UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UMA API PARA SINCRONIZAÇÃO DE DADOS, BASEADA EM MICRO SERVIÇOS, PARA O SUPORTE AO DESENVOLVIMENTO DE APLICAÇÕES MULTIPLATAFORMA *OFFLINE*

FERNANDA ZAMPIERI CANAVER

ORIENTADOR: PROF. DR. DELANO MEDEIROS BEDER

São Carlos - SP
Junho/2017

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UMA API PARA SINCRONIZAÇÃO DE DADOS, BASEADA EM MICRO SERVIÇOS, PARA O SUPORTE AO DESENVOLVIMENTO DE APLICAÇÕES MULTIPLATAFORMA *OFFLINE*

FERNANDA ZAMPIERI CANAVER

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software.
Orientador: Prof. Dr. Delano Medeiros Beder

São Carlos - SP
Junho/2017

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a defesa de Dissertação de Mestrado da candidata Fernanda Zampieri Canaver, realizada em 19/07/2017.

Prof. Dr. Delano Medeiros Beder
(UFSCar)

Prof. Dr. Auri Marcelo Rizzo Vincenzi
(UFSCar)

****
Prof. Dr. Marcelo Morandini
(USP)

Certifico que a sessão de defesa foi realizada com a participação à distância do membro Marcelo Morandini, depois das arguições e deliberações realizadas, o participante à distância está de acordo com o conteúdo do parecer da comissão examinadora redigido no relatório de defesa da aluna Fernanda Zampieri Canaver.

Prof. Dr. Delano Medeiros Beder
Presidente da Comissão Examinadora
(UFSCar)

AGRADECIMENTO

Gostaria de agradecer primeiramente a Deus por tudo e por todos.

Agradeço muito à minha família por todo apoio, incentivo, ajuda, amor e carinho dispendidos à mim.

Expresso também minha gratidão ao meu orientador professor Dr. Delano Medeiros Beder por toda a orientação, apoio, ajuda e dedicação dispendidos neste trabalho. Agradeço também a todos os demais professores que colaboraram muito com a minha formação acadêmica e que me incentivaram a prosseguir minha formação com a pós-graduação *stricto sensu*.

Agradeço a todos os demais amigos por todo o incentivo, apoio e carinho durante o desenvolvimento deste trabalho.

Agradeço também à CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) pela concessão da bolsa para a realização deste trabalho.

RESUMO

Com o aumento do uso de dispositivos como computadores, tablets e smartphones pelas pessoas e a diversidade de plataformas desses dispositivos surge a necessidade de desenvolvimento de aplicações que sejam multiplataforma, de forma que uma mesma aplicação possa ser executada pelo usuário a partir de qualquer dispositivo. Nesse contexto, as aplicações devem suportar algum mecanismo de sincronização de dados para que usuários tenham acesso às mesmas informações em uma dada aplicação independemente de qual dispositivo estejam utilizando. Para que haja a sincronização, pode ser necessária a existência de um sítio para armazenamento e centralização de dados, que seja acessível a todos os dispositivos. Como a disponibilidade de Internet nos dispositivos é um ponto crítico, especialmente nos dispositivos móveis, em que pode-se ocorrer a perda de conexão, é importante que as aplicações lidem com a questão do funcionamento *offline* (quando não há conexão). Este trabalho propõe a criação de uma API baseada em JavaScript e HTML5 a fim de apoiar o desenvolvimento de aplicações multiplataforma híbridas. As funcionalidades da API incluem a sincronização de dados e o funcionamento *offline* da aplicação. Propõe também o uso de um servidor web para o armazenamento e centralização de dados e o uso de micro serviços desenvolvidos em Java com o *framework* Spring Boot para a sincronização de dados entre a API e o servidor web.

Palavras-chave: aplicação multiplataforma, funcionamento *offline*, sincronização de dados, micro serviços.

ABSTRACT

With the increased use of devices such as computers, tablets and smartphones by people and the diversity of platforms these devices arises the need of developing cross-platform applications, in order to the same application can be run by the user from any device. In this context, the applications must support some data synchronization mechanism so that users have access to the same information in an application independently of which device is in use. For there to be synchronization, the existence of a location on the Internet for storage and centralization of data can be required, which is accessible to all devices. The availability of Internet on devices is a critical point, especially on mobile devices, which can happen loss of connection; therefore it is important that applications deal with the issue of working offline (when there is no connection). This work proposes creation of an API based on JavaScript and HTML5 in order to support development of hybrid cross-platform applications. The features of the API include the data synchronization and offline operation of application. It also proposes the use of a web server for storage and centralization of data and the use of microservices developed in Java with Spring Boot framework to synchronize data between the API and the web server.

Keywords: cross-platform, offline operation, data synchronization, microservices.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

AIJ - Artrite Idiopática Juvenil

AJAX - *Asynchronous JavaScript and XML*

API - *Application Programming Interface*

AVA - Ambiente Virtual de Aprendizagem

CSS - *Cascading Style Sheets*

DAO - *Data Access Object*

GIF - *Graphics Interchange Format*

HATEOAS - *Hypermedia As The Engine Of Application State*

HTML - *HyperText Markup Language*

HTTP - *HyperText Transfer Protocol*

JAR - *Java Archive*

JPA - *Java Persistence API*

JPEG - *Joint Photographic Experts Group*

JSON - *JavaScript Object Notation*

JVM - *Java Virtual Machine*

LOA - Laboratório de Objetos de Aprendizagem

PinGO - *Pain Information on the Go*

PNG - *Portable Network Graphics*

REST - *REpresentational State Transfer*

SMTP - *Simple Mail Transfer Protocol*

SOA - *Service Oriented Architecture*

SQL - *Structured Query Language*

UFSCar - Universidade Federal de São Carlos

URI - *Uniform Resource Identifier*

URL - *Uniform Resource Locator*

XML - *eXtensible Markup Language*

SUMÁRIO

Capítulo 1

INTRODUÇÃO

Neste capítulo serão apresentados o contexto, a motivação e os objetivos gerais e específicos de pesquisa. Consta também a organização do trabalho, que é a estrutura desta dissertação.

1.1 Contexto

O uso crescente de dispositivos pelas pessoas como, por exemplo, computadores, *tablets* e *smartphones*, aliado às suas diferentes plataformas, pode ocasionar a necessidade de desenvolvimento de aplicações que sejam multiplaforma, de modo que um usuário de determinada aplicação consiga acessar a mesma aplicação a partir de quaisquer dispositivos. Segundo Lopes (2013), apostar no desenvolvimento para uma determinada plataforma é arriscado, pois há alguns anos apostar no iOS parecia o tiro certeiro para atingir a maioria do mercado, porém hoje o Android é dominante na maior parte do mundo, inclusive no Brasil, e futuramente não se sabe se permanecerá como dominante.

Para que uma mesma aplicação possa ser executada por um mesmo usuário utilizando qualquer dispositivo e considerando que a aplicação utilize informações que precisam estar atualizadas de acordo com os acessos anteriores do usuário na aplicação, independentemente de qual dispositivo foi usado, pode ser necessária a existência de um local comum para a centralização e armazenamento de tais informações e que possa ser acessado pelos dispositivos.

Outra questão importante é quanto à disponibilidade de Internet nos dispositivos para que a aplicação consiga fazer a sincronização de dados. Considerando os dispositivos móveis, sabe-se que é mais comum a perda de conexão devido à mobilidade. O ideal é que nenhuma informação seja perdida em caso de oscilação ou perda de conexão com a Internet.

1.2 Motivação e Objetivos

Um dos desafios relacionado à heterogeneidade de dispositivos é desenvolver aplicações que sejam multiplataforma, ou seja, funcionem normalmente independentemente de plataforma. Por exemplo, seja possível ao usuário utilizar a aplicação em um dispositivo em que tenha instalada uma versão do Android e também em um dispositivo em que tenha instalada uma versão do Windows e, em ambos, ter acesso às mesmas informações (sincronizadas e armazenadas em um local comum na Internet).

A disponibilidade de Internet nos dispositivos apresenta-se também como um ponto crítico. Um problema encontrado nesse contexto é quanto à realização da sincronização de dados da aplicação presente no dispositivo quando o mesmo está sem acesso à Internet, por alguma oscilação ou perda de conexão, tornando a aplicação *offline*. Dessa forma, a aplicação não conseguirá enviar e receber dados via Internet.

Possíveis abordagens quando a aplicação está *offline*:

1) Interrupção da aplicação;
2) Uso da aplicação será parcial: acesso parcial às funcionalidades;
3) Uso da aplicação será completo: acesso completo a todas as funcionalidades.

Nesse contexto, o objetivo geral deste trabalho é propor uma abordagem de forma que o dispositivo comunique-se por meio do mecanismo a ser proposto assim que estiver com acesso à Internet novamente, sincronizando todas as informações que ficaram pendentes sem sincronização devido à falta de comunicação ocorrida por queda ou indisponibilidade de acesso à Internet. E que o uso da aplicação possa ser completo mesmo quando o dispositivo estiver sem conexão com a Internet.

Entre os objetivos específicos tem-se a criação de uma *Application Programming Interface*[1] (API), baseada em JavaScript e HTML5, que dê suporte à comunicação de aplicações multiplataforma híbridas, desenvolvidas com HTML5, *Cascading Style Sheets*[2] (CSS) - que é um mecanismo rápido para adição de estilos - e JavaScript, com o local que será utilizado para a centralização de dados e ao funcionamento *offline* da aplicação.

Uma aplicação híbrida combina elementos de aplicações Web e aplicações nativas. Aplicações Web são generalizadas para várias plataformas e não são instaladas localmente, mas ficam disponíveis na Internet por meio de um navegador. Aplicações nativas são desenvolvidas para uma plataforma específica e são instaladas no dispositivo. As aplicações híbridas, como aplicações nativas, podem aproveitar os muitos recursos do dispositivo disponíveis. Como aplicações Web, elas dependem de HTML sendo processado em um navegador, com a ressalva de que o navegador está incorporado dentro da aplicação (BUDIU, 2013).

Um segundo objetivo específico é a criação do mecanismo para que haja a sincronização de dados, baseada em micro serviços RESTful (caracterizados por seguirem o estilo arquitetural REST). Micro serviços são vantajosos pelo fato, entre outros, de serem independentes, a implementação e implantação de um micro serviço não afetará os demais. Uma das vantagens do uso do estilo arquitetural REST é que pode ser adotado em praticamente qualquer cliente ou servidor com suporte a HTTP.

1.3 Organização do Trabalho

Para a composição do Capítulo 2 foi feita uma revisão bibliográfica com o intuito de fornecer uma fundamentação teórica ao leitor. O capítulo inclui os conceitos, tecnologias e ferramentas relacionados a esta pesquisa. Inclui também trabalhos já realizados que possuem algumas similaridades com o tema desta pesquisa.

[1] Em português: Interface de Programação de Aplicação

[2] <https://www.w3.org/Style/CSS/>

No Capítulo 3 é apresentada a arquitetura proposta neste trabalho. Também são apresentados os detalhes de implementação para o desenvolvimento do trabalho.

No Capítulo 4 são descritos os estudos de caso que foram realizados para a validação da abordagem proposta neste trabalho. Inclui também o detalhamento dos procedimentos realizados em cada estudo de caso.

No Capítulo 5 são apresentadas as contribuições e limitações deste trabalho, bem como os possíveis trabalhos futuros relacionados a este.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

Este capítulo visa fornecer uma fundamentação teórica, abordando os conceitos, as tecnologias e as ferramentas inerentes à realização deste trabalho. Serão discutidos também trabalhos relacionados ao tema desta pesquisa.

2.1 Considerações Iniciais

Este capítulo aborda conceitos, tecnologias e ferramentas que foram utilizados a fim de alcançar os objetivos deste trabalho. Serão apresentados também trabalhos já realizados que são relacionados em alguns aspectos ao tema desta pesquisa.

A organização deste capítulo inclui uma breve explicação a respeito do HyperText Transfer Protocol (HTTP), Model-View-Controller (MVC), Arquitetura Orientada a Serviços, tal seção abrangerá também Serviços Web e Micro Serviços, bem como um *framework* para o desenvolvimento de micro serviços. Inclui também uma seção sobre o desenvolvimento híbrido e funcionamento *offline*, abordando as tecnologias HTML5, JavaScript, AJAX, jQuery, Crosswalk e Electron, posteriormente uma seção com os trabalhos relacionados e, como última seção do capítulo, as considerações finais.

2.2 *HyperText Transfer Protocol* (HTTP)

HyperText Transfer Protocol[3] (HTTP) é um protocolo genérico e sem estado que pode ser usado para diversas tarefas além de seu uso para hipertexto, tais como servidores de nome e sistemas de gerenciamento de objetos distribuídos, por meio da extensão de seus métodos de requisição, códigos de erro e cabeçalhos. Um recurso do HTTP é a tipificação e negociação da representação de dados, permitindo que os sistemas sejam construídos independentemente dos dados que estão sendo transferidos (FIELDING et al., 1999).

Segundo Wong (2000), HTTP é o protocolo por trás da Web e é útil porque provê um caminho padronizado para a comunicação: especifica como clientes requisitam dados e como servidores respondem a essas requisições.

2.3 *Model-View-Controller* (MVC)

O padrão arquitetural *Model-View-Controller*[4] (MVC) separa uma aplicação em três principais grupos de componentes: Modelos, Visões e Controladores. Esse padrão ajuda a alcançar a separação de conceitos. Com o uso desse padrão, as requisições de usuários são roteadas para um Controlador que é responsável por trabalhar com o Modelo para realizar ações do usuário e/ou recuperar resultados de consultas. O Controlador escolhe a Visão que irá exibir ao usuário qualquer dado do Modelo que tenha sido requisitado (SMITH, 2016).

Essa separação é importante porque, por exemplo, a lógica da interface de usuário tende a mudar com mais frequência que a lógica de negócios e, se o código de apresentação e a lógica de negócios forem combinados em um único objeto, é necessário modificar um objeto contendo a lógica de negócios toda vez que for necessário alterar a interface do usuário. Isso pode introduzir erros e exigirá a repetição de testes de toda a lógica de negócios após cada alteração mínima da interface do usuário (SMITH, 2016).

[3] Em português: Protocolo de Tranferência de Hipertexto

[4] Em português: Modelo-Visão-Controlador

Modelo em uma aplicação MVC representa o estado da aplicação e qualquer lógica de negócios ou operações que podem ser realizadas pela aplicação. A lógica de negócios deve ser encapsulada no modelo, incluindo qualquer lógica de implementação para persistir o estado da aplicação (SMITH, 2016).

Visões são responsáveis por apresentar conteúdo por meio da interface do usuário. Deve haver o mínimo de lógica dentro das visões e qualquer lógica dentro das visões deve ser relacionada apenas com a apresentação de conteúdo (SMITH, 2016).

Controladores são os componentes que manipulam a interação do usuário, trabalham com o modelo e, em última instância, selecionam uma visão para exibir as informações. O controlador manipula e responde à entrada e interação do usuário (SMITH, 2016).

No padrão MVC, o controlador é o ponto de entrada inicial e é responsável pela seleção dos tipos de modelo com os quais irá trabalhar e é responsável também por selecionar qual visão irá exibir as informações. Então, o controlador é quem gerencia como a aplicação responde a uma determinada solicitação (SMITH, 2016).

2.4 Arquitetura Orientada a Serviços

A Arquitetura Orientada a Serviços, do inglês *Service Oriented Architecture* (SOA), é um estilo arquitetural que suporta a orientação a serviços. A orientação a serviço é uma maneira de pensar em termos de serviços, de desenvolvimento baseado em serviços e dos resultados dos serviços. Um serviço é uma representação lógica de uma atividade de negócios repetitiva que tem uma saída específica (por exemplo, checar o crédito do cliente, disponibilizar dados do clima, entre outros) e é independente (o que o serviço precisa para operar deve estar contido nele). Um estilo arquitetural é a combinação de características distintas nas quais a arquitetura é realizada ou representada (THE OPEN GROUP, 2009).

