DIEGO D'LEON NUNES
DIÓGENES APARECIDO REZENDE

APLICATIVO PARA CONSULTA DE NOTAS

UNIVERSIDADE DO VALE DO SAPUCAÍ
POUSO ALEGRE – MG
2015

DIEGO D'LEON NUNES
DIÓGENES APARECIDO REZENDE

APLICATIVO PARA CONSULTA DE NOTAS

Trabalho de Conclusão de Curso apresentado ao Curso de Sistemas de Informação da Universidade do Vale do Sapucaí como requisito parcial para obtenção do título de bacharel em Sistemas de Informação

Orientador: Prof. MSc. Roberto Rocha Ribeiro

UNIVERSIDADE DO VALE DO SAPUCAÍ
POUSO ALEGRE – MG
2015

SUMÁRIO

INTRODUÇÃO

Atualmente, com os avanços tecnológicos, as pessoas estão cada vez mais conectadas e procuram soluções para seus problemas, que possam ajudá-las de forma rápida e fácil. Segundo Lecheta (2013), tanto as empresas quantos os desenvolvedores buscam plataformas modernas e ágeis para a criação de aplicações. Esse fato contruibuiu consideravelmente para o crescimento das plataformas móveis de comunicação.

Uma das áreas que mais se expandiu nos últimos anos é a de telefonia móvel. Monteiro (2012, p.1) afirma que "os telefones celulares foram evoluindo, ganhando cada vez mais recursos e se tornando um item quase indispensável na vida das pessoas". Essa evolução no *hardware* possibilitou o crescimento, mobilidade e portabilidade do *software*.

Muito das coisas que antes eram feitas somente em computadores *desktops* já podem ser realizadas nos celulares, como transferências bancárias, localização de taxi, conversas com amigos, entretenimento com jogos e vídeos, entre outros.

Ainda de acordo com Monteiro (2012), a plataforma *Android* se destaca no mercado devido ao grande número de aparelhos espalhados pelo mundo e pela facilidade que provêem aos desenvolvedores. Esta plataforma foi utilizada para o desenvolvimento de vários trabalhos de conclusão de curso como por exemplo Mendes (2011), que criou um aplicativo para que as bandas musicais pudessem ter mais interação com seus fãs. Oglio (2013), do Centro Universitário Univates, criou um sistema que permite o acesso ao portal virtual da sua faculdade.

Pelas facilidades que *smartphones* provêem para conseguir informações rápidas a qualquer hora e local, criou-se um utilitário que possibilita aos usuários consultarem as suas notas, presenças e provas agendadas no portal do aluno.

O objetivo principal desta pesquisa é desenvolver um aplicativo, para dispositivos móveis, na plataforma *Android*, que permita aos alunos da Universidade do Vale do Sapucaí consultarem suas notas, presenças e provas agendadas. Porém o objetivo principal pode ser desmembrado em objetivos menores e mais concisos, com a finalidade de conseguir realizá-lo com maior eficácia. Estes, por sua vez são:

- Levantar requisitos do *software* proposto de acordo com as necessidades dos discentes.
- Desenvolver o aplicativo para dispositivos móveis na plataforma *Android*.
- Desenvolver um *web service* para prover os dados necessários ao aplicativo proposto.

Com esses passos espera-se fazer um *software* eficaz que auxiliará no dia-a-dia dos alunos. A escolha por fazer um aplicativo se deu pela necessidade em acessar o portal do aluno para ter informações referente às disciplinas. Com o projeto espera-se contribuir socialmente facilitando o acesso dos usuários às suas notas, provas agendadas e faltas.

O resultado final desta pesquisa também auxiliará os alunos do curso de Sistemas de Informação que necessitarem saber como se desenvolve um aplicativo na plataforma *Android* ou implementar mais funcionalidades nesse projeto.

A plataforma *Android* será utilizada devido à grande popularidade do sistema operacional. Visando facilitar as pesquisas aos conteúdos publicados no portal do alunos, foi desenvolveu-se um App[1], pela qual os discentes os terão facilmente, pois serão notificados quando houver alguma mudança, como por exemplo, ao ser lançada uma nota.

[1] Abreviação para a palavra *Application*

2 QUADRO TEÓRICO

Neste capítulo serão descritos os principais conceitos e características das tecnologias utilizadas para o desenvolvimento dos *softwares* propostos nos objetivos deste trabalho.

2.1 *Java*

Conforme Deitel e Deitel (2010), *Java* é uma linguagem de programação orientada objetos, baseada na linguagem C++, desenvolvida pela empresa *Sun Microsystem* no ano de 1995, por uma equipe sob liderança de James Gosling.

Segundo Caelum (2015a), para executar as aplicações, o *Java* utiliza uma máquina virtual denominada JVM[1], que liberta os softwares de ficarem presos a um único sistema operacional, uma vez que o programa conversa diretamente com a JMV e fica por conta dela traduzir os *bytecodes* gerados pelo compilador para linguagem de máquina.

De acordo com Oracle (2015a), o *Java* está presente em mais de um bilhão de dispositivos como celulares, computadores, consoles de *games* e pode ser considerado, seguro, rápido e confiável.

Hoje, com a acenção do *Android*, a tendência é aumentar cada vez mais o desenvolvimento em *Java*, uma vez que os aplicativos Android são desenvolvidos nessa linguagem. Nessa pesquisa a linguagem de programação *Java* será utilizado no desenvolvimento tanto do aplicativo quanto do *web service*.

2.2 *Android*

Segundo Monteiro (2012), *Android* é um sistema operacional baseado em *Linux* que utiliza a linguagem de programação *Java* para o desenvolvimento de seus aplicativos. Criado especialmente para dispositivos móveis, começou a ser desenvolvido no ano de 2003 pela então empresa Android Inc, que em 2005 foi agregada ao Google. A partir de 2007 o projeto *Android* uniu-se a *Open Handset Alliance*, uma associação de empresas de *softwares*, *hardwares* e te-

[1] JVM - Java Virtual Machine

lecomunicações, que tem por finalidade desenvolver uma plataforma para dispositivos móveis que seja completa, aberta e gratuita.

Krazit (2009) afirma que o sistema pode rodar em equipamentos de diversos fabricantes, evitando assim ficar limitado a poucos dispositivos. Conforme informações do site Android (2015a), hoje em dia existe mais de um bilhão de aparelhos espalhados pelo mundo com esse sistema operacional.

De acordo com Monteiro (2012), as aplicações são executadas em uma máquina virtual Java denominada *Dalvik*. Cada aplicativo, usa uma instância dessa máquina virtual tornando-o assim mais seguro. Por outro lado, os *softwares* só podem acessar os recursos do dispositivo, como uma lista de contatos, caso seja formalmente aceito pelo usuário nos termos de uso ao instalá-lo.

As configurações de uma aplicação na plataforma *Android* ficam salvas em um arquivo XML denominado *AndroidManifest.xml*, que se localiza na pasta raiz do projeto. Para Lecheta (2010), as informações devem estar entre *tags* correspondentes ao recurso. Para ser possível acessar a *internet* pelo aplicativo é preciso declarar a permissão de acesso da seguinte forma: `<uses-permission android:name="android.permission.internet"/>`.

