologias de *Web services* pelas empresas, a arquitetura SOA passou a ter ampla aceitação. Esta adoção conjunta de *Web services* e SOA levou a uma confusão entre os termos, de maneira que a simples adoção da referida tecnologia por uma organização era confundida com implementação de SOA (ERL et al., 2014). Contudo, enquanto *Web service* remete ao software em si relacionado a um serviço, conforme definições da seção anterior, SOA remete à arquitetura de um software utilizando serviços.

Rotem-Gal-Oz (2012) define SOA como um estilo arquitetural para construir sistemas baseados em interações de componentes fracamente acoplados, grosseiros – no sentido de que devem contemplar uma função de negócio que pode envolver outras funções, e autônomos, chamados de serviços. Ainda de acordo com a definição, cada serviço deve expor processos e comportamentos através de contratos, que são compostos por mensagens em endereços passíveis de descoberta chamados *endpoints*. O comportamento do serviço é dirigido por políticas externas ao serviço; os contratos e mensagens são usados por componentes externos chamados de *service consumers*.

De maneira mais abstrata, Erl (2005) explica que uma arquitetura orientada a serviço pode ser entendida como um ambiente padronizado de acordo com os princípios da orientação a serviço, em que um processo orientado a serviços pode executar. Em 2009, com o objetivo de clarificar e dar direção ao entendimento do termo SOA, foi publicado o Manifesto SOA, que dentre outros aspectos contempla uma definição geral e simples para o termo:

> *"Arquitetura Orientada a Serviço (SOA) é um tipo de arquitetura que resulta da aplicação de orientação a serviço."*

[10] http://ieeexplore.ieee.org/document/4278472/

2.4 Microservices

As arquiteturas baseadas em SOA podem ser chamadas de “primeira geração” de arquiteturas baseadas em serviços. Elas possuíam requisitos como contrato de descoberta do serviço que aumentavam a complexidade da construção dos serviços, o que prejudicou a adoção do modelo proposto por SOA (DRAGONI et al., 2016).

Em meados de 2011, em um *workshop* de arquitetos de software, surgiu o termo *microservice* para nomear um arquitetura de sistemas que estava sendo explorada pelos participantes (FOWLER; LEWIS, 2014). Esta nova abordagem emergiu de possibilidades proporcionadas por novas tecnologias e práticas adotadas no mundo real, tais como virtualização, integração contínua, domain-drive-design, automação de infraestrutura e utilização de grupos pequenos de desenvolvimento (NEWMAN, 2015).

De acordo com Rv (2016), a arquitetura baseada em *microservices* foi inspirada em um padrão arquitetural chamado de Arquitetura Hexagonal, ou *Ports and Adapters Pattern*, proposto por Alistair Cockburn. De forma resumida, este padrão preconiza que a aplicação deve se concentrar em sua função principal, delegando mecanismos de transporte e comunicação para "adaptadores", permitindo que a aplicação seja utilizada igualmente por pessoas, programas, testes automáticos ou *scripts* de execução em lote, isolando o desenvolvimento e os testes da aplicação de seus dispositivos auxiliares de execução e banco de dados, por exemplo. Newman (2015) também cita a arquitetura hexagonal como uma das tecnologias que, em conjunto com outras, proporcionou o surgimento dos *microservices*, reconhecendo sua importância como guia para evitar arquiteturas em camada, evitando a ocultação de partes da lógica de negócio.

As principais diferenças entre os modelos de serviços propostos pela arquitetura de *microservices* e SOA são a granularidade do serviço e sua independência. Os *microservices* têm como objetivo realizar bem uma única tarefa ou capacidade de negócio. Para isto, contam com total independência em relação a outros serviços, no sentido de que não há acoplamento tecnológico: podem usar a linguagem ou banco de dados mais apropriados para seu fim, por exemplo.

Gonzalez (2016) define *microservices* como "pequenos componentes de software que são especializados em uma tarefa e trabalham juntos para alcançar uma tarefa de nível mais alto". O autor defende que a especialização é sempre chave para melhorar a eficiência, ressaltando que fazer uma única coisa de maneira correta é um dos mantras do desenvolvimento de software. Para Dragoni et al. (2016), um *microservice* é um processo independente mínimo interagindo via mensagens, enquanto uma arquitetura de *microservice* é uma aplicação distribuída em que todos os seus módulos são *microservices*. A comunicação entre eles ocorre por meio de Web APIs - *Application Program Interfaces*, de linguagem neutra.

Apesar de compartilhar a característica essencial de foco em orientação a serviço com SOA, a arquitetura de *microservices*, pode ser vista como uma evolução de SOA. O surgimento de SOA facilitou a integração entre aplicações monolíticas por meio de APIs, normalmente utilizando o protocolo SOAP - *Simple Object Access Protocol* - e ESB - *Enterprise Service Bus* - para composição de serviços. Em *microservices* não há esta centralização proporcionada pelo barramento de serviços, a lógica reside nos próprios serviços (SHARMA, 2016).

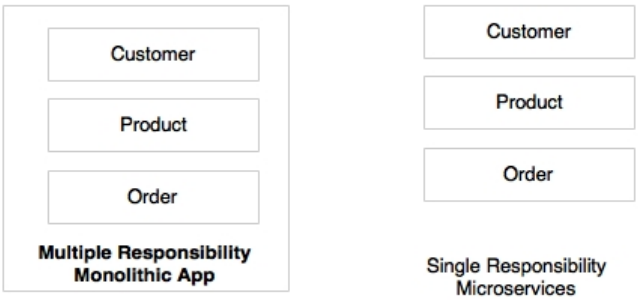
2.4.1 Características

Como a arquitetura de *microservices* é relativamente recente, sendo originada a partir de práticas do mundo real, ainda não há um documento formal elaborado por algum órgão de padronização. Contudo, a literatura permite a identificação de algumas características em comum entre as arquiteturas que utilizam *microservices*.

Uma única capacidade de negócio

Um serviço na arquitetura de *microservices* representa uma única capacidade de negócio, uma única tarefa, com o propósito principal de executá-la da melhor forma possível (MUELLER, 2015). Esta característica é inspirada no Princípio da Responsabilidade Única do padrão de projeto *SOLID*, de Robert C. Martin, da programação orientada a objetos (NEWMAN, 2015; RV, 2016). Isto torna claro para os desenvolvedores a associação do código à sua funcionalidade de negócio relacionada, evitando que um serviço compartilhe responsabilidades com outro ou que possua mais de uma responsabilidade, o que resultaria em um indesejável alto índice de acoplamento entre os serviços. A Figura 4 ilustra o contraste entre as múltiplas responsabilidades de uma aplicação monolítica e microservices com responsabilidades únicas.

Figura 4 – Princípio da Responsabilidade Única



Fonte: Rv (2016)

Fowler e Lewis (2014) destacam que cada *microservice* contempla todos os aspectos do software, incluindo armazenamento de dados, lógica de negócio e interface com usuário.

Para que isto ocorra, os referidos autores consideram que as equipes de desenvolvimento devem ser multi-funcionais, em virtude da Lei de *Conway*[11], em que o projeto de um sistema é uma cópia da estrutura de comunicação da organização. A figura 5 ilustra as consequências desta lei: quando equipes distintas trabalham em suas especialidades o sistema segue a forma de comunicação das equipes, ou seja, separado ou particionado por especialidade; por sua vez as equipes multidisciplinares produzem sistemas individuais completos, contemplando todos os aspectos do software, inclusive a interação com outros sistemas.

Outro aspecto em relação às equipes é que elas serão sempre responsáveis pelo que produzem, quebrando a ideia tradicional de sistemas criados no âmbito de projetos, com início e fim pré-determinados. Como consequência, há a tendência por parte dos membros da equipe de entregar o máximo possível de qualidade do produto, uma vez que ficarão responsáveis pela manutenção durante a vida útil do sistema. (FOWLER; LEWIS, 2014; RV, 2016). Fowler e Lewis (2014) se referem a este aspecto como "*mentalidade de produto*", ressaltando que está intimamente ligado com as capacidades de negócio: ao invés de enxergar um *software* como um conjunto de funcionalidades a serem implementadas, deve-se enxergar como partícipe de uma relação contínua em que o objetivo é fazer o *software* melhorar as capacidades de negócio providas pelos seus usuários. Os autores citam ainda que esta mentalidade poderia ser aplicada em aplicações monolíticas, mas é facilitada nas aplicações baseadas em *microservices* em virtude de sua granularidade.

As características das esquipes envolvidas com o desenvolvimento de *microservices* apontam para outra tendência contemporânea, a de *DevOps*, que consiste resumidamente em um realinhamento entre desenvolvedores e equipe de operações para alcançar uma melhor eficiência (RV, 2016). Normalmente a adoção de *DevOps* implica no uso de práticas ágeis, integração contínua, entrega ou *deploy* automatizado, testes funcionais unitários e de integração do sistema, monitoramento de serviços e infra-estrutura (VENNAM et al., 2015).

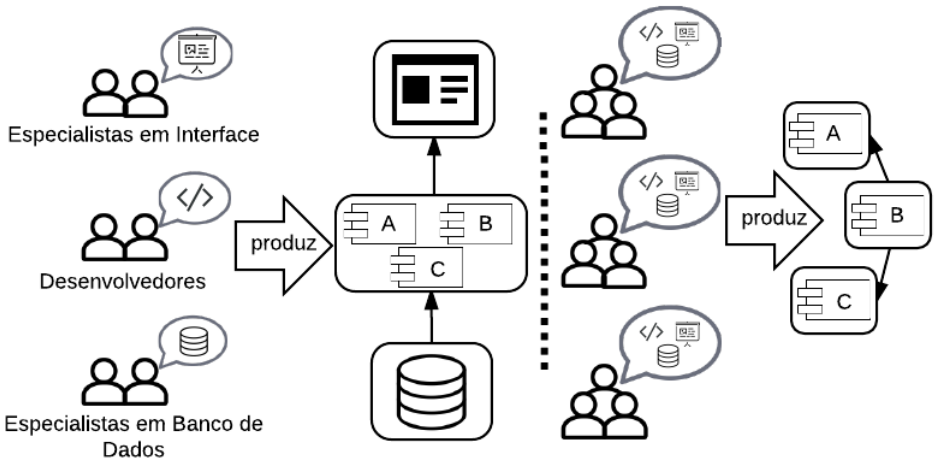
Descentralização de governança e gerenciamento de dados

A independência nativa dos *microservices* provê liberdade para que os desenvolvedores de software escolham as tecnologias ou linguagens que usarão no decorrer da vida útil de um sistema, proporcionando maior facilidade na adoção de novas tendências sem o aprisionamento às escolhas feitas no início do desenvolvimento, minimizando os efeitos e dificuldades de um sistema legado. Em virtude disto, os *microservices* possibilitam reações mais rápidas às inevitáveis demandas de mudança em sistemas (NEWMAN, 2015).

Cada *microservice* é executado em um processo independente, implicando de fato em um sistema operacional executando em uma máquina física ou virtual dedicada para

[11] http://www.melconway.com/Home/Conways_Law.html

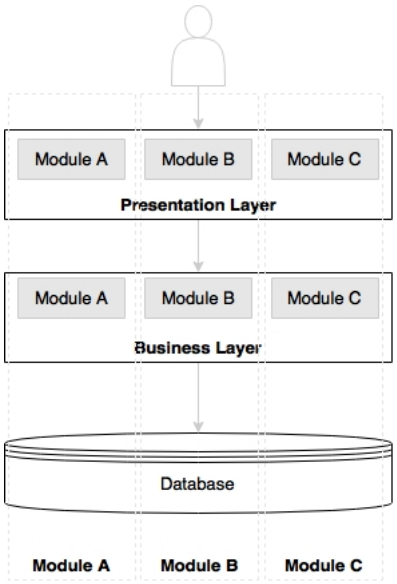
Figura 5 – Contraste entre equipes e arquitetura resultante considerando arquitetura tradicional e microservices em virtude da Lei de Conway



si. Isto proporciona uma grande vantagem em termos de escalabilidade, permitindo que um determinado serviço seja escalonado de forma independente de outros serviços para atender uma maior demanda momentânea, por exemplo (ASHMORE, 2016). Esta é uma grande diferença em relação a aplicações monolíticas, em que recursos tais como banco de dados e base de código são compartilhados por toda a aplicação. A arquitetura de *microservices* preconiza que cada seviço possua seus próprios recursos, possibilitando o *deploy* ou implantação em ambiente de produção de maneira individual independente de outros serviços (VENNAM et al., 2015). As figuras 6 e 7 ilustram as diferenças arquiteturais básicas entre uma arquitetura monolítica e uma arquitetura de *microservices*: enquanto na aplicação monolítica existe um único sistema composto de módulos concentrando informações em um único banco de dados, na arquitetura de *microservices* cada módulo é um *microservice* individual com seu próprio banco de dados.

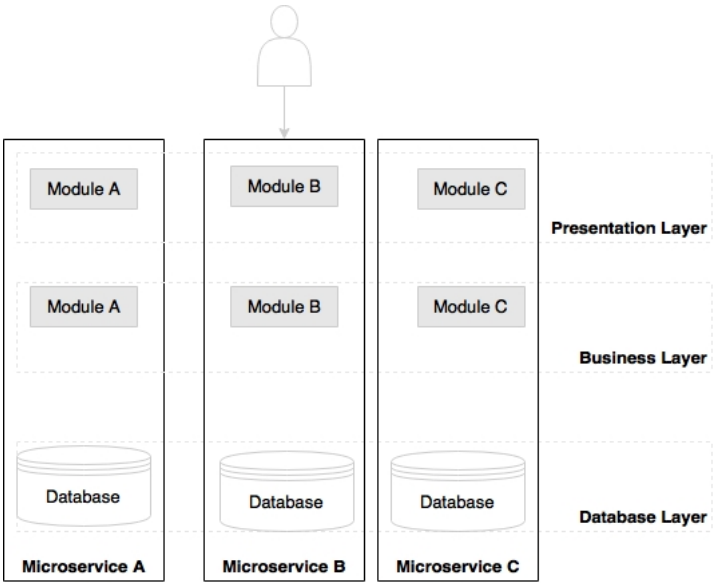
A descentralização dos dados não é só em nível de armazenamento, mas também em nível conceitual, de modelagem. Isto porque para *microservices* distintos uma mesma entidade pode ter diferentes significados ou visões distintas. Fowler e Lewis (2014) citam como exemplo as diferenças entre o conceito de cliente para os setores de vendas e de suporte numa organização: nem todos os clientes de um setor serão do outro, e nem todos os atributos de um cliente serão necessários para os dois setores. Esta diferenciação está relacionada com o conceito de *Bounded Context* elaborado por Evans (2003), em que há uma delimitação do contexto de aplicabilidade de um modelo do domínio de negócio, permitindo um entendimento claro e compartilhado entre os membros da equipe de desenvolvimento sobre o que precisa ser consistente, além da maneira como ocorre o relacionamento com outros contextos. Birchall (2016) ressalta que, no contexto de microservices, a utilização destes contextos implica que cada serviço fica livre para alterar seu próprio modelo de dados sem prejudicar outros serviços.

Figura 6 – Arquitetura de uma aplicação monolítica



Fonte: Rv (2016)

Figura 7 – Arquitetura de *microservices*



Fonte: Rv (2016)

Como consequência desta individualização de recursos que abrange os dados utilizados pelos *microservices*, o conjunto de princípios transacionais de banco de dados conhecido pelo acrônimo *ACID* (Atomicidade, Consistência, Isolamento e Durabilidade) não são garantidos globalmente, ou seja, para a transação de nível mais alto que utiliza os microservices. O *software* precisa tratar a possibilidade de falha em sua implementação e

reverter os dados para um estado consistente (GONZALEZ, 2016). Esta característica é conhecida como *consistência eventual*. De acordo com Bailis e Ghodsi (2013), uma das primeiras definições de consistência eventual surgiu em um artigo de 1988, descrevendo um sistema de comunicação em grupo:

> *"...mudanças feitas em uma cópia eventualmente migrarão para todas. Se toda atividade de atualização parar, após um período de tempo todas as réplicas do banco de dados irão convergir para serem logicamente equivalentes: cada cópia do banco de dados irá conter, em uma ordem previsível, os mesmos documentos; réplicas de cada documento irão conter os mesmos campos."*

As garantias dos princípios *ACID* foram propostas levando em consideração a execução de um banco de dados em um único nó, numa única máquina. Não são viáveis para bancos de dados distribuídos em virtude de alta complexidade e custo. Por definição estes princípios não se aplicam a bancos de dados distribuídos, que estão naturalmente relacionados a outro conjunto de princípios, conhecido pelo acrônimo *BASE* (WILDER, 2012):

- ***B**asically **A**vailable* - o sistema permanecerá disponível e responderá, mesmo que com dados antigos;
- ***S**oft State* - o estado do sistema pode não se encontrar consistente e pode ser corrigido;
- ***E**ventually Consistent* - aceita-se um tempo (retardo) para que uma atualização em um dos nós seja completamente propagada para todos os outros.

No universo de bancos de dados distribuídos, existe um teorema conhecido por *CAP Theorem*, de Eric Brewer, que atesta ser impossível que um sistema ofereça total disponibilidade (*Availability*) e garanta que os usuários terão a última versão dos dados (*Consistency*), na presença de falhas parciais (*Partitions*) (BAILIS; GHODSI, 2013). Entende-se como presença de falhas parciais a falha de comunicação entre dois processos ou grupos de processos em um sistema distribuído. A consistência eventual emerge como uma alternativa de solução para que os sistemas distribuídos priorizem sua disponibilidade na inevitável presença de problemas entre a comunicação de seus processos.

Newman (2015) cita como alternativa para manter a natureza transacional em um sistema distríbuído o uso de transações distribuídas: esta solução tenta espalhar múltiplas transações usando um processo central chamado de "*gerenciador de transação*", que faz uso de um algorítimo para tentar garantir o estado consistente no sistema. O autor cita

ainda o algoritmo "*two-phase commit*", em que na primeira fase cada participante indica que está apto a realizar o *commit* local da transação ao gerenciador de transações, e na segunda fase, após receber retorno positivo de todos os participantes, o gerenciador de conexões indica aos participantes que podem proceder o *commit*. Contudo, conforme o autor, diversos problemas podem emergir desta abordagem, em virtude de problemas nos processos ou na comunicação entre eles, uma vez que após a primeira fase cada participante aguarda o retorno do gerenciador de transações, que pode nunca ocorrer.

Design evolucionário

Esta característica é também uma consequência das anteriores: em virtude da independência dos *microservices* e da descentralização arquitetural, a evolução dos sistemas é sobremaneira facilitada, pois a adição de novas funcionalidades diz respeito a construir novos serviços ou modificar os existentes usando as ferramentas e linguagens mais apropriadas de forma desagregada do *software* existente e de sua infraestrutura. Vennam et al. (2015) ressalta que só é preciso realizar o *deploy* do *microservice* que foi modificado, que a mudança afetará apenas os consumidores deste *microservice* e que, se a interface não foi modificada, todos os consumidores continuarão operando normalmente. As mudanças podem ser feitas rapidamente e de forma frequente com baixo risco.

Fowler e Lewis (2014) pontuam que os praticantes de *microservices* vêem a decomposição dos serviços como ferramenta adicional para controlar as mudanças na aplicação sem gerar lentidão na implementação de mudanças, defendendo que com o uso de ações e ferramentas corretas as mudanças de software podem ser frequentes, rápidas e bem controladas.

Inteligência de negócio nos próprios serviços

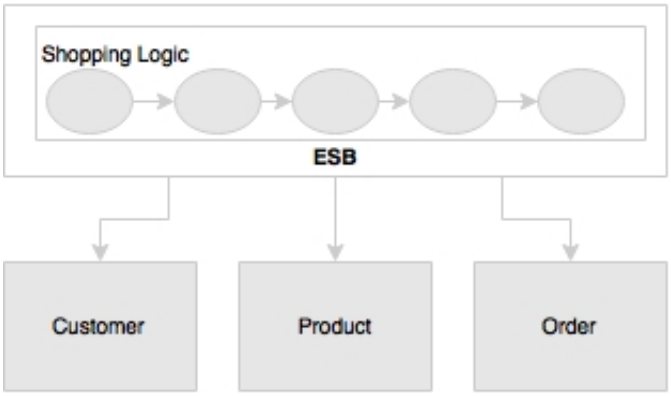
É comum, em sistemas baseados em SOA, a orquestração dos serviços por meio de um *Enterprise Service Bus* (ESB), ou a coreografia por meio da adesão a protocolos complexos como *WS-CDL*[12] ou *BPEL*[13]. Estas abordagens vão além da comunicação entre os serviços, colocam parte da inteligência de negócio fora dos serviços propriamente ditos (FOWLER; LEWIS, 2014; GONZALEZ, 2016), para proporcionar a adequada interação entre eles a fim de atingir um objetivo de negócio específico.

A utilização de *Enterprise Service Bus* em soluções baseadas em *SOA* para mediar a comunicação dos clientes com os serviços, além de se apoderar de parte da inteligência de negócio, dificultando a adição de novas funcionalidades em virtude da complexidade e tempo necessários, pode se tornar um gargalo e ponto único de falha do sistema quando submetido a uma grande quantidade de usuários da internet (VILLAMIZAR et al., 2015).

[12] https://www.w3.org/TR/ws-cdl-10/
[13] http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

A Figura 8 ilustra uma abordagem em que parte da inteligência de negócio extrapola os serviços, em virtude do *ESB* reter a sequência lógica de chamada aos serviços necessários para completar uma tarefa.