SOA representa um modelo em que a lógica de negócios é decomposta em unidades menores e encoraja essas unidades individuais de lógica a existirem autonomamente mas não isoladas umas das outras. Coletivamente, essas unidades

compreendem a lógica de negócios e, individualmente, podem estar distribuídas. Unidades de lógica precisam seguir um conjunto de princípios que possibilitem que cada unidade evolua de forma independente, mantendo ainda uma quantidade suficiente de uniformização e padronização. Em SOA, essas unidades de lógica são conhecidas como serviços (ERL, 2005).

Para manter a independência, os serviços encapsulam a lógica dentro de um contexto distinto. Esse contexto pode ser específico para uma tarefa de negócios, uma entidade de negócios ou algum outro agrupamento lógico. A responsabilidade atribuída a um serviço pode ser pequena ou grande. Portanto, o tamanho e o escopo da lógica representados pelo serviço podem variar (ERL, 2005).

Alguns princípios comuns da orientação a serviço são (ERL, 2005):

- Serviços são reutilizáveis: a orientação a serviços incentiva o reúso em todos os serviços, independentemente se existem requisitos imediatos para reúso. Um serviço é simplesmente uma coleção de operações relacionadas. É, portanto, a lógica encapsulada pelas operações individuais que deve ser considerada reutilizável para justificar a representação de um serviço como reutilizável. Como, por exemplo, o serviço chamado Conta representado na Figura 1;

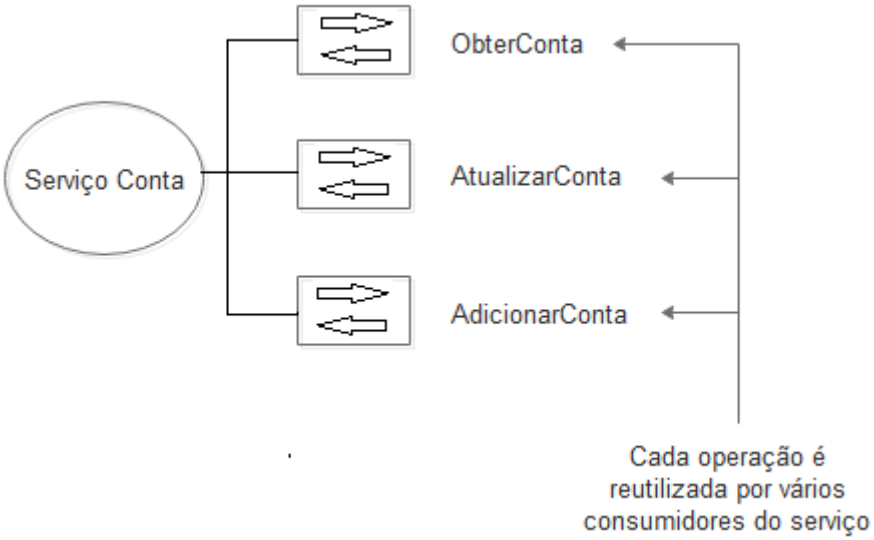

Figura 1 – Um serviço reutilizável expõe operações reutilizáveis.

Fonte: adaptado de Erl (2005).

- Contratos de serviços (documentos de descrição do serviço) que fornecem uma definição formal de: *endpoint* do serviço, cada operação do serviço, cada mensagem de entrada e saída suportada por cada operação, regras e características do serviço e de suas operações. Os contratos precisam ser cuidadosamente mantidos e versionados após sua liberação inicial, já que os consumidores do serviço podem tornar-se dependentes de suas definições;
- Serviços são fracamente acoplados: o acoplamento fraco é uma condição em que um serviço adquire conhecimento sobre outro serviço enquanto permanece independente desse serviço. O acoplamento fraco é conseguido por meio do uso de contratos de serviço que permitem que os serviços interajam dentro de parâmetros predefinidos;
- Serviços abstraem a lógica: princípio que permite que o serviço atue como "caixa preta", ocultando seus detalhes dos consumidores do serviço. Não há limite de quantidade de lógica que um serviço pode representar. Também não há restrição quanto à fonte de lógica da aplicação que um serviço pode utilizar. Por exemplo, um único serviço pode, tecnicamente, expor a lógica da aplicação de dois sistemas diferentes;
- Serviços são compostos: um serviço pode representar qualquer conjunto de lógica de quaisquer tipos de fontes, incluindo outros serviços. A principal razão para implementar esse princípio é garantir que os serviços sejam projetados de forma que possam participar como membros efetivos da composição de outros serviços, se necessário. A composição é simplesmente outra forma de reutilização e, portanto, as operações precisam ser projetadas de forma padronizada e com um nível adequado de granularidade a fim de maximizar as oportunidades de composição;
- Serviços são autônomos: permite que o serviço execute com autonomia de todo o seu processamento. Também elimina dependências de outros serviços, de forma que a implantação e a evolução de um serviço não seja inibida. A autonomia do serviço é uma consideração primordial ao decidir como a lógica da aplicação deve ser dividida em serviços e quais operações devem ser agrupadas em um contexto de serviço. A autonomia não necessariamente concede a um serviço que só ele utilize a lógica que

encapsula, apenas garante que no momento da execução, o serviço tem controle sobre qualquer lógica que representa;

- Serviços são sem estado: os serviços devem minimizar a quantidade de informações de estado que eles gerenciam e a duração para mantê-las. As informações de estado são específicas de dados para uma atividade atual. Enquanto um serviço está processando uma mensagem, por exemplo, está temporariamente com estado. Se um serviço é responsável por manter o estado por períodos mais longos de tempo, sua capacidade de permanecer disponível para outros consumidores do serviço será impedida. Para que um serviço mantenha o menor estado possível, suas operações individuais precisam ser projetadas considerando processamento sem estado. Quanto mais inteligência é acrescentada a uma mensagem, mais independente e autossuficiente ela fica.

2.4.1 Serviços Web

Um serviço web é um pedaço da lógica de negócios, localizado em algum sítio, que é acessível por protocolos de Internet baseados em padrões como *HyperText Transfer Protocol* (HTTP). O uso de um serviço web pode ser algo simples como a realização de autenticação em um site ou algo complexo como facilitar a operação de negócios de uma grande empresa (CHAPPELL; JEWELL, 2002).

Serviços web, na perspectiva da Arquitetura Orientada a Serviços (SOA), possuem as seguintes características principais: são encapsulados e fracamente acoplados (RICCI; DENTI; PIUNTI, 2010).

Segundo Ricci, Denti e Piunti (2010), serviços web representam uma tecnologia de referência para a criação de sistemas distribuídos que precisam suportar interoperabilidade entre aplicações heterogêneas distribuídas por meio de uma rede.

2.4.1.1 Serviços Web RESTful

Representational State Transfer[5] (REST) é um estilo arquitetural para o desenvolvimento de serviços web originado no trabalho de Fielding (2000), que também é coautor do protocolo HTTP. Dessa forma, o estilo arquitetural REST é guiado, dentre outros preceitos, pelas boas práticas de uso de HTTP: uso adequado dos métodos e cabeçalhos HTTP, uso adequado de *Uniform Resource Locator*[6] (URL), uso de códigos de status padronizados para representação de sucesso ou falha e interligações entre vários recursos diferentes (SAUDATE, 2014).

REST é baseado em recursos, que são os conjuntos de dados trafegados pelo protocolo. Para representar dados estruturados, em serviços REST, são usados *eXtensible Markup Language*[7] (XML) ou *JavaScript Object Notation*[8] (JSON) na maioria dos casos. Cada recurso é representado por *Uniform Resource Identifier*[9] (URI), um endereço próprio. Na web, URIs e URLs são essencialmente a mesma coisa (SAUDATE, 2014).

Os serviços web RESTful são caracterizados por seguirem o estilo arquitetural REST. Utilizando serviços web RESTful, o desenvolvimento da comunicação entre o cliente e o servidor para uma aplicação será o mesmo, independentemente de dispositivo, baseada no protocolo HTTP.

Os conceitos de REST devem ser obedecidos para o desenvolvimento de serviços web RESTful. Segundo Saudate (2014), as regras a serem seguidas são:

- Semânticas de recursos: não existem regras quanto à utilizar os recursos no plural ou singular, porém o ideal é que seja mantido um padrão. Todas as URIs, que representam os recursos, no singular ou todas no plural;
- Interação por métodos: para interagir com as URIs devem ser utilizados métodos HTTP. Os recursos são substantivos e os métodos são verbos. Isso significa que os métodos são responsáveis por provocar alterações nos recursos, que são identificados por URIs. Os métodos HTTP são os seguintes:

[5] Em português: Transferência de Estado Representativo

[6] Em português: Localizador Uniforme de Recursos

[7] Em português: Linguagem de Marcação Extensível

[8] Em português: Notação de Objetos JavaScript

[9] Em português: Identificador Uniforme de Recursos

  - *GET*: recupera os dados do recurso, identificado pela URI;
  - *POST*: cria um novo recurso;
  - *PUT*: atualiza um recurso;
  - *DELETE*: exclui um recurso.
- Representações distintas: uso de *media types* para alterar as representações de um mesmo conteúdo sob perspectivas distintas por meio do cabeçalho Accept do HTTP. Por exemplo, o uso do XML ou JSON para dados, ficaria no cabeçalho como *application*/xml ou *application*/json, respectivamente, ou *image*/* para obter imagens. Cabeçalho Accept com o valor *image*/* devido ao fato de que, muitas vezes, não há interesse no tipo da imagem (JPEG, GIF, PNG, entre outros), mas apenas no fato de ela ser uma imagem;
- Uso correto dos códigos de status: implica em conhecimento dos códigos de status do HTTP e correta aplicação dos mesmos para cada situação. Por exemplo, o código de quando ocorre um erro no servidor, o código quando um recurso é criado, o código de quando um recurso buscado não é encontrado, entre outros;
- *Hypermedia As The Engine Of Application State*[10] (HATEOAS): na resposta a uma requisição feita para um determinado recurso, incluir *links* indicando o que pode ser feito em seguida.

2.4.2 Micro Serviços

Micro serviços são uma abordagem para sistemas distribuídos que promovem o uso de serviços de granularidade fina com seus próprios ciclos de vida, que colaboram em conjunto e são modulados principalmente em torno de domínios de negócios (NEWMAN, 2015).

A abordagem de sistemas distribuídos visa decompor uma infraestrutura de serviço monolítica em subsistemas de escalabilidades individuais, que são capazes de serem organizados em uma linha vertical e interconectados por um transporte comum (WOODS, 2015).

[10] Em português: Hipermídia como Mecanismo do Estado da Aplicação

Woods (2015) considera a “Arquitetura de Micro Serviços” como uma sucessora da Arquitetura Orientada a Serviços, sendo que os micro serviços podem ser categorizados na mesma família de “sistemas distribuídos” e carregam muitos dos mesmos conceitos e práticas de SOA. Diferem, no entanto, em termos de escopo de responsabilidade dada a um serviço individual. Em SOA, um serviço pode ser responsável por lidar com uma ampla variedade de domínios de funcionalidades e dados, enquanto que a diretriz geral para um micro serviço é que seja responsável por gerenciar um único domínio de dados e as funções correspondentes a este domínio.

Segundo Newman (2015), micro serviços são pequenos serviços autônomos que trabalham em conjunto e toda comunicação entre eles deve ser feita via chamadas de rede para forçar a separação entre os serviços e evitar os perigos de forte acoplamento.

O estilo arquitetural de micro serviços é uma abordagem para desenvolver uma única aplicação como um conjunto de pequenos serviços, cada um rodando em seu processo e comunicando-se com mecanismos leves (FOWLER;LEWIS, 2014). Para Richardson (2014), serviços individuais são mais fáceis de entender e podem ser desenvolvidos e implantados de forma independente.

Esses serviços devem ter a capacidade de mudarem de forma independente um do outro, e serem implantados sem necessidade de alteração por parte dos consumidores dos serviços. É necessário pensar sobre o que os serviços devem expor e o que eles devem permitir ser ocultado. Se houver muito compartilhamento, os consumidores do serviço tornam-se acoplados às suas representações internas. Isso diminui a autonomia, uma vez que requer uma coordenação adicional com os consumidores quando alterações forem feitas (NEWMAN, 2015).

A regra de ouro é: você pode fazer uma alteração em um serviço e implantá-lo sem alterar qualquer outra coisa? Se a resposta for não, então muitas das vantagens discutidas a seguir serão difíceis de serem alcançadas (NEWMAN, 2015).

2.4.2.1 Vantagens e Desvantagens

Os benefícios de micro serviços são muitos e variados. Micro serviços utilizam os conceitos de sistemas distribuídos e da Arquitetura Orientada a Serviços (NEWMAN, 2015).

Segundo Richardson (2014-2), há diversos benefícios ao utilizar a arquitetura de micro serviços:

- Cada micro serviço é relativamente pequeno, facilitando o entendimento dos desenvolvedores e tornando-os mais produtivos, inclusive porque o ambiente de desenvolvimento funcionará rapidamente devido ao tamanho pequeno do micro serviço;
- Cada serviço pode ser desenvolvido independentemente de outros serviços, facilitando implantar novas versões dos serviços frequentemente;
- É possível escalar o desenvolvimento mais facilmente, pois possibilita a organização do esforço de desenvolvimento em múltiplas equipes, sendo cada equipe responsável por um único serviço. Cada equipe pode desenvolver, implantar e escalar seu serviço independentemente de todas as outras equipes;
- Melhora o isolamento de falhas. Por exemplo, se há problema de memória em um dos serviços, apenas esse será afetado. Os outros serviços continuarão disponíveis para atender as requisições.

Além desses benefícios do uso de micro serviços, podem ser incluídos também os que Fowler (2015) menciona: reforçam a estrutura modular, que é particularmente importante em grandes equipes e, com micro serviços é possível misturar diversas linguagens, *frameworks* de desenvolvimento e tecnologias de armazenamento de dados.

Porém, há alguns inconvenientes relacionados ao uso de micro serviços (RICHARDSON, 2014-2):

- Desenvolvedores devem lidar com a complexidade adicional de criar um sistema distribuído:
    - Ferramentas e IDEs são orientados a construir aplicações monolíticas e não provêm suporte explícito ao desenvolvimento de aplicações distribuídas;
    - Testes são mais difíceis de serem realizados;
    - Deve ser implementado o mecanismo de comunicação entre serviços;
    - Implementar casos de uso que abrangem vários serviços sem o uso de transações distribuídas é difícil;

    - Implementar casos de uso que abrangem vários serviços requer uma coordenação cuidadosa entre as equipes.
- Quanto à implantação em ambiente de produção, há também a complexidade operacional de implantar e gerenciar um sistema composto por diferentes tipos de serviços;
- Aumento do consumo de memória: a arquitetura de micro serviços substitui N instâncias da aplicação monolítica por NxM instâncias dos serviços.

Quando opta-se por construir uma aplicação como um conjunto de micro serviços, é necessário decidir como os clientes da aplicação irão interagir com os micro serviços. Com uma aplicação monolítica há somente um conjunto de *endpoints* (tipicamente replicado, com balanceamento de carga). Entretanto, em uma arquitetura de micro serviços, cada micro serviço expõe um conjunto de *endpoints* que são tipicamente bem granulados (RICHARDSON, 2015).

Um exemplo com algumas diferenças entre a arquitetura monolítica e a arquitetura de micro serviços pode ser visto na Figura 2. Na arquitetura monolítica, toda a lógica de negócios está contida em um único processo que acessa um banco de dados. Já na arquitetura de micro serviços, o sistema é composto de pequenos serviços independentes, cada um rodando como um processo separado e podendo acessar um banco de dados distinto (ALTEXSOFT, 2017).

