Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

FILIPE BAPTISTELLA MAIA

ABORDAGEM ARQUITETURAL ORIENTADA A RECURSOS PARA DESENVOLVIMENTO WEB E MOBILE

Assis/SP
2016

Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

FILIPE BAPTISTELLA MAIA

ABORDAGEM ARQUITETURAL ORIENTADA A RECURSOS PARA DESENVOLVIMENTO WEB E MOBILE

Projeto de pesquisa apresentado ao curso de Ciência da Computação do Instituto Municipal de Ensino Superior de Assis – IMESA e a Fundação Educacional do Município de Assis – FEMA, como requisito parcial à obtenção do Certificado de Conclusão.

Orientando: Filipe Baptistella Maia
Orientador: Prof. MSc. Guilherme de Cleva Farto

Assis/SP
2016

FICHA CATALOGRÁFICA

MAIA, Filipe Baptistella.

Abordagem arquitetural orientada a recursos para desenvolvimento Web e Mobile / Filipe Baptistella Maia. Fundação Educacional do Município de Assis –FEMA – Assis, 2016.

74p.

1. Arquitetura orientada a recursos. 2. REST. 3. Java EE. 4. Google Android.

CDD: 001.6
Biblioteca da FEMA

ABORDAGEM ARQUITETURAL ORIENTADA A RECURSOS PARA DESENVOLVIMENTO WEB E MOBILE

FILIPE BAPTISTELLA MAIA

Trabalho de Conclusão de Curso apresentado ao Instituto Municipal de Ensino Superior de Assis, como requisito do Curso de Graduação, avaliado pela seguinte comissão examinadora:

Orientador: _____
Prof. MSc. Guilherme de Cleva Farto

Examinador: _____
Prof. Dr. Luiz Carlos Begosso

Assis/SP
2016

DEDICATÓRIA

Dedico este trabalho a Deus, a toda minha família, namorada, amigos e todos aqueles que me apoiaram para a conclusão deste trabalho.

AGRADECIMENTOS

Primeiramente, agradeço a **Deus** pela minha vida, por tudo o que fez e o que tem feito, pela força, conquistas e o amor a minha vida.

Agradeço aos meus **pais**, por todos os esforços, dedicações e confiança que depositaram em minha pessoa para me tornar o que sou. Aos meus **irmãos** e toda a **minha família**, que sempre me apoiam e me proporcionam momentos agradáveis aos seus lados.

A minha namorada, **Rafaela Martins**, por estar sempre ao meu lado com toda sua atenção, carinho e amor, ajudando-me com todo seu apoio e motivação para dar continuidade nessa caminhada.

A todos os meus **amigos**, que nesses anos de faculdade proporcionaram momentos alegres e saudáveis. Também, por concederem e partilharem seus conhecimentos para auxiliar-me.

Ao meu orientador, Professor MSc. **Guilherme de Cleva Farto**, pela sua dedicação, profissionalismo e paciência para me orientar e compartilhar de seu conhecimento.

“Esta é a confiança que temos nele, que se lhe pedirmos alguma coisa, segundo a sua vontade, Ele nos ouve.” – I João 5:14.

RESUMO

Com a evolução da Web, inúmeras vantagens surgiram para o mundo corporativo que a utilizam, como também para os seus desenvolvedores, a partir de novas linguagens de computação que fornecem recursos para o desenvolvimento de aplicações mais robustas e interconectadas. Além da evolução da Web, a ascensão da tecnologia móvel, como as televisões *smart*, *smartphones*, *tablets* e demais dispositivos que fazem parte dessa família, trouxeram novas funcionalidades às aplicações Web, as quais têm sido utilizadas como mecanismos para a persistência e manipulação de dados por meio de dispositivos móveis. Para essa troca de dados ocorrer aplica-se o conceito de interoperabilidade entre as aplicações, alcançada por Web Services REST e pelos fundamentos da arquitetura orientada a recursos. O presente trabalho objetiva, inicialmente, a investigação e o relato de um estudo alicerçado na adoção de materiais confiáveis, englobando tecnologias que se relacionam à arquitetura orientada a recursos, bem como à plataforma JEE e o desenvolvimento de serviços REST para a integração com aplicações Android. Também apresenta-se o desenvolvimento de uma aplicação protótipo que incorpora as tecnologias e recursos mencionados. Como resultados, espera-se contribuir com a divulgação dos fundamentos explorados, assim como pela definição arquitetural proposta para integração de aplicações Web e Mobile por meio de serviços REST que pode ser estendida e adaptada a diferentes contextos e necessidades.

Palavras-chave: Arquitetura orientada a recursos, REST, Java EE, Google Android.

ABSTRACT

With the evolution of the Web, many advantages appeared to the corporate world that use, but also for its developers, from new languages that provide computing resources to the development of more robust and interconnected applications. In addition to web development, the rise of mobile technology, such as smart TVs, smartphones, tablets and other devices that are part of that family, they brought new functionality to Web applications, which have been used as mechanisms for persistence and data handling through mobile devices. For this data exchange occurs applies the concept of interoperability between applications, achieved by REST Web Services and the resource-oriented architecture features.This work aims initially to research and report of a grounded study the adoption of reliable materials, including technologies that relate to resource-oriented architecture and the JEE platform and the development of REST services for integration with Android applications. It also aims to present the development of a prototype implementation incorporating the features mentioned technologies. As a result, we expect to contribute to the dissemination of the exploited foundations, as well as defining architectural proposal for the integration of Web applications and Mobile through REST services that can be extended and adapted to different contexts and needs.

Keywords: Resource-oriented architecture, REST, Java EE, Google Android.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1. INTRODUÇÃO

Com a difusão da Internet nas últimas décadas, a *World Wide Web* emergiu, desencadeando várias soluções para desenvolvedores e profissionais de tecnologia. Muitas empresas têm investido em aplicações Web e, com isso, alcançado melhores rendimentos, publicidade e atendimento aos clientes. Segundo Jacyntho (2008), as aplicações Web rapidamente evoluíram tornando-se sistemas de informação complexos, carregados de dados e transações, bem como direcionados a processos de negócio organizacionais.

A Web não foi a única a se propagar, pois os dispositivos móveis também obtiveram inúmeros avanços. Dispositivos menores, mais resistentes, com novas tecnologias integradas, evoluções em seus sistemas operacionais e dentre outras progressões. E, como todo produto, quando a demanda aumenta, significa que existe consumo para tal, assim, não é diferente para esses dispositivos. Só para os celulares, mais de três bilhões de pessoas possuem um. (LECHETA, 2013).

Não só os dispositivos móveis estão com significativo número de usuários, mas também seus aplicativos. Alves (2013) descreve que o aumento do desenvolvimento de aplicações móveis se deve à inclusão de dispositivos sem fios com distintas características, como *smartphones* e *tablets*.

Visando os dois contextos, Web e Mobile, os quais este projeto investiga, é viável explicitar sobre a interoperabilidade das aplicações, possibilitando e ampliando a possibilidade de integração entre elas por meio de Serviços Web. Tal tecnologia tem como objetivo permitir que diferentes aplicações de diversas plataformas comuniquem-se por meio da Internet. (ALVES, 2012).

Há mais de uma arquitetura de serviços Web que por nomenclatura no domínio tecnológico é mais conhecida como *Web Services*. A arquitetura orientada a serviços é uma delas, conceituada como *Service Oriented Architecture* (SOA). Ela possui certas desvantagens, dentre elas a complexidade, pois se torna necessário utilizar de inúmeros artefatos de desenvolvimento como XML, UDDI e SOAP para ser gerenciada. Gouveia e Gouveia (2016) relatam um conjunto de desafios para a arquitetura SOA:

- **Segurança**: Os serviços Web são expostos a vulnerabilidades e ameaças externas, devido a sua interoperabilidade entre os sistemas.
- **Rastreabilidade**: Capacidade de orquestrar processos de negócio, aumenta os requisitos de monitorização e rastreabilidade.
- **Disponibilidade da informação em tempo real:** Por ser uma arquitetura fracamente acoplada, a SOA não se adequa bem a sistemas de tempo real.
- **Competências e experiência na implementação SOA**: Padrões para a implementação da SOA, ainda estão em um grau elevado de imaturidade.
- **Custo**: O custo para implementação de uma SOA requer um alto custo inicial, levando-se em conta a reengenharia de arquiteturas existentes que consomem recursos humanos e financeiros.

Outra arquitetura é a orientada a recursos, originalmente definida como *Resource Oriented Architecture* (ROA). Essa arquitetura originou-se a partir dos obstáculos identificados ao projetar aplicações Web com base em REST. ROA é amparada pelos princípios de REST, porém também está relacionada às tecnologias Web, como o protocolo HTTP, URI e formatos de dados como XML, XHTML e JSON (POLÔNIA, 2011).

Devido às desvantagens mencionadas para a arquitetura SOA, este trabalho pretende abordar a arquitetura orientada a recursos, a qual fornece facilitadores para o desenvolvimento de serviços Web, pois trabalha com protocolos JSON e também XML. É importante ressaltar que há uma busca cada vez maior por essa tecnologia, tendo como exemplo o trabalho de Paiva (2013), que também apresenta a ROA e explana seus conceitos e suas principais características.

Pelo grande avanço dessas ferramentas, o presente projeto teve como proposta pesquisar, fundamentar e apresentar o conteúdo das tecnologias e fundamentos de ROA, *Web* e *Mobile*. Por final, com o objetivo de investigar e relatar, pretende-se desenvolver duas aplicações, uma Web e outra para dispositivos móveis.

1.1. OBJETIVOS

1.1.1. OBJETIVOS GERAIS

O presente trabalho tem como principal objetivo investigar e relatar os conceitos e fundamentos da arquitetura orientada a recursos, expondo suas características, propriedades, vantagens e benefícios em relação à arquitetura orientada a serviços.

Além disso, também serão descritos os conceitos acerca das plataformas Java EE e Google Android e, dessa forma, duas aplicações serão desenvolvidas para os contextos Web e *Mobile* para validar as tecnologias investigadas.

1.1.2. OBJETIVOS ESPECÍFICOS

Para alcançar os objetivos gerais deste trabalho, foram estabelecidas as seguintes atividades:

- Explorar e pesquisar a arquitetura orientada a recursos;
- Explorar e pesquisar a plataforma Java EE, apresentando seu histórico, evolução, APIs e demais características;
- Explorar e pesquisar a plataforma Google Android, relatando o ambiente de desenvolvimento, suas camadas e demais características;
- Projetar e implementar uma solução arquitetural baseada em ROA para *Web Services* e Android.

1.2. JUSTIFICATIVAS

As aplicações, como já mencionado, tanto para a plataforma Web, quanto para as de dispositivos móveis estão em amplo crescimento. Existem muitos aplicativos que fazem o uso de Web Service, *e-commerces, por exemplo*, os quais os mesmos produtos que estão no módulo Web são equivalentes aos da aplicação móvel. Isso, devido aos serviços web se utilizarem de métodos para ter acesso ao mesmo banco ou servidor, sendo simultâneo a escrita e leitura de informações.

De acordo com Lecheta (2013), diversas empresas têm desenvolvido aplicações móveis com o objetivo de acelerar os negócios e integrar as aplicações móveis aos sistemas de *back-end*. Desta forma, as aplicações móveis podem estar literalmente conectadas, sincronizando informações diretamente de um servidor da empresa.

Em virtude da interoperabilidade das aplicações, é importante destacar que há a necessidade de plataformas e arquitetura agnóstica para integração delas, ou seja, que independentemente do ambiente, possam ser executadas em qualquer sistema operacional e se comuniquem indiretamente por meio de uma aplicação ou diretamente a partir de uma solicitação com o caminho do serviço. A interoperabilidade ocorre por meio dos serviços Web. Segundo Lecheta (2015), os *Web Services* são usados como meio de integração e comunicação de sistemas, de tal modo que um sistema possa realizar uma chamada para um serviço de outro sistema.

A plataforma Java EE pode ser executada em qualquer navegador que tenha suporte a ela, e esta contém inúmeros recursos, sendo um deles a integração de *Web Services*. Algumas características da arquitetura orientada a recursos são: integração com a plataforma Java EE, interface uniforme, segurança e criptografia de dados, manipulação dos dados gerados por meio de mensagens XML e JSON, também acesso aos recursos por meio de URIs. De acordo com Richardson e Ruby (2007), a ROA é uma arquitetura que segue os princípios do estilo arquitetural REST e é baseada em tecnologias da Web como HTTP e URI.

1.3. MOTIVAÇÃO

Trabalhos, artigos e pesquisas acadêmicas científicas abordam a arquitetura orientada a recursos, porém há poucos materiais que descrevem em detalhes e apresentam um modelo arquitetural compreensível e aplicado a protótipos de validação. Visando preencher essa lacuna, este trabalho propõe explorar e aplicar os conceitos da arquitetura ROA, contribuindo com pesquisas futuras, bem como a possibilidade de evoluir a definição arquitetural experimentada nesta pesquisa.

Desta forma, este projeto não irá abordar a arquitetura somente no ponto de vista de pesquisa, mas também considerará a implementação e o relato de mecanismos de integração de Web com os dispositivos móveis.

1.4. PERSPECTIVAS DE CONTRIBUIÇÃO

Na finalização deste trabalho, pretende-se disponibilizar o material relatado, contendo a parte teórica e prática da proposta de implementação das aplicações Web e Mobile. Planeja-se, também, disponibilizar um modelo arquitetural que pode ser estendido e/ou adaptado para novos cenários, contribuindo a partir de compartilhamento de experiências, com o desenvolvimento e implementação de novas soluções.

1.5. METODOLOGIA DE PESQUISA

A metodologia de pesquisa deste trabalho está dividida em duas partes:

A primeira é a pesquisa, a qual está alicerçada pela adoção de materiais confiáveis, livros, artigos científicos, artigos técnicos, monografias, dissertações e teses para fundamentar conceitos sobre a arquitetura ROA, Java EE, e a plataforma Google Android. Ademais, será feita a investigação e utilização de ferramentas e abordagens existentes, concluindo-se a primeira fase com o encerramento das pesquisas.

Em seguida, inicia-se a segunda parte, que consiste no desenvolvimento da proposta de trabalho. Serão apresentados os desenvolvimentos para a validação da abordagem proposta. Dessa forma, a segunda etapa da metodologia é destacada pelos conceitos aplicados às aplicações desenvolvidas.

1.6. RECURSOS NECESSÁRIOS

No desenvolvimento deste projeto foram utilizados os seguintes recursos de software e hardware:

- Hardware:
  - Notebook Asus X44-C
    - Processador Intel Core i3 2.20 GHz.
    - Disco Rígido 5400 RPM de 500 GB.

    - Memória de 1333 MHz DDR3 4GB.

- Software
    - ***Eclipse IDE*** – Ambiente de desenvolvimento para a tecnologia Java e *Web Services*. Será utilizada a versão *Luna*.
    - ***Android SDK*** – Ferramentas que possibilitam o desenvolvimento e a execução de aplicações móveis com a tecnologia *Google Android*.
    - ***Android Studio –*** Ambiente de desenvolvimento oficial de aplicativos Android, este baseado na IDE IntelliJ.

1.7. ESTRUTURA DO TRABALHO

O presente projeto será apresentado a partir da seguinte estrutura:

O **Capítulo 1** contextualiza o estudo proposto, apresenta os objetivos, as justificativas, motivações, perspectivas de contribuição e a metodologia a ser utilizada neste projeto.

Em seguida, no **Capítulo 2**, é abordado e explanado a arquitetura orientada a recursos, por meio de conceitos, de suas propriedades, e de um exemplo motivacional da arquitetura.

No **Capítulo 3**, a plataforma Java *Enterprise Edition*, Java EE, é abordada, bem como a o seu histórico e evolução são apresentados. Nesse capítulo, também, o estilo arquitetural REST é retratado, mostrando suas APIs, bibliotecas e os formatos de troca de mensagens.

Posteriormente, o **Capítulo 4** aborda a plataforma Google Android, de tal modo que apresenta a IDE Android Studio e, também, o conjunto de ferramentas que a compõe, e APIs e bibliotecas para utilização de serviços web REST.

Dentro do **Capítulo 5** é apresentada a proposta de trabalho, como são desenvolvidas as aplicações Web e Mobile. Logo após, **no Capítulo 6** é mostrado o estudo de caso como validação da proposta apresentada.

Na conclusão, **Capítulo 7**, é exibido os prós, contras, vantagens, desvantagens e algumas das experiências e lições aprendidas com o estudo de caso. Seguidamente são apresentadas as **referências** utilizadas no projeto.

