INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA SUL-RIO-GRANDENSE - CÂMPUS PASSO FUNDO
CURSO DE TECNOLOGIA EM SISTEMAS PARA INTERNET

MICHEL WILLIAM HUFF

SISTEMA DE GESTÃO DE OCORRÊNCIAS OPERACIONAIS PARA EMPRESAS DE SANEAMENTO

Jorge Luís Boeira Bavaresco

PASSO FUNDO
2016

MICHEL WILLIAM HUFF

SISTEMA DE GESTÃO DE OCORRÊNCIAS OPERACIONAIS PARA EMPRESAS DE SANEAMENTO

Monografia apresentada ao Curso de Tecnologia em Sistemas para Internet do Instituto Federal Sul-rio-grandense, Campus Passo Fundo, como requisito parcial para a obtenção do título de Tecnólogo em Sistemas para Internet.

Orientador: Jorge Luis Boeira Bavaresco

PASSO FUNDO

2016

MICHEL WILLIAM HUFF

SISTEMA DE GESTÃO DE OCORRÊNCIAS OPERACIONAIS PARA EMPRESAS DE SANEAMENTO

Trabalho de Conclusão de Curso aprovado em ____/____/____ como requisito parcial para a obtenção do título de Tecnólogo em Sistemas para Internet

Banca Examinadora:

_____

Jorge Luis Boeira Bavaresco

_____

Carmen Vera Scorsatto

_____

Josué Toebe

_____

Adilso Nunes de Souza

PASSO FUNDO

2016

AGRADECIMENTOS

Agradeço aos meus pais, Dirson e Terezinha, pela compreensão, pelo companheirismo, pela força e por me apoiarem incondicionalmente, não só no período de estudos, mas em toda a vida.

Agradeço a minha namorada Michele pelo apoio, pelos conselhos e pelo carinho (e por ter me aturado neste período tão corrido).

Agradecer, e muito, ao professor Jorge Luís Boeira Bavaresco, que investiu seu tempo e seu conhecimento na busca de tornar este trabalho possível e não mediu esforços para me auxiliar na construção do presente projeto. Da mesma forma, agradecer ao professor Jair José Ferronato, por ter apoiado o desenvolvimento do trabalho e por ter auxiliado em sua revisão.

Por fim, agradecer aos colegas do IFSUL, aos demais professores e servidores do campus e a todas as pessoas que contribuíram para que este trabalho fosse possível e agradecer sempre a Deus, pela proteção e pelo amparo.

RESUMO

A tecnologia imersa no cotidiano das pessoas contribui para que permaneçam cada vez mais tempo conectadas à internet. Este estado de conexão pode ser uma oportunidade para que empresas estreitem os laços de comunicação que possuem com seus clientes ou usuários. Em se tratando de empresas de saneamento, vazamentos e entupimentos de redes de água e esgoto e problemas com valas em vias públicas são ocorrências frequentemente presenciadas pelo público, sendo de vital importância a existência de um canal de comunicação para informá-las. O presente estudo foi desenvolvido com o objetivo de projetar e implementar uma aplicação acessível por navegador de internet que permita aos clientes de empresas de saneamento interagir com as mesmas informando ocorrências operacionais. A aplicação foi desenvolvida sob plataforma Java EE, com persistência de dados executada pela API JPA, disponibilização de recursos por meio de containers EJB e interface com *layout* responsivo implementada com auxílio das bibliotecas PrimeFaces e BootsFaces. Como resultado, a aplicação desenvolvida visou instituir um canal de comunicação alternativo entre empresas de saneamento e o usuário, permitindo maior fluidez na troca de informações.

Palavras-chave: Aplicação. Saneamento. Java. Responsivo.

ABSTRACT

The technology immersed in the people's daily lives contributes to they remain more time connected to the Internet. This state of connectivity can be an opportunity for companies to narrow the communication ties they have with their customers or users. In the case of sanitation companies, leaks and clogging of water networks and sewage and problems with ditches on public roads are often occurrences witnessed by the public, it has vital importance that there is a communication channel to inform them. This study was developed with the objective to project and implement an application for web browser that allows customers of sanitation companies services to interact with them and register operacional occurencies. The application was developed in Java EE platform with data persistence performed by JPA API, available resources through EJB containers and responsive *layout* interface implemented with the help of PrimeFaces and BootsFaces libraries. As a result, the developed application instituted an alternative communication channel between the user and sanitation companies, allowing greater fluidity in the exchange of information.

Keywords: Software. Responsive. Java. Sanitation

LISTA DE FIGURAS

LISTA DE ABREVIAÇÕES E DE SIGLAS

API – *Application Programming Interface*

CSS3 – *Cascading Style Sheet 3*

DDL – *Data Definition Language*

DML – *Data Manipulation Language*

EJB – *Enterprise Java Beans*

GPS – *Global Positioning System*

HTML5 – *Hypertext Markup Language 5*

IFSUL – Instituto Federal Sul-rio-grandense

JAVA EE – *Java Enterprise Edition*

JAVA SE – *Java Standard Edition*

JDBC - *Java Database Connectivity*

JSF – *JavaServer Faces*

JPA – *Java Persistence API*

JSON – *JavaScript Object Notation*

JTA – *Java Transaction API*

JVM – *Java Virtual Machine*

PDL – *Page Description Language*

REST - *Representational State Transfer*

SGBD – Sistema de Gerenciamento de Banco de Dados

SQL – *Structured Query Language*

SOAP - *Simple Object Access Protocol*

SSL – *Secure Socket Layer*

TCC – Trabalho de Conclusão de Curso

UML – *Unified Modeling Language*

URL - *Uniform Resource Locator*

XHTML – Extensible *Hypertext Markup Language*

XML – *Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

A tecnologia presente no cotidiano das pessoas contribui para que permaneçam cada vez mais tempo conectadas à internet. Conectada à rede mundial de computadores, uma pessoa pode utilizar suas aplicações preferidas em diferentes lugares, manipulando seus dados pessoais e executando tarefas pertinentes ao seu cotidiano. Este estado de conexão permitiu grande ascensão dos dispositivos móveis ao mercado, onde portar um exemplar pode significar variadas opções para resolução de situações rotineiras, que vão de uma simples calculadora a aplicativos de controle bancário ou até de automação residencial.

Este fenômeno motivou empresas a desenvolverem canais de comunicação por meio de aplicações e aplicativos, potencializando sua relação para com seus usuários ou clientes. O presente trabalho propõe o desenvolvimento de uma aplicação com o objetivo de constituir um canal de comunicação entre o usuário e empresas de saneamento. Esta aplicação visa contribuir com a agilidade no fornecimento de informações relacionadas a vazamentos de redes de água ou esgoto e situações relacionadas a valas abertas em vias públicas. A adoção deste tema tem por sua importância contribuir com a redução do desperdício de água e contaminação do meio ambiente com resíduos efluentes, e otimizar o tempo de reparo de valas abertas em vias públicas, oriundas da manutenção dos sistema de abastecimento de água e coleta de esgoto.

Para o desenvolvimento deste trabalho foram realizados estudos sobre as tecnologias a serem utilizadas, bem como analisadas as opções que melhor contribuiriam com os objetivos do projeto. Em seguida, foi criada uma modelagem do sistema para a visualização da estrutura a ser implementada, levando em consideração os requisitos levantados e as necessidades apontadas pelos requisitos. Por fim, foi desenvolvida uma aplicação web com as características da modelagem, visando à satisfação dos requisitos.

1.2 OBJETIVOS

Nesta sessão serão descritos os objetivos gerais e específicos do presente trabalho.

1.2.1 Objetivo geral

Desenvolvimento de aplicação com *layout* responsivo para a gestão de ocorrências operacionais em empresas de saneamento.

1.2.2 Objetivos específicos

- Modelar uma aplicação, acessível por navegador de internet, que permita ao usuário registrar ocorrências operacionais de seu cotidiano.
- Permitir maior controle por parte das empresas de saneamento das ocorrências registradas, sendo possível definir prioridades nos planos de execução.
- Alimentar bases de dados com informações relevantes a respeito das solicitações de usuários, permitindo maior agilidade na execução de ações correlacionadas.
- Desenvolver uma aplicação em tecnologia compatível com os padrões atuais utilizados em dispositivos de acesso à internet.

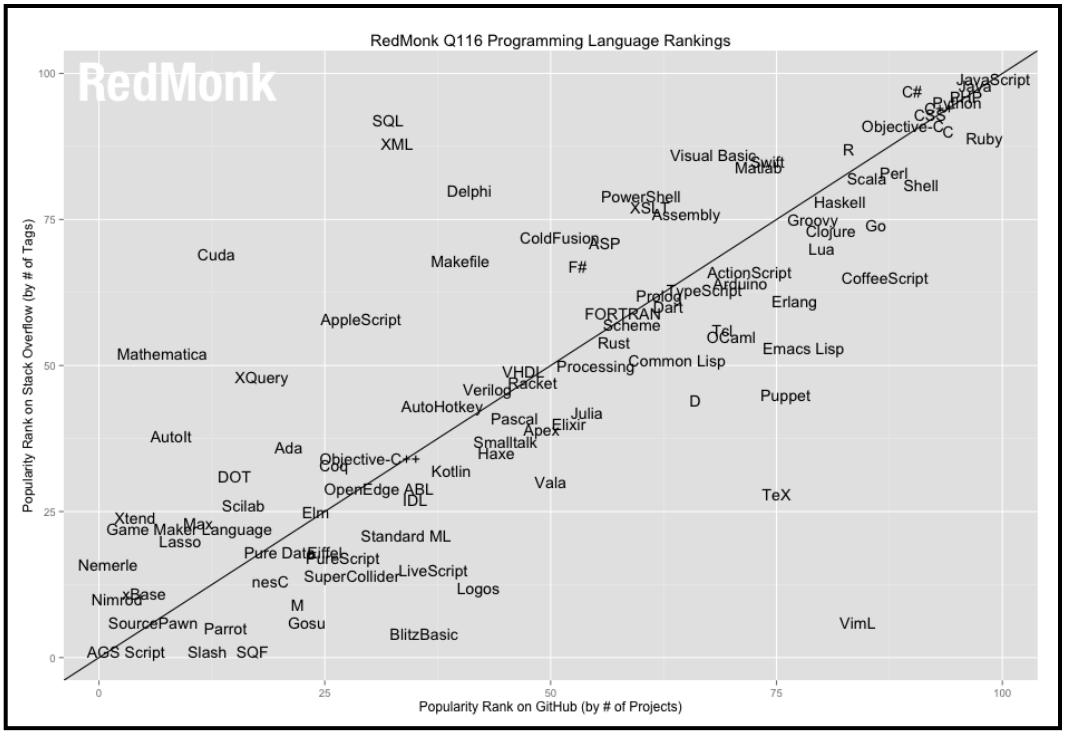
2 REFERENCIAL TEÓRICO

2.1 JAVA

A plataforma Java, que obteve sucesso a partir do ano de 1995, é uma tecnologia utilizada para o desenvolvimento de softwares, páginas web, jogos e aplicativos para dispositivos móveis tendo como um de seus principais diferenciais a portabilidade (ORACLE, 2015) (ARNOLD e GOSLING, 2009).

Conforme o ranking RedMonk (REDMONK, 2016), que mensura a utilização das linguagens de programação para o desenvolvimento de aplicações, a linguagem Java foi a segunda linguagem de programação mais utilizada para projetos até a definição do indicador, em janeiro de 2016. O Ranking RedMonk está ilustrado na Figura 1:

Figura 1: Ranking RedMonk

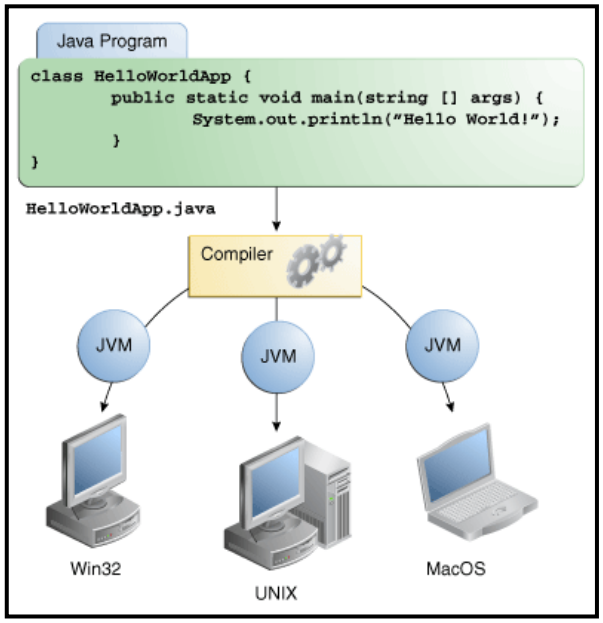


Fonte: REDMONK, 2016

O código fonte Java não é diretamente compilado para a execução do sistema operacional mas sim transformado para a execução em máquina abstrata, no caso, a máquina virtual do Java. Esse código específico recebe o nome de *bytecodes* e

pode ser executado em qualquer máquina capaz de executar a máquina virtual do Java. Essa máquina virtual, conhecida como de JVM, provê acesso a um sistema de execução que disponibiliza acesso à própria máquina e a componentes externos, tais como os fluxos (*stream*) de entrada e saída de dados (ARNOLD e GOSLING, 2009). A Figura 2 apresenta o processo de compilação e execução dos códigos Java:

Figura 2: Processo de Compilação e Execução de códigos Java



Fonte: ORACLE, 2015.

Na linguagem Java, os programas são gerados com base em classes. Essas classes contém características que buscam refletir a realidade que se quer representar. A essas características dá-se o nome de atributos. Também são definidas ações que definem o comportamento da classe ou de determinado atributo, que são conhecidas como métodos. Os métodos também podem ser utilizados para modificar ou resgatar valores atribuídos aos atributos (ARNOLD e GOSLING, 2009).

Forte característica do Java, a presença de modificadores de acesso é de fundamental importância para a implementação de mecanismos de segurança para classes, atributos e métodos. Esses modificadores definem o nível de acesso a determinado recurso, que pode ocorrer somente na própria classe, quando implementado o modificador de acesso *private*, entre classes que possuam relacionamento de herança, quando implementado o modificador *protected* ou entre

classes de um mesmo pacote ou mesmo entre classes de pacotes distintos, quando utilizado o modificador de acesso *public* (ARNOLD e GOSLING, 2009). A Figura 3 apresenta um exemplo de modificador de acesso *public:*

Figura 3: Exemplo de código Java com modificador de acesso *public*

```
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string.
    }

}
```

Fonte: ORACLE, 2015.

2.2 JAVA ENTERPRISE EDITION

O Java EE foi desenvolvido para atender necessidades que foram expostas principalmente por organizações empresariais, nos meados de 1999. Essas necessidades envolviam grandes dificuldades em desenvolver novos sistemas que ainda fossem compatíveis a outras aplicações, desenvolvidas em outras linguagens ou que utilizassem outros protocolos de comunicação. No mesmo contexto, pode-se citar a necessidade de se adaptar as aplicações às novas tecnologias que surgiram nesse período, tais como os *web services*, SOAP e RESTful (GONCALVES, 2013).

O Java EE funciona disponibilizando recursos por meio de *containers* onde um *container* é um ambiente de tempo de execução que fornece serviços tais como controle de concorrência, gerenciamento de ciclo de vida e injeção de dependência. Os *containers* podem ser classificados em quatro grupos, sendo eles *Applet containers*, *application client container*, *web container* e *EJB container* (GONCALVES, 2013). Já os serviços são inúmeros, sendo alguns deles:

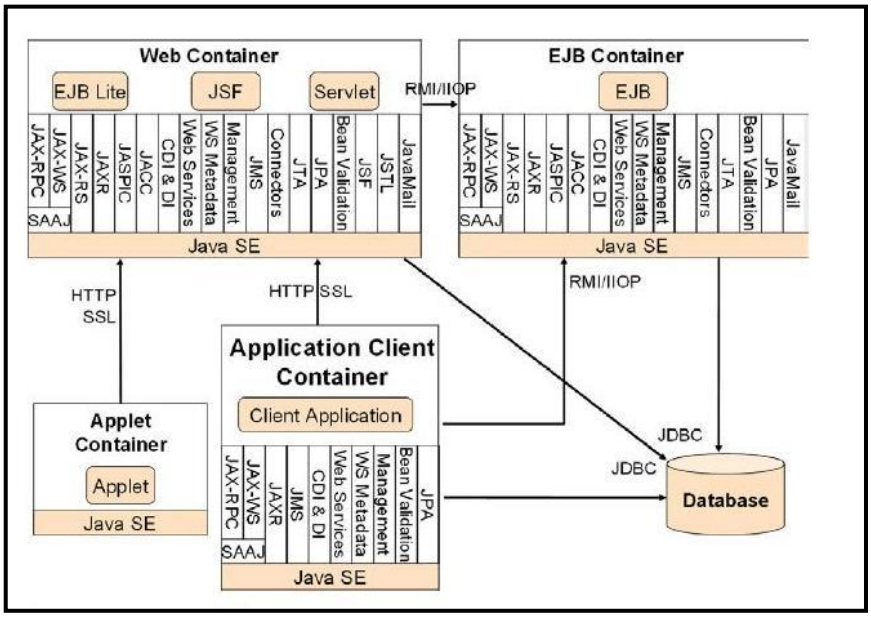
- *Java Transaction API*: Serviço para gerência de transações que ocorrem no *container*, onde este cria uma interface entre a transação e o mecanismo gestor. (GONCALVES, 2013).
- *Java Persistence API*: é responsável pelo mapeamento objeto-relacional de objetos das classes Java, onde permite consultas por meio da

linguagem Java Persistence Query Language (JPQL) (GONCALVES, 2013).

- *Validation*: a *Bean Validation* permite a utilização de mecanismos de validação de dados em classes, métodos e atributos (GONCALVES, 2013).
- *Java Message Service*: é um serviço de troca de mensagens de forma assíncrona entre diferentes componentes da aplicação (GONCALVES, 2013).
- *JavaMail*: permite o envio de e-mails pelas aplicações (GONCALVES, 2013).
- Processadores de XML e JSON: permitem a manipulação e utilização de dados em formatos JSON ou XML (GONCALVES, 2013).
- *Web services*: o Java EE implementa por padrão suporte para web services SOAP e RESTful. Esse suporte compreende manipular dados XML, JSON e informações do protocolo HTTP (GONCALVES, 2013).

Na Figura 4 é apresentada a estrutura de serviços disponibilizada pelos *containers* EJB:

Figura 4: Serviços disponibilizados pelos *containers* Java EE



Fonte: GONCALVES, 2013, p. 5.

2.3 JAVASERVER FACES

O JavaServer Faces é um *framework* baseado em componentes, utilizado para a construção de “RIAs” – *Rich Internet Application* -, interfaces ricas principalmente em recursos visuais, que implementam interatividade às aplicações *web* e as tornam intuitivas (GEARY e HORSTMANN, 2010). O JavaServer Faces foi desenvolvido com base no modelo Java Swing e em *frameworks* de desenvolvimento de interfaces, o que permite ao desenvolvedor utilizar-se da programação de componentes, eventos e interações, ao invés de manipular requisições e linguagem de marcação. O JSF utiliza a linguagem XHTML, que se caracteriza por ser uma linguagem originada do HTML mas com estrutura baseada em XML. Em comparação ao HTML, o XHTML mostra-se vantajoso ao suportar mecanismos de validação utilizados no XML, validando inclusive a formatação do documento (GONCALVES, 2013).