Lecheta (2010) diz que as *intents* são recursos tão importantes que podem ser consideradas como o coração do *Android* e que estão presentes em todas as aplicações. De acordo com K19 (2012, p.29), "são objetos responsáveis por passar informações, como se fossem mensagens, para os principais componentes da API do *Android*, como as *Activities*, *Services* e *Broad-Cast Receivers*". Monteiro (2012) diz que as *Intents* são criadas quando se tem a intenção de realizar algo como por exemplo compartilhar uma imagem, utilizando os APP's já existentes no dispositivo. Existem dois tipos de Intents:

- *Intents* implícitas: quando não é informada qual *Activity* deve ser chamada, ficando assim por conta do sistema operacional verificar qual a melhor opção.
- *Intents* explícitas: quando é informada qual *Activity* deve ser chamada. Usada normalmente para chamar *activities* da mesma aplicação.

Segundo K19 (2012), uma aplicação *Android* pode ser construída com quatro tipos de componentes: *Activity*, *Services*, *Content Providers* e *Broadcast Receivers*.

As *activities* são as telas com interface gráfica, que permitem interações com os usuários. De acordo com Lecheta (2013), cada *activity* tem um ciclo de vida, uma vez que ela pode estar sendo executada, estar em segundo plano ou totalmente destruída.

Toda vez que é iniciada uma *activity*, ela vai para o topo de uma pilha denominada *activity stack*. O bom entendimento de seu ciclo de vida é importante, pois quando uma aplicação é interrompida, é possível salvar as informações ou ao menos voltar ao estágio a qual o usuário se encontrava. Na Figura 1 é demostrado o ciclo de vida de uma *activity*.

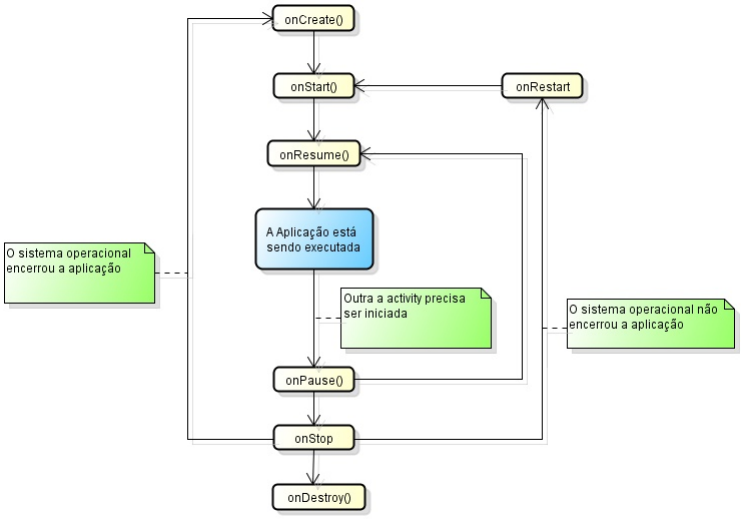

Figura 1 – Ciclo de Vida de uma *Activity*. **Fonte:**Lecheta (2010)

Para que se possa entender melhor, imagina-se o seguinte cenário: um usuário entra no aplicativo de notas da Univás. Para que a *activity* seja criada, é chamado o método `onCreate()`, logo após é executado o método `onStart()` e ao finalizar do ciclo anterior é chamado o `onResume()`, só a partir de então, a *activity* é visualizada pelo discente. Contudo, durante a navegação, o aluno recebe uma ligação, então nessa hora o sistema operacional chama o método `onPause()` para interromper a aplicação e abrir uma outra *activity* para que o usuário possa atender a chamada telefônica. É possível, nesse método, salvar informações que o usuário está utilizando. Ao concluir o método de pausa, é executado o método `onStop()`, a partir de agora a *activity* da Univás não será mais visível ao usuário. Ao encerrar a ligação, há dois caminhos possíveis de se percorrer, o primeiro, seria o caso do sistema operacional encerrar completamente a aplicação, por necessidade de liberar espaço em memória. Dessa forma será necessário chamar o método `onCreate()` novamente seguindo o ciclo normal, porém se não for encerrada completamente, ao findar a ligação será executado o método `onRestart()` e voltar para a *activity* ao qual o usuário se encontrava. Por fim ao encerrar uma *activity* será chamado o método `onDestroy()` finalizando assim o ciclo de vida.

No arquivo *AndroidManifest.xml*, as *activities* devem estar entre as tags `<activity>` `</activity>` e a *activity* principal, ou seja, pela qual será iniciada a aplicação deve conter a tag `<intent-filter>` além de `<action android:name="android.intent.action.MAIN"/>`

indicando que essa atividade deverá ser chamada ao iniciar a aplicação e `<category android:name="and` que implica que esse APP ficará disponível junto aos outros aplicativos no dispositivo.

A *Activity* a ser utilizada para iniciar a aplicação é uma *Navigation Drawer*, que segundo o site Android (2015b), que exibe do lado esquerdo as principais funções do *software*, semelhante a um menu e fica normalmente escondida aparecendo apenas quando clicado no canto superior esquerdo.

Segundo Lecheta (2010), a classe *Service* existe com intuito em executar processos que levarão um tempo indeterminado para serem executados e que normalmente consomem um alto nível de memória e processamento. Esses processos são executados em segundo plano enquanto o cliente realiza outra tarefa. Assim um usuário pode navegar na internet enquanto é feito um *download*. O serviço é geralmente iniciado pelo *Broadcast Receiver* e quem o gerencia é o sistema operacional que só o finalizará ao concluir a tarefa, salvo quando o espaço em memória é insuficiente.

Para Lecheta (2010), a função da classe *Content Provider* é prover conteúdos de forma pública para todas as aplicações, dessa forma essa classe possibilita às aplicações consultar, salvar, deletar e alterar informações no *smartphone*. Assim afirma Lecheta (2010, p.413) "O *Android* tem uma série de provedores de conteúdo nativos, como, por exemplo, consultar contatos da agenda, visualizar os arquivos, imagens e vídeos disponíveis no celular". Portanto, um contato pode ser salvo por um aplicativo e alterado por outro.

Para Lecheta (2010), a classe *Broadcast Receiver* é muito importante para a plataforma *Android*, uma vez que ela é responsável por agir em eventos de uma *intent*.

Essa classe sempre é executada em segundo plano, portanto, quando uma pessoa está utilizando uma aplicação e recebe uma mensagem de SMS, o *Broadcast Receiver* capta essa informação e a processa sem a necessidade do cliente ter que parar de realizar suas tarefas.

A configuração de um *Broadcast Receiver* é feita no *AndroidManifest.xml* pela *tag* `<receiver>` junto com a *tag* `<intente-filter>` e conterá a rotina a ser realizada quando for chamada.

Em uma aplicação, um elemento fundamental é a interface gráfica, que deverá ser organizada, simples e elegante. Conforme Monteiro (2012) esses são os principais *Layouts* do sistema operacional Android:

- *LinearLayout*: permite posicionar os elementos em forma linear, dessa forma quando o dispositivo estiver em forma vertical os itens ficarão um abaixo do outro e quando estiver na horizontal eles ficarão um ao lado do outro.

- *RelativeLayout*: permite posicionar elementos de forma relativa, ou seja um *widget* com relação a outro.

- *TableLayout*: permite criar *layouts* em formato de tabelas. O elemento *TableRow* representa uma linha da tabela e seus filhos são as células. Dessa maneira, caso um *TableRow* possua dois itens, significa que essa linha tem duas colunas.

- *DatePicker*: *widget* desenvolvido para a seleção de datas que podem ser usadas diretamente no *layout* ou através de caixas de diálogo.

- *Spinner*: *widget* que permite a seleção de itens, similar ao *combobox*.