2. RESOURCE-ORIENTED ARCHITECTURE (ROA)

A arquitetura orientada a recursos foi apresentada por Richardson e Ruby (2007) em seu livro. Essa arquitetura foi baseada nos princípios do REST, porém, como vantagem, a mesma utiliza de recursos da Web, como o protocolo HTTP e as URIs.

Segundo os autores da ROA, Richardson e Ruby (2007), REST não é uma arquitetura, mas sim um conjunto de especificações para se desenvolver uma arquitetura. Uma arquitetura pode satisfazer a este conjunto de especificações melhor do que outra, porém, mesmo assim, não existe uma arquitetura REST. Com esse conjunto, desenvolvedores criaram suas próprias arquiteturas e serviços, conforme sua compreensão sobre os termos de REST, acarretando uma gama de *Web Services* híbridos REST-RPC. Como forma de resolução a este problema é apresentado o ROA, um conjunto de regras concretas para desenvolver *Web Services* verdadeiramente REST.

De acordo com Polônia (2011), a arquitetura nasceu das dificuldades enfrentadas por desenvolvedores ao projetar Web com base em REST. Essas dificuldades estão relacionadas à abstração dos conceitos de REST. Em meio às dificuldades está o fato de que REST não está ligado com a Web e suas tecnologias, devido ele ser um estilo arquitetural para sistemas de hipermídia distribuídos. Com a necessidade de uma abordagem mais concreta e pragmática para os problemas surgiu a ROA. Esta arquitetura segue os princípios de REST, mas está vinculada com as tecnologias da Web, como o protocolo HTTP, URI e formatos de dados como o XML, XHTML e JSON.

A arquitetura orientada a recursos contém alguns conceitos e quatro propriedades, os quais serão abordados no decorrer do capítulo. Os conceitos a serem apresentados no presente trabalho são os de recursos, URI e representações. As propriedades dessa arquitetura que serão apresentadas são: a endereçabilidade, a falta de estado, a conectividade e a interface uniforme.

2.1. CONCEITOS DE ROA

Alguns conceitos sobre a arquitetura orientada a recursos são de grande valia abordar, já que elas a estruturam e apresentam-na como uma arquitetura a seguir para o desenvolvimento de serviços REST. O primeiro conceito é o de recurso.

Um recurso pode ser armazenado em um computador e ser representado por um fluxo de bits, como um documento, uma linha em um banco de dados ou o resultado da execução de um algoritmo. Um recurso também pode ser um objeto físico, como uma maça, ou um conceito abstrato, como coragem. Como exemplos de recursos plausíveis, versão de um software, uma relação entre duas pessoas, os próximos cinco números primos depois de um determinado número. (RICHARDSON, RUBY; 2007).

Outro conceito importante sobre a arquitetura ROA, é a *Uniform Resource Identifier* (URI), Identificador Uniforme de Recursos. De forma específica, a URI, é o caminho ou o endereço para tal recurso.

Conforme Saudate (2013), uma URI, como o próprio nome descreve, pode ser usada para identificar qualquer coisa, como dar um caminho para um determinado conteúdo, como também, dar nome a este. Agora, uma URL pode ser utilizada apenas para fornecer caminhos, sendo que uma URL é uma forma de uma URI.

Abaixo, alguns exemplos de URI para uma assimilação melhor desse conceito:

- “/biblioteca/livros” retorna uma lista com todos os livros cadastrados na biblioteca
- “/biblioteca/livros/ano” retorna uma lista de livros da biblioteca de acordo com o ano informado.
- “/biblioteca/livros/categoria” retorna uma lista de livros cadastrados na biblioteca de acordo com a categoria informada.
- “/biblioteca/livros/titulo” retorna uma lista de livros cadastrados na biblioteca de acordo com o titulo informado.

Como forma de uma boa prática e desenvolvimento, Richardson e Ruby (2007), propõem que um recurso e sua URI devam ter uma correspondência clara. Também, que as URIs devem ter uma estrutura.

É significativo salientar que uma URI não pode retornar dois recursos, e que dois recursos não podem ser iguais, mas, também, que um recurso pode ter uma ou várias URIs utilizando-o. Segundo Richardson e Ruby (2007), dois recursos não podem ser iguais, se fossem, seria apenas um recurso. Entretanto, em algum momento dois recursos diferentes podem apontar para os mesmos dados.

Mais um conceito sobre essa arquitetura são suas representações, a forma de representar os recursos, em algum tipo de linguagem de marcação de dados. Um recurso é uma fonte de representações e uma representação são apenas alguns dados sobre o

estado atual do mesmo. (RICHARDSON; RUBY, 2007). Conforme Polônia (2011), o principal critério na escolha de representações é o tipo que os clientes pretendem consumir. A arquitetura orientada a recursos sugere tipos de mídias comum da web como XML, XHTML e JSON.

2.2. SEGURANÇA E CRIPTOGRAFIA

Muitos dados trafegam a rede nesse imenso mundo tecnológico, dentre eles, dados confidenciais, dados que tenham algum valor pessoal ou financeiro. A segurança desses dados são assegurados pelos *Web Services* orientados a recursos, porém, de acordo com os critérios e modos implementados de cada aplicação.

As solicitações GET e HEAD, quando utilizadas apropriadamente, são seguras. Essas solicitações, mais as PUT e DELETE, são idempotentes. Segurança e a idempotência permitem a um determinado cliente em uma rede não confiável, realizar requisições seguras. No caso de se realizar uma solicitação GET e a mesma não ter uma resposta, pode-se tentar novamente, pois é seguro. Mesmo que a primeira solicitação tenha passado, a segunda solicitação não terá nenhum efeito adicional. (RICHARDSON; RUBY, 2007).

No contexto de segurança, um modo de potencializar a mesma é por meio de métodos de criptografia de dados, os quais alteram a ordem dos dados para ficar de forma ilegível de acordo com método e sua chave. Somente quem conhece essa chave consegue decriptografar os dados. Com uma melhor criptografia de dados, obtém-se uma melhor segurança.

Na arquitetura ROA, a criptografia se encaixa na autenticação e autorização. Segundo Richardson e Ruby (2007), quando um cliente do *Web Service* realiza uma requisição HTTP, ele pode incluir algumas credencias na parte de autorização do cabeçalho HTTP, usuário e senha. O serviço verifica as credencias e decide se elas identificam corretamente o cliente como um determinado usuário, e se ele tem permissão para o que deseja fazer, caso tudo esteja correto, o servidor executa a solicitação. Existem vários tipos de padrões de autenticação, a HTTP Basic, HTTP Digest e WSSE. Usar HTTPS, em

vez de HTTP impede que os outros computadores interceptem a conversa entre o cliente e o servidor. Isto se torna importante ao usar a autenticação HTTP Basic.

A autenticação básica é um procedimento de desafio e resposta simples. Para responder este desafio o cliente necessita de um usuário e uma senha. Estas informações podem ser arquivadas em um cache. O nome e a senha são misturadas em um campo de texto, uma String, especificamente, e codificadas por meio da codificação de base 64. O servidor decodifica essa String e a compara com a lista de usuários e senhas para confirmar as credencias. Qualquer pessoa da rede pode descobrir e decodificar essa String, pois a autenticação básica transmite nomes e senhas de usuários em forma de texto. Uma solução para este problema é usar o HTTPS. Protocolo que criptografa todas as comunicações feitas entre o cliente e o servidor. (RICHARDSON, RUBY, 2007).

Em uma autenticação HTTP Digest o cliente combina diversas informações em uma String, sendo elas, o método HTTP e o caminho da requisição, quatro pedaços de informação do desafio, o nome do usuário e a senha, o *nonce client-side* e o número sequencial. Embora essa criptografia seja mais complicada, o processo é o mesmo da autenticação básica. Uma diferença entre as duas é que nem mesmo o servidor consegue descobrir a sua senha a partir da digest. Quando o cliente define inicialmente uma senha, o servidor precisa calcular o *hash* e mantê-lo armazenado. O que garante ao servidor o acesso aos dados necessários para calcular o valor final sem armazenar a senha real do usuário. (RICHARDSON; RUBY, 2007).

O WSSE é como o HTTP Digest, pois o cliente passa a sua senha por meio de um algoritmo de *hash* antes de enviá-la pela rede. Esse padrão tem suas vantagens, como exemplo, ele não encaminha a senha diretamente pela rede, como a autenticação básica, e não requer qualquer configuração especial no lado do servidor como a HTTP Digest, mas, existe uma grande desvantagem, o servidor deve armazenar as senhas em texto puro ou não será capaz de validar as respostas, isto é, se algum indivíduo invadir o servidor vai ter todas as senhas para ele. (RICHARDSON; RUBY, 2007).

2.3. PROPRIEDADES

2.3.1. Endereçabilidade

A propriedade de endereçabilidade ocorre quando o usuário, por meio de uma URI, consegue recuperar um recurso por meio de seu endereço, para uma melhor definição.

A aplicação será endereçável se a mesma exibir os recursos. Uma aplicação endereçável apresenta uma URI para toda informação que possa ser disponibilizada, já que os recursos são recuperados por meio de URIs. Na visão do usuário final, o endereçamento é o ponto de vista mais importante para aplicações *web*. (RICHARDSON; RUBY, 2007).

Em questão de exemplo, considere uma pesquisa feita na aplicação Google, que por sinal é um mecanismo de busca endereçável, para pesquisar uma determinada informação, como REST. Uma URI que a representa seria: www.google.com.br/search?q=REST. Para Richardson e Ruby (2007), se o protocolo HTTP e a ferramenta de pesquisa Google não fossem endereçáveis, a URI que apresentam no livro, como o exemplo acima, não poderia ser endereçável, além do mais, para acontecer um retorno à busca, relatam que teriam de seguir uma série de procedimentos para a mesma ser realizada.

Segundo Silvestre e Polônia (2008), essa propriedade possibilita utilizar o cache de requisições feitas, a qual, quando uma mesma requisição é repetida pode-se usar os dados em cache, para não consumir banda. Sem a propriedade de endereçabilidade não teria como saber as requisições que foram realizadas.

Richardson e Ruby (2007) explanam alguns exemplos de aplicações que também são endereçáveis, como o sistema de arquivos em uma máquina. O terminal de comando pode seguir um caminho para determinado arquivo e utilizá-lo da maneira que quiser e também, às células de uma planilha, pode-se ligar o nome de uma determinada célula a uma fórmula e esta fórmula utilizará o valor que estiver nesta célula. As URIs são os caminhos do arquivo e os endereços da célula da Web.

No momento em que o cliente abrir a URI “/biblioteca/livros/categoria?a=Informática”, informando como parâmetro a categoria informática, o seguinte recurso pode ser recuperado:

```
<livros>
    <livro codigo = "3">
        <ano>Ano</ano>
        <titulo>Titulo do Livro</titulo>
```

```
        <autor>Autor do Livro</autor>
        <categoria>Informática</categoria>
    </livro>
</livros>
```

Quando o cliente acessar a URI “/biblioteca/livros/ano?a=2016”, informando como parâmetro o ano 2016, o seguinte recurso pode ser obtido:

```
<livros>
    <livro codigo = "10">
        <ano>2016</ano>
        <titulo>Titulo do Livro</titulo>
        <autor>Autor do Livro</autor>
        <categoria>Categoria do Livro</categoria>
    </livro>
</livros>
```

2.3.2. Stateless e Stateful

Esta propriedade, *stateless*, também conhecida como falta de estado, está presente no protocolo HTTP. Toda requisição HTTP que o cliente faz, é necessário que na mesma requisição estejam os dados para que o servidor possa atendê-la.

De acordo com Richardson e Ruby (2007), a propriedade *stateless* significa que, toda solicitação HTTP acontece em completo isolamento. No momento em que o cliente realiza uma solicitação HTTP, esta contém todas as informações necessárias para o servidor satisfazer-lhe. O servidor não tem as informações das solicitações anteriores, caso as mesmas forem importantes é necessário que o cliente as envie novamente.

Em termos de endereçamento, todo dado interessante que o servidor pode prover, deve ser exibido como um recurso, e com sua própria URI. Conforme o *stateless* os estados do servidor também são recursos, e os mesmos devem ser dados em suas próprias URIs. (RICHARDSON; RUBY; 2007).

A Figura 1 apresenta um diagrama de uma aplicação de pesquisa com a propriedade *stateless*. O cliente pode solicitar ao servidor tanto uma pesquisa com “REST *Web Services*”, como, também, solicitar a décima página da mesma busca, na ordem que melhor desejar. De acordo com Richardson e Ruby (2007), em uma aplicação com falta de estado, sempre que o cliente faz uma requisição a mesma termina onde se iniciou.

Assim, cada requisição é independente das outras. O cliente pode realizar requisições diversas vezes para os recursos, em qualquer ordem.

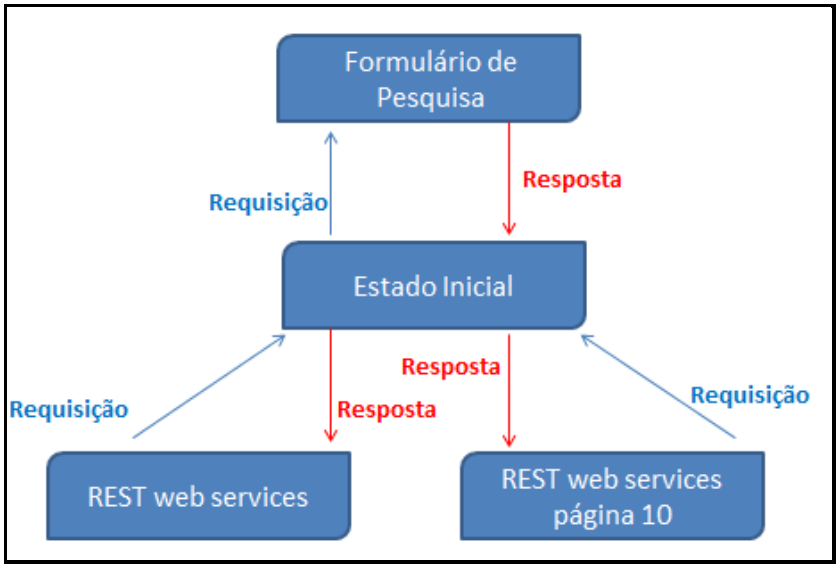

Figura 1 - Exemplo de uma aplicação de busca sem estado.

Um protocolo sem estado, como o HTTP, elimina muitas condições de falha. O servidor não precisa se preocupar com o tempo esgotado do cliente, pois nenhuma interação dura mais que uma solicitação. Também, o mesmo tem o controle onde está o cliente na aplicação, porque é o cliente que faz a requisição já com as informações necessárias. (RICHARDSON; RUBY, 2007).

Em contrapartida a *stateless*, existe a *stateful*, também conhecido como: com estado. Esta, diferentemente da falta de estado, permite que os dados sejam mantidos entre as requisições realizadas para serem usados novamente em outras requisições. A Figura 2 mostra a mesma aplicação de ferramenta de pesquisa da Figura 1, porém, com a propriedade *stateful.*

Para Richardson e Ruby (2007), uma aplicação com estado é mais organizada, caso o HTTP permitisse a interação com estado, as requisições HTTP seriam mais simples. Quando fosse iniciada, pelo cliente, uma seção com o mecanismo de pesquisa, ele poderia ser preenchido automaticamente no formulário de pesquisa. Não teria que ser necessário encaminhar algum dado para a requisição, porque a primeira resposta seria pré-determinada, porém, o mesmo tornando as requisições HTTP mais simples, prejudicaria o protocolo. Seria mais complicado, pois o estado da seção teria de estar em sincronia entre o cliente e o servidor, uma tarefa complexa mesmo em uma rede confiável.

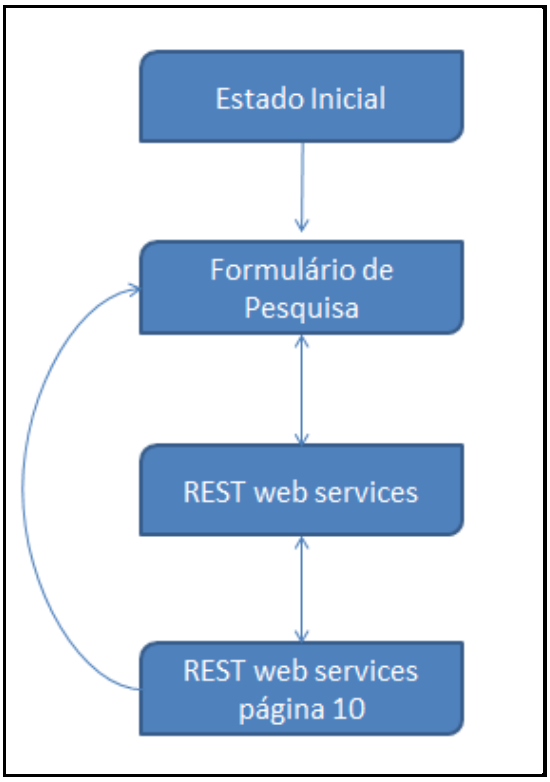

Figura 2 - Exemplo de uma aplicação de busca com estado.

