UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ - UTFPR

CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

MARCIO SANDOVAL TOMÉ

INTEGRAÇÃO DE SISTEMAS *DESKTOP* COM DISPOSITIVOS MÓVEIS UTILIZANDO *WEB SERVICES*

TRABALHO DE DIPLOMAÇÃO

MEDIANEIRA

2011

MARCIO SANDOVAL TOMÉ

INTEGRAÇÃO DE SISTEMAS *DESKTOP* COM DISPOSITIVOS MÓVEIS UTILIZANDO *WEB SERVICES*

Trabalho de Diplomação apresentado à disciplina de Trabalho de Diplomação, do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas – CSTADS – da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Prof. Me. Fernando Schütz.

MEDIANEIRA

2011

Ministério da Educação

Universidade Tecnológica Federal do Paraná

Diretoria de Graduação e Educação Profissional

Coordenação do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

TERMO DE APROVAÇÃO

INTEGRAÇÃO DE SISTEMAS *DESKTOP* COM DISPOSITIVOS MÓVEIS UTILIZANDO WEB SERVICES

Por

Marcio Sandoval Tomé

Este Trabalho de Diplomação (TD) foi apresentado às 09:20 h do dia 24 de novembro de 2011 como requisito parcial para a obtenção do título de Tecnólogo no Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, da Universidade Tecnológica Federal do Paraná, Campus Medianeira. O candidato foi argüido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Me. Fernando Schütz.
UTFPR – Campus Medianeira
(Orientador)

Me. Pedro Luiz de Paula Filho
UTFPR – Campus Medianeira
(Convidado)

Dr. Hermes Irineu Del Monego
UTFPR – Campus Medianeira
(Convidado)

Prof. M.Eng. Juliano Rodrigo Lamb
UTFPR – Campus Medianeira
(Responsável pelas atividades de TCC)

“Não são as respostas que movem o mundo, mas sim as perguntas”.
Canal Futura

RESUMO

TOMÉ, Marcio Sandoval. Integração de Sistemas *Desktop* com Dispositivos Móveis utilizando *Web Services*. Trabalho de Conclusão do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas. Universidade Tecnológica Federal do Paraná. Medianeira, 2011.

O progresso tecnológico em qualquer tipo de negócio é essencial. O surgimento de novas tecnologias e a melhoria das já existentes, permitem o desenvolvimento de novos sistemas que contornam obstáculos técnicos e econômicos que normalmente impedem que essa evolução aconteça. Este projeto apresenta o desenvolvimento de um sistema, como estudo experimental, com baixo custo de implantação e manutenção, para acessar o banco de dados legado de um sistema *desktop*, utilizando *Web Services* e aplicativo cliente para dispositivos móveis com *Android*.

Palavras-chaves: *Web Services*, *Android*, Dispositivos Móveis.

ABSTRACT

TOMÉ, Marcio Sandoval. *Systems Integration with Desktop Mobile Web Services using. Work Completion Degree in Technology Analysis and Systems Development*. Universidade Tecnológica Federal do Paraná. Medianeira, 2011.

Technological progress in any business is essential, the emergence of new technologies and improvement of existing enable the development of new systems that circumvent technical and economic barriers that normally prevent these developments happen. This paper presents a study and development of a system as an experimental study, with low cost of deployment and maintenance, you can access the database from a legacy system using web services and desktop client application for mobile devices with Android.

Palavras-chaves: *Web Services, Android, Móbile Devices, desktop*.

LISTA DE SIGLAS

ADT	*Android Development Tools*
API	*Application Programming Interface*
DNS	*Domain Name Service*
EJB	*Enterprise Java Beans*
GPS	Sistema de Posicionamento Global
HTTP	*Hyper Text Transport Protocol*
IDE	*Integrated Development Environment*
IP	*Internet Protocol*
J2EE	*Java 2 Enterprise Edition*
JEE	*Java Enterprise Edition*
JAAS	*Java Autenthication and Authorization Service*
JAX-B	*Java Api for XML Binding*
JAX-RPC	*Java API for XML - based RPC*
JAX-WS	*Java Api for Xml – Java para Web Services XML*
JDBC	*Java Database Connectivity*
JDK	*Java Development Kit*
JMS	*Java Message Service*
JMX	*Java Management Extensions*
JNDI	*Java Naming and Directory Interface*
JPA	*Java Persistence API*
JPQL	*Java Persistence Query Language*
JRE	*Java Runtime Environment*
JSF	*Java Server Faces*
JSP	*Java Server Pages*
JTA	*Java Transaction API*
LAN	*Local Area Network*
ORM	*Object Relational Mapping*
PC	*Personal Compuiter*
PNBL	Plano Nacional de Banda Larga
REST	*REpresentational State Transfer*
SEI	*Service Endpoint Inteface*

SDK	*Software Development Kit*
SGDB	Sistema Gerenciador de Banco de Dados
SIB	*Service Implementation Bean*
SMS	*Short Message Service*
SOA	*Service Oriented Architecture*
SOAP	*Simple Object Access*
SQL	*Structured Query Language*
UML	*Unified Modeling Language*
URL	*Uniform Resource Locator*
UTFPR	Universidade Tecnológica Federal do Paraná
VPN	*Virtual Private Network* - Rede Privada Virtual
XML	*Extensible Markup Language*
WSDL	*Web Service Definition Language*

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Muitos empresários, ao pesquisar sistemas informatizados para automatizar os processos e gerir suas empresas, ainda optam por adquirir aplicativos *desktop* que não dependam da disponibilidade de conexão com a Internet para funcionar.

Um dos motivos desta opção é o fato de que os tempos de resposta não dependem da disponibilidade ou desempenho da conexão com a Internet. Ao fazer uma avaliação entre aplicativos *Desktop* e *Web*, Bonfandini (2011) fala do crescimento da utilização de sistemas *Web*, mas também fala sobre as alterações nas expectativas dos usuários devido aos tempos de resposta extremamente elevados, em comparação aos sistemas *desktop*.

Outro motivo é a vantagem econômica, pois geralmente estas empresas podem manter somente uma pequena infraestrutura de rede de computadores. O acesso a Internet é muitas vezes, não confiável, sem garantias de funcionamento 24 horas e com oscilação de desempenho. Para ter uma Internet confiável, as empresas necessitam contratar um serviço com conexões baseadas em tecnologias dedicadas, que nada mais é do que um serviço diferenciado para garantia de estabilidade. No entanto, segundo Olhar Digital (2011), alguns planos podem chegar a custar três mil reais por ‘apenas’ dois mega bits de velocidade, o que faz com que esta tecnologia fique totalmente fora do alcance das pequenas empresas.

Olhar Digital (2011) também apresenta, em uma de suas matérias, que a qualidade do serviço de Internet banda larga fornecido no Brasil é desproporcional ao cobrado pelas operadoras; e que isto se deve ao fato que as operadoras estão amparadas pela lei em fornecer somente 10% da velocidade contratada sem sofrer represálias.

Segundo Terra Tecnologia (2011), que referencia estudo realizado pela germânico-americana Nielsen Company, a velocidade da banda larga do Brasil é comparada a de outros oito países, porém destaca que o Brasil é o único que segue fora da curva de tendência: quase metade dos usuários (48%) tem acesso com velocidades médias, um terço em velocidades lentas (31%) e apenas 21% em velocidades rápidas.

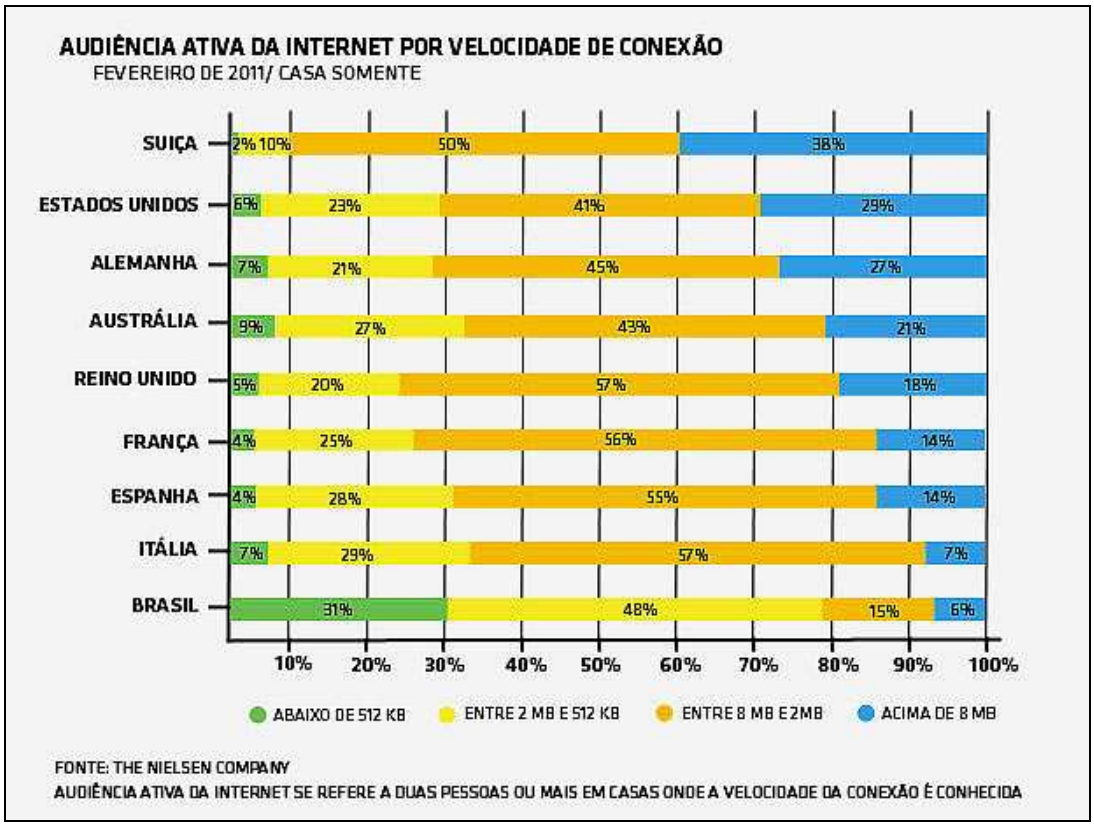


Figura 1 - Comparativo entre a velocidade da Internet banda larga do Brasil com a de outros oito países.
Fonte: TERRA (2011)

Esforços por parte do governo através de um Plano Nacional de Banda Larga (PNBL) tendem a mudar este panorama. Segundo o Ministério das Comunicações (2011) um dos princípios do PNBL é o estímulo ao setor privado, para que este invista na infraestrutura de banda larga, em regime de competição, cabendo ao Estado atuar de forma complementar, focalizando seus investimentos diretos, principalmente em acessos coletivos e em contextos de redução das desigualdades regionais e sociais.

Até que este objetivo seja alcançado pela sociedade pode-se concordar que um sistema *desktop* que não depende da disponibilidade da Internet se torna mais viável que um sistema *Web* quando a Internet fornecida não garantir a disponibilidade necessária. No entanto, existe a necessidade de acessar os dados deste sistema que não utiliza a Internet fora da rede local em que o mesmo foi instalado. E o que fazer neste momento? Investir em um sistema *Web*? Existem outras alternativas?

O investimento em um sistema *Web* não significa apenas um novo sistema, existe toda uma infraestrutura necessária para que este sistema funcione com segurança, desempenho e alta disponibilidade e que tem custo proporcional a estes requisitos. Esta

infraestrutura pode ser terceirizada, mas isso ainda não oferece a disponibilidade que um sistema *desktop* oferece, e nem sempre esta migração de um sistema *desktop* para *Web* vai estar economicamente ao alcance do pequeno empresário.

Em um cenário em que o empresário somente consulta o estoque atual, ou sua previsão de contas a receber e a pagar, a estrutura necessária e manutenção da mesma não exigem um investimento alto, já que a utilização do sistema irá acontecer sob pequena demanda. E um servidor que disponibilize um serviço de armazenamento de banco de dados para o sistema *desktop* e um *Web Service* que forneça, através de um endereço fixo, serviços para acessar informações do banco de dados, possibilitam a estes empresários o acesso às informações de sua empresa em tempo real.

Para atingir este objetivo, o presente trabalho propõe um sistema em *Java* que irá se conectar ao banco de dados utilizado pelo sistema *desktop*, e disponibilizará estas informações na Internet através de um *Web Service.* Também propõe-se o desenvolvimento de um aplicativo cliente, que funcione em celulares com o sistema operacional *Android*, para acessar estes dados *on-line*.

1.1. OBJETIVO GERAL

Desenvolver um sistema composto por um *Web Service* e uma aplicação cliente para dispositivos móveis com sistema operacional *Android*.

1.2. OBJETIVOS ESPECÍFICOS

Como objetivos específicos este trabalho apresenta:

- Construir um referencial teórico sobre tecnologias necessárias para o desenvolvimento de *Web Services* e aplicativos clientes para dispositivos móveis.
- Desenvolver a análise e o projeto de um *Web Service*.

- Desenvolver a análise e o projeto de um aplicativo para dispositivos móveis com *Android* que irá acessar os serviços disponibilizados pelo *Web Service*.
- Implementar o sistema de *Web* Service e o aplicativo cliente para dispositivos móveis com *Android*.
- Testar as aplicações desenvolvidas.

1.3. JUSTIFICATIVA

O projeto é viabilizado por oferecer uma alternativa para acessar os dados de um sistema *desktop* através da Internet sem necessidade de alto investimento em melhor infraestrutura. Geralmente estes sistemas *desktops* são utilizados há vários anos e uma mudança radical não seria viável, pois se a solução é funcional não há motivos para mudar, e principalmente, se o acesso externo aos dados destes sistemas ocorre aleatoriamente.

Celulares com sistema operacional *Android* estão cada vez mais acessíveis e oferecem os recursos necessários para o funcionamento de um aplicativo cliente para o *Web Service*.

Outro ponto importante é que os planos empresariais de banda larga, em boa parte dos fornecedores, já vem com um numero de IP fixo. “Para um cliente empresarial um IP fixo é importante, às vezes ele tem aplicações como servidores *Web* ou banco de dados que rodam na rede dele e que precisam ser sempre acessados pelo mesmo IP” (OLHAR DIGITAL, 2011). Quando não houver disponibilidade de um numero de IP fixo, será possível utilizar tecnologias que fornecem gratuitamente serviços de *DNS (Domain Name Service).*

1.4. ESTRUTURA DO TRABALHO

A estrutura deste Trabalho de Diplomação é composta por cinco capítulos.

O Capítulo um apresenta uma introdução sobre o assunto destacando aspectos de infraestrutura necessária para a migração de sistemas *desktop* que não funciona através de

uma conexão de Internet para sistemas *Web* e aspectos econômicos que inviabilizam uma migração de sistemas em pequenas empresas.

O Capítulo dois apresenta o referencial teórico sobre as tecnologias utilizadas para o desenvolvimento do *Web Service* e do aplicativo cliente para dispositivos móveis com sistema operacional *Android*.

O Capítulo três aborda os materiais e métodos utilizados no desenvolvimento do projeto.

O Capítulo quatro aborda o projeto e desenvolvimento de um *Web Service* que disponibiliza serviços de acesso ao banco de dados legado. E o projeto e desenvolvimento de um aplicativo para dispositivos móveis utilizando o *Android SDK* que consome os serviços disponibilizados pelo *Web Service*.

O Capítulo cinco apresenta as considerações finais sobre o projeto.

2 REFERENCIAL TEÓRICO

Neste capitulo será apresentado um referencial teórico sobre as tecnologias utilizadas para o desenvolvimento do projeto.

2.1. WEB SERVICES

Kopack (2003) define *Web Service* como um sistema que pode ser acessado remotamente, e semelhante à sites *Web* ele é identificado na rede por uma *URL* e fornece ao cliente uma resposta a uma requisição. A diferença está no fato de que, tanto a requisição quanto a resposta, são formadas por um arquivo *XML* formatado de acordo com a especificação do protocolo *SOAP*. E o principal beneficio do uso de arquivos *XML* é a interoperabilidade que ele proporciona e necessária para que sistemas, desenvolvidos em diferentes linguagens de programação e para diferentes plataformas, possam se comunicar.

Conforme definição de Kalin (2010), o *Web Service* trata-se de uma aplicação distribuída, tipicamente oferecida através de *HTTP* (*Hyper Text Transport Protocol*), em que seus componentes podem ser executados em dispositivos distintos.