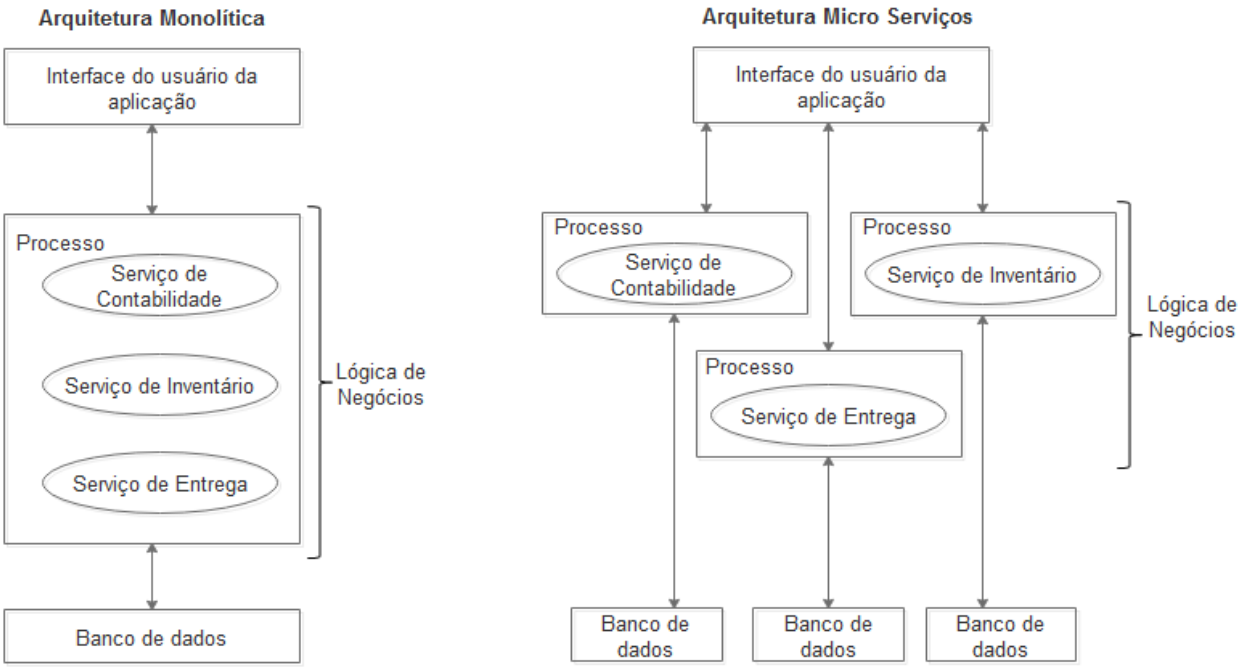

Figura 2 – Exemplo com algumas diferenças entre a arquitetura monolítica e a arquitetura de micro serviços.

Fonte: adaptado de Altexsoft (2017) e Richardson (2014-2)**.**

2.4.2.2 API *Gateway*

Richardson (2015) propõe uma abordagem utilizando uma API *Gateway* para comunicação, conforme ilustrado na Figura 3, entre os clientes (a aplicação móvel e a aplicação web) e os serviços.

Uma API *Gateway* é um servidor que é o único ponto de entrada do sistema e é similar ao padrão Façade (GAMMA et al., 1994). A API *Gateway* encapsula a arquitetura interna do sistema e provê uma API que é adaptada para cada cliente. Deve conter outras responsabilidades como autenticação, monitoramento, balanceamento de carga, *cache*, formato e gerenciamento da requisição e manipulação estática da resposta (RICHARDSON, 2015).

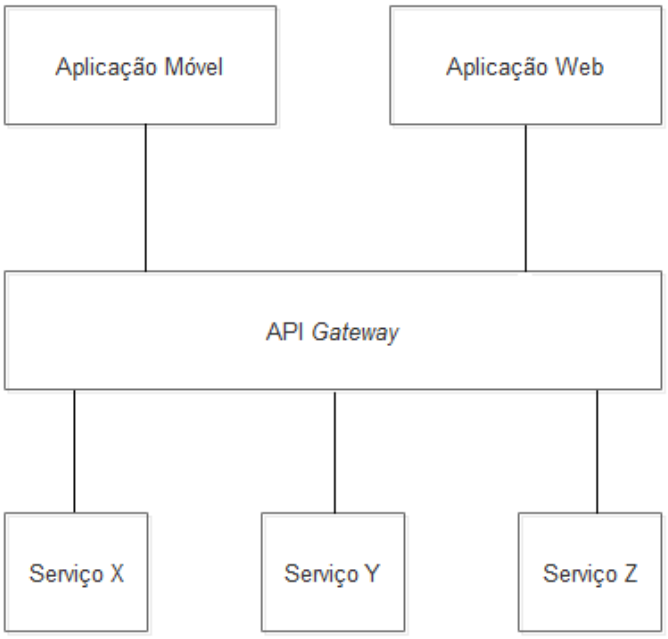

Figura 3 – Abordagem utilizando uma API Gateway para comunicação entre os clientes e os serviços.

Fonte: adaptado de Newman (2015).

A API *Gateway* é responsável pelo roteamento, composição e conversão do protocolo da requisição. Todas as requisições dos clientes primeiro passam pela API *Gateway*. Então, a API *Gateway* encaminha a requisição para o micro serviço apropriado. A API *Gateway* irá frequentemente manipular uma requisição invocando múltiplos micro serviços e agregando os resultados (RICHARDSON, 2015).

O maior benefício do uso da API *Gateway* é que encapsula a estrutura interna da aplicação. Ao invés de ter que invocar serviços específicos, clientes simplesmente conversam com a API *Gateway*. A API *Gateway* provê para cada cliente uma API específica, desta forma reduz o número de requisições entre cliente e API (RICHARDSON, 2015).

Segundo Richardson (2015), a API *Gateway* têm alguns inconvenientes:

- É mais um componente altamente disponível que deve ser desenvolvido, implantado e gerenciado;
- Há o risco dos desenvolvedores terem que esperar em fila para atualizar a API *Gateway*: desenvolvedores devem atualizar a API *Gateway* para expor cada *endpoint* do micro serviço. É importante que o processo de atualização seja o mais leve possível.

2.4.2.3 Spring Boot

O *framework* Spring Boot auxilia no desenvolvimento de micro serviços em Java. Foi construído a partir do *framework* Spring (*framework* de código aberto para a plataforma Java), garantindo todos os seus benefícios e maturidade, e visa a produtividade do desenvolvedor, tornando conceitos como RESTful e ambiente de execução de aplicação web embarcada fáceis de conectar e usar (WOODS, 2015).

Também atua como um "micro-*framework*" permitindo aos desenvolvedores escolherem quais partes do *framework* são necessárias, dessa forma não sobrecarrega o ambiente de execução com dependências que não serão utilizadas. O *framework* Spring Boot também possibilita que aplicações desenvolvidas nele sejam empacotadas em unidades menores para publicação e é habilitado a usar sistemas de *build* para gerar instaláveis como arquivos Java executáveis (JAR) (WOODS, 2015).

Segundo Woods (2015), o *framework* é construído em módulos agregados, conhecidos como "*starters*". Esses módulos são composições de versões interoperáveis de bibliotecas que podem ser usadas para fornecer alguma funcionalidade para uma aplicação. De acordo com Antonov (2015), o Spring Boot disponibiliza mais de 40 módulos diferentes, que provêm bibliotecas de integração com diferentes *frameworks* já prontas para o uso, tais como conexões de banco de dados relacionais e não relacionais, serviços web, integração com redes socias como Facebook, Twitter e LinkedIn, bibliotecas de monitoramento, biblioteca de testes para integração com o JUnit por exemplo, entre muitas outras.

Muitos dos módulos são projetados especificamente para acomodar a arquitetura de micro serviços, expondo funcionalidades chaves para desenvolvedores. Um micro serviço RESTful baseado em HTTP pode ser construído em Spring Boot incluindo os módulos atuador e web. O módulo atuador irá operacionalizar o micro serviço provendo estrutura e *endpoints* para exposição de métricas, parâmetros de configuração e mapeamento de componentes internos, que são úteis para depuração. O módulo web fornecerá o ambiente de execução embarcado e as funcionalidades que permitem a construção da API do micro serviço baseada nos controladores RESTful (também chamados de recursos) (WOODS, 2015).

Cada micro serviço pode ser um projeto Maven com Spring Boot, sendo que Maven[11] é uma ferramenta usada para construir e gerenciar qualquer projeto baseado em Java, de forma a facilitar e organizar o trabalho dos desenvolvedores Java.

Para iniciar a construção de aplicações pode ser utilizado o mecanismo Spring Initializr[12], criado pela equipe do Spring Boot e disponibilizado por meio da Internet. Na página Web do Spring Initializr é possível gerar um projeto Maven com Spring Boot, para isso é preciso informar alguns metadados (grupo e artefato) e as dependências do projeto. Feito isso, o mecanismo é capaz de gerar um pacote de arquivos que correspondem ao projeto Maven com Spring Boot contendo a estrutura básica do projeto para iniciantes do *framework* (WOODS, 2015).

O arquivo “Application.java” funciona como ponto de entrada da aplicação. O Código 1 referente ao arquivo “Application.java” contem o mínimo de código necessário para iniciar o desenvolvimento de um micro serviço. O uso da anotação “@EnableAutoConfiguration”, na linha 3, instrui o Spring Boot a inicializar, configurar e executar a aplicação (WOODS, 2015).

```
1. import org.springframework.boot.SpringApplication;
2. import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
3. @EnableAutoConfiguration
4. public class Main {
5.    public static void main(String[] args) {
6.        SpringApplication.run(Main.class);
7.    }
8. }
```

Código 1 – Código do arquivo “Application.java”.

As configurações e dependências do projeto Maven ficam no arquivo “pom.xml”. Para conexão com banco de dados, as configurações devem ser feitas no arquivo “application.properties”. Um exemplo da implementação de um micro serviço pode ser visto na Seção 3.2.2.

[11] <https://maven.apache.org/what-is-maven.html>

[12] <https://start.spring.io/>

2.5 Desenvolvimento híbrido e funcionamento *offline*

Nesta seção serão abordadas as tecnologias que permitem o desenvolvimento de uma aplicação multiplataforma e com funcionamento *offline*.

2.5.1 HTML5

Hypertext Markup Language[13] (HTML) é a linguagem de marcação central da Internet que foi projetada inicialmente como uma linguagem para descrever semanticamente documentos científicos e, ao longo dos anos, foi adaptada para descrever vários outros tipos de documentos e até mesmo aplicações. A última versão da sintaxe HTML é conhecida como HTML5.

Dispositivos móveis já por um longo tempo suportam armazenamento local de alguma forma, porém recentemente é possível encontrar técnicas padronizadas para a implementação de armazenamento (OEHLMAN; BLANC, 2011).

Dois novos padrões do HTML5 provêm mecanismos para persistência de dados sem necessidade de interagir com qualquer serviço externo ao JavaScript. Essas novas APIs, HTML5 Web *Storage* e Web SQL *Database*, provêm algumas excelentes ferramentas que ajudam a fazer com que aplicações trabalhem em situações *offline* (OEHLMAN; BLANC, 2011).

Segundo Oehlman e Blanc (2011), essencialmente três tipos diferentes de mecanismos de armazenamento do lado do cliente são implementados como parte da especificação do HTML5:

- Web *storage*: frequentemente referida como *local storage*, é um mecanismo do lado do cliente para armazenamento de pares chave/valor. É simples, mas muito eficaz;
- Web SQL *Database*: provê acesso ao banco de dados SQLite-like, que é uma alternativa do lado do cliente para um tradicional sistema de gerenciamento de base de dados relacional que pode ser encontrada do lado do servidor;

[13] <https://www.w3.org/TR/html51/introduction.html#introduction/>

- *Indexed database*: um plano de especificação que foi proposto pela W3C[14] para substituir a atual especificação implementada Web SQL *database*.

O foco deste trabalho é no primeiro mecanismo que será apresentado com mais detalhes na próxima seção.

2.5.1.1 API Web *Storage*

Ambos os objetos *localStorage* e *sessionStorage* implementam a interface *Storage*, que provê os seguintes métodos: *getItem*, *setItem*, *removeItem* e *clear*. Esses quatro métodos compreendem praticamente toda a funcionalidade da API Web *Storage* do HTML5 (OEHLMAN; BLANC, 2011).

Cada item é armazenado nos objetos *localStorage* ou *sessionStorage* passando-se a chave e o valor do item, por meio do método *setItem*. Para o método *getItem* deve ser passada a chave do item que deseja-se recuperar o valor. Para o método *removeItem* deve ser passada a chave do item que deseja-se remover. Já o método *clear* remove todos os itens armazenados no objeto.

Segundo Oehlman e Blanc (2011), em um caso simples, é possível salvar uma cadeia de caracteres ou outros valores de tipos simples. Por exemplo, para criar dois itens no *localStorage* para armazenar preferências:

```
localStorage.setItem('preferences-bgcolor', '#333333');
localStorage.setItem('preferences- textcolor', '#FFFFFF');
```

O mesmo resultado pode ser obtido armazenando apenas um único objeto no *localStorage* (Oehlman e Blanc, 2011):

```
localStorage.setItem('preferences', {
        bgcolor: '#333333',
        textcolor: '#FFFFFF'
});
```

[14] <https://www.w3.org/TR/IndexedDB/>

Para recuperar o objeto de preferências é possível utilizar uma simples chamada (OEHLMAN; BLANC, 2011):

```
var preferences = localStorage.getItem(‘preferences’);
```

Também é possível salvar no Web *Storage* objetos JavaScript utilizando JSON. JSON provê um método eficiente e elegante de armazenamento de dados de objeto JavaScript como uma sequência de texto (*string*). Estando no formato *string*, os dados podem ser enviados para serviços externos ou salvos como pares (OEHLMAN; BLANC, 2011). Um exemplo de como converter um objeto JavaScript em JSON e armazená-lo no objeto *localStorage* seria:

```
var objetoJavaScript = { "nome":"Fulano", "idade":10};
var objetoJSON = JSON.stringify(objetoJavaScript);
localStorage.setItem('pessoa1', objetoJSON);
```

Quando utilizado o objeto *sessionStorage*, os dados armazenados são perdidos ao fechar o navegador/aplicação. Já ao utilizar *localStorage*, os dados armazenados só são removidos com a exclusão manual como, por exemplo:

```
localStorage.removeItem(‘preferences’);
```

No caso acima irá remover apenas o item cuja chave é “*preferences*”. Para remover todos os itens armazenados no objeto *localStorage* pode ser feito da seguinte forma:

```
localStorage.clear();
```

2.5.2 JavaScript, AJAX e jQuery

JavaScript[15] é a linguagem de script[16] (código de programa que não precisa ser pré-processado) usada para adicionar comportamento às páginas web. Pode ser usada para validar os dados de entrada em um formulário, verificando se estão no formato correto, prover a funcionalidade de arrastar e soltar, permitir elementos animados na página como *menus*, entre muitas outras coisas.

XMLHttpRequest[17] é uma API que fornece funcionalidade ao cliente para transferir dados entre um cliente e um servidor. Provê uma maneira fácil de recuperar dados de uma URL sem ter que fazer uma atualização da página inteira. Isso permite que uma página da Web atualize apenas uma parte do conteúdo sem interromper o que o usuário está fazendo. O construtor “XMLHttpRequest()” inicializa um objeto XMLHttpRequest.