2.3.3. Conectividade

O cliente pode navegar na *web* de um lugar para o outro por meio de endereços entre recursos, esta característica é conhecida como conectividade, pois os recursos podem estar ligados uns aos outros.

De acordo com Richardson e Ruby (2007), algumas representações são programadas para terem seus dados extraídos e serem descartados, porém, em muitos *Web Services* REST, estas estruturas de dados serializadas são uma hipermídia, o qual, documentos embutidos não são só dados, mas links com outros recursos.

Seguindo um exemplo de uma pesquisa, caso o usuário vá para o diretório de documentos do Google, sobre determinado assunto, terá como resultado sua pesquisa e um conjunto de links internos para seguir a outras páginas do diretório. Esses links provem acessos para os outros recursos, seja eles do próprio serviço da Google, como também, outros de algum lugar da Web. (RICHARDSON; RUBY, 2007).

A Figura 3 apresenta um exemplo da afirmação acima, realizada na ferramenta de busca Google, onde no corpo da figura se encontram os resultados da pesquisa, e em seu rodapé, os links que encaminham para outras páginas do diretório.

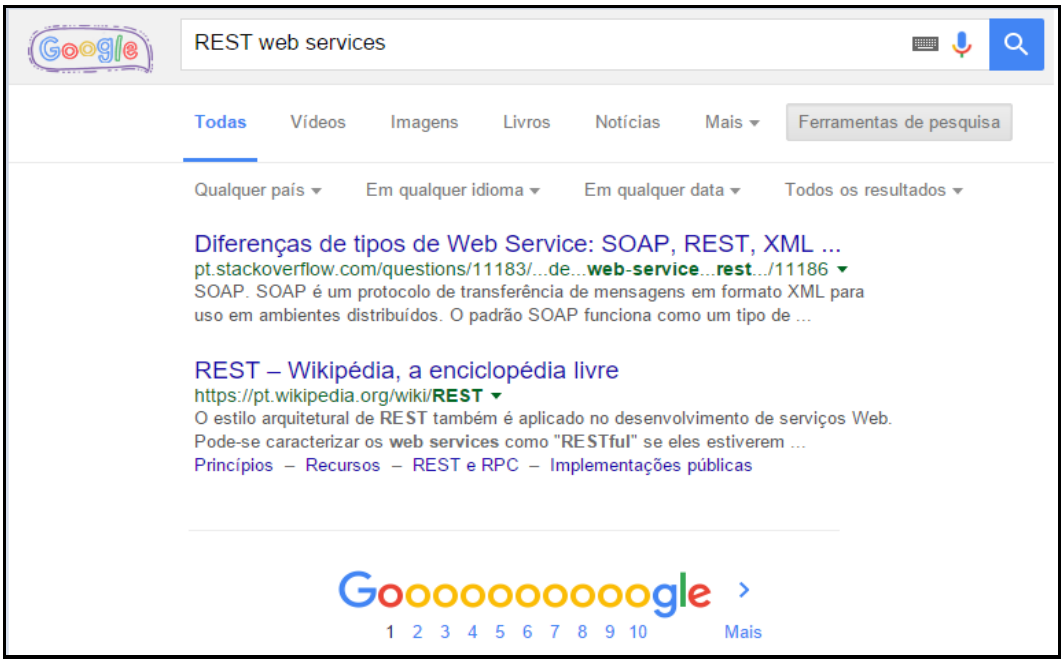

Figura 3 - Exemplo de uma pesquisa na ferramenta de busca Google.

2.3.4. Interface uniforme

Devido à arquitetura orientada a recursos utilizar do protocolo HTTP, todo serviço utilizado, ou executado, é mediado por alguns métodos do protocolo.

A tabela 1, apresenta os métodos que a arquitetura orientada a recursos utiliza, são eles:

MÉTODO	FUNÇÃO
GET	Retorna as estruturas de dados serializadas.
PUT	Atualiza um recurso existente.
DELETE	Apaga um recurso existente.
HEAD	Retorna o cabeçalho de uma representação.
OPTIONS	Retorna quais métodos consegue-se realizar em um recurso.
POST	Criação de um recurso.

Tabela 1 - Métodos HTTP

O método GET, mais comum nas requisições, deve ser usado para páginas que retornam resultados e consultas. Sempre que um endereço de uma página é digitado no navegador uma requisição do tipo GET é realizada. O método PUT deve ser usado para atualizar

registros, o método POST para inserir informações no banco de dados e o método DELETE para excluir dados. (LECHETA, 2015).

Os métodos HEAD e OPTIONS são considerados utilitários. O método HEAD recupera uma representação com metadados, o qual fornece o mesmo que uma requisição GET, porém, sem o corpo da entidade. O método OPTIONS permite saber qual método é permitido realizar em um recurso. (RICHARDSON; RUBY, 2007).

Segundo Richardson e Ruby (2007), o método POST é uma tentativa de criar um novo recurso a partir de outro recurso existente. Esse recurso existente pode ser considerado o pai do novo recurso, se for levado em consideração os termos de estrutura de dados.

Como exemplo de sua utilização, alguns métodos exibidos na tabela 1 são apresentados abaixo:

- **GET** “/empresa/funcionários/código?c=310”: retorna o funcionário com o código de identificação 310 informada como parâmetro.
- **PUT** “/faculdade/aluno/107/atualiza?nome=Novo Nome&ra=123”: atualiza os dados do aluno com a identificação 107 da faculdade, de acordo com os novos dados passados nos parâmetros.
- **DELETE** “/loja/cliente/200/apagar”: apaga os dados e o cliente com a identificação 200.
- **POST** “/varejo/produto/novo?codigo=350&nome=Produto Novo”: cria um novo produto em um varejo de acordo com os dados passados nos parâmetros.

2.4. VANTAGENS E BENEFÍCIOS EM RELAÇÃO À *SERVICE-ORIENTED ARCHITECTURE* (SOA)

Ao longo deste capítulo foram abordados características e conceitos da arquitetura orientada a recursos, apresentando os conceitos que a definem, como também, suas propriedades. Essa arquitetura que utiliza das tecnologias da Web, como o protocolo HTTP, veio para definir regras concretas para desenvolver serviços RESTful, tendo como maior característica os recursos.

Segundo Paiva (2013), a arquitetura permite que o desenvolvimento da aplicação seja realizado em um tempo menor, trazendo como vantagem, a diminuição do custo da

mesma, tendo em vista que o programador foca somente no que lhe interessa, eliminando o restante que é comum a todas as aplicações.

Interagir com *Web Services* baseados em SOAP de inicio parece ser uma boa ideia, porém, ao se aprofundar nos estudos percebe-se que não é tão simples quanto se gostaria. (SAUDATE, 2013). O SOAP por ser um grande XML, perdeu espaço para os serviços RESTful, devido estes exibirem uma sintaxe mais enxuta e trocar informações em formatos mais leves. No mundo móvel, aplicações necessitam economizar recursos quando trafegam informações, e esse é o motivo do SOAP estar perdendo espaço para o REST. (LECHETA, 2015).

Os serviços SOAP, WSDL e a pilha WS - * não têm os benefícios dos orientados a recursos, como também, não são endereçáveis, não podem ser colocados em cache ou se apropriarem bem da conectividade e não respeitam qualquer interface uniforme. Na prática, voltam também a apresentar problemas de interoperabilidade quando servem a uma grande variedade de clientes. (RICHARDSON; RUBY, 2007).

De acordo com os autores da ROA, Richardson e Ruby (2007), a abordagem orientada a recursos é completa, conforme os princípios de Turing. Ela pode modelar qualquer aplicação, mesmo uma de um tipo complexo.

Alguns *Web Services* SOA são muito complexos, como é o caso dos que utilizam do envelope WSDL, além de serem complexos, são extensos, tanto que é necessário utilizar de ferramentas para modelar eles. Segundo Richardson e Ruby (2007), a maior parte da WSDL é realizada por ferramentas automatizadas. Para serviços simples, pode-se partir de um arquivo WSDL gerado automaticamente e realizar alguns ajustes no mesmo, porém, para desenvolver, além disso, é necessário o uso de ferramentas.

Equivale lembrar que, a segurança desses é forte e bem vistas pelos desenvolvedores. Conforme Richardson e Ruby (2007), os conceitos de segurança nos protocolos baseados em SOAP, são mais especificados e difundidos do que nos protocolos HTTP.

Porém, os serviços orientados a recursos também garantem segurança, seja pelo protocolo HTTPS, um protocolo HTTP com uma camada de segurança adicional, o protocolo SSL/TLS, ou seja, pela simplicidade dos mesmos. De acordo com Richardson e Ruby (2007), o HTTPS provou ser suficiente na prática, para garantir a segurança, por exemplo, de cartões de crédito enviados pela rede. A melhor proteção que se pode ter é de que as arquiteturas orientadas a recursos promovem a simplicidade e a uniformidade.

Quando o programador está tentando desenvolver uma aplicação que seja segura, nem a complexidade, nem um grande número de interfaces são vantajosos.

2.5. EXEMPLO MOTIVACIONAL DA ARQUITETURA

A Figura 4 apresenta uma representação arquitetural da ROA, onde o servidor com os serviços RESTful foi desenvolvido seguindo os princípios da arquitetura ROA.

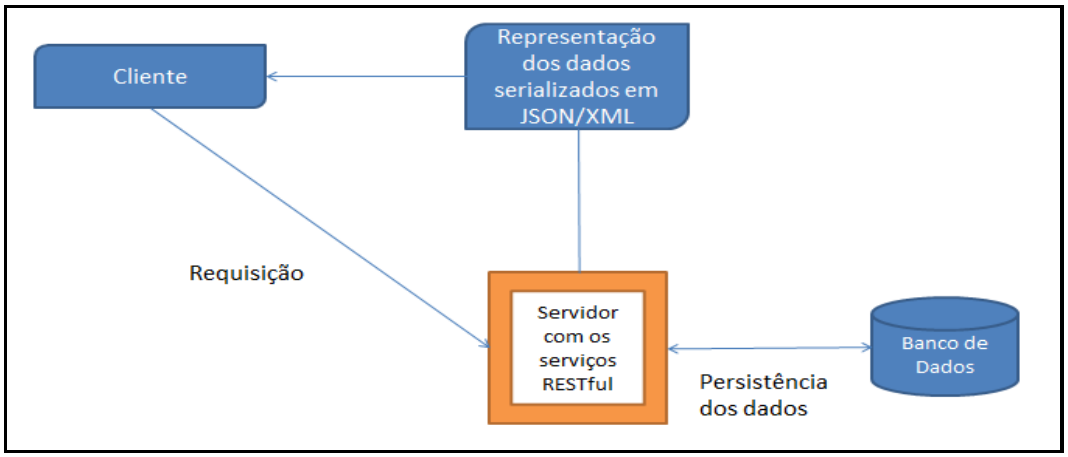

Figura 4 - Representação da Arquitetura ROA.

Quando o cliente, por meio de uma URI, como exemplo de uma: “/biblioteca/livros”, enviar uma requisição GET para o servidor, este último vai buscar os dados no banco e retornar uma representação de determinado recurso para o cliente. Seguindo o exemplo da URI, esse recurso seria uma lista de livros para o cliente. Sua representação poderia ser em JSON ou em XML.

Caso o cliente queira realizar outra requisição, uma POST ou DELETE, o fluxo seria o mesmo. Por meio de uma URI o cliente envia a requisição, o servidor recebe e faz a persistência no banco, e como recurso a ser retornado, o servidor poderia encaminhar uma representação para o cliente contendo uma confirmação de que a operação foi bem sucedida.

3. PLATAFORMA JAVA ENTERPRISE EDITION (JEE)

Atualmente, as aplicações Web possuem regras de negócio muito complicadas, e codificá-las representam um grande trabalho. Essas regras são conhecidas como requisitos funcionais de uma aplicação. Outros requisitos, os não funcionais, são os que necessitam ser alcançados por meio da infraestrutura, os quais são: persistência em banco de dados, transação, acesso remoto, *Web Services*, gerenciamento de *threads*, gerenciamento de conexões HTTP, cache de objetos, gerenciamento da sessão web, balanceamento de carga, dentre outras. Em um cenário em que os programadores teriam muito trabalho a fazer caso necessitassem escrever o código dos requisitos não funcionais, a Sun criou uma série de especificações que, quando implementadas, podem ser usadas para tirar proveito e reutilizar toda essa infraestrutura já pronta. A plataforma Java EE. (CAELUM, 2016).

A plataforma Java *Enterprise Edition* (JAVA EE) é um padrão para desenvolver aplicações Java de grande porte voltada, ou não, para internet. Ela contém bibliotecas e funções para elaborar software distribuído, baseado em componentes modulares que são processados em servidores de aplicações e que suportam escalabilidade, segurança, integridade e outros requisitos de aplicações corporativas. (ANDRADE, 2015).

Essa plataforma contém algumas especificações, APIs, para proceder com um desenvolvimento para a mesma. Segundo Andrade (2015), a plataforma Java EE é conhecida como uma plataforma guarda-chuva, porque dispõe de várias especificações com visões distintas. Dentre essas as mais notáveis são:

- ***Servlets***: componentes Java processados no servidor para criar conteúdo dinâmico para a web, como exemplo, HTML e XML.
- **JavaServer *Pages* (JSP):** um aperfeiçoamento de *Servlets*, o qual possibilita que aplicações Java web tornem-se mais simples de manter. Parecida com tecnologias como ASP e PHP, mas com a diferença de ser mais robusta, pois conta com as facilidades da plataforma Java.
- **JavaServer *Faces* (JSF):** *framework* web baseado em Java, o qual tem o foco de minimizar a dificuldade de desenvolver telas de sistemas web, por meio de um modelo de componentes reutilizáveis.

- **Java *Persistence* API (JPA):** API padrão do Java que faz a persistência de dados, onde essa utiliza do conceito de mapeamento objeto-relacional. Uma grande vantagem da mesma é que ela incorpora uma alta produtividade para sistemas que usam o banco de dados.
- ***Enterprise* Java *Beans* (EJB):** componentes que processam em servidores de aplicação e tem como principais objetivos, oferecer simplicidade e produtividade no desenvolvimento de componentes distribuídos, transacionados, seguros e portáveis.

A Figura 5 mostra a plataforma Java EE, e como a mesma possibilita desenvolver aplicações multicamadas com base na web.

Figura 5 - Plataforma Java EE (IN: ORACLE, 2013).

3.1. HISTÓRICO E EVOLUÇÃO

A plataforma Java *Enterprise Edition*, como já foi definida, consiste em um conjunto de APIs que a compõem. Porém, nem sempre ela teve esse nome, de inicio, o seu nome foi J2EE. É importante ressaltar que, os dados apresentados nessa seção foram retirados da linha do tempo da história do Java, a qual pode ser encontrada no site Oracle (Java *Timeline*, 2016).

Em 1999, a Sun em seu pacote de anúncios para o mercado, apresentou o J2EE, Java 2 *Plataform*, *Enterprise Edition*, com foco em aplicações que utilizam servidores. No ano de 2001, em uma nova versão, trazia como algo novo o suporte nativo para as tecnologias *Web Services*. Com o decorrer dos anos, novas APIs foram sendo integradas e evoluídas nessa plataforma, e gerando novas versões.

O nome mudou para Java EE, tornando-se conhecida por esse nome no ano de 2006, junto com o Java SE, e o Java ME. De novidade, o Java tornou disponível seu código sob a licença GNU, *General Puplic License*, a mesma do Linux.

Na versão lançada em 2009, trouxe como uma das novidades os *Web Services* RESTful. E em sua última versão, lançada em 2013, dispõe de uma infraestrutura escalável, simplificando a criação de aplicativos HTML5.

3.2. REST *WEB SERVICES*

O estilo de arquitetura *Representational State Transfer* (REST) foi originado por Roy Fielding, no ano 2000, em sua tese de doutorado. Fielding (2000) descreve-a como um estilo híbrido, que deriva de vários estilos arquiteturais baseados em rede e combinado com restrições que a definem como uma interface uniforme para sistemas hipermídia, como a Web.