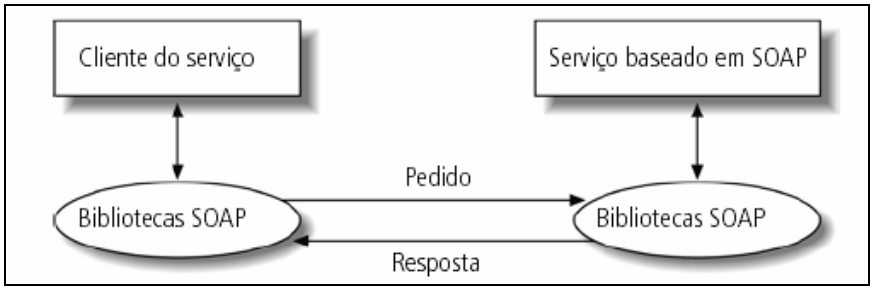


Figura 2 - Arquitetura de um típico *Web Service* baseado em *SOAP*.
Fonte: Kalin(2010)

Um *Web Service* pode ser implementado utilizando *JAX-WS* (*Java API for XML – Java* para *Web Services XML)* que esta disponível no *Java SE 7* (*Java Standard Edition 7*).

O *JAX-WS* suporta serviços em SOA e de estilo REST. A versão atual de *JAX-WS* é 2.x, a versão 1.x tem um rótulo diferente: *JAX-RPC*, sendo que *JAX-WS* estende as

capacidades de *JAX-RPC*. Nos próximos capítulos é falado sobre *Web Services* do estilo SOA utilizando o *JAX-WS*.

Kalin (2010) esclarece que, a forma de implementação de um *Web Service* poderia ser feita com uma única classe *Java*, mas, seguindo as melhores práticas, deve haver uma interface que declare os métodos chamada de SEI (*Service Endpoint Interface*), e uma implementação, que defina os métodos declarados na interface, chamada de SIB (*Service Implementation Bean*). Depois de compilado o *Web Service* está pronto para ser publicado em um Servidor de Aplicação *Java*.

Para testar o *Web Service* basta abrir e visualizar, através de um navegador, o documento WSDL (*Linguagem de Definição de Web Service*), "O navegador é aberto para uma *URL* que tem duas partes. A primeira parte é a *URL* publicada na aplicação *Java TimeServerPublisher*: http://127.0.0.1:9876/ts. Anexa a esta *URL* é a *string* de consulta ?wsdl. O resultado é http://127.0.0.1:9876/ts?wsdl" (KALIN, 2010).

Kalin (2010) apresentando das principais seções de um arquivo *WSDL*, explica que a seção *portType*, próxima ao topo do documento, agrupa as operações que o *Web Service* oferece, que são os métodos *Java* declarados na *SEI* e implementados no *SIB*. É como uma interface *Java* que apresenta as operações do *Web Service* de forma abstrata, sem demonstrar nenhum detalhe da implementação. Outra seção que merece interesse é a seção *service*, a *URL* é chamada de *service endpoint*, e informa onde os serviços podem ser acessados.

Vohra (2011) explica que um serviço da *Web* em *JAX-WS* consiste, basicamente, em uma classe *Java* com a declaração da anotação `javax.jws.WebService.` Esta classe não pode ser `abstract` ou `final`, e seus métodos que serão expostos como operações devem ser `public.`

A Figura 3 apresenta um exemplo de classe de implementação de serviços da *Web* chamada `HelloWSImpl`, comentada com a anotação `@WebService` que implementa a interface `HelloWS`. A classe de implementação contém um método `hello` comentado com a anotação `@WebMethod`, e que toma um parâmetro *String* com o nome, e retorna uma mensagem *Hello* que contém esse nome.

```java
package hello_webservice;

import javax.ejb.Remote;
import javax.ejb.Stateless;
import javax.jws.WebMethod;
import javax.jws.WebService;

@Stateless
@WebService(targetNamespace = "http://hello_webservice/")
@Remote(HelloWS.class)
public class HelloWSImpl implements HelloWS {
    @WebMethod(operationName = "hello")
    public String hello(String name) {
            // replace with your impl here
             return "Hello "+name +" Welcome to Web Services!";

    }
}
```

Figura 3 - Classe de implementação de serviços da Web HelloWSImpl.java

A interface de terminal em serviço `HelloWS` é apresentado na Figura 4.

```java
package hello_webservice;

public interface HelloWS {
    public String hello(String name);
}
```

Figura 4 - Interface de terminal em serviço HelloWS.java

O WSDL do serviço da Web é mostrado na Figura 5.

```
- <definitions targetNamespace="http://hello_webservice/" name="HelloWSImplService">
  - <types>
    - <xsd:schema>
        <xsd:import namespace="http://hello_webservice/" schemaLocation="http://marcio:8080/HelloWSImplService/HelloWSImpl?xsd=1"/>
      </xsd:schema>
    </types>
  - <message name="hello">
      <part name="parameters" element="tns:hello"/>
    </message>
  - <message name="helloResponse">
      <part name="parameters" element="tns:helloResponse"/>
    </message>
  - <portType name="HelloWSImpl">
  - <operation name="hello">
      <input wsam:Action="http://hello_webservice/HelloWSImpl/helloRequest" message="tns:hello"/>
      <output wsam:Action="http://hello_webservice/HelloWSImpl/helloResponse" message="tns:helloResponse"/>
    </operation>
  </portType>
- <binding name="HelloWSImplPortBinding" type="tns:HelloWSImpl">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
  - <operation name="hello">
      <soap:operation soapAction=""/>
    - <input>
        <soap:body use="literal"/>
      </input>
    - <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
- <service name="HelloWSImplService">
  - <port name="HelloWSImplPort" binding="tns:HelloWSImplPortBinding">
      <soap:address location="http://marcio:8080/HelloWSImplService/HelloWSImpl"/>
    </port>
  </service>
</definitions>
```

Figura 5 - WSDL do serviço da Web

Kalin (2010) também explica que varias linguagens possuem utilitários para gerar código de suporte ao cliente a partir de um *WSDL*, o utilitário *Java* é chamado de *wsimport.*

Neste projeto, o utilitário *wsimport* não é necessário. A aplicação cliente, que acessa o *Web Service*, foi implementada para a plataforma *Android* utilizando o *plug-in* para a *IDE Eclipse* chamado de *Android SDK* e a biblioteca *KSOAP2*. E a biblioteca *KSOAP2* tem a característica de ser mais flexível quanto à geração do código de suporte, necessário ao cliente para acessar o *Web Service*.

Rabello (2009) explica que *Android* ainda não possui uma *API* embutida que trata especificamente de *Web Services*, para isso ele possui a capacidade de integrar as mesmas bibliotecas de *JAVA SE* com a versão de *KSOAP2*.

Para ser oferecido através de *http*, o *Web Service* precisa de um endereço fixo e conhecido pelos clientes que irão acessar seus serviços. Para isto, boa parte dos fornecedores de planos de Internet empresarial de Internet já fornece número de IP fixo e quando não houver disponibilidade existem servidores na Internet que fornecem gratuitamente serviços de *DNS (Domain Name Service)*. "Um servidor *DNS* traduz ou relaciona os endereços IP em nomes de computadores ou *página*s de Internet. Um exemplo prático: para acessar uma página da Internet ou um computador da rede, não é necessário decorar seu IP para acessá-lo, basta decorar seu nome que o *DNS* faz a tradução" (SILVA, 2003).

Também existem serviços gratuitos de *VPN*, que criam uma Rede Virtual Privada entre dois computadores, garantindo maior confiabilidade e segurança. "Existem servidores *DNS* para a tradução de endereços externos (públicos na Internet) e internos (dentro de uma *LAN*). Quando estabelecida uma *VPN*, é disponibilizado um endereço interno para o usuário remoto poder trafegar na rede interna" (SILVA, 2003).

2.2. SERVIDORES DE APLICAÇÃO

Borrielo (2006) explica que servidores de aplicação que utilizam à plataforma J2EE estão se tornando cada vez mais freqüentes e recomendados para empresas, destacando as características de alta disponibilidade, alta capacidade para acesso simultâneo nos containers WEB e EJB, facilidades de configuração e utilização de banco de dados, facilidade de

alteração, publicação e disponibilização de sistemas em tempo real, segurança, balanceamento de carga e tolerância à falhas, como alguns dos recursos que estes servidores oferecem.

Gomes (2000) explica que J2EE fornece ao programador recursos para desenvolver softwares exatamente para ambientes corporativos. Formado por um conjunto de recursos padronizados, que fornecem os principais serviços necessários para desenvolver um sistema multicamada de qualidade.

Oliveira (2004) explica o porquê J2EE é uma poderosa plataforma para desenvolvimento de aplicativos corporativos, "Características como portabilidade e interoperabilidade fizeram com que a tecnologia se difundisse e se consolidasse pela indústria de TI. Vale lembrar que Java permite o uso de qualquer sistema operacional" (OLIVEIRA, 2004).

Caelum (2011) explica que *Java EE* é apenas um grande PDF, uma especificação, detalhando quais especificações fazem parte deste. Para usar o *software*, é necessário fazer o *download* de uma implementação dessas especificações.

Caelum (2011) também explica que existem diversas dessas implementações de *Java EE*. A Sun/Oracle desenvolve uma dessas implementações, o *Glassfish* que é *open source* e gratuito, porém não é o líder de mercado apesar de ganhar força nos últimos anos. Alguns *softwares* implementam apenas uma parte dessas especificações do *Java EE*, e portanto não é totalmente correto chamá-los de servidor de aplicação. A partir do *Java EE 6*, existe o termo "*application server web profile*", para poder se referenciar a servidores que não oferecem tudo, mas um grupo menor de especificações, consideradas essenciais para o desenvolvimento *Web*.

É importante destacar as *API's* disponibilizadas pelo *Java EE*, implementadas no *Glassfish* e como elas são utilizadas neste projeto. Segundo Caelum (2011) elas são:

- *JavaServer Pages (JSP), Java Servlets, Java Server Faces (JSF).* Neste projeto futuramente poderá ser desenvolvida uma aplicação utilizando esta API.
- *Enterprise Javabeans Components (EJB*) e *Java Persistence API (JPA).* (objetos distribuídos, clusters, acesso remoto a objetos etc). Este projeto faz uso desta duas *API's* para facilitar o trabalho de acesso aos dados do sistema *desktop* através do *Web Service*.
- *Java API for XML Web Services (JAX-WS), Java API for XML Binding (JAX-B)* (trabalhar com arquivos *XML* e *Web Services*). Também foi utilizado neste projeto para gerar o *Web Service.*

- *Java Autenthication and Authorization Service (JAAS)* (*API* padrão do *Java* para segurança). Esta *API* também poderá ser utilizada futuramente.
- *Java Transaction API (JTA*) (controle de transação no contêiner). Todo o controle de transação do projeto do *Web Service* é feito com *JTA.*
- *Java Message Service (JMS)* (troca de mensagens assíncronas). Esta *API* também poderá ser utilizada futuramente.
- *Java Naming and Directory Interface (JNDI)* (espaço de nomes e objetos). É utilizada no projeto, pois é ela que permite que aplicações cliente descubram e obtenham dados ou objetos através de um nome.
- *Java Management Extensions (JMX)* (administração da sua aplicação e estatísticas sobre a mesma). Esta *API* também poderá ser utilizada futuramente.

2.3. SISTEMA OPERACIONAL ANDROID

O sistema operacional *Android* foi utilizado como plataforma para desenvolvimento do aplicativo cliente para o *Web Service*.

Segundo Google b (2010), "O *Android* é uma pilha de *software* para dispositivos móveis que inclui um sistema operacional, *middleware* e aplicativos importantes. O *SDK* do *Android* fornece as ferramentas e *APIs* necessárias para começar a desenvolver aplicativos que executam em dispositivos com *Android*".

Lecheta (2009) também fala das características do sistema operacional para Aplicativos Móveis criado pela Google, "Inclui um sistema operacional baseado no *Linux* e diversas aplicações, com uma rica interface gráfica, um *browser* para navegar na Internet, integração com o *Google Maps*, suporte a multimídia, GPS, banco de dados integrado, jogos em 3D e muito mais." (LECHETA, 2009).

Vascolcelos (2011 b) explica que o *Android* tem uma estrutura especial para aproveitar o uso de recursos como imagens, vídeos, etc. E também possibilita a configuração de recursos alternativos para suportar configurações especificas como o tamanho da tela, orientação, etc. "O Android automaticamente verifica seus *resources* e ajusta de acordo com as configurações fornecidas pelo aparelho" (VASCONCELOS, 2011).

A Figura 6 apresenta a estrutura de um projeto *Android* gerado pela IDE Eclipe SDK.

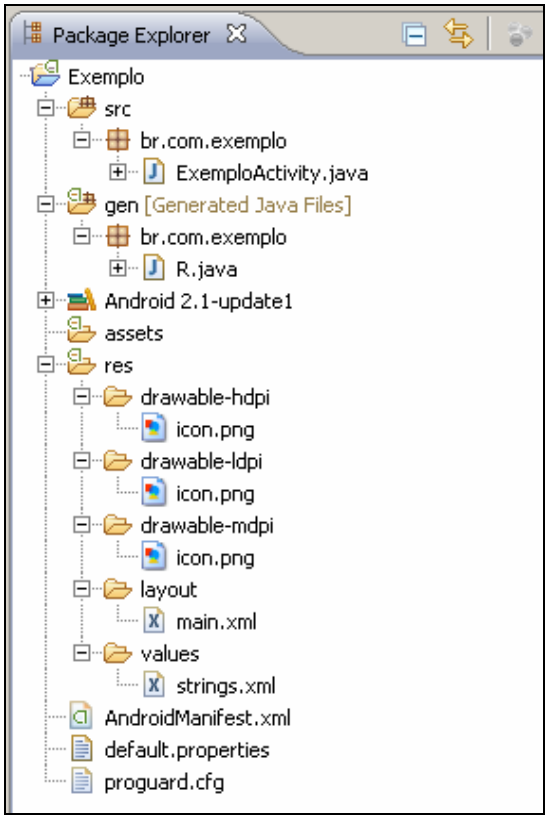


Figura 6 - Estrutura de um projeto Android gerado pela IDE Eclipe SDK

A estrutura de pastas apresentada na Figura 06 é explicada por Vascolcelos (2011 b) como:

- A pasta `src` é onde fica o código fonte da aplicação.
- A pasta `gen` (*Generated source folder*) possui somente uma única classe gerada automaticamente pelo *plugin* do *eclipse* chamada 'R'que contém ids para todos os itens da pasta `res`, para seja possível usar todos os *resources* da aplicação.
- Na pasta `res` ficam os *resources* da aplicação, em `res/drawable` ficam as imagens onde a divisão entre *hdpi (High dpi), mdpi (Medium dpi), ldpi (Low dpi)* é para guardar as imagens em resoluções diferentes, em `res/layout` ficam os arquivos XMLs que representam o *layout* das *Activities*, e em `res/values` ficam os XMLs que guardam *Strings* que podem ser usadas na aplicação.
- Na pasta `assets` ficam arquivos que podem ser abertos por *stream* dentro da aplicação.

Segundo Rabelo (2009) “Em *Android*, todos os componentes de interface gráfica são representados por subclasses de `android.view.View`, que representam os componentes gráficos (os chamados *widgets*) e a classe `android.view.ViewGroup`, que representa um container de *Views* e também *ViewGroups*” (RABELLO, 2009).

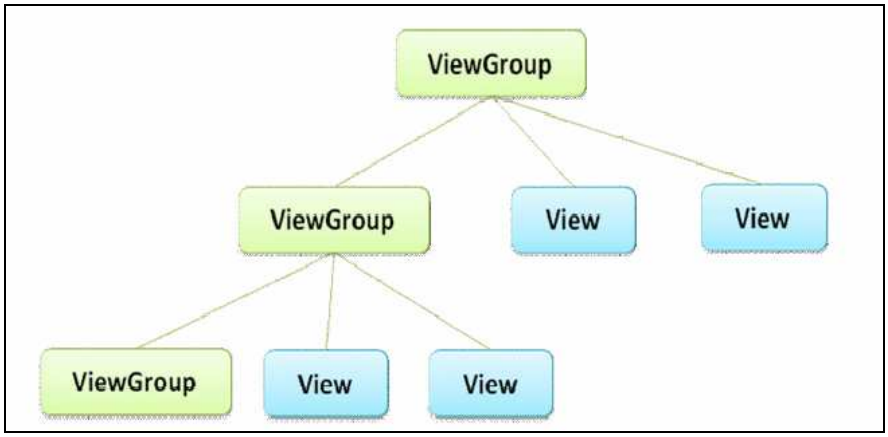


Figura 7 - Hierarquia de componentes de tela.
Fonte: RABELLO (2009).

A pasta `src` é onde fica o código fonte do projeto e quando um projeto é criado pelo *Eclipse SDK*, automaticamente é gerada uma classe *Activity*. A Figura 8 apresenta o código fonte gerado automaticamente nesta classe.