- *ListViews*: permite exibir itens em uma listagem. Dessa forma, em uma lista de compras, clicando em uma venda é possível listar os itens dessa venda selecionada.

- *Action Bar*: um item muito importante, pois apresenta na parte superior aos usuários as opções existentes no aplicativo.

- *AlertDialog*: apresenta informações aos usuários através de uma caixa de diálogo. Comumente utilizado para perguntar ao cliente o que deseja fazer quando ele seleciona algum elemento.

- *ProgressDialog* e *ProgressBar*: utilizado quando uma aplicação necessita de um recurso que levará um certo tempo para executar, como por exemplo, fazer um *download*, pode ser feito uma animação informando ao usuário o progresso da operação.

- *SQLite*: é um banco de dados embarcado na plataforma *Android*, que armazena tabelas, *views*, índices, *triggers* em apenas um arquivo. Somente é possível acessá-lo pela aplicação a qual o criou e é excluído caso o aplicativo seja removido.

Além dos recursos acima citados, um outro *widget* que pode-se destacar é o *ExpandableListView*, que para Android (2015c), exibe os itens em forma de uma lista, similar ao *ListView*, o que diferencia-o é que ele mostra uma lista de dois níveis de rolagem vertical, em vez de abrir uma outra tela.

Para uma maior interação, as aplicações normalmente utilizam API's de terceiros, como o Google *Maps*, quando necessita encontrar alguma localização. Para Monteiro (2012) essa comunicação pode utilizar o REST, que envia requisições através da URL. Ao receber informações pedidas a um outro serviço, que pode estar no padrão XML ou JSON. O REST será detalhado mais adiante.

Outra ferramenta importante e muito utilizada do *Android* é a notificação. Segundo Phillips e Hardy (2013), quando uma aplicação está sendo executada em segundo plano e necessita comunicar-se com o usuário, o aplicativo cria uma notificação. Normalmente as notificações aparecem na barra superior, o qual pode ser acessado arrastando para baixo a partir da parte superior da tela. Assim que o usuário clica na notificação, ela cria uma *intent* abrindo a aplicação em questão.

Com a ideia de desenvolver um aplicativo para dispositivos móveis, a plataforma *Android* foi escolhida devido ao seu destaque no mercado e pela facilidade que apresenta aos usuários e desenvolvedores.

2.3 Android Studio

Umas das ferramentas mais utilizadas para o desenvolvimento em Android é o *Eclipse IDE*, contudo a Google criou um *software* especialmente para esse ambiente, chamado *Android Studio*. Segundo Gusmão (2014), *Android Studio* é uma IDE baseado no ItelliJ *Idea* e foi apresentado na conferência para desenvolvedores I/O de 2013.

De acordo com Hohensee (2013), o *Android Studio* tem um sistema de construção baseado em *Gradle*, que permite aplicar diferentes configurações no código quando há necessidade de criar mais de uma versão, como por exemplo, um *software* que terá uma versão gratuita e outra paga, melhorando a reutilização do código. Com o *Gradle* também é possível fazer os *downloads* de todas as dependências de uma forma automática sem a necessidade de importar bibliotecas manualmente.

Hohensee (2013) afirma que o *Android Studio* é um editor de código poderoso, pois tem como característica a edição inteligente, que ao digitar já completa as palavras reservadas do *Android* e fornece uma organização do código mais legível.

Segundo Android (2015d), a IDE tem suporte para a edição de interface, o que possibilita ao desenvolvedor arrastar os componentes que deseja. Ao testar o aplicativo, ela permite o monitoramento do consumo de memória e de processador por parte do utilitário.

Gusmão (2014) diz que a plataforma tem uma ótima integração com o *GitHub* e está disponível para *Windows*, *Mac* e *Linux*. Além disso os programadores terão disponíveis uma versão estável e mais três versões que serão em teste, chamadas de *Beta*, *Dev* e *Canary*.

Devido à fácil usabilidade e por ser a IDE oficial para o desenvolvimento *Android*, escolheu-se esse ambiente para a construção do aplicativo.

2.4 *Web Services*

Nos tempos atuais, com o grande fluxo de informação que percorre pelas redes da *internet*, é necessário um nível muito alto de integração entre as diversas plataformas, tecnologias e sistemas. Como uma provável solução para esse ponto, já existem as tecnologias de sistemas distribuídos. Porém essas tecnologias sofrem demasiadamente com o alto acoplamento de seus componentes e também com a grande dependência de uma plataforma para que possam funcionar. Com intuito de solucionar estes problemas e proporcionar alta transparência entre as várias plataformas, foram criados as tecnologias *web services*.

De acordo com Erl (2015, s.p):

> No ano de 2000, a W3C (*World Wide Web Consortium*) aceitou a submissão do *Simple Object Access Protocol* (SOAP). Este formato de mensagem baseado em XML estabeleceu uma estrutura de transmissão para comunicação entre aplicações (ou entre serviços) via HTTP. Sendo uma tecnologia não amarrada a fornecedor, o SOAP disponibilizou uma alternativa atrativa em relação aos protocolos proprietários tradicionais, tais como CORBA e DCOM.

Considera-se então a existência dos *web services* a partir daí. De acordo com Durães (2005), *Web Service* é um componente que tem por finalidade integrar serviços distintos. O que faz com que ele se torne melhor que seus concorrentes é a padronização do XML (*Extensible Markup Language*) para as trocas de informações. A aplicação consegue conversar com o servidor através do WSDL que é o documento que contém a estrutura do *web service*.

Segundo Coulouris et al. (2013), “Um serviço *Web* (*Web service*) fornece uma interface de serviço que permite aos clientes interagirem com servidores de uma maneira mais geral do que acontece com os navegadores *Web*”. Ainda de acordo com Coulouris et al. (2013), os clientes (que podem ser desde um navegador até mesmo outro sistema) acessam serviços *Web* fazendo uso de requisições e respostas formatadas em XML e sendo transmitidos pelo uso do protocolo HTTP. O uso dessas tecnologias tende a facilitar a comunicação entre as diversas plataformas, e atende de uma melhor forma que as tecnologias existentes. Porém, para que haja uma interação transparente e eficaz, entre as diversas plataformas, é necessário uma infraestrutura um pouco mais complexa para integrar todas essas tecnologias. Essa infraestrutura é composta pelas tecnologias já citadas e por outros componentes essenciais para disponibilização de serviços *web*, como mostra a Figura 2.

Os *web services* geralmente fazem uso do protocolo SOAP, para estruturar e encapsular as mensagens trocadas. De acordo com Coulouris et al. (2013, p.381), "o protocolo SOAP é projetado para permitir tanto interação cliente-servidor de forma assíncrona pela *Internet*".

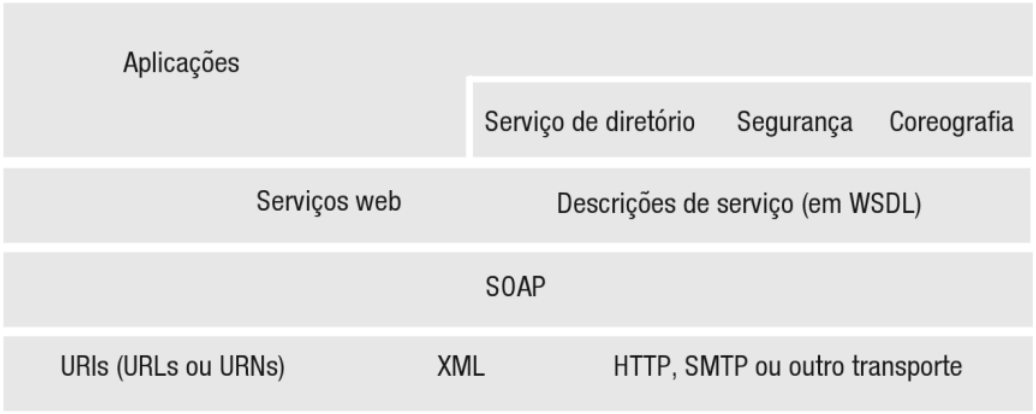

Figura 2 – Infraestrutura e componentes dos serviços *web*. **Fonte:**Coulouris et al. (2013)

Segundo Sampaio (2006, p.27), "o SOAP foi criado inicialmente, para possibilitar a invocação remota de métodos através da internet".