As aplicações desenvolvidas com JSF são aplicações web padrão que recebem requisições via *Faces Servlet* e produzem HTML. O JSF permite que o programador manipule em sua aplicação eventos, componentes Swing e *listeners* sendo possível inclusive renderizá-la para diferentes dispositivos (*layout* responsivo) (GONCALVES, 2013).

As aplicações web em geral podem interagir com o usuário atualizando ou modificando apenas alguns dos componentes de sua interface durante sua operação. Utilizar AJAX na construção das páginas web é um dos meios mais comuns de se conseguir interatividade e implementar modificações de forma assíncrona. O JavaServer Faces possui bibliotecas específicas para a utilização de AJAX, que permitem a manipulação de eventos e atualização de campos, como o que ocorre em uma página HTML comum provida de AJAX (GONCALVES, 2013).

O JavaServer Faces possui mecanismos diferenciados para manipulação de inclusões externas. Para imagens foi designada a *tag* <h:graphicImage> que trabalha com a importação de imagens. Para códigos JavaScript utiliza-se a *tag* <h:outputScript> e para códigos CSS <h:outputStylesheet>. Além disso, arquivos necessários para o funcionamento da aplicação podem ser armazenados em um diretório chamado *resources*, previamente identificado pelo JSF como uma espécie de repositório da aplicação (GONCALVES, 2013).

2.4 BIBLIOTECAS DE COMPONENTES PARA O JAVA SERVER FACES

Existem diferentes bibliotecas de componentes visuais baseadas no Java Server Faces. São exemplos dessas bibliotecas:

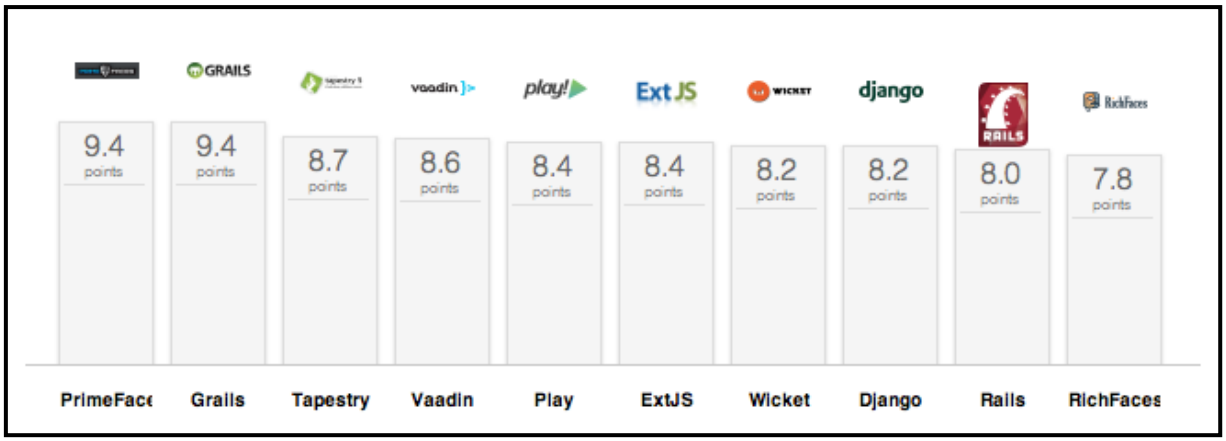
2.4.1 PrimeFaces

O PrimeFaces é uma biblioteca de componentes de interface para aplicações web cujo desenvolvimento foi baseado no JavaServer Faces. O PrimeFaces possui um conjunto de mais de 100 componentes de interfaces e sua utilização se dá por meio de um arquivo *.jar* único que não requer configurações adicionais nem dependências para funcionar (JONNA, 2014).

O PrimeFaces possui componentes equivalentes aos componentes mais comuns do html e ainda oferece suporte a utilização de gráficos (*charts*), galerias de imagens e componentes para reprodução de mídias em geral (Flash, QuickTime, MP3, AVI e PDF). O PrimeFaces dispõe de diálogos personalizados, onde o desenvolvedor pode exibir diálogos de seleção de dados ou mensagens ao usuário da aplicação. A biblioteca também possui compatibilidade ao uso de AJAX e permite a implementação de *layout* responsivo em seus componentes (PRIMEFACES, 2014).

A utilização do PrimeFaces ultrapassou os índices de utilização de outros *frameworks* conhecidos, conforme pesquisa realizada pela DevRates ilustrada e na Figura 5:

Figura 5: Preferência de frameworks por desenvolvedores



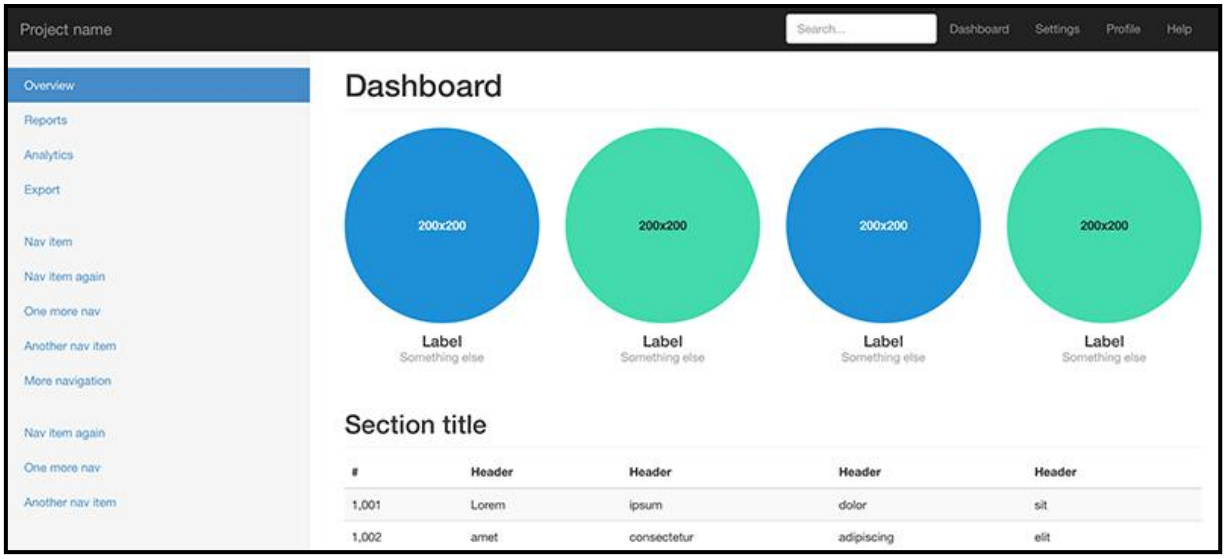
Fonte: PrimeFaces, 2014.

2.4.2 BootsFaces

O BootsFaces é uma biblioteca de componentes visuais para o JavaServer Faces, baseada no estilo visual do *framework* Bootstrap. O Bootsfaces, assim como o PrimeFaces, é disponibilizado em um arquivo *.jar*, possui suporte à utilização de AJAX e permite integração com outros *frameworks* como por exemplo o PrimeFaces, o OmniFaces e o ButterFaces (BOOTSFACES, 2016).

A biblioteca implementa componentes previamente preparados para interfaces *desktop* e *mobile,* desta forma, permitindo o desenvolvimento de *layout* responsivo para as aplicações. Para atender aos recursos presentes nos novos dispositivos móveis, segundo o desenvolvedor, estuda-se atualmente um meio de viabilizar à biblioteca o suporte a detecção de gestos do utilizador (*swipe gesture*) (BOOTSFACES, 2016). Um exemplo de *template* construído a partir da biblioteca *BootsFaces* é apresentado na Figura 6:

Figura 6: Interface desenvolvida a partir da biblioteca BootsFaces



Fonte: BOOTSFACES, 2016.

2.5 JAVA PERSISTENCE API

As aplicações são construídas de forma a gerarem, manipularem e receberem dados e muitos desses dados devem ser necessariamente guardados de forma persistente, ou seja, de forma que, quando o objeto da classe Java que os armazena

for destruído ou inutilizado, que se armazene as informações que ele continha em seus atributos de forma definitiva. Uma das formas mais comuns e seguras de se armazenar dados é em um banco de dados. Entretanto, um objeto de uma classe Java não possui as mesmas características de uma tabela de um banco de dados relacional, e é para viabilizar este intermédio que o Java possui uma API especializada na persistência de informações chamada Java Persistence API (GONCALVES, 2013).

A Java Persistence API foi desenvolvida para fazer o mapeamento objeto-relacional que envolve transformar uma entidade (objeto de uma classe Java) e seus atributos em registros em uma tabela do banco de dados. O termo entidade é equivalente ao termo objeto exceto pelo fato de que os objetos existem somente em memória e as entidades são objetos direcionados a persistência. A API utiliza a linguagem Java Persistence Query Language (JPQL) para consultas e manutenções e consegue efetuar acessos ao banco durante a manipulação das entidades pela aplicação (GONCALVES, 2013).

Para que uma classe seja uma entidade, algumas condições devem ser respeitadas:

- Deve receber a anotação "*@javax.persistence.Entity"* (GONCALVES, 2013).
- Um identificador deve ser definido, como uma chave primária única, e deve receber a anotação @javax.persistence.Id (GONCALVES, 2013).
- A classe da entidade deve implementar um método construtor sem argumentos (GONCALVES, 2013).

Quando uma entidade é persistida, seu nome é atribuído ao nome da tabela no banco e seus atributos tornam-se colunas da tabela, para posterior armazenamento das informações (GONCALVES, 2013).

A JPA utiliza uma especificação armazenada em um arquivo *.xml* que armazena informações referentes à conexão com o banco de dados. Essa especificação recebe o nome de Unidade de Persistência. Além da Unidade de Persistência, a JPA pode trabalhar integrada à Bean Validation API para que ocorra a validação dos dados na pré-persistência. Consolidada a validação, ocorre então o mapeamento objeto-relacional da entidade, efetuado com as seguintes anotações (GONCALVES, 2013):

- *@Table*: permite a manipulação das características da tabela gerada (GONCALVES, 2013).
- *@Id e @GeneratedValue*: permite a definição de uma chave primária por meio de um atributo único. A anotação *@GeneratedValue* permite a geração automática do valor da chave identificadora (GONCALVES, 2013).
- *@Column*: permite a manipulação das características da coluna gerada (GONCALVES, 2013).
- *@Temporal:* permite definir que determinada coluna do banco receberá valores referentes a datas e horários (GONCALVES, 2013).

Os relacionamentos entre entidades podem receber anotações *@OneToOne*, *@ManyToOne*, *@OneToMany* e *@ManyToMany* (GONCALVES, 2013).

2.6 BEAN VALIDATION API

Uma das grandes preocupações dos desenvolvedores é garantir que os dados que serão processados e armazenados por suas aplicações sejam válidos. Para facilitar essa importante tarefa, o Java possui uma API de validação chamada *Bean Validation API*, que trabalha com padrões de validação de dados em *beans*, atributos, construtores, retornos de métodos e parâmetros (GONCALVES, 2013).

Na *Bean Validation API*, a implementação de uma restrição acontece por meio de uma anotação que pode validar o tipo de um dado, seu tamanho ou sua obrigatoriedade (GONCALVES, 2013). São exemplos de anotações:

- *Not Null:* representa a obrigatoriedade da existência desse dado (GONCALVES, 2013).
- *Not Blank:* semelhante ao *Not Null*, mas permite a inserção de valores nulos (GONCALVES, 2013).
- *Not Empty:* verifica se uma estrutura está vazia, por exemplo, uma string (GONCALVES, 2013).
- *Max:* utilizado para inteiros e decimais, define o valor máximo a ser atribuido (GONCALVES, 2013).
- *Min:* utilizado para inteiros e decimais, define o valor mínimo a ser atribuido (GONCALVES, 2013).

- *Size:* define o tamanho máximo de um valor inteiro ou decimal (GONCALVES, 2013).
- *Email:* valida o conteúdo de uma estrutura alfanumérica verificando se é compatível com um endereço e-mail (GONCALVES, 2013).

De semelhante modo, pode-se criar restrições personalizadas, onde o desenvolvedor irá definir as regras de validação dos dados por meio da anotação *@Pattern* (GONCALVES, 2013).

Nos métodos, as validações podem acontecer como pré ou pós condições ou validar os argumentos que foram passados para a função. Na Figura 7 é apresentado um exemplo de anotação para validação de dados:

Figura 7: Código com validação de dados em métodos

```
@AssertTrue
public Boolean validate(@NotNull CreditCard creditCard) {

  return validationAlgorithm.validate(creditCard.getNumber(), creditCard.getCtrlNumber());
}
```

Fonte: GONCALVES, 2013, p.81

Ao construir uma restrição e validar uma determinada estrutura, é possível que se obtenham erros oriundos da validação dos dados. Para isso, a *Bean Validation API* possui mecanismos de gerencia de mensagens, que podem ser disparadas quando um dado não respeita a validação a que foi submetido. As mensagens de validação da API são úteis para informar o usuário de que ocorreu um erro ao validar determinada informação e apresentar ao mesmo a estrutura correta a ser inserida (GONCALVES, 2013).

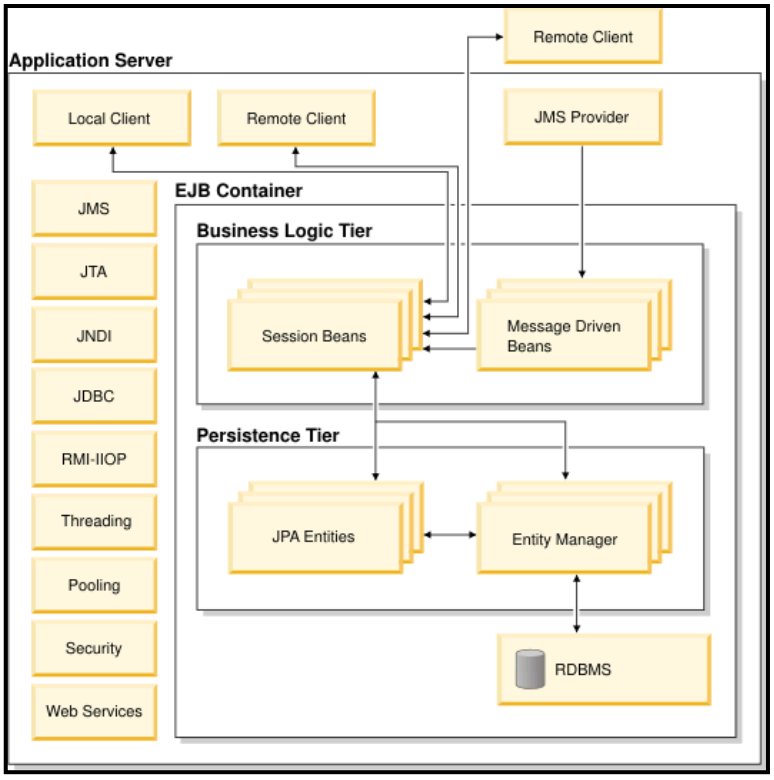
2.7 ENTERPRISE JAVABEANS

As entidades presentes em um código Java podem possuir métodos para validação e para execução de operações sobre seus atributos. Contudo existem operações complexas e interações (comunicação com outras classes persistentes ou comunicação com serviços externos) as quais seu desenvolvimento na camada de persistência torna-se impróprio. Por este motivo, vê-se a necessidade de implementação de uma camada denominada camada de negócio, que no Java EE

pode ser desenvolvida usando-se *Enterprise JavaBeans* (EJBs) (GONCALVES, 2013).

Os *Enterprise JavaBeans* são componentes que operam no lado do servidor, encapsulando a lógica de negócio e gerenciando os processos que envolvam transações, bem como a parte de segurança envolvida nesses processos. Os EJBs possuem funções integradas para manipulação de mensagens, gerência de acesso remoto, *web services* (RESTful e SOAP), ciclo de vida dos componentes entre outras funções, além de possuir integração com outros componentes da plataforma Java EE tais como a *Java Transaction API* (JTA), a *Java Persistence API* (JPA) e os *drivers* de conexão, tais como o *driver* JDBC. Por estes motivos, os EJBs podem ser utilizados para a construção da camada de negócio da aplicação (GONCALVES, 2013). A Figura 8 contém um exemplo da estrutura de um EJB:

Figura 8: Estrutura de um EJB



Fonte: IBM Knowledge Center, 2013.

Os Enterprise JavaBeans permitem que aplicações escritas em linguagens diferentes do Java utilizem, por exemplo, um *web service* que opera sob plataforma

Java EE e acessem seus recursos como se estas fossem aplicações Java (Heffelfinger, 2015). Essa característica dá-se pois os EJBs criam um *container* para a implantação do código e por meio deste *container* torna-se possível a disponibilização de recursos para controle sobre transações e concorrência além de permitir implementações de segurança sobre o acesso aos recursos (GONCALVES, 2013).

Os *Enterprise JavaBeans* podem ser implementados sob três anotações: como *Singleton*, onde existe somente uma instância da classe, que é compartilhada pelos componentes de toda a aplicação; como *Stateful*, onde cada chamada de método cria uma nova instância da classe e armazena seu estado; ou *Stateless* onde uma instância da classe pode ser utilizada por qualquer requisição de clientes, não armazenando estados. (GONCALVES, 2013).

2.8 DESIGN RESPONSIVO DE INTERFACES

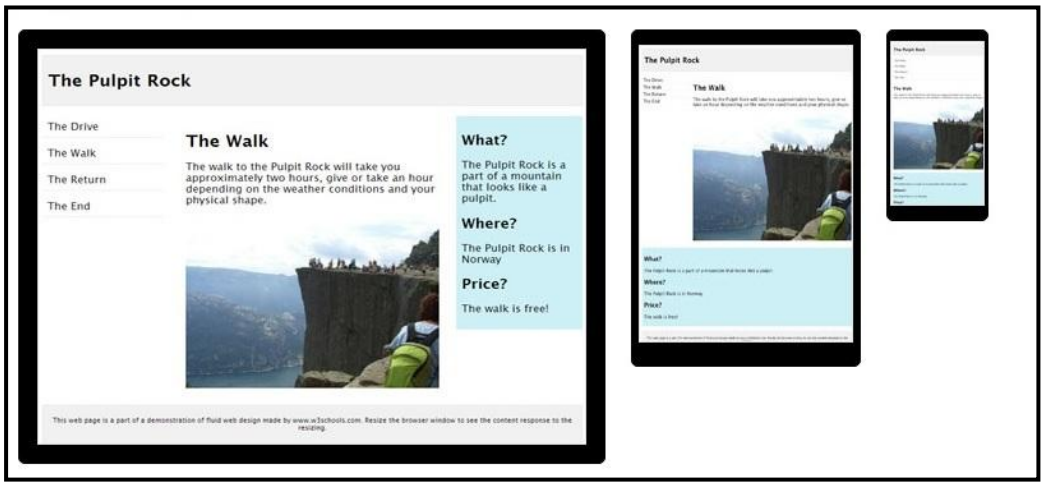
O conceito de Design Responsivo de Interfaces traz uma preocupação não somente com a beleza visual das interfaces, ricas em efeitos e cores, mas sim, com a possibilidade dessas interfaces se adaptarem a todos os tamanhos de telas de dispositivos, às inúmeras especificações de resolução de telas, sugerindo abandonar o pré-fixado e previsível e adotar o flexível (ZEMEL, 2012).