ExemploActivity.java

```
package br.com.exemplo;

import android.app.Activity;

public class ExemploActivity extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

Figura 8 - Código fonte gerado automaticamente em uma *Activity*

A programação em *Android* depende da implementação de algumas classes padrão existentes na arquitetura do Sistema Operacional. As sessões que seguem apresentam estas classes com uma definição, quando são necessárias e como elas são utilizadas.

2.3.1. ACTIVITY

Segundo Silva (2011) todas as classes de uma aplicação de *Android* devem ser derivadas da classe *Activity* (Atividade) e possui, como método principal, o método `onCreate`. Dentro desse método ele invoca o método `onCreate` da super classe passando mesmo parâmetro (o `savedInstanceState`), logo após esse método, vem o método `setContentView`, responsável por exibir a tela da minha aplicação, baseado nos *layouts xml*. Por padrão ele chama o arquivo `main.xml`.

Vascolcelos (2011 a) explica que cada *Activity*, é uma tela da aplicação, onde é possível adicionar componentes (*Views*) e programar eventos. O aplicativo iniciará na *Activity* que for declarada no `AndroidManifest.xml` como sendo a principal. A Figura 9 apresenta o código do arquivo `AndroidManifest.xml`.

Figura 9 - Arquivo AndroidManifest.xml

Vascolcelos (2011 a) explica que para colocar os componentes na tela, existe um arquivo chamado `main.xml` dentro da pasta `res/layout`, e usa-se `R.layout.main` para referir a ele, o `setContentView` vai ler esse XML e carregá-lo na tela. A Figura 10 apresenta o código gerado dentro do arquivo `main.xml`.

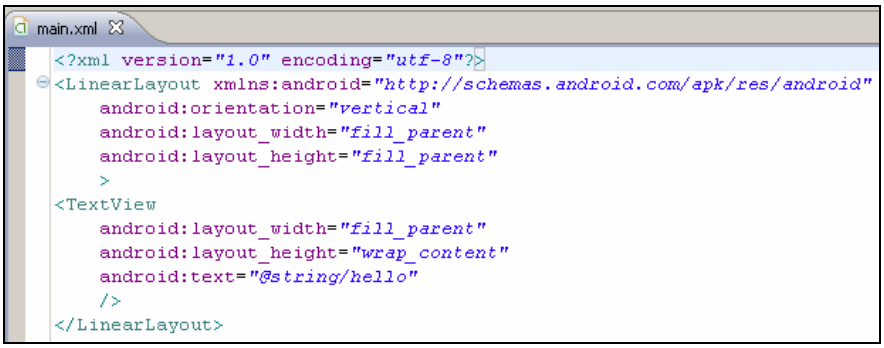


Figura 10 - Arquivo de *layout* main.xml

Vascolcelos (2011 a) explica o conteúdo deste arquivo `main.xml` da seguinte forma: O `LinearLayout` que ocupa a tela inteira (`fill_parent, fill_parent`) com um `TextView` que ocupa toda largura dentro do *Layout (`fill_parent`)*, mas ocupa só o necessário em altura (`wrap_content`). Existem três valores possíveis para os atributos `android:layout_(width/heigth)`.

- **`match_parent`** – O componente irá ocupar todo o espaço disponível;
- **`fill_parent`** – Igual ao `match_parent`, mas esse valor foi depreciado e pode-se usar `match_parent` no lugar;
- **`wrap_content`** – O componente ocupara apenas o tamanho necessário que ele precise para exibir seu conteúdo corretamente;

O atributo `android:text` tem o valor "`@string/hello`", essa *String* está declarada em `res/values/strings`, e é possível usar esses valores pelo nome do tipo (`@strings/`) e nome do valor "`hello`". A Figura 11 apresenta o código do arquivo `strings.xml`.

Figura 11 - Arquivo Strings.xml

A Figura 12 apresenta o resultado no emulador deste projeto exemplo utilizado até agora para explicar como funciona a estrutura de uma aplicação *Android*.

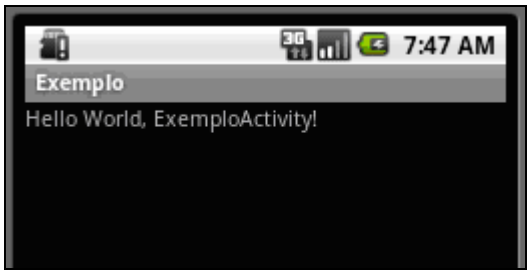


Figura 12 - Resultado do exemplo de projeto Android.

Vascolcelos (2011 a) explica que, para referenciar os componentes via programação, dentro do arquivo de *layout*, é necessário declarar um id para os mesmos no atributo `android:id`. Após o comando `setContentView(R.layout.main)` pode-se acessar os componentes da tela pelo método `findViewById(int id),` onde o id já deve estar definido nos componentes pelo atributo `android:id`, que são acessíveis pela classe R.

2.3.2. INTENT

Como uma aplicação em *Android* é composta de várias telas é necessária uma maneira de carregar uma nova tela a partir de outra.

Neto (2010) explica que, para chamar uma nova tela, um serviço ou alternar entre telas, sempre é necessário usar uma *Intent*. E é através desta *Intent* que serão passados os parâmetros (objetos) para a outra tela. Pode-se chamar uma nova tela através do comando apresentado na Figura 13.

```
Intent intent = new Intent();
intent.setClassName("br.com.eversource.android",  "br.com.eversource.android.DetailActivity");
intent.putExtra("registro", registro);
intent.putExtra("algumaString", algumaString);
startActivity(intent);
```

Figura 13 - Carregando uma nova Activity a partir de outra.

E recuperar este parâmetro na *Activity* aberta com o código apresentado na Figura 14.

```
Intent intent = getIntent();
Registro registro = (Registro) intent.getSerializableExtra("registro");
String algumaString = intent.getString("algumaString");
```

Figura 14 - Recuperando o valor do parametros a partir da outra *Activity*.

O registro da *Activity* no `AndroidManifest.xml` é feito de acordo com o código apresentado na Figura 15.

```
<activity android:name=".DetailActivity " android:label="Titulo" />
```

Figura 15 - Registro do *Activity no Androidmanifest.xml*.

"Para fazer a passagem de objetos entre *Activities via Intent*, basta fazer com que eles implementem a interface `java.io.Serializable`, inclusive adicionando-os a listas (a classe `ArrayList` é serializada)" (LEAL, 2011). Para passar o objeto utiliza-se o código apresentado na Figura 16.

```
intent.putSerializable("identificador", Objeto);
ou
intent.putSerializable("identificador", new ArrayList<Classe>);
```

Figura 16 - Passando objetos para uma nova *Activity* a partir de outra.

E para recuperar na outra *Acitivty* utiliza-se o código apresentado na Figura 17.