Asynchronous JavaScript and XML[18] (AJAX) utiliza o objeto XMLHttpRequest por meio do JavaScript para se comunicar com os servidores. Embora o nome represente JavaScript Assíncrono e XML, as requisições podem ser síncronas e enviar e receber informações em outros formatos além de XML, incluindo JSON, HTML e arquivos de texto. Por poder trabalhar de forma assíncrona, o AJAX possibilita a realização de requisições para obter novos dados sem a necessidade de recarregar um documento (YORK, 2015).

jQuery[19] é uma biblioteca JavaScript rápida, pequena e rica em funcionalidades que disponibiliza uma API de uso facilitado e é compatível com diversos navegadores. Segundo Silva (2013), a palavra-chave que resume o desenvolvimento com jQuery é simplicidade.

jQuery provê suporte ao AJAX, tornando mais simples o desenvolvimento de requisições HTTP. Por exemplo, para fazer uma requisição *GET* de qualquer tipo com AJAX usando jQuery o método genérico utilizado é o *get()*. Cada método é um membro do objeto jQuery, então a chamada ao método *get()* ficaria *$.get()*. Sendo

[15] <https://www.w3.org/wiki/The_web_standards_model_-_HTML_CSS_and_JavaScript/>

[16] <https://www.w3.org/standards/webdesign/script>

[17] <https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest>

[18] <https://developer.mozilla.org/en-US/docs/AJAX/Getting_Started>

[19] <https://jquery.com/>

que jQuery está contido dentro da variável cifrão, $, que é uma variável JavaScript (YORK, 2015). Um exemplo de como usar o método *get()* seria:

```
$.get(url, function (dados) {
    //tratamento dos dados retornados da requisição
});
```

2.5.3 Electron

Electron[20] é um *framework* para a criação de aplicações híbridas *desktop* (compatíveis com Windows, Mac e Linux) a partir de tecnologias web como JavaScript, HTML e CSS.

Segundo Patel (2015), Electron é composto de dois projetos populares:

- Chromium: é o mais famoso projeto de navegador da equipe do Google e é um projeto de código aberto. É responsável pela renderização da aplicação;
- IO js: é uma plataforma JavaScript construída em tempo de execução do V8[21], que é um mecanismo JavaScript de código aberto e alto desempenho da Google. IO js é responsável pelo acesso aos recursos da plataforma nativa como métodos de API.

De acordo com Patel (2015), a aplicação *desktop* construída pelo *framework* Electron é controlada principalmente por dois diferentes processos:

- Processo principal: é o principal processo em uma aplicação Electron e responsável pela criação de páginas web usando instâncias *BrowserWindow* (classe que representa uma janela do navegador) e um recurso URL. Em outras palavras, o processo principal é o processo Navegador;
- Processo renderizador: é responsável por exibir páginas web em *BrowserWindow*. Usa a arquitetura de multiprocesso do Chromium[22]. Cada página web tem seu próprio processo renderizador.

[20] <http://electron.atom.io/>

[21] <https://developers.google.com/v8/>

[22] <http://dev.chromium.org/developers/design-documents/multi-process-architecture>

2.5.4 Crosswalk

O projeto Crosswalk é um framework para a criação de aplicações híbridas para dispositivos móveis a partir de tecnologias web como JavaScript, HTML5 e CSS.

Muitas aplicações híbridas para dispositivos móveis precisam não apenas de um navegador, mas de todos os recursos de uma aplicação nativa. Há muitas APIs que aplicações móveis nativas (aplicações nativas em dispositivos móveis) acessam, mas que são ausentes ou são pouco suportadas em tempo de execução para aplicações móveis híbridas (aplicações híbridas em dispositivos móveis). O Projeto Crosswalk tenta resolver essas discrepâncias e fechar a lacuna entre as aplicações nativa e web (FISCHER, 2016).

Por exemplo, SIMD é uma tecnologia que permite uma aplicação executar múltiplas instruções em paralelo. Essa capacidade costumava estar disponível apenas para aplicações nativas. Recentemente, uma biblioteca JavaScript tornou essa capacidade acessível para aplicações web. O projeto Crosswalk permite que uma aplicação web possa usar esse recurso de ponta em todos os dispositivos e melhorar muito o desempenho (SPENCER, 2015).

Apache Cordova é um conjunto de APIs de dispositivos em JavaScript que fornece aos desenvolvedores de aplicações móveis, desenvolvidas com HTML5, acesso às funcionalidades específicas de cada plataforma, por exemplo acelerômetro, câmera, GPS, entre outros. O projeto Crosswalk suporta APIs Cordova que sejam da versão 3 ou superiores (FISCHER, 2016).

2.6 Trabalhos relacionados

Nesta seção, serão apresentados alguns trabalhos que utilizam sincronização de dados entre as aplicações nos dispositivos com um local para centralização de informações disponibilizado por meio da Internet. Entre eles, alguns tratam também da questão do funcionamento da aplicação quando o dispositivo está sem acesso à Internet e/ou da questão do desenvolvimento de aplicações multiplataforma.

O primeiro trabalho é o de Ijtihadie et al. (2010), da área de Educação, que propõe um protótipo de uma aplicação web *offline* para sincronização de tarefa/atribuição do Ambiente Virtual de Aprendizagem (AVA). Segundo Ijtihadie et al. (2010), utilizando o AVA, um professor pode criar um conjunto de questões com a ferramenta de questionário e depois agrupá-las formando um questionário *online* que pode ser atribuído como tarefa para cada aluno inscrito no curso. O aluno poderá escolher entre responder o questionário disponibilizado *online*, no período em que estiver na escola, ou *offline*, no período em que estiver na sua casa, por exemplo, utilizando seu telefone móvel (IJTIHADIE et al., 2010).

Considerando que o telefone móvel de cada aluno é capaz de acessar a rede da escola por meio da infraestrutura sem fio, o aluno poderá sincronizar o questionário por meio da aplicação proposta, Quizpoint, levá-lo para responder em sua casa e, antes da data limite para a submissão do questionário respondido estipulada pelo professor, deve submetê-lo em algum momento que estiver na escola (IJTIHADIE et al., 2010).

Para a implementação da aplicação móvel, Ijtihadie et al. (2010) utilizaram o iPhone da Apple e a funcionalidade Web *Storage* do HTML5. No primeiro acesso do usuário, depois de ter obtido o identificador único da aplicação Quizpoint, o aluno deverá obter a Aplicação Web *Offline* por meio da URL mencionada na aplicação Quizpoint acessando pelo computador. Uma vez que o aluno acesse a URL, o telefone móvel fará o *download* de todos os arquivos relacionados à aplicação. Depois de completado o *download*, a aplicação é capaz de operar *offline* (IJTIHADIE et al., 2010).

O trabalho de Ijtihadie et al. (2010) relata que existe um botão para a submissão do questionário finalizado, para que o aluno submeta a tarefa quando estiver na escola, com seu telefone móvel conectado à rede da escola, porém não menciona como é feita a sincronização no momento da submissão da tarefa ao AVA. Outro ponto a ser observado também, que não consta no trabalho, é na inserção pelo professor, por exemplo, de uma nova questão ao questionário no AVA, o que aconteceria caso o aluno já tivesse feito o *download* e respondido o questionário, se teria como fazer o *download* apenas da nova questão, ou se teria que fazer o *download* de todo o questionário e responder novamente a todas as questões.

Outro trabalho é o de Peters et al. (2011), que propõe um protocolo para replicação de base de dados em ambientes móveis utilizando REST. Para Peters et

al. (2011), devido ao grande aumento de dispositivos móveis e a capacidade de acessar dados de qualquer lugar e em qualquer horário utilizando a internet, replicação tornou-se uma importante técnica para melhorar o desempenho, disponibilidade e escalabilidade de sistemas distribuídos em ambientes móveis.

O protocolo proposto por Peters et al. (2011), chamado *Client Centric Replication*, para replicação de base de dados relacionais do servidor web no cliente que, no caso, é o dispositivo móvel, não utiliza mecanismos de bloqueio e tem como objetivo também o funcionamento *offline*. Posteriormente, para a sincronização, só os dados alterados serão sincronizados entre servidor e cliente, utilizando a interface REST, e a responsabilidade pela resolução de possíveis conflitos fica no lado do cliente.

Na arquitetura proposta por Peters et al. (2011), que consiste em um único servidor e um número arbitrário de clientes, no lado do cliente existe a aplicação móvel e o banco de dados replicado e do lado do servidor a aplicação web RESTful e o banco de dados. A operação de sincronização entre a aplicação móvel e a aplicação web RESTful é separada em duas partes (PETERS et al., 2011):

- *Master-to-client-integration*: requisita os dados modificados do servidor e realiza o processamento no cliente;
- *Client-to-master-integration*: envia apenas os dados modificados para o servidor. Existe uma estrutura de log no cliente para identificar os dados que foram modificados.

Um ponto crítico a ser observado é que não há exemplos de tamanhos limites de base de dados relacionais no trabalho de Peters et al. (2011) de forma que a replicação da base no dispositivo móvel seja possível e não acarrete em mau funcionamento do aparelho, já que, na maioria das vezes, o espaço de armazenamento disponível em dispositivos móveis é bem limitado em comparação ao espaço de armazenamento disponível em um servidor.

Outro trabalho é o de Lomotey e Deters (2013), da área médica, em colaboração com o setor de Geriatria do hospital City em Saskatoon, Canadá. Lomotey e Deters (2013) propõem a aplicação Med *App* para dispositivos como *tablet*, *smartphone* e computador, abordando a questão de aplicação multiplataforma, e uma camada de *middleware* para comunicação com o Sistema de Informação de Saúde, já existente do hospital, em que há alguns componentes distribuídos e descentralizados.

A camada móvel consiste na aplicação para os dispositivos. Médicos e demais profissionais de saúde podem instalar a aplicação para interagirem com o sistema e acessar as informações médicas que desejarem, como dados demográficos do paciente, sinais vitais, consultas, etc. É utilizado *cache* para armazenar as informações médicas de forma que possibilite o funcionamento *offline* da aplicação em caso de falta de conexão com a Internet. A aplicação é multiplataforma e para a avaliação foi utilizado o iPad da Apple. (LOMOTEY; DETERS, 2013).

A camada *middleware* utiliza a abordagem REST, fazendo com que diferentes dados médicos possam ser manipulados usando os verbos do HTTP (LOMOTEY; DETERS, 2013).

Nesse trabalho de Lomotey e Deters (2013), não são detalhadas quais tecnologias foram empregadas para o desenvolvimento da aplicação para os dispositivos, apenas menciona que utiliza *cache* e padrão de projeto para aplicações Web. Os mesmos autores, em um outro trabalho, Lomotey e Deters (2014), relatam que utilizaram HTML5 para o desenvolvimento da aplicação multiplataforma Med *App* e base de dados local no dispositivo para armazenamento das mesmas informações médicas citadas no trabalho anterior para funcionamento *offline*. Porém, não há detalhamento de como é feita a sincronização dos dados.

O trabalho de Kazi e Deters (2013) apresenta uma aplicação móvel de saúde chamada *Pain Information on the Go* (PinGO) desenvolvida com a colaboração do Laboratório de Pesquisa em Bioinformática da Universidade de Saskatchewan para pacientes com Artrite Idiopática Juvenil (AIJ). PinGO é um diário da dor em forma de questionário desenvolvido para examinar e avaliar a melhoria dos músculos de pacientes com AIJ, com idade entre 8 e 18 anos, em um treinamento de resistência conduzido por um período de 6 semanas.

O resultado do experimento visa prover recomendações sobre o tipo de exercício e sua frequência que pacientes precisam praticar para controlar a dor, reduzindo a inflamação e melhorando o músculo e sua condição de saúde como um todo. Ao longo do treinamento, foi concedido um *tablet* Android com a aplicação para cada paciente. A aplicação consiste em 3 tipos diferentes de questionários de exercício, chamados: Exercício Diário, Antes do exercício e Depois do exercício (KAZI; DETERS, 2013).

Para o desenvolvimento da aplicação foram utilizadas algumas tecnologias Web como o HTML5, JavaScript, jQueryMobile (versão 1.2) e, para armazenamento, foi utilizada a base de dados relacional Web SQL embutida no navegador, por meio do *framework* de código aberto PhoneGap. Os clientes, que são os dispositivos com a aplicação instalada, se comunicarão com os serviços web RESTful hospedados no servidor, que se comporta como uma camada de *middleware* entre os clientes e a camada hospedada em nuvem que gerencia os eventos e os canais lógicos que representam um grupo ou conjunto de grupos de interesses em determinado tipo de dado (KAZI; DETERS, 2013). No trabalho de Kazi e Deters (2013) a aplicação é multiplataforma, só não é tratada a questão de funcionamento *offline* da aplicação.

O trabalho de Guedes, Junior e Oliveira (2016) propõe um *framework* chamado Offdroid que fornece um mecanismo de persistência, replicação e sincronização de dados, contemplando a criação, exclusão, atualização e exibição de dados persistentes ou requeridos, mesmo se o dispositivo móvel não estiver conectado à rede.

Offdroid é baseado na plataforma Android e objetiva apoiar o desenvolvimento de aplicações que necessitam adaptar-se à instabilidade de acesso à rede. As aplicações serão capazes de executar as operações sem necessidade de conexão com a internet, usando apenas o banco de dados local. A comunicação entre o dispositivo móvel e o servidor de aplicação é feita usando REST (GUEDES; JUNIOR; OLIVEIRA, 2016):

- Quando o *framework* executa uma requisição REST no servidor de aplicação, é feita uma cópia dos dados retornados no banco de dados do dispositivo para que os dados possam ser usados quando o dispositivo estiver sem conexão com a Internet;
- Assim que o dispositivo tiver conexão com a Internet, uma tarefa assíncrona verifica se houve alguma modificação no banco de dados local quando o dispositivo estava *offline* (sem conexão com a Internet). Em caso afirmativo, a comunicação é feita para que haja a sincronização de dados.

No trabalho de Guedes, Junior e Oliveira (2016) o *framework* proposto utiliza banco de dados local para armazenamento, não especificam o que é utilizado no servidor e o *framework* é baseado na plataforma Android. Então, é tratada a questão do funcionamento *offline* da aplicação, porém não é tratada a questão multiplataforma.

2.7 Considerações Finais

Micro serviços são pequenos serviços autônomos que podem ser desenvolvidos e implantados de forma independente e, em conjunto, compõem a lógica de negócios da aplicação. Podem ser desenvolvidos como pequenos serviços web seguindo o estilo arquitetural REST. Serviços web que seguem as boas práticas de REST são conhecidos como serviços web RESTful. Com o *framework* Spring Boot é possível criar micro serviços em Java.

Ao utilizar HTML5 e JavaScript no desenvolvimento de uma aplicação, é possível por meio de tecnologias tais como Electron e Crosswalk, disponibilizar a aplicação para diversas plataformas, fazendo com que a aplicação comporte-se como nativa no dispositivo.

Quanto aos trabalhos relacionados, o que difere neste trabalho é que a solução proposta tem como objetivo ser uma solução genérica, não focando em atender apenas um único caso real ou uma plataforma específica e, quanto ao armazenamento de dados nos dispositivos, neste trabalho a proposta não é a replicação do banco de dados do servidor no dispositivo e sim que apenas os dados necessários sejam armazenados enquanto a aplicação estiver *offline*.

No próximo capítulo será possível identificar como os conceitos, tecnologias e ferramentas abordados nesse capítulo foram empregados para a realização do trabalho proposto.

Capítulo 3

ARQUITETURA PROPOSTA

Neste capítulo será apresentada a arquitetura proposta neste trabalho e serão apresentados também detalhes da implementação.

3.1 Arquitetura do Trabalho

Este trabalho propõe a criação de uma API baseada em JavaScript e HTML5 a fim de apoiar o desenvolvimento de aplicações multiplataforma híbridas, que são aplicações desenvolvidas com HTML5, CSS e JavaScript. As funcionalidades da API incluem a sincronização de dados e o funcionamento *offline* da aplicação. Propõe também o uso de um servidor web para o armazenamento e centralização de informações, de forma que o servidor esteja acessível para os dispositivos por meio da Internet.