Figura 8 – *ESB* retendo parte da inteligência de negócio



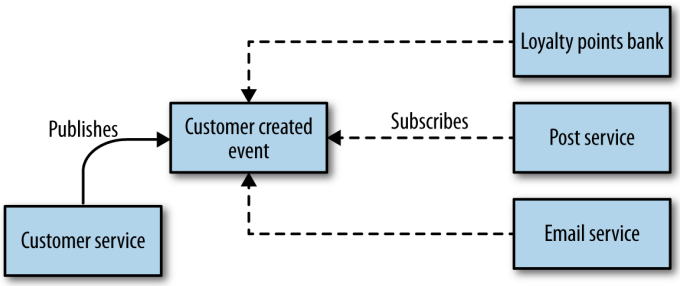
Fonte: Rv (2016)

Apesar da não utilização de um ESB no estilo de SOA, existe no contexto de microservices o padrão API *Gateway*, que evita a intensa comunicação com os microservices de forma individual, promovendo um único ponto de entrada e proporcionando isolamento do cliente quanto a descoberta dos serviços, além de possibilitar diferentes APIs para tipos de clientes distintos (RICHARDSON, 2015).

Por natureza a arquitetura baseada em microservices preconiza que toda a inteligência de negócio resida nos próprios serviços, sem vazar a lógica para a camada de comunicação, sem uma entidade centralizadora que conheça os passos para coordená-los a fim de atingir um objetivo de negócio mais abstrato (RV, 2016; NEWMAN, 2015; FOWLER; LEWIS, 2014; GONZALEZ, 2016). Newman (2015) coloca que pode-se utilizar alternativamente a coreografia aliada com ocorrência de eventos, em que serviços menores seriam executados em decorrência destes eventos, para em conjunto completar um objetivo de negócio de alto nível. Contudo, também é possível utilizar a orquestração por meio de um serviço específico que conheça os passos e coordene a execução de outros serviços (NEWMAN, 2015; RV, 2016). Os tipos de composição dos serviços "orquestração" e "coreografia" podem ser entendidos analogamente como um maestro regendo uma orquestra e uma equipe de dança coreografada em que os participantes reagem à música ou aos movimentos de seus parceiros (JOSUTTIS, 2007). A título de exemplo, as Figuras 9 e 10 ilustram as diferentes abordagens para serviços relacionados ao registro de um cliente: a criação um registro de pontos de fidelidade, o envio físico de um pacote de boas-vindas pelo correio e um *e-mail* de boas-vindas.

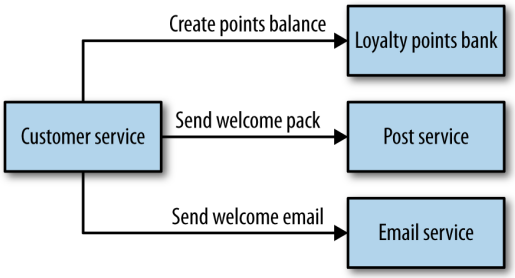
Em virtude desta característica, o protocolo de comunicação usado na interação

Figura 9 – Exemplo de integração de serviços baseada em coreografia



Fonte: Newman (2015)

Figura 10 – Exemplo de integração de serviços baseada em orquestração



Fonte: Newman (2015)

entre os serviços é o mais simples possível, com a única responsabilidade de prover a comunicação entre eles. Normalmente as arquiteturas baseadas em microservices utilizam *APIs* de linguagem neutra baseadas no protocolo HTTP, como REST - *Representational State Transfer* (VENNAM et al., 2015; BALALAIE; HEYDARNOORI; JAMSHIDI, 2016; MUELLER, 2015).

Projetado para ser resiliente

Uma vez que uma aplicação baseada em *microservices* usa os serviços como componentes, em um nível alto de granularidade, é preciso que haja tolerância à ocorrência de erros ou falhas dos serviços (FOWLER; LEWIS, 2014). Neste tipo de arquitetura acredita-se que a ocorrência de erros é inevitável, determinando como um dos objetivos de negócio a manutenção do funcionamento do sistema o máximo de tempo possível, mesmo sob a ocorrência de falhas parciais, tornando-o resiliente. Normalmente são adotados alguns padrões de projeto voltados para testar os sistemas na ocorrência de problemas, como *Circuit Breaker pattern* e *Bulkhead pattern* (VENNAM et al., 2015).

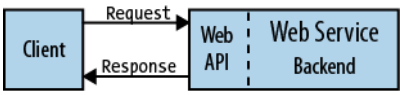
Puglisi (2015) ressalta a facilidade de isolar problemas em sistemas baseados em *microservices*, permitindo que o restante do sistema funcione normalmente, contribuindo

para uma estabilidade global do sistema.

2.5 REST Web APIs

APIs (Application Program Interfaces) são interfaces de programação entre componentes de *software*, que especificam como pode ocorrer a interação entre eles (PUGLISI, 2015). Biehl (2015) ressalta que tradicionalmente o *software* sempre foi usado por pessoas através de interface com o usuário, uma tela contemplando as funções disponíveis. Contudo, continua o autor, cada vez mais o *software* é utilizado por outros *softwares*, o que requer a utilização de *APIs* para esta interação. *Web APIs* são portas de entrada para *web services*, recebendo e respondendo requisições de seus clientes (MASSE, 2011), conforme exposto na Figura 11.

Figura 11 – Web API



Fonte: Masse (2011)

Apesar do termo "*REST*" (*Representational State Transfer*) parecer um tópico recente, em virtude de seu relacionamento com discussões sobre *cloud computing*, dispositivos móveis e *RESTful web services*, seu surgimento remonta ao final do século XX (BOJINOV, 2016), quando a Web se popularizava demandando esforços de padronização dos protocolos de comunicação.

Em meados de 1990, Roy Fielding trabalhou na elaboração da primeira versão do protocolo *HTTP* - Hyper Text Transfer Protocol, junto com outros nomes conhecidos da internet, como Tim Bernes-Lee e Henry Frystyk. Em 1999 foi elaborada com a participação deste mesmo grupo a versão 1.1, definida pela RFC2616, que está em uso até os dias atuais (RICHARDSON; RUBY; AMUNDSEN, 2013; BOJINOV, 2016). Neste ínterim entre as versões 1.0 e 1.1 do protocolo HTTP, foi concebido em 1994 o estilo arquitetural REST, usado pela equipe que trabalhava na especificação do protocolo HTTP para guiar o projeto e o desenvolvimento de uma arquitetura para Web moderna (FIELDING, 2000). Pouco depois, no ano de 2000, Fielding publicou sua tese de doutorado introduzindo formalmente o estilo arquitetural.

Fielding (2000) define REST como um estilo arquitetural híbrido para aplicações baseadas em rede, derivado de vários outros estilos abordados em seu trabalho. Por estilo arquitetural o autor entende um conjunto coordenado de restrições que atuam sobre as funções dos elementos arquiteturais e sobre os relacionamentos entre eles. Ainda segundo

o autor, a expressão "Representational State Transfer" representada pelo acrônimo REST, pretende evocar o comportamento ideal de uma aplicação Web bem projetada:

> *"...uma rede de páginas web (uma máquina de estados virtual), em que o usuário avança através da aplicação selecionando links (transições de estado), resultando na próxima página (representando o próximo estado da aplicação) sendo transferida para o usuário e exibida para seu uso." (FIELDING, 2000)*

Uma Web API que atende às restrições do estilo arquitetural REST é chamada de *REST API*, e seus serviços disponibilizados podem ser chamados de *RESTful Web Services* (MASSE, 2011). Antes de abordar o conjunto de restrições arquiteturais do estilo arquitetural REST proposto por Roy Fielding, é preciso conhecer o significado de alguns termos fundamentais para o seu entendimento, a saber: hipermídia, recursos e representações.

2.5.1 Hipermídia, recursos e representações

Em alguns trechos de seu trabalho Fielding (2000) descreve REST como um estilo arquitetural para sistemas de hipermídia distribuídos. O conceito de hipermídia é uma extensão de hipertexto, ambos inicialmente discutidos por Ted Nelson em 1965 em um projeto intitulado Xanadu, que definia hipertexto como um texto escrito de forma não sequencial que se ramifica e permite a escolha pelo leitor, mais apropriado para leitura em uma tela interativa (LOUVEL; TEMPLIER; BOILEAU, 2012).

A extensão hipermídia tem relação também com o termo "multimídia", entendido como o uso de múltiplos recursos tais como texto, sons, gráficos e vídeos para transmitir informação (KHOSROW-POUR, 2014). A diferença entre eles é a não-linearidade da estrutura das informações: com hipermídia, o usuário tem opção de escolha entre as partes da informação que vai acessar. Para BIEBER et al. (1997), hipermídia encoraja os autores a estruturar seus conteúdos como uma rede associativa de nós e links de relacionamento entre eles, quebrando a estrutura linear que domina documentos impressos e habilitando o acesso à informação na ordem mais apropriada para os usuários.

Este aspecto de relacionamento entre as partes dos conteúdos das informações é contemplado por Fielding (2000) em sua definição de hipermídia:

> *"Hipermídia é definida pela presença de informações de controle da aplicação embutida na apresentação da informação ou em uma camada acima. Hipermídia distribuída permite que a apresentação e as informações de controle sejam armazenadas em localizações remotas."*

As opções que o usuário dispõe e disporá para acessar conteúdos estão embutidas nas próprias informações já consumidas pelo mesmo, e também nas que vier a consumir. Estes são os dados de controle da aplicação, que podem ser armazenados remotamente. Um exemplo clássico de hipermídia é a própria *World Wide Web*. Os conteúdos acessíveis pelos usuários remetem a outro conceito fundamental no estilo REST, os "recursos".

Os "recursos" em REST, objetivamente falando, remetem a tudo o que pode ser acessado ou manipulado pela *Web* em um sistema, como *websites*, documentos, vídeos e imagens. São fundamentais em qualquer sistema baseado na *Web*, ao ponto da própria *Web* ser conhecida como orientada a recursos (PARASTATIDIS; WEBBER; ROBINSON, 2010). Berners-Lee et al. (2004) classificam estes como "recursos de informação", porque todas as suas características essenciais podem ser transportadas em uma mensagem, mas ressaltam que o conceito de recurso é mais amplo: objetos ou coisas do mundo real como um carro ou uma pessoa também são recursos. Não são considerados recursos de informação em virtude de suas naturezas, contudo, podem ser abstraídos por meio de descrições que formariam recursos de informação. Enquanto hipermídia está relacionada com a navegabilidade entre conteúdos, os recursos são os conteúdos propriamente ditos.

Este entendimento amplo do termo "recurso" está alinhando com a definição de Fielding (2000), mas esta vai além: um recurso é uma abstração, um mapeamento conceitual para um conjunto de entidades, não as entidades obtidas propriamente ditas em determinado momento. Percebe-se que nesta definição um recurso é uma função que mapeia entidades num determinado momento do tempo. Por exemplo, o recurso "campeão atual da copa do mundo", que retorna o país cuja seleção masculina de futebol venceu a última edição da copa, no período de 2014 até os dias atuais retornaria Alemanha, já entre 2010 e 2014 retornaria a Espanha. Note que o mapeamento pode retornar mais de uma ou mesmo nenhuma entidade, que seria o caso dos recursos "países campeões de copas do mundo" e "campeão da copa do mundo de 2018" respectivamente, este último porque ainda não ocorreu o referido evento.

Apesar deste conceito mais abstrato, todo conteúdo requisitado na Web é um recurso, como dito anteriormente. Contudo, o requisitante não recebe o recurso propriamente dito, e sim uma *representação* deste recurso. Representações são uma espécie de fotografia do estado de um recurso num dado momento do tempo, materializadas em dados legíveis por máquinas num documento ou arquivo juntamente com os meta-dados para interpretá-los (VARANASI; BELIDA, 2015). Na visão de Bloomberg (2013), representações são os dados que provêm a manifestação concreta dos recursos, o que de fato é enviado para o cliente, conforme illustra a Figura 12. Os meta-dados incluem o tipo dos dados utilizados na representação, como *text/HTML*, *application/XML* e *application/JSON*. Os tipos de dados possíveis para as representações são padronizados pela *IANA - Internet Assigned Numbers*

Authority[14].

Figura 12 – Exemplo de representação de recurso usando formato *application/JSON*

```
HTTP/1.1 200 OK
Content-Type: application/JSON
...
{
        "pessoa": {
                "nome": "Joao Alberto",
                "idade": "35",
                "identidade": "12345"
                }
}
```

2.5.2 Identificação de Recursos

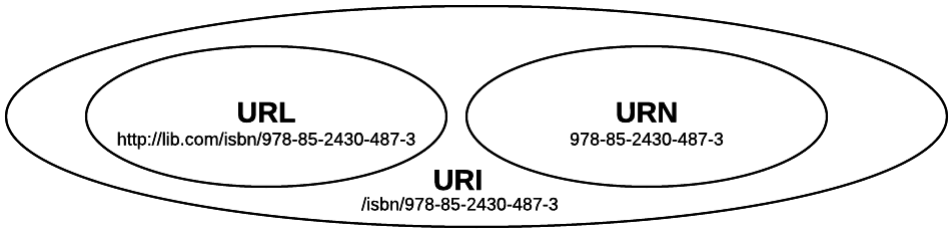
Na arquitetura da *Web*, para que um cliente requisite um recurso a um servidor, ele precisa saber identificá-lo. A identificação é feita por meio de *URIs (Uniform Resource Identifier)* (VARANASI; BELIDA, 2015). Berners-Lee et al. (1998) especificam formalmente o termo URI na RFC2396, e seu relacionamento com URL (*Unified Resource Locator*) e URN (*Unified Resource Name*):

- URI - provê um meio simples e extensível para identificar recursos, usando um conjunto compacto de caracteres que segue uma gramática estabelecida. É um superconjunto que engloba URL e URN, conforme exibido na Figura 13. Exemplo: "/livros/isbn/978-85-2430-487-3".
- URL - São URIs que contêm o servidor e o protocolo de comunicação. Exemplo: "http://bibliotecaexemplo.com/livros/isbn/978-85-2430-487-3". As URLs que apontam para *web services* também são chamadas de *endpoints*.
- URN - Se refere à parte da URI que identifica de forma única e persistente um recurso mesmo que ele se torne indisponível. Bloomberg (2013) ressalta que URNs identificam recursos em um sistema de nomes particular, mas não especificam a localização nem como acessá-los. Ex.: "978-85-2430-487-3".

A sintaxe global de uma URI é definida por Berners-Lee et al. (1998) na RFC2396 conforme especificado na Figura 14, em duas partes que indicam o esquema de identificação do recurso seguido de seu caminho de localização atendendo às especificações de formato

[14] https://www.iana.org/

Figura 13 – Diagrama de Venn dos conjuntos de endereçamento URI, URL e URN



do esquema. A RCF2396 apresenta também uma gramática completa para construção de URIs atendendo aos diversos esquemas possíveis. A Figura 15 ilustra o esquema utilizado para o protocolo HTTP conforme estabelecido por Fielding et al. (1999) na RFC2616.

Figura 14 – Sintaxe geral de uma URI

```
scheme:scheme-specific-part
```

Fonte: RFC2396

Figura 15 – Sintaxe do esquema HTTP

```
http_URL = "http:" "//" host [":" port ] [ abs_path ["?" query ]]
```

Fonte: RFC2616

A criação de URIs intuitivas é encorajada no âmbito da utilização de *Web APIs*, facilitando o entendimento para os desenvolvedores das aplicações clientes (PARASTATIDIS; WEBBER; ROBINSON, 2010). Neste aspecto, com o objetivo de abstrair um espaço comum de um conjunto de recursos similares, foi criado o conceito de *URI Template*, para descrever uma faixa de URIs por meio do uso de variáveis (GREGORIO et al., 2012). Normalmente *URI Templates* são utilizadas em documentações de *Web APIs*.

Tabela 1 – Exemplos de URI Templates

Exemplos de Templates	**Variaveis**	**URIs expandidas**
http://dic.o/~{username}/	username	http://dic.o/~fred/ http://dic.o/~mark/
http://dic.o/{term:1}/{term}	term:1, term	http://dic.o/c/cat http://dic.o/d/dog
http://dic.o/search{?q,lang}	q, lang	http://dic.o/search?q=cat&lang=en http://dic.o/search?q=chien&lang=fr

2.5.3 Restrições REST

Baseando-se no trabalho de Fielding (2000) e na análise de Richardson, Ruby e Amundsen (2013), as restrições arquiteturais do estilo *REST* podem ser resumidas da

seguinte maneira:

- **Cliente-servidor** - Proveniente do estilo arquitetural com mesmo nome, dominante na *internet*, esta restrição determina que um componente cliente que deseja um serviço deve enviar uma requisição a um servidor, que pode rejeitar ou aceitar a requisição, enviando uma resposta para o cliente.
- **Sem estado** - O estado da aplicação é de responsabilidade do cliente. Para o servidor o cliente só existe durante a execução de uma requisição.
- ***Cache*** - O cliente pode guardar respostas obtidas do servidor para não precisar fazer novas requisições quando possível. As respostas providas pelo servidor precisam informar se podem ser guardadas e por quanto tempo, por questões de segurança da informação.
- **Sistema em camadas** - Elementos intermediários, como um servidor *proxy*, podem ser inseridos entre o cliente e o servidor com o objetivo de adicionar funcionalidades como balanceamento de carga e verificações de segurança do sistema.
- **Código sob demanda** - O servidor pode enviar em suas respostas, além de dados, código a ser executado pelo cliente, com objetivo de prover o *know-how* de processamento dos dados ao cliente. Entre as vantagens, a extensibilidade da aplicação por meio de adição de funcionalidades e a performance em virtude da execução local no cliente. Contudo, esta característica pode trazer problemas quando os clientes não confiam nos servidores.
- **Interface uniforme** - Esta restrição é subdividida em quatro partes:
  - **Identificação de recursos** - cada um dos recursos disponibilizados deve ser identificado por uma *URI* estável, que não muda com o tempo, mesmo que o estado do recurso se modifique.
  - **Manipulação de recursos através de representações** - Os recursos são descritos por meio de representações. A manipulação do estado do recurso por parte do cliente é feita pelo envio de representações para o servidor.
  - **Mensagens auto-descritivas** - Toda a informação necessária para entender uma requisição do cliente ou uma mensagem de resposta enviada pelo servidor está contida na própria mensagem.
  - **Hipermídia como o motor do estado da aplicação** - Para manipular o estado da aplicação, o cliente faz uso dos *links* contidos nas representações recebidas. Ou seja, o servidor precisa disponibilizar em suas respostas as outras opções de navegabilidade que o cliente pode fazer uso, como um cardápio com as opções disponíveis. Isto facilita a extensibilidade da aplicação.

Para Fielding (2000), a aplicação das restrições do estilo *REST* em conjunto, provê escalabilidade da interação de componentes, generalidade de interfaces, *deploy* independente de componentes, e a utilização de componentes intermediários que objetivam a redução da latência de rede na interação, além de reforçar a segurança e encapsular sistemas legados.

2.5.4 Modelo de Maturidade de Richardson

Baseando-se no suporte que um serviço *web* promove aos conceitos de recursos, URIs, protocolo HTTP e hipermídia, Richardson (2008) criou um modelo de maturidade que envolve quatro níveis:

- Nível 0: O serviço utiliza apenas uma URI e um método HTTP. Como exemplo, o autor cita XML-RPC e serviços SOAP. Toda informação sobre a operação está contida nos dados submetidos na requisição;
- Nível 1: O serviço faz uso do conceito de recursos, utilizando múltiplas URIs;
- Nível 2: Além de utilizar recursos, faz uso dos métodos HTTP para indicar a operação realizada;
- Nível 3: Os recursos descrevem as suas próprias capacidades e interconexões, contendo os *links* para guiar o cliente. É a restrição do estilo REST referente à hipermídia como mecanismo do estado da aplicação, também conhecida pelo acrônimo HATEOAS (*Hypermedia As The Engine Of the Application State*).

2.5.5 *REST* e o protocolo *HTTP*

Apesar do estilo *REST* ter sido proposto por um dos membros da equipe que trabalhou na especificação do protocolo *HTTP*, ele não é um estilo arquitetural específico para um determinado protocolo: é abstrato, para qualquer protocolo baseado em *hypertext* (BLOOMBERG, 2013). Em virtude disto, todas as suas restrições arquiteturais não estão relacionadas a nenhuma característica de implementação em particular. Contudo, o protocolo *HTTP* é o principal protocolo de transferência de dados na *Web* atualmente, contribuindo para que a utilização de *REST* com este protocolo seja a implementação mais comum (PURUSHOTHAMAN, 2015).

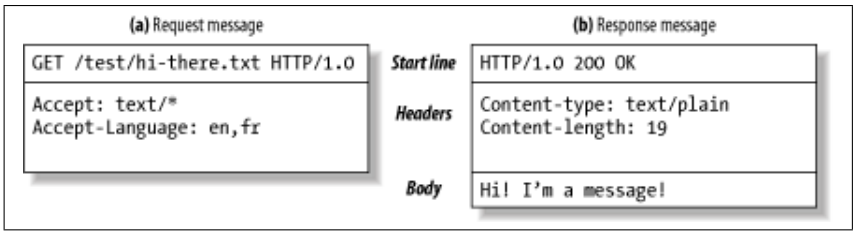
Uma transação *HTTP* consiste em uma requisição feita pelo cliente e uma resposta emitida pelo servidor, por meio de blocos de dados formatados chamados de “mensagens HTTP” (TOTTY et al., 2002). A Figura 16 contempla exemplos de mensagens de requisição e resposta *HTTP*. O protocolo *HTTP* provê oito métodos, também chamados de “verbos”, que permitem a interação e manipulação de recursos, dentre os quais os mais comuns são

GET - para obter a representação de um recurso; POST - para criar um novo recurso baseado em uma representação; PUT - para atualizar ou substituir um recurso baseado numa representação; e DELETE - para excluir um recurso. (VARANASI; BELIDA, 2015).