As mensagens SOAP possuem um elemento envelope, que de acordo com Saudate (2013, p.19), "é puramente um *container* para os elementos *Header* e *Body*". O elemento *header* transporta metadados relativos à requisição tais como autenticação, endereço de retorno da mensagem, etc. Já o elemento *body* carrega o corpo da requisição, que nada mais é do que o nome da operação e paramêtros referentes à mesma. É válido lembrar que todas requisições são trocadas usando SOAP, e usam o XML como formato oficial.

Os *web services* além de fornecerem uma padronização de comunicação entre as várias tecnologias existentes, proveem transparência na troca de informações. Isso contribui para que as novas aplicações consigam se comunicar com aplicações mais antigas ou aplicações contruídas sobre outras plataformas.

Além das tecnologias *web services* tradicionais, existem os *web services* REST que também disponibilizam serviços, porém não necessitam de encapsulamento de suas mensagens assim como os *web Services* SOAP. Este fato influencia diretamente na performance da aplicação, haja vista que não sendo necessário o encapsulamento da informação requisitada ao *web service*, somente é necessário o processamento e tráfego da informação que realmente importa. As caracteristícas do padrão REST serão abordadas na próxima seção.

2.4.1 REST

Segundo Saudate (2012), REST[2], desenvolvido por Roy Fielding na defesa de sua tese de doutorado. Segundo o próprio Fielding (2000) REST é um estilo que deriva dos vários

[2] REST -*Representational State Transfer* ou Transferência de Estado Representativo.

estilos arquitetônicos baseados em rede e que combinado com algumas restrições, fornecem uma interface simples e uniforme para fornecimento de serviços[3].

Rubbo (2015) afirma que os dados e funcionalidades de um sistema são considerados recursos e podem ser acessados através das URI's *(Universal Resource Identifier)*, facilitando dessa forma a comunicação do servidor com o cliente. Um serviço contruído na arquitetura REST basea-se fortemente em recursos. Saudate (2012), explica ainda que os métodos do HTTP podem fazer modificações nos recursos, da seguinte forma:

- GET: para recuperar algum dado.
- POST: para criar algum dado.
- PUT: para alterar algum dado.
- DELETE: para excluir algum dado.

Como o próprio Fielding (2000) também foi um dos criadores de um dos protocolos mais usados na web, o HTTP, pode-se dizer que o REST foi concebido para rodar sobre esse protocolo com a adição de mais algumas características que segundo Saudate (2013), foram responsáveis pelo sucesso da web:

- URLs bem definidas para recursos;
- Utilização dos métodos HTTP de acordo com seus propósitos;
- Utilização de *media types* efetiva;
- Utilização de *headers* HTTP de maneira efetiva;
- Utilização de códigos de *status* HTTP;

Segundo Godinho (2009), não há um padrão de formato para as trocas de informações, mas as que mais são utilizadas é o XML[4] e o JSON[5]. O REST é o mais indicado para aplicações em dispositivos móveis, devido a agilidade que proporciona na comunicação entre cliente e servidor.

[3] Tradução e resumo de informações de responsabilidade dos autores da pesquisa.
[4] XML - *Extensible Markup Language*.
[5] JSON - *JavaScript Object Notation*.

2.5 *Apache Tomcat*

De acordo com Tomcat (2015), *Apache Tomcat* é uma implementação de código aberto das especificações *Java Servlet* e *JavaServer Pages*. O *Apache Tomcat* é um *Servlet Container*, que disponibiliza serviços através de requisições e respostas. Caelum (2015b) afirma que ele é utilizado para aplicações que necessitam apenas da parte *Web* do Java EE[6].

Segundo Tomcat (2015), o projeto desse *software* começou com a *Sun Microsystems*, que em 1999 doou a base do código para *Apache Software Foundation*, e então seria lançada a versão 3.0.

Conforme Devmedia (2015), para o desenvolvimento com *Tomcat* é necessária a utilização das seguintes técnologias:

- JAVA: é utilizado em toda parte lógica da aplicação.
- HTML: é utilizado na parte de interação com o usuário.
- XML: é utilizado para as configurações do *software*.

Desta forma, o cliente envia uma requisição através do seu navegador, o servidor por sua vez a recebe, executa o *servlet* e devolve a resposta ao usuário.

2.6 PostgreSQL

Para Milani (2008), todas as aplicações que armazenam informações para o seu uso posterior devem estar integradas a um banco de dados, seja armazenando em arquivos de textos ou em tabelas. Por isso, o *PostgreSql* tem por finalidade armazenar e administrar os dados em uma solução de informática.

Postgresql (2015a, s.p) define que "o *Postgresql* é um SGBD (Sistema Gerenciador de Banco de Dados) objeto-relacional de código aberto, com mais de 15 anos de desenvolvimento. É extremamente robusto e confiável, além de ser extremamente flexível e rico em recursos."

Conforme afirma Milani (2008), o PostgreSql é um SGDB[7] de código aberto originado na Universidade de *Berkeley*, na Califórnia (EUA) no ano de 1986, pelo projeto *Postgres* desenvolvido por uma equipe sob liderança do professor Michael Stonebraker. Ele possui os

[6] EE - Sigla para enterprise edition
[7] SGDB - Sistema Gerenciador de Banco de Dados

principais recursos dos bancos de dados pagos e está disponível para os sistemas operacionais *Windows*, *Linux* e *Mac*. Atualmente existem bibliotecas e *drivers* para um grande número de linguagens de programação, entre as quais podem ser citadas: C/C++, PHP, *Java*, ASP, *Python* etc.

De acordo com Postgresql (2015b), existem sistemas com o *PostgreSql* que gerenciam até quatro *terabytes* de dados. Seu banco não possui um tamanho máximo e nem um número máximo de linhas por tabela. Contudo, uma tabela pode chegar a ter um tamanho de trinta e dois *terabytes* e cada campo a um *gigabyte* de informação.

Segundo Milani (2008), são características do *PostgreSql*:

- Suporte a ACID (Atomicidade, Consistência, Isolamento e Durabilidade).
- Replicação de dados entre servidores.
- *Cluster*.
- *Multithreads*.
- Segurança SSL[8] e criptografia.

É através do *Postgresql* que o *web service* armazenará e posteriormente retornará os dados dos discentes para o aplicativo *Andorid*.