Com a grande ascensão de aparelhos *smartphones* e *tablets*, que possuem diferentes tamanhos de telas, as empresas viram-se forçadas a produzir versões específicas de seus sites para atender a este público. A primeira solução adotada foi a utilização de subdomínios, onde a detecção de um dispositivo móvel pelo site direcionava o usuário a um domínio especializado. Contudo, a produção de designs responsivos apresentou uma nova solução para esta problemática, onde o mesmo site, sem subdomínios, atende a todas as especificações de diferentes dispositivos, sem perder qualquer característica dos elementos visuais implementados na interface (ZEMEL, 2012).

Importante frisar que um *layout* responsivo não significa apenas diminuir e aumentar elementos conforme o tamanho da tela, mas sim, reorganizar o *layout* da interface sempre que houver mudança nas dimensões do *display* (PETERSON,

2014). A Figura 9 apresenta um exemplo da transformação sofrida por uma interface que implementa *layout* responsivo, quando acessada por diferentes dispositivos:

Figura 9: Exemplo de Interface Responsiva



Fonte: W3C, 2016.

Ao criar uma página que utiliza *layout* responsivo, o desenvolvedor deve se ater a substituir as medidas fixas como pixels, pontos e centímetros por medidas em porcentagem e medidas relativas como o '*em'.* (ZEMEL, 2012) (PETERSON, 2014).

Páginas web que não implementam design responsivo acabam "presas" à detecção de dispositivo, funcionalidade que direciona o usuário à uma página específica para seu dispositivo, normalmente armazenada em um subdomínio. Caso a detecção falhe, o usuário poderá visualizar uma página com *layout* inadequado ao seu dispositivo (PETERSON, 2014). Em um *layout* responsivo, a detecção acontece por meio de *media queries*, que são mecanismos de teste que verificam algumas características do dispositivo em que está ocorrendo o acesso (POWERS, 2012). A utilização de *media queries* permite a adaptação do *layout* (que consiste em implementar diferentes *layouts* para a mesma interface) sem redirecionar a navegação e os testes podem ocorrer em vários instantes, evitando que o usuário seja redirecionado incorretamente para uma página com layout incompatível (PETERSON, 2014).

2.9 HTML5

HTML, sigla de *Hyper Text Markup Language*, é uma linguagem de hipertextos, sendo que um hipertexto é todo texto presente em páginas web, potencializado pela possibilidade de conexão a outros documentos da web pelos chamados *links* (SILVA, 2015).

O HTML5 transformou a linguagem de marcação HTML em uma API de desenvolvimento web. Isso se deve ao fato da linguagem ter recebido novas marcações e muitos componentes terem recebido melhorias em suas funcionalidades (BROWN, BUTTERS e PANDA, 2014).

Os novos campos de inserção permitem a criação de controles em formulários sem a necessidade de bibliotecas JavaScript. Um exemplo é o campo de inserção *slider*, que necessitava de inclusões externas para seu funcionamento e agora pode ser implementado simplesmente pela tag *<input type=range>*. Campos e-mail e url agora possuem validação no lado cliente e foram criados elementos *audio* e *video*, que incorporam os arquivos de mídia aos documentos HTML. Para manipulação de imagens, gráficos e jogos foi criado o elemento *canvas*, que utiliza gráficos vetoriais para manipular os tipos citados (BROWN, BUTTERS e PANDA, 2014).

Alguns dos novos elementos ou tipos de elementos do HTML5 são:

- Campo *Input* tipo “tel” – executa a validação para telefones (BROWN, BUTTERS e PANDA, 2014).
- Campo *Input* tipo “email” – executa a validação para endereços e-mail (BROWN, BUTTERS e PANDA, 2014).
- Campo *Input* tipo “url” – executa a validação para URLs (BROWN, BUTTERS e PANDA, 2014).
- Campo *Input* tipo “file” – não é novidade no HMTL, mas a novidade se dá pela possibilidade de se enviar múltiplos arquivos (BROWN, BUTTERS e PANDA, 2014).
- Campo *Input* tipo “search” – facilitador visual para campo busca (BROWN, BUTTERS e PANDA, 2014).
- Campo *Input* tipo “range” – permite a implementação de campos de inserção com formatos diferentes, como por exemplo, o *slider* (BROWN, BUTTERS e PANDA, 2014).

- Campo *Input* tipo “number” – executa validação para formatos numéricos (BROWN, BUTTERS e PANDA, 2014).
- Campo *Input* tipo “date” ou “time” – executa validação para datas e horários (BROWN, BUTTERS e PANDA, 2014).
- Campo *vídeo* – permite a inclusão de executores de vídeos na página (BROWN, BUTTERS e PANDA, 2014).
- Campo *audio* – permite a inclusão de executores de áudio na página (BROWN, BUTTERS e PANDA, 2014).
- Campo *canvas* – permite a ilustração de formas geométricas e elementos visuais (figuras, por exemplo) (BROWN, BUTTERS e PANDA, 2014).

2.10 CSS3

Cascading Style Sheets, que tiveram sua primeira versão lançada em 1996, são scripts que definem propriedades de estilo de um elemento HTML (PETERSON,2014). Os códigos CSS contém seletores que aplicam estilos para a renderização dos elementos a que forem designados. Um seletor pode ser atribuído a um elemento, uma classe ou a um identificador específico e pode aplicar estilizações para cores, formatos, tamanho de fonte, posição e dimensões (POWERS, 2012). Em questão de medidas, é possível definir áreas em centímetros, pixels, porcentagem e agora, para adequação ao conceito de *layout* responsivo, em ‘*em’* (POWERS, 2012) (ZEMEL, 2012).

Com a ascensão do HTML5, o desenvolvimento do CSS3 acabou sendo também voltado para suprir a estilização dos novos elementos constantes na linguagem. Os elementos de desenho e gráficos, definidos pela *tag* ‘*canvas’* receberam atribuições para estilização de linhas, imagens, plano de fundo e cores (POWERS, 2012).

Para atender aos princípios do Design Responsivo de Interfaces, o CSS3 implementa os chamados *media queries*, que são estruturas que analisam características do dispositivo que está acessando a determinada página e aplicam as definições de estilo adequadas para a resolução da tela. (POWERS, 2012). Esse mecanismo permite que o desenvolvedor utilize a mesma página HTML para

dispositivos com diferentes características de tamanho de tela e resolução (POWERS, 2012).

A utilização dos *media queries* pode ser feita dentro do próprio código CSS por meio de anotações específicas como o *@media* (POWERS, 2012), e que é exemplificada na Figura 10:

Figura 10: Utilização de *media queries* embutido ao código CSS.

```
@media only screen and (min-width: 481px) and (max-width: 768px) {
    /* Styles for screen widths between 481px and 768px */
}
```

Fonte: POWERS, 2012, p. 395.

Algumas características que os *media queries* avaliam para a escolha do código CSS adequado são:

- *Width:* avalia a largura da área da tela do dispositivo (POWERS, 2012).
- *Height:* avalia a altura da área da tela do dispositivo (POWERS, 2012).
- *Device-width:* avalia a largura do dispositivo (POWERS, 2012).
- *Device-height:* avalia a altura do dispositivo (POWERS, 2012).
- *Orientation:* percebe a orientação atual do dispositivo (POWERS, 2012).
- *Resolution:* detecta a resolução da tela do dispositivo (POWERS, 2012).

Embora os códigos CSS tenham a especificação normatizada pela W3C, algumas versões de navegadores, especialmente as mais antigas do Microsoft Internet Explorer não são completamente compatíveis ao seu uso. Por essa razão, é necessário o desenvolvimento de códigos alternativos para a estilização de documentos nesses navegadores (POWERS, 2012).

2.11 BANCO DE DADOS POSTGRESQL

O PostgreSQL é um sistema de banco de dados objeto-relacional com código fonte aberto, que pode ser implementado em plataformas UNIX, Linux, MAC e Windows. O desenvolvimento deste sistema SGBD iniciou-se em 1986, na Universidade da Califórnia em Berkeley, com o professor Michael Stonebraker. Logo após, em 1995, dois alunos do professor Stonebraker, Andrew Yu e Jolly Chen,

desenvolveram uma nova versão do PostgreSQL que utilizava como base um subconjunto estendido do SQL, e que foi denominado Postgres95. Mas foi a partir de 1996, onde um grupo de desenvolvedores que não faziam parte da Universidade integrou-se ao projeto, que o Postgres95 avançou significativamente. Nesse ano, o projeto tornou-se *open-source* e com o conhecimento de diferentes desenvolvedores, o código fonte do sistema tornou-se consistente e uniforme, além de ter recebido correções e implementações que ocasionaram em funcionalidades ao SGBD (POSTGRESQL, 2016).

Atualmente, o PostgreSQL tem suporte para grande variedade de comandos SQL, apresenta funcionalidades tais como controle de concorrência, controle de transações, tolerância a falhas e *backup*, além de suportar o armazenamento de inúmeros tipos de dados tais como inteiros, *strings* e decimais além de poder armazenar sons, imagens e vídeo. Também implementa grande variedade de interfaces para comunicação com aplicações nas linguagens C / C ++, Java, .Net , Perl, Python, Ruby, Tcl , ODBC , entre outras (POSTGRESQL, 2016).

O PostgreSQL é uma ferramenta de manipulação de dados mas também de programação avançada, composta por gatilhos, funções, verificação de integridade e códigos com funcionalidades implementadas em linguagem *PL/pgSQL.* O *PL/pgSQL* é uma linguagem de programação própria para o PostgreSQL que foi desenvolvida para ser utilizada em conjunto com comandos SQL (KROSING, ROYBAL e MLODGENSKI, 2013). Para exemplificar o uso de funções complexas em PL/pgSQL, apresenta-se na Figura 11 o código de uma função que recebe uma imagem, converte-a e grava sua miniatura na base de dados:

Figura 11: Função em linguagem PL/pgSQL para criação de miniaturas

```
CREATE FUNCTION save_image_with_thumbnail(image64 text)
  RETURNS int
AS $$
import Image, cStringIO
size = (64,64) # thumbnail size

# convert base64 encoded text to binary image data
raw_image_data = image64.decode('base64')

# create a pseudo-file to read image from
infile = cStringIO.StringIO(raw_image_data)
pil_img = Image.open(infile)
pil_img.thumbnail(size, Image.ANTIALIAS)

# create a stream to write the thumbnail to
outfile = cStringIO.StringIO()
pil_img.save(outfile, 'JPEG')
raw_thumbnail = outfile.getvalue()

# store result into database and return row id
q = plpy.prepare('''
  INSERT INTO photos(image, thumbnail)
  VALUES ($1,$2)
  RETURNING id''', ('bytea', 'bytea'))
res = plpy.execute(q, (raw_image_data,raw_thumbnail))

# return column id of first row
return res[0]['id']
$$ LANGUAGE plpythonu;
```

Fonte: KROSING, ROYBAL e MLODGENSKI, 2013, p.152

2.12 UNIFIED MODELING LANGUAGE

A *Unified Modeling Language* (UML) foi criada a partir da união de três metodologias de representação que, em 1990, eram utilizadas para a modelagem de sistemas: o método de Booch, o método OMT (*Object Modeling Technique*) de Jacobson, e o método OOSE (*Object-Oriented Software Engineering*) de Rumbaugh. Para que essa união ocorresse, os pesquisadores envolvidos receberam o apoio da Rational Software, uma das maiores empresas no ramo de desenvolvimento da época. Em 1997, a OMG (*Object Management Group)* tornou a UML linguagem padrão para o desenvolvimento de modelagens de sistemas. A partir daí, a UML recebeu uma atualização denominada UML 2.0 no ano de 2005 e ainda recebe modificações com versionamento, encontrando-se atualmente na versão 2.3 beta (GUEDES, 2009).

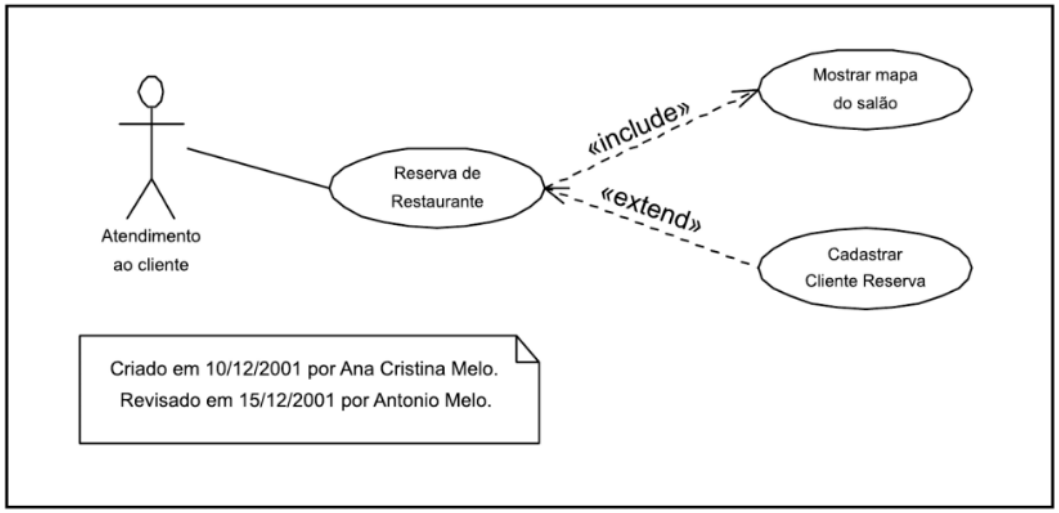
Conforme GUEDES, a definição da linguagem UML dá-se por “uma linguagem visual utilizada para modelar softwares baseados no paradigma de orientação a objetos” (2009, p.19). Define-se como visual pois a produção da linguagem são diagramas de representação dos fluxos, interações ou relações que ocorrerão no software, facilitando sua implementação (MELO, 2010). Diagramas presentes na UML, conforme GUEDES (2009, p.31 - 42):

- Diagrama de Casos de Uso
- Diagrama de Classes
- Diagrama de Objetos
- Diagrama de Pacotes
- Diagrama de Sequência
- Diagrama de Comunicação
- Diagrama de Máquina de Estados
- Diagrama de Atividades
- Diagrama de Visão Geral de Interação
- Diagrama de Componentes
- Diagrama de Implantação
- Diagrama de Estrutura Composta
- Diagrama de Tempo ou Temporização

Dentre os diagramas listados, explanam-se os destacados e utilizados neste trabalho. São eles:

Diagramas de Casos de Uso: Os diagramas de casos de uso procuram representar de que forma os atores (que podem ser usuários, sistemas externos ou componentes de hardware) irão utilizar o sistema por meio dos casos de uso (serviços) relacionados às funcionalidades do software (GUEDES, 2009). Segundo MELO, os diagramas de Casos de Uso tem “o objetivo de demonstrar o comportamento de um sistema (ou parte dele), por meio de interações com atores” (2010, p. 56). É um dos diagramas mais informais da UML, e serve principalmente para a identificação dos comportamentos do sistema (GUEDES, 2009). Um exemplo de Diagrama de Casos de Uso é apresentado na Figura 12:

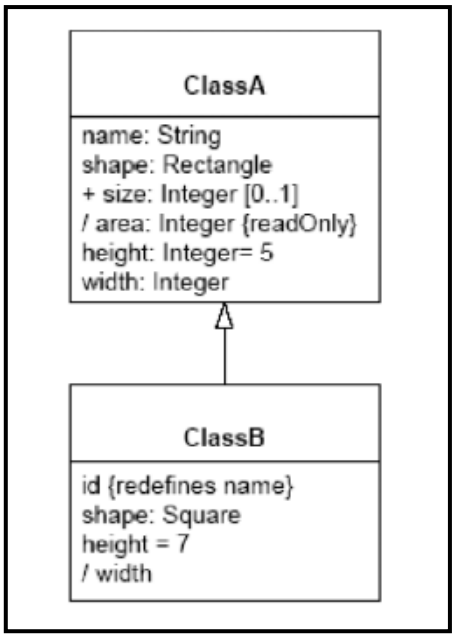
Figura 12: Exemplo de Diagrama de Caso de Uso



Fonte: MELO, 2010, p. 66.

Diagrama de Classes: os diagramas de classes são representações que servem de apoio para os demais diagramas da UML. Esse tipo de diagrama “define a estrutura das classes utilizadas pelo sistema, determinando os atributos e métodos que cada classe tem, além de estabelecer como as classes se relacionam e trocam informações entre si” (GUEDES, 2009, p.33). Um exemplo de Diagrama de Classes de uso é apresentado na Figura 13:

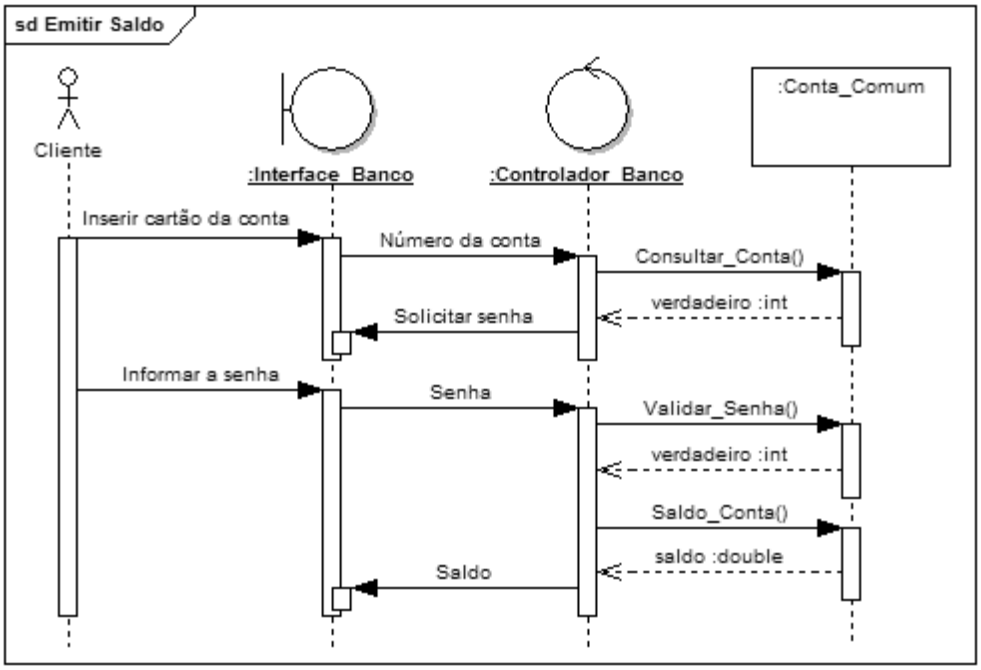
Figura 13: Exemplo de Diagrama de Classes



Fonte: MELO, 2010, p. 111.