De acordo com a especificação RFC2616 (FIELDING et al., 1999), a primeira linha de uma requisição especifica o verbo utilizado, a localização do recurso, e a versão do protocolo HTTP. Totty et al. (2002) dividem uma mensagem HTTP em três partes:

- Linha inicial - indicando o que fazer numa requisição ou o que ocorreu numa resposta;
- Campos de cabeçalho - opcionalmente após a linha inicial há campos chave/valor separados por ":".
- Corpo - Separando-se dos elementos iniciais por uma linha em branco, opcionalmente há o corpo da mensagem com qualquer tipo de dados, em qualquer sentido: na requisição ou na resposta.

Figura 16 – Partes de uma mensagem HTTP



Fonte: (TOTTY et al., 2002)

As respostas do servidor às requisições recebidas ocorrem por meio de mensagens *HTTP* com *Status Codes* na primeira linha. Estes códigos também são especificados na RFC2616, e são agrupados de acordo com sua semântica, por exemplo: 4XX (códigos entre 400 e 499) indicam erros no cliente, enquanto 5XX (códigos entre 500 e 599) erros no servidor.

2.5.6 Idempotência e Segurança em REST

Varanasi e Belida (2015) destacam dois aspectos a considerar dos verbos HTTP utilizados em uma arquitetura que segue o estilo REST: idempotência e segurança. Uma operação ou requisição é considerada idempotente se produz o mesmo estado em um recurso do servidor independente do número de vezes que for realizada. Exemplos de verbos idempotentes são GET, PUT e DELETE. No caso do DELETE, caso o cliente realize uma segunda requisição, certamente a resposta enviada pelo servidor será diferente porque o recurso já foi excluído, mas a essência da idempotência é o estado do recurso e não a resposta enviada ao cliente.

O outro aspecto, a segurança dos verbos, diz respeito àqueles que não causam nenhuma mudança no estado do recurso, como GET e HEAD. São métodos apenas de leitura, que podem ser executados várias vezes sem causar alteração no estado do recurso. Assim como na idempotência, a resposta enviada pelo servidor pode não ser a mesma, pois o recurso pode ter sido modificado por outro cliente, mas o estado do recurso não foi alterado em decorrência da requisição, por isso estes métodos são considerados seguros.

2.5.7 Segurança de *Web APIs*

Em aplicações *web* tradicionais, incluindo sistemas legados monolíticos, a segurança no acesso à informação é normalmente garantida por meio de autenticação com nome de usuário e senha previamente cadastrados pelo usuário. As *Web* APIs, no entanto, podem ser utilizadas tanto por usuários como por máquinas, dispositivos ou outros *softwares*. Em virtude disto, a segurança destas APIs pode ser provida de outras formas. Varanasi e Belida (2015) listam seis abordagens de segurança mais populares no contexto de Web APIs:

- Segurança baseada em sessão de usuário - similar à tradicional autenticação de usuário com login e senha: quando a requisição chega ao servidor este verifica se o usuário possui uma sessão autenticada. Caso negativo, solicita ao usuário a entrada de suas credenciais. Esta abordagem, no entanto viola o princípio REST de não manter estado no servidor;
- Autenticação Básica HTTP - o cliente adiciona à requisição um cabeçalho de autenticação contendo usuário e senha separados pelo caractere ':' e codificados em Base64;
- Autenticação *Digest* HTTP - utiliza funções *hash* sobre usuário, senha, um *token* gerado pelo servidor, método HTTP e URI utilizados.
- Segurança baseada em certificados - utiliza uma Entidade Certificadora para garantir a identidade do cliente.
- XAuth e OAuth - protocolos que provêem um mecanismo de acessar recursos protegidos em lugar do usuário sem a necessidade de armazenar sua senha.

OAuth 2.0

Dentre as formas de segurança de *Web APIs* o protocolo OAuth 2.0 é um dos mais populares. Para compreender o fluxo deste protocolo, é importante distinguir os conceitos de autenticação e autorização: o primeiro diz respeito a verificar a identidade do usuário que está tentando acessar uma aplicação ou serviço; o segundo é o processo de

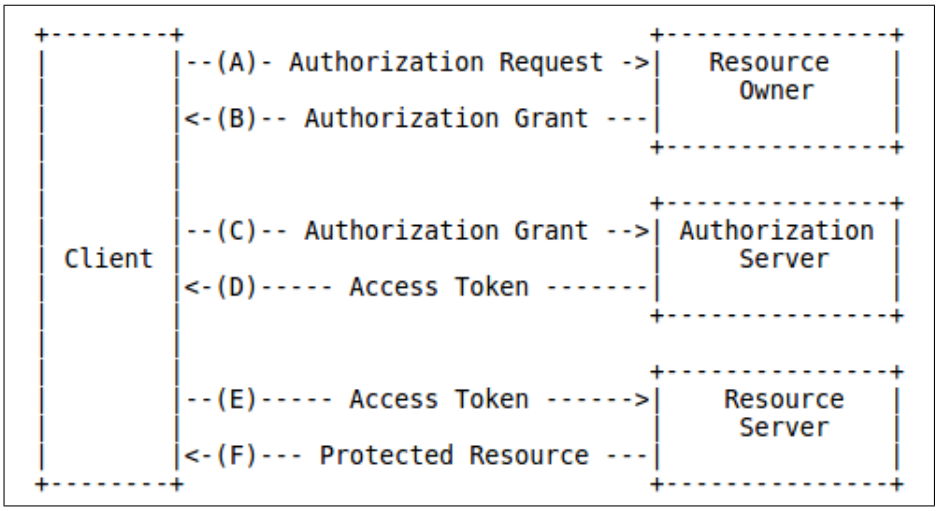
verificar o que um usuário autenticado pode fazer na aplicação (PURUSHOTHAMAN, 2015). Conforme a RFC6749, O *framework* OAuth 2.0 define os seguintes papéis:

1. **Dono do Recurso** - a entidade ou pessoa capaz de fornecer acesso a um recurso protegido;
2. **Servidor do Recurso** - o servidor que armazena os recursos protegidos;
3. **Aplicação Cliente** - a aplicação que realiza requisições a recursos protegidos no lugar do Dono do Recurso e com sua autorização;
4. **Servidor de Autorização** - o servidor que provê *tokens* de acesso para a Aplicação Cliente após autenticar o Dono do Recurso e obter sua autorização.

A Figura 17 ilustra de maneira abstrata o fluxo de ações relacionadas com a autorização obtida com o *framework* OAuth 2.0. As credenciais do Dono do Recurso, ou usuário, normalmente não são informadas diretamente na Aplicação Cliente por questões de segurança. Ao invés disto, a aplicação cliente utiliza o navegador *web* para redirecionar o usuário para uma página de autenticação própria do Servidor de Autorização, que após a devida autenticação provê a concessão de autorização. A concessão de autorização prevista no OAuth 2.0 pode ser de 4 tipos especificados pela própria RFC ou por tipos estendidos. Dentre os tipos especificados o principais são (BIHIS, 2015):

- **Concessão de Código de Autorização**, concessão **server-side** ou **cliente confiável** - Neste tipo de concessão, após a autenticação do Dono do Recurso, o Servidor de Autenticação retorna para a aplicação cliente um Código de Autorização, que deve ser usado pela aplicação cliente para obter o *token* de acesso. Este é o tipo de concessão de autorização recomendado pela RFC6749 por razões de segurança, porque possibilita a autenticação da própria aplicação cliente (além do Dono do Recurso) e permite que o *token* de acesso seja enviado diretamente para a aplicação ao invés do navegador *web* do usuário. É conhecido tambem como *server-side* ou *cliente confiável* porque está relacionado com Aplicações Clientes que são capazes de armazenar e transmitir de forma segura informações confidenciais, como aplicações típicas de 3 camadas: cliente, servidor e banco de dados.
- **Concessão Implícita**, concessão **client-side** ou **cliente não confiável** - Neste tipo de concessão, após a autorização do Dono do Recurso, o Servidor de Autorização provê de forma direta o *token* de acesso. Também conhecido como *client-side* ou cliente não confiável porque está relacionado a aplicações nativas de navegadores de internet, como aplicações JavaScript/HTML, em que não há um servidor disponível para armazenar informações confidenciais.

Figura 17 – Fluxo abstrato do OAuth 2.0.



Fonte: RFC6749

Bihis (2015) destaca dois usos práticos do protocolo OAuth 2.0 comumente encontrados: identidade federada, que consiste em permitir que um usuário acesse uma aplicação com a conta de outra, por exemplo o login em aplicações com a conta do *Facebook*; e autoridade delegada: quando um serviço acessa outro serviço em lugar do usuário.

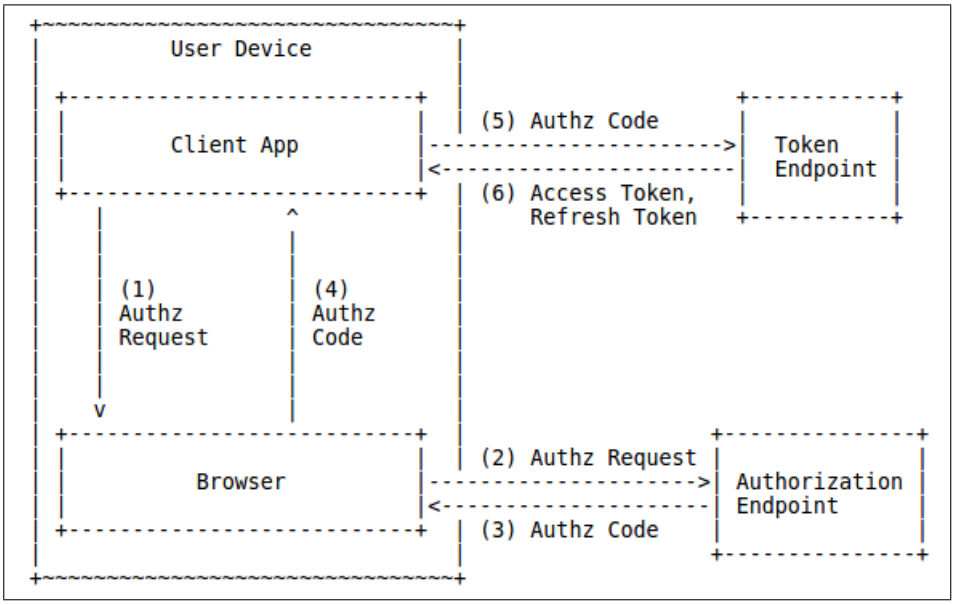
Em relação a aplicativos de dispositivos móveis, a RFC6749 informa que pode ser utilizado tanto a Concessão de Código de Autorização quanto a Concessão Implícita, e ilustra possibilidades de interação entre o aplicativo nativo e o navegador de internet para obtenção de respostas do Servidor de Autorização. Dentre estas possibilidades, há uma em que o sistema operacional do dispositivo disponibiliza um esquema de URI próprio para que aplicação possa receber o retorno do Servidor de Autorização. Boyd (2012) ressalta que nos casos em que a aplicação nativa dispõe de um servidor *mobile backend*, os fluxos tradicionais da especificação do OAuth 2.0 podem ser utilizados com este servidor, mas quando não dispõe é preciso usar o que o autor chama de fluxo de aplicação nativa, similar aos fluxos tradicionais, no entanto apresentando duas restrições: a ausência do servidor para obter o código de autorização ou *token* de acesso e a necessidade de armazenar no dispositivo as credenciais da aplicação cliente, necessárias quando o tipo de concessão de autorização envolve o Código de Autorização.

Neste contexto, também há um *draft* da IETF[15], no estado de "melhor prática", que detalha um fluxo de autorização específico para aplicativos nativos de dispositivos móveis, semelhante à Concessão de Código de Autorização, mas detalhando a interação entre o aplicativo em si e o *browser* de internet do dispositivo (DENISS; BRADLEY, 2016), conforme exposto na Figura 18. Este draft também alerta para a insegurança de armazenar no código do aplicativo as credenciais da aplicação cliente, uma vez que o código do

[15] Internet Engineering Task Force. https://www.ietf.org/

aplicativo é passível de engenharia reversa. O referido risco também é explicado na própria especificação do OAuth, a RFC6749 (LODDERSTEDT; MCGLOIN; HUNT, 2013). Para contornar este problema, o *draft* recomenda não exigir credenciais de cliente com senha das aplicações nativas, ou caso o faça, não distribuir novos *tokens* de acesso tomando como base o fato de que o usuário concedeu permissão anteriormente para a mesma aplicação cliente, pois esta pode ter sido falsificada. Outras alternativas, abordadas pela RFC6819, que trata especificamente de aspectos de segurança do protocolo OAuth, são a utilização de armazenamento seguro provido pelos sistemas operacionais dos dispositivos ou a utilização de servidores *backend* para realizar o armazenamento de credenciais de aplicação cliente ou *tokens* de acesso.

Figura 18 – Fluxo de autorização para aplicativos nativos de dispositivos móveis.



Fonte: Deniss e Bradley (2016)

2.6 Trabalhos relacionados

Com base na metodologia utilizada foram encontrados trabalhos correlatos envolvendo modernização de sistemas, mas não com o fim de disponibilização em dispositivos computacionais móveis, o que torna este o principal diferencial deste trabalho.

Kangasaho (2016) em sua dissertação de mestrado aborda a modernização de sistemas legados com a técnica de REST *Wrapping*. O autor destaca a importância de negócio dos sistemas legados para as organizações, reconhecendo que possuem funcionalidades críticas para o negócio sem as quais as organizações não existiriam, mas ressalta a dificuldade de mantê-los e evoluí-los. Apresenta como alternativas a retirada do sistema e sua completa substituição por um novo, o que considera de alto risco e alto custo; ou a modernização do sistema, que pode ser categorizada em redesenvolvimento, migração e

wrapping. Especificamente seu trabalho situa REST *Wrapping* no contexto de modernização de sistemas e tenta identificar que tipos de sistemas legados são mais indicados para esta técnica, abordando dois estudos de caso presentes na literatura e apresentando dois novos de sua própria experiência.

A modernização de aplicações legadas para disponibilização em *cloud computing*, como SaaS (Software as a Service), é abordada por Zhang et al. (2009) em um artigo científico. Os autores propõem uma metodologia de sete etapas, através da qual se analisa a arquitetura atual do sistema legado, projeta-se o modelo da nova e aplica-se a transformação deste modelo utilizando *Model Driven Architecture* [16]. Esta técnica permite o uso de ferramentas para criar web services, que invocarão funcionalidades do sistema legado. É uma forma de aplicação da técnica de *Wrapping* do sistema legado. Por fim, a metodologia contempla aspectos de *deploy* do sistema em plataformas de computação em nuvem.

No contexto da arquitetura de *microservices*, Balalaie, Heydarnoori e Jamshidi (2016) descrevem em um artigo a sua experência de migração incremental de um sistema comercial para esta arquitetura, explicando como se deu a adoção conjunta com as práticas de *DevOps*[17]. A experiência dos autores proporcionou a elaboração de outro documento: um relatório técnico contendo o que eles chamam de padrões reusáveis de migração. Nestes padrões, diversos problemas da migração são abordados e soluções são propostas, de acordo com um metamodelo próprio.

Levcovitz, Terra e Valente (2016) definem uma técnica para identificar e extrair *microservices* de um sistema monolítico. A técnica envolve um mapeamento entre as diferentes partes do sistema, desde as tabelas do banco de dados à fachada do sistema, estabelecendo critérios para identificar partes candidatas à migração para arquitetura de *microservices* com base nas dependências. O trabalho conta ainda com um relato da aplicação da técnica a um sistema bancário.

A Tabela 2 resume os trabalhos correlatos encontrados e faz uma comparação sucinta de seus objetivos com os deste trabalho.

[16] http://www.omg.org/mda/
[17] http://www.jedi.be/presentations/agile-infrastructure-agile-2008.pdf

Tabela 2 – Comparativo com trabalhos correlatos.

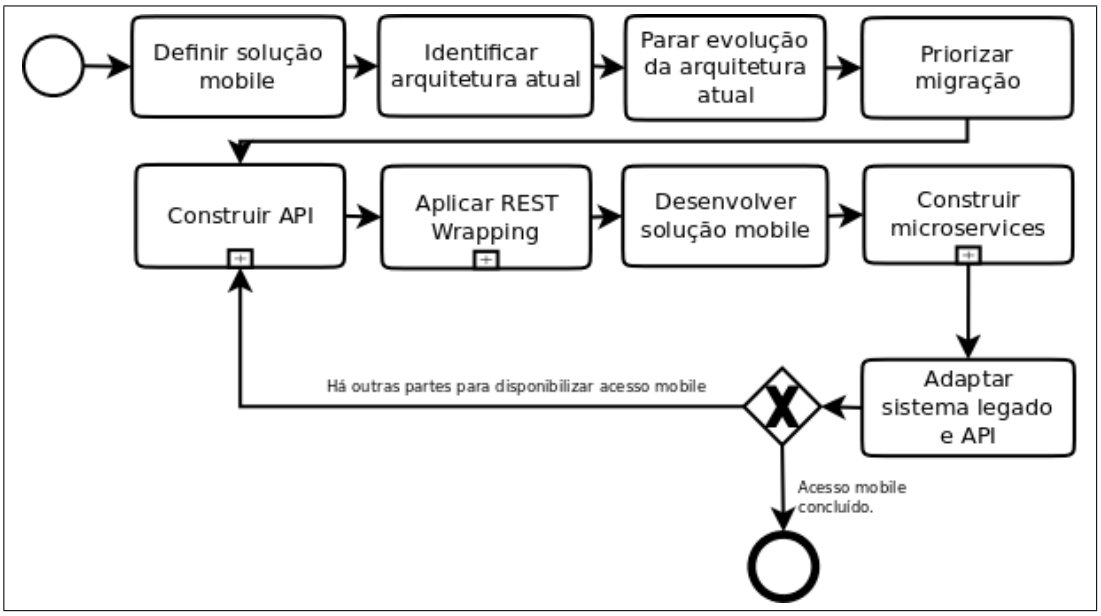
Trabalho	**Principais diferenças**
Kangasaho (2016) situa a técnica de REST Wrapping no contexto de modernização; Analisa alguns estudos de caso.	Neste trabalho a técnica REST Wrapping é utilizada em uma etapa intermediária para acelerar a disponibilização do sistema legado em dispositivos móveis, mas ainda se extrai partes do sistema que são convertidas em microservices eliminando problemas relacionados à tecnologia legada.
Zhang et al. (2009) cria metodologia de sete etapas para modernizar sistemas utilizando *Model Driven Architecture* com objetivo de aplicar SOA Wrapping e disponibilizar *Web services*, abordando aspectos de deploy da aplicação em *cloud computing*.	O objetivo fim da modernização foi a tecnologia de *Web services*, enquanto neste trabalho utilizou-se a arquitetura de microservices com *RESTful web services*, que é mais moderna. Apesar de abordar *deploy* em plataformas de nuvem, que não foi abordado neste trabalho, também não envolveu dispositivos móveis computacionais.
Balalaie, Heydarnoori e Jamshidi (2016) descrevem experiência de migração incremental para arquitetura de *microservices* em conjunto com práticas DevOps em um estudo de caso, de onde derivam um relatório técnico com padrões reusáveis de migração.	Utilizou uma demanda real, enquanto este trabalho usa um sistema real mas sem envolvimento da organização, nem pretensão de utilizar os resultados; este trabalho não envolve uma equipe de DevOps. O diferencial é a etapa intermediária do processo para acelerar a disponibilização em dispositivos móveis.
Levcovitz, Terra e Valente (2016) criam, analisam e aplicam em um estudo de caso uma técnica para identificar e extrair *microservices*, usando critérios para identificar partes do sistema apropriadas à migração de arquitetura.	O estudo de caso utilizado pelos autores foi com um sistema bancário, enquanto neste trabalho utilizou-se um sistema acadêmico. Os dispositivos móveis também não foram abordados, contudo os autores utilizaram a identificação de critérios para classificar as partes do sistema legado em apropriadas ou não para a migração de arquitetura, em virtude principalmente de dependências de natureza transacional de operações bancárias.

3 ABORDAGEM PROPOSTA

3.1 Visão geral do processo

O "aprisionamento" tecnológico inerente a um sistema monolítico legado pode dificultar ou até mesmo impossibilitar o uso e a integração destes sistemas em outras plataformas, como os dispositivos móveis. Este capítulo, baseando-se nas referências teóricas abordadas, propõe um conjunto de etapas de alto nível para habilitar a integração dos sistemas monolíticos com outras plataformas por meio da utilização de *microservices*, utilizando-se também da aplicação da técnica de modernização *REST Wrapping* como etapa transitória em virtude das dificuldades e tempo necessários para realizar a quebra completa do sistema monolítico em partes menores. Por fim, o sistema terá sua arquitetura modernizada, propiciando novas possibilidades tecnológicas para sua evolução e manutenção. A Figura 19 ilustra as etapas do processo, que são detalhadas em seguida.

Figura 19 – Abordagem proposta.