2.7 UML

De acordo com Booch, Rumbaugh e Jacobson (2012) "A UML (*Unified Modeling Language*) é uma linguagem-padrão para a elaboração da estrutura de projetos de *software*". Na decada de 80 seguindo o surgimento e a evolução das linguagens de programação orientadas a objetos, foram surgindo linguagens de modelagens orientadas a objetos, como um modo alternativo de análise e projeto de *software* usadas na época. De acordo com Guedes (2011, p.19):

> A UML surgiu da união de três métodos de modelagem: o método de Booch, o método OMT (*Object Modeling Technique*) de Jacobson, e o método OOSE (*Object-Oriented Software Engineering*) de Rumbaugh. Estes eram, até meados da década de 1990, os métodos de modelagem orientada a objetos mais populares entre os profissionais da área de desenvolvimento de *software*. A união desses métodos contou com o amplo apoio da *Rational Software*, que a incentivou e financiou.

[8] SSL -*Secure Socket Layer*

Segundo Booch, Rumbaugh e Jacobson (2012, p.13) "A UML é independente de processo, apesar de ser perfeitamente utilizada em processo orientado a casos de usos, centrado na arquitetura, interativo e incremental". A linguagem de modelagem UML, além de fornecer um vocabulário próprio, também provê uma série de diagramas que tem inúmeras finalidades diferentes. Tais finalidades e suas subdivisões estão representados na Figura 3.

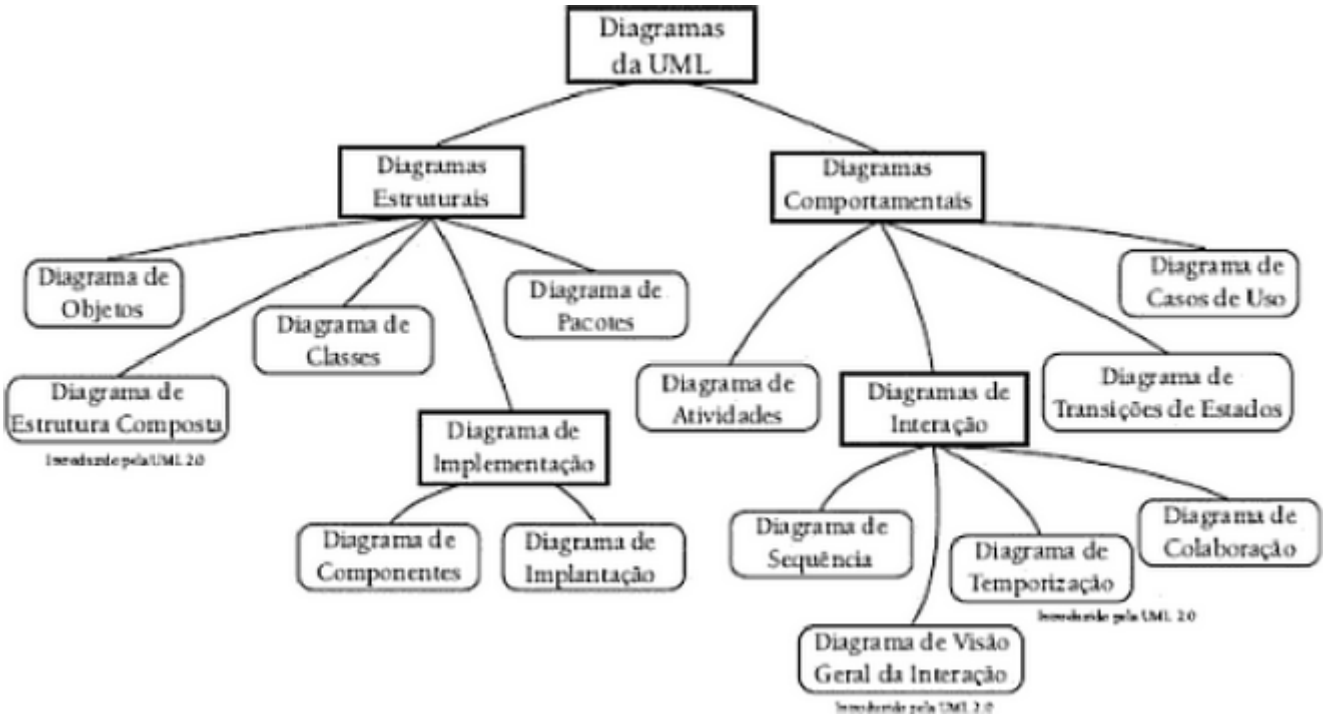

Figura 3 – Diagramas definidos pela UML. **Fonte:**Bezerra (2015)

A linguagem de modelagem UML não é processo rígido e permite uma adequação de acordo com a situação do projeto em que é aplicada. Por permitir essa flexibilidade e prover suporte adequado para determinados casos de um projeto, será utilizada a linguagem de modelagem UML para o desenvolvimento desta pesquisa.

2.8 Google Cloud Messaging

Para que os graduandos sejam notificados quando houver alguma mudança no portal do aluno, será utilizada uma API oferecida pela *Google* denominada *Google Cloud Messaging* ou simplesmente GCM[9], um recurso que tem por objetivo notificar as aplicações Android. Segundo Leal (2014), ele permite que aplicações servidoras possam enviar pequenas mensagens de até 4 KB[10] para os aplicativos móveis, sem que este necessite estar em execução. Ainda de acordo com Leal (2014) para o bom funcionamento do recurso apresentado, são necessários os seguintes componentes:

- *Sender ID*[11]: é o identificador do projeto. Será utilizado pelo servidores da *Google* para identificar a aplicação que envia a mensagem.
- *Application ID*: é o identificador da aplicação Android. O identificador é o nome do pacote do projeto que consta no *AndroidManifest.xml*.
- *Registration ID*: é o identificador gerado pelo servidor GCM quando aplicação Android se conecta a ele. Este deve ser enviado também à aplicação servidora.
- *Sender Auth Token*: é uma chave que é incluída no cabeçalho quando a mensagem é enviada da aplicação servidora para o GCM. Essa chave serve para que a API da *Google* possa enviar as mensagens para o dispositivo correto.

De acordo com os componentes acima citados, quando uma aplicação servidora enviar uma mensagem para o aplicativo Android, na verdade está enviando para o servidor GCM que será encarregado de enviar a mensagem para a aplicação *mobile*.

[9] *Google Cloud Messaging*
[10] *KB - Kilobytes*
[11] Identity

2.9 *Jersey*

Atualmente um padrão para desenvolvimento de serviços *web* vem sendo bastante adotado, trata-se do padrão arquitetural REST. De acordo com Saudate (2012), a linguagem *Java* possui uma especificação própria para desenvolvimento de serviços REST desde de setembro de 2008, que é a JSR311, ou como é popularmente chamado JAX-RS. Esta especificação provê um conjunto de API's simples, para facilitar o desenvolvimento de serviços *web*. De acordo com Oracle (2015b) "JAX-RS é uma API da linguagem de programação *Java* projetada para tornar mais fácil desenvolver aplicações que usam a arquitetura REST"[12]. Através desta especificação torna-se mais facíl e ágil a contrução de serviços *web* baseados em REST.

Como JAX-RS é apenas uma especificação, ela necessita então de uma implementação. Uma das implementações desta especificação é o *framework Jersey*. Segundo Oracle (2015c) "*Jersey*, a implementação de referência de JAX-RS, implementa suporte para as anotações definidas no JSR 311, tornando mais fácil para os desenvolvedores a construir serviços *Web RESTful* usando a linguagem de programação *Java*"[13]. Além das anotações que facilitam seu uso, *Jersey* pode prover serviços com uma infinidade de tipos de mídias, tais como XML e JSON entre outros.