Diagrama de Sequência: segundo GUEDES, “um diagrama de sequência é um diagrama comportamental que se preocupa com a ordem temporal em que as mensagens são trocadas entre os objetos envolvidos em um determinado processo” (2013, p.35). Para construção deste diagrama, extrai-se dos diagramas de casos de uso e diagrama de classes os objetos referentes às classes envolvidas em determinado processo e define-se o evento gerador do processo citado. Após isso, são ilustradas as trocas de mensagens entre objetos até a finalização do processo (GUEDES, 2009). Um exemplo de Diagrama de Sequência é apresentado na Figura 14:

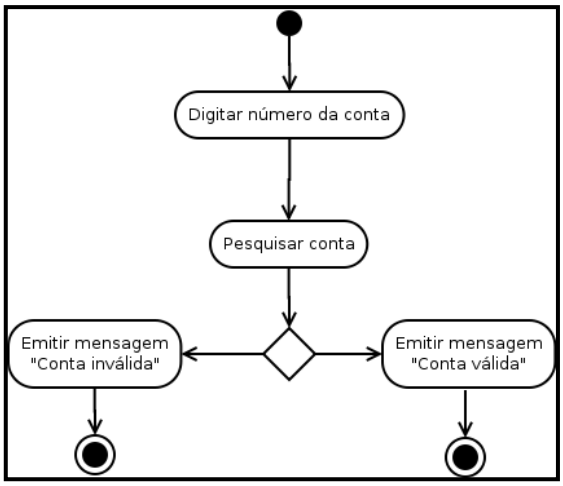
Figura 14: Exemplo de Diagrama de Sequência



Fonte: GUEDES, 2009, p. 36.

Diagrama de Atividades: o diagrama de atividades é oriundo do diagrama Gráfico de Estados (hoje conhecido como Máquina de Estados) e tem por finalidade ilustrar as etapas necessárias para que se logre êxito em uma determinada atividade. Este tipo de diagrama busca demonstrar o fluxo de controle de determinado processo até sua finalização (GUEDES, 2009). Um exemplo de Diagrama de Atividades é apresentado na Figura 15:

Figura 15: Exemplo de Diagrama de Atividades



Fonte: BEZERRA JR, 2012.

2.13 PESQUISAS RELACIONADAS

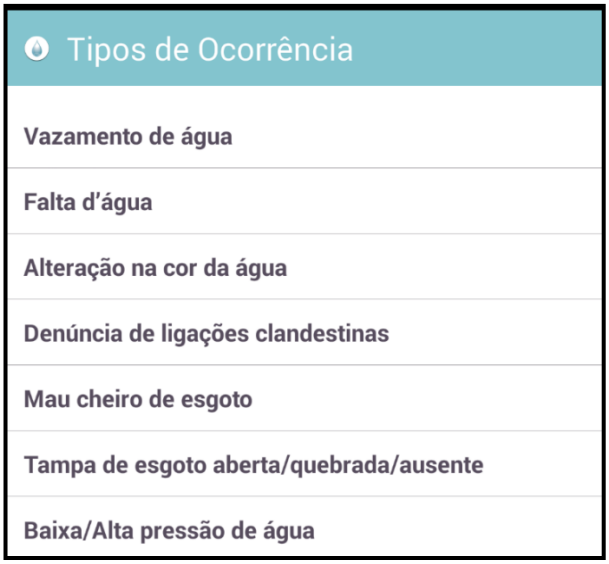
O aplicativo para plataforma *Android* Agualert[1], desenvolvido pela empresa *Swell It Solutions* para a Companhia Catarinense de Águas e Saneamento – Casan - é um pleno exemplo de investimento tecnológico para a ampliação dos canais de interação da empresa com o público. Conforme o superintendente da região Bombinhas / Porto Belo, Lucas Arruda, “Precisamos facilitar o contato do usuário com a CASAN para agilizar a solução de problemas" (CASAN, 2015). No aplicativo, é possível registrar[2] informações em modo anônimo ou em modo identificado, bem como utilizar o sistema de GPS de um aparelho *smartphone* para que o aplicativo registre a localização do informante automaticamente, facilitando o preenchimento das informações no sistema.

Após as definições de identificação e localização, o Agualert permite ao usuário informar demandas com base em uma lista predefinida de possíveis ocorrências, como ilustrado na figura 16:

[1] Disponível em https://play.google.com/store/apps/details?id=com.swellitsolution.agualert&hl=pt_BR

[2] Experimentação executada diretamente no aplicativo Agualert para teste.

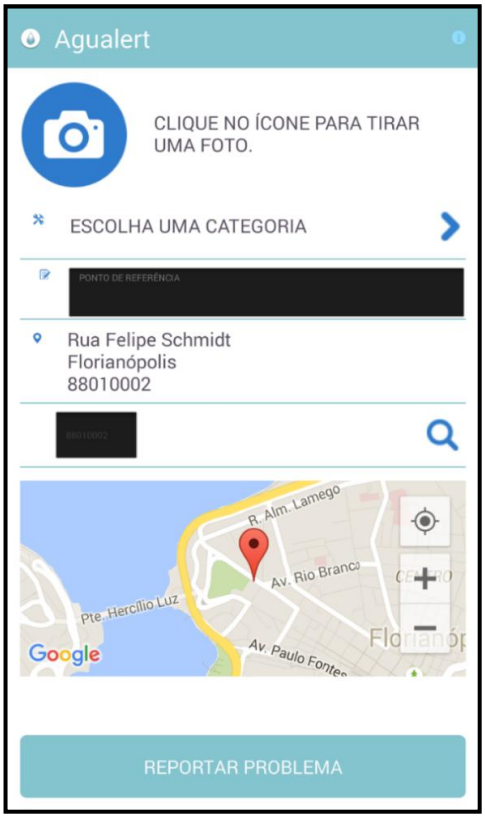
Figura 16: Interface do Aplicativo Agualert com listagem de ocorrências



Fonte: Aplicativo Agualert, 2015.

Também é possível dimensionar a gravidade do problema enviando uma fotografia pelo aplicativo, que possui suporte para este recurso: A interface que apresenta estas funcionalidades é apresentada na Figura 17:

Figura 17: Interface do Aplicativo Agualert com recurso fotografia e GPS.



Fonte: Aplicativo Agualert, 2015.

Após o envio da demanda, a mesma é recebida na empresa que a direciona a um dos setores de execução, onde a solicitação será de fato atendida. Por fim, são registradas no sistema, sob o protocolo repassado previamente ao usuário, as informações pertinentes à execução dos serviços ou esclarecimentos sobre a demanda requisitada (CASAN, 2015).

3 METODOLOGIA

Para a elaboração deste trabalho foram realizadas coletas de dados sobre os processos comerciais de uma empresa de saneamento atuante na região sul do Brasil, além de pesquisas que buscaram aporte bibliográfico sobre aplicações no âmbito do saneamento. Para facilitar a comparação com a proposta do trabalho, foram utilizados, majoritariamente, os dados fornecidos por uma empresa de saneamento que atua no estado do Rio Grande do Sul. Essa empresa permitiu a divulgação dos dados, mas solicitou que não fossem vinculados ao nome da organização.

Com base nos requisitos levantados a partir da análise do sistema e nos objetivos do projeto, foi desenvolvida uma modelagem da aplicação por meio da linguagem de modelagem UML (Unified Modeling Language). Pesquisas bibliográficas foram realizadas para obtenção de conhecimento sobre as tecnologias a serem utilizadas. Esta modelagem serviu como base para a escolha das tecnologias que implementaram a aplicação.

Foi definido que o Sistema de Gerenciamento de Banco de Dados a ser utilizado seria o PostgreSQL. Para a persistência dos dados, a tecnologia escolhida foi o *Java Persistence API* e a camada que forma a interface de interação com o usuário foi desenvolvida a partir do *Framework Java Server Faces* e da biblioteca BootsFaces, que abstrai a utilização das linguagens *HTML e CSS*, implementando desta forma o design responsivo da aplicação.

3.1 ESTUDO DE CASO:

Esta sessão apresentará o estudo de caso efetuado bem como irá relatar o método utilizado atualmente pela empresa usada como referência para o tratamento das ocorrências operacionais informadas. Também serão citadas as possíveis vulnerabilidades deste método de gestão.

3.1.1 Histórico da Empresa

A empresa utilizada como referência atua no ramo de saneamento há aproximadamente cinquenta (50) anos. Possui grandes sistemas de abastecimento de água tratada, estações próprias de bombeamento e tratamento de água. Geralmente, a água bruta é captada de mananciais superficiais como rios e lagos, mas em cidades que não possuem estes recursos a empresa investe na perfuração de poços artesianos.

Outro ramo de atuação da empresa é a coleta e tratamento de resíduos efluentes. A empresa possui sistemas de coleta desses resíduos, que os destinam a estações que efetuam o tratamento dos rejeitos oriundos principalmente da atividade humana e devolvem o subproduto despoluído ao meio ambiente.

Ainda assim existem determinados serviços que são terceirizados. Um exemplo citado pela empresa é a reposição do pavimento em valas abertas nas vias públicas, que apenas é fiscalizado pela empresa contratante.

A empresa entrevistada utiliza um sistema comercial de grande porte para a gestão de suas informações. Esse sistema é utilizado para fins cadastrais, registros operacionais e gerência de mecanismos de cobrança.

3.1.2 Método atual de gestão de informações

O sistema utilizado pela empresa abrange cadastros de usuários, cadastros de ruas, ordens de serviço, processos de infração, solicitações e ocorrências de usuários, cobrança e faturamento, bem como os relatórios pertinentes a todas essas atividades. Conforme informações da empresa, por questões de segurança e pela arquitetura utilizada para o desenvolvimento do sistema, o mesmo não é acessível pelo público geral, somente por colaboradores da empresa. Desta forma, quando um usuário deseja solicitar ou informar uma demanda, fica condicionado ao contato telefônico que pode ser feito por meio do *call center* da empresa ou pelos telefones locais dos escritórios, ou, se em horário compatível, o usuário pode se deslocar até o escritório local da empresa.

Quando uma demanda de usuário requer a confecção de uma ordem de serviço, o processo inicia-se com a solicitação e o encaminhamento da ordem de serviço ao

setor responsável. O colaborador responsável pelo setor tem acesso ao relatório das ordens de serviço pertinentes ao seu setor. O responsável pelo setor destina a ordem de serviço a uma de suas equipes de campo, para a execução do serviço. Como as ordens de serviço são impressas, as equipes necessitam retornar ao escritório local para a obtenção de novas ordens de serviço.

Após a execução, as ordens de serviço retornam ao setor de origem, onde são analisadas e os dados informados pelas equipes de campo são registrados no sistema comercial. Quando determinado serviço requer a abertura de vala no pavimento da via pública, fica a cargo do responsável pelo setor registrar a necessidade da reposição do pavimento que no caso dessa empresa, é solicitada a uma empresa terceirizada contratada para este fim. O usuário não recebe nenhum retorno de informações referente à execução do serviço, o mesmo somente é informado se efetuar novo contato telefônico ou se comparecer ao escritório da empresa.

A reposição do pavimento é controlada pela empresa apenas por uma planilha digital desenvolvida para este fim, fora do sistema comercial principal. Esta planilha não registra fotos e apenas possui dados essenciais sobre os serviços que a empresa terceirizada deverá executar tais como o endereço onde será feita a reposição do pavimento e a metragem a ser pavimentada.

Importante ressaltar que, caso o usuário queira informar uma demanda qualquer e venha a entrar em contato com o *call center* da empresa (ao invés dos telefones dos escritórios) deverá obrigatoriamente informar o código referente ao imóvel onde será efetuado o serviço, mesmo que este serviço seja executado na rua e não no imóvel.

3.1.3 Análise sobre o método utilizado atualmente

Observa-se no relato do método utilizado atualmente pela empresa citada que existem itens que podem restringir a comunicação entre o usuário e a empresa de saneamento. Inicialmente, cita-se que a empresa não apresenta nenhuma opção de comunicação por meio da internet, onde o cliente possa obter um protocolo de atendimento ou relatar seu problema a um atendente, mesmo que em uma espécie de *chat*. Dessa forma, elimina-se grande parte das possibilidades de comunicação

para usuários de dispositivos como computadores e *tablets.* Importante ressaltar que a empresa possui em sua página na internet a opção “Fale Conosco”, mas não é gerado nenhum tipo de protocolo de atendimento para esse contato.

Outro item que pode restringir a possibilidade de comunicação do usuário é a obrigatoriedade, quando se utiliza o *call center* como meio de informação, de se informar um código referente ao móvel solicitante do serviço. Essa regra pode acabar impedindo que uma pessoa informe a empresa de que existe uma ocorrência em uma determinada rua que não tenha relação com sua moradia. Portanto, se o usuário não portar uma fatura de serviços ou não possuir em mãos o código referente ao imóvel de destino não poderá efetivar seu pedido por meio do *call center.*

Como último recurso, o usuário pode então dirigir-se ou entrar em contato com o escritório local da empresa. Esse poderia ser um item sem qualquer restrição, se os escritórios funcionassem em turno ininterrupto. Como isso não acontece, novamente encontra-se uma barreira para a informação de uma ocorrência.

4 MODELAGEM DO SISTEMA

Para modelagem do sistema, analisou-se o contexto apresentado pela empresa entrevistada e foram extraídas informações relevantes para o desenvolvimento da aplicação. Neste processo, apresentam-se os requisitos funcionais e não funcionais e os diagramas que ilustram o funcionamento e a estrutura do software.

4.1 REQUISITOS FUNCIONAIS

- **RF1.** O sistema deverá manter o cadastro de ocorrências operacionais.
- **RF2.** O sistema deverá manter o cadastro de usuários, para a identificação do usuário responsável pela informação de determinada ocorrência.
- **RF3.** O sistema deverá manter o cadastro de equipes de colaboradores, para a identificação da equipe responsável pela execução das ordens de serviço.
- **RF4.** O sistema deverá manter o cadastro da Unidade Orgânica, contendo ruas e bairros.
- **RF5.** O sistema deverá dispor de mecanismo de consulta por protocolo de solicitação.
- **RF6.** O sistema deverá manter o cadastro de imóveis vinculados ao sistema de saneamento, tendo em vista que uma ocorrência pode estar vinculada a um imóvel.
- **RF7.** Para a interface colaborador, o sistema deverá, obrigatoriamente, exigir a autenticação do usuário por meio de um método de *login*.
- **RF8.** Na sessão de usuários colaboradores, o sistema deverá permitir a geração de relatórios das ocorrências registradas.
- **RF9.** O sistema deverá apresentar ao usuário comum apenas a interface simplificada, que permite o registro de ocorrências e consulta por protocolo. Já ao usuário colaborador, o sistema deverá apresentar a interface completa, com filtros, relatórios e campos para o registro dos dados referentes à execução de determinado serviço. Também fica a cargo do sistema gerenciar as permissões de cada tipo de usuário.

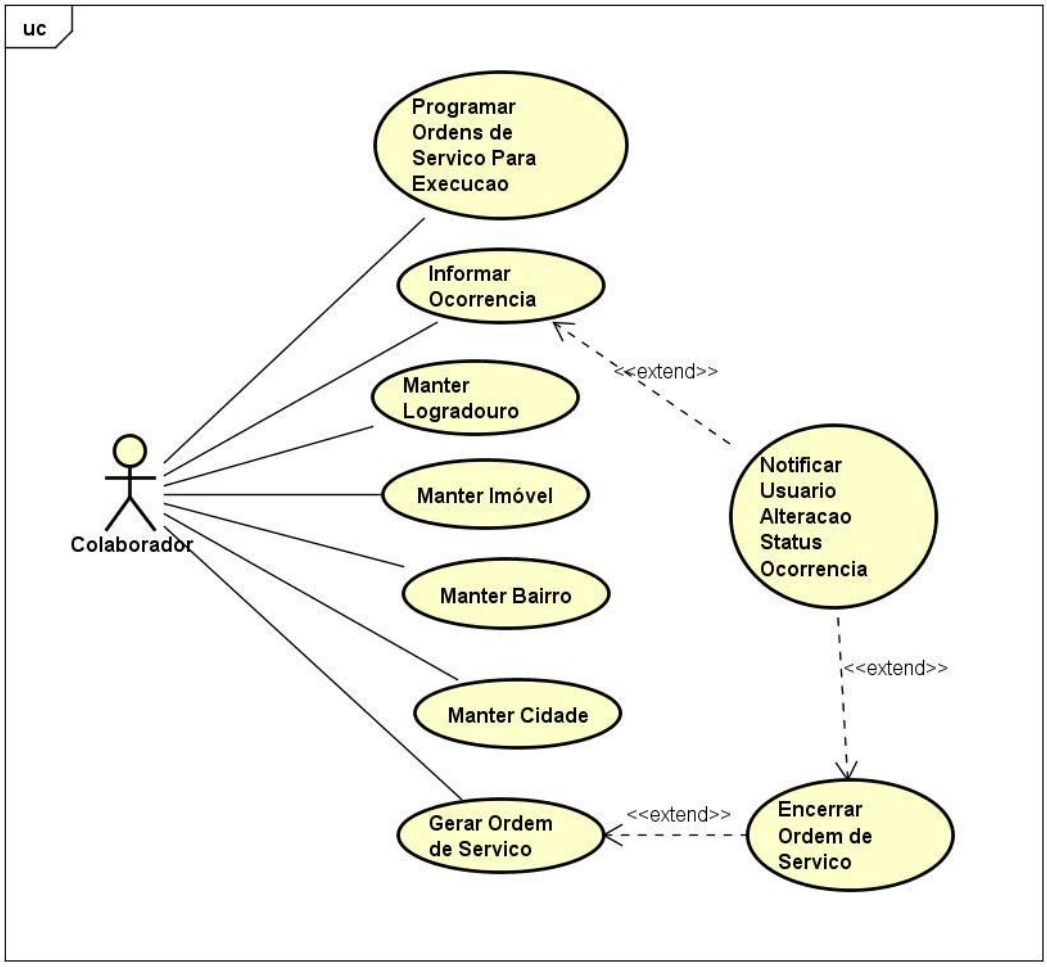
4.2 REQUISITOS NÃO FUNCIONAIS

- **RNF1.** O sistema deverá utilizar design responsivo, para implementar uma interface adequada ao maior número possível de dispositivos que possam utilizá-lo.
- **RNF2.** A aplicação deverá funcionar em diferentes plataformas de sistemas operacionais.
- **RNF3.** O sistema deverá ser implementado em tecnologia compatível aos padrões atuais de computação.

4.3 DIAGRAMAS DE CASO DE USO

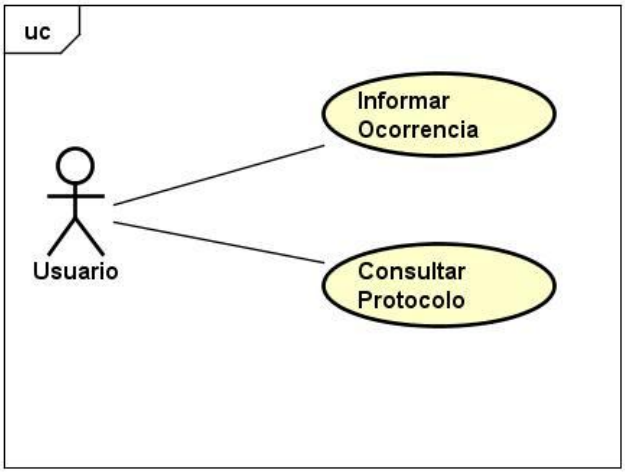
Os diagramas de caso de uso são de fundamental importância para a visualização simplificada das ações dos atores e das funcionalidades do sistema. Os diagramas a seguir foram criados com base nos requisitos levantados a partir estudo de caso apresentado.

Figura 18: Caso de Uso Usuário Colaborador.