3.2 Definir solução *mobile*

A escolha sobre o tipo de solução mobile que será utilizada deve levar em consideração principalmente a diversidade de tipos de dispositivos móveis que o público alvo do sistema possui e a intenção de alcance da organização. Em virtude das vantagens das aplicações nativas, principalmente relacionadas à experiência do usuário e responsividade, este é um ponto de partida interessante. Contudo, há a desvantagem de especificidade por plataforma, podendo demandar maior esforço de desenvolvimento.

As soluções híbridas e os *mobile-sites* têm um alcance maior em virtude de sua programação interna independente de plataforma, mas não possuem o mesmo nível de responsividade das aplicações nativas. Pode-se realizar uma análise interna ou aplicar pesquisa com o público alvo do sistema para identificar qual tipo de solução é a mais adequada: aplicação nativa, *mobile-sites* ou híbridos (WISNIEWSKI, 2011). Há casos em que não é fácil à organização identificar todos os tipos de dispositivos móveis e sistemas operacionais utilizados por seus usuários finais, como exemplo as instituições bancárias. Nestas situações é preferível utilizar uma solução *mobile* abrangente como os *m-sites* ou híbridos.

Conforme Wroblewski (2011), existem razões suficientes para que as organizações desenvolvam tanto soluções específicas por plataforma como *m-sites*: as aplicações nativas contam com o maior suporte de acesso aos recursos de *hardware* do dispositivo, podem ser disponibilizadas em lojas de aplicativos dos dispositivos, podem executar processos em *background* e contam com maior fluidez de *interface*; os *m-sites* permitem que usuários compartilhem links de conteúdos da aplicação, não necessitam que o usuário realize *download* de atualizações e podem ter seus conteúdos indexados em mecanismos de busca com facilidade. Agindo desta maneira os benefícios de ambas soluções poderão ser aproveitados, mas, considerando o maior esforço inicial para isso, é preciso começar com uma solução e gradativamente ampliar as opções.

Em resumo, para auxiliar na tomada de decisão do tipo de solução inicial a explorar, se a organização conhece o principal tipo de dispositivo móvel utilizado pelo público alvo e conta com o *know-how* de desenvolvimento nesta plataforma, pode escolher como solução móvel a aplicação nativa. Do contrário, deve começar por uma opção de maior alcance.

3.3 Identificar arquitetura atual

Entender a arquitetura existente é essencial para planejar e executar a modernização do sistema. Pode-se utilizar, como referências, documentações específicas sobre a arquitetura, caso existam, ou realizar uma análise do código legado. Em alto nível, as arquiteturas de aplicações monolíticas são semelhantes conforme já explorado: camadas de apresentação, negócio e banco de dados; contudo, conhecer as subdivisões destas camadas identificando o relacionamento com o código fonte será necessário para realizar as intevenções necessárias.

É comum que, em virtude do tempo em utilização do sistema legado, os documentos referentes à arquitetura sejam perdidos Zhang et al. (2009), demandando um esforço maior para desenhar a arquitetura do sistema com base no entendimento do código e sua estrutura. Para a iniciativa de Modernização Guiada à Arquitetura do OMG (Object Management

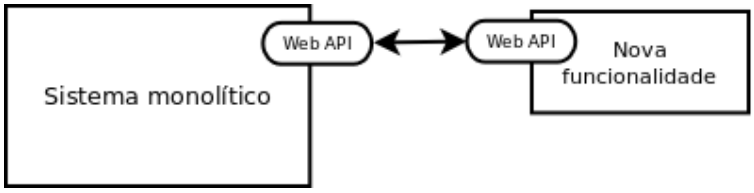
Group)[1] a representação da arquitetura existente é pré-condição para a modernização.

Portanto, é necessário que seja obtido ou construído o desenho arquitetural do sistema, que deve ser conhecido por todos os envolvidos com a sua modernização.

3.4 Parar evolução da arquitetura atual

É comum a demanda por evolução de sistemas de informação para oferecer novas funcionalidades a seus usuários. Se a demanda de evolução for passível de acesso *mobile*, esta não deve ser realizada seguindo a arquitetura monolítica existente, pois aumentaria o volume de trabalho de extração dos *microservices*. É a "lei dos buracos" referida por Richardson (2016): "Se estiver num buraco, pare de cavar". Durante o esforço de migração, caso haja necessidade de implementação de novas funcionalidades ou módulos, estes devem ser desenvolvidos como serviços independentes da aplicação principal, comunicando-se com elas por meio de *Web APIs*, que serão disponibilizadas no sistema legado no passo "Aplicar REST Wrapping", conforme exposto na Figura 20. Mesmo que a evolução não seja objetivo de acesso *mobile* por parte da organização, ela pode seguir a nova arquitetura com o objetivo de parar a evolução do sistema monolítico e aproveitar os benefícios da arquitetura de *microservices*.

Figura 20 – Evolução durante migração



Apesar desta orientação, é preciso observar que o novo serviço será também uma pequena aplicação monolítica, que deve ser posteriormente reanalisada em ciclos incrementais para diminuir a granularidade dos serviços se for necessário. Há outros aspectos a considerar, como a segurança na comunicação entre os sistemas, que será abordado em etapa posterior do processo.

3.5 Priorizar migração

A necessidade de escalabilidade é um dos critérios indicados na escolha das partes do sistema legado para iniciar a migração para microservices (RICHARDSON, 2016). Isto justamente porque ao migrar para *microservices* a escalabilidade individual será sobremaneira facilitada e menos onerosa. Contudo, como o foco deste trabalho é a disponibilização gradual para dispositivos móveis, é preciso analisar se faz sentido iniciar pelas

[1] http://www.omgwiki.org/admtf/doku.php

partes que demandam maior escalabilidade, ou se o público alvo do sistema almeja o acesso a outras partes do sistema. É provável que a organização já tenha conhecimento sobre qual parte de seu sistema é mais demandada para acesso *mobile*.

As partes que serão migradas devem ser identificadas bem como os seus relacionamentos. Com o desenho geral da arquitetura do sistema legado, deve-se buscar identificar os módulos da camada de negócio da aplicação. Com esta identificação, é possível definir a granularidade inicial dos futuros *microservices*. Para isto, este processo propõe os seguintes critérios:

1. Objetivos de migração baseados na demanda de utilização por dispositivos móveis - Idealmente, devem ser partes pequenas: iniciar o processo desta maneira facilitará o entendimento e a ambientação à nova arquitetura de forma incremental (RICHARDSON, 2016). Caso a organização não possua ou conheça esta demanda, este processo propõe uma análise organizacional sobre o que agregaria mais valor para o negócio:

    a) Selecionar partes do sistema que são gargalos de desempenho - desta forma, além de disponibilizar o acesso *mobile* para os usuários nesta parte do sistema, a parte migrada contará com o benefício adicional de desempenho à medida que a organização efetue a devida escalabilidade, que é facilitada na nova arquitetura (GONZALEZ, 2016; RICHARDSON, 2016).

    b) Realizar uma pesquisa com os usuários do sistema para identificar a demanda de acesso *mobile* prioritária (WISNIEWSKI, 2011).

    c) Funcionalidades mais utilizadas do sistema.

2. Agrupamentos de funcionalidades afins - agrupar um conjunto de funcionalidades altamente coesas em um único serviço no início pode evitar complicações iniciais na migração (CONANT, 2016);

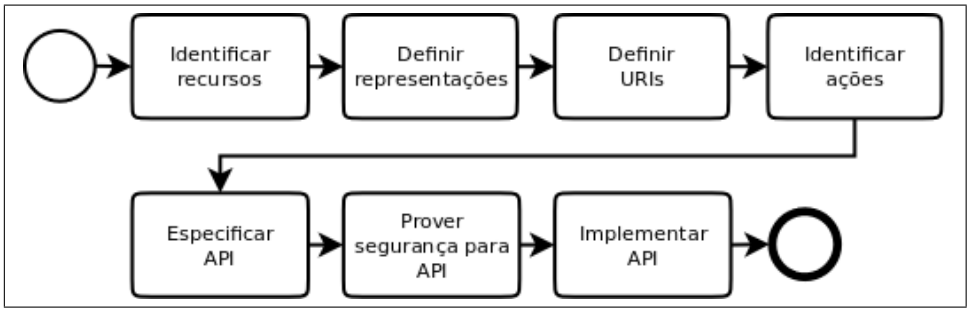
3. Restrições ocasionais em virtude da natureza transacional de alguma funcionalidade - as operações que exigem um nível imediato de consistência dos dados, em virtude do princípio da atomicidade da transação, não devem ser particionadas. É ocaso de operações bancárias, por exemplo (CONANT, 2016).

Como visto na fundamentação teórica, uma característica essencial dos microservices é a responsabilidade por uma única tarefa. Contudo, iniciar a migração com uma granularidade mínima dos serviços em algumas ocasiões pode ser dificultoso, pelas razões apresentadas anteriormente, de maneira que, nestes casos é preferível uma granularidade maior para posteriormente quebrar em partes menores. A utilização dos critérios apresentados objetivam alcançar um bom balanceamento de benefício e facilidade no início da migração.

3.6 Construir a API do sistema

Com a definição das partes do sistema que serão utilizadas inicialmente no processo, é possível identificar os recursos que serão acessados pelos dispositivos móveis. Desta forma, é preciso elaborar a API que receberá as requisições de acesso aos recursos, atuando de acordo com o padrão API *Gateway* (RICHARDSON, 2015). As etapas para construção da API estão dispostas na Figura 21. A especificação *OpenAPI*[2] pode ser utilizada tomando como base os recursos que serão acessados do sistema monolítico e o que deverá ser acessado pelos usuários dos dispositivos móveis.

Figura 21 – Subprocesso Construir API



3.6.1 Identificar Recursos

O principal conceito do estilo arquitetural REST é o de "recurso". Os recursos representam os objetos ou entidades de negócio, como produtos ou clientes por exemplo. É comum na atualidade construir APIs com o estilo REST utilizando predominantemente o protocolo HTTP (RICHARDSON, 2016). Esta etapa consiste em identificar conceitualmente os recursos inseridos no contexto do sistema legado que serão disponibilizados pela API.

Uma vez que a API irá intermediar o acesso *mobile* ao sistema legado num primeiro momento - porque posteriormente partes do sistema legado serão extraídas para *microservices* independentes - a identificação dos recursos da API é facilitada em virtude da análise realizada na etapa anterior do processo: as entidades de banco de dados ou objetos existentes no código fonte (no caso de sistemas desenvolvidos em linguagens orientadas a objetos) que o sistema disponibiliza em sua camada de negócio servirão de guia para esta etapa.

3.6.2 Definir Representações

A representação de um recurso na prática é o conjunto de dados trocado entre os clientes e os servidores de um sistema no estilo arquitetural REST, incluindo os metadados,

[2] https://www.openapis.org/

que informam como os dados devem ser interpretados além de conter informações extras dos recursos como criptografia e informações de validação (PURUSHOTHAMAN, 2015; VARANASI; BELIDA, 2015).

Uma API pode prover diferentes tipos de representações para diferentes tipos de clientes. Os *media-types* mais comuns das representações em REST APIs são (1)*application/xml* e (2)*application/json* (PURUSHOTHAMAN, 2015). Como o alvo deste processo é a integração com dispositivos móveis, que possuem restrições de armazenamento e desempenho de rede se comparados a *desktops*, o *media-type application/json* é indicado em virtude de seu menor tamanho ou *payload*, mas outros formatos podem ser necessários para fins diversos da organização.

3.6.3 Definir URIs

Após identificados os recursos e definida a representação que será disponibilizada, é preciso endereçar estes recursos para que sejam acessíveis pelos clientes. A RFC6570, conforme já visto, descreve padrões formais para o endereçamento de recursos usando variáveis ou *URI templates*. Adicionalmente, existem boas práticas ou convenções (MASSE, 2011) que tratam de aspectos não abordados pela norma, tais como:

- Utilizar substantivos em lugar de verbos - as ações sobre os recursos são definidas na própria requisição *http*, e não na URI. Desta forma, por exemplo, em lugar de usar "/consultarUsuarios", deve-se usar "/usuarios" com o verbo GET da requisição HTTP;
- Usar nomes no plural - mesmo que a URI especifique um recurso apenas, é indicado simplificar com o nome no plural ao invés de utilizar URIs distintas com singular e plural. Desta forma, considera-se que um recurso faz parte de uma coleção de recursos. Exemplo: "/usuarios/joao";
- Não indicar a extensão do arquivo na URI - o protocolo HTTP prevê o uso de um cabeçalho especificamente para indicar o tipo do conteúdo.

3.6.4 Identificar Ações

Da mesma forma que a identificação dos recursos se beneficia da análise dos dados disponibilizados pelo sistema monolítico, a identificação das ações que devem ser disponibilizadas sobre eles também o faz: as funções e métodos existentes na camada de negócio da aplicação são as ações em potencial. Mas é preciso restringir as operações com base nas permissões que o tipo de usuário *mobile* terá.

3.6.5 Especificar API

A especificação da API em linguagem neutra, independente de plataforma, além de facilitar sua documentação, pode ser utilizada por ferramentas que geram código automaticamente em linguagens específicas, tanto para o lado servidor quanto para o lado cliente. O *framework Swagger*[3] disponibiliza uma ferramenta chamada *Swagger Editor*, que permite a descrição de APIs usando a especificação *OpenAPI*[4], um padrão que surgiu a partir da *Swagger Specification*[5]. Este padrão utiliza um conjunto próprio de identificadores e valores possíveis que são descritos no formato YAML[6] ou JSON [7]. O Código 3.1 ilustra um exemplo de especificação *OpenAPI* disponibilizando um recurso chamado "users" utilizando o *media-type application/json* com mensagens específicas para os códigos de retorno HTTP 200 e 404.

Código 3.1 – Exemplo de descrição de API com a especificação OpenAPI

```
1  swagger: '2.0'
2  info:
3    title: Example API
4  host: host.example.com
5  schemes:
6    - https
7  basePath: /v1
8  produces:
9    - application/json
10 paths:
11   /users/{id}:
12     get:
13       summary: Users of the system
14       parameters:
15         - name: id
16           ...
17       responses:
18         200:
19           description: An user
20         404:
21           description: Not found
```

Ao especificar a API, a sua documentação é sobremaneira facilitada. Ferramentas como a *Swagger UI*[8] podem ser utilizadas para que, a partir da especificação, uma página

[3] Swagger - um *framework* que possui um conjunto de ferramentas para projetar, construir, documentar e consumir RESTful APIs. http://swagger.io/

[4] https://www.openapis.org/

[5] http://swagger.io/specification/

[6] YAML - *Ain't Markup Language* - é um padrão amigável de serialização e deserialização de dados. http://yaml.org/

[7] JSON - *JavaScript Object Notation* - é um formato leve para troca de dados. http://www.json.org/.

[8] http://swagger.io/swagger-ui/

de documentação com estilos prontos seja criada, permitindo sua disponibilização no próprio servidor que hospedará a API, por exemplo. A documentação é fundamental para que os desenvolvedores que trabalharão nas aplicações clientes saibam como requisitar os recursos oferecidos.

3.6.6 Prover segurança para a API

Para acessar os recursos de uma API, a aplicação cliente precisa ter autorização concedida pelo usuário do sistema. O *framework* OAuth 2.0 destaca-se como principal protocolo de autorização de Web APIs (BIHIS, 2015). Nesta etapa, é necessário disponibilizar um Servidor de Autorização para acessar a API conforme prevê o referido protocolo. A organização pode desenvolvê-lo, contratá-lo ou utilizar uma das várias opções de código aberto existentes.

Ao utilizar OAuth2, todas as requisições que chegarem à API deverão possuir um *access token*, que é uma chave de validade temporária gerada pelo Servidor de Autorização associada ao usuário que permitiu acesso aos seus dados. A API só deve aceitar a requisição se esta chave for válida e associada ao usuário relacionado com os dados que são requisitados.

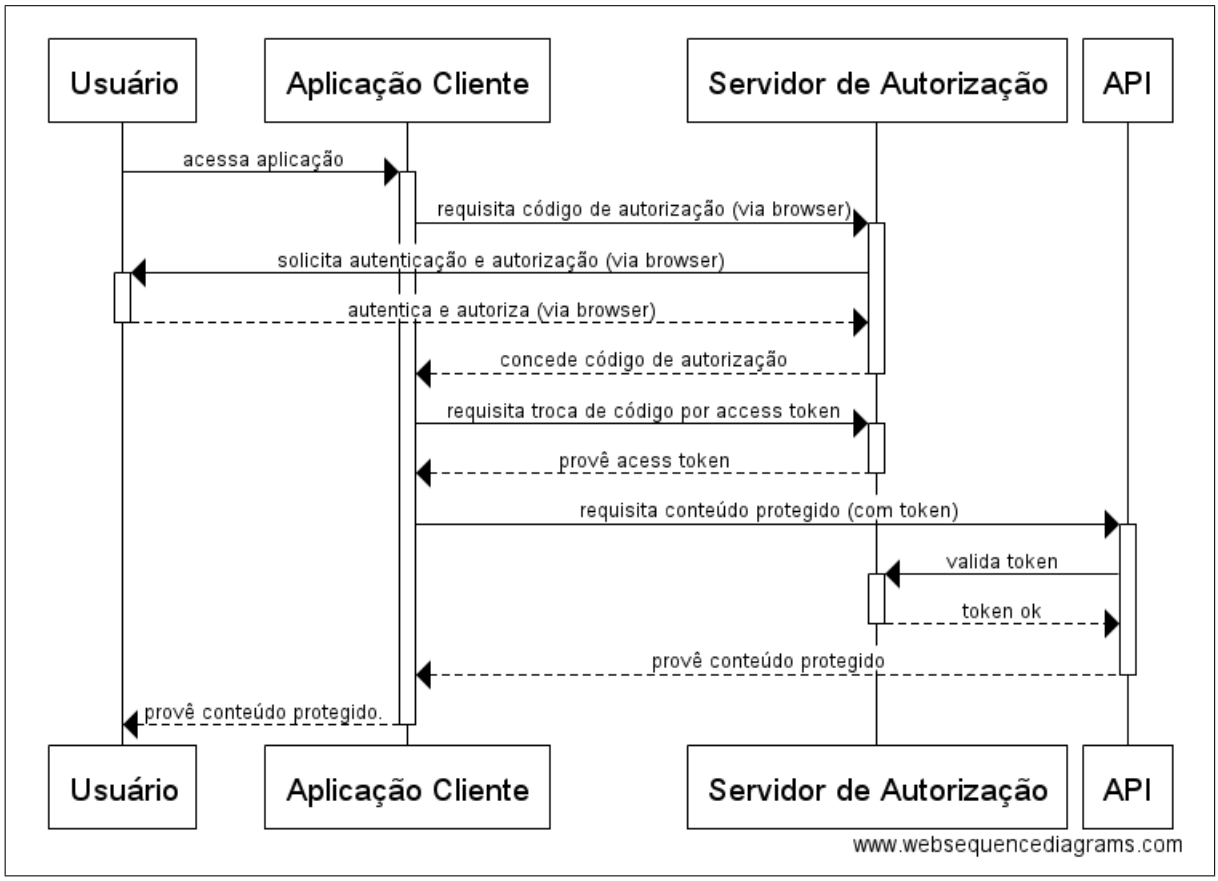
O fluxo de concessão de autorização sugerido por este processo, no caso de aplicativos nativos de dispositivos móveis, deve ser o de Concessão de Código de Autorização com interação entre o aplicativo e o navegador de internet do dispositivo, usando como *callback* o esquema de URI customizada provida pelo sistema operacional do dispositivo. Desta forma, o desenvolvimento de aplicações clientes pode ser estendido para além da equipe de desenvolvimento da organização que possui o sistema, pois as credenciais do usuário não serão de conhecimento da aplicação cliente. A Figura 22 ilustra o fluxo completo de autorização, desde o início de uso aplicação do dispositivo móvel à negociação entre a aplicação e o servidor de autorização *OAuth2* para conseguir o conteúdo protegido da API.

3.6.7 Implementar API

A geração de código automática para servidores baseada na especificação da API cria a base de sua implementação numa linguagem específica, contemplando basicamente o código necessário para receber as requisições HTTP nas URIs desejadas. Contudo, é preciso complementar esta implementação, desenvolvendo na linguagem escolhida o comportamento e as respostas para cada requisição recebida, além de outros aspectos como segurança, monitoramento e métricas, quando necessários.

Conforme visto, na arquitetura de *microservices*, as regras de negócio devem residir nos próprios serviços, portanto o comportamento a ser implementado para as requisições da API se restringirá essencialmente a garantir a segurança de acesso à informação e encaminhar as devidas requisições para os *microservices*, ou para o sistema monolítico

Figura 22 – Diagrama de sequência para autorizar acesso à API



legado, que num primeiro momento concentrará toda a implementação das regras de negócio. Um exemplo relacionado à garantia de segurança no acesso às informações por parte da API consiste na implementação de um filtro de autenticação, que é responsável por verificar se toda requisição possui em seu cabeçalho HTTP um *token* de acesso válido provido por um servidor de autorização.