O *framework Jersey* tem amplo suporte para os varios métodos HTTP. Fazendo uso dele pode-se facilmente implementar recursos REST. Além disso *Jersey* pode rodar tanto em servidores que implementem a especificação *Servlet* ou não. Este *framework* será usado para contruir o que seria a parte responsável por prover os serviços para o aplicativo *Android*.

2.10 *Hibernate*

Com a evolução e popularização da linguagem *Java*, e com o seu uso cada vez maior em ambientes corporativos, percebeu-se que, perdia-se muito tempo com a confecção de queries SQL[14] usadas nas consultas em bancos de dados relacionais e com a construção do código JDBC[15] que era responsável por trabalhar com estas consultas. Além disso era notório que, mesmo a linguagem SQL sendo padronizada, ela apresentava diferenças significativas entre os diversos bancos de dados existentes. Isso fazia com que a implementação de um *software* ficasse

[12] Tradução e resumo de informações de responsabilidade dos autores da pesquisa.
[13] Tradução e resumo de informações de responsabilidade dos autores da pesquisa.
[14] *SQL - Structured Query Language*
[15] JDBC - *Java Database Connectivity*

amarrada em um banco de dados específico e era extremamente custosa uma mudança poterior. Além disso havia o problema de lidar diretamente com dois paradigmas um pouco diferentes: o orientado a objeto e o relacional. Com o intuito de resolver esses problemas é que surgiram os *frameworks* para ORM[16] tais como *Hibernate*, *EclipseLink*, *Apache OpenJPA* entre outros.

Conforme surgiam novas alternativas e implementações para sanar esses problemas, surgia um novo problema: a falta de padronização entre os *frameworks* de ORM. Para resolver esse problema foi criada o JPA[17] que de acordo com Keith e Schincariol (2009, p.12) "nasceu do reconhecimento das demandas dos profissionais e as existentes soluções proprietárias que eles estavam usando para resolver os seus problemas"[18].

A especificação JPA foi concebida sendo a terceira parte da especificação EJB[19], e deveria atender ao propósitos de persistêcia de dados desta especificação. De acordo com Keith e Schincariol (2009, p.12) JPA é um *framework* leve baseado em POJO's,[20] para persistência de dados em *Java*, e que embora o mapeamento objeto relacional seja seu principal componente, ele ainda oferece soluções de arquitetura para aplicações corporativas escaláveis[21].

O *framework Hibernate* é uma das implementações da especificação JPA. De acordo com Sourceforge (2015) o *Hibernate* é uma ferramenta de mapeamento relacional, muito popular entre aplicações *Java* e implementa a *Java Persistence API*. Foi criado por uma comunidade de desenvolvedores, do mundo todo, que eram liderados por Gavin King. De acordo com Jboss (2015) "*Hibernate* cuida do mapeamento de classes *Java* para tabelas de banco de dados, e de tipos de dados *Java* para tipos de dados SQL".

O *Hibernate* está bastante difundido na comunidade de desenvolvedores *Java* ao redor do mundo, pelo fato de ser simples de usar, e por evitar esforços desnecessários na parte de infraestrutura das aplicações onde é usado, mantendo assim o foco na lógica de negócio. A pricipais vantagens do uso do *Hibernate* segundo Sourceforge (2015) são:

- Provedor JPA: além de sua API nativa , o *hibernate* também é uma implentação da especificação JPA, podendo assim ser facilmente usado em qualquer ambiente de apoio JPA.
- Persistência idiomática: permite que sejam construídas classes persistentes orientadas a objetos, e que suportem herança e polimorfismo entre outras estratégias, sem a necessidade da contrução de estruturas especiais para tal fim.

16 ORM - Object-relational Mapping
17 JPA - *Java Persistence API*
18 Tradução de responsabilidade dos autores da pesquisa.
19 EJB - *Enterprise Java Bean*
20 POJO - *Plain Old Java Object*
21 Tradução e resumo de informações de responsabilidade dos autores da pesquisa.

- Performance e suporte: permite que sejam usadas várias estratégias de de inicialização. Além disso não necessita de tabelas especiais no banco de dados. Mostra-se vantajoso também por gerar a maior parte do SQL necessário e evitar esforço desnecessário por parte do desenvolvedor, além de ser mais rápido que o JDBC puro.

- Escalável: o *Hibernate* foi projetado para trabalhar em *clusters* de servidores de aplicações e oferecer uma estrutura muito escalável, que se comporta bem tanto com um número muito baixo de usuários até números muitos elevados de usuários.

- Confiável: sua confiabilidade e estabilidade são comprovadas pelo seu grande uso e aceitação atualmente.

- Extensível: Hibernate é altamente configurável e extensível[22].

O *Hibernate* será usado nesta pesquisa com o intuito de fazer a gerência dos dados coletados e que serão providos para o aplicativo *Android* através do *webservice*, em conjunto com o banco de dados.

[22] Tradução e resumo de informações de responsabilidade dos autores da pesquisa.

3 QUADRO METODOLÓGICO

Nesse capítulo serão apresentados os métodos adotados para se realizar esta pesquisa, tais como tipo de pesquisa, contexto, procedimentos, entre outros.

3.1 Procedimentos e Resultados

3.1.1 *Web service*

Nesse sessão serão descritos os procedimentos realizados para o desenvolvimento do *webservice* responsável por prover os dados necessários ao aplicativo. Além disso serão descritas as configuração necessárias para a montagem do ambiente de desenvolvimento e sua posterior implantação.

3.1.1.1 Montagem do Ambiente de Desenvolvimento

No que diz respeito à contrução do *webservice*, foi necessária a instalação e configuração de um ambiente de desenvolvimento compatível com as necessidades apresentadas pelo *software*.

A princípio foi instalado o *Servlet Container Apache Tomcat* em sua versão de número 7. Esse *Servlet Container* foi instalado pois implementa a API da especificação *Servlets* 3.0 do *Java*. Isso era necessário pelo fato que o *framework Jersey* usa *servlets* para disponibilizar serviços REST. Além disso o *Apache Tomcat* foi escolhido, para que o *WebService* pudesse fornecer os serviços necessários para o consumo, na arquitetura REST, que sugere o uso do protocolo HTTP[1] para troca de mensagens, pois além da funcionalidade com *Servlets*, o *Apache Tomcat* também é um servidor HTTP.

O *Apache Tomcat* foi instalado, por meio do *download* de um arquivo `.zip` no site oficial do mesmo. A instalação consiste apenas em extrair os dados do arquivo em uma pasta da preferência do desenvolvedor. Esta abordagem permitiu a integração do *Apache Tomcat*

[1] HTTP - Hypertext Transfer Protocol

com o IDE[2] *Eclipse*, além disso foi possível controlar e monitorar, o servidor de aplicações através da IDE. Além da configuração necessária para integrar o servidor à IDE, nem uma outra configuração foi necessária.

Para armazenar os dados gerados e/ou recebidos, foi necessário fazer a intalação do Sistema Gerenciador de Banco de Dados(SGBD) *PostGreSql* na sua versão de número 9.4. Como está sendo usado um sistema operacional baseado em GNU/Linux como ambiente de desenvolvimento, o mesmo foi instalado através do gerenciador de pacotes da distribuição. Além disso foi necessário criar um usuário no SGDB que tivesse permissão suficiente apenas para fazer as operações referentes ao banco de daodos do *web service*, evitando assim a necessidade de se trabalhar diretamente com um usuário master do SGBD.