```
ArrayList<Classe> lista  = (ArrayList<Classe>)
getIntent().getSerializableExtra("lista");
```

Figura 17 - Recuperando objetos a partir da outra *Activity*.

2.3.3. ADAPTER

"Para preencher diversos componentes de UI como *ListView*, *Spinner* (*combo box*) e *GridView*, o *Android* utiliza o conceito de *Adapters*. Esse é um padrão de projeto muito útil e serve, como o próprio nome diz, para adaptar uma lista de objetos para uma lista de objetos de interface gráfica" (LEAL, 2010).

Leal (2010) explica também, que é necessário criar uma classe que implementa `android.widget.BaseAdapter`. Esta classe terá dois atributos que são inicializados no construtor da classe, um atributo da classe `amdroid.content.Context`, que será utilizado para carregar o arquivo de layout e uma lista de objetos da classe POJO para adaptar. Para criar um *adapter*, é necessário implementar quatro métodos: `getCount, getItem, getItemId e getView`. O método `getCount` a quantidade de linhas que o adaptador representa, bastando retornar o tamanho da lista. O método `getItem` serve para acessar o objeto que o adaptador está representando, basta retornar o objeto da posição da lista. Já o método `getItemId` serve pra retornar um identificador do objeto da posição passada como

parâmetro. O método `getView` é o mais importante, é ele que vai pegar um objeto da lista, carregar o arquivo de *layout* e atribuir os valores dos atributos do objeto para a *view* que representará a linha da lista.

Leal (2010) também explica que, para carregar o arquivo de *layout* utiliza-se a classe `android.view.LayoutInflater`. Após é retornado um objeto *View* que representa a árvore de *Views* definida no arquivo *XML* e pega-se as referências para os *TextViews* para passar os valores dos atributos da classe para eles. No final retorna-se a *View* que foi carregada e modificada.

2.3.4. BIBLIOTECA KSOAP2

Para que a aplicação cliente em Android possa acessar o *Web Service* foi utilizada a biblioteca "`ksoap2-android-assembly-2.4-jar-with-dependencies`", que possui as classes que geram as requisições e recebem as respostas, necessárias para se comunicar com o *Web Service*.

3 MATERIAIS E MÉTODOS

Todo o projeto, tanto o *Web Service* quanto o aplicativo cliente em *Android* foi desenvolvido utilizando o Java SDK (*Software Develpment Kit*) na versão 7. O *Java SDK* é composto por dois pacotes de software: *JDK (Java Development Kit)*, que contém ferramentas de software para o desenvolvimento de aplicações utilizando a Plataforma *Java*, e o *JRE (Java Runtime Environment)*, que contém ferramentas de *software* necessárias à execução de aplicações *Java*.

Para desenvolver o *Web Service* e o aplicativo cliente para *Android* foi utilizada a ferramenta de desenvolvimento IDE (*Integrated Development Environment*) *Eclipse SDK*, na versão *Hélios Service Release 2*.

Este trabalho de diplomação fez uso da UML (*Unified Modeling Language*) para a elaboração da documentação resumida de análise e projeto dos sistemas *Web Service* e aplicativo *Android*.

Outras tecnologias e detalhes sobre a programação/desenvolvimento do projeto serão demonstradas nas seções deste capitulo.

3.1. DESCRIÇÃO DO PROJETO

O usuário tem a necessidade de acessar, através da Internet, informações sobre aniversariantes cadastrados em um banco de dados legado de um sistema que não funciona através de uma conexão de Internet. Para isto, foi desenvolvido um *Web Service*, que se conecta ao banco de dados legado, e fornece serviços para que aplicações clientes possam acessá-los.

Como aplicação cliente foi projetado e desenvolvido um aplicativo para dispositivos móveis com sistema operacional *Android*. Este aplicativo realiza consultas ao serviço do *Web Service* que retorna uma lista de aniversariantes de uma determinada data, com os dados de contato dos mesmos. Após a realização do contato com o aniversariante, o aplicativo utiliza um serviço de registro do contato, disponibilizado pelo *Web Service*, que cadastra as informações sobre o contato realizado no banco de dados legado.

3.2. CASOS DE USO

Nesta sessão serão apresentados os casos de uso que envolvem a utilização do aplicativo *Android* por um único ator Usuário para acessar os serviços do *Web Service*.

3.2.1. DIAGRAMA DE CASOS DE USO

Os casos de uso são apresentados na Figura 18.

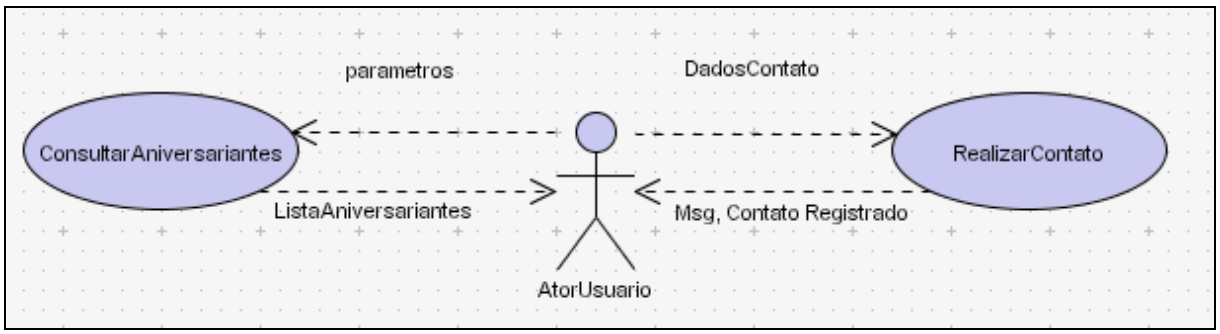


Figura 18 - Diagrama de casos de uso.

3.2.2. DESCRIÇÃO DOS CASOS DE USO

Nesta seção será apresentada uma descrição de cada caso de uso com seus cursos normais e alternativos, que demonstram ao programador as regras que envolvem a utilização do aplicativo cliente e sua comunicação com o *Web Service*.

Número: 01.
Caso de uso: ConsultarAniversariante.
Descrição: Este caso de uso trata da consulta de aniversariantes.
Ator: Usuário.

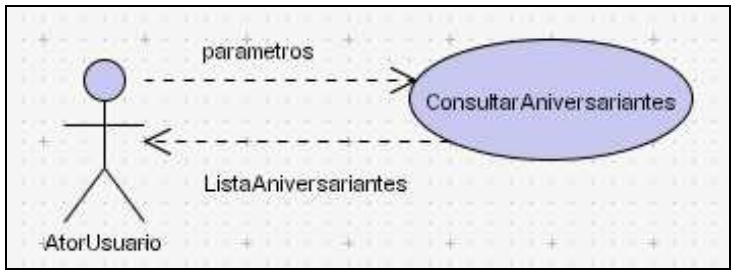


Figura 19 - Caso de uso ConsultarAniversariante

Curso Normal:

1. O usuário solicita uma consulta de aniversariantes.
2. O sistema verifica que existe uma conexão ativa com a Internet.
3. O sistema abre a tela de consulta de aniversariantes.
4. O usuário informa uma data, opcionalmente um nome e solicita uma consulta.
5. O sistema solicita o serviço de consulta de aniversariantes do *Web Service*.
6. O *Web Service* retorna uma lista com todos os aniversariantes do dia.
7. O sistema mostra uma lista com os dados dos aniversariantes e encerra o caso de uso.

Cursos Alternativos:

2. O sistema verifica que não existe uma conexão ativa com a Internet.
   2.1 O sistema exibe a mensagem que a Internet não esta conectada, atualiza o valor do campo statusConexão e encerra o caso de uso.
6. O *Web Service* não responde.
   6.1 O sistema exibe a mensagem com o erro e encerra o caso de uso.
6. O *Web Service* retorna uma lista de aniversariantes vazia.
   6.1 O sistema exibe a mensagem que não existem aniversariantes e encerra o caso de uso.

Número: 02.
Caso de uso: RealizarContato.
Descrição: Este caso de uso trata da realização do contato com o aniversariante.
Ator: Usuário.

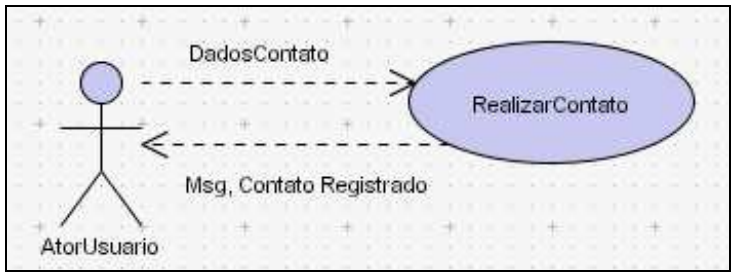


Figura 20 - Caso de uso RealizarContato

Curso Normal:

1. Após a realização do caso de uso ConsultarAniversariante o usuário seleciona o usuário que deseja realizar um contato.
2. O sistema verifica que o contato com o aniversariante ainda não foi realizado e mostra um dialogo com as opções de contato: ligar para telefone fixo, ligar para telefone celular, enviar SMS.
3. O usuário seleciona a opção de contato: Ligar para telefone fixo.
4. O sistema inicia a discagem para o numero de telefone fixo do aniversariante.
5. O usuário encerra o contato com o aniversariante.
6. O sistema pergunta se o usuário deseja registrar o contato.
7. O usuário informa que deseja registrar o contato.
8. O sistema solicita o serviço de registro de contato ao *Web Service*.
9. O *Web Service* retorna uma confirmação do registro do contato.
10. O sistema atualiza o valor do campo “Contato” do Aniversariante da lista com a data de realização do contato, mostra a lista de aniversariantes e encerra o caso de uso.

Cursos Alternativos:

2. O sistema verifica que o contato com o aniversariante já foi realizado.
   2.1. O sistema exibe a mensagem que a o contato com o aniversariante já foi realizado e encerra o caso de uso.
3. O usuário seleciona a opção de contato: Ligar para telefone celular.
   3.1. O caso de uso é realizado a partir do passo 4 com o numero de telefone celular do aniversariante.
3. O usuário seleciona a opção de contato: Enviar SMS.
   3.1. O sistema exibe uma tela para o usuário digitar a mensagem SMS.

3.2. O usuário digita a mensagem e seleciona o botão enviar.

3.2. O sistema exibe a mensagem de SMS enviado e realiza o passo 6.

7. O usuário informa que não deseja registrar o contato.

7.1. O sistema mostra a lista de aniversariantes e encerra o caso de uso.

9. O *Web Service* não responde.

9.1 O sistema exibe a mensagem com o erro.

9. O *Web Service* retorna que o registro do contato não foi realizado.

9.1 O sistema exibe a mensagem que não foi possível registrar o contato, mostra a lista de aniversariantes e encerra o caso de uso.

3.3. DIAGRAMAS DE CLASSES

O diagrama de classes apresentado na Figura 21 apresenta as classes existentes no projeto do *Web Service*.

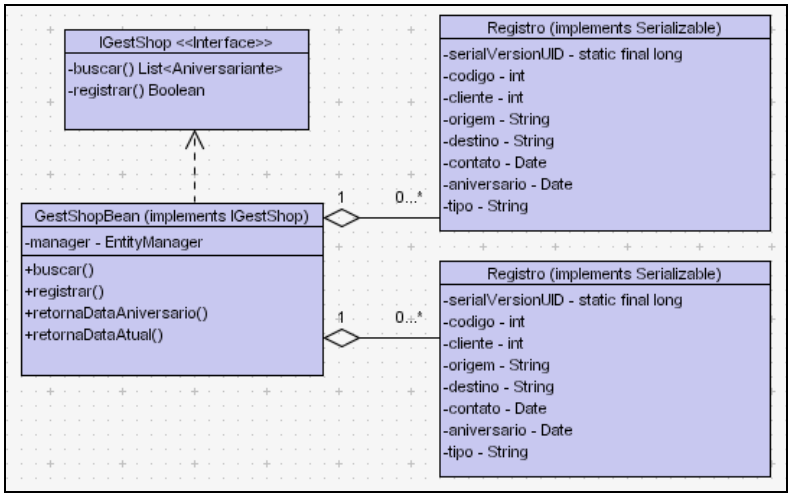


Figura 21 - Diagrama de Classes do aplicativo *Android*.

O diagrama de classes apresentado na Figura 22 apresenta as classes existentes no projeto do aplicativo cliente.

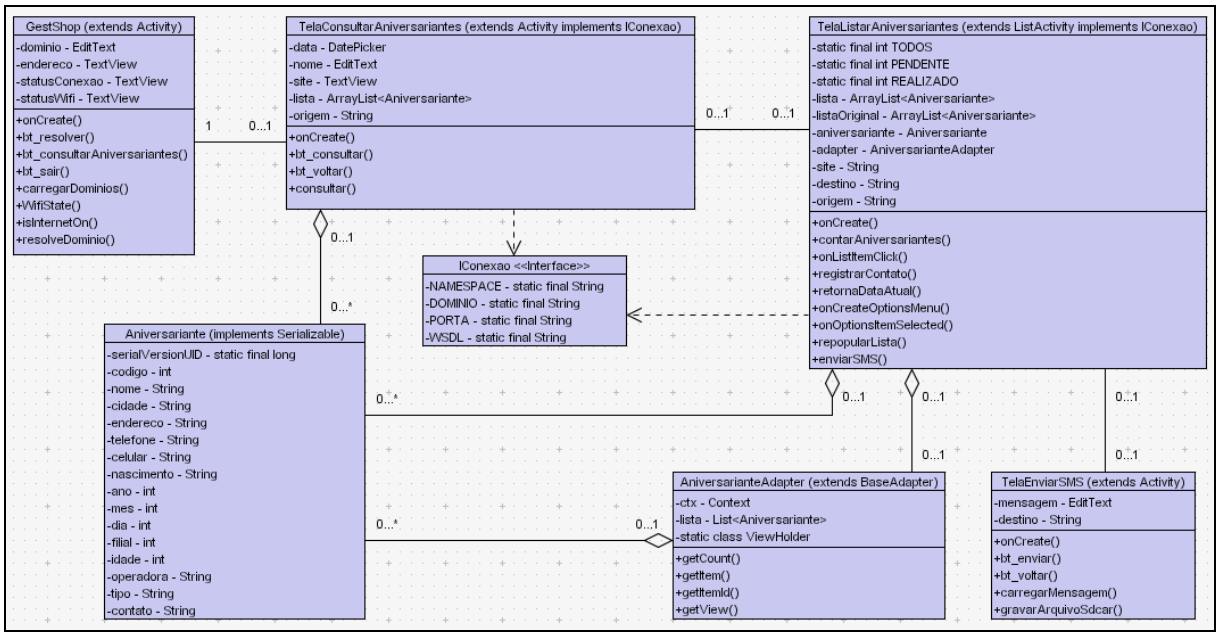


Figura 22 - Diagrama de Classes do aplicativo *Android*.

Os diagramas de classes apresentados nas Figuras 21 e 22 demonstram de forma ilustrativa as classes criadas nos projetos do *Web Service* e do aplicativo cliente em *Android*.

3.4. DIAGRAMAS DE SEQÜÊNCIAS

A Figura 23 apresenta o diagrama de seqüências que representa de forma ilustrativa o caso de uso ConsultarAniversariante.

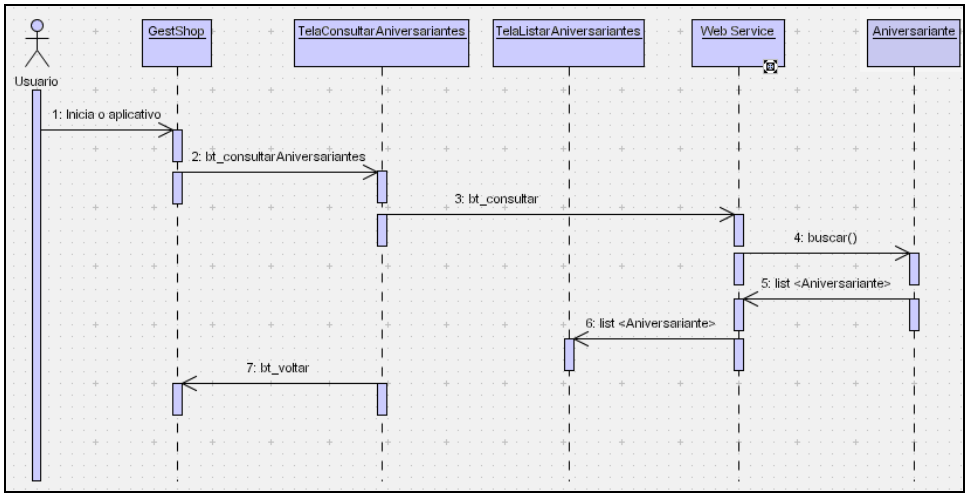


Figura 23 - Diagrama de Seqüência ConsultarAniversariante.

A Figura 24 apresenta o diagrama de seqüências que representa de forma ilustrativa o caso de uso RealizarContato.

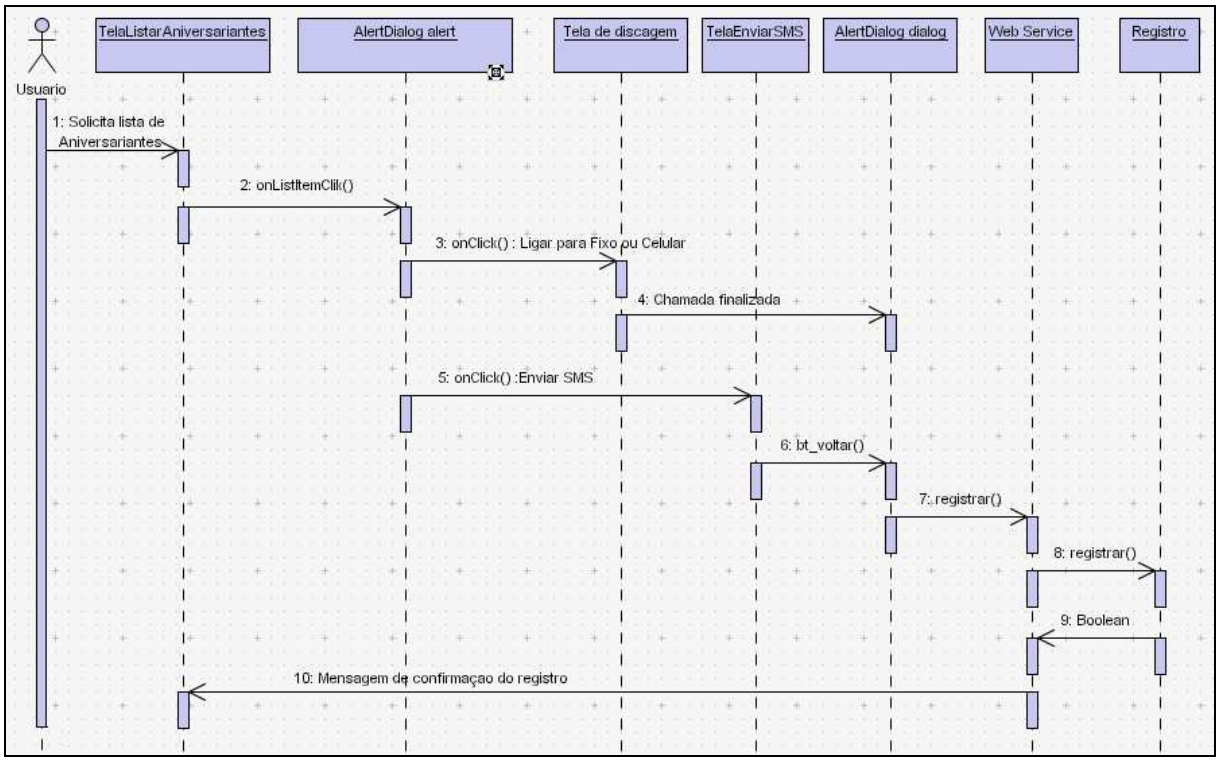


Figura 24 - Diagrama de Seqüência RealizarContato.

Os diagramas de seqüências apresentados nas Figuras 23 e 24 demonstram a lógica de consulta de aniversariantes pelo ator Usuário, o processo de realização do contato e o registro do mesmo.

3.5. DIAGRAMA DE ATIVIDADES

O diagrama de atividades apresentado na Figura 25 demonstra de forma ilustrativa as regras que envolvem a utilização do aplicativo cliente e sua comunicação com o *Web Service*.

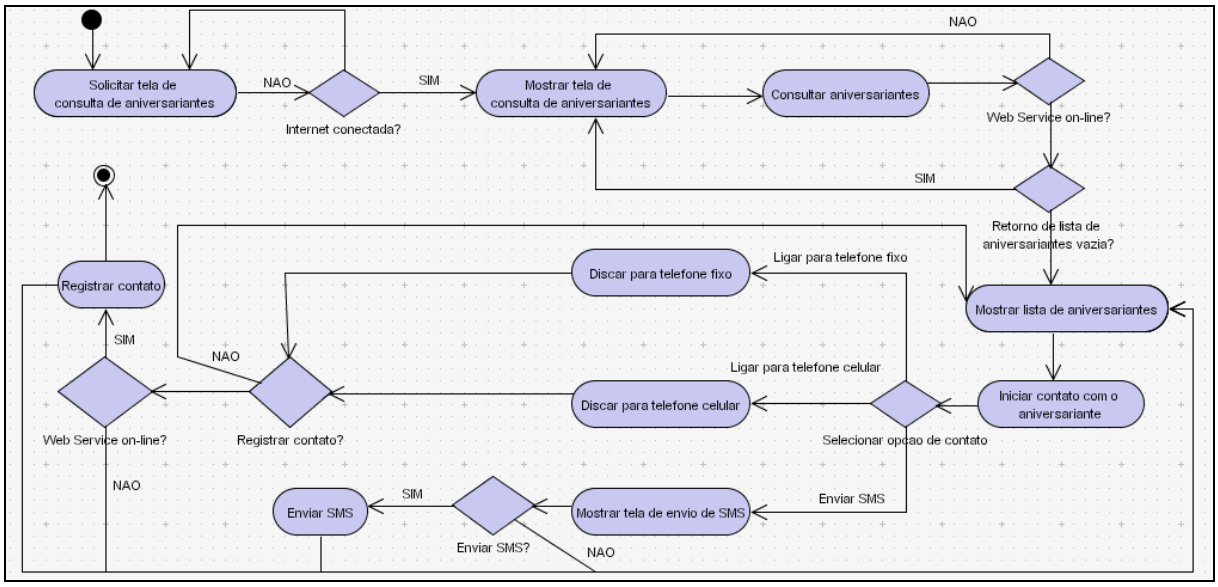


Figura 25 - Diagrama de Atividades

3.6. DIAGRAMA DE IMPLANTAÇÃO

O diagrama de implantação apresentado na Figura 26 demonstra de forma ilustrativa os componentes que formam a infraestrutura necessária para comunicação entre o aplicativo cliente em *Android* e o *Web Service*.

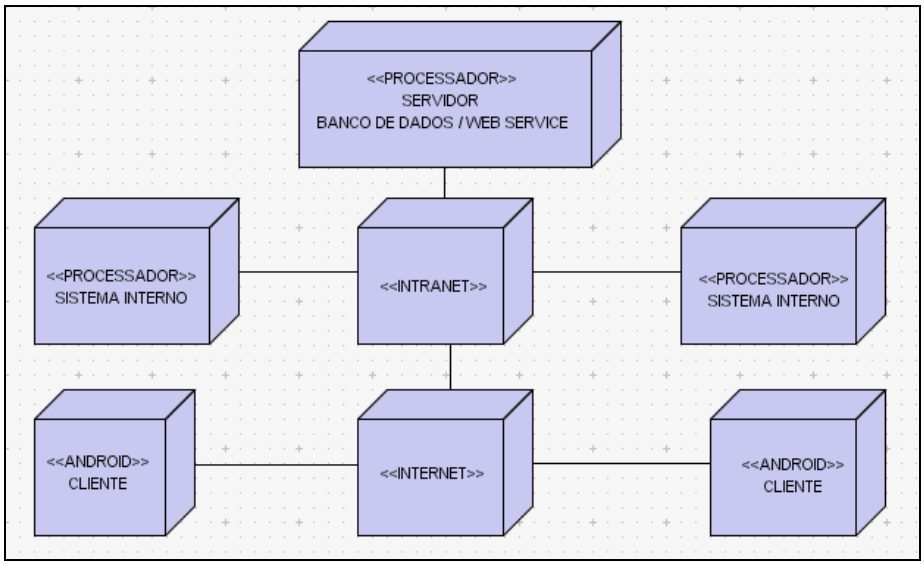


Figura 26 - Diagrama de Implantação

3.7. ESTRUTURA DO WEB SERVICE

O *Eclipse SDK* possibilita a instalação do *plugin chamado GlassFish Java EE Application Server Plugin for Eclipse*, que integra o *GlassFish* com o *Eclipse SDK*, e possui as bibliotecas necessárias para desenvolvimento do *Web Service*, utilizando recursos de infraestrutura disponibilizados pelo servidor de aplicação *GlassFish 3.1*, tais como: *Enterprise Javabeans Components (EJB), Java Persistence API (JPA), Java API for XML Web Services (JAX-WS), Java API for XML Binding (JAX-B), Java Transaction API (JTA) e Java Naming and Directory Interface (JNDI)*.

Com base nos objetivos deste projeto, foi optado por utilizar o servidor de aplicação *Oracle Glassfish Server 3.1*, pelos seguintes motivos:

- Testou-se as implementações *Glassfish*, *JBoss* e *Tomcat*, e o *Glassfish* atendeu as necessidades com maior rapidez e qualidade. Quanto ao *JBoss* e *Tomcat* os projetos testados ainda não funcionam totalmente e poderão ser objetos de novas pesquisas após a conclusão deste projeto.
- É um servidor de aplicação *open source* e não irá gerar custo adicional ao cliente.
- A utilização de um servidor de aplicação como o *Glassfish* em vez de apenas um servidor *Web*, deixa a oportunidade de expandir o projeto com novas aplicações.
- O *Glassfish* possui um *framework* chamado *Toplink* para fazer o mapeamento objeto-relacional (*ORM - Object Relational Mapping*). O que facilita muito o trabalho de acesso aos dados do sistema *desktop* através de *drivers JDBC.* Além de prover uma transação a nível de objetos, suportando gerenciamento direto de transações, ou externo utilizando *JTA* para interagir com o monitor de transações do *EJB Container*.

O *Web Service* desenvolvido no GlassFish 3.1 e sua estrutura gerada inicialmente a partir de um *EJB Project* é apresentada na Figura 27.

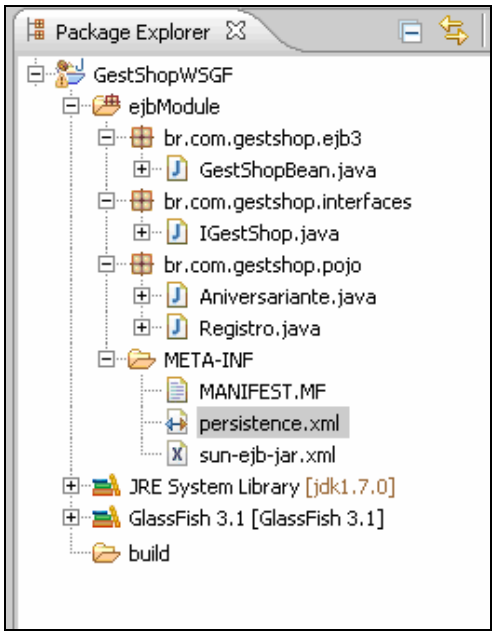


Figura 27 - Estrutura do Web Service

Dentro do pacote "`br.com.gestshop.interfaces`" existe uma interface chamada `IGestShop.java` que declara os métodos que serão as operações ou serviços do *Web Service*. Como visto no referencial teórico esta interface é chamada de *SEI: Interface Endpoint* do Serviço (*Service Endpoint Inteface*). A Figura 28 apresenta o conteúdo da interface `IGestShop.java`.