A API é uma biblioteca JavaScript. Dessa forma, o cliente (aplicação no dispositivo) deve utilizá-la para sincronização de dados com o servidor e, também, para o funcionamento *offline* da aplicação.

A comunicação da API com o servidor web para a sincronização se dá por meio de serviços web RESTful. O trabalho inclui também o uso da arquitetura baseada em micro serviços para a criação dos serviços web RESTful. Os micro serviços são hospedados no servidor web. A API envia requisições para a API *Gateway* (também hospedada no servidor web) e a API *Gateway* redireciona tais requisições para o(s) micro serviço(s) apropriado(s). A arquitetura proposta está representada na Figura 4.

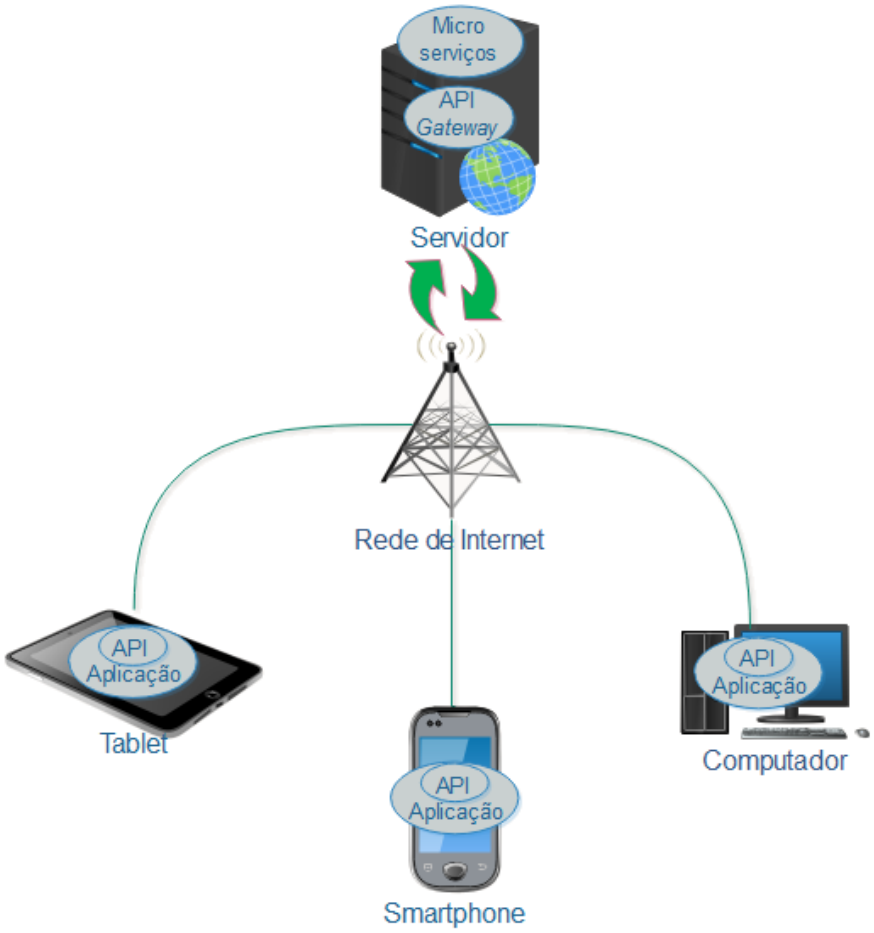

Figura 4 – Arquitetura proposta.

3.2 Implementação

Nesta seção será apresentada, de forma detalhada, a implementação da API, dos micro serviços e da API *Gateway* que fazem parte da arquitetura proposta, bem como as ferramentas e tecnologias que foram empregadas.

3.2.1 API

Para implementar a API proposta foram utilizados HTML5, JavaScript, jQuery e AJAX, que foram explicados na seção 2.3. A API fornece as seguintes funções:

- ***_get(url)***: recupera os dados do recurso. É responsável pela requisição GET do HTTP e recebe como parâmetro a URI para indicar o recurso. A

URI também pode conter a chave identificadora do recurso para recuperar apenas um objeto. Os dados do recurso são representados como um arquivo JSON;

- ***_post(url, json)***: inclui um novo recurso. É responsável pela requisição POST do HTTP e recebe como parâmetro a URI para indicar o recurso e o arquivo JSON que contém os dados do recurso que será incluído;
- ***_put(url, json)***: altera um recurso existente. É responsável pela requisição PUT do HTTP e recebe como parâmetro a URI para indicar o recurso juntamente com a chave identificadora do recurso e o arquivo JSON que contém os dados do recurso que será alterado;
- ***_delete(url)***: remove um recurso existente. É responsável pela requisição DELETE do HTTP e recebe como parâmetro a URI para indicar o recurso juntamente com a chave identificadora do recurso que será removido.

Ao receber uma nova requisição, por meio de uma chamada da aplicação, a API verifica se há conexão com a Internet:

- Em caso positivo, a API enviará todas as requisições pendentes armazenadas no objeto ***localStorage*** do HTML5 *Web Storage* para os micro serviços; A API enviará também a nova requisição que acabou de receber da aplicação e armazenará o conteúdo da resposta utilizando o objeto ***localStorage*** para que a aplicação possa trabalhar *offline*, sem conexão com a Internet, o mais semelhante possível de quando tem conexão com a Internet;
- Em caso negativo, a API verificará se existe resposta armazenada para a mesma requisição no objeto ***localStorage*** do HTML5 *Web Storage* para usá-la, caso contrário, a API armazena a requisição recebida utilizando o objeto ***localStorage*** para ser enviada aos micro serviços assim que houver conexão com a Internet.

Abaixo está o código em JavaScript com a implementação da função ***_post*** (Código 2). Na linha 3 é chamada a função que verifica se há conexão com a Internet. Na linha 4 é chamada a função que verifica se existe alguma requisição

pendente e, em caso positivo, ela será enviada. Na linha 24 é chamada a função que armazena a requisição no caso da inexistência de conexão com a Internet.

```
1. function _post(url, json) {
2. var data;
3. if (doesConnectionExist(url)) {
4.     verifyLocalStorage();
5.     $.ajax({
6.         type: 'POST',
7.         url: url,
8.         contentType: 'application/json; charset=utf-8',
9.         data: JSON.stringify(json),
10.        dataType: 'json',
11.        cache: false,
12.        async: false,
13.        success: function (retorno) {
14.            console.log(retorno);
15.            data = retorno;
16.        },
17.        error: function (xhr) {
18.            console.log(xhr);
19.            dados = JSON.stringify(json);
20.        }
21.    });
22. }
23. else
24.     saveLocal('post', url, json);
25. return data;
26. }
```

Código 2 – Implementação da função _post.

A API, com suas funcionalidades, servirá de apoio ao desenvolvimento de aplicações híbridas (criadas com HTML5, CSS e JavaScript).

Neste trabalho é apoiado o uso do Electron (Seção 2.5.3), para aplicações *desktop*, e o uso do Crosswalk (Seção 2.5.4), para aplicações móveis, para gerar as aplicações para diversas plataformas a partir dos mesmos arquivos que foram desenvolvidos. No entanto, este processo de geração está fora do escopo deste trabalho.

3.2.2 API *Gateway*

Para desenvolver a API *Gateway* foram utilizados o *framework* Spring Boot, Maven e Netflix Zuul[23] (biblioteca de serviços), que encaminhará as requisições recebidas para os micro serviços apropriados. A API *Gateway* fica hospedada no servidor web e apoia a comunicação entre a API e os micro serviços. A API *Gateway* recebe uma requisição da API e redireciona tal requisição para o micro serviço apropriado; esse redirecionamento ocorre baseado no conteúdo do arquivo "application.properties" do projeto da API *Gateway*, conforme pode ser visto como exemplo no Código 3:

```
zuul.routes.contato.path=/contato-ws/**
zuul.routes.contato.url=http://localhost:8083
```

Código 3 – Código do arquivo "application.properties" do projeto da API *Gateway*.

O Código 3 determina que todas as requisições que chegam para a API *Gateway* em "/contato-ws" devem ser redirecionadas para "http://localhost:8083" que é a URI do micro serviço Contato. Então, ao ser implantado um novo micro serviço (Seção 3.2.3) são incluídas também duas linhas no arquivo "application.properties" da API *Gateway*, da mesma forma que no Código 3, porém referentes ao novo micro serviço.

3.2.3 Micro Serviços

Os micro serviços são criados em Java utilizando o *framework* Spring Boot (Seção 2.4.2.3). Os micro serviços são desenvolvidos seguindo o padrão MVC (Seção 2.3). Como nos micros serviços não há interface de usuário, a visão do padrão MVC não é utilizada, apenas modelos e controladores são utilizados.

Os micro serviços são hospedados no servidor web que utiliza Apache Tomcat[24]. Para o armazenamento de dados no servidor é utilizado o banco de dados MySQL. Então, os micro serviços comunicam-se com o banco de dados MySQL para consultar, incluir, alterar e remover dados.

[23] <https://spring.io/guides/gs/routing-and-filtering/>

[24] <http://tomcat.apache.org/>

Para exemplificar a implementação de um micro serviço, o Código 4 exibe o conteúdo da classe “Contato.java”, que é o modelo do padrão MVC, e o Código 5 exibe o conteúdo da classe “ContatoController.java”, que é o controlador do padrão MVC, sendo ambas as classes pertencentes ao micro serviço Contato. É utilizado também o padrão *Data Access Object* (DAO) na classe “ContatoDAO.java” (Código 6). Segundo Elliott, O'Brien e Fowler (2008), o padrão DAO ajuda a isolar o código da aplicação do código que acessa e manipula registros em um banco de dados.

Na classe “Contato.java” (Código 4) são utilizadas notações Java Persistence API[25] (JPA) como, por exemplo, “@Entity” (Linha 9)” e “@Table (name = "contato")” (Linha 10) que indicam que a classe Contato refere-se à tabela contato (Linha 10) do banco de dados.

Na classe “ContatoDAO.java” é utilizado Spring Data JPA[26] para armazenar e recuperar dados em um banco de dados relacional, como pode ser visto no Código 6 (Linha 6). O banco de dados utilizado foi o MySQL. As configurações de banco de dados ficam no arquivo “application.properties”.

```
1. package br.com.contato.model;
2. import java.io.Serializable;
3. import javax.persistence.Column;
4. import javax.persistence.Entity;
5. import javax.persistence.GeneratedValue;
6. import javax.persistence.GenerationType;
7. import javax.persistence.Id;
8. import javax.persistence.Table;

9. @Entity
10.@Table(name = "contato")
11.public class Contato implements Serializable{
12.     @Id
13.     @GeneratedValue(strategy = GenerationType.IDENTITY)
14.     @Column(name="id")
15.     private int id;

16.     @Column(name="nome")
17.     private String nome;

18.     @Column(name="telefone")
```

[25] <http://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html>

[26] <https://spring.io/guides/gs/accessing-data-jpa/>

```
19.     private String telefone;

20.     @Column(name="email")
21.     private String email;

22.     public String getNome() {
23.             return nome;
24.     }
25.     public void setNome(String nome) {
26.             this.nome = nome;
27.     }

28.     public String getTelefone() {
29.             return telefone;
30.     }
31.     public void setTelefone(String telefone) {
32.             this.telefone = telefone;
33.     }

34.     public String getEmail() {
35.             return email;
36.     }
37.     public void setEmail(String email) {
38.             this.email = email;
39.     }

40.     public int getId() {
41.             return id;
42.     }
43.     public void setId(int id) {
44.             this.id = id;
45.     }

46.}
```

Código 4 – Classe “Contato.java”.

```
1. package br.com.contato.controller;
2. import java.util.List;
3. import org.springframework.beans.factory.annotation.Autowired;
4. import org.springframework.http.HttpStatus;
5. import org.springframework.http.ResponseEntity;
6. import org.springframework.web.bind.annotation.CrossOrigin;
7. import org.springframework.web.bind.annotation.PathVariable;
8. import org.springframework.web.bind.annotation.RequestBody;
9. import org.springframework.web.bind.annotation.RequestMapping;
10.import org.springframework.web.bind.annotation.RequestMethod;
11.import org.springframework.web.bind.annotation.RestController;
12.import br.com.contato.dao.ContatoDAO;
```

```
13.import br.com.contato.model.Contato;

14.@RestController
15.@RequestMapping(value = ContatoController.PATH)
16.public class ContatoController {

17.     public static final String PATH = "/contato";
18.     @CrossOrigin
19.     @RequestMapping(method = RequestMethod.GET)
20.     public ResponseEntity<List<Contato>> get () {
21.             return new ResponseEntity<List<Contato>>(contatoDAO.findAll(),
                                                                HttpStatus.OK);
22.     }

23.     @CrossOrigin
24.     @RequestMapping(value="/{id}", method = RequestMethod.GET)
25.     public ResponseEntity<Contato> get (@PathVariable("id") int id) {
26.             return new ResponseEntity<Contato>(contatoDAO.findOne(id),
                                                        HttpStatus.OK);
27.     }

28.     @CrossOrigin
29.     @RequestMapping(value="/{id}", method = RequestMethod.DELETE)
30.     public HttpStatus delete (@PathVariable("id") int id) {
31.             contatoDAO.delete(id);
32.             return HttpStatus.OK;
33.     }

34.     @Autowired
35.     private ContatoDAO contatoDAO;

36.     @CrossOrigin
37.     @RequestMapping(method = RequestMethod.POST)
38.     public ResponseEntity<Contato> save (@RequestBody final Contato
                                                    contatoEntity)
39.     {
39.             return new ResponseEntity<Contato>(contatoDAO.save(contatoEntity),
                                                        HttpStatus.CREATED);
40.     }

41.     @CrossOrigin
42.     @RequestMapping(value="/{id}", method = RequestMethod.PUT)
43.     public ResponseEntity<Contato> alter (@PathVariable("id") int id,
                                            @RequestBody final Contato contatoEntity)
44.     {
45.             return new ResponseEntity<Contato>(contatoDAO.save(contatoEntity),
                                                        HttpStatus.OK);
46.     }
47.}
```

Código 5 – Classe “ContatoController.java”.

```
1. package br.com.contato.dao;
2. import org.springframework.data.jpa.repository.JpaRepository;
3. import org.springframework.stereotype.Repository;
4. import br.com.contato.model.Contato;

5. @Repository
6. public interface ContatoDAO extends JpaRepository<Contato, Integer>{
7. }
```

Código 6 – Classe "ContatoDAO.java".

Os micro serviços seguem os conceitos do estilo arquitetural REST. Sendo assim, a interação com os micro serviços é feita por meio de métodos HTTP: *GET*, *POST*, *PUT* e *DELETE*. As requisições devem ser feitas para a API *Gateway* que redirecionará para o micro serviço apropriado.

3.2.3.1 Método *GET*

Responsável por retornar o(s) recurso(s). A requisição pode vir ou não acompanhada do identificador do recurso.

- Sem o identificador do recurso na URL: A requisição deve ser feita no seguinte formato: "http://<endereco da API *Gateway*>/<recurso>". Retorna o status HTTP, por exemplo status 200 para sucesso, e o JSON contendo uma lista com todos os recursos disponíveis (armazenados).