Eclipse ide

Maven plugin

3.1.1.2 Desenvolvimento

Essas classes foram criadas fazendo uso de anotações próprias do *Hibernate*, que é um *framework* que implementa a especificação JPA[3]. Essas classes fazem parte dos mecanismos de persistêcia de dados e são simplesmente t ou seja, objetos simples que contêm somente atributos privados e os métodos *getters* e *setters* que servem apenas para encapsular estes atributos. Uma das classes criadas, foi a classe `Aluno.java` que representa a tabela `alunos` no banco de dados e está representada na Figura 4.

[2] IDE - *Integrated Development Environment*

[3] JPA - *Java Persistense API*

```java
@Entity
@Table(name = "alunos")
public class Aluno {

    @Id
    @SequenceGenerator(
            name = "id_aluno",
            sequenceName = "seq_id_aluno",
            allocationSize = 1
            )
    @GeneratedValue(
            generator = "id_aluno",
            strategy = GenerationType.IDENTITY)
    @Column(name = "id_aluno", nullable = false)
    private Long idAluno;

    @Column(name = "id_externo", nullable = false)
    private Long idDbExterno;

    @Column(length = 100, nullable = false)
    private String nome;

    @Column(nullable = false)
    private Integer periodo;

    @Column(length = 100, nullable = false)
    private String email;

    @OneToMany(mappedBy = "aluno")
    private List<Evento> eventos;

    @OneToMany(mappedBy = "aluno")
    private List<Disciplina> disciplinas;

    /**
     *
     * GETTERS E SETTERS
     *
     */

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result +
                ((idAluno == null) ? 0 : idAluno.hashCode())
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Aluno other = (Aluno) obj;
        if (idAluno == null) {
            if (other.idAluno != null)
                return false;
        } else if (!idAluno.equals(other.idAluno))
            return false;
        return true;
    }

}
```

Figura 4 – Classe `Aluno`. **Fonte:**Elaborado pelos autores.

Foram criadas outras classes *Java* com a mesma finalidade da anterior, porém com pequenas diferenças no que diz respeito à atributos, metodos e anotações. Estas classes representam, de maneira individual, as tabelas no banco de dados. Certos atributos dessas classes têm por finalidade representar as colunas de cada tabela. Já os atributos que armazenam instâncias de outras classes ou até mesmo conjuntos (coleções) de instâncias representam os relaciona-

mentos entre as tabelas. E por fim, para cada classe que representa uma entidade, foi necessário implementar os métodos `hashCode` e `equals`, para que estas pudessem facilmente ser comparadas e diferenciadas em relação aos seus valores, haja visto que cada instância destas classes representa um registro no banco de dados.

Em seguida à criação das entidades, foi necessário configurar o arquivo `persistence.xml` que fica dentro do *classpath* do projeto *Java* ou seja, dentro da mesma pasta onde estão contidos pacotes do projeto. Este arquivo é extremamente importante, pois é nele que estão todas as configurações relativas à conexão com o banco de dados, configurações referentes ao Dialeto SQL que vai ser usado para as consultas e configurações referentes ao *persistence unit* que é o conjunto de classes mapeadas para o banco de dados. O arquivo `persistence.xml` está exposto no código 5.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="WsAppUnivas" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <properties>
            <property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost:5432/wsappunivas" />
            <property name="javax.persistence.jdbc.user" value="postgres" />
            <property name="javax.persistence.jdbc.password" value="password" />

            <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver" />
            <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="hibernate.temp.use_jdbc_metadata_defaults" value="false"></property>
            <property name="hibernate.hbm2ddl.auto" value="create" />
        </properties>
    </persistence-unit>
</persistence>
```

Figura 5 – Arquivo `persistence.xml`. **Fonte:**Elaborado pelos autores.

Em seguida à confecção do `persistence.xml` foi criada a classe `JpaUtil` que está representada na Figura 6. Esta classe é responsável por criar uma `EntityManagerFactory` que é uma fábrica de instâncias de `EntityManager` que nada mais é que um *persistence unit* ou unidade de persistência. Essa classe tem a responsabilidade de prover um modo de comunicação entre a aplicação e o banco de dados. No entanto a classe `JpaUtil` cria uma única instância de `EntityManagerFactory`, que é responsável por disponibilizar e gerenciar as instâncias de `EntityManager` de acordo com a necessidade da aplicação.

```java
public class JpaUtil {
    private static EntityManagerFactory factory;

    static {
        factory = Persistence.createEntityManagerFactory("WsAppUnivas");
    }

    public static EntityManager getEntityManager() {
        return factory.createEntityManager();
    }

    public static void close() {
        factory.close();
    }

}
```

Figura 6 – Classe `JpaUtil`. **Fonte:**Elaborado pelos autores.

Em seguida à construção das classes que fazem a parte da persistência de dados, foi desenvolvido a parte de disponibilização de serviços *RESTful*, fazendo uso do *framework Jersey*. Com isso pode-se construir a classe que representa o primeiro serviço do *webservice*, que é a classe `Alunos`. Essa classe representa um contexto REST, e portanto, dispõe de alguns recursos. Esses recursos fazem a recuperação e a transmissão dos dados do *webservice* para o aplicativo *Android*. Essa classe e seus respectivos métodos estão representada na Figura 7.

```java
@Path("/alunos")
public class AlunosService {

    /*
     * Busca um Aluno e a suas informações e eventos
     */
    @GET
    @Path(" /{ $cod } ")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Alunos getAlunoById(@PathParam("cod") Long idAluno) {
        Alunos alunos = new Alunos();
        AlunoCtrl ctrl = new AlunoCtrl();
        alunos.setAlunos(ctrl.getById(idAluno));