```
package br.com.gestshop.interfaces;

public interface IGestShop {

        List<Aniversariante> buscar(
                        String nome,
                        int por_nome,
                        int mes,
                        int dia);

        Boolean registrar(
                        int cliente,
                        String origem,
                        String destino,
                        String aniveresario,
                        int idade,
                        String tipo) throws ParseException;
}
```

Figura 28 - Interface `IGestShop.java`

Dentro do pacote "`br.com.gestshop.ejb3`" existe uma classe chamada `GestShopBean.java` que implementa a interface `IGestShop.java` e define o que cada método declarado na interface deve realizar. Como visto no referencial teórico esta implementação é chamada de SIB: Bean de Implementação do Serviço (*Service Implementation Bean*), neste projeto ele foi implementado como um EJB de Sessão sem

estado (*Enterprise Java Bean*). A Figura 29 apresenta o conteúdo da classe `GestShopBean.java`.

```
package br.com.gestshop.ejb3;

@Stateless
@WebService(targetNamespace = "http://ejb3.gestshop.com.br/")
@Remote(IGestShop.class)
public class GestShopBean implements IGestShop{

	@PersistenceContext(unitName="pu1")
	private EntityManager manager;

	@WebMethod
	public List<Aniversariante> buscar(
			@WebParam(name="nome") String nome,
			@WebParam(name="por_nome") int por_nome,
			@WebParam(name="mes") int mes,
			@WebParam(name="dia") int dia) {

		Query q = manager.createNamedQuery("Aniversariante.buscar");
		q.setParameter("nome", nome+"%");
		q.setParameter("por_nome", por_nome);
		q.setParameter("mes", mes);
		q.setParameter("dia", dia);
		@SuppressWarnings("unchecked")
		List<Aniversariante> cadastros = (List<Aniversariante>) q.getResultList();
		return cadastros;

	}

	@WebMethod
	public Boolean registrar(
			@WebParam(name="cliente") int cliente,
			@WebParam(name="origem") String origem,
			@WebParam(name="destino") String destino,
			@WebParam(name="aniversario") String aniversario,
			@WebParam(name="idade") int idade,
			@WebParam(name="tipo") String tipo) throws ParseException {

		Registro r  = new Registro();
		r.setCliente(cliente);
		r.setOrigem(origem);
		r.setDestino(destino);
		r.setContato(retornaDataAtual());
		r.setAniversario(retornaDataAniversario(aniversario, idade));
		r.setTipo(tipo);

		try{
			manager.persist(r);
			return true;
		}catch (Exception e){
			return false;
		}

	}

	@WebMethod(exclude = true)
	private Date retornaDataAniversario(String aniversario, int idade) throws
							ParseException {

		String str = aniversario;
		SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
		Date data = sdf.parse(str);
		Calendar calendar = Calendar.getInstance();
		calendar.setTime(data);
		calendar.add(calendar.YEAR, idade);
		return calendar.getTime();

	}

	@WebMethod(exclude = true)
	public Date retornaDataAtual() throws ParseException {

		SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
		Date y = new Date();
		String str = sdf.format(y);
		Date data = sdf.parse(str);
		return data;

	}

}
```

Figura 29 - Classe `GestShopBean.java`

A função desta classe é executar as regras de negócio da aplicação. A anotação `@Stateless`, identifica para o EJB Container que esta classe é um *Stateless Session Bean* e

não irá manter o estado de conversacional entre as diversas invocações, como visto anteriormente ele precisa ser um EJB de sessão sem estado.

Usando anotação `@WebService` indica-se ao servidor de aplicação que este *Stateless Session Bean* implementa um *Web Service*. O *GlassFish* necessita também de uma definição quanto ao targetNamespace do *Web Service*, para isto foi definido como `(targetNamespace = "http://ejb3.gestshop.com.br/").`

A anotação `@Remote` define a interface `IGestShop.java` como remota, oferecendo acesso a clientes remotos.

Os métodos são anotados com `@WebMethod` que indica ao servidor de aplicação que este método será exposto aos clientes remotos como um serviço que poderá ser consumido. Alguns métodos possuem a anotação `@WebMethod(exclude = true),` que indica que este método "não" será exposto como um serviço aos clientes remotos.

Os parâmetros dos métodos também são anotados com `@WebParam(name="nome"),` isto é necessário porque algumas aplicações clientes, como é o caso do *Android*, não encontram os parâmetros se não tiver esta anotação.

O servidor de aplicação *GlassFish* 3.1 já vem com o *provider* de persistência pré-configurado chamado *TopLink*. Para utilizar ele é necessário declarar um *EntityManager* dentro do *Session Bean* usando a anotação `@PersistenceContext`. É necessário também definir a `(unitName="pu1"),` onde "`pu1`" é unidade de persistência utilizada, ou seja, indica ao *provider* qual arquivo *persistence.xml* que contem as propriedades que referenciam o banco de dados que será utilizado. A Figura 30 apresenta o conteúdo do arquivo *persistence.xml*.

```
<persistence 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"
        version="1.0">
        <persistence-unit name="pu1" transaction-type="JTA">
                <jta-data-source>jdbc/__Firebird</jta-data-source>
            <class>br.com.gestshop.pojo.Aniversariante</class>
            <class>br.com.gestshop.pojo.Registro</class>
                <properties>
                    <property name="eclipselink.logging.level" value="INFO" />
                    <property name="eclipselink.show_sql" value="true" />
                </properties>
        </persistence-unit>
</persistence>
```

Figura 30 - Arquivo persistence.xml

Dentro do arquivo `persistence.xml` existe uma referencia a um *datasource* "`<jta-data-source>jdbc/__Firebird</jta-data-source`" que representa

uma ligação genérica com uma fonte de dados, ou seja, o banco de dados legado do sistema *desktop*.

Este *datasource* foi configurado a partir do *GlassFish Server Administration Console*, acessado utilizando um *browser* através do endereço “http://localhost:4848”. A Figura 31 apresenta a tela inicial deste *console* de administração do *GlassFish* 3.1.

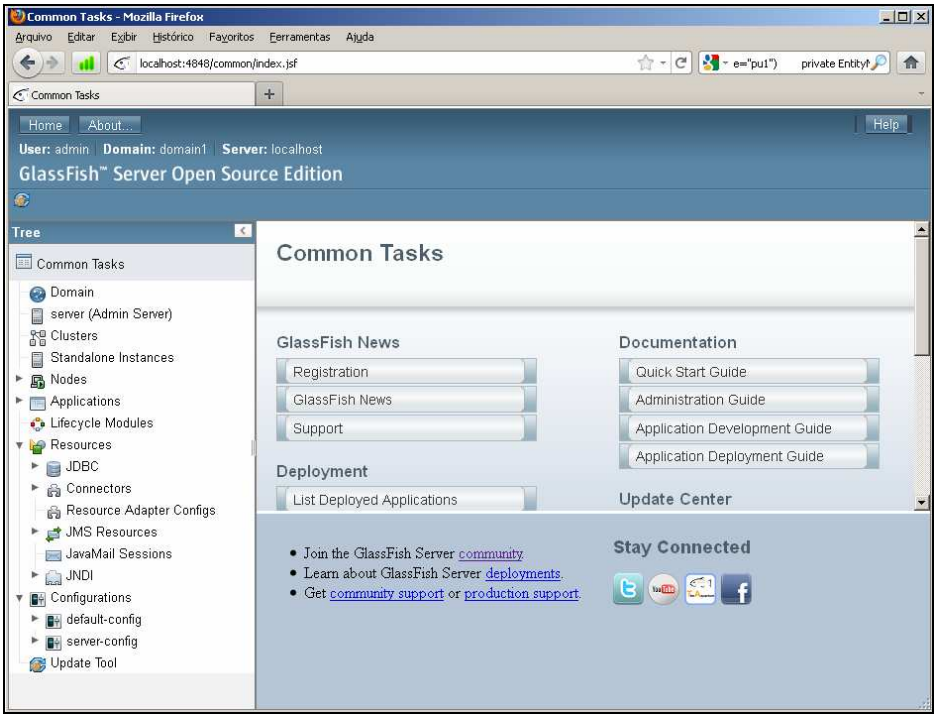


Figura 31 - Console de administração do GlassFish 3.1.

O banco de dados acessado funciona com o SGDB (Sistema Gerenciador de Banco de Dados) *Firebird*. E para que este pool de conexão funcione é necessário colocar os *jar’s* da biblioteca *Jaybird-2.1.6JDK_1.6* dentro da pasta “GLASSFISH_HOME \glassfish\domains\domain1\lib” do *GlassFish 3.1*, estes jar’s são: `jaybird-2.1.6.jar`, `jaybird-full-2.1.6` *e* `jaybird-pool-2.1.6`.

No console do GlassFish 3.1, através do menu *Resources, JDBC, JDBC Connection Pools* foi possível criar um pool de conexão com o banco de dados. Através da aba *General* foram configurados alguns campos necessários como mostrado na Figura 32.

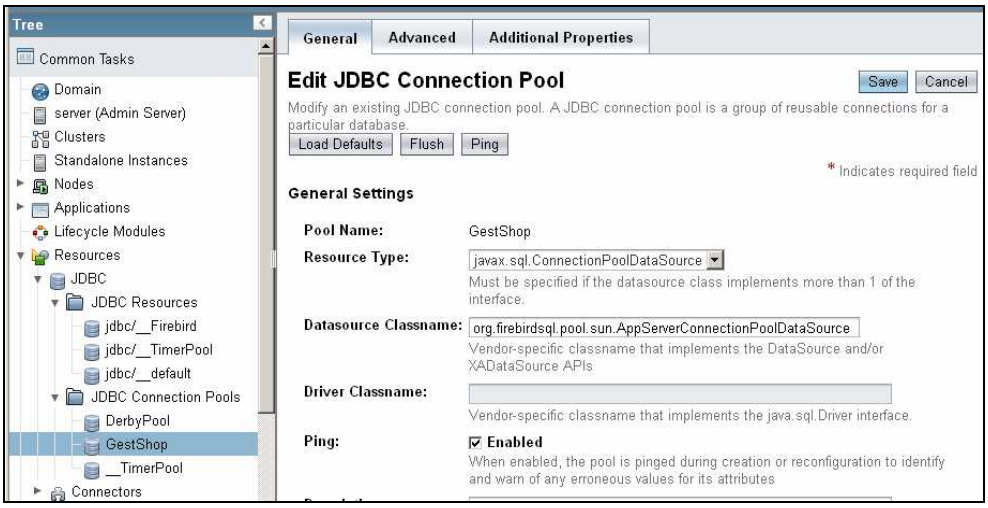


Figura 32 - Administração do GlassFish 3.1 – JDBC Connection Pools (General).

Através da aba *Additional Properties* foram informadas às propriedades que referenciam o banco de dados a ser acessado. A Figura 33 mostra este procedimento.

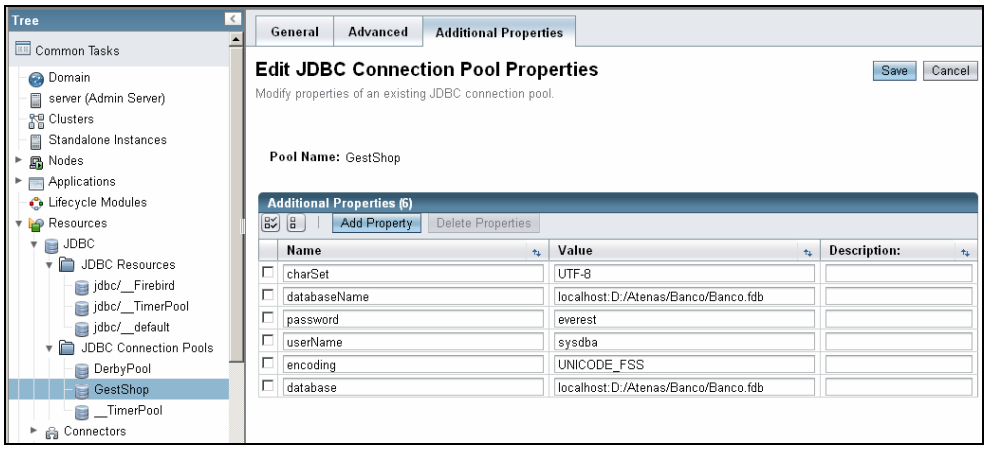


Figura 33 - Administração do GlassFish 3.1 – JDBC Connection Pools (Additional Properties).

Outra configuração necessária para que o *datasource* funcione corretamente é criar um *JDBC Resources*, como mostrado na Figura 34.

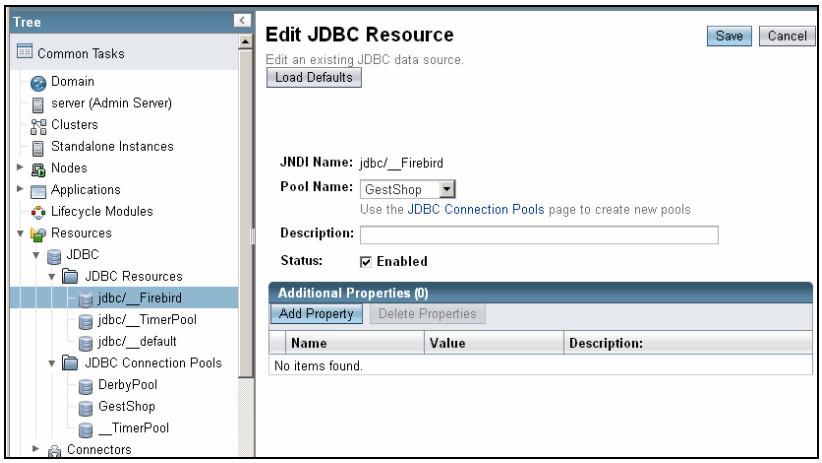


Figura 34 - Administração do GlassFish 3.1 – JDBC Resources.

Com o *datasource* configurado, o próximo passo foi criar as classes que representam os dados na nossa aplicação. Dentro do pacote "`br.com.gestshop.pojo`" foram criadas as classes `Aniversariante` e `Registro`, elas são os *Entity Beans*, o que no EJB3 quer dizer que são classes que recebem o apoio do *EJB Container* para fazer a persistência dos dados, ou seja, nestas classes são declaradas anotações para que o *framework ORM (Objetc Relational Mapping) TopLink* possa fazer o mapeamento entre os objetos e as tabelas do banco de dados.

As Figuras 35a e 35b apresenta o conteúdo da classe *Entity Bean* `Aniversariante.java`.

```
package br.com.gestshop.pojo;
@Entity
@Cache(isolation=CacheIsolationType.ISOLATED,   type=CacheType.NONE)
@NamedQueries( {
  @NamedQuery(name = "Aniversariante.buscar",
      query = "SELECT a FROM Aniversariante a WHERE
                  (:por_nome = -1 or upper(a.nome) like upper(:nome))
                              and a.mes = :mes and a.dia = :dia ")
})
@Table(name = "VW_ANDROID_ANIVERSARIANTE")
public class Aniversariante implements java.io.Serializable {
        private static final long serialVersionUID = 1L;
        private int codigo;
        private String nome;
        private String cidade;
        private String endereco;
        private String telefone;
        private String celular;
        private String nascimento;
        private int ano;
        private int mes;
        private int dia;
        private int filial;
        private String operadora;
        private String tipo;
        private String contato;

        @Id
        @Column(name = "ID", unique = true, nullable = false)
        public int getCodigo() {
                return this.codigo;
        }
```

Figura 35a - Classe *Entity Bean* `Aniversariante.java`

```
        public void setCodigo(int codigo) {
                this.codigo = codigo;
        }

        @Column(name = "NOME", nullable = false, length = 200)
        public String getNome() {
                return this.nome;
        }

        public void setNome(String nome) {
                this.nome = nome;
        }

//Outros métodos getters e setters e construtores
}
```

Figura 35b - *Classe Entity Bean* `Aniversariante.java`

Para identificar ao ORM que esta classe é um *Entity Bean* foi definida a anotação `@Entity`. As classes *Entity Bean* precisam implementar a classe `java.io.Serializable` para funcionar. A anotação `@Table(name = "VW_ANDROID_ANIVERSARIANTE")`, que informa ao ORM que a classe `Aniversariante` faz referencia a uma *View* chamada "`VW_ANDROID_ANIVERSARIANTE`".

Optou-se por uma *View* para filtrar somente os dados necessários para a aplicação a partir da tabela onde são gravados os dados dos clientes do sistema *desktop*. E também para poder já formatar estes dados facilitando assim a manipulação deles nas classes. Lembrando que a classe *Entity Bean* `Aniversariante` somente será utiliza como fonte de consulta dos aniversariantes, e não será implementado nenhum método com operações de inserir, alterar ou excluir no *Session Bean*. As Figuras 36a e 36b apresenta o conteúdo da *View* `VW_ANDROID_ANIVERSARIANTE`.