REST, para uma melhor definição, é um conjunto de regras a seguir para o desenvolvimento de *Web Services* que se baseia nos métodos do protocolo HTTP. Segundo Richardson e Ruby (2007), a arquitetura orientada a recursos é REST, porém, REST não é uma arquitetura, mas um conjunto de critérios de desenvolvimento.

Os serviços web que seguem esse conjunto de regras são denominados de RESTful. De acordo com Lecheta (2015), REST é um conjunto de técnicas de desenvolvimento de *Web Services* fortemente ligadas ao protocolo HTTP e a outros padrões da web. *Web Services* construídos com estas técnicas são chamados de *Web Services* RESTful.

As restrições, citadas no primeiro parágrafo, que compõem o estilo arquitetural são:

Cliente-Servidor: Onde o cliente envia uma requisição com dados, ou exigindo dados ao servidor, o qual, este tem um conjunto de serviços disponíveis. Este vai interpretá-la e decidir se irá executá-la ou não, assim, logo enviar uma resposta ao cliente que efetuou a requisição.

Essa separação por preocupação, o cliente de dados e serviços do servidor, é o princípio por trás de cliente-servidor, permitindo melhorar a portabilidade da interface do usuário por meio de várias plataformas e melhorar a escalabilidade, simplificando os componentes do servidor. (FIELDING, 2000)

Sem Estado: É uma restrição para interação cliente-servidor, onde o cliente fica com o estado da seção. Cada requisição efetuada pelo mesmo deve conter todos os dados para que o servidor consiga interpretá-la sem utilizar de seus serviços e dados. Deste modo, o servidor fica isento do estado de seção, não armazenando nenhum.

A restrição sem estado potencializa as propriedades visibilidade, confiabilidade e escalabilidade. A visibilidade é aperfeiçoada, pois um sistema de monitoramento não tem a necessidade de analisar de uma única requisição para determinar a natureza da mesma. A confiabilidade é aprimorada porque esta torna mais simples a tarefa de recuperação de falhas parciais. Também, a escalabilidade é melhorada já que não é necessário armazenar o estado entre solicitações, tornando sua tarefa de gerenciamento de recursos mais simples e rápida. No entanto, esta restrição traz certa desvantagem, ou seja, a queda de desempenho da rede, já que o estado de seção não fica no servidor, assim, interagindo muitas vezes com o cliente e sobrecarregando a rede. (FIELDING, 2000)

Cache: Para melhorar o desempenho da rede afetada pela restrição sem estado e a restrição de sistemas em camadas, descrita acima, a restrição cache armazena dados no servidor para que não sejam requisitados ao cliente quando necessário, diminuindo, assim, a interação entre os dois. Segundo Fielding (2000), a vantagem de acrescentar a restrição cache é que ela tem o potencial para suprimir parcial ou completamente algumas interações, aprimorando a eficiência, escalabilidade e desempenho para o usuário, diminuindo a latência média de interações.

Interface Uniforme: Como principal restrição, sua interface uniforme entre o cliente e servidor, é a característica que permite diferencia-la de outros estilos arquiteturais baseados em rede. Para obter essa interface uniforme, Fielding (2000) destaca quatro condições para orientar o comportamento dos componentes: Identificação de recursos, manipulações de recursos, mensagens auto descritivas e hipermídia.

Sistema em camadas: A restrição de sistema em camadas foi adicionada no REST, a fim de melhorar ainda mais a escalabilidade.

O sistema em camadas permite uma arquitetura composta de camadas hierárquicas, onde os componentes só podem enxergar a camada com qual está interagindo. As camadas podem ser usadas para encapsular serviços legados e para proteger os novos

serviços. Como desvantagem do mesmo é sua perda de desempenho, devido a sobrecarga e a latência adicionada no tratamento de dados. (FIELDING, 2000).

***Code-On-Demand*:** Como restrição opcional, esta restrição permite que o cliente possa fazer download e executar o código na forma de *applets* ou *scripts*. Deste modo, o cliente ganha simplicidade em seu lado, melhorando sua extensibilidade, porém, como desvantagem diminui sua visibilidade. (FIELDING, 2000).

3.2.1. APIs e bibliotecas

Para os *Web Services* REST existem diversas APIs, para diferentes linguagens, como exemplo, a API da *Amazon*, S3 REST API. O *Twitter* também possui uma API REST, assim como o *Flickr* tem a sua API.

Para a linguagem Java existe a API Java API *for* RESTful *Services*, mais famosa por JAX-RS. Esta API possui uma implementação oficial da Oracle, a biblioteca Jersey, a qual será utilizada neste trabalho. Também, a API JAX-RS possui outras implementações, como a biblioteca Apache Wink, Restlet e a RESTEasy.

3.2.2. Uso de XML e JSON para troca de mensagens

Para troca de mensagens irá ser detalhado duas linguagens de marcação de dados, as quais são comumente utilizadas em REST: *Extensible Markup Language* (XML) e JavaScript *Object Notation* (JSON).

A linguagem XML, também usada em *Web Services* baseados no protocolo SOAP, é de fácil entendimento e de grande similaridade ao HTML pelo seu nível de estruturação de código em *tags*. Saudate (2013) relata a semelhança entre as duas, porém cita as características do XML. Como exemplo, o arquivo XML tem um elemento raiz, similar ao HTML, mas sua diferença é a flexibilidade do elemento ter qualquer nome. Outra característica citada pelo autor é as seções específicas do XML quanto ao fornecimento de instruções de processamento, onde são interpretadas por processadores XML que não fazem parte dos dados.

Uma dificuldade quanto a essa linguagem de marcação é quando muitos dados são retornados por meio de um documento XML, a mensagem acaba se tornando grande, com diversas *tags* pai e filhas, tornando-a difícil sua leitura.

Em contrapartida, e uma alternativa quanto ao XML, outro formato de resposta de mensagem, é o JSON, devido sua simplicidade. Este é uma estrutura de dados em JavaScript, onde a estrutura é composta sempre por chave e seu respectivo valor.

Os objetos JSON podem ser compactados e enviados em apenas uma linha. Onde, que se comparado ao XML, é mais enxuta e seu formato é menor, pois não é necessário abrir e fechar as *tags*, assim, diminui o tráfego de dados na rede, essencial para aplicativos de dispositivos móveis (LECHETA, 2015).

3.2.3. Exemplo motivacional

Abaixo pode se observar um método para validar credenciais, o qual retorna uma estrutura de dados JSON. Consegue-se notar o uso do GET, método HTTP que retorna uma representação de um recurso.

```
@Path("/login")
public class Login {
 @GET
 @Path("/realizar")
 @Produces(MediaType.APPLICATION_JSON)
 public String realizar(@QueryParam("usuario") String usuario, @QueryParam("senha")
String senha){
    String resposta = "";
    if(verificaCredencial(usuario, senha)){
      resposta = Utils.elaborarJSON("login", usuario, true);
    }
    return resposta;

 }}
```

Como já descrito, o conceito de representação é a maneira de representar os dados em estruturas serializadas. Pode-se observar um objeto JSON, constituído por <chave>:<valor>, que o método acima se encarrega de retornar quando o cliente abrir a URI “/login/realizar?usuario=Filipe&senha=123”, passando os devidos parâmetros.

```
{ "login":
       {"usuario" : "Filipe",
          "senha" : "123",
```

```
        "status" : "true"
        }
}
```

Caso o método que verifica as credenciais retornasse, com a mesma URI apresentada, um objeto serializado na linguagem de marcação XML, a seguinte representação poderia ser obtida.

```
<login>
    <usuario>Filipe</usuario>
    <senha>123</senha>
    <status>true</status>
</login>
```

4. PLATAFORMA GOOGLE ANDROID

No decorrer destes últimos anos em que a mobilidade surgiu e tornou-se popular entre as pessoas de todo o mundo, certas praticidades foram sendo implantadas e adotadas no decorrer destes anos pelos seus usuários.

Com esses dispositivos, alguns cabíveis na palma da mão, como os *smartphones*, o usuário consegue ter grandes funcionalidades a centímetros de distância, como exemplo, o seu acesso aos e-mails. O que antes era necessário ter um computador conectado a internet para poder visualizá-los, com esses consegue-se lê-los, em qualquer lugar, a qualquer momento tendo apenas o aplicativo de e-mail instalado e uma internet móvel, a que as operadoras de telecomunicação oferecem aos seus clientes.

Não só isso, existem aplicativos dos mais variados tipos e gostos, culinária, jogos, informativos e muitos outros. Alguns desses caem na graça, facilmente, de quem usufrui da tecnologia móvel, a exemplo aplicativos que utilizam do serviço de internet para troca de mensagens entre celulares. A grande vantagem é que não há cobrança pelas mensagens transmitidas, utilizando o exemplo acima, como uma troca de e-mails, que da mesma forma não existe cobrança. Também, dependendo do aplicativo consegue-se efetuar uma ligação para outro aparelho, enviar áudios, arquivos de imagem, vídeos e demais funções que cada qual tem para se diferenciarem.

Esses dispositivos e aplicativos necessitam de um sistema operacional (SO) para funcionarem, e é nesse exato contexto que a plataforma Google Android se encaixa. Essa plataforma atualmente está presente em relógios (Android *Wear*), celulares, *tablets*, televisões e agora, também, em carros (Android Auto). De acordo com o site oficial do Android (2016), o Android move mais de um bilhão de smartphones e *tablets*.

A plataforma Google Android foi construída a partir de uma aliança entre empresas renomadas mundialmente, liderando esse projeto a gigante Google. Esta aliança foi denominada como *Open Handset Alliance* (OHA).

Segundo Lecheta (2013), o objetivo desse grupo é definir uma plataforma única e aberta para celulares para atender os consumidores com o produto final. Outro objetivo é criar uma plataforma moderna e flexível para o desenvolvimento de aplicações coorporativas. O resultado dessa aliança foi o Android. O autor também exprimiu que a OHA foi criada

para manter uma plataforma-padrão, a qual, todas as tecnologias novas que chegam ao mercado estejam englobadas em uma única solução.

Sendo uma plataforma livre, baseada no Kernel do Linux, este SO é multitarefa, pois este realiza a execução de mais de um aplicativo ao mesmo tempo, de modo que, consegue executar uma atualização de uma aplicação em segundo plano, enquanto ao mesmo tempo permite o usuário jogar.

De acordo com Lecheta (2013), o sistema operacional do Android foi baseado no Linux, kernel 2.6, e o mesmo é encarregado de gerenciar a memória, os processos, *threads*, e a segurança dos diretórios, além de redes e drivers. Os aplicativos disparam um novo processo no sistema operacional, alguns podem ter uma tela de exibição para o usuário, ou, também podem apenas ser executados em segundo plano por tempo indeterminado. Vários processos e aplicações podem ser executados simultaneamente, e o kernel do sistema operacional é o responsável pelo controle de memória.

Outra vantagem por ser uma plataforma livre, é que cada fabricante de aparelhos podem ter o seu próprio Android, personalizando-o. É possível substituir as aplicações nativas por aplicações próprias, sendo assim, os fabricantes podem colocar seus próprios aplicativos dentro do sistema e vender seus dispositivos.

No entanto, esta vantagem não vale somente para os fabricantes de aparelhos, mas também para os usuários do sistema operacional Android. Existem maneiras de trocar a distribuição do aparelho, já que o código-fonte do Android é distribuído, e em certos aparelhos a atualização do sistema não está mais disponível. É possível, também, ter um sistema Android alternativo, como, em questão de exemplo, uma distribuição da *CyanogenMod*.

Conforme Lecheta (2013), o fato de o Android ser de código-aberto auxilia o seu aperfeiçoamento, tendo em vista que desenvolvedores de todos os lugares do mundo podem contribuir para o seu código fonte, incluindo novas funcionalidades ou corrigindo falhas.

Vale salientar que o sistema operacional Android, está em constante evolução e tem muito mais a crescer no mercado, como também, expandir para mais dispositivos de diferentes tipos. Sua versão mais recente, no momento da elaboração deste trabalho, é a versão 6.0, *Marshmallow*.

4.1. CAMADAS DA PLATAFORMA

A arquitetura da plataforma Android é divida em camadas, onde cada camada tem suas funções para gerenciar dentro da plataforma. Conforme Assis (2010), o Android pode ser entendido como uma pilha de softwares, onde, cada pilha agrupa programas que tem funções específicas dentro do sistema operacional.

A Figura 6 apresenta as camadas da plataforma, cinco camadas precisamente, onde se verifica que a terceira camada é dividida em duas, bibliotecas nativas e a execução do Android.

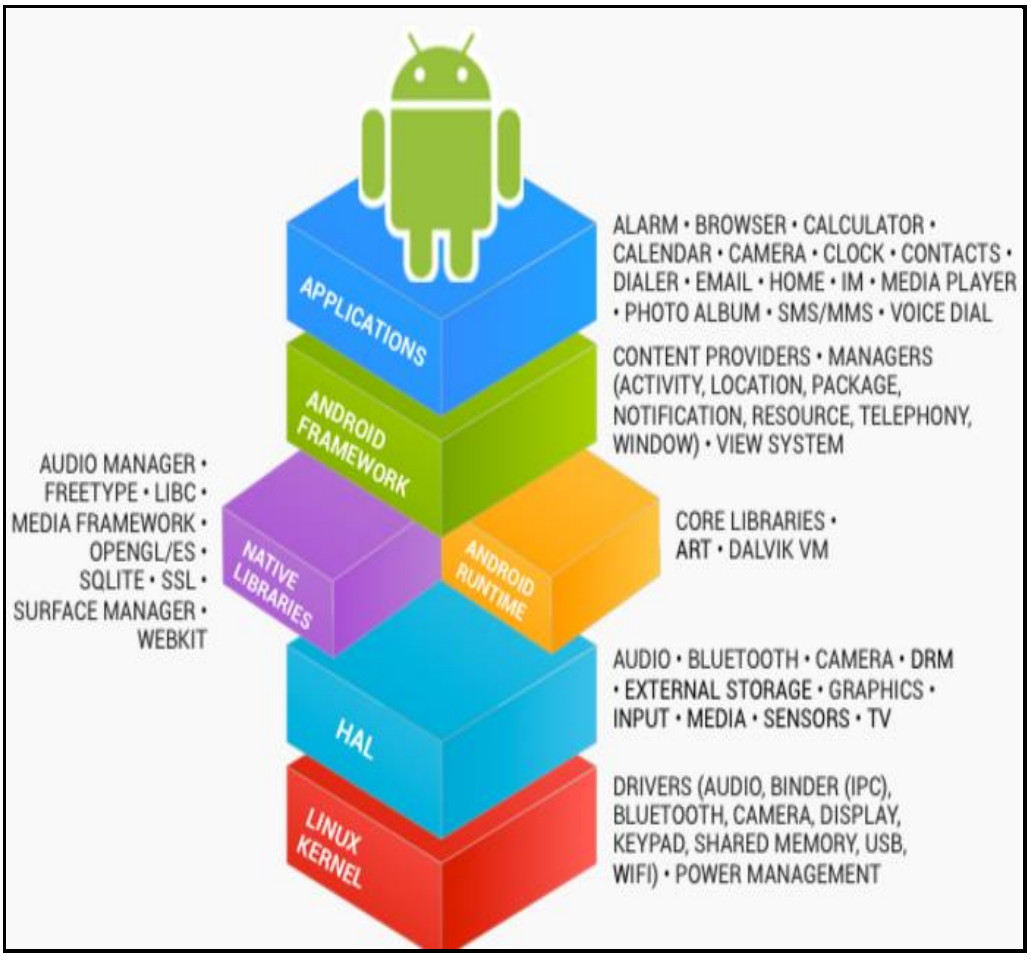

Figura 6 - Camadas da plataforma Android (IN: SOURCE ANDROID, 2016).

Primeira camada, visualizando do nível mais baixo, é o Kernel do Linux. Conforme foi descrito anteriormente, a plataforma foi baseada no Linux, Kernel 2.6. De acordo com Gomes (et. al, 2012), esta camada tem como função a abstração entre o hardware e os aplicativos, também, pelos serviços principais do sistema, como gerenciar a memória e processos. Muitas funções do Kernel são usadas diretamente pelo Android, porém, modificações foram realizadas para melhorar a memória e tempo de processamento das

aplicações. Dentre essas, novos dispositivos de drivers, adições no sistema de gerenciamento de energia e um sistema que possibilita encerrar processos, por meio de critérios, quando a memória está com pouco espaço.