Fonte: Do Autor

Figura 19: Caso de Uso Usuário Comum



Fonte: Do autor

4.4 DESCRIÇÃO DOS CASOS DE USO

A Documentação dos Casos de Uso contém a descrição, o detalhamento de cada interação de um ator com os casos de uso, com outros atores ou entre casos de uso. Nessa sessão serão apresentadas as descrições referentes aos casos de uso ilustrados nas Figuras 18 e 19.

4.4.1 Caso de Uso Programar Ordens de Serviço Para Execução

Nome do Caso de Uso	**Programar Ordem de Serviço Para Execução**
Caso de Uso Geral	
Ator Principal	Colaborador
Atores Secundários	
Resumo	Caso de Uso que descreve como é feita a distribuição das ordens de serviço para as equipes, respeitando sua especialidade.
Pré-Condições	A equipe deve estar previamente cadastrada no sistema.
Pós-Condições	A ordem de serviço está preparada para a execução.
Fluxo Principal	
Ações do Ator	**Ações do Sistema**
1. Informar o código da ordem de serviço.	
	2. Consultar a ordem de serviço e informar se a mesma já não está programada.
3. Preencher o campo “EQUIPE” com o número da equipe a qual esta ordem de serviço será destinada.	
Restrições / Validações / Regras de Negócio	
Fluxo Alternativo	
Ações do Ator	**Ações do Sistema**

4.4.2 Caso de Uso Informar Ocorrência

Nome do Caso de Uso	**Informar Ocorrência**
Caso de Uso Geral	
Ator Principal	Colaborador ou Usuário Comum
Atores Secundários	
Resumo	Caso de Uso que descreve como é feito o registro de uma ocorrência.
Pré-Condições	É necessário existir uma ocorrência a ser informada.
Pós-Condições	A ocorrência foi registrada e o protocolo foi informado.
Fluxo Principal	
Ações do Ator	**Ações do Sistema**
1. Solicitar o registro de uma ocorrência.	
	2. Exibir formulário de registro de ocorrências.
3. Preencher as informações referentes à ocorrência.	
	4. Informar o protocolo de registro de ocorrência.
Restrições / Validações / Regras de Negócio	
Fluxo Alternativo I – Ocorrência já cadastrada	
Ações do Usuário	**Ações do Sistema**
	1. Informar o protocolo da ocorrência.

4.4.3 Caso de Uso Manter Imóvel

Nome do Caso de Uso	**Manter Imóvel**
Caso de Uso Geral	
Ator Principal	Colaborador
Atores Secundários	
Resumo	Caso de Uso que descreve como o cadastro de imóveis recebe atualizações e manutenções, permitindo inclusão, listagem e alteração – não permite exclusão pela necessidade de se manter histórico.

Pré-Condições	A rua em que o imóvel se localiza deve estar previamente cadastrada no sistema.
Pós-Condições	O imóvel cadastrado no sistema e código do imóvel disponível para consulta.
Fluxo Principal	
Ações do Ator	**Ações do Sistema**
1. Solicitar o cadastro de um imóvel	
	2. Exibir o formulário de cadastro de imóveis.
3. Preencher as informações referentes às características e localização do imóvel.	
	4. Com base nas informações prestadas, verificar se imóvel já possui cadastro.
	5. Informar o código do imóvel.
Restrições / Validações / Regras de Negócio	1. Imóvel deve ter cadastro único (exceto em caso de condomínios ou excepcionalidades). Verificar pelo nome da rua e o número se o imóvel já possui cadastro.
Fluxo Alternativo I – Imóvel já cadastrado	
Ações do Ator	**Ações do Sistema**
	1. Informar o código do imóvel.

4.4.4 Caso de Uso Gerar Ordem de Serviço

Nome do Caso de Uso	**Gerar Ordem de Serviço**
Caso de Uso Geral	
Ator Principal	Colaborador
Atores Secundários	
Resumo	Caso de Uso que descreve como são geradas as ordens de serviço a partir da análise das ocorrências.
Pré-Condições	O imóvel ou rua para qual a ordem estará destinada devem estar cadastrados previamente no sistema.
Pós-Condições	Ordem de serviço gerada no sistema sem parecer de execução.

Fluxo Principal	
Ações do Ator	**Ações do Sistema**
1. Solicitar a inclusão de uma ordem de serviço para uma determinada ocorrência registrada.	
	2. Verificar se não há ordem de serviço de mesmo tipo para este imóvel (para rua não se aplica).
	3. Gerar o desdobramento e exibir o formulário para preenchimento dos dados da ordem de serviço.
Restrições / Validações / Regras de Negócio	
Fluxo Alternativo	
Ações do Ator	**Ações do Sistema**

4.4.5 Caso de Uso Encerrar Ordem de Serviço

Nome do Caso de Uso	**Encerrar Ordem de Serviço**
Caso de Uso Geral	Gerar Ordem de Serviço
Ator Principal	Colaborador
Atores Secundários	
Resumo	Caso de Uso que descreve como as informações da ordem de serviço de uma equipe de campo são registradas no sistema.
Pré-Condições	A ordem de serviço deve ser gerada anteriormente.
Pós-Condições	Ordem de serviço com parecer de execução.
Fluxo Principal	
Ações do Ator	**Ações do Sistema**
1. Solicitar o encerramento de uma ordem de serviço.	
	2. Verificar se a ordem estava programada.
	3. Apresentar formulário de encerramento da ordem.
4. Registrar as informações da ordem	

de serviço.	
Restrições / Validações / Regras de Negócio	1. A ordem deverá estar programada.
Fluxo Alternativo I – Ordem não programada	
Ações do Ator	**Ações do Sistema**
1. Preencher no formulário de encerramento o número da equipe que executou a ordem.	

4.4.6 Caso de Uso Notificar Usuário Alteração Status Ocorrência

Nome do Caso de Uso	**Notificar Usuário Alteração Status Ocorrência**
Caso de Uso Geral	
Ator Principal	Colaborador
Atores Secundários	
Resumo	Caso de Uso que descreve como o sistema notifica os usuários no momento em que ocorre uma alteração de status de uma ocorrência.
Pré-Condições	1. O usuário deverá ter um email válido cadastrado no sistema. 2. O usuário não pode ser anônimo.
Pós-Condições	Notificação de usuário enviada.
Fluxo Principal	
Ações do Ator	**Ações do Sistema**
	1. Verificar a alteração do status da ocorrência.
	2. Notificar o usuário enviando email referente à alteração de status.
Restrições / Validações / Regras de Negócio	
Fluxo Alternativo	
Ações do Ator	**Ações do Sistema**

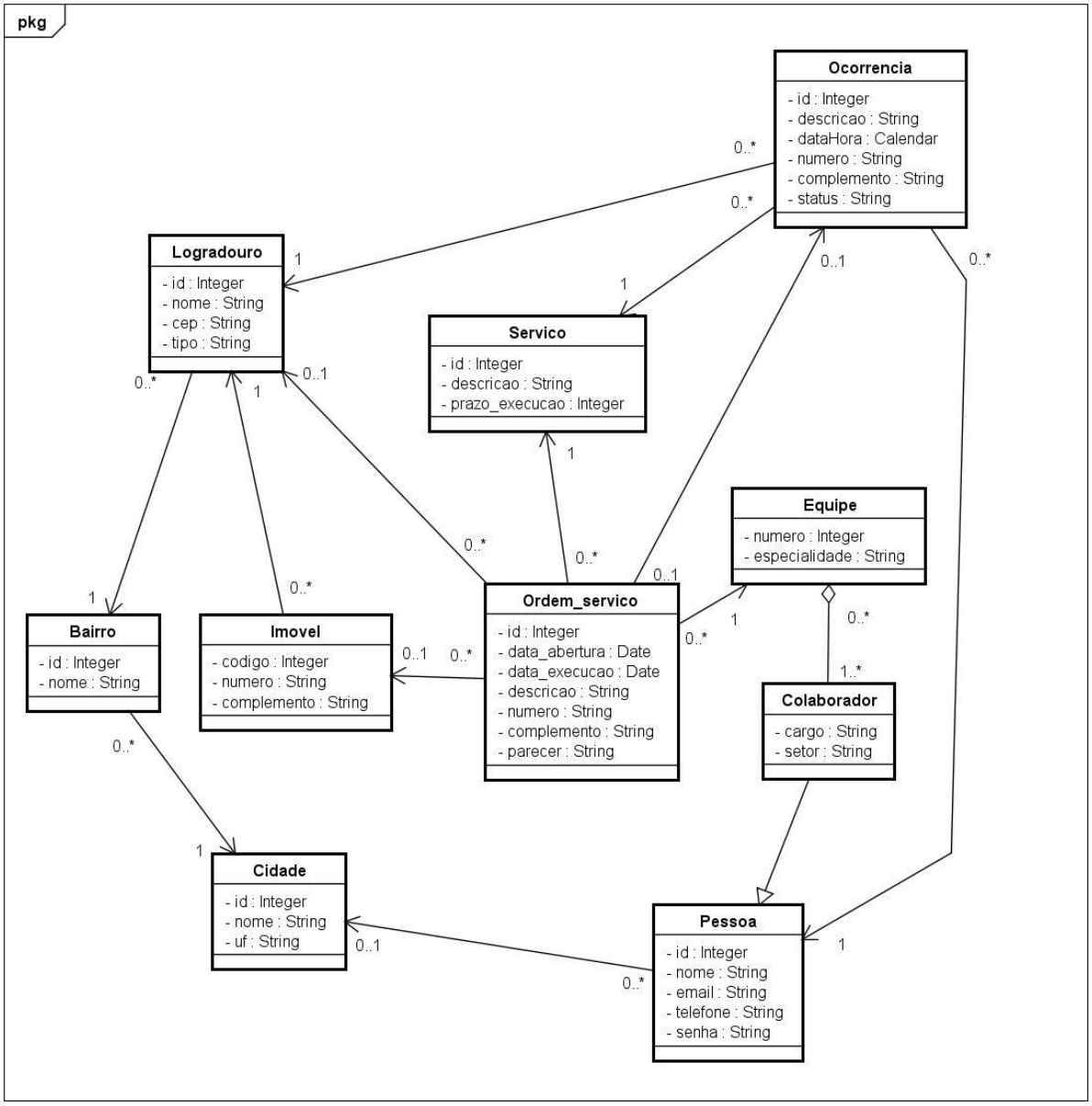
4.4.7 Caso de Uso Consultar Protocolo

Nome do Caso de Uso	**Consultar Protocolo**
Caso de Uso Geral	
Ator Principal	Usuário
Atores Secundários	
Resumo	Caso de Uso que descreve como um usuário pode consultar informações referentes a uma ocorrência informada por meio do protocolo de registro.
Pré-Condições	Deve existir um protocolo a ser informado.
Pós-Condições	Lista com as informações do protocolo informado.
Fluxo Principal	
Ações do Ator	**Ações do Sistema**
1. Solicitar a consulta por protocolo.	
	2. Exibir o formulário de preenchimento do protocolo.
3. Informar o protocolo.	
	4. Exibir as informações inerentes ao protocolo informado.
Restrições / Validações / Regras de Negócio	O cliente deve informar um protocolo válido.
Fluxo Alternativo	
Ações do Ator	**Ações do Sistema**

4.5 DIAGRAMA DE CLASSES

O diagrama de classes tem como principal finalidade ilustrar de que forma as classes estarão organizadas e relacionadas dentro da aplicação, listado seus atributos, métodos e relacionamentos. O diagrama apresentado na Figura 20 foi desenvolvido para ilustrar as características da persistência do sistema proposto.

Figura 20: Diagrama de Classes da camada de persistência



Fonte: Do Autor

A classe “Ocorrencia” possui o atributo “descrição”, que armazena a descrição do problema relatado pelo usuário. Os atributos “número” e “complemento” são utilizados para localização sendo inerente ao endereço de onde a empresa encontrará o objeto da solicitação do usuário. Já os atributos “dataHora”, “status” e “id” são vinculados ao andamento do processo do usuário, onde “dataHora” registrará a data e a hora do registro da ocorrência relatada pelo usuário e “status” exibe o andamento da solicitação, ou seja, em que nível de resolução essa solicitação se encontra além do atributo “id” que serve como número identificador e protocolo para posterior consulta ou auditoria.

A classe “Ordem_Servico” possui o atributo “tipo”, que designa se esta ordem diz respeito a um imóvel em específico (tipo “imóvel”) ou se trata de uma demanda identificada apenas pelo logradouro (tipo “logradouro”). O atributo “id” é o código identificador da ordem de serviço. Já o atributo “data_abertura” registra a data de inclusão da ordem de serviço no sistema, enquanto o atributo “data_execução” armazena a data em que o serviço foi de fato executado. O atributo “descrição” informa que atividade ou serviço deve-se realizar no endereço emitido na ordem, enquanto os atributos “número” e “complemento” compõe a informação referente ao endereço, sendo estes de preenchimento opcional. O atributo “parecer” armazena o parecer quanto ao que foi executado pela equipe de campo.

A classe “Imóvel” possui o atributo “id” para a identificação do imóvel, o atributo “número” que armazena o número real do imóvel e o atributo “complemento” que guarda informações adicionais referentes à localização do imóvel. A classe “Logradouro” possui o atributo identificador “id”, o atributo “nome” que designa o nome destinado ao logradouro, o atributo “tipo” que identifica parte da denominação do logradouro (rua, estrada, travessa) e o atributo “cep” que armazena o Código de Endereçamento Postal da via.

A classe “Bairro”, além do atributo identificador “id” contém o atributo “nome” que registra o nome do bairro. Já na classe “Cidade”, são registrados o nome da cidade por meio do atributo “nome” e a unidade federativa desta cidade por meio do atributo “uf”, além do atributo identificador “id”.

A classe “Pessoa” serve como elemento de herança para a classe “Colaborador”. A classe “Pessoa” compõe-se dos atributos “id”, como identificador, “nome” que armazena o nome da pessoa a ser registrada e “email” que contém um endereço e-mail para login no sistema, além da senha de acesso, registrada no atributo “senha” O telefone de contato do indivíduo será armazenado por meio do atributo “telefone”. Na classe “Colaborador”, serão requeridos o cargo do colaborador por meio do atributo “cargo” e seu setor de trabalho, armazenado no atributo “setor”.

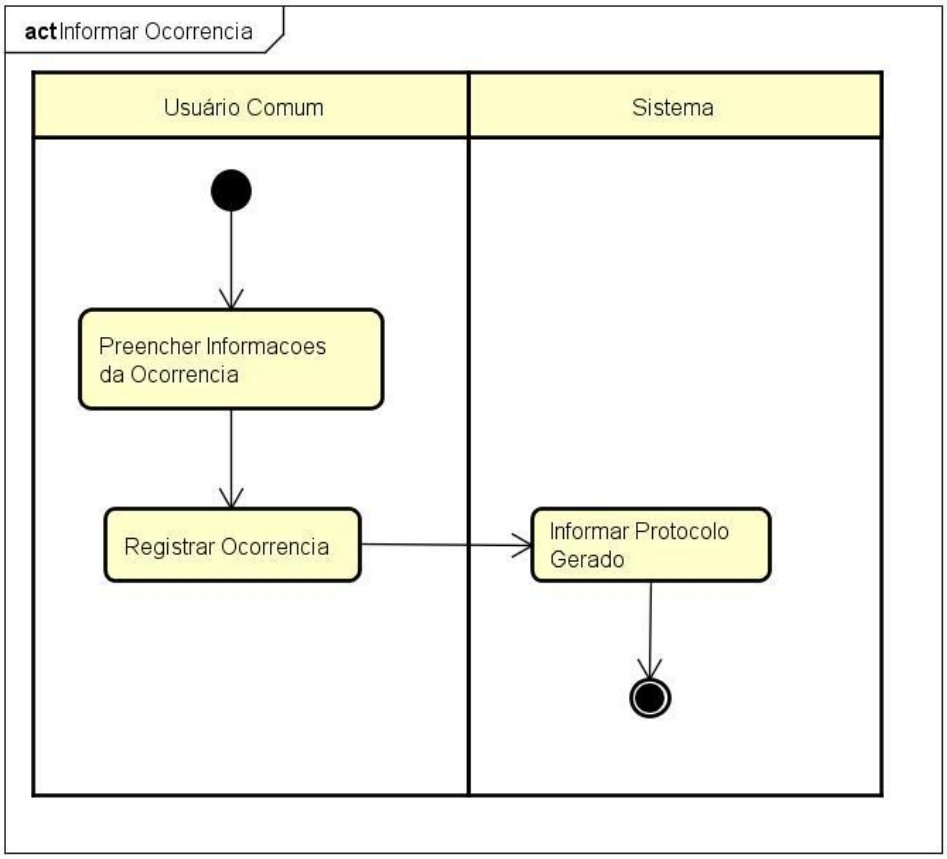
A classe “Serviço” é composta pelos atributos “id”, que será o identificador, “descrição” que apresentará o nome ou uma breve descrição sobre o serviço registrado e o atributo “prazo_execucao” que informará o número máximo, em dias, que este serviço poderá levar até sua plena conclusão. A classe “Equipe”, que representa os grupos de colaboradores a efetuarem as ordens de serviço, contará

com o atributo identificador “numero” e com uma breve descrição da especialidade da equipe, que será informada pelo atributo “especialidade”.

4.6 DIAGRAMAS DE ATIVIDADES

Os diagramas de atividades são fundamentais para a ilustração das etapas envolvidas em um determinado processo. Os diagramas de atividades das Figuras 21, 22 e 23 foram elaborados com base nos diagramas de casos de uso e diagrama de classes apresentados:

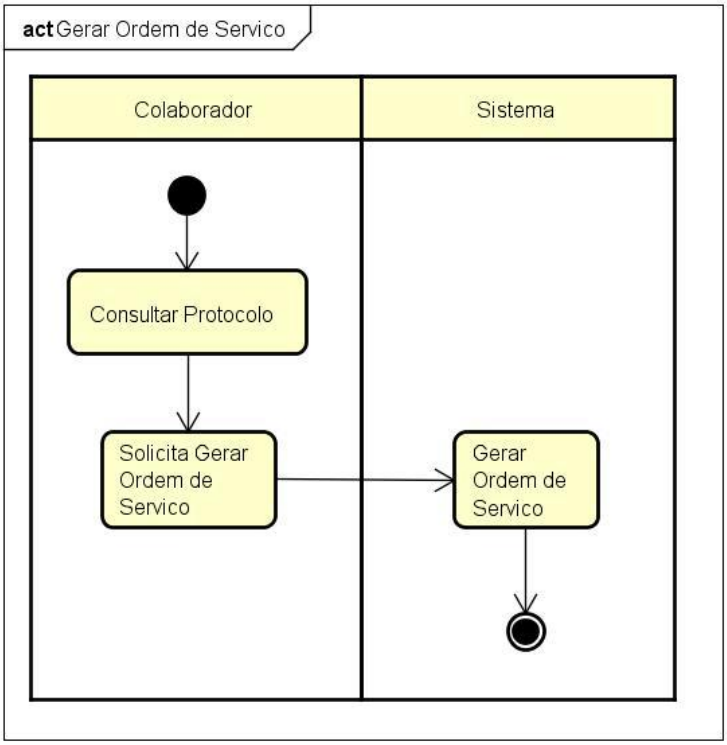
Figura 21: Diagrama de Atividades Informar Ocorrência



Fonte: Do Autor

O ciclo do diagrama apresentado inicia-se e no primeiro momento é requerido o preenchimento dos dados a ocorrência. Em seguida, o sistema persistirá a ocorrência informada e informará o id retornado do banco de dados para protocolo de registro.