```
CREATE VIEW VW_ANDROID_ANIVERSARIANTE
(ID, NOME, CIDADE, ENDERECO, TELEFONE, CELULAR, NASCIMENTO,
 FILIAL, OPERADORA, ANO, MES, DIA, TIPO, CONTATO)
AS
SELECT
  TCAD.COD_CADASTRO AS ID,
  CAST(TCAD.DESC_CADASTRO AS VARCHAR(200)) AS NOME,
  CAST(COALESCE(TCID.DESC_CIDADE,'') || '-' ||
    COALESCE(TCID.DESC_UF,'') AS VARCHAR(100)) AS CIDADE,
  CAST(COALESCE(TCAD.DESC_ENDERECO,'') || ' ' ||
    COALESCE(TCAD.NUM_ENDERECO,'') || ' ' ||
    COALESCE(TCAD.DESC_BAIRRO,'') || ' ' ||
    COALESCE(TCAD.DESC_CEP,'') AS VARCHAR(200)) AS ENDERECO,
  CAST(COALESCE(TCAD.DESC_FONE,' ') AS VARCHAR(30)) AS TELEFONE,
  COALESCE(TCAD.DESC_CELULAR,' ') AS CELULAR,
  (CAST(EXTRACT(DAY FROM TCAD.DT_NASCIMENTO) AS VARCHAR(2)) || '/' ||
  CAST(EXTRACT(MONTH FROM TCAD.DT_NASCIMENTO) AS VARCHAR(2)) || '/' ||
  CAST(EXTRACT(YEAR FROM TCAD.DT_NASCIMENTO) AS VARCHAR(4)) ) AS NASCIMENTO,
  TCAD.COD_FILIAL AS FILIAL,
```

Figura 36a - Código SQL da *View* `VW_ANDROID_ANIVERSARIANTE`

```
  COALESCE(TCAD.DESC_OPERADORA_CELULAR,' ') AS OPERADORA,
  CAST(EXTRACT(YEAR FROM TCAD.DT_NASCIMENTO) AS INTEGER) AS ANO,
  CAST(EXTRACT(MONTH FROM TCAD.DT_NASCIMENTO) AS INTEGER) AS MES,
  CAST(EXTRACT(DAY FROM TCAD.DT_NASCIMENTO) AS INTEGER) AS DIA,
  CAST('CLIENTE' AS VARCHAR(10)) AS TIPO,
  COALESCE(CAST((SELECT
     (CAST(EXTRACT(DAY FROM DT_CONTATO) AS VARCHAR(2)) || '/' ||
     CAST(EXTRACT(MONTH FROM DT_CONTATO) AS VARCHAR(2)) || '/' ||
     CAST(EXTRACT(YEAR FROM DT_CONTATO) AS VARCHAR(4)) )
   FROM
     TBLJ_ANIVERSARIANTE_CONTATO
   WHERE
     COD_CADASTRO = TCAD.COD_CADASTRO
     AND
     DESC_TIPO_CADASTRO = 'CLIENTE'
     AND
     EXTRACT(DAY FROM DT_ANIVERSARIO) = EXTRACT(DAY FROM TCAD.DT_NASCIMENTO)
     AND
     EXTRACT(MONTH FROM DT_ANIVERSARIO) = EXTRACT(MONTH FROM TCAD.DT_NASCIMENTO)
     AND
     EXTRACT(YEAR FROM DT_ANIVERSARIO) = EXTRACT(YEAR FROM TCAD.DT_NASCIMENTO) +
     (EXTRACT(YEAR FROM CURRENT_DATE) - EXTRACT(YEAR FROM TCAD.DT_NASCIMENTO)))
         AS VARCHAR(10)),'NAO REALIZADO') AS CONTATO
FROM
    TBLJ_CADASTRO TCAD,
    TB_CIDADE TCID
WHERE
  TCAD.COD_CIDADE = TCID.COD_CIDADE
  AND
  TCAD.IND_TIPO_CADASTRO = 'CLIENTE'
  AND
  TCAD.DT_NASCIMENTO IS NOT NULL
```

Figura 36b - Código SQL da *View* VW_ANDROID_ANIVERSARIANTE

Esta *View* retorna dados sobre aniversariantes da tabela `TBLJ_CADASTRO`, onde são cadastrados os clientes do sistema *desktop*. E também, para cada registro de aniversariante, é verificado na tabela `TBLJ_ANIVERSARIANTE_CONTATO` se já existe um registro de contato referente à data de aniversario consultada, retornando a data deste contato.

A anotação `@Cache (isolation = CacheIsolationType.ISOLATED, type = CacheType.NONE)` definida nesta classe tem a função de desabilitar o *cache* do servidor. Esta anotação é necessária, porque quando os dados dos clientes são alterados, estas alterações não são visualizadas pelo aplicativo cliente em uma nova consulta.

A anotação `@NamedQueries` é utilizada para organizar um *array* de `@NamedQuery` que traduz consultas *JPA* para uma consulta *SQL*, para escrever as consultas utiliza-se a linguagem *JPA Query Language (JPQL)*.

Para cada atributo declarado na classe é necessário gerar seus respectivos métodos `getter` e `setter`, sendo que na declaração dos métodos `getters` é necessário anotar os mesmos com `@Column` que identifica ao *ORM* o nome do campo da tabela do banco de dados, no qual estes atributos se referenciam, alem de definir algumas propriedades dos campos como `name, nullable e length`.

Também é necessário identificar o atributo que corresponde à chave primaria da tabela com a anotação `@Id.`

A classe *Entity Bean* `Registro.java` é utilizada para persistir no banco de dados um registro com dados referente ao contato realizado com o cliente. A Figura 37 apresenta o conteúdo da classe *Entity Bean* `Registro.java`.

```
package br.com.gestshop.pojo;
@Entity
@Table(name = "TBLJ_ANIVERSARIANTE_CONTATO")
public class Registro implements java.io.Serializable {
        private static final long serialVersionUID = 1L;

        private int codigo;
        private int cliente;
        private String origem;
        private String destino;
        private Date contato;
        private Date aniversario;
        private String tipo;

        @Id
        @Column(name = "COD_ANIVERSARIANTE_CONTATO", unique = true, nullable = false)
        @GeneratedValue(strategy = GenerationType.IDENTITY, generator =
            "GENLJ_ANIVERSARIANTE_CONTATO")
        public int getCodigo() {
                return codigo;
        }

        public void setCodigo(int codigo) {
                this.codigo = codigo;
        }

        @Temporal(TemporalType.DATE)
        @Column(name = "DT_CONTATO", nullable = false)
        public Date getContato() {
                return contato;
        }

        //Outros métodos getters e setters e construtores
}
```

Figura 37 - Classe *Entity Bean* `Registro.java`

Esta classe *Entity Bean* `Registro.java` possui algumas anotações comuns à classe `Aniversariante.java` já explicadas anteriormente, porém existem algumas exceções que são:

- A ausência da anotação `@Cache, @NamedQueries e @NamedQuery` pelo fato da aplicação não realizar consultas para esta classe;
- A referencia da anotação `@Table` muda para "`TBLJ_ANIVERSARIANTE_CONTATO`";
- A declaração da anotação `@GeneratedValue (strategy = GenerationType.IDENTITY, generator = "GENLJ_ANIVERSARIANTE_CONTATO")` no método `getter` do atributo

que também é identificado com `@Id`. Que diz ao *ORM* sobre a estratégia de geração de código para a chave primária, que neste projeto é gerado por um *generator* já existente no banco de dados.

- A declaração da anotação `@Temporal(TemporalType.DATE)` nos métodos `getters` dos atributos do tipo *Date.*

Com o servidor de aplicação iniciado e realizado o *deploy* da aplicação, pode-se identificar o caminho do arquivo *WSDL* do *Web Service*. Através do console do *GlassFish* 3.1, no menu *Applications*, é possível visualizar o mesmo clicando sobre o link do campo *WSDL*. Este acesso é demonstrado na Figura 38.

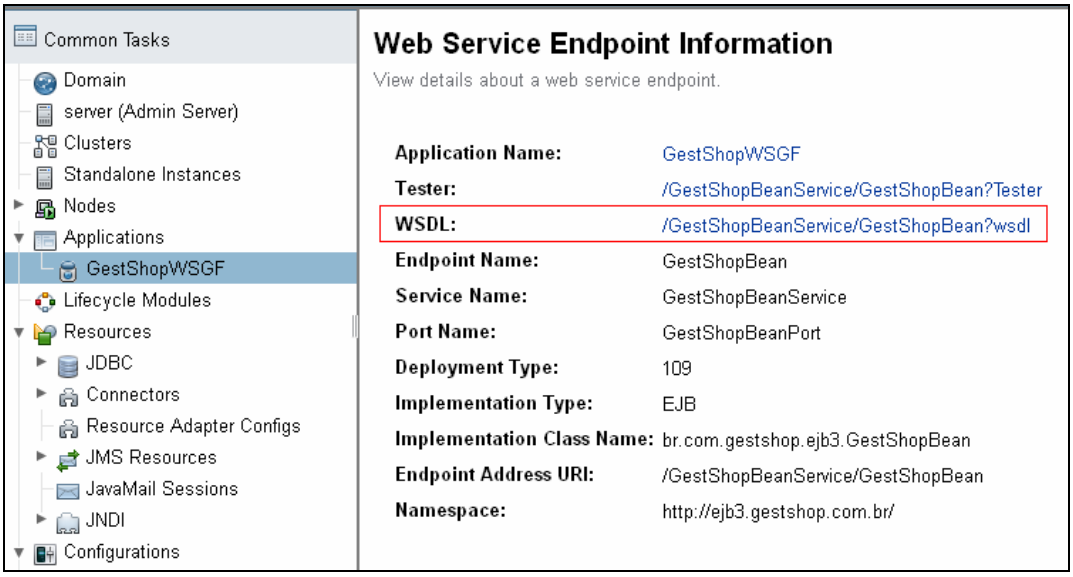


Figura 38 - Acessando o Web Service pelo Console de administração do GlassFish 3.1

A Figura 39 apresenta o arquivo *WSDL* do *Web Service* visualizado através de um *browser*.

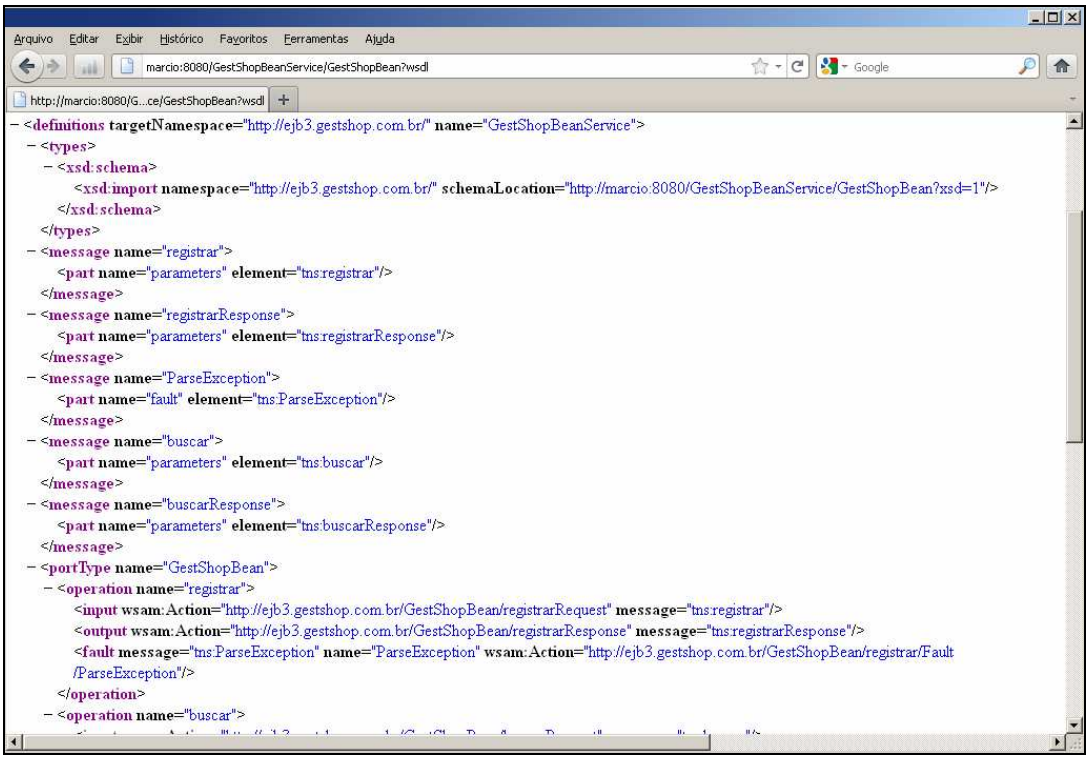


Figura 39 - Arquivo WSDL do Web Service.

No capitulo 4 sobre Resultados e Discussões é apresentado um teste do o *Web Service*.

3.8. ESTRUTURA DO APLICATIVO ANDROID

Para desenvolver o aplicativo cliente foi utilizado o *Android SDK*, juntamente com a ferramenta de desenvolvimento *Eclipse SDK*. O *Google* também disponibiliza a instalação de um *Plug-in ADT*, que integra o *Eclipse SDK* ao *Android SDK* e de um emulador de *smartphones*/celulares para rodar e testar as aplicações desenvolvidas.

A Figura 40 apresenta a estrutura do projeto do aplicativo cliente em *Android*.

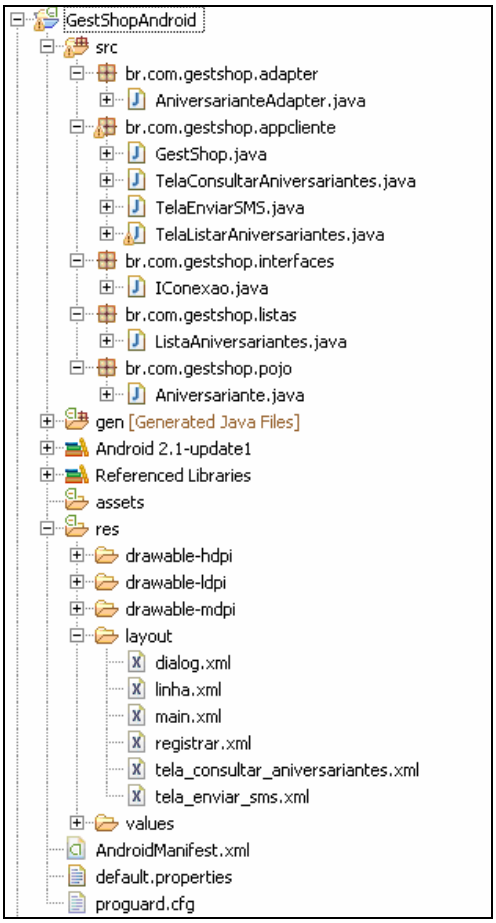


Figura 40 - Estrutura do aplicativo Android.

Na raiz do projeto existe o arquivo `AndroidManifest.xml`, em que esta declarado que a classe *Activity* que será iniciada ao abrir o aplicativo será `GestShop.java`. A Figura 41 apresenta o conteúdo do arquivo `AndroidManifest.xml`.

GestShopAndroid Manifest

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="br.com.gestshop.appcliente"
      android:versionCode="1"
      android:versionName="1.0">
    <uses-sdk android:minSdkVersion="7" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.CALL_PHONE"></uses-permission>
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
    <uses-permission android:name="android.permission.READ_PHONE_STATE" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.SEND_SMS"></uses-permission>
    <uses-permission android:name="android.permission.RECEIVE_SMS"></uses-permission>
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".GestShop"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".TelaConsultarAniversariantes" android:label="Consultar Aniversariantes" />
        <activity android:name=".TelaListarAniversariantes" android:label="Aniversariantes" />
        <activity android:name=".TelaEnviarSMS" android:label="Enviar SMS" />
    </application>
</manifest>
```

Figura 41 - Arquivo AndroidManifest.xml.

Além de declarar a classe que será iniciada ao abrir a aplicação, este arquivo possui a declaração das permissões sobre funcionalidades do *Android*, através de `uses-permission` e um `intent-filter` que diz como uma nova *Activity* será carregada.