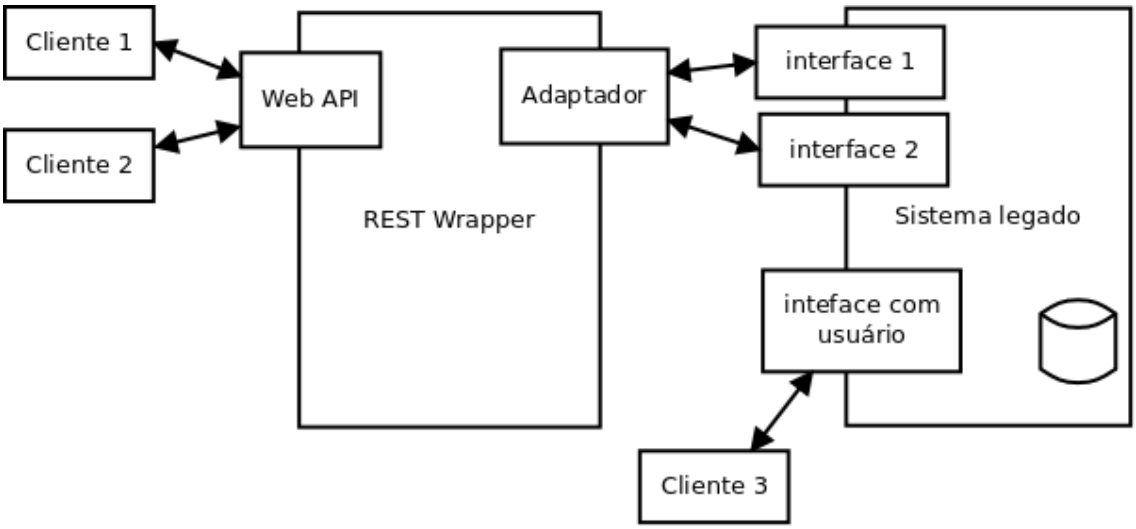
3.7 Aplicar REST *Wrapping*

A extração de partes do sistema monolítico em microservices pode ser um trabalho que demanda um grande período de tempo, a depender do tamanho da aplicação, do entendimento das regras de negócio implementadas, e principalmente do grau de acoplamento do código legado. Em virtude disto, e da emergente demanda por acesso *mobile*, é possível utilizar a técnica de REST *Wrapping* como etapa intermediária, possibilitando que a API consuma os recursos diretamente da aplicação monolítica, permitindo o imediato desenvolvimento de aplicações clientes voltadas para os dispositivos móveis enquanto os *microservices* são construídos.

Os detalhes de implementação da técnica dependem do sistema monolítico legado: sua plataforma, linguagem e arquitetura. Contudo, a ideia geral da técnica é permitir que

o sistema legado seja envolto por uma tecnologia que habilita acesso a suas funcionalidades ou dados por meio de uma API REST. A Figura 23 mostra a arquitetura geral da técnica, em que um software ou biblioteca também chamado de REST *Wrapper* intermedia comunicações provenientes de requisições à uma Web API com interfaces existentes no sistema legado por meio de um adaptador. De certa forma, este passo já provê a modernização necessária para que dispositivos móveis consumam recursos do sistema legado, contudo, o aprisionamento tecnológico, a dificuldade de manutenção e outros problemas inerentes ao sistema permanecerão existindo.

Figura 23 – Arquitetura da técnica REST Wrapping



Fonte: Adaptado de (KANGASAHO, 2016)

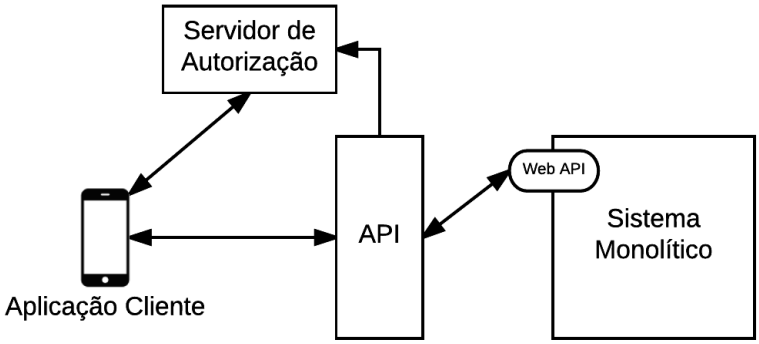
O trabalho de identificação de recursos realizado na construção da API deve ser utilizado como referência, expondo na aplicação legada os recursos necessários para que a API os consuma. As representações e URIs não necessariamente devem ser as mesmas utilizadas pela API, pois esta pode prover representações com menos dados em virtude das restrições de permissão dos usuários *mobile*.

3.8 Desenvolver solução mobile

Neste ponto, o sistema monolítico legado provê serviços intermediados por uma API, utilizando controle de acesso baseado no protocolo OAuth 2.0. Pode-se agora desenvolver os aplicativos para dispositivos móveis e finalmente obter o acesso desejado aos recursos do sistema em novas plataformas. Como dito anteriormente, é possível gerar automaticamente, a partir da especificação da API, bibliotecas de código que facilitam o consumo dos recursos providos por ela. Estas bibliotecas devem ser usadas pela aplicação cliente e alimentadas com os requisitos de segurança exigidos pela API, para que possam requisitar seus recursos. A Figura 24 ilustra uma visão geral da arquitetura dos componentes até então abordados,

simplificando a representação da técnica REST *Wrapping* aplicada ao sistema legado com uma Web API anexada.

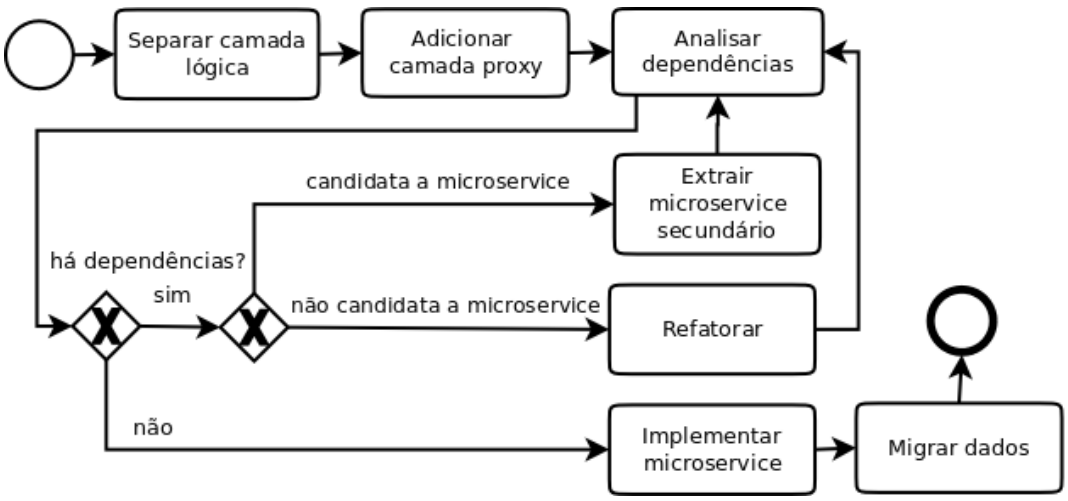
Figura 24 – Visão geral da arquitetura com REST Wrapping



3.9 Construir *microservices*

Considerando que o processo de modernização é do tipo *White Box*, sendo preciso conhecer detalhes da regras de negócio por meio de análise de código e documentação (WEIDERMAN et al., 1997), para que partes do sistema monolítico dêem lugar a microservices independentes, esta é a etapa mais difícil e longa do processo proposto. Diversos problemas podem ser encontrados, principalmente em relação à estrutura de armazenamento de dados da aplicação, uma vez que inclusive estas dependências deverão ser eliminadas já que os *microservices* terão seus próprios repositórios de dados. A Figura 25 exibe os passos para converter uma determinada parte do sistema legado em um *microservice*, realizando uma análise das dependências existentes para refatorar ou aplicar o processo de forma recursiva quando a dependência também deve ser convertida.

Figura 25 – Subprocesso para extrair *microservices*



3.9.1 Separar camada lógica

As partes do sistema que serão extraídas não contemplarão a interface com o usuário, e sim as regras de negócio e persistência de dados. Por isso, é preciso estabelecer, caso já não exista no sistema legado, uma separação clara entre o código das camadas de apresentação - ou interface com o usuário - e de regras de negócio (CONANT, 2016).

A complexidade desta etapa será basicamente determinada pelo nível de organização e acoplamento de código do sistema legado. Deve-se identificar os trechos que podem ser tratados em isolamento de acordo com os domínios do negócio, tanto na camada de negócio como na de persistência, caso já não estejam separados. É possível que o sistema já possua alguma organização de código que estabeleça relacionamento com as entidades de domínio de negócio, facilitando a identificação. Em aplicações que utilizam a linguagem Java, por exemplo, o código é organizado e estruturado por *packages*.

Contudo, mesmo que haja um nível de estruturação de código que facilite o entendimento e a identificação de entidades de negócio, pode ser que a organização não seja rigorosamente respeitada, havendo vazamento das regras de negócio para a camada de apresentação, o que pode dificultar o processo de separação e consequentemente a migração como um todo.

3.9.2 Adicionar camada *proxy*

À medida que os *microservices* forem desenvolvidos, será preciso fazer com que o sistema legado consuma-os. Para concentrar e padronizar o local de alteração para invocar os microservices dentro do sistema, pode-se utilizar uma camada *proxy* que receberá as solicitações da camada de apresentação e poderá requisitar à camada de negócio da aplicação ou encaminhar para um *microservice* (CONANT, 2016). Esta camada funcionará como uma fachada para os *microservices* dentro do sistema legado.

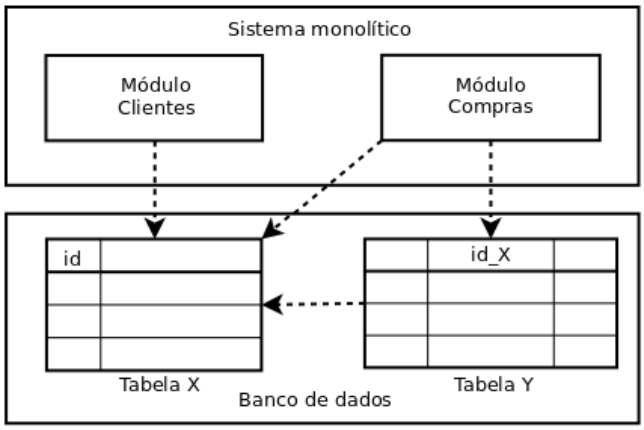
A ideia desta camada é similar ao padrão de projeto *Proxy*, em que uma classe funciona como uma interface para outra entidade. O próprio padrão de projeto pode ser utilizado nas classes do sistema legado caso este seja implementado em uma linguagem orientada a objetos.

3.9.3 Analisar dependências

A parte do sistema legado que será implementada como um *microservice* pode possuir dependências dentro do sistema monolítico tanto a nível de código como a nível de banco de dados: outras partes do sistema que fazem acesso diretamente aos seus dados ou funções, além de restrições ou referências internas no próprio banco de dados, como chaves estrangeiras ou *procedures* de banco. Estas dependências precisam ser desfeitas uma vez que a aplicação monolítica não possuirá mais o código e os dados relacionados,

que agora serão propriedade dos *microservices* (NEWMAN, 2015). Para acessar estes dados, o sistema monolítico deverá consumir seus respectivos *microservices*, que serão chamados especificamente pelo *proxy* da camada de negócio referente à parte extraída. A figura 26 ilustra um exemplo em que o módulo Compras do sistema monolítico consulta diretamente dados pertinentes ao módulo Clientes para obter informações do cliente ao invés de requisitar ao módulo respectivo, com base na chave estrangeira que possui.

Figura 26 – Acoplamento por chave estrangeira



Fonte: Adaptado de Newman (2015)

Cada dependência deverá ser analisada se pode ser uma candidata a *microservice* no contexto do que será migrado. O resultado da análise poderá culminar em:

1. Extração de *microservice* secundário - caso seja considerado que sua extração imediata trará valor de negócio no contexto da extração principal. A parte do sistema relacionada com a dependência deverá ser extraída de forma recursiva, utilizando o mesmo processo de extração;

2. Refatoramento - não sendo considerada candidata imediata à extração, a dependência deverá ser eliminada por meio de refatoramento do sistema monolítico: as chamadas aos dados ou funções da parte que será extraída devem ser modificadas para utilizar o *proxy* da camada de negócio referente à parte extraída. Em caso de restrição por chave estrangeira, esta deve ser eliminada no banco de dados, refatorando o esquema relacional. Se for uma dependência em função ou *procedure* chamada por outra parte do sistema, esta lógica deverá sair do nível do banco de dados e passar para a aplicação, que usará a camada *proxy* da parte extraída para obter os dados necessários.

É importante ressaltar que deve-se ponderar a possibilidade de desfazer as alterações do sistema monolítico com um adequado mecanismo de controle de versões do sistema, caso ocorram problemas com o *microservice* na nova arquitetura. Uma sugestão é identificar

a última versão estável antes das modificações, facilitando a volta do código alterado ou excluído.

3.9.4 Implementar *microservice*

Um requisito do tipo de modernização *White Box* é o entendimento das regras de negócio que foram implementadas, para sua reestruturação ou refatoramento (WEIDERMAN et al., 1997). Deve-se buscar inicialmente a documentação das partes que serão migradas com o intuito de obter um entendimento de alto nível. Também é importante conversar com pessoas que utilizam o sistema legado, em busca de informações não contempladas pela documentação. Por fim, é preciso realizar uma análise completa do código fonte das partes que serão extraídas. Com a clara especificação das regras de negócio que precisam ser implementadas, bem como do que deverá ser consumido pela API, os *microservices* podem ser construídos.

Quanto à tecnologia que será utilizada pelos *microservices*, relativa à linguagem de programação e banco de dados, não há nenhuma restrição relacionada com a tecnologia usada no sistema legado, qualquer opção pode ser escolhida pela equipe de desenvolvimento uma vez que a comunicação entre os *microservices* e o sistema legado ocorrerá por meio de requisições à Web APIs com o protocolo HTTP. Esta é uma das principais vantagens da arquitetura de microservices, deixando o sistema apto a utilizar inclusive tecnologias futuras que ainda serão desenvolvidas.

A parte extraída pode fazer referências a outras partes do sistema legado, que deverão ser expostas por uma *web api* usando a técnica *REST Wrapping*. Também há situações em que existem dados estáticos armazenados no banco que são compartilhados por toda a aplicação, como exemplo os códigos de telefonia dos Estados brasileiros. Nestes casos, em virtude da natureza predominantemente estática dos dados, deve-se ponderar se é melhor criar um *microservice* específico ou embutir estes dados em nível de código, usando arquivos de configuração ou bibliotecas, por exemplo.

Diferentemente de uma aplicação monolítica, em que a lógica de negócio fica toda concentrada no próprio sistema, a utilização de uma arquitetura de *microservices* tem a lógica de negócio espalhada entre eles, cada um com sua especificidade e conhecimento de negócio. Para realizar operações complexas normalmente será necessário haver interação entre os microservices. Em virtude disto, é preciso definir o tipo de comunicação entre processos que será utilizado: síncrono/assíncrono, um para um, ou um para muitos. Estas definições dependerão do contexto do negócio da aplicação, mas é possível que vários tipos de interação coexistam na arquitetura do sistema.

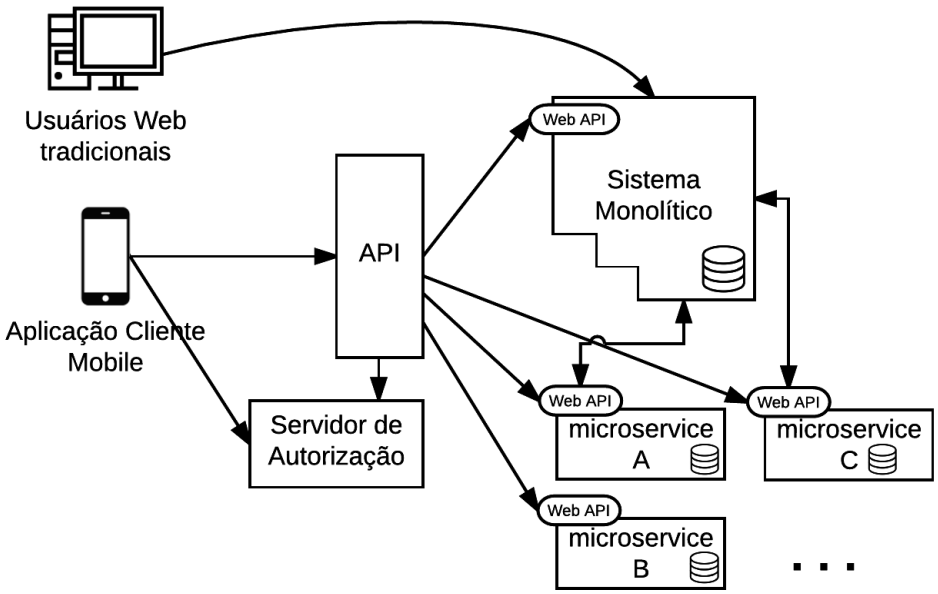
3.9.5 Migrar dados

Com a extração da lógica de negócio para o *microservice* finalizada, deve-se agora migrar os dados existentes no banco de dados monolítico para o banco de dados do serviço. Como as modelagens dos bancos não serão necessariamente iguais, é possível que seja necessário realizar as devidas conversões e transformações dos dados.

Neste ponto do processo de migração para a nova arquitetura, todas as dependências em nível de banco de dados do sistema monolítico em relação à parte extraída já devem ter sido eliminadas: chaves estrangeiras, *procedures*, funções, etc. Cabe ressaltar que a consistência destes dados distribuídos deverá ser garantida por um mecanismo de manutenção da consistência, que pode ser uma solução baseada em eventos: um serviço atualiza seus dados e notifica o evento a um *message broker*, que comunica a ocorrência do evento para outros serviços e por fim estes se atualizam. Esta é a consistência eventual dos dados.

A Figura 27 ilustra a arquitetura resultante, em que partes do sistema monolítico foram extraídas. O API *gateway* é responsável por intermediar o acesso mobile devidamente autorizado pelo Servidor de Autorização previsto no protocolo OAuth 2.0, consumindo recursos tanto da própria aplicação legada - que foi envolta pela técnica de REST Wrapping - quanto dos microservices existentes; enquanto que os clientes tradicionais da aplicação continuam executando o acesso direto a ela, por meio de seu controle de acesso existente. A aplicação legada atua como um *microservice* em relação à comunicação interna entre eles para obter os dados necessários.

Figura 27 – Visão geral da arquitetura resultante



3.10 Adaptar sistema legado e API

Concluída a extração da parte do sistema que foi convertida em um *microservice* independente, é necessário ajustar a camada lógica do sistema legado para utilizar o *microservice* criado em lugar de suas camadas internas, bem como a API para consumir os dados também do *microservice* em lugar da aplicação legada, de acordo com o contexto de negócio do que foi extraído.

Ao realizar este procedimento, todo o código existente nas camadas internas da aplicação legada pertinente ao que foi convertido em *microservice* se tornará inútil, poluindo o sistema (VISAGGIO, 2001). É preciso excluir este código com a devida cautela de um adequado controle de versões, para que o sistema legado possa voltar a ser utilizado juntamente com os dados migrados em caso de problemas com a nova arquitetura que justifiquem esta ação.

3.11 Considerações finais

A abordagem de modernização proposta neste capítulo é um misto da técnica de migração e *wrapping*, oferecendo de forma incremental agilidade na disponibilização do sistema legado em dispositivos móveis e a migração definitiva para uma nova arquitetura, a de *microservices*. Com isto, a demanda emergente por acesso mobile dos usuários é atendida ao passo que o sistema é modernizado para atender de forma mais eficiente, e sem restrições tecnológicas, as demandas futuras de evolução e manutenção.

A arquitetura de *microservices*, produto final do processo de modernização especificado, está alinhada com outra tendência tecnológica contemporânea: *DevOps*. Apesar de terem surgido e evoluído de forma independente, a arquitetura de *microservices* é entendida como necessária para a adoção de *DevOps*, porque não há sentido em automatizar os ciclos de entrega ou *deploy* de aplicações monolíticas, sendo impraticável em alguns casos (RV, 2016). Da mesma forma, as práticas *DevOps* são entendidas como pré-requisitos para adotar a arquitetura de *microservices* (VENNAM et al., 2015).

Os sistemas legados que já executam na Web, e em virtude disto podem ser acessados por meio de navegadores nativos dos dispositivos móveis, também podem ser modernizados com esta abordagem, considerando que não estão otimizados para as restrições destes dispositivos, especialmente o tamanho da tela. Esta não otimização demanda do usuário a ampliação (*zoom*) constante da tela, o que não promove uma boa experiência de uso.

4 ESTUDO DE CASO

O estudo de caso foi realizado em um sistema de gestão acadêmica universitária que está operacional há aproximadamente 15 anos, o SIGA[1] da UFPE (Universidade Federal de Pernambuco) desenvolvido na plataforma J2EE com a linguagem de programação Java e banco de dados Oracle 11g[2]. Originalmente o sistema visava atender apenas os cursos de graduação, registrando notas, frequência, matrículas, dentre outros aspectos relacionados. Contudo, com o passar do tempo, novos módulos foram adicionados, fazendo o sistema abranger também domínios de negócio relativos à pós-graduação, pesquisa, gestão de recursos humanos, planejamento estratégico, gestão patrimonial, dentre outros.

Em meados de 2015 foi iniciado um projeto para modernizá-lo abrindo o sistema para acesso *mobile* por meio de *REST Web APIs*. Até o presente momento, notas e informações dos discentes de graduação são consumidos pela aplicação cliente, além da Caderneta Eletrônica dos docentes, na qual os mesmos poderão utilizar os dispositivos móveis para informar a frequência, diário de classe e notas dos discentes.