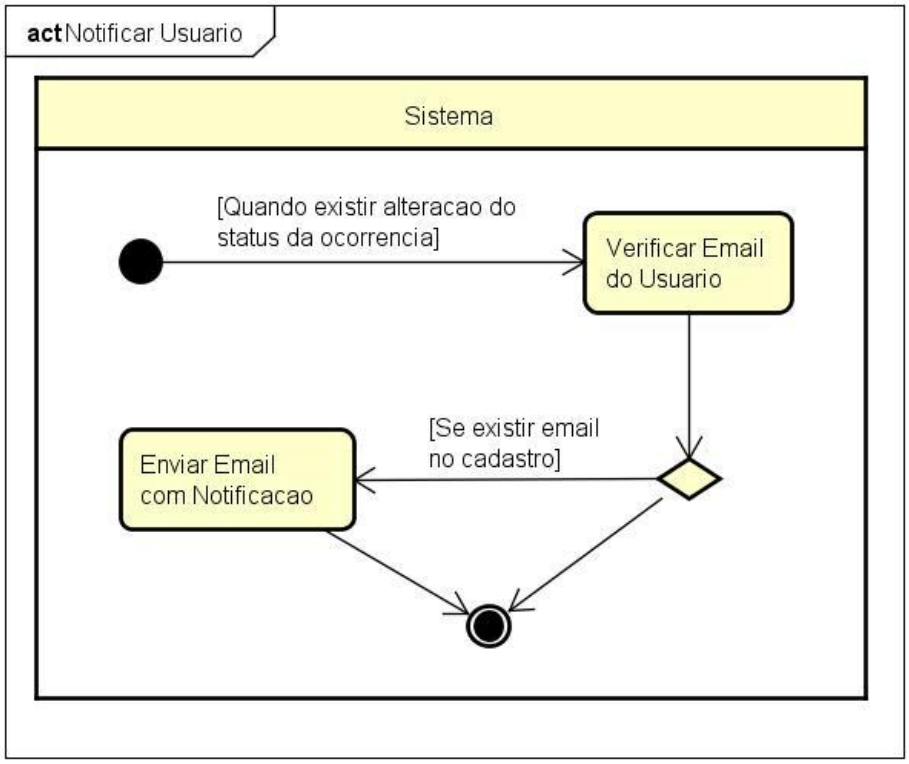
Figura 22: Diagrama de Atividades Gerar Ordem de Serviço



Fonte: Do Autor

No diagrama apresentado, o ciclo inicia-se pela consulta ao protocolo da ocorrência já registrada. Em seguida, o ator colaborador solicita a abertura de uma ordem de serviço para posterior execução. Então, o sistema emitirá uma ordem de serviço que posteriormente poderá ser programada ou diretamente encerrada.

Figura 23: Diagrama de Atividades Notificar Usuário



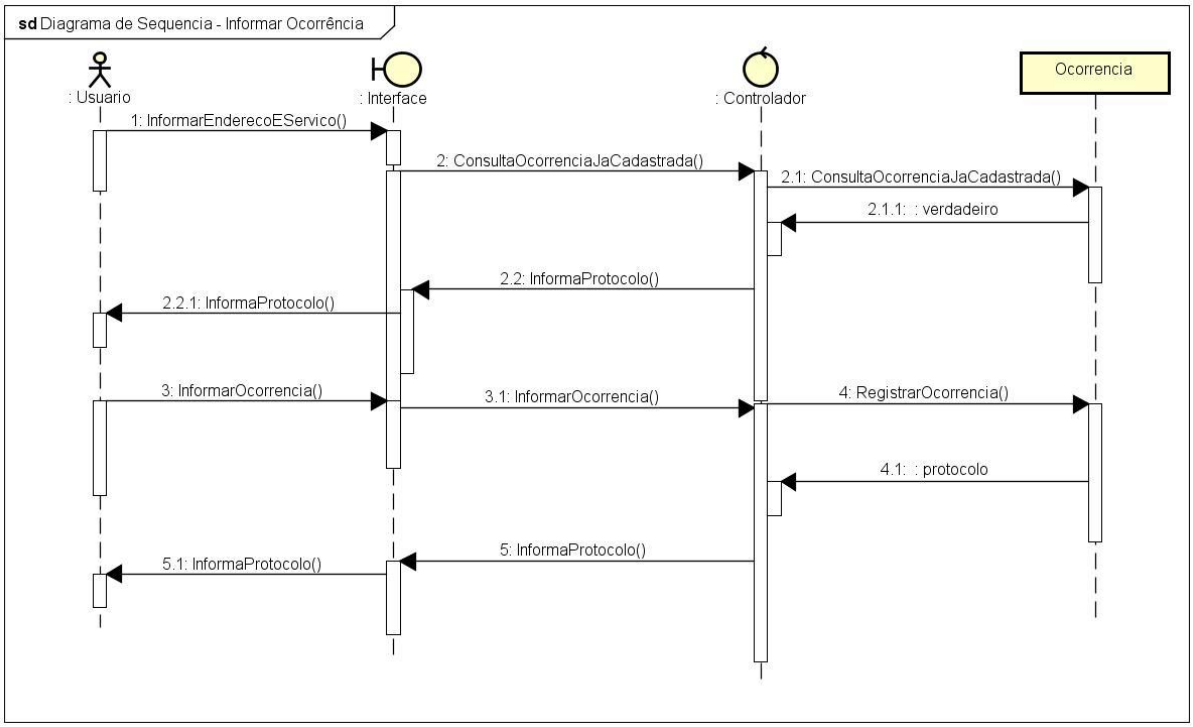
Fonte: Do Autor

O ciclo do diagrama apresentado se iniciará no instante em que houver uma alteração no status de uma ocorrência, devido ao encerramento de uma ordem de serviço ou a uma operação manual de alteração de status. Em seguida, o sistema verificará se existe um endereço e-mail cadastrado para o usuário relacionado à ocorrência. Se houver, o sistema enviará um e-mail informando o usuário sobre a alteração do status da ocorrência. Se não houver, encerra-se o ciclo.

4.7 DIAGRAMA DE SEQUÊNCIA

Os diagramas de sequência são relevantes para a ilustração da ordem temporal dos acontecimentos, seja envio de mensagens ou operações. O diagrama a presente na Figura 24 foi desenvolvido a partir dos diagramas de caso de uso e diagrama de classes:

Figura 24: Diagrama de Sequência Informar Ocorrência



Fonte: Do Autor

No diagrama de sequência apresentado, o usuário dispara uma mensagem solicitando informar uma ocorrência e informando os respectivos endereço e tipo da ocorrência. Ao receber a mensagem da interface, o controlador verifica se existe ocorrência já cadastrada para esse endereço com o mesmo tipo de serviço. Caso exista, retorna o protocolo, que será informado para o usuário pela interface. Caso não exista ocorrência similar, o sistema permitirá que o usuário informe os demais dados e conclua o informe de sua ocorrência. Ao receber os dados da interface, o controlador registra a ocorrência e fornece o protocolo da ocorrência, que será informado ao usuário por meio da interface.

5 DESENVOLVIMENTO

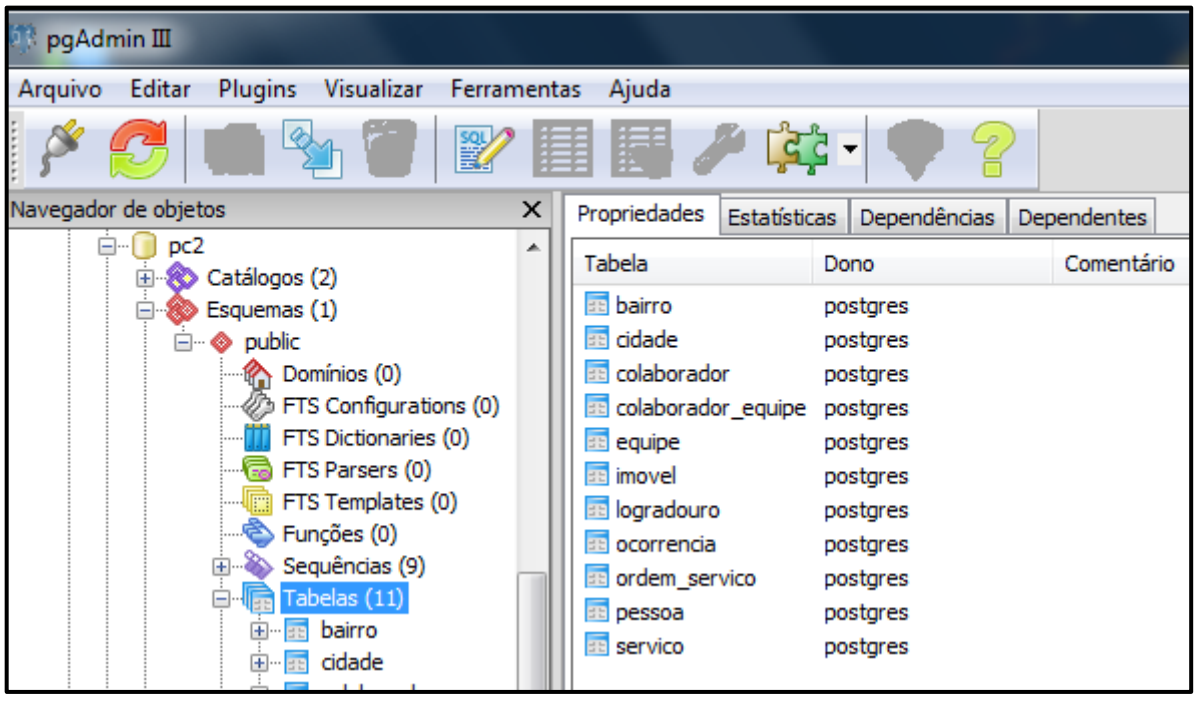
Nesta sessão será detalhado o processo de desenvolvimento da aplicação, a estrutura em camadas do software e o ambiente computacional envolvido em seu funcionamento.

5.1 AMBIENTE DE DESENVOLVIMENTO E RECURSOS

5.1.1 IDE e Servidores

Para o desenvolvimento da aplicação de controle de ocorrências operacionais, foram necessários dois servidores e a utilização de um software IDE (*Integrated Development Environment*) para a construção do código e gestão do sistema. Para a gerência e manutenção do banco de dados, foi utilizado o Sistema de Gerenciamento de Banco de Dados PostgreSQL dotado de seu servidor de banco de dados, o qual realizará o armazenamento dos dados da aplicação e a comunicação com a mesma. A Figura 25 apresenta o banco de dados da aplicação através da ferramenta PGAdmin III:

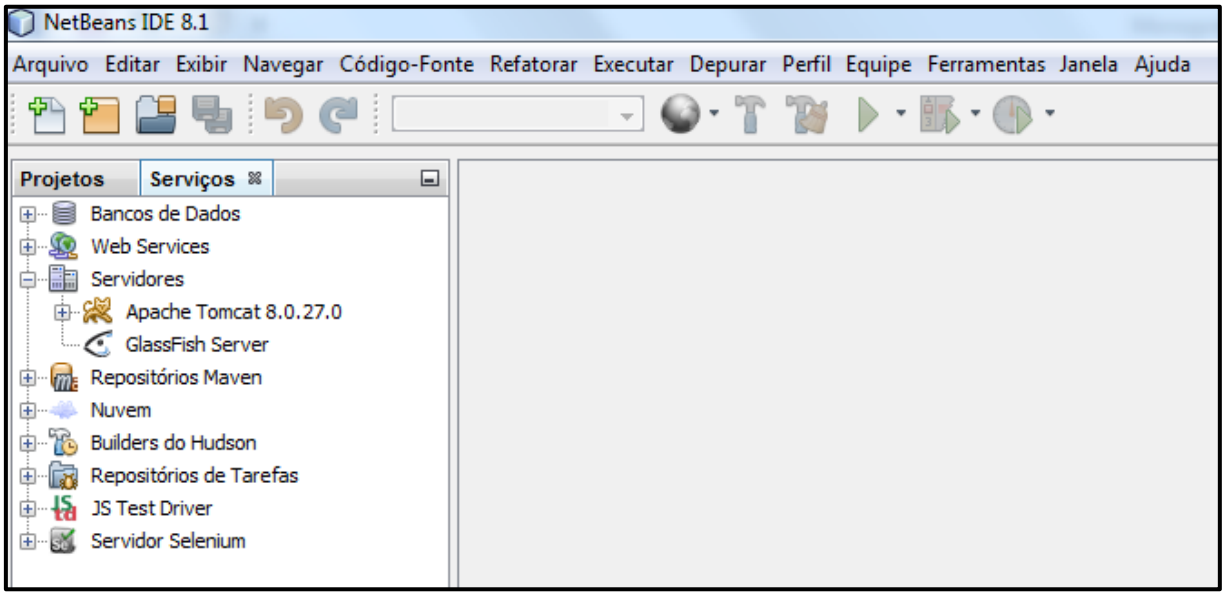
Figura 25: Banco de Dados da Aplicação



Fonte: Do Autor

O servidor de aplicação escolhido para a implantação do sistema foi o *GlassFish Server* versão 4.0, servidor de código fonte aberto mantido pela empresa Oracle. O *GlassFish Server* apresenta suporte aos contêineres EJB do Java, bem como as demais estruturas do Java Enterprise Edition como os *Enterprise Java Beans,* a *Bean Validation API* e a *Java Persistence API*, que foram utilizadas no projeto. A IDE escolhida para o desenvolvimento foi o NetBeans IDE 8.1, a qual possui integração com o servidor *GlassFish*. A IDE NetBeans 8.1 e o Servidor *GlassFish* podem ser visualizados na Figura 26:

Figura 26: IDE NetBeans e Servidor GlassFish Server



Fonte: Do Autor

Dentre os arquivos necessários para a utilização da JPA, o arquivo *persistence.xml* contém informações a respeito da unidade de persistência do projeto e as configurações para que a unidade de persistência possa consolidar a comunicação com o banco de dados. Neste arquivo, o elemento *<persistence-unit>* armazena o nome da unidade de persistência e a propriedade *transaction-type* deste elemento define a especificação de transação a ser utilizada, neste caso, a *Java Transaction API.* Por meio do elemento *<provider>*, define-se o Hibernate como provedor de persistência de dados e o elemento *<jta-data-source>* armazena o diretório (por meio da JNDI) e o nome da fonte de dados a ser utilizada. A propriedade *<property name="hibernate.hbm2ddl.auto" value="update"/>* define o método automático de atualização das tabelas do banco de dados e a propriedade

<property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect"/> define o “dialeto PostgreSQL” como padrão para a construção de consultas SQL. As classes que implementam objetos que fazem parte da persistência também são relacionadas neste arquivo. Um fragmento de código do arquivo *persistence.xml* é apresentado na Figura 27:

Figura 27: Fragmento do conteúdo do arquivo *persistence.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http:
  <persistence-unit name="PC2PU" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/pc2Web</jta-data-source>
    <class>br.edu.ifsul.modelo.Bairro</class>
    <class>br.edu.ifsul.modelo.Cidade</class>
    <class>br.edu.ifsul.modelo.Colaborador</class>
    <class>br.edu.ifsul.modelo.Equipe</class>
    <class>br.edu.ifsul.modelo.Imovel</class>
    <class>br.edu.ifsul.modelo.Logradouro</class>
    <class>br.edu.ifsul.modelo.Ocorrencia</class>
    <class>br.edu.ifsul.modelo.OrdemServico</class>
    <class>br.edu.ifsul.modelo.Pessoa</class>
    <class>br.edu.ifsul.modelo.Servico</class>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect"/>
      <property name="hibernate.transaction.jta.platform" value="org.hibernate.service.jta.
      <property name="hibernate.classloading.use_current_tccl_as_parent" value="false"/>
    </properties>
  </persistence-unit>
</persistence>
```

Fonte: Do Autor

O arquivo *glassfish_resources.xml* armazena informações que facilitam a criação do pool de conexões e do recurso JDBC no servidor GlassFish, os quais tornar-se-ão a fonte de dados da aplicação. Neste arquivo são armazenadas informações a respeito da autenticação exigida pelo banco por meio das propriedades *serverName* (nome do servidor), *portNumber* (número da porta de conexão), *dataBaseName* (nome do banco de dados), *User* (Usuário para login), *Password* (Senha), *URL* (“caminho” para efetuar a conexão) *e driverClass* (classe do driver JDBC). No elemento <*jdbc-resource*> são armazenados o diretório JNDI da fonte de dados e o nome do pool de conexões por meio das propriedades *jndi-name* e *pool-name*, respectivamente. Um fragmento do conteúdo do arquivo *glassfish_resources.xml* pode ser visualizado na Figura 28:

Figura 28: Fragmento de código do arquivo *glassfish_resources.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE resources PUBLIC "-//GlassFish.org//DTD GlassFish Application Server
<resources>
    <jdbc-connection-pool allow-non-component-callers="false" associate-with-th
        <property name="serverName" value="localhost"/>
        <property name="portNumber" value="5432"/>
        <property name="databaseName" value="pc2"/>
        <property name="User" value="postgres"/>
        <property name="Password" value="postgres"/>
        <property name="URL"
                  value="jdbc:postgresql://localhost:5432/pc2"/>
        <property name="driverClass" value="org.postgresql.Driver"/>
    </jdbc-connection-pool>
    <jdbc-resource enabled="true" jndi-name="jdbc/pc2Web" object-type="user"
                   pool-name="post-gre-sql_pc2Web_postgresPool"/>
</resources>
```

Fonte: Do Autor

5.1.2 Bibliotecas e Frameworks

Para desenvolver o controle de ocorrências operacionais, foram necessários recursos fornecidos por frameworks e bibliotecas de componentes desenvolvidas para o *Java Server Faces*. Estes recursos foram utilizados para a composição da interface de usuário, para validação e persistência de dados e para a comunicação entre a aplicação, o servidor e o sistema de banco de dados. As bibliotecas utilizadas no projeto são:

- Biblioteca Hibernate JPA 2.1.4.3.10

A biblioteca *Hibernate JPA* é uma das bibliotecas de persistência de dados que implementa a especificação *Java Persistence API* e foi utilizada como estratégia para o mapeamento objeto-relacional de objetos Java.

- Biblioteca Hibernate Validator 5.2.4

A biblioteca *Hibernate Validator*, que implementa a *Bean Validation API*, foi utilizada para a validação dos dados que compõe o objeto persistido, evitando inconsistências no banco de dados e na aplicação.

- Biblioteca BootsFaces 0.9.1

A biblioteca *BootsFaces* é uma biblioteca de elementos visuais para o JSF baseada no estilo visual da biblioteca *Bootstrap*. Sua utilização se deve ao fato de que seus componentes são preparados para utilização em *layout* responsivo, atendendo aos objetivos do projeto.

- Biblioteca PrimeFaces 6.0

A biblioteca *PrimeFaces* é uma biblioteca de elementos visuais já consolidada no desenvolvimento de interfaces com JSF. Foi utilizada como complemento à estrutura desenvolvida a partir da biblioteca *BootsFaces*.