A classe `GestShop.java` esta no pacote "`br.com.gestshop.appcliente`" e sua estrutura é apresentada na Figura 42.

```
package br.com.gestshop.appcliente;

import java.io.BufferedReader;

public class GestShop extends Activity implements IConexao{
    EditText dominio;
    TextView endereco;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        dominio = (EditText) findViewById(R.campo.dominio);
        endereco = (TextView) findViewById(R.id.endereco);
        TextView statusConexao = (TextView) findViewById(R.id.statusconexao);
        TextView statusWifi = (TextView) findViewById(R.id.statuswifi);

        dominio.setText("http://192.168.1.102");
        carregarDominios();

        //verifica se wifi esta ligada
        if(WifiState())
            statusWifi.setText("Wifi ligada.");
                else
                    statusWifi.setText("Wifi desligada.");

        //verifica se internet esta conectada
        if(isInternetOn())
            statusConexao.setText("Internet conectada.");
                else
                    statusConexao.setText("Internet desconectada.");
    }

    public void bt_resolver(View view){
        resolveDominio();
    }

    public void bt_consultarAniversariantes(View view){
        if(isInternetOn()){
            resolveDominio();
            Intent it = new Intent(this, TelaConsultarAniversariantes.class);
            Bundle b = new Bundle();
            String site = String.valueOf(endereco.getText())+":"+PORTA+WSDL;
            b.putString("site", site);
            it.putExtras(b);
            startActivity(it);
            gravarArquivoSdcar(dominio.getText().toString());
        }
        else{
            Toast.makeText(this, "Internet desconectada.", Toast.LENGTH_SHORT).show();
        }
    }

    public void bt_sair(View view){
        super.onDestroy();
        super.finish();
    }
```

Figura 42 - Classe *Activity* `GestShop.java`.

O comando `setContentView` lê o arquivo de *layout* `main.xml` e exibe esta *Activity* na tela. Este layout possui um *widget EditText* `domínio` onde o usuário poderá digitar um domínio no qual o *Web Service* poderá ser conectado, ou digitar o número de IP do

servidor, caso o usuário digite o domínio existe um *Button* `bt_resolver` que executa a função `resolveDominio()` que irá resolver o número de IP do servidor conforme o domínio digitado.

Depois de resolvido o domínio o endereço IP do servidor irá aparecer no *TextView* `endereço`. Ao carregar este *Acivity* é executada a função `carregarDominio()`, que seta o conteúdo do *TextView* domínio com o valor do ultimo domínio digitado e também através das funções `WifiState() e isInternetOn()` verifica se a Wifi está ligada e se existe uma conexão ativa com a Internet, setando valores nos *TextView* `statusWifi` e `statusConexao` para informar ao usuário.

O botão `bt_sair` fecha o aplicativo e o botão `bt_consultarAniversariantes` executa um pedido de carregamento da *Activity* `TelaConsultarAniversariantes.java` passando para esta *Activity* um parâmetro `site` que contém o endereço do *WSDL* do *Web Service* no qual o aplicativo irá se conectar.

A *Activity* `TelaConsultarAniversariantes.java,` apresentada na Figura 43, é composta por *widgets* que possibilitam o usuário configurar parâmetros para a consulta de aniversariantes e consultar uma lista de aniversariantes.

```
package br.com.gestshop.appcliente;
import java.util.ArrayList;

public class TelaConsultarAniversariantes extends Activity implements IConexao{

    DatePicker data;
    EditText nome;
    TextView site;
    ArrayList<Aniversariante> lista = new ArrayList<Aniversariante>();
    String origem;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.tela_consultar_aniversariantes);

        nome = (EditText) findViewById(R.campo.nome);
        site = (TextView) findViewById(R.id.wsdl);
        data = (DatePicker) findViewById(R.id.data);

        TelephonyManager manager = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
        origem = manager.getSimSerialNumber();

        Bundle b = getIntent().getExtras();
        site.setText(b.getString("site"));
    }

    public void bt_consultar(View view){
        consultar();
    }

    public void bt_voltar(View view){
        super.finish();
    }
```

Figura 43 - Classe *Activity* `TelaConsultarAniversariantes.java`.

O comando `setContentView` lê o arquivo de *layout* `tela_consultar_aniversariantes.xml` e exibe esta *Activity* na tela. Este *layout* possui um *widget DatePicker* `data` em que o usuário poderá informar qual data que deseja

consultar, um *EditText* `nome` para o usuário poder informar um nome para filtrar a consulta, um *TextView* `site` que apresenta o endereço do *WSDL* que será conectado, um *Button* `bt_voltar` que irá fechar a tela e um *Button* `bt_consultar` que executa a função `consultar()` que consulta a lista de aniversariantes no *Web Service*.

Ao carregar esta tela é obtido a referencia dos componentes no arquivo de *layout*, e setado o valor da variável `origem` com o número do aparelho e o valor do *TextView* `site` com o valor recuperado do parâmetro `site`.

Adicionando a biblioteca *KSOAP2* ao *Bulid Path* do projeto é possível utilizar as classes responsáveis por conectar e acessar um *Web Service* a partir do *Android*.

As Figuras 44a e 44b apresentam o método `consultar()`.

```
private void consultar() {
    try{
        SoapObject envio = new SoapObject(NAMESPACE, "buscar");
        SoapSerializationEnvelope envelope = new SoapSerializationEnvelope(SoapEnvelope.VER11);
        if(nome.getText().equals("")){
            envio.addProperty("por_nome", -1);
            envio.addProperty("nome", "todos");
        } else {
            envio.addProperty("por_nome", 0);
            envio.addProperty("nome", nome.getText()+"%");
        }
        envio.addProperty("dia", data.getDayOfMonth());
        envio.addProperty("mes", data.getMonth()+1);
        envelope.setOutputSoapObject(envio);
        HttpTransportSE androidHttpTransport = new HttpTransportSE(String.valueOf(site.getText()));
        androidHttpTransport.call("buscar", envelope);
        SoapObject resultado = (SoapObject) envelope.bodyIn;

        lista = new ArrayList<Aniversariante>();
        for (int i = 0; i < resultado.getPropertyCount(); i++) {
            SoapObject ob = (SoapObject) resultado.getProperty(i);
            Aniversariante a = new Aniversariante();
            a.setCodigo(Integer.parseInt(ob.getProperty("codigo").toString()));
            a.setNome(ob.getProperty("nome").toString());
            a.setTelefone(ob.getProperty("telefone").toString());
            a.setCelular(ob.getProperty("celular").toString());
            a.setOperadora(ob.getProperty("operadora").toString());
            a.setNascimento(ob.getProperty("nascimento").toString());
            a.setEndereco(ob.getProperty("endereco").toString());
            a.setCidade(ob.getProperty("cidade").toString());
            a.setFilial(Integer.parseInt(ob.getProperty("filial").toString()));
            a.setAno(Integer.parseInt(ob.getProperty("ano").toString()));
            a.setIdade(data.getYear() - a.getAno());
            a.setTipo(ob.getProperty("tipo").toString());
```

Figura 44a -Método que consulta a lista de aniversariantes no *Web Service*

```
            a.setContato(ob.getProperty("contato").toString());
            lista.add(a);
        }
        if (lista.isEmpty()){

            Toast.makeText(this, "Nenhum aniversariante encontrado.", Toast.LENGTH_SHORT).show();

        }else{
            Intent it = new Intent(this, TelaListarAniversariantes.class);
            Bundle b = new Bundle();

            //b.putParcelable("ListaAniversariantes", lista);
            b.putSerializable("lista", lista);
            b.putString("origem", (String) origem);
            b.putString("site", (String) site.getText());
            it.putExtras(b);

            startActivity(it);
        }

    }catch (Exception e){
        new AlertDialog.Builder(this)
        .setTitle("Erro. Verifique o status do WebService.")
        .setMessage(e.getMessage())
        .setNeutralButton("OK", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dlg, int sumthin) {
            }
        }).show();
    }
}
```

Figura 44b - Método que consulta a lista de aniversariantes no *Web Service*

Neste método é instanciado um objeto `org.ksoap2.serialization.SoapObject` chamado `envio`, passando como parâmetros para o construtor, o *namespace* e a operação que será utilizada. Um envelope de transporte `org.ksoap2.serialization.SoapSerializationEnvelope`, e utilizado o método `setOutputSoapObject`, passando como parâmetro o objeto envio. Um objeto de transporte `org.ksoap2.serialization.HttpTransportSE`, passando como parâmetro ao construtor, endereço do *WSDL* do *WebService*, ou seja, o valor do *TextView* site. Também é utilizado o método `call` do objeto de transporte, passando como parâmetros, o nome da operação e o envelope que irá conter o retorno do *Web Service*. E por fim, é instanciado um novo objeto `org.ksoap2.serialization.SoapObject`, chamado `resultado`, que será atribuído o retorno do envelope.

Com o `resultado`, o retorno é explorado através dos métodos `getPropertyCount()` e `getProperty(0)`, para popular a lista de objetos `ListaAniversariantes.java` chamada `lista`.

Após popular a `lista`, é carregado um *ListActivity* chamado `TelaListarAniversariantes.java`, que exibe os dados dos aniversariantes na tela, passando os parâmetros origem, site e a lista. A Figura 45 apresenta o conteúdo da classe `TelaListarAniversariantes.java`.

```
package br.com.gestshop.appcliente;
import java.text.ParseException;
public class TelaListarAniversariantes extends ListActivity implements IConexao{
    private static final int TODOS = Menu.FIRST;
    private static final int PENDENTE = Menu.FIRST+1;
    private static final int REALIZADO = Menu.FIRST+2;
    ArrayList<Aniversariante> lista;
    ArrayList<Aniversariante> listaOriginal;
    Aniversariante a = new Aniversariante();
    AniversarianteAdapter adapter;
    String site;
    String destino;
    String origem;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Bundle b = getIntent().getExtras();
        site = b.getString("site");
        origem = b.getString("origem");
        lista = (ArrayList<Aniversariante>) getIntent().getSerializableExtra("lista");

        listaOriginal = new ArrayList<Aniversariante>();
        for(Aniversariante an : lista){
            Aniversariante ob = new Aniversariante();
            ob.setCodigo(an.getCodigo());
            //também é setado o valor dos outros atributos
            listaOriginal.add(ob);
        }
        adapter = new AniversarianteAdapter(this, lista);
        setListAdapter(adapter);
        this.setTitle(contarAniversariantes(lista));
        repopularLista("PENDENTE");
    }
```

Figura 45 - Classe *Activity* TelaListarAniversariantes.java

Para que seja possível exibir todos os valores dos atributos de um objeto `Aniversariante.java` em um item da lista, foi necessário criar um *adapter* do tipo da classe `Aniversariante.java`, chamado `AniversarianteAdapter.java`. Ao carregar esta tela, um objeto `AniversarianteAdapter` com o nome `adapter` é instanciado, passando como parâmetros para o construtor o contexo e a lista de objetos do tipo da classe `ListaAniversariantes.java`. Após é utilizado o método `setListAdapter` para setar o objeto *adapter* no *ListActivity*. Também é realizado uma cópia da nossa lista de aniversariantes chamada `listaOriginal`, utilizada no método `repopularLista()`, que executa um filtro nos itens exibidos na tela com base nos parâmetros “`PENDENTE`”, “`REALIZADO`” ou “`TODOS`”.

Outra função executada é a `contarAniversariantes()`, que verifica a quantidade de aniversariantes da lista e a quantidade em que ainda não foi realizado o contato, para setar este valores no titulo do *ListActivity*.

Para que a aplicação faça algo quando clicar em um item da lista, foi implementado o método `onListItemClick`, A Figura 46 apresenta o conteúdo deste método.

```
protected void onListItemClick(ListView l, View v, int position, long id) {
    a = lista.get(position);
    if(a.getContato().equals("NAO REALIZADO")){
        LayoutInflater inflater = (LayoutInflater) this
                .getSystemService(LAYOUT_INFLATER_SERVICE);
        View layout = inflater.inflate(R.layout.dialog,
                (ViewGroup) findViewById(R.id.layout_root));

        TextView txt_nome = (TextView) layout.findViewById(R.dialog.nome);
        txt_nome.setText("Nome: " + a.getNome());
        TextView txt_telefone = (TextView) layout.findViewById(R.dialog.telefone);
        txt_telefone.setText("Fixo: " + a.getTelefone());
        TextView txt_celular = (TextView) layout.findViewById(R.dialog.celular);
        txt_celular.setText("Celular: " + a.getCelular());

        final CharSequence[] items = { "Ligar para Fixo", "Ligar para Celular", "Enviar SMS" };
        AlertDialog.Builder alert = new AlertDialog.Builder(this);
        alert.setTitle("Escolha uma opção de contato");
        alert.setSingleChoiceItems(items, -1,
                new DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog,
                            int item) {
                        Intent dial = null;
                        if (items[item] == "Ligar para Fixo") {
                            dial = new Intent(
                                    android.content.Intent.ACTION_CALL,
                                    Uri.parse("tel: "
                                            + a.getTelefone()));
                            startActivity(Intent.createChooser(
                                    dial, "Discando...."));

                            destino = a.getTelefone();
                            registrarContato();
                        } else
                        if (items[item] == "Ligar para Celular") {
                            dial = new Intent(
                                    android.content.Intent.ACTION_CALL,
                                    Uri.parse("tel: "
                                            + a.getCelular()));
                            startActivity(Intent.createChooser(
                                    dial, "Discando...."));

                            destino = a.getCelular();
                            registrarContato();
                        } else
                        if (items[item] == "Enviar SMS") {
                            enviarSMS(a.getCelular().toString());
                            destino = a.getCelular();
                            registrarContato();
                        }

                    }

                });
        alert.show();
    }
    else{
        new AlertDialog.Builder(this)
        .setTitle("Informação")
        .setMessage("O contato com este cliente ja foi realizado. ")
        .setNeutralButton("OK", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dlg, int sumthin) {
            }
        }).show();
    }
}
```

Figura 46 - Método onListItemClick().

O método `onListItemClick` mostra na tela um `AlertDialog.Builder` chamado `alert`, que mostra uma lista com três opções de contato: `Ligar para Fixo`, `Ligar para Celular` e `Enviar SMS`. As duas primeiras opções iniciam uma chamada para o numero selecionado. A segunda carrega uma nova *Activity* chamada

`TelaEnviarSMS.java`, passando um parâmetro com o numero de celular do aniversariante, e que é utilizada para enviar uma mensagem. A Figura 47 apresenta o conteúdo da `Activity TelaEnviarSMS.java`.

```
package br.com.gestshop.appcliente;

import java.io.BufferedReader;

public class TelaEnviarSMS extends Activity {
    EditText mensagem;
    String destino;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        this.setContentView(R.layout.tela_enviar_sms);

        mensagem = (EditText) findViewById(R.campo.mensagem);
        carregarMensagem();

        Bundle b = getIntent().getExtras();
        destino = b.getString("destino");
    }

    public void bt_enviar(View view){
        SmsManager smsM = SmsManager.getDefault();
        smsM.sendTextMessage(destino, null, mensagem.getText().toString(), null, null);
        gravarArquivoSdcar(mensagem.getText().toString());
        Toast.makeText(this, "E-mail enviado.", Toast.LENGTH_SHORT).show();
    }

    public void bt_voltar(View view){
        gravarArquivoSdcar(mensagem.getText().toString());
        super.finish();
    }
```

Figura 47 - Classe *Activity* `TelaEnviarSMS.java`

A *Activity* `TelaEnviarSMS.java` é composta por *widgets* que possibilitam o usuário digitar uma mensagem e enviar ou voltar a lista de aniversariantes.

O comando `setContentView` ir lê o arquivo de *layout* `tela_enviar_sms.xml` e exibe esta *Activity* na tela. Este *layout* possui um *widget EditText* chamado `mensagem` para o usuário poder digitar uma mensagem, um *Button* chamado `bt_voltar` que irá fechar a tela, e um *Button* chamado `bt_enviar` que utiliza a classe `android.telephony.SmsManager` para enviar a mensagem SMS.

4 RESULTADOS E DISCUSSÕES

Neste capitulo serão apresentados os resultados alcançados com o desenvolvimento do *Web Service* e do aplicativo cliente para *Android*.

4.1. SISTEMA WEBSERVICE

Para apresentar o resultado do *Web Service* utilizou-se o teste que existe dentro do console do GlassFish 3.1. A Figura 48 apresenta a pagina de testes do *Web Service*, utilizando o método `buscar`.