Sua segunda camada é a de Hardware. Nesta camada estão os módulos ou peças que compõem o aparelho celular. Câmeras, módulos *Bluetooth*, sensores de orientação, proximidade, entre outros mais.

Como foi descrito anteriormente, a terceira camada é divida em duas partes. Uma parte é formada por bibliotecas nativas em C/C++, a qual fornece aos desenvolvedores e suas aplicações algumas funções para utilizarem de serviços do sistema operacional. Como exemplo, certa aplicação que necessita persistir dados, requerendo o uso de um banco de dados, para esta existe a biblioteca *SQLite*.

Outra parte dessa camada é onde ocorre a execução do sistema operacional Android, o Android *Runtime*, onde está a máquina virtual Dalvik. Segundo Lecheta (2013), para construir aplicações em Android é utilizada a linguagem Java, porém, não há uma Java *Virtual Machine* (JVM) em seu sistema operacional. No entanto, existe uma máquina virtual chamada Dalvik, a qual tem uma melhor execução em dispositivos móveis.

A quarta camada é a de framework, o Android *Framework*. A camada de *framework* disponibiliza aos desenvolvedores as mesmas interfaces de programação de aplicativos (APIs) que são usadas para aplicações do sistema operacional Android. Esse *framework* foi feito para abstrair a complexidade e facilitar a reutilização de procedimentos. A camada funciona como um caminho para a camada de bibliotecas do sistema operacional, as quais serão acessadas por meio de APIs contidas no *framework* (Gomes et. al, 2012).

Última e quinta camada é a de aplicações, podendo-se ver na Figura 6 que a camada está no topo da pilha. Essa é a camada que mais importa para um usuário comum, que usa apenas dos aplicativos que seu sistema oferece. Nessa camada encontram-se as aplicações nativas, tais como, alarme, calculadora, galeria de fotos, contatos e outras mais aplicações.

4.2. AMBIENTE DE DESENVOLVIMENTO

4.2.1. ANDROID SDK

A fim de se desenvolver para plataforma Android é necessário ter o conjunto de ferramentas Android SDK, que é um software onde o desenvolvedor ou uma empresa, faz os downloads de APIs e determinadas bibliotecas para poder trabalhar com o Android. Também, neste conjunto, há um emulador de Android, o Android *Virtual Devices*, onde o desenvolvedor pode executar seu aplicativo. No próxima seção o emulador será abordado.

Conforme Lecheta (2013), o Android SDK é o software usado para desenvolver aplicações no Android, que possui um emulador de celular, ferramentas utilitárias e uma API completa para a linguagem Java.

4.2.2. ANDROID STUDIO

Para codificar, o desenvolvedor necessita de um ambiente de desenvolvimento integrado, *Integrated Development Environment* (IDE), para auxiliá-lo. A IDE oficial para desenvolvimento Android, atualmente, é o Android Studio. Essa IDE é baseada em outro ambiente de desenvolvimento, a IntelliJ, da JetBrains.

A ferramenta de desenvolvimento Android Studio, conforme o site da mesma, *Developer Android* (2016), conta com um editor inteligente que tem conclusão de código avançado, refatoração e análise do mesmo. Também tem um novo sistema de automatização de builds, o *Grandle*, que gerencia as dependências, ou bibliotecas de aplicativos com o Maven.

Importante ressaltar que existem outros ambientes que se consegue desenvolver para a plataforma, como exemplo o Eclipse, da IBM. A empresa Google descontinuou atualizações para o *plug-in* Android Eclipse Tools, porém, ainda consegue-se desenvolver com a IDE Eclipse, sendo necessário o *plug-in* instalado na IDE.

4.2.3. ANDROID VIRTUAL DEVICES (AVDs)

Dentro do conjunto de ferramentas que compõem o Android SDK está o Android *Virtual Devices*, conhecido também só pelas inicias, AVD. Esta ferramenta é um emulador de celular, onde torna possível que o desenvolvedor teste suas aplicações. A Figura 7 apresenta uma AVD emulando uma versão do Android 4.4.4.

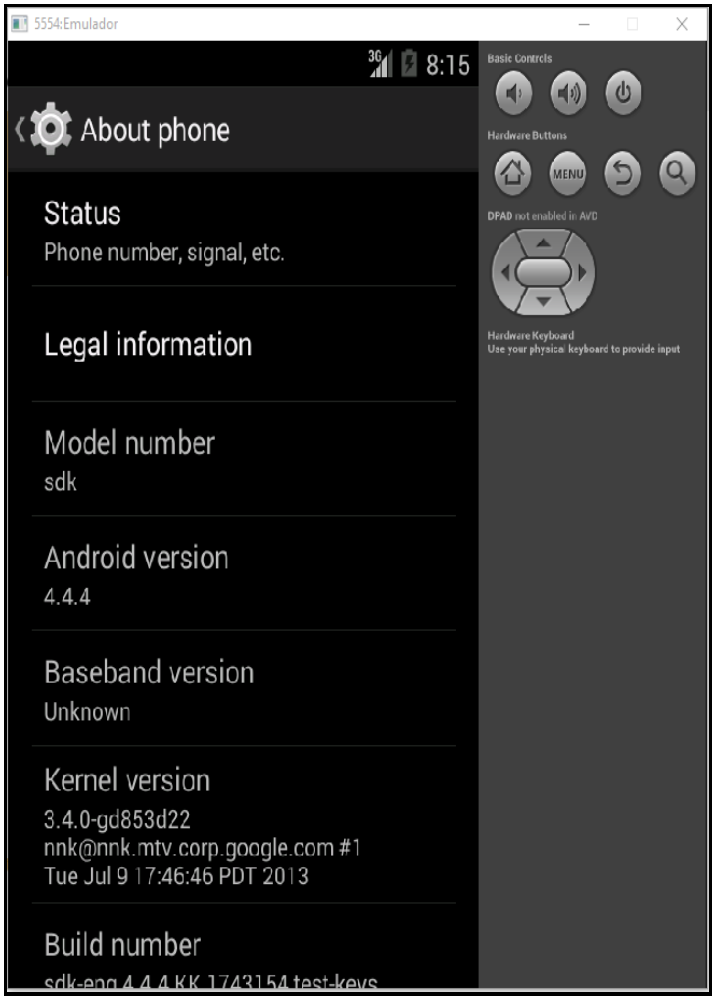

Figura 7 - Emulador Android.

Há diversos dispositivos que o desenvolvedor pode escolher para executar sua aplicação, o mesmo serve para a versão do Android. Pode-se também escolher a CPU que deseja utilizar, a quantidade de memória e algumas outras opções. Simulando verdadeiramente um celular.

Existem, também, outros meios de executar os aplicativos Android, por meio de outros emuladores, como o *Genymotion*. Este é livre para uso pessoal, porém, para programadores independentes, pequenas ou grandes empresas é necessário fazer um

plano pago para poder utilizar-se do mesmo. Esse emulador necessita de uma máquina virtual para funcionar, o *VirtualBox*, da Oracle.

Outra forma de executar uma aplicação é por meio de um celular Android, que necessita estar conectado no computador por meio de uma USB. De acordo com Lecheta (2013), pode-se plugar um celular em uma porta USB do computador e executar os aplicativos no próprio celular. Com isso tem a facilidade de testar os aplicativos em aparelhos reais, tornando o desenvolvimento mais produtivo. O único pré-requisito é ter o driver USB do celular instalado no computador.

4.2.4. APIs e bibliotecas para REST

Os serviços web REST, abordado no capítulo três, é um meio para aplicações interagirem entre si. Assim, uma forma de uma aplicação em Android se comunicar com outra plataforma externa, uma Web, é a partir dos *Web Services.*

Assim, por meio dos mesmos, tanto uma aplicação, ou as duas, podem utilizar desses serviços para ler, atualizar e/ou gravar novos dados em uma base de dados que compartilham, tudo isso em tempo real. Lecheta (2013) relata que, muitas aplicações móveis, cada vez mais, estão trabalhando literalmente conectadas a internet, atualizando e acessando dados em tempo real.

Algumas bibliotecas para Android de serviços web REST, podem ser conferidas: Android Asynchronous HTTP Client; RESTDroid; Retrofit; RoboSpice;

4.2.5. Exemplo motivacional

A Figura 8 mostra um processo de desenvolvimento para construir uma aplicação Android, desde sua configuração em uma máquina, instalando o Android SDK, já descrito em uma das subseções acima, até sua publicação para uso dos usuários.

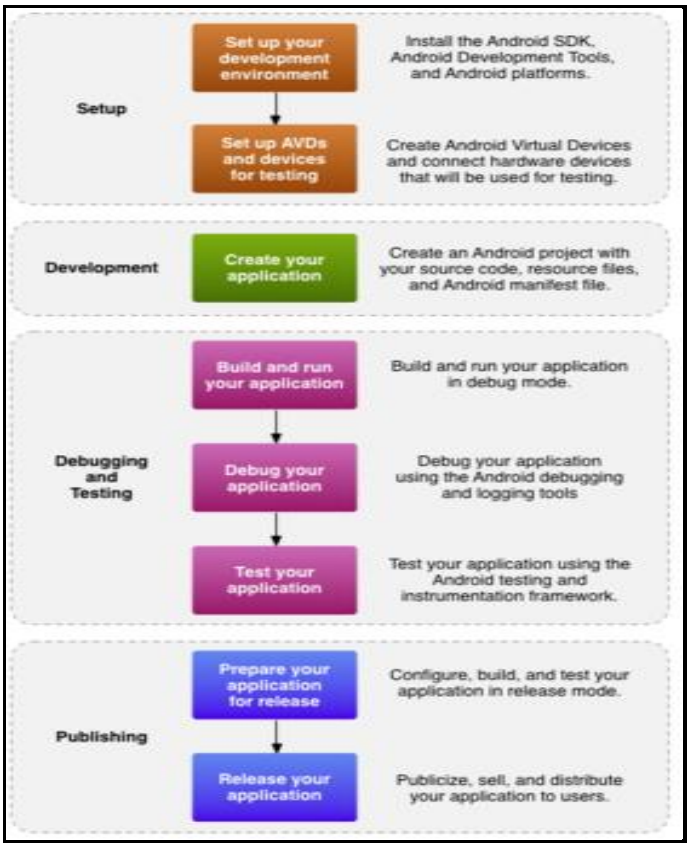

Figura 8 - Processo de desenvolvimento de aplicações Android. (IN: DEVELOPER ANDROID, 2016).

Na Figura 9 é apresentado um exemplo de uma aplicação feita na plataforma Google Android, acessando os *Web Services* RESTful, localizados no servidor, onde esses serviços estão do lado da plataforma Java EE, que é o servidor que se encarrega de fazer o acesso e a persistência no banco de dados.

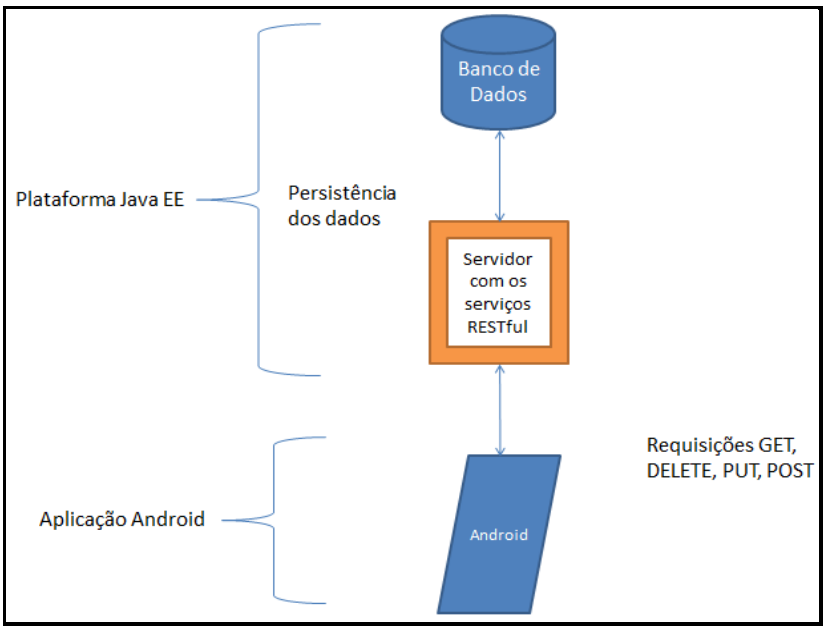

Figura 9 - Exemplo de funcionamento entre uma aplicação Android e *Web Services* RESTful.

Conforme o exemplo motivacional apresentado no Capítulo 3, representando o serviço de verificar as credenciais, e como funcionamento da Figura 9, abaixo é exibido um método que o invoca, por meio da plataforma Android com a URI deste serviço.

```
public void acessandoWebService(){
       RequestParams parametros = new RequestParams();
       parametros.put("usuario","Filipe");
       parametros.put("senha","123");
      AsyncHttpClient clienteAndroid = new AsyncHttpClient();
      clienteAndroid.get("http://IPSERVIDOR:8080/projeto/login/realizar",parametros,
                           new AsyncHttpResponseHandler() {
         @Override
         public void onSuccess(String repesentacao) {
            try {
                 JSONObject objJSON = new JSONObject(repesentacao);
                 if(objJSON.getBoolean("status")){
                   Toast.makeText(getApplicationContext(), "Credenciais válidas " +
               objJSON.getString("usuario"), Toast.LENGTH_SHORT).show();}
                 else{
                   Toast.makeText(getApplicationContext(),"Credenciais inválidas",
Toast.LENGTH_SHORT).show();
                           }
            } catch (JSONException e) {

                     e.printStackTrace();
             }}}); }
```

Importante observar nessa função que os parâmetros da URI são passados por um objeto da classe *RequestParams*, o qual atribui os valores por chave e valor, assim, similar a passagem de parâmetro diretamente na URI. É possível, também, observar no método que caso a requisição tenha obtido sucesso o seu retorno é um texto, e depois o mesmo é convertido para uma representação de um objeto JSON, sendo recuperado pela sua chave.

A aplicação Android neste contexto funciona como o cliente. É ela quem envia ao servidor requisições com as requeridas informações, assim, o último recupera e/ou persiste os dados.

5. PROPOSTA DE TRABALHO

Como proposta, o presente trabalho tem o desenvolvimento de uma aplicação *Web* e uma Android, onde os dados entre as mesmas sejam transparentes, ocorrendo, assim, a interoperabilidade entre elas. Essas aplicações têm como objetivo comercializar produtos de forma genérica, incrementando, com isso, as vendas das empresas que se utilizarem das aplicações, por meio da Internet.

A Figura 10 apresenta uma representação arquitetural de como as aplicações Android e *Web* vão se comunicar, partilhando dos mesmos dados e interagindo entre si.

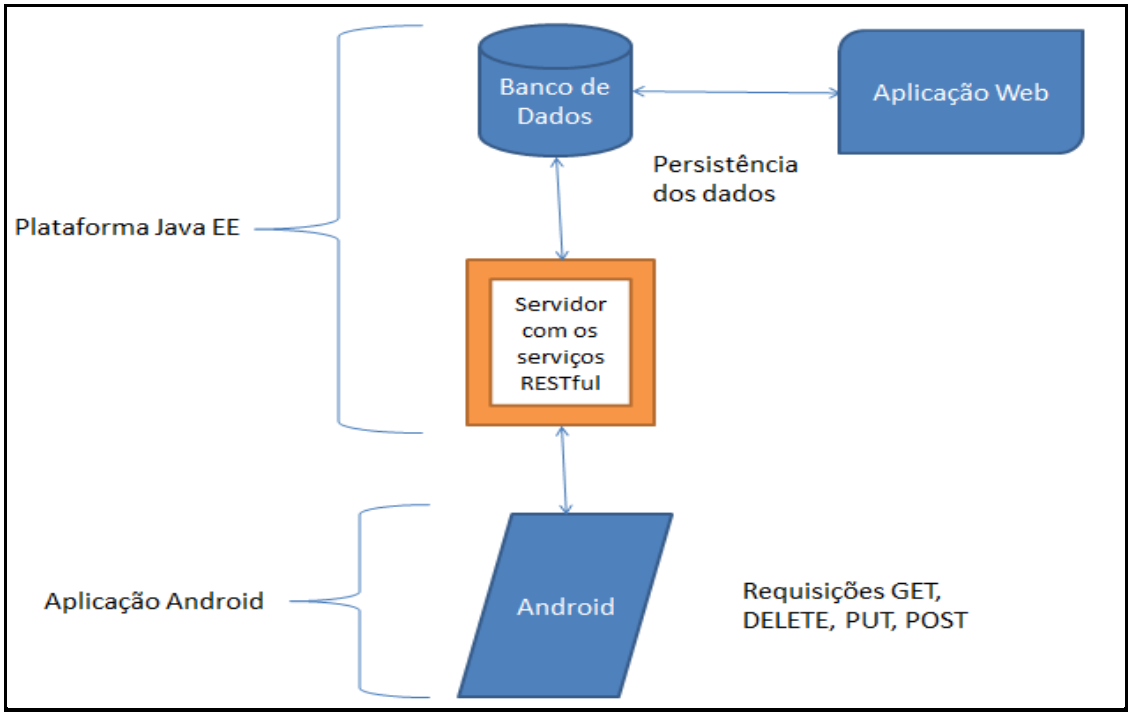

Figura 10 - Comunicação entre as aplicações Android e Web.