Este estudo de caso não utiliza o produto deste projeto, uma vez que ele foi desenvolvido por outra equipe com seus próprios métodos. Para fins deste trabalho, todas as mudanças no sistema legado para habilitar os serviços REST foram implementadas desde o início, aproveitando-se apenas a biblioteca Java existente em virtude do projeto anterior. É preciso observar também que o projeto referido não contemplou a extração de *microservices* da aplicação legada, apenas sua abertura para consumo usando APIs REST.

4.1 Arquitetura do sistema

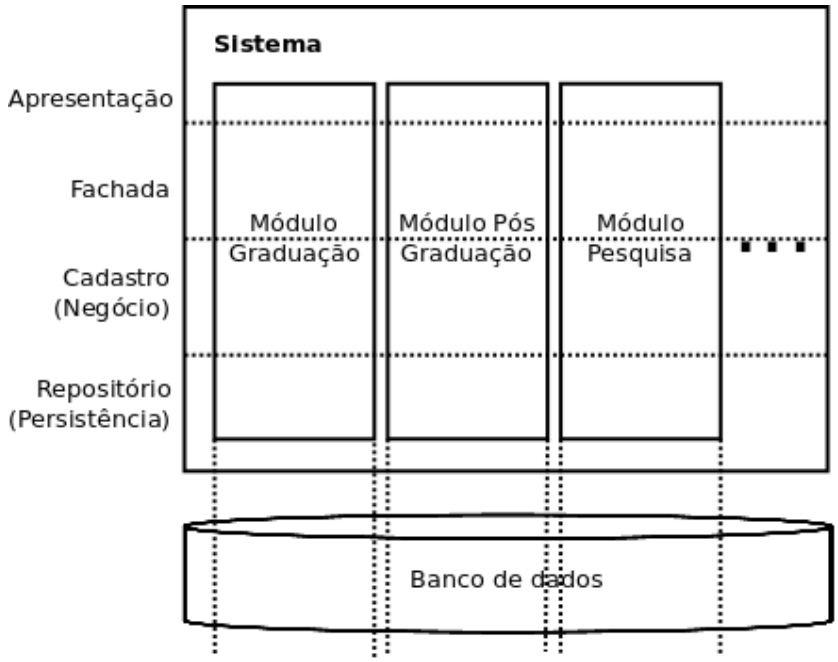
Como qualquer sistema monolítico, ele é uma aplicação única utilizando um único banco de dados. Internamente, a aplicação se divide nas camadas citadas a seguir e exibidas na figura 28.

- Apresentação - Responsável pela interface com o usuário. Há basicamente 3 tecnologias distintas utilizadas nesta camada: Java Server Pages - JSPs, Java Servlets e Java Server Faces - JSF.
- Fachada - São os controladores da aplicação, responsáveis por abrir conexões de banco de dados e garantir o aspecto transacional: confirmando ou cancelando as operações em caso de erros.

[1] http://www.siga.ufpe.br/
[2] http://www.oracle.com/technetwork/pt/database/enterprise-edition/overview/index.html

- Cadastro - Responsável pela implementação das regras de negócio.
- Repositório - A camada de acesso ao banco de dados, onde as consultas e outras operações de banco são concentradas.

Figura 28 – Arquitetura do estudo de caso.



4.2 Objetivos de acesso *mobile*

As partes escolhidas como objetivos de acesso *mobile* neste estudo de caso são as relacionadas às informações dos discentes de pós-graduação, uma vez que é interesse da organização e já foi feita iniciativa para os alunos da graduação conforme exposto no início do presente capítulo. As informações propriamente ditas são os conceitos obtidos pelos discentes e as notícias cadastradas no sistema, como prazos para realização de matrícula ou avisos sobre ocorrência de eventos acadêmicos. Esta escolha leva em consideração sugestão de iniciar com partes pequenas, facilitando o início da migração e a ambientação com a nova arquitetura. O tipo de solução *mobile* utilizado foi aplicação nativa, especificamente para dispositivos com sistema operacional Android.

Os conceitos obtidos pelos alunos de Pós-Graduação, assim como as notas de Graduação, são obtidos na fachada do sistema pelo *ControladorAcademico*. Este controlador chama métodos da classe *CadastroDadosHistoricos*, que por sua vez utiliza a camada de repositório usando a classe *RepositorioDadosHistoricos*. Ou seja, o código relacionado ao modelo de negócio referente aos conceitos obtidos está bem separado e organizado no sistema. A nível de banco de dados, a camada de repositório consulta uma tabela chamada *DADO_HIST*.

De forma análoga, o código responsável pelas notícias cadastradas no sistema também está bem separado e estruturado em todas as camadas. Contudo, há mais de uma implementação que representa ponto de acesso para as notícias em cada camada, em virtude da junção de uma subaplicação à aplicação principal. Não é possível eliminar um dos pontos sem efetuar refatoramento do código.

4.3 API discentePos

A API deste estudo de caso será referida como API discentePos e atuará como API *gateway* no acesso ao sistema legado e aos *microservices*. Com base no código do que foi selecionado como objetivo, identificaram-se os seguintes recursos: notícias e conceitos obtidos. Pelo código da aplicação em si, que originalmente atendia apenas ao nível de graduação, os conceitos obtidos pelos discentes são chamados de "dados históricos", mas considerando o contexto de négocio da pós-graduação e os objetivos deste trabalho, optou-se por nomear este recurso como "conceitos obtidos", por fazer mais sentido para os discentes de pós-graduação.

As representações dos recursos identificados utilizarão o *media-type* application/json. As propriedades das representações serão menores do que a quantidade de dados do modelo utilizado na aplicação monolítica, em virtude dos objetivos de consulta para o discente da pós-graduação.

A tabela abaixo resume os recursos disponibilizados pela API, bem como suas URIs e ações. Para o aplicativo cliente nos dispositivos móveis são permitidas apenas operações de consulta, por isso todos os recursos utilizam apenas o verbo HTTP GET. Posteriormente, os *microservices* extraídos contemplarão operações de inserção, atualização e deleção.

Tabela 3 – Recursos identificados para a API no estudo de caso.

Recurso	**URI**	**método HTTP**	**descrição**
noticias	/noticias	GET	Retorna todas as notícias cadastradas
noticias	/noticias/codigoNoticia	GET	Retorna uma notícia pelo seu código.
conceitosObtidos	/conceitosObtidos/matricula	GET	Retorna conceitos obtidos pelo discente.
userInfo	/userInfo	GET	Retorna informações do usuário que autorizou o acesso.

Foi utilizada a especificação OpenAPI 2.0 para descrever a API discentePos, com

uso da ferramenta *Swagger Editor*. A seguir são exibidos no Código 4.1 trechos com os principais *endpoints* da API, utilizados para obtenção dos conceitos obtidos pelo discente por parte do aplicativo cliente. A especificação completa da API está disponível no Apêndice A.

```
/conceitosObtidos/{matricula}:
  get:
    summary: Retorna conceitos obtidos pelo discente.
    description: Consulta todas as disciplinas cursadas pelo discente
       que possuem conceito atribuído.
    parameters:
      - name: matricula
        in: path
        description: Número de matrícula do discente.
        required: true
        type: integer
    responses:
      200:
        description: Sucesso
        schema:
          type: array
          items:
            $ref: '#/definitions/ConceitoObtido'
      404:
        $ref: "#/responses/RecursoNaoEncontrato"
      500:
        $ref: "#/responses/ErroNoServidor"
    security:
      - OauthSecurity: []
```

Código 4.1 – Especificação do endpoint da API para retornar os conceitos obtidos

A segurança no acesso à API é provida pelo protocolo OAuth 2.0. O site da especificação disponibiliza *links* para recursos relacionados, inclusive implementações de código aberto. Uma delas é a *APIs Secure Project*[3], que foi personalizada para atender este estudo de caso. As alterações realizadas no projeto estão relacionadas a utilização de banco de dados MySQL[4] em lugar de armazenamento em memória para persistir os códigos de autorização e *access tokens* concedidos, além de autenticar o usuário utilizando o banco de dados do sistema legado. O Apêndice B exibe o código responsável pelo filtro de autenticação da API: toda requisição deve possuir um *access token*, que é validado com o Servidor de Autorização.

Com a geração de código automática proporcionada pela ferramenta *Swagger*

[3] https://github.com/OAuth-Apis/apis
[4] https://www.mysql.com/

Editor, foi possível gerar a base da implementação do código do servidor: *endpoints* para recebimento das requisições e modelos. A linguagem de programação utilizada foi Java com a biblioteca *Jersey*[5], implementação de referência da especificação JAX-RS[6]. O Código 4.2 ilustra o código gerado, que delega a ação de resposta para outro objeto que deve ser implementado manualmente pelo desenvolvedor, para requisitar a informação protegida ao sistema legado.

```
1
2  @Path("/conceitosObtidos")
3  @Produces({ "application/json" })
4  public class ConceitosObtidosApi  {
5      private final ConceitosObtidosApiService delegate =
           ConceitosObtidosApiServiceFactory.getConceitosObtidosApi();
6      @GET
7      @Path("/{matricula}")
8      @Produces({ "application/json" })
9      public Response conceitosObtidosMatriculaGet(@PathParam("matricula")
           Integer matricula, @Context SecurityContext securityContext)
10           throws NotFoundException {
11       return delegate.conceitosObtidosMatriculaGet(matricula,
             securityContext); }
```

Código 4.2 – Código gerado em Java para acessar os conceitos obtidos. As anotações referentes à documentação foram omitidas

4.4 REST Wrapping do sistema legado

Em virtude de sua programação sobre a plataforma J2EE, habilitar o sistema legado com uma REST API é um processo facilitado pelas diversas bibliotecas Java existentes para este fim. Foi utilizada a biblioteca Jersey, implementação de referência da especificação JAX-RS. Novas classes foram criadas referentes aos recursos disponibilizados, NoticiasResource.java e ConceitosObtidosResource.java, que são responsáveis por acessar a camada de fachada do sistema para obter os dados, serializando-os de acordo com o modelo do recurso solicitado.

O Código 4.3 ilustra a implementação da classe ConceitosObtidosResource dentro do sistema legado, habilitando um *endpoint* "conceitosObtidos/matricula", que realiza uma chamada à fachada do sistema para consultar os conceitos obtidos de um discente específico.

[5] https://jersey.java.net/
[6] JAX-RS é uma especificação de API Java para RESTful Web Services. https://jax-rs-spec.java.net/

```
1  @Path("conceitosObtidos")
2  public class ConceitoObtidoResource {
3  ...
4  @GET
5  @Path("/{matricula}")
6  @Produces(MediaType.APPLICATION_JSON)
7  public Response consultarNotas(@PathParam("matricula") int matricula)
8   {
9   ArrayList<ConceitoObtido> listaConceitosObtidos =
10          new ArrayList<ConceitoObtido>();
11  ResponseBuilder resposta = null;
12  try {
13  RepositorioDadosHistoricoIteravel dadosHistoricos = this.sistema.
        getControladorAvaliacaoDiscente().
        consultarDadosHistoricoPorVinculoIteravel(matricula);
14          ...
```

Código 4.3 – Trecho da implementação do recurso ConceitoObtido no sistema legado.

4.5 App discentePos

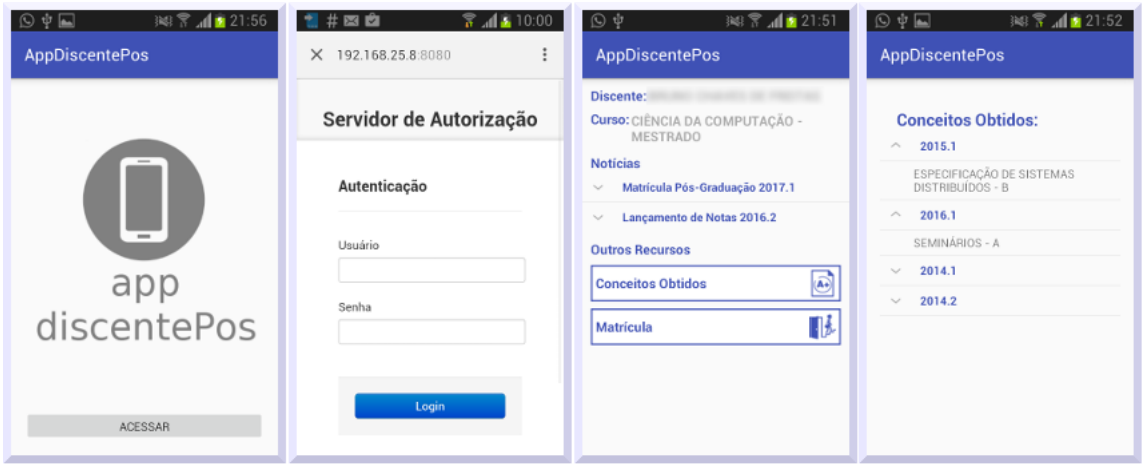
Análogo à API, o aplicativo cliente desenvolvido será chamado de App discentePos. Foi desenvolvido em Java utilizando a SDK Android 25. Neste ponto da aplicação do processo ao estudo de caso, o aplicativo foi desenvolvido para acessar a API, que obtém os recursos acessando diretamente o sistema legado por meio sua *Web API* recém-habilitada. Os objetivos do aplicativo consistem em:

1. Obter autorização de acesso aos recursos protegidos com o Servidor de Autorização;
2. Obter notícias em vigor cadastradas no sistema legado;
3. Obter conceitos obtidos do discente que autorizou o acesso.

Para conceder a autorização, a implementação realizada do Servidor OAuth disponibiliza o *endpoint* "/authorize", por meio do qual a aplicação cliente deve informar os parâmetros de URL referentes à identificação do cliente e a URI de retorno, previamente cadastrados no Servidor de Autorização. O app discentePos foi programado para acessar a URI de obtenção do código de autorização, que redireciona o usuário para o navegador padrão do dispositivo móvel e carrega a página de autenticação do servidor de autorização. A Figura 29 mostra o aplicativo em funcionamento: desde sua tela inicial, autenticação e exibição de conteúdo acessado.

A especificação da API também proporcionou a geração automática de uma biblioteca para consumir a API discentePos no aplicativo cliente. Basicamente esta biblioteca simplifica o acesso à URL e o provimento dos dados de segurança necessários para acessar a API. No Android a chamada à API é assíncrona: uma outra *thread* diferente da principal

Figura 29 – Aplicativo cliente nativo em dispositivo Android.



executa a conexão e obtém o retorno, chamando a função de retorno ou *callback*, que processa o resultado e exibe os dados para o usuário. O Código 4.4 mostra esta chamada assíncrona e também a verificação de existência da chave de acesso.

```
1  public class ConceitosObtidosActivity extends AppCompatActivity {
2
3   @Override
4   protected void onCreate(Bundle savedInstanceState) {
5  ...
6     ApiClient customClient = Configuration.getDefaultApiClient();
7
8     OAuth OauthSecurity = (OAuth) customClient.getAuthentication("
          OauthSecurity");
9     String access_key = AuthPersist.getInstance(getApplicationContext()).
          readAuthState().getAccessToken();
10    OauthSecurity.setAccessToken(access_key);
11
12    DefaultApi apiInstance = new DefaultApi(customClient);
13    ...
14    apiInstance.conceitosObtidosMatriculaGetAsync(matricula, new
          ApiCallback<List<ConceitoObtido>>() {
15    ...
```

Código 4.4 – Trecho da implementação da aplicação cliente em Android.

4.6 Extração dos *microservices*

De acordo com o processo proposto, é preciso delinear uma separação no código do sistema legado entre as camadas de interface com o usuário e as regras de negócio. No sistema estudado, esta divisão está formalmente estabelecida de acordo com a definição de sua arquitetura: as regras de negócio devem ser concentradas na camada de Cadastro. As

classes originais foram movidas para um pacote "legado", e novas classes, que substituirão as originais funcionando como uma camada *Proxy*, foram implementadas utilizando a mesma interface.

A análise das dependências do cadastro de notícias resultou em referências de outra partes do sistema, além de próprio Cadastro, à tabela ACS_NOTICIA do banco de dados: um mapeamento do *Hibernate*[7]. Por meio desta dependência encontrada, foi possível verificar que havia mais de uma fachada e cadastro no sistema legado para estes dados. Foi aplicada a mesma camada proxy a este cadastro em virtude dos objetivos deste trabalho, contudo esta situação ilustrou uma possibilidade de refatoramento retirando duplicidade de códigos no sistema legado.

O cadastro de notícias original é um conjunto de operações básicas para criar, consultar, atualizar e apagar dados. Todas as operações precisam ser implementadas no serviço, uma vez que tanto a aplicação cliente Android quanto o próprio sistema legado irão consumir este serviço. Para fins de exemplificação de um dos maiores benefícios da arquitetura de *microservices*, foi utilizada outra linguagem na implementação, *JavaScript* sobre a plataforma NodeJS[8], em virtude de sua popularidade contemporânea e bom suporte a serviços REST.

Em relação ao cadastro dos dados históricos, o modelo do sistema legado associado ao recurso de conceitos obtidos é mais complexo do que o de notícias. Em sua análise de dependências, foram localizadas 171 referências à tabela responsável pela persistência, DADO_HIST, de outras partes do sistema além do seu próprio cadastro. Cadastros de outras entidades como componente curricular, atividade acadêmica, avaliação docente, dispensa de disciplinas, etc., fazem acesso direto à tabela de dados históricos. Este é o caso em que deve-se primeiro eliminar as dependências antes de realizar a extração deste *microservice*. Neste estudo, a dependência referente à dispensa de disciplinas foi identificada como candidata a *microservice* secundário no contexto da extração principal.

Realizando o processo recursivo de análise de dependências sobre o cadastro de dispensas de disciplinas, foram encontradas 7 referências à tabela DISPENSA nos seguintes repositórios:

1. Repositório de Componentes Curriculares - um método chamado "consultarComponenteCurricularParaDelecao" buscava dispensas de disciplinas além de outros dados relacionados com o componente curricular a ser excluído. No caso da existência de dispensas, a exclusão não é permitida. Para contornar, a referência direta ao nome da tabela foi comentada e o código foi refatorado na camada de cadastro dos componentes curriculares para realizar previamente uma consulta ao cadastro de

[7] *Hibernate* é um *framework* de mapeamento objeto/relacional. http://hibernate.org/

[8] NodeJS é uma plataforma de execução de Javascript construída sobre o motor Javascript V8 do navegador de internet Chrome. https://nodejs.org/en/.

dispensas: se não houver dispensas cadastradas com o componente, o sistema legado chama o método referido para realizar as outras verificações. O Código 4.5 ilustra o refatoramento.

```
1  public final boolean consultarComponenteCurricularParaDelecao(int
       codigoComponenteCurricular)
2  {
3   //Verificando se existe dispensa com o componente que será deletado
4   boolean existeDispensa = false;
5   try {
6    cadastroDispensas.consultarDispensasPorParametrosIteravel(null, 0,
         null, codigoComponenteCurricular);
7    return false;
8   } catch (EntradaInexistenteException excecao) {
9    existeDispensa = false;
10  }
11  return this.getRepositorioComponentesCurriculares().
       consultarComponenteCurricularParaDelecao(
       codigoComponenteCurricular);
12 }
```

Código 4.5 – Refatoramento do cadastro de Componente Curricular para consultar o Cadastro de Dispensas ao invés de acessar sua tabela diretamente.

2. Repositório de Programas de Formação - havia um método chamado "consultarProgramaFormacaoDaDispensa" que resgatava o programa de formação relacionado ao componente curricular de uma dispensa. Tratava-se de um código morto, pois não é utilizado no sistema. O trecho foi comentado.

3. Respositório de Vínculos de Candidatos ao ENADE - Exame Nacional de Desempenho de Estudantes: o ENADE é um exame aplicado em nível nacional que avalia o desempenho dos estudantes de cursos superiores no Brasil. A inscrição dos estudantes que atendem aos critérios de participação, dentre os quais carga horária cursada mínima, é de responsabilidade das Instituições de Ensino Superior. O sistema legado, por meio das regras de negócio implementadas associadas a este repositório, fornece uma lista dos estudantes que devem ser inscritos baseado nos critérios devidos. Este trecho do repositório ENADE faz referência à tabela de dispensa de disciplinas para obter a carga horária dispensada do discente e somar à cursada, identificando assim os discentes que devem participar do processo. O refatoramento aqui se deu da mesma forma: a referência direta foi comentada e a camada de Cadastro do ENADE foi refatorada para chamar anteriormente o cadastro das dispensas, obter os dados necessários, e prosseguir com as demais verificações e processamentos.

4. HistóricoVínculo (classe à parte da estrutura de arquitetura do sistema): esta classe, chamada diretamente pela camada de fachada, é responsável pela montagem do histórico escolar dos estudantes. A referência à tabela DISPENSA está dentro de um laço e verifica se cada componente do histórico foi dispensado, para que em caso positivo seja capturada a carga horária dispensada. É um código morto, pois não é utilizada dentro da classe. Há um comentário indicando que a carga horária que deve ser exibida no histórico escolar é a do próprio componente curricular, independente se está associado a uma dispensa e da carga horária dispensada concedida.

5. ConsultarIntegralização (classe à parte da estrutura de arquitetura do sistema): responsável por computar os requisitos de conclusão de curso pelo discente: carga horária cumprida, componentes curriculares obrigatórios, eletivos, etc. A soma das carga horárias das dispensas obtidas pelo discente é considerada nesta integralização, por isso esta classe consulta a tabela DISPENSA. O trecho em questão foi substituído por uma consulta ao Cadastro de Dispensas.

6. Repositório de Vínculos de Discentes: há um método chamado "deletarVinculoEmCascata" que varre todas as dependências de um vínculo de discente e as exclui antes de excluir o vínculo propriamente dito. Uma destas dependências é o registro da tabela DISPENSAS. O trecho foi substituído por uma chamada ao cadastro de Dispensas.