  Exemplo da requisição para o micro serviço responsável por contatos: "URL: http://localhost:8080/api-gateway/contato". Dessa forma, a API *Gateway* redirecionará a requisição para o micro serviço Contato (tal requisição será tratada pela Linha 20 do Código 5) e o JSON de retorno do micro serviço Contato será:

```
JSON de retorno:
[
   {
      "id": 1,
      "nome": "contato01",
      "telefone": "(xx) 99999-1111",
      "email": "teste@teste.com"
```

```
  },
  {
     "id": 2,
     "nome": "contato02",
     "telefone": "(xx) 99999-2222",
      "email": "teste2@teste.com
  },
  {
     "id": 3,
     "nome": "contato03",
     "telefone": "(xx) 99999-1111",
      "email": "teste@teste.com"
  }
]
```

- Com o identificador do recurso na URL: A requisição deve ser feita no seguinte formato: “http://<endereco da API *Gateway*>/<recurso>/<identificador do recurso>”. Retorna o status HTTP, por exemplo 200 para sucesso, e o JSON contendo o recurso solicitado, quando houver.

Exemplo da requisição para o micro serviço responsável por contatos: “URL: http://localhost:8080/api-gateway/contato-ws/contato/1”. Dessa forma, a API *Gateway* redirecionará a requisição para o micro serviço Contato (tal requisição será tratada pela Linha 25 do Código 5) e o JSON de retorno do micro serviço Contato será:

```
JSON de retorno:
{
  "id": 1,
  "nome": "contato01",
  "telefone": "(xx) 99999-1111",
  "email": "teste@teste.com”
}
```

3.2.3.2 Método *POST*

Responsável pela criação de um novo recurso. A requisição deve ser feita no seguinte formato: “http://<endereco da API Gateway>/<recurso>” e deve conter também no corpo (*body*) da requisição o JSON com as informações para a criação do novo recurso. Retorna o status HTTP, por exemplo status 201 para quando a solicitação tiver sido concluída com sucesso, e o JSON contendo o recurso criado.

Exemplo da requisição e do corpo da requisição para o micro serviço responsável por contatos: “URL: http://localhost:8080/api-gateway/contato”.

```
JSON (corpo da requisição):
{
   "nome": "novo contato",
   "telefone": "(xx) 99999-xxxx",
   "email": "testex@teste.com"
 }
```

Dessa forma, a API *Gateway* redirecionará a requisição para o micro serviço Contato (tal requisição será tratada pela Linha 38 do Código 5) e o JSON de retorno do micro serviço Contato será:

```
JSON de retorno:
{
   "id": 44,
   "nome": "novo contato",
   "telefone": "(xx) 99999-xxxx",
   "email": "testex@teste.com"
}
```

3.2.3.3 Método *PUT*

Responsável pela alteração de um recurso já existente. A requisição deve ser feita no seguinte formato: http://<endereco da API Gateway>/<recurso>/<identificador do recurso> e deve conter no corpo (*body*) da

requisição o JSON com as informações do recurso a ser alterado. Retorna o status HTTP, por exemplo 200 para quando a solicitação tiver sido concluída com sucesso, e o JSON contendo o recurso alterado.

Exemplo da requisição e do corpo da requisição para o micro serviço responsável por contatos: “URL: http://localhost:8080/api-gateway/contato/1”.

```
JSON (corpo da requisição):
{
        "id": 1,
        "nome": "contato01alterado",
        "telefone": "(xx) 99999-1111",
        "email": "teste@teste.com"
 }
```

Dessa forma, a API *Gateway* redirecionará a requisição para o micro serviço Contato (tal requisição será tratada pela Linha 43 do Código 5) e o JSON de retorno do micro serviço Contato será:

```
JSON de retorno:
{
   "id": 1,
   "nome": "contato01alterado",
   "telefone": "(xx) 99999-1111",
   "email": "teste@teste.com"
 }
```

3.2.3.4 Método *DELETE*

Responsável pela exclusão de um recurso. A requisição deve ser feita no seguinte formato: http://<endereco da API Gateway>/<recurso>/<identificador do recurso>. Retorna o status HTTP, por exemplo status 200 quando a solicitação for atendida com sucesso. Exemplo da requisição para o micro serviço responsável por contatos: “URL: http://localhost:8080/api-gateway/contato/45”. Dessa forma, a API

Gateway redirecionará a requisição para o micro serviço Contato (tal requisição será tratada pela Linha 30 do Código 5).

3.2.4 Ferramenta e Script

Para otimizar a criação dos micro serviços, foi desenvolvido neste trabalho também uma ferramenta, implementada em Java, que a partir da classe contendo apenas os atributos a serem gerenciados (armazenados), produz todos os arquivos necessários que compõem o projeto do micro serviço correspondente, conforme ilustrado na Figura 5.

A ferramenta recebe também o diretório em que está o template – que é o modelo, também desenvolvido neste trabalho, que a ferramenta utilizará para produzir os arquivos necessários do projeto do micro serviço – e o diretório indicando em que deverão ser salvos os arquivos que compõem o projeto do micro serviço. Para produzir o projeto do micro serviço a partir do template e da classe em Java (contendo apenas os atributos) a ferramenta utiliza JavaParser[27].

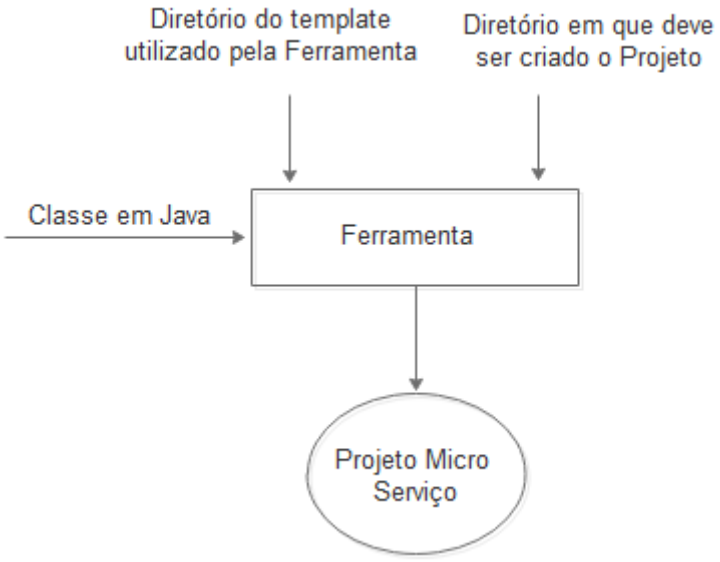

Figura 5 – Criação do projeto do micro serviço.

Para exemplificar a classe em Java com os atributos, na Figura 6 está representada a classe Contato com seus atributos (arquivo nomeado “Contato.java”). Os arquivos “Contato.java”, “ContatoController.java” e “ContatoDAO.java” referentes

[27] <http://javaparser.org/>

ao Código 4, Código 5 e Código 6, respectivamente, que foram discutidos anteriormente, fazem parte do projeto do micro serviço Contato e foram produzidos por essa ferramenta.

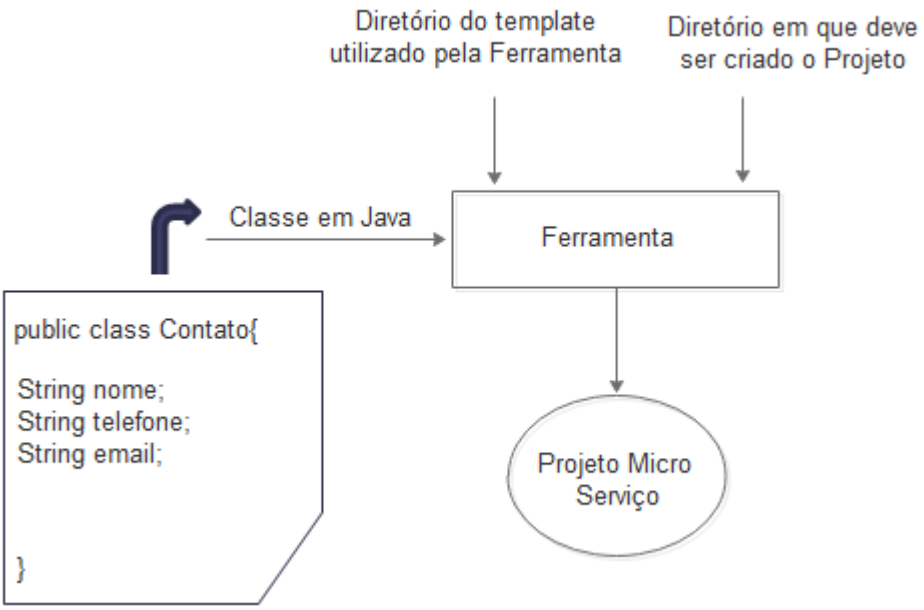

Figura 6 – Criação do projeto do micro serviço Contato.

Com o projeto do micro serviço gerado, é necessário compilá-lo. Após a compilação do projeto, um arquivo .WAR é gerado e colocado na pasta de aplicações do Apache Tomcat, como pode ser visto na Figura 7. Com isso, o micro serviço já está implantado e pronto para ser utilizado. A URL referente ao micro serviço criado e implantado será composta por: "<endereço da API *Gateway*>/<nome da classe em Java>". Por exemplo, para o micro serviço Contato, a URL é "<endereço da API *Gateway*>/Contato".

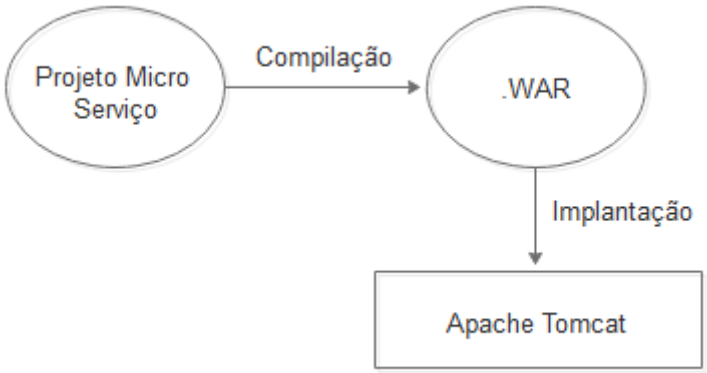

Figura 7 – Implantação do micro serviço.

Neste trabalho foi criado também um script que executa a ferramenta desenvolvida e, a partir do projeto do micro serviço gerado pela ferramenta (Figura 5 e Figura 6) realiza a compilação e implantação do micro serviço no Apache Tomcat (Figura 7). O script também insere no arquivo “application.properties” da API *Gateway* as duas linhas mencionadas na Seção 3.2.2, para que a API *Gateway* já esteja pronta para receber requisições e redirecioná-las ao novo micro serviço.

O script e a ferramenta devem ser executados no servidor web, que utiliza Apache Tomcat.

3.3 Considerações Finais

Nesse capítulo foram apresentados a arquitetura proposta e detalhes da implementação deste trabalho. A API foi desenvolvida utilizando HTML5, JavaScript, jQuery e AJAX. Para a criação da API *Gateway* foram utilizados o *framework* Spring Boot, Maven e Netflix Zuul. Os micro serviços são desenvolvidos em Java, utilizando o *framework* Spring Boot. Para automatizar a criação dos micro serviços foram desenvolvidos uma ferramenta e um script neste trabalho que a partir da classe em Java contendo apenas os atributos a serem gerenciados (armazenados) produz o micro serviço correspondente.

Capítulo 4

ESTUDOS DE CASO

Neste capítulo serão apresentados dois estudos de caso que foram conduzidos para a avaliação da abordagem proposta neste trabalho.

4.1 Considerações Iniciais

O trabalho proposto foi aplicado em dois estudos de caso em cenários distintos. Um deles é com uma aplicação de Gerenciamento de Contatos. O outro é com o jogo Forca do Laboratório de Objetos de Aprendizagem[28] (LOA) da Universidade Federal de São Carlos (UFSCar), em São Carlos. Ambos carecem de sincronização de dados e funcionamento *offline*.

Para a validação da arquitetura proposta foram realizados também testes unitários para os micro serviços utilizando o *framework* de testes para Java: JUnit[29]. Os testes unitários atingiram uma cobertura de 100% para os Controladores (Seção 3.2.3) dos micro serviços.

4.2 Aplicação de Gerenciamento de Contatos

A aplicação de Gerenciamento de Contatos foi criada com CSS, JavaScript e HTML5. A aplicação utiliza a API proposta e tem as seguintes funcionalidades: listar,

[28] <http://www.loa.sead.ufscar.br/>

[29] <http://junit.org/junit5/>

incluir, alterar e remover contatos, de acordo com Figura 8, Figura 9, Figura 10 e Figura 11, respectivamente. A API envia as requisições da aplicação (dispositivo) para a API *Gateway* (servidor web) e suporta o funcionamento *offline* da aplicação. O micro serviço referente à aplicação, para quem a API *Gateway* redirecionará as requisições recebidas da API, foi criado utilizando a ferramenta e script discutidos na Seção 3.2.4.

Figura 8 – Aplicação de Gerenciamento de Contatos.

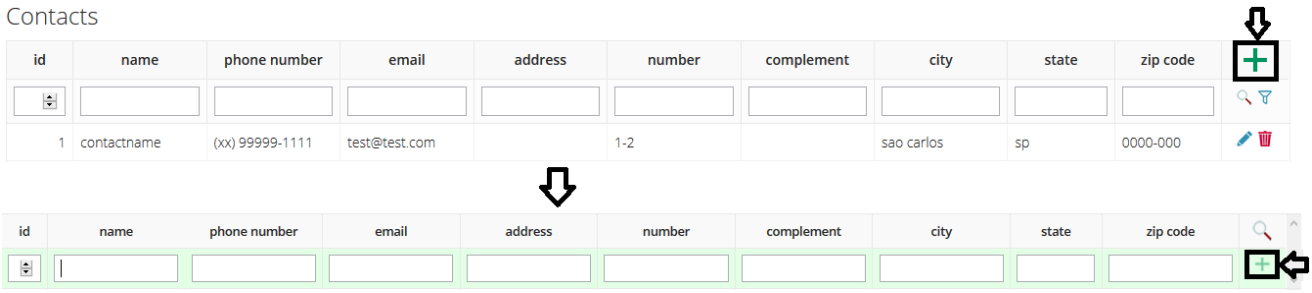

Figura 9 – Aplicação de Gerenciamento de Contatos: Para incluir um novo contato.

Figura 10 – Aplicação de Gerenciamento de Contatos: Para alterar um contato.

Figura 11 – Aplicação de Gerenciamento de Contatos: Para remover um contato.

Para listar todos os contatos (Figura 8) a aplicação chama a função "***_get***" da API, como pode ser visto abaixo na linha 2 do Código 7 (em JavaScript):

```
1. function loadData () {
```

```
2. json = _get('http://localhost:8080/api-gateway/contato-ws/contato');
3. db.contatos = json;
4. return json;
5. }
```

Código 7 – Implementação da função "*loadData*" da aplicação de Gerenciamento de Contatos.

Abaixo está o código JavaScript da aplicação com a implementação da função "***insertItem***" – responsável por incluir um novo contato (Figura 9) – que chama a função "***_post***" da API na linha 3 do Código 8 (em JavaScript).

```
1. function insertItem(insertingContact) {
2. var json_post = { "name": insertingContact.name,
        "phone_number": insertingContact.phone_number,
        "email": insertingContact.email,
        "address": insertingContact.address,
        "number": insertingContact.number,
        "complement": insertingContact.complement,
        "city": insertingContact.city,
        "state": insertingContact.state,
        "zip_code": insertingContact.zip_code };
3. var ret = _post('http://localhost:8080/api-gateway/contact-ws/contact', json_post);
4. if (ret != undefined) {
5.      insertingContact.id = ret.id;
6. }
7. this.contatos.push(insertingContact);
8. }
```

Código 8 – Implementação da função "*insertitem*" da aplicação de Gerenciamento de Contatos.