- Driver JDBC PostgreSQL versão 9.4.1208

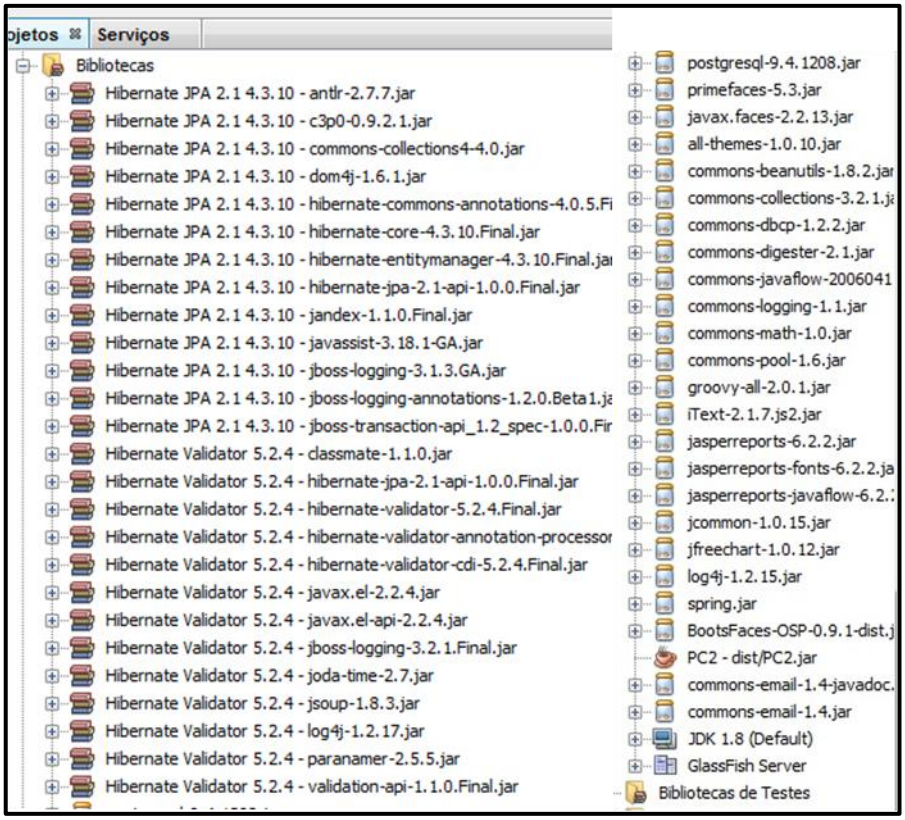
O *driver JDBC do PostgreSQL* fornece recursos para a conexão da aplicação com o banco de dados, permitindo a manipulação dos dados e manutenção das tabelas do banco.

- Biblioteca Commons Email 1.4

A biblioteca Commons Email é baseada na biblioteca JavaMail e fornece recursos para que uma aplicação Java envie e-mails de forma automática por meio de servidores SMTP. As configurações necessárias para o envio de e-mails resumem-se em apontar o servidor SMTP, configurar as credenciais de autenticação e formular a mensagem a ser enviada pela aplicação.

Na Figura 29 pode ser visualizada a estrutura de bibliotecas utilizada no projeto:

Figura 29: Estrutura de bibliotecas do projeto.



Fonte: Do Autor

5.2 ESTRUTURA E *LAYOUT* DA APLICAÇÃO

Nesta sessão serão explanadas a arquitetura em camadas do software e a composição do *layout* responsivo implementado.

5.2.1 Camada de Modelo

A camada de modelo da aplicação é responsável por armazenar as classes que definirão os objetos instanciados pelo sistema. Estes objetos são utilizados pela biblioteca de persistência para a execução do mapeamento objeto-relacional que irá compor o banco de dados do projeto. A partir da JPA, a biblioteca cria correspondências entre objetos e tabelas do banco, onde se baseia nas anotações presentes nos objetos para definir propriedades das tabelas e restrições para as informações que serão armazenadas no banco.

As classes, para que possam fazer parte da persistência, devem respeitar o padrão *JavaBeans*, que define a estrutura a ser seguida pela classe para sua persistência. Para ser persistida, a classe deve implementar a interface *Serializable* e deve conter atributos encapsulados, manipuláveis por métodos *getter* e *setter.* O código fonte da classe *Ocorrencia* está ilustrado na Figura 30:

Figura 30: Código da classe *Ocorrencia* da camada modelo

```
@Entity
@Table(name = "ocorrencia")
public class Ocorrencia implements Serializable {
    @Id
    @SequenceGenerator(name = "seq_ocorrencia", sequenceName = "seq_ocorrencia_id", allocationSize = 1)
    @GeneratedValue(generator = "seq_ocorrencia", strategy = GenerationType.SEQUENCE)
    private Integer id;

    @NotBlank(message = "Informe a descrição da ocorrência!")
    @Length(max = 50, message = "A descrição da ocorrência não deve ter mais que {max} caracteres!")
    @Column(name = "descricao", length = 50, nullable = false)
    private String descricao;

    @NotNull(message = "Data e Hora não foram registradas!") //o usuário não informa, é automático
    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "dataHora", nullable = false)
    private Calendar dataHora;

    @NotNull(message = "Informe o logradouro para a ocorrência!")
    @ManyToOne
    @JoinColumn(name = "logradouro", referencedColumnName = "id", nullable = false)
    private Logradouro logradouro;

    @Length(max = 10, message = "O número do imóvel não deve possuir mais que {max} caracteres!")
    @Column(name = "numero", length = 10)
    private String numero;
```

Fonte: Do Autor

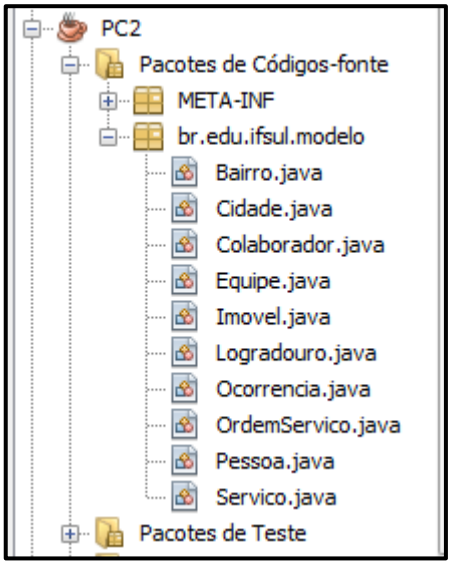
As anotações contidas no código da classe permitem que a biblioteca de persistência identifique propriedades dos elementos persistidos. A anotação *@Entity* identifica que a classe em questão é uma entidade e representa uma tabela do banco de dados, sendo o nome da tabela definido pela anotação *@Table(name = "nome_da_tabela").* Os atributos da entidade representam colunas na tabela do banco de dados onde o nome da tabela é apontado pela anotação *@Column*. A anotação *@Id* define que este atributo representa o identificador da tabela e no caso da entidade *Ocorrencia,* o valor do identificador é gerado automaticamente no banco de dados, por este motivo utiliza-se a anotação *@GeneratedValue* para defini-lo. A estratégia de geração dos valores é a geração sequencial, definida pela anotação *GenerationType.SEQUENCE* onde o gerador recebe a anotação *@SequenceGenerator.* A manipulação dos relacionamentos entre entidades se dá

por meio da anotação *@JoinColumn* (ou *@JoinColumns*) onde essa anotação contém o nome da coluna que será referenciada pela chave estrangeira.

Para a validação de informações armazenadas pelos atributos da entidade, existem anotações que verificam tipos e tamanho dos dados informados pela interface. A anotação *@NotBlank* permite a verificação de um dados em tipo *String* para que não se armazene valores em branco, assim como a anotação *@NotEmpty* define que não serão aceitos dados do tipo *String* com conteúdo vazio. A anotação *@NotNull* define que não serão aceitos valores nulos para um determinado atributo e anotação *@Lenght* registra o tamanho do conteúdo que será aceito para determinado atributo. *A* anotação *@Temporal* define que determinado atributo armazenará dados relacionados tempo (datas ou horas).

Para manter o projeto em conformidade com o que foi definido na modelagem, a camada de modelo do projeto contém as classes Bairro, Cidade, Colaborador, Equipe, Imovel, Logradouro, Ocorrencia, OrdemServico, Pessoa e Servico. A estrutura da camada de modelo do projeto foi apresentada na Figura 31:

Figura 31: Estrutura da Camada de Modelo do Projeto



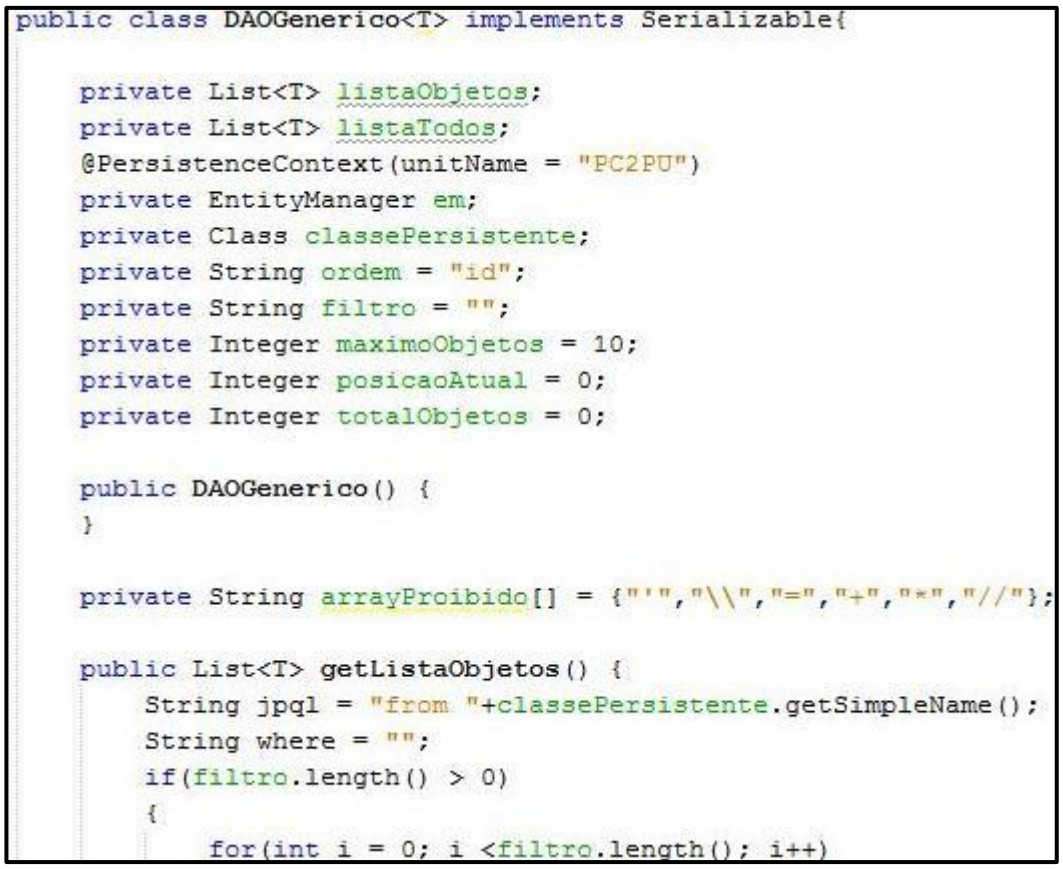
Fonte: Do Autor

5.2.2 Camada DAO

Na aplicação de controle de ocorrências operacionais, a camada DAO (*Data Access Object*) é responsável por gerenciar a comunicação entre o banco de dados, a unidade de persistência e os recursos da aplicação que utilizam os dados persistidos. Para efetivar a comunicação, a camada DAO utiliza um gestor de entidades chamado *EntityManager* (este que utiliza as classes da camada de modelo) e utiliza a unidade de persistência definida na anotação *@PersistenceContext,* presente no código da classe. Estabelecer a comunicação entre o banco de dados e a aplicação visa, sobretudo, viabilizar as principais operações executadas sobre os dados persistidos, que são a inclusão, exclusão, leitura e atualização. A disponibilização destes recursos foi feita por contêineres EJB devido a sua excelência em questões de segurança e gerenciamento de transações e concorrência.

No presente projeto, para permitir a reutilização de código, foi desenvolvida uma classe DAO genérica com métodos adaptados para utilização sobre as classes da camada de modelo, onde as classes especializadas podem utilizar essa estrutura por meio de herança entre classes. Alguns dos métodos implementados fornecem recursos como paginação, busca por registros no banco de dados e filtro e ordenação para resultados de pesquisas. Esse procedimento permitiu que as classes DAO específicas fossem desenvolvidas com estrutura simplificada, o que permite maior facilidade de manutenção no código. As classes específicas receberam a anotação *@Stateful,* para que guardem o estado de cada sessão e ambas as estruturas, genérica e específicas, implementam a interface *Serializable*. A Figura 32 possui um fragmento do código da classe *DAOGenerico.java*:

Figura 32: Fragmento de código da classe DAOGenerico.java

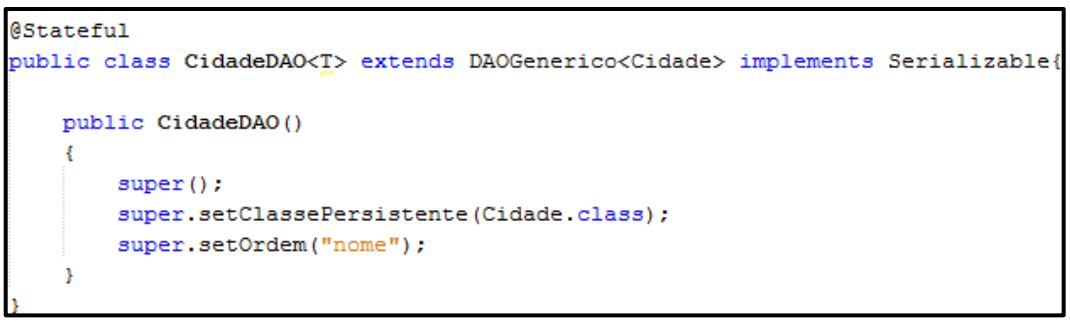


```
public class DAOGenerico<T> implements Serializable{

    private List<T> listaObjetos;
    private List<T> listaTodos;
    @PersistenceContext(unitName = "PC2PU")
    private EntityManager em;
    private Class classePersistente;
    private String ordem = "id";
    private String filtro = "";
    private Integer maximoObjetos = 10;
    private Integer posicaoAtual = 0;
    private Integer totalObjetos = 0;

    public DAOGenerico() {
    }

    private String arrayProibido[] = {"'","\\","=","+","*","//"};

    public List<T> getListaObjetos() {
        String jpql = "from "+classePersistente.getSimpleName();
        String where = "";
        if(filtro.length() > 0)
        {
            for(int i = 0; i <filtro.length(); i++)
```

Fonte: Do Autor

As classes DAO específicas recebem como parâmetro uma das classes persistentes da camada de modelo. Essa classe é utilizada como base para a execução dos métodos da classe *DAOGenerico.* O parâmetro que será utilizado na ordenação dos resultados de pesquisa também pode ser definido, mas é opcional. Para as peculiaridades de cada classe, foram implementados métodos nas classes específicas ou sobrescritos os métodos da classe genérica conforme a necessidade específica. O código da classe *CidadeDAO.java* foi apresentado na Figura 33.

Figura 33: Código da classe CidadeDAO.java



```
@Stateful
public class CidadeDAO<T> extends DAOGenerico<Cidade> implements Serializable{

    public CidadeDAO()
    {
        super();
        super.setClassePersistente(Cidade.class);
        super.setOrdem("nome");
    }
}
```

Fonte: Do Autor

5.2.3 Conversores

A interface das aplicações, usualmente não consegue manipular estruturas complexas das linguagens nem mesmo consegue convertê-las para que sejam exibidas em tela. Para exemplificar, pode-se citar que um dado do tipo *Calendar*, que armazena tipos temporais, quando exibido em tela necessita conversão para *String*, tendo em vista que os elementos visuais da interface não conseguem manipular dados em tipos temporais.

Para viabilizar a conversão de objetos *Java* para *Strings* e *Strings* para objetos *Java* foram implementados no sistema de ocorrências conversores que implementam a interface Converter do pacote *javax.faces.* Quando criado, o conversor implementa os métodos abstratos da interface Converter. O método *getAsObject()* recebe uma *String* como parâmetro e retorna um objeto da classe designada ao conversor. Já o método *getAsString()* recebe um objeto e retorna uma *String*. Cada classe criada representa um tipo de objeto que poderá ser convertido. Para que um conversor funcione, torna-se necessária a utilização de uma instância da classe *EntityManager* visto que o conversor necessita utilizar as estruturas da camada de modelo. A Figura 34 apresenta o código de um conversor de objetos do tipo Cidade para tipos *String* e também a conversão inversa.

Figura 34: Conversor para objetos do tipo *Cidade*

```
@Override
public Object getAsObject(FacesContext fc, UIComponent uic, String string) {
    if(string == null || string.equals("Selecione um registro"))
    {
        return null;
    }
    return em.find(Cidade.class, Integer.parseInt(string));
}

@Override
public String getAsString(FacesContext fc, UIComponent uic, Object o) {
    if(o == null)
    {
        return null;
    }
    Cidade obj = (Cidade) o;
    return obj.getId().toString();
}
```

Fonte: Do Autor

5.2.4 Camada de Controladores

A camada de controladores, presente no sistema de ocorrências operacionais, tem por objetivo consolidar a comunicação entre os componentes da interface e a camada de persistência. Os controladores fornecem métodos para que os componentes da camada de visão possam requisitar à camada DAO a persistência e manipulação dos dados informados pelo usuário nas listagens e formulários da interface. Inicialmente, para caracterizar a classe como *bean,* é necessário implementar a anotação *@ManagedBean* e um nome pelo qual a classe será requisitada, por meio do atributo *name*.