7. Arquivos HandlerPesquisarDispensa.jsp, HandlerTelaAlterarDispensaInterna.jsp e HandlerAlterarDispensaExterna: todos estes arquivos JSP fazem referência direta à tabela de Dispensa sem chamar a fachada do sistema. Todos eles são código morto, pois não são utilizados no sistema. Foram comentados.

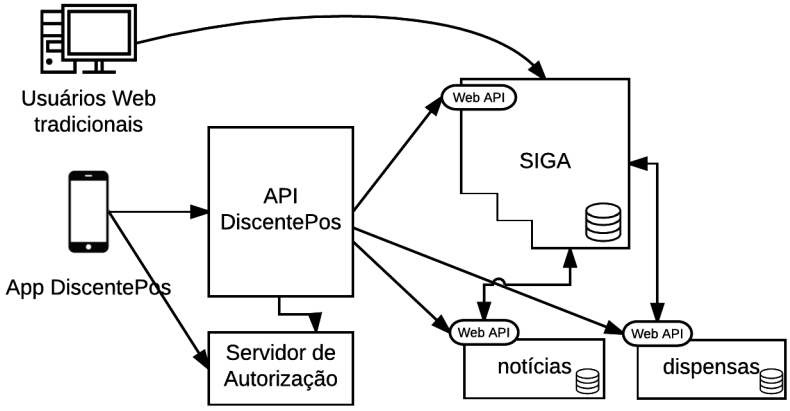
Após realizado o refatoramento, foi implementado o *microservice* secundário de Dispensas de Disciplinas com a linguagem *JavaScript* na plataforma NodeJS. Esta foi uma das 171 dependências da tabela relacionada aos Dados Históricos. Para realizar esta extração, todas as demais dependências deveriam ser analisadas e refatoradas ou extraídas conforme prescreve o processo proposto, mas este trabalho extrapola o interesse deste estudo de caso, que consiste em aplicar e validar em um escopo limitado da aplicação, e não realizar todo o trabalho manual, uma vez que demandaria muito tempo e uma equipe com várias pessoas envolvidas.

Para finalizar, após a extração dos *microservices* de notícias e dispensas de disciplinas, foi preciso adaptar a API para consumir os recursos a partir dos serviços extraídos em lugar do sistema legado, e a camada *Proxy* para buscar os *microservices* em lugar da camada de cadastro legada, encerrando a aplicação do processo.

4.7 Resultados

Como resultado da aplicação do processo proposto no sistema do estudo de caso dois *microservices* foram extraídos, referentes à gestão de notícias e dispensas de disciplinas para discentes. O sistema legado foi devidamente modificado em sua camada de cadastro para consumir estes serviços, assim como a API discentePos. Um aplicativo para dispositivos móveis foi desenvolvido para o sistema operacional Android. A arquitetura final após a aplicação do processo está representada na Figura 30.

Figura 30 – Arquitetura após aplicação do processo proposto.



Os microservices foram criados com a linguagem de programação JavaScript sobre a plataforma NodeJS, para demonstrar uma das principais vantagens desta arquitetura: a liberdade de tecnologia utilizada. O banco de dados de cada serviço também foi distinto do banco de dados do SIGA: o MySQL.

Como o *microservice* de dispensas possui uma regra de negócio que envolve a carga horária da disciplina dispensada, foi preciso habilitar no sistema legado uma *Web API* secundária para consultar os componentes curriculares, a fim de obter a carga horária original do componente curricular referente à disciplina dispensada.

Referente à obtenção de informações do usuário que realiza autenticação no dispositivo móvel, especificamente nome, matrícula e curso, o servidor de autorização foi programado para adicionar estas informações aos registros dos códigos de autorização concedidos. A API DiscentePos, por meio do *endpoint* "/userInfo", consulta os dados referentes ao código utilizado no servidor de autorização e obtém estas informações de usuário. Esta abordagem foi utilizada em virtude do escopo restrito deste trabalho. Contudo, seria mais adequado que o servidor de autorização fosse o mais genérico possível ao associar um código de autorização a um usuário, pois este pode possuir diversos papéis ou perfis dentro do sistema, como discente, docente ou funcionário. Cabe à API do sistema prover informações dentro de contextos específicos.

Por ser um sistema desenvolvido em J2EE, o sistema SIG@ é um sistema Web e pode ser acessado por navegadores de internet de quaisquer dispositivos computacionais,

incluindo os móveis. Contudo, não há uma versão *mobile* do sistema, de forma que a mesma interface apresentada em computadores pessoais é também utilizada em dispositivos móveis, demandando constantes operações de ampliação (*zoom*) dos usuários. Neste sentido, a vantagem das modificações realizadas por este trabalho proporcionaram uma experiência de uso mais adequada por meio de uma aplicação nativa do sistema operacional *Android*.

Avaliando a aplicação do processo foi possível perceber as dificuldades que podem ser encontradas na extração dos *microservices*: múltiplas fachadas existentes, acesso direto ao banco de dados por camadas indevidas, acesso a tabelas de banco de dados por partes de domínios de negócio distintos, etc. Como a extração em si já demanda o entendimento das regras de negócio implementadas, o que pode levar tempo, estas características dificultaram ainda mais o processo, além de adicionar riscos de problemas em virtude dos refatoramentos. Como lição aprendida, observa-se que o processo poderia classificar partes como boas ou más candidatas à extração para arquitetura de *microservices* em virtude destas dificuldades.

5 CONCLUSÕES

Prover o adequado acesso a sistemas de informação por meio de dispositivos móveis é mandatório: uma demanda criada pelos próprios usuários conforme constatado em estatísticas (WROBLEWSKI, 2011). Os sistemas legados podem não estar aptos a esta tendência em virtude de sua tecnologia defasada. O processo de habilitar estes sistemas ao acesso via dispositivos móveis pode ser uma oportunidade de modernização. Neste contexto, a arquitetura de *microservices* é uma tendência para reestruturar sistemas monolíticos, quebrando amarras tecnológicas e abrindo portas para novas demandas de evolução atuais e futuras.

Como visto, a arquitetura de *microservices* surgiu a partir de práticas em comum de equipes de tecnologia da informação como uma forma revolucionária de "quebrar" grandes sistemas em mínimos serviços com foco único e totalmente independentes entre si, comunicando-se por mensagens. Estas práticas foram proporcionadas pelo avanço de outras tecnologias relacionadas, principalmente a virtualização. Contudo, este processo de "quebra" do sistema legado pode ser bastante demorado, em virtude essencialmente do acoplamento de código existente e do tamanho do sistema. O tempo necessário pode ser um empecilho para as demandas cada vez mais crescentes de acesso *mobile* a estes sistemas. Para contornar esta questão, este trabalho propôs uma etapa intermediária de modernização, utilizando a técnica de REST *Wrapping*, de maneira que a integração dos sistemas legados com os dispositivos móveis é priorizada. Por fim, o processo culmina com a extração de partes do sistema legado que são convertidas para a arquitetura de *microservices*, tornando o sistema apto a utilizar quaisquer tecnologias de linguagem de programação ou banco de dados.

Os sistemas legados persistem nas organizações em virtude de sua criticidade e importância, de maneira que interromper o funcionamento destes sistemas normalmente ocasiona sérios problemas na execução dos processos de negócio das organizações. Em virtude disto, a modernização de um sistema legado não pode interromper abruptamente seu funcionamento, deve ser transparente para seus usuários. Neste sentido, o processo proposto neste trabalho mostrou-se alinhado com esta perspectiva de continuidade de funcionamento enquanto em paralelo esforços de modernização são gradativamente desempenhados. O débito técnico existente em um sistema legado, acumulado no decorrer dos anos com manutenções emergenciais para atender solicitações constantes e urgentes por ajustes ou novos requisitos, pode ser gradativamente eliminado com as etapas especificadas de migração de arquitetura, ao passo em que o sistema continuará operacional atendendo as demandas organizacionais e disponibilizando o acesso *mobile* para seus usuários.

As técnicas de modernização de sistemas identificadas na pesquisa foram o redesenvolvimento, o *wrapping* e a migração. O redesenvolvimento, conforme estudado, é uma técnica arriscada porque demanda uma completa recriação do sistema, sendo portanto de alto custo em todos os aspectos, inclusive em relação ao tempo necessário. O *wrapping* se mostra como a técnica mais rápida, simples e de fácil implementação. Contudo, nesta técnica, o sistema legado permanecerá existindo com todo o conjunto de tecnologia legada que utiliza, dificultando futuras manutenções em suas funcionalidades. Já a migração, em virtude de sua natureza incremental e a possibilidade de ser adotada em conjunto com a técnica de *wrapping*, se mostra como uma alternativa que a longo prazo traz um maior benefício.

Neste contexto, foi possível identificar o tempo como um fator importante na escolha da técnica de modernização de um sistema legado: quanto menor o tempo disponível para alcançar os objetivos de modernização, mais indicada é a técnica de *wrapping*. Porém, a manutenção do sistema legado também deve ser considerada um fator significativo, pois quanto maior é a demanda de manutenção, maior é o estímulo para adotar a técnica de migração, que de forma incremental irá mover o sistema legado para uma nova tecnologia, facilitando futuras manutenções.

A pergunta de pesquisa definida neste trabalho busca uma forma de modernizar um sistema legado para que seja acessível por dispositivos móveis frente às dificuldades impostas pela tecnologia legada e a necessidade de continuidade de funcionamento do sistema, que por sua natureza é crítico para a organização. Como resposta à esta questão, este trabalho conclui que a adoção mista das técnicas de *wrapping* e migração com o encadeamento lógico das ações propostas proporciona o atendimento à demanda emergente de acesso *mobile* para sistemas de informação enquanto gradativamente migra o sistema para uma nova arquitetura, facilitando a sua evolução e a adoção de futuras tecnologias de desenvolvimento. Este trabalho utiliza como principal técnica a migração, uma vez que o resultado final da aplicação é uma parte do sistema legado em nova arquitetura; mas, para priorizar o acesso por dispositivos móveis, como já dito, o processo faz uso da técnica de *wrapping* como etapa intermediária.

As dificuldades e riscos pertinentes à aplicação do conjunto de ações especificadas não foram plenamente abordados. A análise das dependências poderia ser mais ampla de maneira a abordar estes aspectos para criar um mapeamento das dependências e estabelecer critérios que auxiliem na decisão por migrar ou não uma determinada parte do sistema para a nova arquitetura. Considerando que o acesso *mobile* já é possível com a etapa intermediária de REST *Wrapping*, os prós e contras de continuar a extração de um determinado *microservice* do sistema legado devem ser cuidadosamente avaliados pela organização, pois na nova arquitetura baseada em *microservices* o sistema legado poderá continuar existindo, funcionando como uma espécie de macro serviço. Contudo,

todas as suas dificuldades, especialmente relacionadas a manutenção, permanecerão. A implementação de novas funcionalidades, no entanto, sendo feita com base na nova arquitetura, terá toda a liberdade tecnológica dos *microservices*, comunicando-se tanto com o sistema legado quanto com os *microservices* existentes.

Como a execução das atividades propostas por este trabalho não teve apoio de uma equipe da organização detentora do sistema utilizado no estudo de caso, ou seja, não foi um projeto da organização e sim uma iniciativa individual que foi autorizada a utilizar o sistema, todas as atividades foram desempenhadas por uma única pessoa, de maneira que não é possível extrair lições aprendidas de diferentes atores de uma equipe envolvida com a aplicação do processo.

Este trabalho abordou aspectos essenciais para habilitar o acesso *mobile* e quebrar o sistema legado em *microservices*, contudo, não contemplou a possibilidade de interrupção da extração em virtude das possíveis dificuldades encontradas relacionadas a dependências existentes, como no estudo de caso apresentado. Constata-se que é preciso ponderar se o esforço necessário para eliminar dependências relacionadas a uma determinada parte do sistema compensa o benefício de tê-la executando na arquitetura de microservices, uma vez que pode permanecer no sistema legado coexistindo com a nova arquitetura. Por isso o estabelecimento de critérios para identificar partes que são boas candidatas à extração é importante.

5.1 Limitações

Diversos aspectos inerentes à arquitetura de *microservices* não foram abordados neste trabalho, por não fazerem parte do seu escopo. Contudo, ao adotar esta arquitetura, é fundamental observá-los:

- Descoberta dos serviços - Os *microservices* possuem URLs, a API precisa ter conhecimento destas URLs para requisitar os serviços. Estas URLs podem ser codificadas na própria API, contudo, num contexto de múltiplos microservices, manter o controle manual dos endereços não é viável. Este aspecto também tem relação com o balanceamento de carga dos microservices. Existem diferentes mecanismos de descoberta de serviço, que fazem uso de um registro dos serviços existentes na aplicação.
- Segurança - Utilizando o API Gateway com OAuth 2.0 o acesso externo à aplicação terá a segurança de acesso, contudo, internamente os microservices também podem implementar algum nível de segurança no acesso. Uma das possibilidades é a utilização de assinaturas nos cabeçalhos HTTP utilizando JSON Web Tokens - JWT[1].

[1] https://jwt.io/

- Estratégia de *Deploy* - A técnica de virtualização já era utilizada em sistemas monolíticos, e também pode ser para os *microservices*, que além desta possibilidade contam com a técnica de *containers*: uma evolução da virtualização no nível do sistema operacional, mais leve, que permite a execução de processos com recursos isolados dentro de um mesmo sistema operacional (RICHARDSON, 2016). A estratégia de deploy também deve considerar a quantidade de instâncias do serviço por servidor.
- Gerenciamento dos dados distribuídos - Consistência eventual dos dados é uma característica inerente aos microservices, em virtude disto é preciso definir o mecanismo de manutenção da consistência dos dados distribuídos. Normalmente este problema é solucionado com a utilização de eventos: um *message broker* é utilizado para receber e publicar a ocorrência de eventos para outros serviços, que podem agir por si mesmos parar atingir a consistência de seus próprios dados.

5.2 Trabalhos futuros

A principal deficiência encontrada no processo proposto quando foi aplicado ao estudo de caso, conforme já discutido nas considerações finais, foi a ausência de um mapeamento das dependências apontando critérios para identificar quais partes do sistema não são boas candidatas à migração para arquitetura de *microservices*, como no trabalho realizado por Levcovitz, Terra e Valente (2016). Esta é uma indicação de trabalho para otimizar o processo proposto. Além disto, também pode-se considerar a abordagem dos aspectos citados na seção anterior, principalmente os tópicos referentes ao gerenciamento dos dados distribuídos, que podem representar um grande desafio no processo de migração do sistema legado; e a segurança na comunicação entre os *microservices*.

REFERÊNCIAS

APPOLINÁRIO, F. *METODOLOGIA DA CIÊNCIA: Filosofia e prática da pesquisa ? 2ª edição revista e atualizada.* 2. ed. [S.l.]: Cengage Learning, 2012. 240 p. ISBN 9788522111770. Citado na página 16.

ASHMORE, D. C. *Microservices for Java EE Architects: Addendum for The Java EE Architect's Handbook 1.* [S.l.]: DVT Press, 2016. Citado na página 35.

BAGHDADI, Y.; AL-BULUSHI, W. A guidance process to modernize legacy applications for SOA. *Service Oriented Computing and Applications*, v. 9, n. 1, p. 41–58, 2015. ISSN 1863-2394. Disponível em: <http://dx.doi.org/10.1007/s11761-013-0137-3>. Citado na página 30.

BAILIS, P.; GHODSI, A. Eventual Consistency Today: Limitations, Extensions, and Beyond. *Commun. ACM*, v. 56, n. 5, p. 55–63, 2013. Disponível em: <http://doi.acm.org/10.1145/2447976.2447992>. Citado na página 37.

BALALAIE, A.; HEYDARNOORI, A.; JAMSHIDI, P. Migrating to Cloud-Native Architectures Using Microservices: An Experience Report. In: CELESTI, A.; LEITNER, P. (Ed.). *Advances in Service-Oriented and Cloud Computing: Workshops of ESOCC 2015, Taormina, Italy, September 15-17, 2015, Revised Selected Papers.* Cham: Springer International Publishing, 2016. p. 201–215. ISBN 978-3-319-33313-7. Citado 3 vezes nas páginas 40, 53 e 54.

BERNERS-LEE, T.; BRAY, T.; CONNOLLY, D.; COTTON, P.; FIELDING, R.; JECKLE, M.; LILLEY, C.; MENDELSOHN, N.; ORCHARD, D.; WALSH, N.; WILLIAMS, S. *Architecture of the World Wide Web, Volume One.* W3C, 2004. Disponível em: <https://www.w3.org/TR/2004/REC-webarch-20041215>. Citado na página 43.

BERNERS-LEE, T.; FIELDING, R.; IRVINE, U.; MASINTER, L. *RFC2396: Uniform Resource Identifiers (URI) Generic Syntax.* 1998. Disponível em: <https://www.ietf.org/rfc/rfc2396.txt>. Citado na página 44.

BIEBER, M.; VITALI, F.; ASHMAN, H.; BALASUBRAMANIAN, V.; OINAS-KUKKONEN, H. Fourth generation hypermedia: some missing links for the World Wide Web. *International Journal of Human-Computer Studies*, v. 47, n. 1, p. 31–65, 1997. ISSN 1071-5819. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1071581997901300>. Citado na página 42.

BIEHL, M. *API Architecture: The Big Picture for Building APIs.* 1. ed. [S.l.]: CreateSpace Independent Publishing Platform, 2015. 192 p. ISBN 150867664X. Citado na página 41.

BIHIS, C. *Mastering OAuth 2.0.* Packt Publishing, 2015. ISBN 9781784395407. Disponível em: <https://www.safaribooksonline.com/library/view/mastering-oauth-20/9781784395407/>. Citado 3 vezes nas páginas 50, 51 e 62.

BIRCHALL, C. *Re-Engineering Legacy Software.* Manning Publications, 2016. Disponível em: <https://www.safaribooksonline.com/library/view/re-engineering-legacy-software/9781617292507/>. Citado na página 35.

BISBAL, J.; LAWLESS, D.; WU, B.; GRIMSON, J. Legacy Information Systems: Issues and Directions. *IEEE Softw.*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 16, n. 5, p. 103–111, 1999. ISSN 0740-7459. Disponível em: <http://dx.doi.org/10.1109/52.795108>. Citado 2 vezes nas páginas 26 e 27.

BLOOMBERG, J. *The Agile Architecture Revolution: How Cloud Computing, REST-Based SOA, and Mobile Computing Are Changing Enterprise IT*. John Wiley & Sons, 2013. ISBN 9781118409770. Disponível em: <https://www.safaribooksonline.com/library/view/the-agile-architecture/9781118417874/>. Citado 4 vezes nas páginas 30, 43, 44 e 47.

BOHIUM, K. Responsive Web Design, Discoverability, and Mobile Challenge. *Library Technology Reports*, August-Sep, 2013. Disponível em: <https://journals.ala.org/index.php/ltr/article/viewFile/4507/5286>. Citado na página 23.

BOJINOV, V. *RESTful web API design with Node.js : design and implement efficient RESTful solutions with this practical hands-on guide*. Packt Publishing, 2016. ISBN 9781786469137. Disponível em: <https://www.safaribooksonline.com/library/view/restful-web-api/9781786469137/>. Citado na página 41.

BOYD, R. *Getting started with OAuth 2.0*. O'Reilly Media, Inc., 2012. ISBN 9781449311605. Disponível em: <https://www.safaribooksonline.com/library/view/getting-started-with/9781449317843/>. Citado na página 51.

BRODIE, M. L.; STONEBRAKER, M. *Legacy Information Systems Migration: Gateways, Interfaces, and the Incremental Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995. ISBN 1-55860-330-1. Citado 2 vezes nas páginas 15 e 25.

COMELLA-DORDA, S.; WALLNAU, K.; SEACORD, R.; ROBERT, J. *A Survey of Legacy System Modernization Approaches*. Pittsburgh, PA, 2000. Disponível em: <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=5093>. Citado 2 vezes nas páginas 25 e 26.

CONANT, S. *Learning Path: Issues and Next Steps in Software Architecture*. 2016. Disponível em: <https://www.safaribooksonline.com/library/view/learning-path-issues/9781491960967/>. Citado 2 vezes nas páginas 58 e 66.

DELGADO, J. A Service-Based Framework to Model Mobile Enterprise Architectures. In: *Handbook of Research on Mobility and Computing*. [s.n.], 2011. Disponível em: <https://www.safaribooksonline.com/library/view/handbook-of-research/9781609600426/978-1-60960-042-6.ch053.xhtml>. Citado na página 19.

DENISS, W.; BRADLEY, J. *OAuth 2.0 for Native Apps - draft-ietf-oauth-native-apps-06*. 2016. Disponível em: <https://tools.ietf.org/html/draft-ietf-oauth-native-apps-06>. Citado 2 vezes nas páginas 51 e 52.

Di Nitto, E.; GHEZZI, C.; METZGER, A.; PAPAZOGLOU, M.; POHL, K. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, v. 15, n. 3, p. 313–341, 2008. ISSN 1573-7535. Disponível em: <http://dx.doi.org/10.1007/s10515-008-0032-x>. Citado na página 27.

DRAGONI, N.; GIALLORENZO, S.; LAFUENTE, A. L.; MAZZARA, M.; MONTESI, F.; MUSTAFIN, R.; SAFINA, L. Microservices: yesterday, today, and tomorrow. 2016.

Disponível em: <http://arxiv.org/abs/1606.04036>. Citado 3 vezes nas páginas 17, 27 e 32.