Para alterar um contato (Figura 10) a aplicação chama a função "***_put***" da API conforme pode ser visto na linha 3 do Código 9 (em JavaScript).

```
1. function updateItem (updatingContact) {
```

```
2. var json_put = { "name": updatingContact.name,
        "phone_number": updatingContact.phone_number,
        "email": updatingContact.email,
        "address": updatingContact.address,
        "number": updatingContact.number,
        "complement": updatingContact.complement,
        "city": updatingContact.city,
        "state": updatingContact.state,
        "zip_code": updatingContact.zip_code };
3. _put('http://localhost:8080/api-gateway/contato-ws/contato/' + updatingContact.id,
json_put);
4. }
```

Código 9 – Implementação da função *"updateitem"* da aplicação de Gerenciamento de Contatos.

Para remover um contato (Figura 11) a aplicação chama a função "***_delete***" da API como pode ser visto na linha 4 do Código 10 (em JavaScript).

```
1. function deleteItem (deletingContact) {
 2. var contactIndex = $.inArray(deletingContact, this.contatos);
3. this.contatos.splice(contactIndex, 1);
4. _delete('http://localhost:8080/api-gateway/contato-ws/contato/' +
deletingContact.id);
5. }
```

Código 10 – Implementação da função *deleteItem* da aplicação de Gerenciamento de Contatos.

Todas essas funções foram apresentadas na Seção 3.2.1.

A criação da aplicação móvel híbrida foi realizada por meio da ferramenta Intel XDK[30] que utiliza Crosswalk (Seção 2.5.4). Para produzir a aplicação móvel híbrida basta importar seu projeto HTML5 no Intel XDK e escolher para qual plataforma deseja exportar: Android, iOS ou Windows (Figura 12 e Figura 13). Por exemplo, escolhendo-se a plataforma Android são gerados dois arquivos APK para ambas as arquiteturas x86 e ARM.

Os testes foram feitos com os dispositivos conectados e não conectados à Internet. Quando o dispositivo está sem conexão com a Internet o usuário é avisado

[30] <https://software.intel.com/pt-br/intel-xdk>

como na Figura 14, entretanto a aplicação funciona corretamente: o usuário é capaz de incluir, alterar, buscar e remover contatos. Quando a conexão é restabelecida, todas as ações que ficaram pendentes – sem sincronização – devido à falta de conexão são sincronizadas.

Todas as funcionalidades da aplicação funcionaram corretamente em ambas as situações: dispositivo com e sem conexão com a Internet. Do ponto de vista do usuário da aplicação, a perda de conexão não interferiu seu uso da aplicação.

Figura 12 – Aplicação de Gerenciamento de Contatos simulada no dispositivo iPhone 4S da Apple usando a ferramenta Intel XDK.

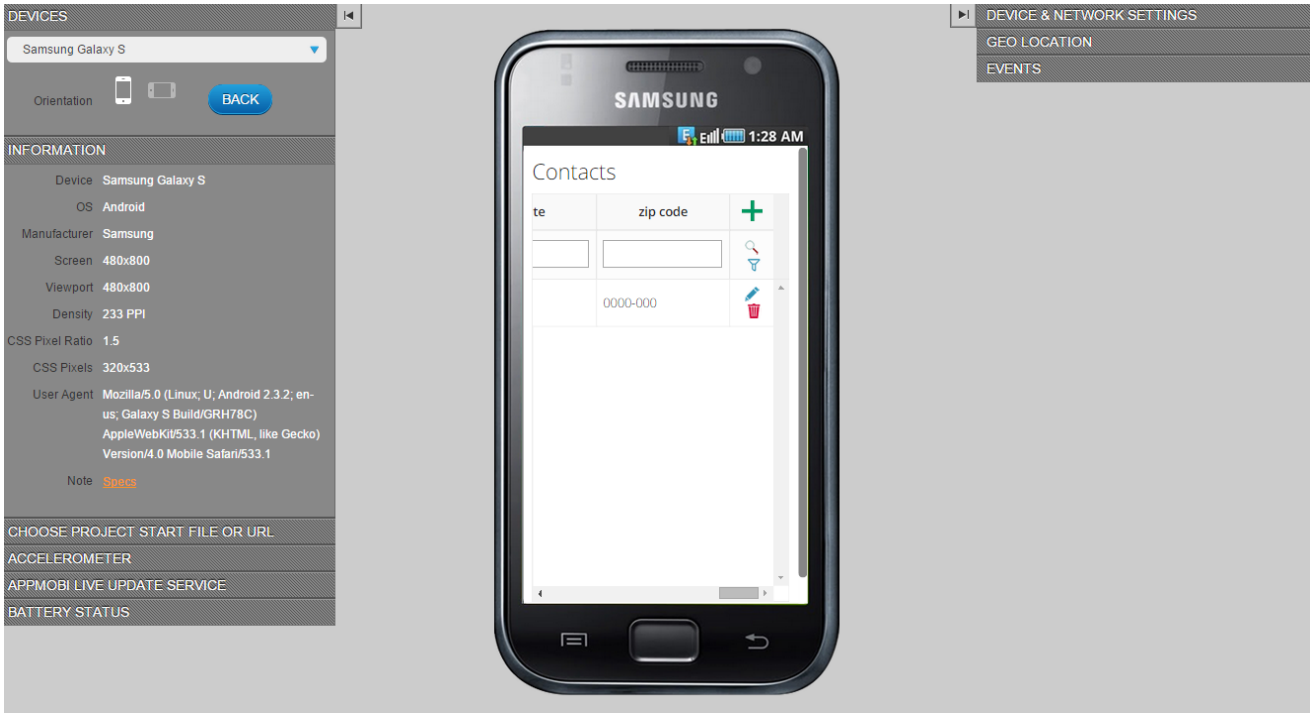

Figura 13 – Aplicação de Gerenciamento de Contatos simulada no dispositivo Galaxy S da Samsung usando a ferramenta Intel XDK.

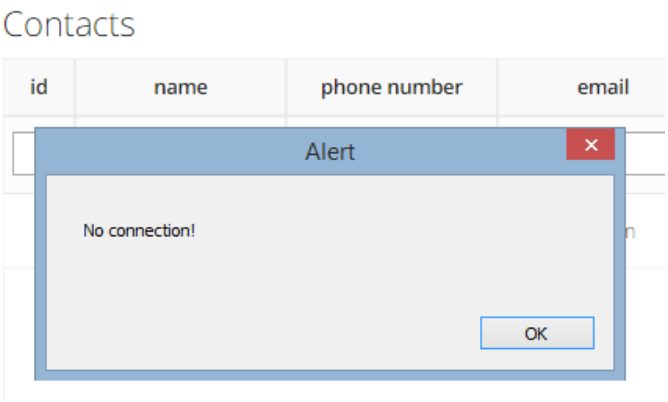

Figura 14 – Aviso de que não há conexão com a Internet

4.3 Jogo Forca

O jogo educacional chamado Forca tem como objetivo colocar as letras até formar a palavra que corresponde à resposta correta ou até que as chances sejam terminadas a partir da pergunta que é dada. Este jogo criado no LOA foi desenvolvido em JavaScript, CSS e HTML e necessita de sincronização de dados e operação *offline*.

A sincronização de dados é necessária para atualizar a pontuação do usuário e o *ranking* de todos os usuários do jogo. O funcionamento *offline* do jogo é necessário para que a pontuação do usuário não seja perdida em caso de falta de conexão com a Internet e também para que o *ranking* do jogo esteja disponível para o usuário, mesmo quando o dispositivo está sem conexão com a Internet.

Portanto, é importante para o jogo usar a API e o mecanismo de sincronização de dados propostos neste trabalho para ter sincronização de dados e funcionamento *offline* com pouco esforço, conforme pode ser visto no Código 11 e no Código 12 que mostram como ficaram as duas funções do jogo que realizam chamadas à API (linha 3 do Código 11 e linha 4 do Código 12) e são responsáveis por salvar a pontuação do jogador e obter o *ranking* do jogo. O jogo funcionou corretamente com e sem conexão com a Internet.

```
1. function salvaPontuacao(jogador, pontos) {
2. var json_post = { "jogador": jogador, "pontos": pontos, "data": new Date() };
3. var retorno = _post('http://localhost:8080/api-gateway/ranking', json_post);
4. }
```

Código 11 – Implementação da função responsável por salvar a pontuação do jogador.

```
1. function obtemRanking(){
2. ranking = [{}];
3. $.ajaxSetup({ async: false });
4. _get('http://localhost:8080/api-gateway/ranking').then(function(data){
5.     ranking = data; // executado após a conclusão da chamada
6. });
7. ranking.sort(compare); //ordena o resultado
8. ranking = ranking.slice(0,5); //exibe apenas os 5 primeiros
9. criarCamadaRanking(); //função responsável pela exibição do ranking para o
                                    //usuário
10. console.log(ranking);
11. $.ajaxSetup({ async: true });
12. }
```

Código 12 – Implementação da função responsável por obter o *ranking* do jogo.

A API envia as requisições para a API *Gateway* que redireciona tais requisições para o micro serviço. O micro serviço *Ranking* foi desenvolvido utilizando a ferramenta e o script discutidos na Seção 3.2.4. A classe “Ranking.java*”*, com os atributos a serem gerenciados (armazenados), utilizada pela ferramenta para criação do micro serviço *Ranking* pode ser vista no Código 13.

```
public class Ranking{
String jogador;
int pontos;
String data;
}
```

Código 13 – Classe “Ranking.java”.

Para testar a aplicação foi utilizada a ferramenta Electron que permite produzir versões *desktop* do jogo. O jogo educativo Forca pode ser visto na Figura 15. Ao clicar no botão Play inicia o jogo como na Figura 16. Então o usuário pode selecionar as letras, uma por uma, como na Figura 17 até que a resposta correta seja formada ou as chances estiverem finalizadas. O usuário tem cinco chances de cometer um erro. A pontuação máxima de cada resposta correta é 5. Cada letra errada subtrai 1 do valor máximo. A pontuação é calculada apenas no final da palavra ou das chances.

Figura 15 – Jogo educacional chamado Forca.

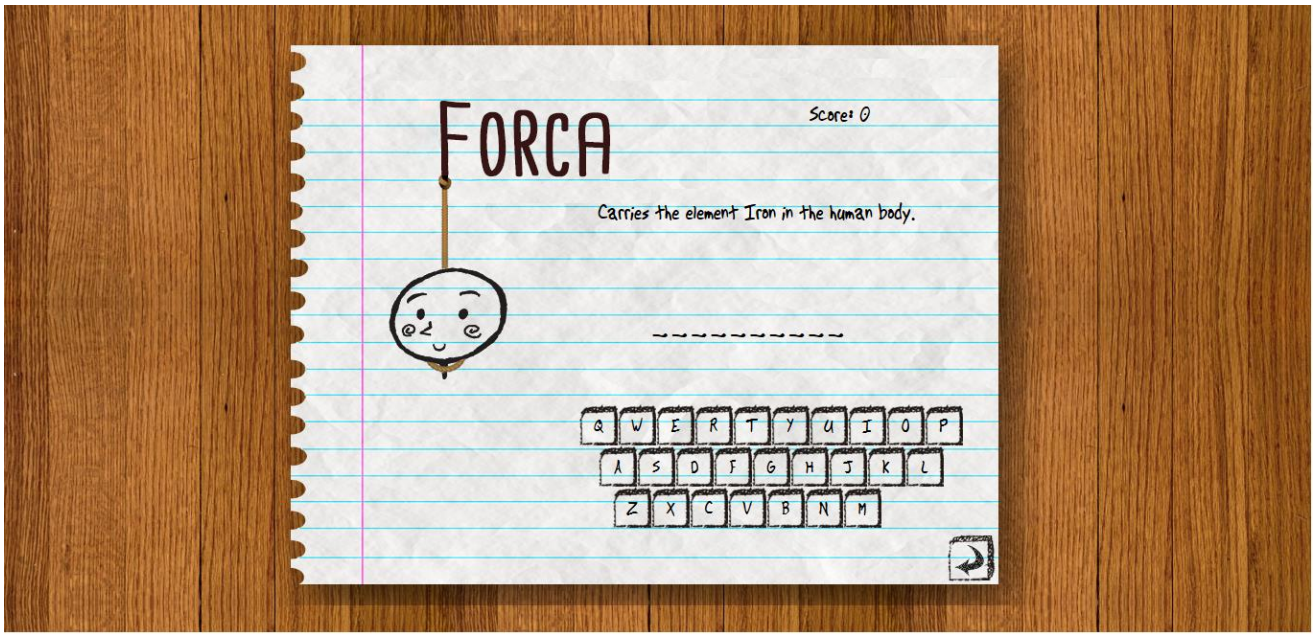

Figura 16 – Jogo Forca iniciado.

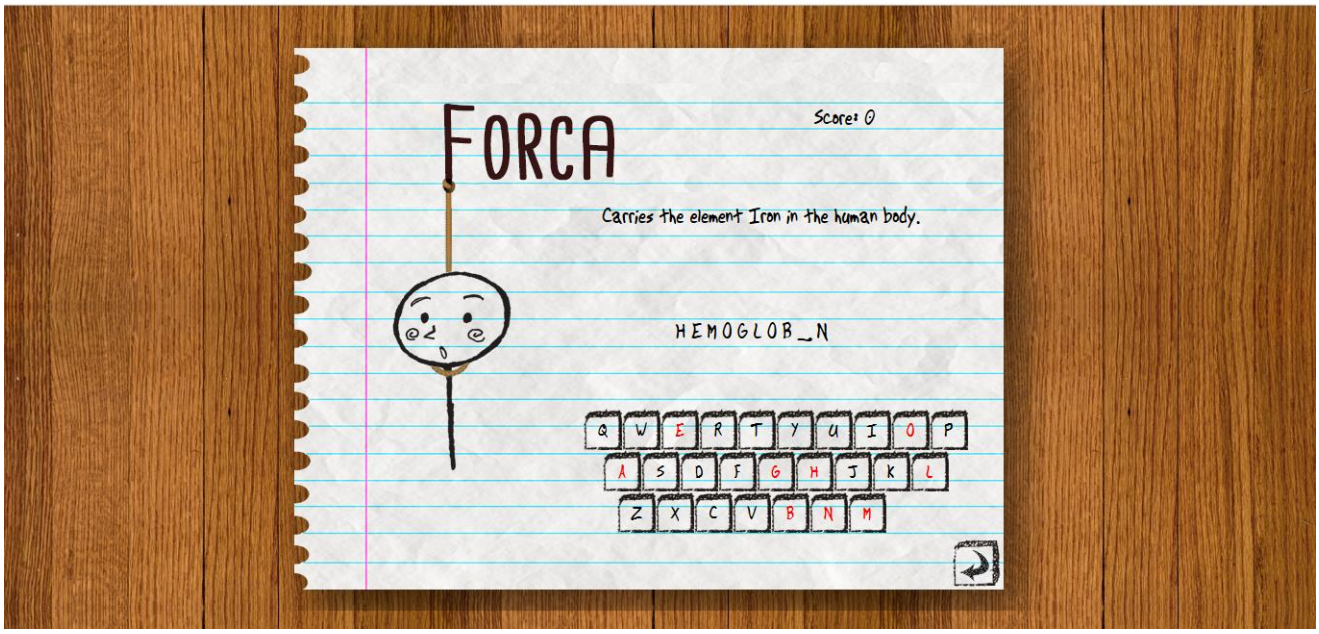

Figura 17 – Jogo Forca depois de selecionada a décima letra. A letra A não pertence a resposta, então agora há somente 4 chances para cometer erros.

4.4 Considerações Finais

Os estudos de caso para validarem este trabalho foram feitos com a aplicação de Gerenciamento de Contatos e com a refatoração do jogo educacional Forca para adicionar as funcionalidades. Ambos envolvendo sincronização de dados e funcionamento *offline e* funcionaram corretamente quando conectados e não conectados à Internet. O código fonte da API, API *Gateway*, micro serviços e aplicações dos estudos de caso podem ser obtidos a partir do link: “https://github.com/LOA-SEAD/synchronization-api”.