        return alunos;
    }
    /*
     * Busca os eventos de um Aluno pelo seu id
     */
    @GET
    @Path("/eventos/{ $cod }")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Eventos getAll(@PathParam("cod") Long idAluno) {
        Eventos eventos = new Eventos();
        EventoCtrl ctrl = new EventoCtrl();
        eventos.setEventos(ctrl.getById(idAluno));
        return eventos;
    }
}
```

Figura 7 – Classe `AlunosService`. **Fonte:**Elaborado pelos autores.

O *webservice* pode fazer a busca de alunos pelo `id` passado ou retornar uma coleção de eventos vinculados a um alunos, dependendo do recurso acessado. Os tipos de dados que o *webservice* consome e retorna é o JSON[4]. Não foi necessário fazer nenhuma implementação

[4] JSON - *Javascript Object Notation*

adicional relativa a este formato, pois o próprio *framework Jersey* faz o tratamento e a conversão dos tipos de entrada e saída de dados. No caso do saída de dados, faz a conversão de objetos *Java* para JSON. E no caso de entrada tranforma um JSON em objeto *Java* já conhecido pelo *webservice*. Com isso concluiu-se o desenvolvimento do *webservice* que fornece os dados para o aplicativo.

Para que fosse possível transmitir dados para o aplicativo, era necessário receber as informações do sistema acadêmico da referida instituição, haja vista que o *web service* é independente do mesmo. Para esse propósito é necessário contruir um módulo que faça a importação dos dados necessários para a base de dados do *web service*. Este por sua vez terá a responsabilidade de fazer a importação dos dados periodicamente, e ainda tratar os tipos de dados recebidos para tipos aplicáveis ao banco de dados local. Além disso é preciso notificar o módulo responsável por invocar o serviço *Google Cloud Messaging* para que os dispositivos dos alunos aos quais houveram atualizações nos dados, fossem notificados e fizessem acesso ao *web service* para solicitar esses dados atualizados.

Os procedimentos acima citados foram os passos até agora realizados com o propósito de se alcançar os resultados esperados para essa pesquisa.

3.1.1.3 Implantação

REFERÊNCIAS

ANDROID. : **A história do Android.** 2015. Disponível em: <https://www.android.com/history/>. Acesso em: 25 de Fevereiro de 2015.

ANDROID. : **Creating a Navigation Drawer**. 2015. Disponível em: <https://developer.android.com/training/implementing-navigation/nav-drawer.html>. Acesso em: 28 de julho de 2015.

ANDROID. : **ExpandableListView.** 2015. Disponível em: <http://developer.android.com/reference/android/widget/ExpandableListView.html>. Acesso em: 24 Agosto de 2015.

ANDROID. : **Android Studio Overview.** 2015. Disponível em: <http://developer.android.com/tools/studio/index.html>. Acesso em: 12 de Março de 2015.

BEZERRA, E. : **Princípios De Análise E Projeto De Sistemas Com Uml**. 3ª. ed. São Paulo: Elsevier, 2015.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. : **UML:** guia do usuário. 2ª. ed. Rio De Janeiro: CAMPUS, 2012.

CAELUM. : **Apostila Java e Orientação a Objetos.** 2015. Disponível em: <http://www.caelum.com.br/apostila-java-orientacao-objetos/o-que-e-java/#2-3-maquina-virtual>. Acesso em: 18 de Setembro de 2015.

CAELUM. : **Java para Desenvolvimento Web.** 2015. Disponível em: <https://www.caelum.com.br/apostila-java-web/o-que-e-java-ee/#3-4-servlet-container>. Acesso em: 15 de Fevereiro de 2015.

COULOURIS, G. et al. : **Sistemas Distribuídos** conceitos e projeto. 5ª. ed. Porto Alegre: Bookman Editora, 2013.

DEITEL, H.; DEITEL, P. : **Java como Programar**. São Paulo: Pearson Prentice Hall, 2010.

DEVMEDIA. : **Conheça o Apache Tomcat.** 2015. Disponível em: <http://www.devmedia.com.br/conheca-o-apache-tomcat/4546>. Acesso em: 08 de Março de 2015.

DURãES, R. : **Web Services para iniciantes**. 2005. Disponível em: <http://imasters.com.br/artigo/3561/web-services/web-services-para-iniciantes/>. Acesso em: 10 de Março de 2015.

ERL, T. : **Introdução às tecnologias Web Services:** soa, soap, wsdl e uddi. 2015. Disponível em: <http://www.devmedia.com.br/introducao-as-tecnologias-web-services-soa-soap-wsdl-e-uddi-parte1/2873>. Acesso em: 26 de Abril de 2015.

FIELDING, R. T. : **Architectural Styles and the Design of Network-based Software Architectures**. Tese (Doutorado) — University of California, 2000.

GODINHO, R. : **Criando serviços REST com WCF.** 2009. Disponível em: <https://msdn.microsoft.com/pt-br/library/dd941696.aspx>. Acesso em: 01 de Março de 2015.

GUEDES, G. T. A. : **UML 2 :** uma abordagem prática. 2ª. ed. São Paulo: Novatec, 2011.

GUSMãO, G. : **Google lança versão 1.0 do IDE de código aberto Android Studio.** 2014. Disponível em: <http://info.abril.com.br/noticias/it-solutions/2014/12/google-lanca-versao-1-0-do-ide-de-codigo-aberto-android-studio.shtml>. Acesso em: 03 de Março de 2015.

HOHENSEE, B. : **Getting Started with Android Studio.** Gothenburg: [s.n.], 2013.

JBOSS. : **Hibernate Getting Started Guide**. 2015. Disponível em: <http://docs.jboss.org/hibernate/orm/5.0/quickstart/html/>. Acesso em: 20 de Setembro de 2015.

K19. : **Desenvolvimento mobile com Android**. 2012.

KEITH, M.; SCHINCARIOL, M. : **Pro JPA 2:** *Mastering the Java Persistence API*. New York: Apress, 2009.

KRAZIT, T. : **Google's Rubin:** android 'a revolution'. 2009. Disponível em: <http://www.cnet.com/news/googles-rubin-android-a-revolution/>. Acesso em: 20 de Fevereiro de 2015.

LEAL, N. : **Dominando o Android:** do básico ao avançado. 1ª. ed. São Paulo: Novatec, 2014.

LECHETA, R. R. : **Google Android:** aprenda a criar aplicações para dispositivos móveis com android sdk. 2ª. ed. São Paulo: Novatec, 2010.

LECHETA, R. R. : **Google Android:** aprenda a criar aplicações para dispositivos móveis com o android sdk. 3ª. ed. São Paulo: Novatec, 2013.

MENDES, E. V. : **Um aplicativo para Android visando proporcionar maior interação de uma banda musical e seus seguidores.** Pato Branco: Universidade Tecnológica Federal do Paraná, 2011.

MILANI, A. : **PostgreSQL.** São Paulo: Novatec, 2008.

MONTEIRO, J. B. : **Google Android:** crie aplicações para celulares e tablets. São Paulo: Casa do Código, 2012.

OGLIO, M. D. : **Aplicativo Android para o ambiente UNIVATES Virtual.** Lajeado: Univates, 2013.

ORACLE. : **O que é a Tecnologia Java e porque preciso dela?** 2015. Disponível em: <https://www.java.com/pt_BR/download/faq/whatis_java.xml>. Acesso em: 17 de Setembro de 2015.

ORACLE. : ***the java ee 6 tutorial*:***Creating a RESTful Root Resource Class*. 2015. Disponível em: <http://docs.oracle.com/javaee/6/tutorial/doc/giepu.html>. Acesso em: 20 de Setembro de 2015.

ORACLE. : ***the java ee 6 tutorial*:***Building RESTful Web Services with JAX-RS*. 2015. Disponível em: <http://docs.oracle.com/javaee/6/tutorial/doc/giepu.html>. Acesso em: 20 de Setembro de 2015.

PHILLIPS, B.; HARDY, B. : **Android Programming:** the big nerd ranch guide. Atlânta: Big Nerd Ranch, 2013.

POSTGRESQL. : **O que é PostgreSQL?** 2015. Disponível em: <https://wiki.postgresql.org/wiki/Introdu%C3%A7%C3%A3o_e_Hist%C3%B3rico>. Acesso em: 11 de de 2015.

POSTGRESQL. : **Sobre o PostgreSQL.** 2015. Disponível em: <http://www.postgresql.org.br/old/sobre>. Acesso em: 11 de de 2015.

RUBBO, F. : **Construindo RESTful Web Services com JAX-RS 2.0.** 2015. Disponível em: <http://www.devmedia.com.br/construindo-restful-web-services-com-jax-rs-2-0/29468>. Acesso em: 03 de Março de 2015.

SAMPAIO, C. : **SOA e Web Services em Java**. 1ª. ed. Rio de Janeiro: Brasport, 2006.

SAUDATE, A. : **REST:** construa api's inteligentes de maneira simples. São Paulo: Casa do Código, 2012.

SAUDATE, A. : **SOA aplicado:** integrando com web serviços e além. 1ª. ed. São Paulo: Casa do Código, 2013.

SOURCEFORGE. : **Hibernate**. 2015. Disponível em: <http://sourceforge.net/projects/hibernate/>. Acesso em: 20 de Setembro de 2015.

TOMCAT, A. : **The Tomcat Story.** 2015. Disponível em: <http://tomcat.apache.org/heritage.html>. Acesso em: 08 de Março de 2015.