A empresa, ou o vendedor, irá utilizar do módulo Web, o qual será desenvolvido com a especificação JSF, da plataforma Java EE. Esse módulo é onde as empresas poderão realizar seu cadastro e de seus produtos para efetuarem as vendas. No mesmo módulo acontecem as configurações dos parâmetros de vendas, como exemplos, o tipo de pagamento que será aceito, e o tempo mínimo para envio do produto físico, inclusive o tempo para cancelamento que o usuário final terá para cancelar uma compra ou alterar algum produto na mesma.

A movimentação das vendas será realizada por meio dos dispositivos móveis Android. O aplicativo, quando acessado pela primeira vez, vai apresentar uma tela para verificar as credenciais do usuário final. Caso este não seja cadastrado, poderá realizá-lo a partir da tela de verificação de credenciais. O cadastro irá requisitar dados necessários para realizar uma compra. Logo, o usuário poderá fazer compras por meio do aplicativo e, também, avaliar as empresas.

A interligação entre os módulos e a base de dados será efetuada por meio dos *Web Services* RESTful, seguindo os conceitos da arquitetura ROA, a qual executará paralelamente no servidor entre a aplicação Web. Esses serviços serão acionados quando o dispositivo móvel necessitar, assim que o usuário final entrar em determinada empresa para realizar uma compra, a aplicação móvel vai apresentar os produtos que estão à venda, recuperando os dados da base por meio dos serviços. Também, quando o usuário efetuar uma compra vai ocorrer a inserção dos dados no banco. Assim, a aplicação Web que partilha da mesma base, recebe a requisição de compra e realiza os processos para confirmação e entrega.

A Figura 11 apresenta um processo de compra realizado entre as aplicações, onde o usuário final, na aplicação Android, efetua a compra, os dados passam pelo servidor com serviços RESTful, chegando a compra na aplicação Web, a qual retorna um novo estado para determinada compra, de acordo com a decisão tomada pelo administrador. A entrega do produto físico ao consumidor determinaria o final do processo.

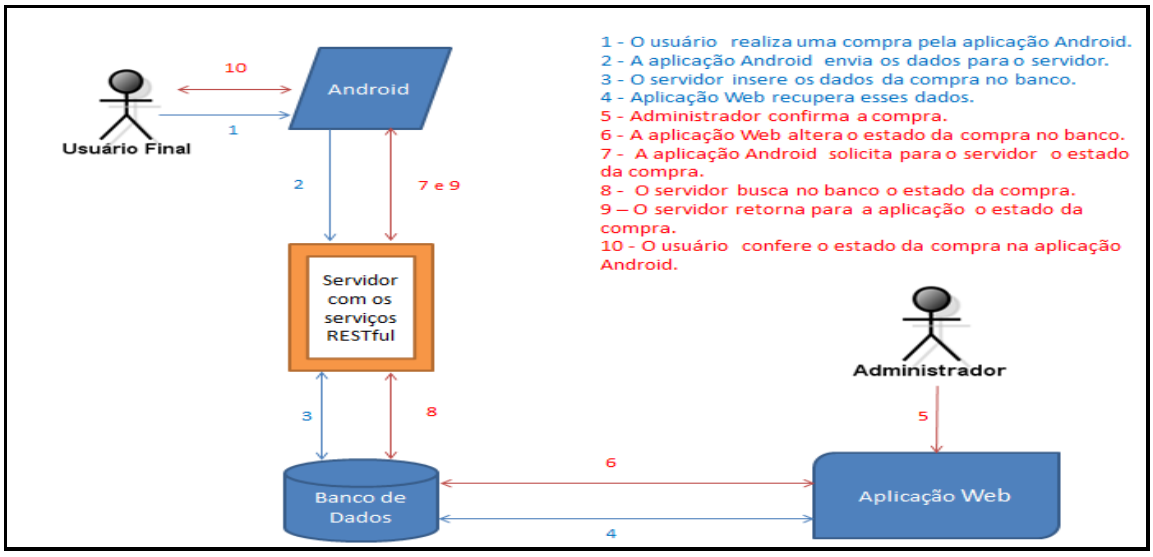

Figura 11 - Processo de compra realizado entre as aplicações.

6. DESENVOLVIMENTO DO TRABALHO

O desenvolvimento do trabalho foi realizado com três tecnologias, sendo duas delas pertencentes à plataforma Java EE e outra à dispositivos móveis.

A especificação *JavaServer Faces*, uma das tecnologias utilizadas, mais conhecida como JSF, é uma tecnologia baseada em componentes reutilizáveis. A outra tecnologia é a *Web Services* RESTful, cujas requisições se baseiam nos métodos do protocolo HTTP. A terceira e última tecnologia utilizada no desenvolvimento deste trabalho é a plataforma Google Android, sistema operacional onde suas aplicações são desenvolvidas com a linguagem Java, além de seu grande destaque nos últimos anos.

O projeto foi divido em três partes, nomeado de **Compra da Hora**, visando à criatividade de trazer um sentido duplo para o nome, onde em sentido tempo pode se fazer a compra em qualquer horário, já que muitas empresas podem comercializar seus produtos, como um supermercado que abre logo pela manhã e uma lanchonete que funcione a noite. O outro sentido é de que fazer a compra pelo aplicativo será tranquila, pela praticidade, simplicidade e facilidade em adquirir um produto.

A Figura 12 apresenta o diagrama de entidade-relacionamento existente entre as tabelas das três partes que compõem o projeto, sendo elas: Android, Web e Serviços Restful. Apenas duas partes persistem dados no banco, aplicação JSF e os serviços Web, de tal modo que, a aplicação Android apenas envia e recebe dados dos *Web Services*.

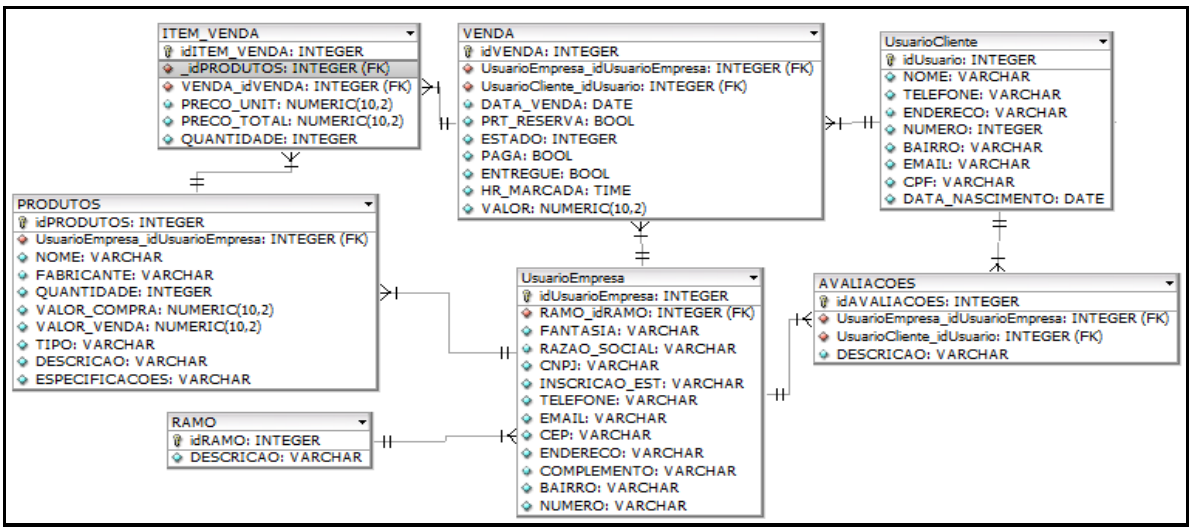

Figura 12 - Diagrama de Entidade Relacionamento (DER).

Entre todas existem algo em comum, como o pacote *Value Object* (VO). Nele estão as classes com os atributos referentes às tabelas, contendo, essas classes, os métodos de acesso aos atributos, os *getters* e *setters.* A diferença do uso desse pacote entre as aplicações Java EE e a de Android são algumas anotações.

As anotações, que são denotadas com o caractere @, são referentes ao Java *Persistence* API (JPA), responsável pela persistência dos dados no banco. Os atributos da classe são mapeados para que o *framework* que implementa a API do JPA possa fazer essa persistência no banco, atualizando, inserindo e consultando os dados por meio do mapeamento, o que permite ao programador trabalhar apenas com objetos da classe ao invés de comandos SQLs.

```
@Entity
@Table(name="ramo")
public class Ramo implements Serializable{

	private static final long serialVersionUID = 1L;

	@Id
	@Column(name="idramo")
	private int idRamo;

	@Column(name="descricao")
	private String descricao;

	/* MÉTODOS CONSTRUTORES E DE ACESSO AOS ATRIBUTOS */
}
```

Acima é apresentado um VO, referente à classe Ramo, podendo-se observar o uso da anotação *@Entity* e outras anotações referentes à tabela e colunas do banco, bem como de mapeamentos do atributo a outras classes que serão explicadas abaixo:

- ***@Entity**:* Todos os VOs que serão manipulados por meio do JPA devem conter a expressão @Entity, que indica que há um mapeamento daquele VO para ser trabalho pelo JPA.

- ***@Table**:* Indica a tabela que aquela classe representa no banco de dados. Por padrão o JPA entende que o nome da classe é o nome da tabela, e esta anotação permite que a tabela e a classe possam ter nomes diferentes.
- ***@Id:*** Indica ao atributo da classe que o mesmo representa a chave primária da tabela.
- ***@Column:*** Indica o nome da coluna que aquele atributo representa no banco de dados. Por padrão, também, o JPA entende que o nome do atributo é o mesmo da coluna no banco, e esta anotação permite que a coluna e o atributo da classe possam ter nomes distintos.

Outro pacote, o qual apenas as aplicações da plataforma Java EE compartilham, é o Data Access Object (DAO). Dentro deste estão as classes que irão fazer a persistência no banco de dados por meio de um *framework* que implementa o JPA, sendo utilizado, neste caso, o *Hibernate*, da RedHat.

Para configurar o JPA é necessário ter um arquivo XML dentro do projeto, contendo as configurações do provedor JPA, como já descrito, o *Hibernate*, do banco de dados, e outras como a de apresentação das instruções SQL de leitura e escrita realizadas no banco direto no *console* da IDE. Abaixo é apresentado o conteúdo do arquivo.

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence version="1.0"

    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence

http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
    <persistence-unit name="persistenciaDados"
        transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <properties>
            <property name="hibernate.connection.username"
                value="postgres" />
            <property name="hibernate.connection.password"
                value="Filipe" />
            <property name="hibernate.connection.driver_class"
                value="org.postgresql.Driver" />
            <property name="hibernate.connection.url"
```

```
                value="jdbc:postgresql://localhost:5432/CompraDaHora" />
            <property name="hibernate.cache.provider_class"
                value="org.hibernate.cache.NoCacheProvider" />
            <property name="hibernate.hbm2ddl.auto" value="update" />
            <property name="hibernate.dialect"
                value="org.hibernate.dialect.PostgreSQLDialect" />
            <property name="hibernate.show_sql"
                value="true" />
            <property name="hibernate.format_sql"
                value="true" />
            <property name="use_sql_comments"
                value="true" />
        </properties> </persistence-unit></persistence>
```

Com o arquivo de configuração é possível obter uma instância de *EntityManagerFactory* por meio de uma unidade de persistência, definida no projeto como "*persistênciaDados*" como pode-se observar no fragmento acima. O objeto *EntityManagerFactory* permite instanciar um objeto do tipo *EntityManager,* o qual é responsável por fazer consultas e persistência de dados no banco de dados. Logo abaixo dois fragmentos de código para melhor entendimento.

```
public class JPAUtil {

	private static EntityManagerFactory emf = null;

	private JPAUtil(){
		emf = Persistence.
				createEntityManagerFactory("persistenciaDados");
	}

	public static EntityManagerFactory getInstance(){
		if(emf == null){
			new JPAUtil();
		}
		return emf;
	}
}
```

```
public class RamoDAO {
	private EntityManagerFactory emf;
	private EntityManager em;
	public RamoDAO() {
		emf = JPAUtil.getInstance();
		em = emf.createEntityManager();
	}
	public Ramo buscarPorIndentificador(int id) throws Exception {
		Ramo objeto = new Ramo();
		try {
			em.getTransaction().begin();
			objeto = em.find(Ramo.class, id);
			em.getTransaction().commit();
			////emf.close();
		} catch (Exception e) {
			em.getTransaction().rollback();

			throw new Exception("Erro na busca por id");
		}

		return objeto;
	}

}
```

6.1. APLICAÇÃO *WEB – JAVASERVER FACES*

O objetivo da aplicação Web é que empresas cadastrem seus produtos para que possam realizar as vendas por meio da aplicação Android. Para desenvolver a aplicação Web foi utilizado o *JavaServer Faces* 2.

A especificação JSF, é baseada no padrão de projeto MVC (*Model View Controller*), usado em diversos *frameworks* como também em projetos, separando o sistema em camadas, em funções diferentes por setor.

Para a manipulação dos dados, tais como, seleção, inserção, deleção e autenticação de dados a partir da lógica de negócio implementado, tem como responsável a camada

Model, que é representada pelos pacotes já descritos, o Value Object e Data Access Object. A camada *View* é encarregada de exibir o *design* para o usuário e também tem como atividade enviar para a camada de controle as ações do usuário, ou seja, suas requisições. Por último, a camada *Controller* tem como função o controle de eventos. Esta camada recebe as requisições do usuário por meio da camada *View*, servindo como ponte entre as camadas de manipulação de dados e de visão que por final recebe os dados requisitados.

A JavaServer Faces, como descrito, é baseada em componentes, o que facilita no desenvolvimento de aplicações e aumenta sua produtividade. Esses componentes são reutilizáveis, como campos de entrada e saída de texto, tabelas, botões e muitos outros. O JSF é altamente flexível. Esta API permite que novos componentes sejam criados ou até mesmo estendidos de componentes existentes dentro da API.

Existem alguns frameworks que agregam mais produtividade e beleza para fazer as interfaces gráficas, com componentes personalizados e estendidos da API JSF, como botões, diálogos, menus, tabelas, layouts e até gráficos. Como exemplo desses *frameworks* o RichFaces, da RedHat, um pouco mais robusto do que outros, mas ainda assim conta com componentes ricos e interessantes, além de contar com um guia on-line para ajudar o desenvolvedor. Outra biblioteca, a qual foi usada para desenvolver a aplicação Web, é o Primefaces, da *Prime Technology*, o qual apresenta componentes mais interessantes do que outros, ultrapassando mais de cem, além de proporcionar grande suporte para programadores usarem seus componentes, temas gratuitos e pagos e suporte para dispositivos móveis.

A interface gráfica da aplicação Web foi feita com a biblioteca do PrimeFaces, aproveitando os seus componentes pré-construídos para fazer a aplicação. Ainda para melhorar o visual, foi feito um tema com a ferramenta *ThemmeRoller*. Esta ferramenta constrói temas em cima da biblioteca JavaScript JQuery, deixando as aplicações que utilizam de seus temas mais interativas e com propriedades gráficas melhores. Além de tudo, ela proporciona um ambiente gráfico para realizar este feito, bastando baixa-lo logo após criar o tema. Dentro do arquivo existem outros que devem ser colocados dentro do projeto Web, sendo estes o *.css*, *.js*, e as imagens.

O primeiro proporciona o *design* da aplicação, características para deixar os componentes do framework igual ao tema que construiu, já o arquivo de JavaScript, o .js, disponibiliza para aplicação algumas funções com eventos de tela, como efeitos ao abrir diálogos ao

acionar botões e dentre outros mais. Por último, as imagens, que são os ícones padrões para utilizar em botões, saída de textos, tabelas, e vários outros.

Para utilizar o tema, o arquivo de script e as imagens, é necessário adicionar em cada tela *.xhtml* o seguinte código:

```
<html>
<h:head>
        <title>Compra da Hora</title>
        <h:outputStylesheet library="css" name="compraDaHora.css"/>
        <h:outputScript library="scripts" name="compraDaHora.js"/>
</h:head>
<h:body>
    ...
</h:body>
</html>
```

Na propriedade *library* deve-se inserir o caminho do diretório, que deve estar dentro do projeto, onde se encontra cada arquivo, comumente colocado dentro de *resources.*

As páginas de interação com o usuário foram criadas com o Extensible HyperText Markup Language (XHTML), a qual pode ser executada em qualquer navegador independente da plataforma ou sistema operacional utilizado. A linguagem XHTML foi reformulada da linguagem de marcação HTML e baseada em xml.

Abaixo é apresentado um fragmento de código, um *template*, algo muito utilizado nas aplicações JSF, pois o mesmo pode ser reutilizado, diminuindo o código e potencializando a produtividade já que o *layout* da página se torna padrão, sendo utilizado em todas as páginas.