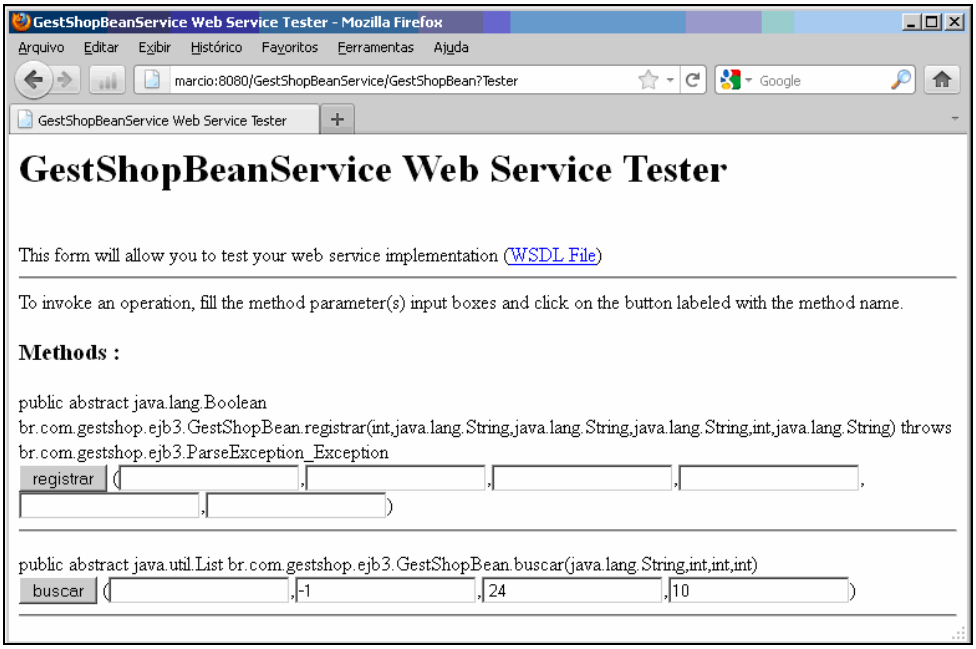
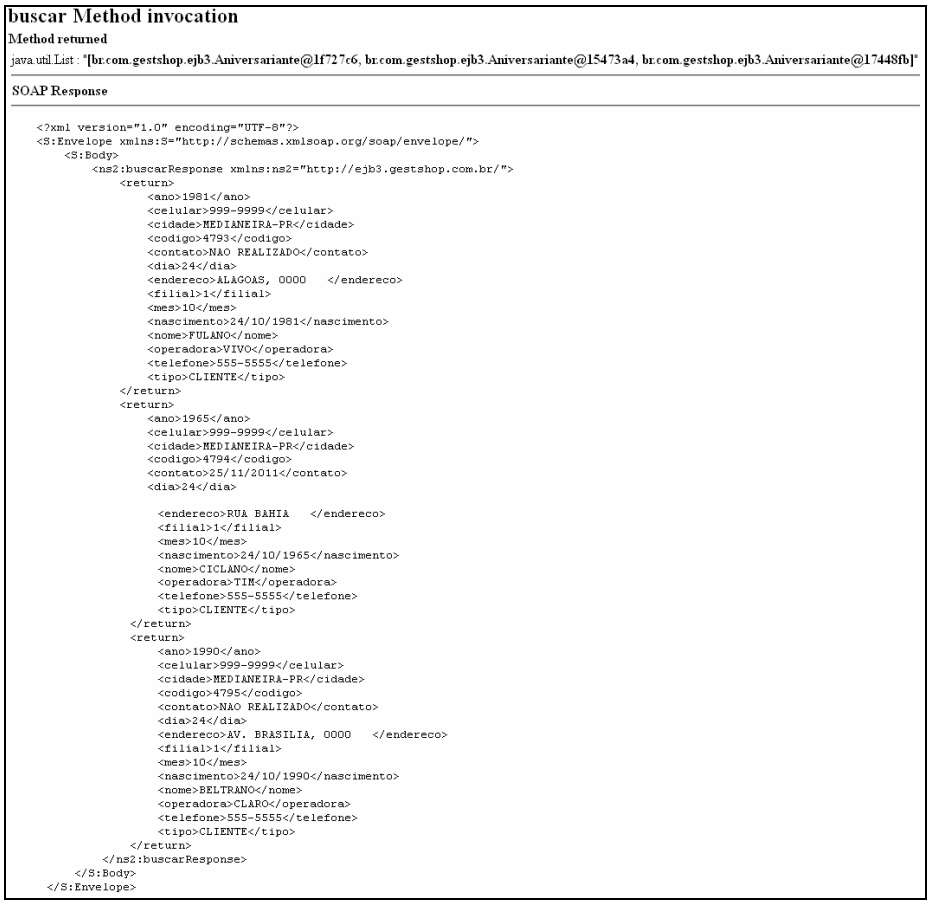


Figura 48 - Testando o Web Service pelo Console de administração do GlassFish 3.1.

São consultados os aniversariantes, sem informar um nome de cliente como parâmetro, que fazem aniversário no dia 24 do mês 10. O resultado desta consulta é apresentado na Figura 49.



buscar Method invocation

Method returned

java.util.List : "[br.com.gestshop.ejb3.Aniversariante@1f727c6, br.com.gestshop.ejb3.Aniversariante@15473a4, br.com.gestshop.ejb3.Aniversariante@17448fb]"

SOAP Response

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:buscarResponse xmlns:ns2="http://ejb3.gestshop.com.br/">
            <return>
                <ano>1981</ano>
                <celular>999-9999</celular>
                <cidade>MEDIANEIRA-PR</cidade>
                <codigo>4793</codigo>
                <contato>NAO REALIZADO</contato>
                <dia>24</dia>
                <endereco>ALAGOAS, 0000   </endereco>
                <filial>1</filial>
                <mes>10</mes>
                <nascimento>24/10/1981</nascimento>
                <nome>FULANO</nome>
                <operadora>VIVO</operadora>
                <telefone>555-5555</telefone>
                <tipo>CLIENTE</tipo>
            </return>
            <return>
                <ano>1965</ano>
                <celular>999-9999</celular>
                <cidade>MEDIANEIRA-PR</cidade>
                <codigo>4794</codigo>
                <contato>25/11/2011</contato>
                <dia>24</dia>

                 <endereco>RUA BAHIA   </endereco>
                 <filial>1</filial>
                 <mes>10</mes>
                 <nascimento>24/10/1965</nascimento>
                 <nome>CICLANO</nome>
                 <operadora>TIM</operadora>
                 <telefone>555-5555</telefone>
                 <tipo>CLIENTE</tipo>
             </return>
             <return>
                 <ano>1990</ano>
                 <celular>999-9999</celular>
                 <cidade>MEDIANEIRA-PR</cidade>
                 <codigo>4795</codigo>
                 <contato>NAO REALIZADO</contato>
                 <dia>24</dia>
                 <endereco>AV. BRASILIA, 0000   </endereco>
                 <filial>1</filial>
                 <mes>10</mes>
                 <nascimento>24/10/1990</nascimento>
                 <nome>BELTRANO</nome>
                 <operadora>CLARO</operadora>
                 <telefone>555-5555</telefone>
                 <tipo>CLIENTE</tipo>
             </return>
        </ns2:buscarResponse>
    </S:Body>
</S:Envelope>
```

Figura 49 - Resultado do teste do *Web Service* realizado pelo Console de administração do GlassFish 3.1.

O resultado obtido é uma lista com três objetos da classe `Aniversariante`.

4.2. APLICATIVO ANDROID

A tela apresentada ao abrir o aplicativo tem uma interface com um campo para digitar o domínio que o *Web Service* está hospedado, e um botão para resolver este domínio, descobrindo assim o endereço IP do servidor. Abaixo são exibidos botões, para sair da aplicação e para abrir a tela de consulta de aniversariantes, a Figura 50 apresenta a tela inicial do aplicativo.

Figura 50 - Tela inicial do aplicativo Android.

Selecionado o botão “Consultar Aniversariantes”, é apresentada uma nova tela onde o usuário pode informar os parâmetros de consulta da lista de aniversariantes. Primeiro a data que deseja consultar e depois um filtro, opcional, para usuário poder digitar um nome. Também exibe botões, com opção de voltar e consultar a lista com base nos parâmetros informados. A Figura 51 apresenta a tela de consulta de aniversariantes.

Figura 51 - Tela Consultar Aniversariantes.

Selecionado o botão “Consultar”, o aplicativo se conecta novamente ao *Web Service* para utilizar a operação de consulta, e exibe uma nova tela com uma lista de aniversariantes retornada. O *screenshot* da lista de aniversariantes está demonstrada na Figura 52.

Figura 52 - Tela com a lista de aniversariantes.

Esta tela ainda oferece opções de *menu*, onde o usuário pode selecionar se deseja visualizar, somente os aniversariantes sem contato realizado, com contato realizado ou visualizar todos os aniversariantes. As opções do *menu* são demonstradas na Figura 53.

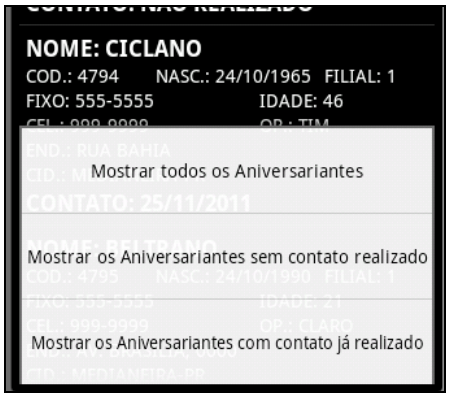


Figura 53 - Opções de filtros para a lista de aniversariantes.

Selecionado um aniversariante, é apresentada uma lista de opções de contato. As opções de contato são demonstradas na Figura 54.

Figura 54 - Opções de contato com o aniversariante.

Selecionado uma das duas primeiras opções, o aparelho inicia uma discagem para o numero selecionado. A Figura 55 apresentada a tela de discagem do *Android*.

Figura 55 - Discando no Android.

Selecionando a terceira opção, uma nova tela é exibida na tela com um campo para digitar uma mensagem, e botões de para voltar e enviar a mensagem SMS. A Figura 56 apresenta a tela para envio de SMS.

Figura 56 - Tela Enviar SMS.

Seja qual for a opção selecionada, ao terminar o contato, o sistema exibe uma caixa de dialogo perguntando ao usuário se o mesmo deseja registrar o contato. A Figura 57 apresenta a caixa de dialogo.

Figura 57 - Dialogo apresentado com opção de registro do contato.

Se o usuário selecionar a opção “SIM”, o aplicativo conecta novamente ao *Web Service*, para utilizar a operação que registra os dados do contato no banco de dados do sistema *desktop*, e atualiza o campo “CONTATO”, exibido no item da lista, com a data em que foi realizado o contato.

Após o procedimento de contato, o usuário poderá iniciar um novo contato ou fechar o aplicativo.

5 CONSIDERAÇÕES FINAIS

Neste capítulo serão apresentadas as considerações finais deste trabalho de diplomação.

5.1. CONCLUSÃO

Através deste projeto concluiu-se que, é possível integrar sistemas *desktop*, adquiridos há vários anos, com novas tecnologias de sistemas de informação, como é o caso de dispositivos movéis, através de uma solução que se encaixa na infraestrutura tecnológica atualmente disponível em pequenos negócios e com baixo custo.

Foi necessária a busca de informações de uma variedade de tecnologias empregadas no projeto. Gastou-se mais tempo em pesquisa de referencial teórico, para entender melhor sobre o funcionamento das tecnologias, e a procura de técnicas de desenvolvimento, do que na codificação e testes. Todas as tecnologias utilizadas atenderam as necessidades do projeto de forma funcional e se mostraram de fácil entendimento, principalmente as que envolviam a construção do aplicativo em *Android*, já o *Web Service* exigiu uma busca maior de conhecimento sobre servidores de aplicação.

5.2. TRABALHOS FUTUROS/CONTINUAÇÃO DO TRABALHO

Este projeto terá continuidade com o estudo de melhorias na interface do aplicativo *Android* e alternativas para verificar a disponibilidade do *Web Service* de uma forma mais rápida, visto que quando o mesmo encontra-se indisponível a resposta acaba demorando muito para retornar. E também o estudo de outras alternativas para a aplicação cliente, visto que o aplicativo cliente desenvolvido somente funciona em dispositivos móveis com a plataforma *Android*.

Outro estudo que pode ser realizado é a pesquisa comparativa de outros servidores de aplicação, ou mesmo, servidores *Web*, que possam atender o objetivo proposto consumindo menos recursos do servidor e com tempos de resposta mais satisfatórios. Apesar da solução utilizada já atender bem as necessidades propostas.

REFERENCIAS BIBLIOGRÁFICAS

BONFANDINI, Eduardo. **Avaliação heurística comparativa de aplicativos desktop e web**. Disponível em <http://trialforce.nostaljia.eng.br/wp-content/uploads/others/Avaliacao_heuristica_comparativa_de_aplicativos_desktop_e_web.pdf > Acessado em 26/08/2011.

BORRIELO, Daniela. GOMES, Rafael V. A.. JUNIOR, Antonio P. Castro. SANTOS, Roberto F. T. **Análise comparativa entre servidores de aplicação livres que seguem a plataforma J2EE**. Anais da 58ª Reunião Anual da SBPC, Florianópolis, SC, 2006. Disponível em <http://www.sbpcnet.org.br/livro/58ra/SENIOR/RESUMOS/resumo_394.html> Acessado em 08/11/2011.

CAELUM. **FJ-21: Java para Desenvolvimento Web**. Apostilas gratuitas. Disponível em < http://www.caelum.com.br/download/caelum-java-web-fj21.pdf> Acessado em 18/10/2011.

GOMES, Handerson Ferreira. **A plataforma J2EE**. Webinsider, 2000. Disponível em < http://webinsider.uol.com.br/2000/12/05/a-plataforma-j2ee/> Acessado em 08/11/2011.

GOOGLE a. Android**. Ksoap2 Android**. Disponível em <http://code.google.com/p/ksoap2-android/ > Acessado em 19/10/2011.

GOOGLE b. Google Code. **Projetos do Google para o Android**. Disponível em <http://code.google.com/intl/pt-BR/android/> Acessado em 09/08/2011.

KALIN, Martin. **Java Web Services: Implementando.** Rio de Janeiro, RJ. Ed. Altabooks, 2010. ISBN 9788576084242.

LEAL, Nelson G. de Vasconcelos. **Adapter Eficiente no Android**. Publicado em 24/03/2011. Debug is on the table. Disponível em < http://nglauber.blogspot.com/2011/03/adapter-eficiente-no-android.html> Acessado em 24/10/2011.

LEAL, Nelson G. de Vasconcelos. **Android: Passando objetos em Intents**. Publicado em 11/01/2010. Debug is on the table. Disponível em <http://nglauber.blogspot.com/2010_01_01_archive.html> Acessado em 24/10/2011.

LECHETA, Ricardo R. . **Google Android: Aprenda a criar aplicações.** São Paulo, SP : Novatec, 2009. ISBN. 8575221868 / 9788575221860.

MINISTERIO DAS COMUNICAÇÕES. **Um plano nacional para banda Larga: O Brasil em alta velocidade**. Programa Nacional de Banda Larga. Disponível em < http://www.mc.gov.br/images/pnbl/o-brasil-em-alta-velocidade1.pdf> Acessado em 14/10/2011.

NETO, Antônio Marin. **Android: Trocando de tela e passando parâmetros**. Neto Marin Mobility blog. Disponível em < http://netomarin.com/blog/20101018/android-trocando-de-tela-e-passando-parametros/> Acessado em 19/10/2011.

OLHAR DIGITAL. **Provedoras de banda larga podem oferecer só 10 porcento da velocidade contratada. E você assinou este contrato**. Central de vídeos. Disponível em <http://olhardigital.uol.com.br/produtos/central_de_videos/provedoras-de-banda-larga-podem-oferecer-so-10-porcento-da-velocidade-contratada.-e-voce-assinou-este-contrato...> Acessado em 14/10/2011.

OLIVEIRA, Eric C. M. . **Conhecendo a plataforma J2EE – um breve overview**. Linha de Código, 2004. Disponível em <http://www.linhadecodigo.com.br/Artigo.aspx?id=333> Acessado em 08/11/2011.

POTTS, Stephen. KOPACK, Mike. **Aprenda Web Services em 24 horas**. Tradução de Marcos Vieira. Ed. Campus, 2003. ISBN : 853521321X.

RABELLO, Ramon Ribeiro. **Utilizando Web Services em Android**. Revista WebMobile, ed. 23. Publicado em 03/06/2009. Disponível em <http://www.cesar.org.br/site/utilizando-web-services-em-android/ > Acessado em 18/10/2011.

SILVA, Lino Sarlo da. **VPN: Virtual Private Network**. São Paulo, SP : Novatec, 2003. ISBN. 8575220330.

SILVA, Luciano Alves da. **Programando passo a passo**. Apostila de Android. Disponível em <http://www.portalandroid.org/comunidade/viewtopic.php?f=7&t=2528> Acessado em 19/10/2011.

TERRA. **Estudo: no Brasil, banda larga segue com velocidade "média"**. Noticias Tecnologia. Publicado em 04/04/2011. Disponível em <http://tecnologia.terra.com.br/noticias/0,,OI5045874-EI12884,00-Estudo+no+Brasil+banda+larga+segue+com+velocidade+media.html> Acessado em 14/10/2011.

VASCOLCELOS a, Marcos Antonio. **Android – Activity**. Publicado em 03/02/2011. Mark Vasconcelos Creative Solutions. Disponível em <http://markytechs.wordpress.com/2011/02/03/android-activity/> Acessado em 19/10/2011.

VASCOLCELOS b, Marcos Antonio. **Android – Estrutura e organização da aplicação**. Mark Vasconcelos Creative Solutions. Publicado em 24/01/2011. Disponível em <http://markytechs.wordpress.com/2011/01/24/android-estrutura-e-organizacao-da-aplicacao/> Acessado em 19/10/2011.

VOHRA, Deepak. **Acessando um Serviço da Web em JAX-WS a partir do Android**. IBM developerWorks. Publicado em 01/07/2011. Disponível em <http://www.ibm.com/developerworks/br/library/ws-android/index.html?ca=drs- > Acessado em 18/10/2011.