ERL, T. *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005. ISBN 0131858580. Citado 3 vezes nas páginas 28, 29 e 31.

ERL, T.; CHELLIAH, P.; GEE, C.; KRESS, J.; MAIER, B.; NORMANN, H.; SHUSTER, L.; TROPS, B.; UTSCHIG, C.; WIK, P.; WINTERBERG, T. *Next Generation SOA: A Concise Introduction to Service Technology & Service-Orientation*. 1st. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014. ISBN 0133859045, 9780133859041. Citado 2 vezes nas páginas 30 e 31.

ESPOSITO, D. *Architecting Mobile Solutions for the Enterprise*. Microsoft Press, 2012. 472 p. ISBN 9780735663022. Disponível em: <https://www.safaribooksonline.com/library/view/architecting-mobile-solutions/9780735671324/>. Citado 5 vezes nas páginas 19, 20, 21, 22 e 23.

EVANS, E. *Domain-driven design : tackling complexity in the heart of software*. Addison-Wesley Professional, 2003. ISBN 0321125215. Disponível em: <https://www.safaribooksonline.com/library/view/domain-driven-design-tackling/0321125215/>. Citado na página 35.

FIELDING, R.; GETTYS, J.; MOGUL, J.; FRYSTYK, H.; MASINTER, L.; LEACH, P.; BERNERS-LEE, T. *RFC: 2616 - Hypertext Transfer Protocol – HTTP/1.1*. 1999. Disponível em: <https://www.ietf.org/rfc/rfc2616.txt>. Citado 2 vezes nas páginas 45 e 48.

FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. 162 p. Tese (Doutorado) — University of California, Irvine, 2000. Disponível em: <http://www.ics.uci.edu/{~}fielding/pubs/dissertation/top.h>. Citado 5 vezes nas páginas 41, 42, 43, 45 e 47.

FOWLER, M.; LEWIS, J. *Microservices*. 2014. Disponível em: <http://martinfowler.com/articles/microservices.html>. Citado 9 vezes nas páginas 14, 16, 32, 33, 34, 35, 38, 39 e 40.

GONZALEZ, D. *Developing microservices with Node.js : learn to develop efficient and scalable microservices for server-side programming in Node.js using this hands-on guide*. [s.n.], 2016. ISBN 9781785887406. Disponível em: <https://www.safaribooksonline.com/library/view/developing-microservices-with/9781785887406/>. Citado 5 vezes nas páginas 32, 37, 38, 39 e 58.

GREGORIO, J.; FIELDING, R.; HADLEY, M.; NOTTINGHAM, M.; ORCHARD, D. *RFC 6570 - URI Template*. 2012. Disponível em: <https://tools.ietf.org/html/rfc6570>. Citado na página 45.

HAAS, H. Reconciling Web Services and REST Services. In: *European Conference on Web Services (ECOWS)*. [s.n.], 2005. Disponível em: <https://www.w3.org/2005/Talks/1115-hh-k-ecows>. Citado na página 30.

HWANG, K.; FOX, G. C.; DONGARRA, J. J. *Distributed and cloud computing : from parallel processing to the Internet of things*. [s.n.], 2013. ISBN 9780128002049. Disponível em: <https://www.safaribooksonline.com/library/view/distributed-and-cloud/9780123858801/>. Citado na página 27.

JOSUTTIS, N. M. *SOA in practice*. O'Reilly, 2007. 324 p. ISBN 9780596529550. Disponível em: <https://www.safaribooksonline.com/library/view/soa-in-practice/9780596529550/>. Citado na página 39.

KANGASAHO, M. *Legacy application modernization with REST wrapping*. Tese (Doutorado) — University of Helsinki, 2016. Disponível em: <https://helda.helsinki.fi/handle/10138/167171>. Citado 6 vezes nas páginas 15, 25, 27, 52, 54 e 64.

KHOSROW-POUR, M. *Encyclopedia of information science and technology*. IGI Global, 2014. ISBN 9781466658882. Disponível em: <https://www.safaribooksonline.com/library/view/encyclopedia-of-information/9781466658882/>. Citado na página 42.

LEE, V.; SCHNEIDER, H.; SCHELL, R. *Mobile applications : architecture, design, and development*. Pearson Education, 2004. 340 p. ISBN 9780131172630. Disponível em: <https://www.safaribooksonline.com/library/view/mobile-applications-architecture/0131172638/>. Citado na página 19.

LEVCOVITZ, A.; TERRA, R.; VALENTE, M. T. Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems. *CoRR*, abs/1605.0, 2016. Disponível em: <http://arxiv.org/abs/1605.03175>. Citado 3 vezes nas páginas 53, 54 e 86.

LODDERSTEDT, T.; MCGLOIN, M.; HUNT, P. OAuth 2.0 Threat Model and Security Considerations. 2013. Disponível em: <https://tools.ietf.org/html/rfc6819>. Citado na página 52.

LOUVEL, J.; TEMPLIER, T.; BOILEAU, T. *Restlet in action : developing RESTful web APIs in Java*. Manning, 2012. 432 p. ISBN 9781935182344. Disponível em: <https://www.safaribooksonline.com/library/view/restlet-in-action/9781935182344/>. Citado na página 42.

MASSE, M. *REST API Design Rulebook*. O'Reilly Media, Inc., 2011. ISBN 9781449310509. Disponível em: <https://www.safaribooksonline.com/library/view/rest-api-design/9781449317904/>. Citado 3 vezes nas páginas 41, 42 e 60.

MUELLER, J. P. *Security for Web Developers*. O'Reilly Media, Inc., 2015. ISBN 9781491928646. Disponível em: <https://www.safaribooksonline.com/library/view/security-for-web/9781491928684/>. Citado 2 vezes nas páginas 33 e 40.

NEWMAN, S. *Building microservices: designing fine-grained systems*. O'Reilly Media, Inc., 2015. 473 p. ISBN 9781491950357. Disponível em: <https://www.safaribooksonline.com/library/view/building-microservices/9781491950340/>. Citado 7 vezes nas páginas 32, 33, 34, 37, 39, 40 e 67.

NILSSON, S. *Application modernization: approaches, problems and evaluation*. Tese (Doutorado) — Umea University, 2015. Disponível em: <urn:nbn:se:umu:diva-112058>. Citado na página 25.

PAPAZOGLOU, M. P.; TRAVERSO, P.; DUSTDAR, S.; LEYMANN, F. SERVICE-ORIENTED COMPUTING: A RESEARCH ROADMAP. *International Journal of Cooperative Information Systems*, v. 17, n. 02, p. 223–255, 2008. Disponível em: <http://www.worldscientific.com/doi/abs/10.1142/S0218843008001816>. Citado 3 vezes nas páginas 27, 28 e 29.

PARASTATIDIS, S.; WEBBER, J.; ROBINSON, I. *REST in practice*. O'Reilly Media, Inc., 2010. ISBN 9780596805821. Disponível em: <https://www.safaribooksonline.com/library/view/rest-in-practice/9781449383312/>. Citado 2 vezes nas páginas 43 e 45.

PESSOA, S. *Dissertação Não é Bicho Papão*. [S.l.]: Rocco, 2005. 157 p. ISBN 9788532518866. Citado na página 17.

PRODANOV, C. C.; FREITAS, E. C. *Metodologia do trabalho científico: métodos e técnicas da pesquisa e do trabalho acadêmico*. Novo Hamburgo: FEEVALE, 2013. ISBN 9788577171583. Citado na página 17.

PUGLISI, S. *RESTful Rails development : building open applications and services*. O'Reilly Media, Inc., 2015. ISBN 9781491910856. Disponível em: <https://www.safaribooksonline.com/library/view/restful-rails-development/9781491910849/>. Citado 2 vezes nas páginas 40 e 41.

PURUSHOTHAMAN, J. *RESTful Java Web Services*. 2. ed. Packt Publishing, 2015. ISBN 9781784399092. Disponível em: <https://www.safaribooksonline.com/library/view/restful-java-web/9781784399092/>. Citado 3 vezes nas páginas 47, 50 e 60.

RAMANATHAN, R.; RAJA, K. *Service-Driven Approaches to Architecture and Enterprise Integration*. Hershey, PA, USA: IGI Global, 2013. ISBN 1466641932, 9781466641938. Disponível em: <http://dl.acm.org/citation.cfm?id=2531469>. Citado na página 28.

RICHARDSON, C. *Justice Will Take Us Millions Of Intricate Moves*. 2008. Disponível em: <https://www.crummy.com/writing/speaking/2008-QCon/act3.html>. Citado na página 47.

RICHARDSON, C. Decompose that WAR? A pattern language for microservices. In: *QCon São Paulo*. InfoQ, 2015. Disponível em: <https://www.infoq.com/br/presentations/um-padrao-de-linguagem-para-microsservicos>. Citado 2 vezes nas páginas 39 e 59.

RICHARDSON, C. *Microservices: From Design to Deployment*. NGINX, 2016. Disponível em: <https://www.nginx.com/blog/microservices-from-design-to-deployment-ebook-nginx/>. Citado 4 vezes nas páginas 57, 58, 59 e 86.

RICHARDSON, L.; RUBY, S.; AMUNDSEN, M. *RESTful Web APIs*. O'Reilly Media, Inc., 2013. ISBN 9781449358068. Disponível em: <https://www.safaribooksonline.com/library/view/restful-web-apis/9781449359713/>. Citado 2 vezes nas páginas 41 e 45.

ROTEM-GAL-OZ, A. *SOA patterns*. Manning, 2012. ISBN 9781933988269. Disponível em: <https://www.safaribooksonline.com/library/view/soa-patterns/9781933988269/>. Citado na página 31.

RV, R. *Spring microservices*. Packt Publishing Limited, 2016. ISBN 9781786466686. Disponível em: <https://www.safaribooksonline.com/library/view/spring-microservices/9781786466686/>. Citado 6 vezes nas páginas 32, 33, 34, 36, 39 e 70.

SHARMA, S. *Mastering microservices with java*. Packt Publishing Limited, 2016. ISBN 9781785285172. Disponível em: <https://www.safaribooksonline.com/library/view/mastering-microservices-with/9781785285172/>. Citado 2 vezes nas páginas 14 e 33.

STERLING, C.; BARTON, B. *Managing Software Debt: Building for Inevitable Change*. Addison-Wesley Professional, 2010. ISBN 9780321700582. Disponível em: <https://www.safaribooksonline.com/library/view/managing-software-debt/9780321700582/>. Citado na página 25.

TOTTY, B.; SAYER, M.; REDDY, S.; AGGARWAL, A.; GOURLEY, D. *HTTP: The Definitive Guide*. O'Reilly Media, Inc., 2002. 635 p. ISBN 9781565925090. Disponível em: <https://www.safaribooksonline.com/library/view/http-the-definitive/1565925092/>. Citado 2 vezes nas páginas 47 e 48.

VARANASI, B.; BELIDA, S. *Spring REST*. Apress, 2015. ISBN 9781484208236. Disponível em: <https://www.safaribooksonline.com/library/view/spring-rest/9781484208236/>. Citado 5 vezes nas páginas 43, 44, 48, 49 e 60.

VENNAM, R.; NARAIN, S.; MARTINS, M.; LAMPKIN, V.; JOSHI, S.; GUPTA, M.; GUCER, V.; GLOZIC, D.; FERREIRA, C. M.; EATI, K.; Van Duy, N.; DAYA, S. *Microservices from theory to practice : creating applications in IBM Bluemix using the microservices approach*. IBM Redbooks, 2015. ISBN 9780738440811. Disponível em: <https://www.safaribooksonline.com/library/view/microservices-from-theory/9780738440811/>. Citado 5 vezes nas páginas 34, 35, 38, 40 e 70.

VILLAMIZAR, M.; GARCES, O.; CASTRO, H.; VERANO, M.; SALAMANCA, L.; CASALLAS, R.; GIL, S. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: *2015 10th Computing Colombian Conference (10CCC)*. IEEE, 2015. p. 583–590. ISBN 978-1-4673-9464-2. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7333476>. Citado na página 38.

VISAGGIO, G. Ageing of a data?intensive legacy system: symptoms and remedies. *Journal of Software Maintenance and Evolution: Research and Practice*, Wiley Online Library, v. 13, n. 5, p. 281–308, 2001. ISSN 1532-0618. Citado 2 vezes nas páginas 25 e 70.

WEIDERMAN, N.; NORTHROP, L.; SMITH, D.; TILLEY, S.; WALLNAU, K. *Implications of Distributed Object Technology for Reengineering*. Pittsburgh, PA, 1997. Disponível em: <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=12835>. Citado 4 vezes nas páginas 26, 27, 65 e 68.

WILDER, B. *Cloud Architecture Patterns*. O'Reilly Media, 2012. 182 p. ISBN 978-1-4493-1977-9. Disponível em: <https://www.safaribooksonline.com/library/view/cloud-architecture-patterns/9781449357979/>. Citado na página 37.

WISNIEWSKI, J. Mobile that works for your library. *Online. Academic OneFile*, Jan.-Feb„ 2011. Citado 7 vezes nas páginas 20, 21, 22, 23, 24, 56 e 58.

WROBLEWSKI, L. Mobile First. A Book Apart, 2011. Disponível em: <https://www.safaribooksonline.com/library/view/mobile-first/9780133052893/>. Citado 6 vezes nas páginas 14, 16, 19, 23, 56 e 83.

ZHANG, J.; SNOOK, R.; SCHRAG, D.; RAY, S.; MATHUR, A.; CHANDGADKAR, O.; BARCIA, R.; WILLIAMSON, L. *Enterprise class mobile application development : a complete lifecycle approach for producing mobile apps*. IBM Press, 2015. ISBN 9780133478679. Disponível em: <https://www.safaribooksonline.com/library/view/enterprise-class-mobile/9780133478679/>. Citado na página 24.

ZHANG, W.; BERRE, A. J.; ROMAN, D.; HURU, H. A. Migrating Legacy Applications to the Service Cloud. In: *14th Conference Companion on Object Oriented Programming Systems Languages and Applications*. [s.n.], 2009. Disponível em: <https://www.researchgate.net/publication/285760060{_}Migrating{_}Legacy{_}Applications{_}to{_}th>. Citado 3 vezes nas páginas 53, 54 e 56.

APÊNDICES

APÊNDICE A – ESPECIFICAÇÃO API DISCENTEPOS

```yaml
#Swagger Version
swagger: '2.0'

#Metadata
info:
  version: "0.1.0"
  title: API discentePos
schemes:
  - http
host: 192.168.25.8:9090
basePath: /v1/apiDiscentePos
produces:
  - application/json

#Endpoints
paths:
  /noticias:
    get:
      summary: Retorna todas as notícias cadastradas.
      description:
        Retorna todas as noticias cadastradas, independente da data de início, data de expiração ou do perfil de usuário a que a notícia se destina.
      parameters:
        - name: emVigor
          in: query
          description: Indica se deve retornar apenas notícias cuja data de início seja igual ou inferior à data atual e a data término seja posterior à data atual.
          required: false
          type: boolean
      responses:
        200:
          description: Sucesso
          schema:
            type: array
            items:
              $ref: '#/definitions/Noticia'
        404:
```

```
36            $ref: "#/responses/RecursoNaoEncontrato"
37          500:
38            $ref: "#/responses/ErroNoServidor"
39    /noticias/{codigoNoticia}:
40      get:
41        summary: Retorna uma notícia.
42        description: Consulta uma notícia pelo seu código.
43        parameters:
44          - $ref: "#/parameters/codigoNoticia"
45        responses:
46          200:
47            description: Sucesso
48            schema:
49              $ref: '#/definitions/Noticia'
50          404:
51            $ref: "#/responses/RecursoNaoEncontrato"
52          500:
53            $ref: "#/responses/ErroNoServidor"
54        security:
55          - OauthSecurity: []
56    /conceitosObtidos/{matricula}:
57      get:
58        summary: Retorna conceitos obtidos pelo discente.
59        description: Consulta todas as disciplinas cursadas pelo discente
             que possuem conceito atribuído.
60        parameters:
61          - name: matricula
62            in: path
63            description: Número de matrícula do discente.
64            required: true
65            type: integer
66        responses:
67          200:
68            description: Sucesso
69            schema:
70              type: array
71              items:
72                $ref: '#/definitions/ConceitoObtido'
73          404:
74            $ref: "#/responses/RecursoNaoEncontrato"
75          500:
76            $ref: "#/responses/ErroNoServidor"
77        security:
78          - OauthSecurity: []
79    /userInfo:
80      get:
81        summary: Dados do usuário
```

```
82        description: Retorna informações sobre o usuário que autorizou
              acesso aos recursos
83        responses:
84          200:
85            description: Sucesso
86            schema:
87              $ref: "#/definitions/UserInfo"
88        security:
89          - OauthSecurity: []
90
91  #Definições de segurança
92  securityDefinitions:
93    OauthSecurity:
94      type: oauth2
95      flow: accessCode
96      authorizationUrl: http://localhost:8080/oauth2/authorize
97      tokenUrl: http://localhost:8080/oauth2/token
98      scopes:
99        noticias_leitura: Escopo para consulta de notícias
100
101 #Segurança aplicada à API
102 security:
103   - OauthSecurity:
104     - noticias_leitura
105
106
107 #Parametros comuns
108 parameters:
109   codigoNoticia:
110     name: codigoNoticia
111     in: path
112     description: Código da notícia
113     required: true
114     type: integer
115
116 #Modelos
117 definitions:
118   Noticia:
119     required:
120       - codigoNoticia
121     type: object
122     properties:
123       codigoNoticia:
124           type: integer
125       nomeNoticia:
126         type: string
127       descricaoNoticia:
```

```
128           type: string
129         ordem:
130           type: integer
131         geral:
132           type: string
133         listaCodigosTiposPerfisFuncionais:
134           type: array
135           items:
136             type: integer
137         dataInicio:
138           type: string
139           format: date
140     ConceitoObtido:
141       type: object
142       properties:
143         disciplina:
144           type: string
145         periodo:
146           type: string
147         conceito:
148           type: string
149     Erro:
150       properties:
151         codigo:
152           type: string
153         mensagem:
154           type: string
155     UserInfo:
156       properties:
157         nome:
158           type: string
159         matricula:
160           type: string
161         curso:
162           type: string
163 #Respostas comuns
164 responses:
165     ErroNoServidor:
166       description: Erro inesperado no servidor.
167       schema:
168         $ref: "#/definitions/Erro"
169     RecursoNaoEncontrato:
170       description: Recurso não encontrado.
171       schema:
172         $ref: "#/definitions/Erro"
```

Código A.1 – Especificação API discentePos

APÊNDICE B – FILTRO DE AUTENTICAÇÃO DA API DISCENTEPOS

```
1  @Provider
2  @Priority(Priorities.AUTHENTICATION)
3  public class AuthenticationFilter implements ContainerRequestFilter {
4      @Override
5      public void filter(ContainerRequestContext requestContext) throws
           IOException {
6          // Extrai cabeçalho AUTHORIZATION
7          String authorizationHeader =
8              requestContext.getHeaderString(HttpHeaders.AUTHORIZATION);
9
10         // Verifica se o cabeçalho possui token do tipo Bearer
11         if (authorizationHeader == null || !authorizationHeader.
               startsWith("Bearer ")) {
12                 System.out.println("Request sem token Bearer.");
13             throw new NotAuthorizedException("Authorization header must
                   be provided");
14         }
15
16         // Extrai o token do cabeçalho
17         String token = authorizationHeader.substring("Bearer".length()).
               trim();
18         try {
19             validateToken(token, requestContext);
20         } catch (Exception e) {
21                 e.printStackTrace();
22                 requestContext.abortWith(
23                 Response.status(Response.Status.UNAUTHORIZED).build());
24         }
25     }
26
27     private void validateToken(String token, ContainerRequestContext
           requestContext) throws Exception {
28         CredentialsProvider credsProvider = new
               BasicCredentialsProvider();
29         credsProvider.setCredentials(
30                 new AuthScope("localhost", 8080),
31                 new UsernamePasswordCredentials("9fe2c7b7-6767-46fa-
                       a3bf-ac1cd87a87ec", "e6f948ed-3399-4df5-ac6e-
                       ab8293f51501"));
```

```
32            CloseableHttpClient httpclient = HttpClients.custom().
                  setDefaultCredentialsProvider(credsProvider).build();
33            try {
34                HttpGet httpget = new HttpGet("http://localhost:8080/v1/
                      tokeninfo?access_token=" + token);
35                CloseableHttpResponse response = httpclient.execute(httpget
                      );
36                if(response.getStatusLine().getStatusCode() >= 400 &&
                      response.getStatusLine().getStatusCode() <= 500){
37                    throw new Exception();
38                }
39                try {
40                    System.out.println(response.getStatusLine());
41                    String rawResponse = EntityUtils.toString(response.
                          getEntity(), "UTF-8");
42
43                    ObjectNode  node = new  ObjectMapper().readValue(
                          rawResponse, ObjectNode.class);
44                    String jsonPrincipal = null;
45                    if(node.has("principal")){
46                            jsonPrincipal = node.get("principal").toString
                                ();
47                        ObjectMapper mapper = new ObjectMapper();
48                        Usuario principal = mapper.readValue(jsonPrincipal,
                              Usuario.class);
49                        APISecurityContext securityContext = new
                              APISecurityContext(principal);
50                        requestContext.setSecurityContext(securityContext);
51                    }
52                } finally {
53                    response.close();
54                }
55            } finally {
56                httpclient.close();
57            }
58        }
59    }
```

Código B.1 – Filtro de autenticação da API discentePos