```
<?xml version='1.0' encoding='UTF-8' ?>
    <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
    <html xmlns="http://www.w3.org/1999/xhtml"
          xmlns:ui="http://java.sun.com/jsf/facelets"
          xmlns:h="http://java.sun.com/jsf/html"
          xmlns:p="http://primefaces.org/ui"
          xmlns:f="http://xmlns.jcp.org/jsf/core">
        <h:head>
      <title><ui:insert  name="title">Título</ui:insert></title>
           <h:outputStylesheet library="css" name="cmpHora.css"/>
```

```
            <h:outputScript library="scripts" name="cmpHora.js"/>
        </h:head>
        <h:body>
            <p:growl id="growl" life="3000"/>
          <header>
              <div style="margin-left: 8px;">
                  <ui:include src="/menu.xhtml"/>
              </div>
          </header>
            <div style="padding: 0px 8px;">
                    <ui:insert name="body"></ui:insert>
            </div>
        </h:body>
    </html>
```

Pode-se notar no fragmento acima o uso dos *facelets*, biblioteca utilizada para criar modelos de páginas, onde a *tag <ui:insert>* representa um local dentro do modelo que será usado para as páginas que a implementarem, assim, diferenciadas pelo atributo *name*. Logo abaixo o uso de um *template*, para melhor esclarecer:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:p="http://primefaces.org/ui">

    <ui:composition template="/template.xhtml">
        <ui:define name="title">
            <h:outputText value="Compra da Hora"></h:outputText>
        </ui:define>
        <ui:define name="body">
           <p:panel header="Bem vindo ao Compra da Hora">
            <h:graphicImage library="images" name="compradaHora.png"/>
           </p:panel>
        </ui:define>
    </ui:composition>
</html>
```

Para fazer uso do modelo de página é necessário usar a *tag <ui:composition>* e passar no atributo *template,* o caminho onde situa-se a página padrão. No corpo da *tag* implementa os trechos do layout, que no fragmento do *template* foram definidos como "*title*" e "*body*", tal que, são definidos a partir da *tag <ui:define>*.

Analisando o fragmento acima, percebe-se que não foi necessário adicionar o arquivo de *script* e de *design* na tela, devido o *template* já usar esses arquivos, apresentando, assim, uma diminuição de linhas de código e a reutilização do que já está pronto.

A Figura 13 apresenta o resultado dos fragmentos acima:

Figura 13 - Página inicial da aplicação Web

6.2. *WEB SERVICES* RESTFUL

No presente desenvolvimento do projeto, os Web Services Restful criados disponibilizam para aplicação Android alguns serviços e mecanismos para que a mesma consiga consultar e persistir dados no banco de dados.

Os serviços foram desenvolvidos utilizando à biblioteca Jersey, implementação oficial da Oracle em cima da API JAX-RS, facilitando, assim, a troca de mensagens no formato JSON entre a aplicação Android e os *Web Services*. Para usar essa biblioteca é necessário registrar um *Servlet* no arquivo *web.xml* da aplicação.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    metadata-complete="true"
    id="WebApp_ID" version="3.1">
    <display-name>CompraDaHoraRWS</display-name>
    <servlet>
        <servlet-name>Jersey REST Service</servlet-name>
       <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-
class>
        <init-param>
            <param-name>jersey.config.server.provider.packages</param-name>
            <param-value>br.com.rest.servicos</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>Jersey REST Service</servlet-name>
        <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Com o Jersey configurado como Servlet na aplicação é possível utilizar essa biblioteca nos serviços. Seguindo a base de exemplos já mostrados anteriormente, abaixo é exibido o serviço da classe *Ramo.*

```
@Path("/ramo")
@Produces(MediaType.APPLICATION_JSON + ";charset=utf-8")
@Consumes(MediaType.APPLICATION_JSON + ";charset=utf-8")
public class RamoResource {
	private static RamoBO bo = RamoBO.getInstance();
	@GET
	@Path("{id}")
	public String buscarPorIndentificador(@PathParam("id") int id) throws Exception{
		return construir("unico", bo.buscarPorIndentificador(id));
	}
	@GET
	public String buscarTodos() throws Exception{
```

```
            return construir("lista", bo.buscarTodos());
      }
            @Override
      public String construir(String acao, Object objeto) {
            JSONArray json = new JSONArray();
            try {
                  if(objeto != null){
                        if(acao.equals("unico")){
                              Ramo ramo = (Ramo) objeto;
                              JSONObject ob = new JSONObject();
                        ob.put("id", ramo.getIdRamo());
                  ob.put("descricao",ramo.getDescricao());
                                    json.put(ob);

                        }else if(acao.equals("lista")){
                              List<Ramo> c =  (List<Ramo>) objeto;
                              for (Ramo ramo : c) {
                                    JSONObject ob = new JSONObject();
                                    ob.put("id", ramo.getIdRamo());
                                    ob.put("descricao",ramo.getDescricao());
                                    json.put(ob);
                              }}}}catch(Exception e){
                        e.printStackTrace();
                  }
            return json.toString();
}}
```

Analisando o fragmento de código acima, nota-se o uso de algumas anotações e a construção de um objeto JSON para ser retornado a quem invocar o método a partir da URI. Algumas anotações são descritas:

- ***@Path**:* Define o caminho para acessar esse serviço;
- ***@Produces:*** Define o tipo de conteúdo que o método retorna, que no caso deste serviço o conteúdo que os métodos retornam é no formato JSON;
- ***@Consumes:*** Define o tipo de conteúdo que o método consome, que neste caso, também, é no formato JSON;
- ***@PathParam:*** Recebe como parâmetro o valor passado na URI;

A fim de melhorar os exemplos acima apresentados, a Figura 14 exibe o acesso a um serviço da classe Ramo, o método “buscar todos”, utilizando como aplicação cliente um navegador.

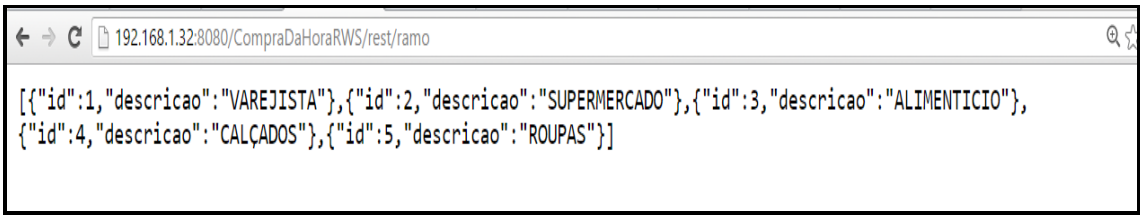

Figura 14 - Acesso a um serviço da classe Ramo

Existem outras anotações que também representam requisições baseadas nos métodos HTTP, são elas:

- **@POST:** Comumente usada para inserir novos registros;
- **@PUT:** Geralmente usada para atualizar registros;
- **@DELETE:** Certamente usada para deletar registros;

Essas três anotações podem ser conferidas no serviço do usuário cliente, apresentado no fragmento de código:

```
	@DELETE
	@Path("{id}")
	public Response remover(@PathParam("id") int id) throws Exception{
		bo.remover(bo.buscarPorIndentificador(id));
		return Response.status(201).entity("Deletado com sucesso").build();
	}
	@POST
	@Path("/post")
	public Response inserirUsuario(String json) {
		try {
			UsuarioCliente usr = jsonParseUsuario(json);
			bo.inserir(usr);
			return Response.status(201).entity("Inserido com sucesso").build();
		} catch (JSONException e) {
		    Return Response.status(406).entity("Erro ao
inserir"+e.getMessage()).build();
		} catch (Exception e) {
		  return Response.status(406).entity("Erro ao inserir" +
e.getMessage()).build();
		}}
```

```
	@PUT
	@Path("/put")
	public Response atualizarUsuario(String json) {
		try {
			UsuarioCliente usr = jsonParseUsuario(json);
			bo.atualizar(usr);
			return Response.status(201).entity("Inserido com sucesso").build();
		} catch (JSONException e) {
			return Response.status(406).entity("Erro ao inserir"+e.getMessage()).build();
		} catch (Exception e) {
			return Response.status(406).entity("Erro ao inserir" + e.getMessage()).build();  }
```

6.3. APLICAÇÃO ANDROID

A aplicação móvel foi desenvolvida com o objetivo de funcionar fazendo uso somente dos *Web Services*, não se utilizando do banco de dados integrado que o Android disponibiliza, o SQLite. O aplicativo tem como função fazer o cadastro do consumidor e, também, visualizar e realizar as compras dos produtos.

O projeto da aplicação Android é dividido em alguns pacotes para facilitar o uso e a reutilização de código, sendo eles: *Value Object* (VO), já descrito, Restfulws, *Adapter* e o *View*.

6.3.1. Pacote *View*

O pacote *View* é responsável por conter as *activities* da aplicação, as quais têm como objetivo fazer o controle de eventos da interface gráfica, ou seja, para cada tela criada é necessário ter uma *acitivty* correspondente e dentro dela deve existir os eventos implementados.

Para construir as interfaces gráficas, os *layouts,* deve-se trabalhar com XML, onde se pode utilizar de componentes avançados, personalizados, imagens e outros para criar as telas, porém, não é necessário codificar as telas somente por meio de *tags*, mas também

é possível montar as telas por meio de um ambiente gráfico, como exemplo a que a ferramenta Android Studio disponibiliza, utilizada neste presente trabalho. Com a ferramenta gráfica é possível montar, alinhar e fazer a tela visualizando ao mesmo tempo o resultado, algo muito útil para desenvolvedores.

A Figura 15 expõe a tela de *login*, representada pelo e-mail e senha, onde, dentro da *activity* apresentada logo após a figura, são usados para autenticar o usuário.

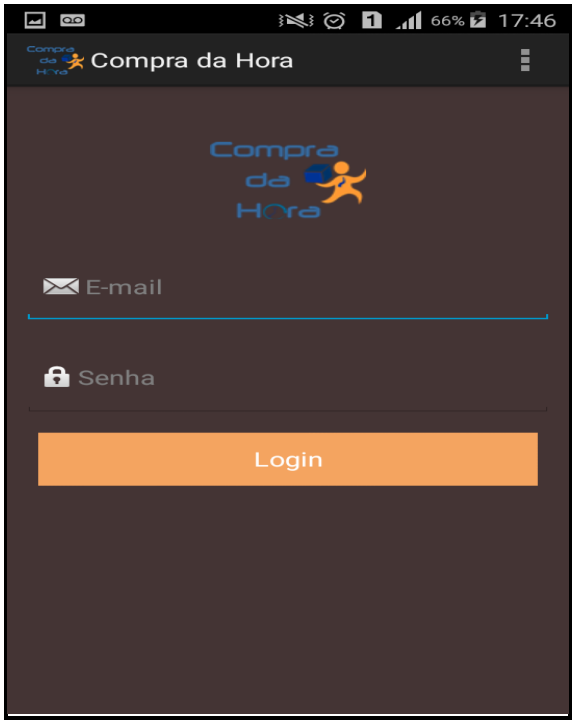

Figura 15 – Tela de *login* da aplicação Android

```
public class MainActivity extends ActionBarActivity {

    private Button btnLogin;
    private EditText txtUsuario;
    private EditText txtSenha;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        txtUsuario = (EditText) findViewById(R.id.txtEmailUsuario);
        txtSenha = (EditText) findViewById(R.id.txtSenhaUsuario);
        btnLogin = (Button) findViewById(R.id.btnLogin);
        btnLogin.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                final String usuario = txtUsuario.getText().toString();
                final String senha = txtSenha.getText().toString();
```

```
                if (!usuario.equals("") && !usuario.equals(" ")) {
                    if (!senha.equals("") && !senha.equals(" ")) {
                        if (verficaConexao()) {
                            if(Utility.validarEmail(usuario)){
                                verificaUsuario(usuario, senha);
                            }else {
                                Toast.makeText(MainActivity.this, "Por favor,
coloque um email válido", Toast.LENGTH_SHORT).show();
                            }}
                    } else {
                        Toast.makeText(MainActivity.this, "SENHA DEVE ESTAR
PREENCHIDA", Toast.LENGTH_LONG).show();
                    }
                } else {
                    Toast.makeText(MainActivity.this, "USUÁRIO DEVE ESTAR
PREENCHIDO", Toast.LENGTH_LONG).show();
                }
            }
        });
    }
```

6.3.2. Pacote Adapter

Em aplicações para a plataforma Android, quando é necessário preencher componentes personalizados, como um *Spinner* ou *ListView,* é preciso fazer uso dos adaptadores, os quais têm como função fazer a adaptação de uma lista de objetos e apresentá-los conforme a implementação nos componentes personalizados. As classes que fazem uso desses adaptadores devem estender de *BaseAdapter ou de ArrayAdapter*, assim, as mesmas têm de implementar os métodos exigidos, os quais são apresentados depois do fragmento de código exibindo um adaptador da classe Ramo.

```
public class RamoAdapter extends BaseAdapter {

    private List<RamoVO> ramos = null;
    private LayoutInflater inflater = null;
    private ViewHolder holder = null;
    public RamoAdapter(Context context, List<RamoVO> ramos){
        inflater = LayoutInflater.from(context);
        this.ramos = ramos;
    }

    static class ViewHolder {
       // public TextView lblId;
        public TextView lblDescricao;

    }

    @Override
    public int getCount() {
```

```
        return ramos.size();
    }

    @Override
    public Object getItem(int position) {
        return ramos.get(position);
    }

    @Override
    public long getItemId(int position) {
        return ramos.get(position).getIdRamo();
    }

    @Override
    public View getView(int position, View view, ViewGroup parent) {
        if(view == null){
            view = inflater.inflate(R.layout.item_lista_ramos, null);
            holder = new ViewHolder();
            //holder.lblId = (TextView) view.findViewById(R.id.lblId);
            holder.lblDescricao = (TextView) view.findViewById(R.id.lblEmpresa);

            view.setTag(holder);
        }else {
            holder = (ViewHolder) view.getTag();
        }

        RamoVO ramoVO = ramos.get(position);
       // holder.lblId.setText(String.valueOf(ramoVO.getIdRamo()));
        holder.lblDescricao.setText(ramoVO.getDescricao());

        return view;
    }

    public void swapItens(List<RamoVO> ramos){
        this.ramos = ramos;
        notifyDataSetChanged();
    }
}
```

- **getCount**: Retorna a quantidade de elementos pertencentes a lista de objetos;
- **getItem**: Retorna um objeto da lista de acordo com a posição passada por parâmetro;
- **getItemId**: Retorna um identificador de acordo com a posição passada por parâmetro. O identificador, normalmente corresponde ao código de cada objeto;
- **getView**: Responsável por atribuir valores aos componentes personalizados;

6.3.3. Pacote Restfulws

O pacote Restfulws é responsável por conter o acesso aos serviços Web dentro da aplicação móvel, que realiza, por meio de URIs, acesso aos Web Services.

A biblioteca utilizada para fazer o papel de cliente da aplicação é a Android *Asynchronous Http Client*, a qual foi baseada nas bibliotecas *Apache's HttpClient*, e usa dos métodos HTTP para realizar as requisições ao servidor.