Capítulo 5

CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Neste capítulo serão apresentadas as considerações finais deste trabalho e serão apresentados também possíveis trabalhos futuros relacionados à esse.

5.1 Contribuições e Limitações

Este trabalho contribuiu para apoio ao desenvolvimento de aplicações multiplataforma híbridas com as funcionalidades de sincronização de dados e funcionamento *offline* por meio da criação da API e do mecanismo para sincronização de dados baseado em micro serviços.

Ao utilizar a abordagem proposta neste trabalho, os dados da aplicação não serão perdidos devido a alguma oscilação ou perda de conexão com a Internet. O usuário continuará utilizando a aplicação e, quando a conexão for restabelecida, os dados que ficaram pendentes sem sincronização serão sincronizados.

Neste trabalho foi criada também uma ferramenta para otimizar a criação de micro serviços em Java. A partir da classe em Java contendo apenas os atributos a serem gerenciados (armazenados), a ferramenta gera todos os arquivos que compõem o micro serviço desejado. Foi desenvolvido também um script que, a partir dos arquivos que são criados pela ferramenta, realiza a compilação do micro serviço e a implantação do micro serviço no Apache Tomcat.

Com este trabalho espera-se que as aplicações já existentes possam incorporar as funcionalidades de sincronização de dados e funcionamento *offline*

despendendo pouco esforço. E que as novas aplicações que necessitem de sincronização de dados e funcionamento *offline* já possam ser criadas utilizando a arquitetura e implementação apresentadas neste trabalho.

Uma limitação é que o escopo deste trabalho está focado para aplicações desenvolvidas com HTML e JavaScript. Uma outra limitação é o não tratamento quanto à resolução de conflitos. Todos os dados alterados no dispositivo serão enviados ao servidor.

5.2 Trabalhos Futuros

Um possível trabalho futuro está relacionado a uma das limitações apresentadas na seção anterior e envolveria a verificação da utilização da API em aplicações nativas como, por exemplo, aplicações desenvolvidas em Java para o sistema operacional Android. Seria possível pois, a partir do Java 7, existe uma classe chamada “ScriptEngineManager” que possibilita que um código Java invoque um código JavaScript. Porém, outro mecanismo de persistência teria que ser utilizado, pois o recurso *localStorage* do HTML5 não estaria acessível.

Um outro trabalho futuro seria o levantamento e análise de alternativas para o tratamento de uma outra limitação apresentada na seção anterior, quanto à resolução de conflitos. De forma que alguma possível alternativa poderia ser implementada na arquitetura proposta neste trabalho.

REFERÊNCIAS

ALLEN, S.; GRAUPERA, V.; LUNDRIGAN, L. Desenvolvimento Profissional Multiplataforma para Smartphone, iPhone, Android, Windows Mobile e BlackBerry. 1ª edição. Rio de Janeiro: Alta Books, 2012.

ALTEXSOFT. Using Microservices for Legacy System Modernization. 2017. Disponível em: < https://www.altexsoft.com/blog/engineering/using-microservices-for-legacy-system-modernization/>.

ANTONOV, A. Spring Boot Cookbook: Over 35 recipes to help you build, test, and run Spring applications using Spring Boot. 1ª edição. Birmingham: Packt Publishing, 2015.

BUDIU, R. Mobile: Native Apps, Web Apps, and Hybrid Apps. 2013. Disponível em: <https://www.nngroup.com/articles/mobile-native-apps/>

CHAPPELL, D. A.; JEWELL, T. Java Web Services. 1ª edição. Estados Unidos: O'Reilly Media, 2002.

ELLIOTT, J.; O'BRIEN, T.; FOWLER, R. Harnessing Hibernate: Step-by-step Guide to Java Persistence. 1ª edição. Estados Unidos: O'Reilly Media, 2008.

ERL, T. Service Oriented Architecture: Concepts, Technology, and Design. 1ª edição. Estados Unidos: Prentice Hall, 2005.

FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. Tese (Doutorado em Ciência da Computação e Informação) – Universidade da Califórnia, Irvine. 2000.

FIELDING, R. T. *et al.* Hypertext Transfer Protocol - HTTP/1.1. World Wide Web Consortium and The Internet Society. 1999. Disponível em: <https://www.w3.org/Protocols/rfc2616/rfc2616.html>.

FISCHER, P. Crosswalk Overview. Developer Zone. Intel, 2016. Disponível em: <https://software.intel.com/en-us/xdk/docs/crosswalk-application-runtime-overview>.

FOWLER, M. Microservice Trade-Offs. 2015. Disponível em: <http://martinfowler.com/articles/microservice-trade-offs.html#diversity>.

FOWLER, M.; LEWIS, J. Microservices. 2014. Disponível em: <http://martinfowler.com/articles/microservices.html>.

GAMMA, E. *et al*. Design Patterns: Elements of Reusable Object-Oriented Software. 1ª edição. Estados Unidos: Addison-Wesley, 1994.

GUEDES, J. G. F.; JUNIOR, G. S. A.; OLIVEIRA, V. J. G. L. OffDroid: A Framework for Offline Working in Android Applications. In: Webmedia '16 Proceedings of the 22nd Brazilian Symposium on Multimedia and the Web. 2016. p. 95 – 98. ISBN: 978-1-4503-4512-5.

IJTIHADIE, R. M. *et al.* Offline web application and quiz synchronization for e-learning activity for mobile browser. In: TENCON 2010 - 2010 IEEE Region 10 Conference. [S.l.], 2010. p. 2402 – 2405. ISBN: 978-1-4244-6889-8.

KAZI, R.; DETERS, R. RESTful dissemination of healthcare data in mobile digital ecosystem. 2013 7th IEEE International Conference on Digital Ecosystems and Technologies (DEST). [S.l.], 2013. p. 78-83. ISSN: 2150-4938.

LOMOTEY, R. K.; DETERS, R. Consuming Web Services on Mobile Devices for Improved mHealth. In: Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems. Nova York, Estados Unidos: ACM, 2013. (SIGSPATIAL'13), p. 420-423. ISBN: 978-1-4503-2521-9.

LOMOTEY, R. K.; DETERS, R. Mobile-Based Medical Data Accessibility in mHealthMobile. In: Cloud Computing, Services, and Engineering (MobileCloud), 2014 2nd IEEE International Conference on. [S.l.], 2014. p. 91-100.

LOPES, S. A Web Mobile: Programe para um mundo de muitos dispositivos. 1ª edição. São Paulo: Casa do Código, 2013.

NEWMAN, S. Building Microservices: Designing Fine-Grained Systems. 1ª edição. Sebastopol: O'Reilly Media, 2015.

OEHLMAN, D.; BLANC, S. Pro Android Web Apps: Develop for Android using HTML5, CSS3 & JavaScript. The "Build Once" Approach for Mobile App Development. Apress, 2011.

PATEL, S. K. Quick Desktop Application Development Using Electron: Develop Desktop Application Using HTML, CSS e JavaScript. Createspace, 2015.

PETERS, M. *et al*. A client centric replication model for mobile environments based on restful resources. In: Proceedings of the Workshop on Posters and Demos Track. Nova York, Estados Unidos: ACM, 2011. (PDT '11), n. 22. ISBN: 978-1-4503-1073-4.

RICCI, A.; DENTI, E.; PIUNTI, M. A platform for developing SOA/WS applications as open and heterogeneous multi-agent systems. In: Multiagent and Grid Systems. Amsterdã, Holanda: IOS Press, 2010. v.6, n.2, p. 105-132. ISSN: 1574-1702.

RICHARDSON, C. Building Microservices: Using an API Gateway. 2015. Disponível em: <https://www.nginx.com/blog/building-microservices-using-an-api-gateway/>.

RICHARDSON, C. Microservices: Decomposing Applications for Deployability and Scalability. 2014. Disponível em: <http://www.infoq.com/articles/microservices-intro>.

RICHARDSON, C. Pattern: Microservices Architecture. 2014-2. Disponível em: <http://microservices.io/patterns/microservices.html>.

SAUDATE, A. REST: Construa API's inteligentes de maneira simples. São Paulo: Casa do Código, 2014.

SILVA, M. S. jQuery - A Biblioteca do Programador JavaScript: Aprenda a criar efeitos de alto impacto em seu site com a biblioteca JavaScript mais utilizada pelos desenvolvedores web. 3ª edição. São Paulo: Novatec, 2013.

SMITH, S. Overview of ASP.NET Core MVC. 2016. Disponível em: <https://docs.microsoft.com/en-us/aspnet/core/mvc/overview>.

SPENCER, R. Intro to the Crosswalk Project. Intel, 2015. Disponível em: <https://software.intel.com/en-us/videos/intro-to-the-crosswalk-project>.

THE OPEN GROUP. SOA Source Book. 1ª edição. Zaltbommel: Van Haren Publish, 2009.

WONG, C. HTTP Pocket Reference: Hypertext Transfer Protocol. 1ª edição Sebastopol: O'Reilly Media, 2000.

WOODS, D. Building Microservices with Spring Boot. 2015. Disponível em: <http://www.infoq.com/articles/boot-microservices>.

YORK, R. Web Development with jQuery. 1ª edição. Indianapolis: John Wiley & Sons, 2015.

Apêndice A

CÓDIGO DA IMPLEMENTAÇÃO DA API

```
var verify = true;

function _get(url) {
    var d = $.Deferred();
    if (doesConnectionExist(url)) {
        verifyLocalStorage();
        $.getJSON(url, function (data) {
            saveLocal('get', url, data);
            d.resolve(data);
        });
    }
    else
    {
        //comparar a url salva em localStorage com a solicitada
        alert('No connection!');
        for (var i = 0, len = localStorage.length; i < len; ++i) {
            var key = localStorage.key(i);
            if (key.split('_')[0] == 'get' && key.split('_')[2] == url) {
                var retorno = localStorage.getItem(key);
                d.resolve(JSON.parse(retorno));
                break;
```

```
            }
        }
    }
    return d.promise();
}

function _getid(url) {
    if (doesConnectionExist(url)) {
        verifyLocalStorage();
        $.getJSON(url, function (data) {
            saveLocal('get', url, data);
        });
    }
    else {
        //comparar a url salva em localStorage com a solicitada
        for (var i = 0, len = localStorage.length; i < len; ++i) {
            var key = localStorage.key(i);
            if (key.split('_')[0] == 'get' && key.split('_')[2] == url)
                return JSON.parse(localStorage.getItem(key));
        }
    }
}

function _post(url, json) {
    var data;
    if (doesConnectionExist(url)) {
        verifyLocalStorage();
        $.ajax({
            type: 'POST',
            url: url,
            contentType: 'application/json; charset=utf-8',
            data: JSON.stringify(json),
            dataType: 'json',
```

```
            cache: false,
            async: false,
            success: function (retorno) {
                console.log(retorno);
                alert('post - deu certo');
                data = retorno;
            },
            error: function (xhr) {
                console.log(xhr);
                alert('post - n deu certo');
                alert(xhr);
                dados = JSON.stringify(json);
                alert('post dados:' + dados);
                alert(dados.nome);
            }
        });
    }
    else
        saveLocal('post', url, json);
    return data;
}

function _put(url, json) {
    if (doesConnectionExist(url)) {
        verifyLocalStorage();
        $.ajax({
            type: 'PUT',
            url: url,
            contentType: 'application/json; charset=utf-8',
            data: JSON.stringify(json),
            dataType: 'json',
            cache: false,
            async: false,
```

```
            success: function (data) {
                console.log(data);
            },
            error: function (xhr) {
                console.log(xhr);
            }
        });
    }
    else {
        saveLocal('put', url, json);
    }
}

function _delete(url) {
    if (doesConnectionExist(url)) {
        verifyLocalStorage();
        $.ajax({
            type: 'DELETE',
            url: url,
            async: false,
            success: function (data) {
                console.log(data);
                alert('delete - deu certo')
            },
            error: function (xhr) {
                console.log(xhr);
                return xhr;
            }
        });
    }
    else
        saveLocal('delete', url, null);
}
```

```
//----------------------

//web storage

window.onload = function () {
    if (typeof (Storage) != "undefined") {
        verifyLocalStorage();
    }
    else {
        alert("Não há suporte para Web Storage");
    }
};

function verifyLocalStorage()
{
    if (verify) {
        verify = false;
        if (localStorage.length > 0) {
            var chave = localStorage.getItem('chave');
            if (chave != null) {
                chave = JSON.parse(localStorage.getItem('chave'));
                var chave_aux = chave.slice();
                for(var i=0, len = chave.length; i<len; i++)
                {
                    alert('i=' + i);
                    var url = chave[i].replace(chave[i].split('_')[0] + '_', '');
                    url = url.replace(chave[i].split('_')[1] + '_', '');
                    if(doesConnectionExist(url))
                    {
                        alert('i=' + i + chave[i].split('_')[0]);
                        if (chave[i].split('_')[0] == 'post') {
```

```
                    if(localStorage.getItem(chave[i]) != null)
                        _post(url, JSON.parse(localStorage.getItem(chave[i])));
                }
                else if (chave[i].split('_')[0] == 'put') {
                    if (localStorage.getItem(chave[i]) != null)
                        _put(url, JSON.parse(localStorage.getItem(chave[i])));
                }
                else if (chave[i].split('_')[0] == 'delete') {
                    _delete(url);
                }

                alert('chave[i] = ' + chave[i]);
                localStorage.removeItem(chave[i]);
                //remover
                chave_aux.splice(i, 1);
            }
        }
        chave = chave_aux.splice();
        localStorage.removeItem('chave');
        localStorage.setItem('chave', JSON.stringify(chave));
      }
    }
    else
      alert('localStorage.length == 0');
    verify = true;
  }
}

function doesConnectionExist(url) {
  var xhr = new XMLHttpRequest();
  var randomNum = Math.round(Math.random() * 10000);

  xhr.open('GET', url + "?rand=" + randomNum, false);
```

```
    try {
        xhr.send();

        if (xhr.status >= 200 && xhr.status < 304) {
            return true;
        } else {
            return false;
        }
    } catch (e) {
        return false;
    }
}


function removechave(key)
{
    var chave = localStorage.getItem('chave');
    if (chave != null) {
        chave = JSON.parse(localStorage.getItem('chave'));
        if (chave.indexOf(key) > -1) {
            chave.splice(chave.indexOf(key), 1);
        }
        localStorage.setItem('chave', JSON.stringify(chave));
    }
}

function saveLocal(verb, url, json)
{
    if (verb == 'get' || verb == 'put') {
        //comparar a url salva em localStorage com a solicitada
        var j = 0;
        for (var i = 0, len = localStorage.length; i < len; ++i) {
```

```
            var key = localStorage.key(j);
            if (verb == 'get' && key.split('_')[0] == 'get' && key.split('_')[2] == url)
                localStorage.removeItem(key);
            else if (verb == 'put' && key.split('_')[0] == 'put' && key.split('_')[2] == url) {
                localStorage.removeItem(key);
                removechave(key);
            }
            else
                j++;
        }
    }
    var data = new Date();
    var month = data.getMonth() + 1;
    var now = data.getDate() + '' + month + '' + data.getFullYear() + '|' +
data.getHours() + ':' + data.getMinutes() + ':' + data.getSeconds() + ':' +
data.getMilliseconds();
    localStorage.setItem(verb + '_' + now + '_' + url, JSON.stringify(json));
    if(verb == 'post' || verb == 'put' || verb == 'delete')
    {
        var chave = localStorage.getItem('chave');
        if (chave != null) {
            chave = JSON.parse(localStorage.getItem('chave'));
            chave.push(verb + '_' + now + '_' + url);
            localStorage.setItem('chave', JSON.stringify(chave));
        }
        else {
            chave = [];
            chave.push(verb + '_' + now + '_' + url);
            localStorage.setItem('chave', JSON.stringify(chave));
        }
    }
}
```