Abaixo é disponibilizado dois fragmentos de código que tem como objetivo realizar a requisição *@GET,* percorrendo os dados formatados em JSON e transformando-os em uma lista de objetos da classe de Ramos.

```
public class RamoRWS {

    private static AsyncHttpClient clienteAndroid = new AsyncHttpClient();

    private static String TODOS = "ramo";

       public static void selecionarTodos(RequestParams params,
AsyncHttpResponseHandler asyncHttpResponseHandler){
        clienteAndroid.get(IConstantsRWS.IP_PORTA + IConstantsRWS.PROJETO +
TODOS, params, asyncHttpResponseHandler);
    }
}
public class RamoListAcitivity extends Activity {
    private List<RamoVO> ramos = null;
    private RamoVO ramo = null;
    private ListView lstView;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.lista_ramos);
        lstView = (ListView) findViewById(R.id.lstView);
        ramos = new ArrayList<RamoVO>();
        adapter = new RamoAdapter(RamoListAcitivity.this, ramos);
        lstView.setAdapter(adapter);
        lstView.setCacheColorHint(Color.TRANSPARENT);
        carregarRamos();
    }

    private void carregarRamos() {

        RamoRWS.getInstance().selecionarTodos(null, new
AsyncHttpResponseHandler() {
            @Override
            public void onSuccess(int statusCode, Header[] headers, byte[]
responseBody) {

                String jsonString = new String(responseBody);
                try {
                    JSONArray array = new JSONArray(jsonString);
                    ramos = new ArrayList<RamoVO>();
                    for (int i = 0; i < array.length(); i++) {
```

```
                    JSONObject jsObj = array.getJSONObject(i);
                    RamoVO r = new RamoVO();
                    r.setIdRamo((Integer) jsObj.get("id"));
                    r.setDescricao((String) jsObj.get("descricao"));
                    ramos.add(r);
                }
                adapter.swapItens(ramos);
            } catch (JSONException e) {
                e.printStackTrace();
            }
        }});}
```

Analisando os fragmentos, pode-se notar o uso dos objetos *JSONObject* e JSONArray, os quais foram usados para construir a mensagem JSON do serviço de *Ramo*. Dessa forma, na aplicação Android foi feito o caminho inverso para a leitura dos dados serializados.

Na Figura 16, são apresentados os dados serializados recebidos e já formatados, e, logo após, estes dados impressos na tela.

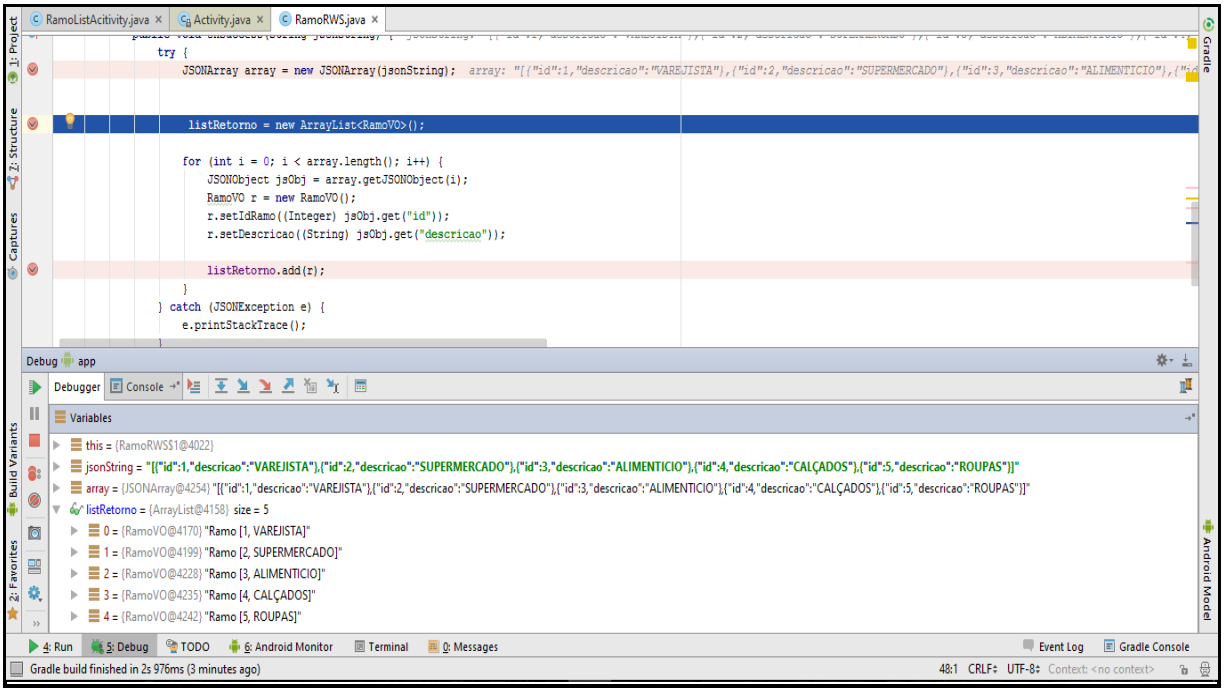

Figura 16 – Dados recebidos do serviço de ramos e formatados como objeto.

A Figura 17 mostra os dados recebidos do serviço da classe Ramos e impressos na tela.

Figura 17 – Dados serializados formatados e impressos na tela.

Para finalizar os exemplos já apresentados, serão exibidos pedaços de código representando as requisições *@POST* e *@PUT* do usuário cliente, seguindo o serviço de usuário cliente já apresentado no desenvolvimento do trabalho.

```
public class UsuarioClienteRWS {

    private static UsuarioClienteRWS INSTANCE = null;
    private static AsyncHttpClient clienteAndroid = null;
    private boolean bVerificaUsuario = false;
    private static String POST = "cliente/post";
    private static String PUT = "cliente/put";
    private UsuarioClienteRWS(){
        clienteAndroid = new AsyncHttpClient();
    }

    public static UsuarioClienteRWS getInstance(){

        if(INSTANCE == null){
            INSTANCE = new UsuarioClienteRWS();
        }

        return INSTANCE;
    }


    public static void verificarUsuario(String usuario, String senha,
AsyncHttpResponseHandler asyncHttpResponseHandler){
        clienteAndroid.get(IConstantsRWS.IP_PORTA + IConstantsRWS.PROJETO +
"cliente/usuario/" + usuario +"/"+ senha, asyncHttpResponseHandler);
    }

    public static void inserirUsuario(Context context, StringEntity entity,
AsyncHttpResponseHandler asyncHttpResponseHandler){
        entity.setContentType(new BasicHeader(HTTP.CONTENT_TYPE,
"application/json"));
```

```
        clienteAndroid.post(context, IConstantsRWS.IP_PORTA +
IConstantsRWS.PROJETO + POST, entity, "application/json",
asyncHttpResponseHandler);

    }

    public static void atualizarUsuario(Context context, StringEntity entity,
AsyncHttpResponseHandler asyncHttpResponseHandler){
        entity.setContentType(new BasicHeader(HTTP.CONTENT_TYPE,
"application/json"));
        clienteAndroid.put(context, IConstantsRWS.IP_PORTA +
IConstantsRWS.PROJETO + PUT, entity, "application/json",
asyncHttpResponseHandler);

    }
}
```

Pode se observar no fragmento acima o uso das requisições *put* e *post*, as quais têm como diferença da requisição *get,* o envio de dados ao servidor por meio de uma StringEntity, sendo, no caso do desenvolvimento desse trabalho um objeto do tipo String com dados do tipo JSON para serem interpretados no lado do servidor.

```
public class DadosClienteActivity extends ActionBarActivity {

public void salvarDados() {

        usuarioClienteVO.setNome(
                txtNome.getText().toString());
        usuarioClienteVO.setCpf(
                txtCpf.getText().toString());
        usuarioClienteVO.setEndereco(
                txtEndereco.getText().toString());
        usuarioClienteVO.setNumero(
                Integer.parseInt(txtNumero.getText().toString()));
        usuarioClienteVO.setBairro(
                txtBairro.getText().toString());
        usuarioClienteVO.setTelefone(
                txtTelefone.getText().toString());
        usuarioClienteVO.setEmail(
                txtEmail.getText().toString());
        usuarioClienteVO.setDataNascimento(
                DateUtil.parse(txtDataNasc.getText().toString()));

if (inserir) {
   try {
      UsuarioClienteRWS.getInstance().inserirUsuario(
               DadosClienteActivity.this, new
StringEntity(usuarioParseJson(usuarioClienteVO)),
                                    new AsyncHttpResponseHandler() {
   @Override
   public void onSuccess(int statusCode, Header[] headers, byte[] responseBody)
{
     super.onSuccess(statusCode, headers, responseBody);
                        if (statusCode == 201) {
                            Toast.makeText(
                                    DadosClienteActivity.this, "Parabéns! Agora
```

```
 você pode usar o aplicativo e desfrutar dos bons produtos :)",
Toast.LENGTH_LONG).show();
                        }
                    }
                });
            } catch (UnsupportedEncodingException e) {
                e.printStackTrace();
            } catch (JSONException e) {
                e.printStackTrace();
            }
        } else if (!inserir) {

            try {
                UsuarioClienteRWS.getInstance().atualizarUsuario(
                        DadosClienteActivity.this, new
StringEntity(usuarioParseJson(usuarioClienteVO)),
                            new AsyncHttpResponseHandler() {
                    @Override
                    public void onSuccess(int statusCode, Header[] headers,
byte[] responseBody) {
                        super.onSuccess(statusCode, headers, responseBody);
                        if (statusCode == 201) {
                            Toast.makeText(DadosClienteActivity.this, "Dados
atualizados com sucesso!", Toast.LENGTH_LONG).show();
                        }
                    }
                });
            } catch (UnsupportedEncodingException e) {
                e.printStackTrace();
            } catch (JSONException e) {
                e.printStackTrace();}}}}
```

O fragmento acima apresenta uma *activity* recuperando os dados do usuário cliente para enviá-los ao servidor. Analisando o código, pode-se notar o envio de um objeto *String,* serializado por meio da StringEntity, contendo dados do tipo JSON, tornando, assim, o envio de mensagem de maneira fácil e simples entre o cliente e o servidor.

7. CONCLUSÃO

Com os avanços e a grande utilidade dos serviços das aplicações móveis, já descritos no início deste trabalho, sobre as aplicações móveis, surge diversas oportunidades e ideias lucrativas para empresas investirem em aplicativos para esses aparelhos, que acompanham grande parte da população e permitem, como o próprio nome denomina, a mobilidade e o fácil acesso aos dados.

Este presente trabalho teve como objetivo desenvolver uma pesquisa teórica, tendo em vista a aquisição e o compartilhamento de conhecimentos acerca do desenvolvimento de Web Services em REST. Em complemento à pesquisa, foi conduzido um trabalho de desenvolvimento de um ambiente capaz de se comunicar com a base de dados por meio dos conhecimentos adquiridos.

O desenvolvimento deste trabalho buscou a implementação de uma definição arquitetural orientada a recursos, utilizando-se a plataforma Java, bem como a plataforma Google Android. Dessa forma, foram propostos serviços que possibilitam a utilização da aplicação *Mobile* em conjunto à Web.

Conclui-se que os objetivos propostos nesta pesquisa foram alcançados, podendo-se afirmar que a troca de dados realizados por meio dos serviços orientados a recursos traz, de maneira simples, uma melhor fluidez e agilidade na troca de mensagens. Juntamente com os serviços, os dispositivos móveis possibilitam muito mais do que lazer e contribuem com distintas áreas e atividades, ressaltando o crescente investimento por parte das empresas de tecnologia com o objetivo de expandir os negócios por meio de dispositivos móveis.

7.1. TRABALHOS FUTUROS

Para trabalhos futuros, pode-se desenvolver uma melhor implementação para fornecer novos níveis de segurança a todos os módulos, assegurando um maior controle na troca de dados entre as aplicações Web e *Mobile*. Outro recurso a ser desenvolvido, tendo a base de implementação de mais níveis de segurança, pode ser a adoção de novas funcionalidades de cartões de crédito e outros meios de pagamento, para que os clientes

das empresas efetuem transações monetárias e de produtos por meio de novos serviços Web.

Somando-se ao aprimoramento dos níveis de segurança, diversas melhorias podem ser propostas nas aplicações, de tal modo que, análise de dados das compras e avaliações, realizadas pelo cliente, resultem em indicadores para as empresas. Assim, não só o aprimoramento de recursos para as empresas será desenvolvido, mas também para os clientes, a fim de que tenham controle das finanças a partir de seus dados dentro do aplicativo, gerando relatórios e gráficos na aplicação.

REFERÊNCIAS

ALVES, Emanuel José Branco. **Comunicação e filiação em redes ad-hoc móveis com participantes desconhecidos**. 2013. 66p. Dissertação (mestrado) – Engenharia Informática – Universidade de Lisboa, 2013.

ALVES, Fagner Valério de Freitas**. Utilização De *Web Services* Para Integração De Sistemas.** 2012. 31p. Monografia – Unidade Tiradentes – FATEC-SP Faculdade de Tecnologia de São Paulo, São Paulo, 2012.

ANDRADE, Thiago Faria. **Java EE 7 com JSF, PrimeFaces e CDI.** 2. ed. AlgaWorks Softwares, 2015.

ANDROID. **História do Android**. Disponível em: <https://www.android.com/intl/pt-BR_br/history/>. Acesso em: 25 de Janeiro de 2016.

ASSIS, Ângelo Ferreira. **Proposta de Middleware para compressão adaptativa de dados em ambientes Android**. 2010. 35p. Monografia – Instituto de Ciências Exatas – Universidade Federal de Ouro Preto, Ouro Preto, Minas Gerais, 2010.

CAELUM. **O que é Java EE?.** Disponível em: < https://www.caelum.com.br/apostila-java-web/o-que-e-java-ee/>. Acesso em: 20 de Janeiro de 2016.

DEVELOPER ANDROID. **Develop.** Disponível em: <http://developer.android.com/intl/pt-br/develop/index.html>. Acesso em: 27 de Janeiro de 2016.

FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures.** 2000. 162p. Tese (doutorado) - University of California, Irvine, 2000.

GOMES, Rafael Caveari; FERNANDES, Jean Alves; FERREIRA, Vinicius Corrêa. **Sistema Operacional Android.** 2012. Universidade Federal Fluminense. Disponível em: < http://www.midiacom.uff.br/~natalia/2012-1-sisop/tgrupo1.pdf >. Acesso em: 07 de Fevereiro de 2016

GOUVEIA, Miguel. GOUVEIA, Vitorino. **Service Oriented Architecture (SOA), Desafios para o Processo de Desenvolvimento de Software**. Disponível em: <http://isg.inesc-id.pt/alb/Modules/StandardToolkit/Documents/ViewDocument.aspx?Mid=379&ItemId=242 > Acesso em: 11 de Outubro de 2015.

JACYNTHO, Mark Douglas de Azevedo. **Processos para Desenvolvimento de Aplicações Web.** 2008. 25p. Monografia – Departamento de Informática – PUC Pontifícia Universidade Católica Do Rio De Janeiro, Rio de Janeiro, 2008.

LECHETA, Ricardo R. **Google Android -** Aprenda a criar aplicações para dispositivos móveis com o Android SDK. 3. ed. São Paulo: Novatec Editora, 2013.

LECHETA, Ricardo R. ***Web Services* RESTful –** Aprenda a criar *Web Services* RESTful em Java na nuvem do Google. 1. ed. São Paulo: Novatec Editora, 2015.

ORACLE. **Java Timeline** – Java 1995 – 2015, 20 years. Disponível em: < http://oracle.com.edgesuite.net/timeline/java/>. Acesso em: 23 de Janeiro de 2016.

PAIVA, Mayco Ribeiro De. **Abordagem Do Modelo Arquitetural Orientado a Recursos para Integração de Aplicações Java EE e Google Android.** 106p. Monografia - Fundação Educacional do Município de Assis – FEMA – Assis, 2013.

POLÔNIA, Pablo Valério. **Proposta de Arquitetura Orientada a Recursos para Scada na Web.** 2011. 142p. Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-graduação em Engenharia de Automação e Sistemas.

RICHARDSON, Leonard; RUBY, Sam. **RESTful Serviços Web** – *Web Services* para o Mundo Real. 1. ed. Tradução de Eveline Vieira e Patricia Azeredo. Rio de Janeiro, 2007.

SAUDATE, Alexandre. **REST Construa APIs Inteligentes de Maneira Simples.** 1. ed. São Paulo: Caso do Código, 2013.

SILVESTRE, Erich. POLÔNIA, Pablo Valério. **Uma Aplicação da Arquitetura Orientada a Recursos.** 2008. 271p. Monografia - Departamento de Informática e Estatística – UFSC Universidade Federal de Santa Catarina, Santa Catarina, 2008.