Os atributos que compõe esta classe podem ser EJBs da camada DAO, que fazem a manipulação das informações requisitadas ao banco de dados. Um *ManagedBean* tem seu ciclo também definido por anotações. Na presente aplicação, os *beans* foram criados com escopo de visão, por meio da anotação *@ViewScoped,* tendo seu ciclo limitado ao tempo de atividade da tela criada pela camada de visão. Das ações atribuídas ao controlador, pode-se verificar *listeners* para a inclusão de novos registros, alteração e exclusão de registros, execução de paginação nas listas além de ações específicas como preenchimento automático de campos texto com data e hora. A Figura 35 apresenta um fragmento de código do controlador controleLogin:

Figura 35: Fragmento de código do controlador controleLogin

```
@ManagedBean(name = "controleLogin")
@SessionScoped
public class ControleLogin implements Serializable{
    @EJB
    private PessoaDAO<Pessoa> dao;
    private Pessoa usuarioLogado;
    private Integer id;
    private String nome;
    private Colaborador colaboradorLogado;
    private boolean colaborador = false;
    private String email;
    private String senha;
```

Fonte: Do Autor

Os controladores também auxiliam a aplicação na execução de tarefas que envolvem a utilização de bibliotecas. As Figuras 36 e 37 ilustram como foram desenvolvidos os métodos que executam a validação de usuário e senha para

efetuar login no sistema, implementação do controlador *controleLogin* e como a aplicação notifica um usuário via e-mail a respeito da mudança de status da ocorrência informada, implementação do controlador *controleOcorrencia.*

Figura 36: Método *efetuarLogin* no controlador *controleLogin*

```
public String efetuarLogin()
{
    if(dao.login(email, senha))
    {
        if(colaborador == true)
        {
          colaboradorLogado = (Colaborador) dao.localizaPorEmail(email);
          if(colaboradorLogado.getCargo() == null)
          {
              colaboradorLogado = null;
              return "/login?faces-redirect=true";
          }
          else
          {
              nome = colaboradorLogado.getNome();
              id = colaboradorLogado.getId();
              UtilMensagens.mensagemInformacao("Login de colaborador efetuado
              return "/index?faces-redirect=true";
          }
        }
        else
        {
            usuarioLogado = dao.localizaPorEmail(email);
            nome = usuarioLogado.getNome();
            id = usuarioLogado.getId();
            UtilMensagens.mensagemInformacao("Login efetuado com sucesso!");
            return "/index?faces-redirect=true";
```

Fonte: Do Autor

O método *efetuarLogin()* não recebe parâmetros, mas se utiliza dos atributos *email* e *senha* provenientes da própria classe controle*Login* para a autenticação do usuário. O método utiliza uma instância da classe *PessoaDAO* para executar a verificação do e-mail e da senha e como resultado da verificação retorna uma *String* com o caminho de redirecionamento da página. Além disso, o valor de atributos auxiliares também é modificado, como por exemplo os atributos *colaboradorLogado,* que determina se o usuário autenticado é do tipo Colaborador, e *usuarioLogado*, que determina se o usuário autenticado é do tipo Pessoa.

A aplicação pode requisitar auxilio de bibliotecas externas para a execução de tarefas por meio da camada de controladores. A Figura 37 apresenta um exemplo dessa utilização:

Figura 37: Envio de e-mail por meio da biblioteca Commons Email

```
public String notificar(String endereco)
{
    SimpleEmail email = new SimpleEmail();

    try {
            email.setDebug(true);
            email.setHostName("smtp.email.com");
            email.setAuthentication("email@email.com","email");
            email.setSSL(true);
            email.setSmtpPort(465);
            email.setSSLOnConnect(true);
            email.addTo(endereco);
            email.setFrom("aplicacaosaneamento@gmail.com");
            email.setSubject("Alteração de Status de Ocorrência");
            email.setMsg("Informamos a alteração de status da ocorrência
            email.send();
            System.out.println("OK");
            return "Sim";

    } catch (Exception e) {
```

Fonte: Do Autor

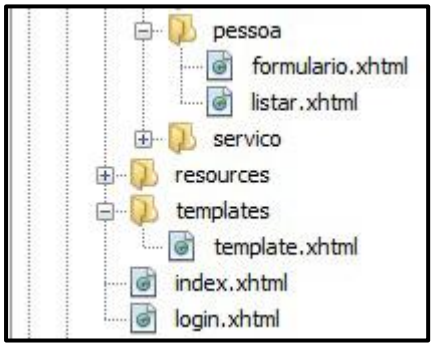
A modelagem da aplicação prevê que a cada alteração de status de uma ocorrência, uma notificação deve ser enviada ao usuário informante, se este estiver cadastrado. A forma de comunicação escolhida foi o e-mail e para a gestão automática desta demanda foram utilizados recursos da biblioteca *Commons Email.* No controlador *controleOcorrencia,* foi criado um método denominado *notificar(String endereco)* que possui um objeto do tipo *SimpleEmail*, uma das implementações da biblioteca *Commons Email*. Por meio do objeto *SimpleEmail* é possível enviar mensagens simples em texto por meio de servidores de e-mail SMTP. No método citado, foram definidos o servidor de envio do e-mail por meio do método *setHostName()*, os atributos de autenticação por meio do método *setAuthentication()*, a porta de conexão com o servidor por meio do método *setSmtpPort* e a utilização de SSL para conexão por meio dos métodos *setSSL()* e *setSSLOnConnect().* Para a configuração da mensagem, foram definidos o destinatário da mensagem por meio do método *addTo()*, o remetente por meio do método *setFrom()*, o assunto da mensagem enviada por meio do método *setSubject()* e a mensagem por meio do método *setMsg().* Por fim, para executar o envio, é invocado o método *send()* que finaliza a composição da mensagem e a encaminha ao servidor de e-mail.

5.2.5 Camada de Visão

A camada de visão da aplicação de gestão de ocorrências operacionais foi desenvolvida com o objetivo de abranger diferentes tamanhos de tela de dispositivos, implementando desta forma *layout* responsivo. A interface do sistema é composta de páginas *.xhtml* desenvolvidas a partir do framework *Java Server Faces* em conjunto com as bibliotecas *PrimeFaces* e *BootsFaces.* A utilização do framework e das bibliotecas permite a abstração do desenvolvimento com as linguagens *CSS* e *HTML,* o que visa simplificar a construção das páginas. No *layout*, adotou-se o estilo visual dos componentes da biblioteca *BootsFaces*, que é baseada na biblioteca *Bootstrap*, utilizada para aplicações web em geral.

Para facilitar a manutenção e reutilização de código foi utilizado o *Facelets* do JSF, que é uma linguagem de descrição de páginas (PDL) para o JSF. Com a utilização do *Facelets* os arquivos que compõe o *layout* da aplicação são importados para dentro do *template* criado por meio de *tags*. Na Figura 38, apresenta-se a estrutura de arquivos que compõe o *layout*:

Figura 38: Estrutura Geral da Interface



Fonte: Do Autor

O arquivo *template.xhtml* armazena o corpo do documento que compõe o *layout* básico da interface. Neste arquivo foram incluídas áreas editáveis que podem ser utilizadas por outros arquivos que compõe a interface da aplicação e que utilizam o template. Foi incluído o menu da aplicação por meio do elemento *<b:dropMenu>* e a área onde é apresentado o conteúdo das paginas foi incorporada ao documento por meio do elemento *<ui:insert>* identificada pelo *name* conteúdo, tornando-se uma região editável para ser utilizada pelas páginas que usam o template, onde podem

ser exibidas as listas de informações provenientes do banco de dados ou telas de edição de entidades. O código fonte do arquivo *template.xhtml* pode ser visualizado na Figura 39.

Figura 39: Fragmento do código arquivo *template.xhtml*

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:b="http://bootsfaces.net/ui"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
    <h:head>
        <title><ui:insert name="titulo">Sistema Móvel de Ocorrências</ui:insert></title>
   </h:head>
    <h:body>
        <b:container>
            <h:form id="formMenu">
                <b:navBar brand="Empresa de Saneamento" brand-href="/PC2Web/">
                    <b:navbarLinks ...19 linhas />
                </b:navBar>
            </h:form>
            <ui:insert name="conteudo">
            </ui:insert>
        </b:container>
    </h:body>
</html>
```

Fonte: Do Autor

O arquivo denominado *index.xhtml*, cujo código é apresentado na Figura 41, foi criado para armazenar tela inicial da aplicação. Neste arquivo foram utilizados elementos de composição do JSF por meio de *Facelets*, como o elemento *<ui:composition>* que foi utilizado para incorporar a página ao *template*. Também foram definidas áreas para título e conteúdo, por meio das *tags <ui:define>*.

Figura 41: Código do arquivo *index.xhtml*

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
    <ui:composition template="/templates/template.xhtml">
        <ui:define name="titulo">Sistema OCORRENCIAS</ui:define>
        <ui:define name="conteudo"></ui:define>
    </ui:composition>
</html>
```

Fonte: Do Autor

O menu do sistema, que tem seu código apresentado na Figura 40, disponibiliza os recursos fornecidos pelos métodos da camada de controladores. Cada item do menu aciona um método desenvolvido na camada de controladores que redireciona a navegação para a tela de manutenção da classe ou entidade em questão. A propriedade *rendered* do elemento *dropMenu* é utilizada para mostrar ou não determinadas funcionalidades, com base no tipo de usuário que está acessando. O código apresentado na Figura 40 representa o padrão utilizado para o desenvolvimento do menu da aplicação.

Figura 40: Código do menu *drop* do sistema

```
<b:navCommandLink action="#{controleOcorrencia.listar()}" style="cursor: pointer"
                  icon="glyphicon-comment" value="Ocorrências" ajax="false"/>
<b:navCommandLink style="cursor: pointer" icon="glyphicon-search" value="Busca Protocolo"
                  ajax="false" action="#{controleOcorrencia.buscar()}"/>
<b:dropMenu value="Cadastros" rendered="#{controleLogin.colaboradorLogado != null}"
            icon="glyphicon-list-alt">
        <b:navCommandLink style="cursor: pointer" icon="glyphicon-th" value="Bairros"
                          ajax="false" action="#{controleBairro.listar()}"/>
        <b:navCommandLink style="cursor: pointer" icon="glyphicon-th-large" value="Cidades"
                          ajax="false" action="#{controleCidade.listar()}"/>
        <b:navCommandLink style="cursor: pointer" icon="glyphicon-user" value="Colaboradores"
                          ajax="false" action="#{controleColaborador.listar()}"/>
        <b:navCommandLink style="cursor: pointer" icon="glyphicon-home" value="Imóveis"
                          ajax="false" action="#{controleImovel.listar()}"/>
        <b:navCommandLink style="cursor: pointer" icon="glyphicon-road" value="Logradouros"
                          ajax="false" action="#{controleLogradouro.listar()}"/>
        <b:navCommandLink style="cursor: pointer" icon="glyphicon-list-alt" value="Ordens de Serviço"
                          ajax="false" action="#{controleOrdemServico.listar()}"/>
        <b:navCommandLink style="cursor: pointer" icon="glyphicon-wrench" value="Serviços" ajax="false"
                          action="#{controleServico.listar()}"/>
</b:dropMenu>
```

Fonte: Do Autor

As áreas de interface que possuem listagem de registros possuem tabelas de dados, caixas de seleção e campos que permitem executar filtros para os registros apresentados. Também constam na listagem, botões que permitem ações de alteração e exclusão dos registros relacionados. No rodapé do formulário, foram desenvolvidos botões que permitem a navegação entre as páginas de registros. Quando uma lista de informações é requisitada, os controladores fazem requisições de dados por meio da camada DAO, enviando estes dados para a exibição em elementos *<p:dataTable>*. Incorporados às listagens, existem componentes visuais que executam métodos da camada de controle acesso às manutenções dos dados listados, como o que acontece nos elementos *<b:commandButton>*. Como exemplo, a Figura 42 apresenta o código de um formulário de listagem:

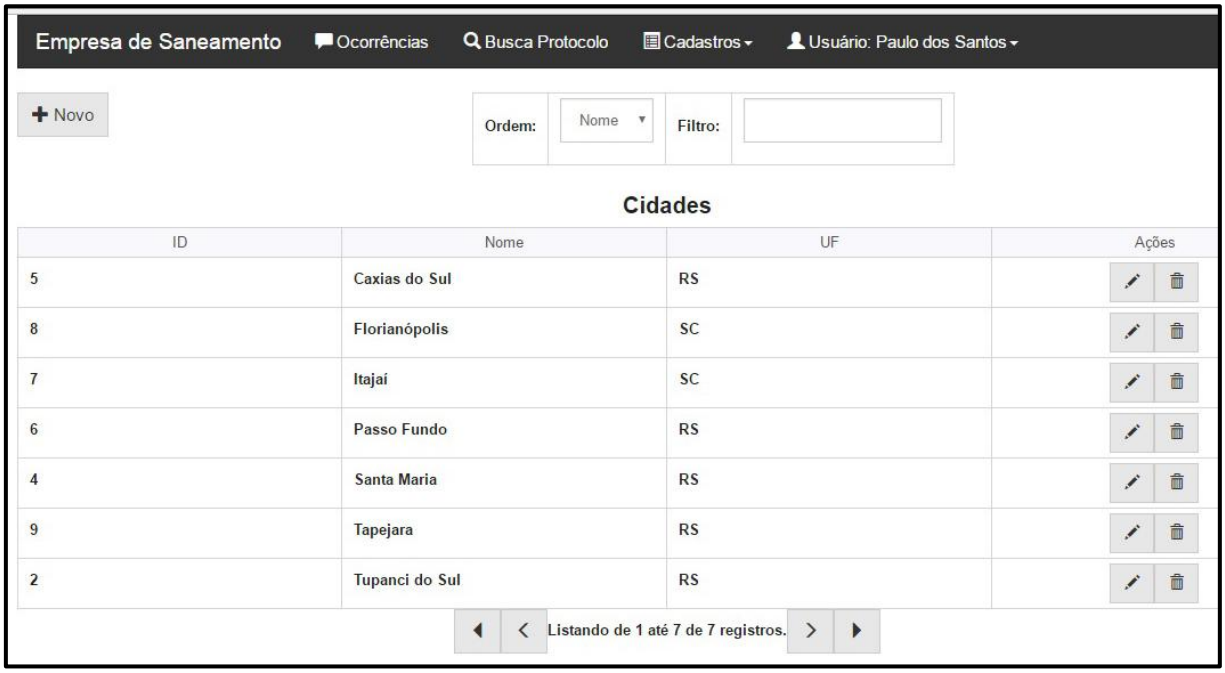
Figura 42: Fragmento de código do arquivo *listar.xhtml*

```
<ui:define name="titulo">Manutenção de Servicos</ui:define>
<ui:define name="conteudo">
    <h:form id="formListagem">
        <h:panelGroup rendered="#{!controleServico.editando}">
            <p:messages/>
            <div align="left" style="float: left" class="ui-fluid">
            <b:commandButton value="Novo" icon="glyphicon-plus"  actionListener="#{controleServico.novo()}"
                                        update=":formEdicao formListagem"/>
            </div>
            <div ...15 linhas />
            <div class="ui-fluid" align="center">
                <br/><p:outputLabel style="font-size: 20px;" value="Serviços"/>
            </div>
            <p:dataTable value="#{controleServico.dao.listaObjetos}" var="obj"
                         reflow="true" id="listagem">
                <p:column headerText="ID">
                    <p:outputLabel value="#{obj.id}"/>
                </p:column>
                <p:column headerText="Descrição">
                    <p:outputLabel value="#{obj.descricao}"/>
                </p:column>
                <p:column headerText="Prazo de Execução">
                    <p:outputLabel value="#{obj.prazo_execucao}"/>
                </p:column>
```

Fonte: Do Autor

O resultado visual da implementação do código da Figura 42, que implementa um formulário de listagem de dados pode ser observado na Figura 43:

Figura 43: Formulários de listagem para a sessão Serviços



Fonte: Do Autor

O formulário de listagem, ao receber um comando de edição ou inclusão de informações, pode requisitar que estes dados sejam manipulados em um formulário específico para edição, desenvolvido separadamente no arquivo *formulario.xhtml*. Nos formulários de edição do projeto foram utilizados campos <*b:input*> para a edição de informações baseadas em texto e campos <*b:selectOneMenu*> para a seleção de dados em formato de opções, como o que acontece em registros extraídos de relacionamentos entre tabelas do banco de dados. Para que os formulários do arquivo *listar.xhtml* possam requisitar um formulário de outro arquivo, foi utilizada a tag <*p:include*> para incorporação. O formulário de edição do cadastro de Serviços está ilustrado na Figura 44:

Figura 44: Fragmento de código do formulário de edição de Serviços

```
<f:facet name="header">
    <p:outputLabel value="Edição de Servicos"/>
</f:facet>
<b:inputText label="ID" id="txtID" value="#{controleServico.objeto.id}" readonly="true"
             size="10"/>
<b:inputText label="Descrição" id="txtDescricao" value="#{controleServico.objeto.descricao}"
             size="40" maxlength="40"
             required="true"/>
<b:inputText label="Prazo de Execução" id="prazo" value="#{controleServico.objeto.prazo_exe
             size="4" maxlength="4"
             required="true"/>
```

Fonte: Do Autor

O resultado visual da implementação do código da Figura 44, que implementa um formulário de edição de dados pode ser observado na Figura 45:

Figura 45: Formulário de edição para a sessão Serviços

Fonte: Do Autor

A interface do sistema foi desenvolvida a partir de regras de permissão de acesso, onde cada tipo de usuário visualiza somente os elementos autorizados para seu perfil. Ao usuário anônimo, ou seja, que não está autenticado, são disponibilizadas apenas funcionalidades de registro de nova ocorrência e pesquisa por protocolo e não é permitido nenhum tipo de listagem de informações (exceto a pesquisa por protocolo). O cadastro de novo usuário e o acesso à página de *login* também são disponibilizados para este perfil de usuário, o que pode ser visualizado na Figura 46:

Figura 46: Interface para usuário anônimo

Fonte: Do Autor

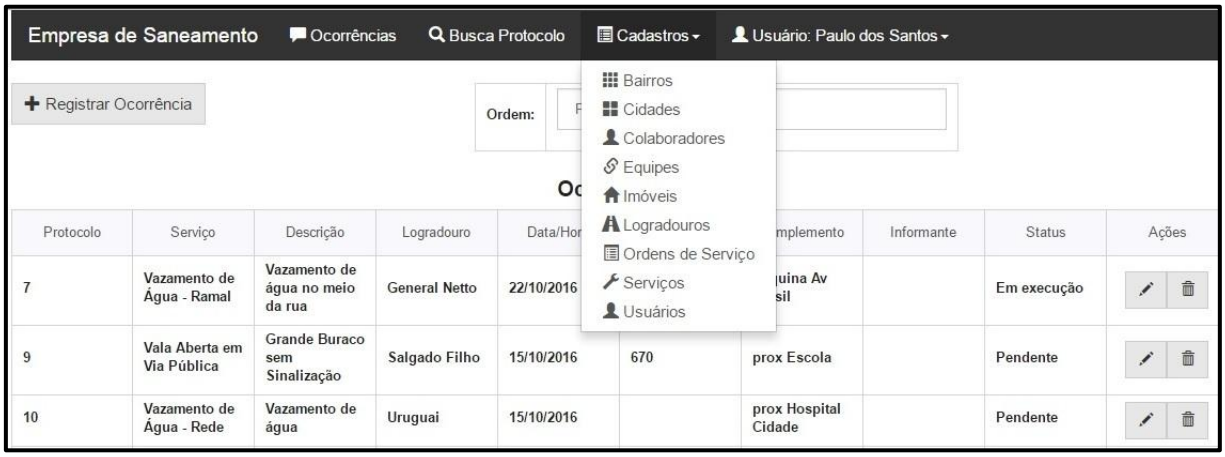
Para o usuário autenticado e não colaborador, são disponibilizadas as funcionalidades de registro de nova ocorrência com informante identificado (o que permite a notificação do usuário no momento em que o status de sua ocorrência for alterado), acesso ao cadastro de novos usuários, pesquisa por protocolo e listagem das ocorrências registradas pelo seu usuário, além do acesso à página de *login.* A Figura 47 apresenta a interface para o usuário autenticado:

Figura 47: Interface para usuário autenticado com lista de ocorrências do usuário

Fonte: Do Autor

O usuário colaborador tem amplo controle dos recursos da aplicação. Além de todas as funcionalidades listadas para os demais usuários, o usuário colaborador possui permissão para acesso às rotinas de manutenção de todos os cadastros da aplicação, tendo permissão de leitura, inclusão, exclusão e alteração de registros. O colaborador também recebe permissão para gerar relatórios das informações constantes na aplicação. A Figura 48 apresenta os variados controles disponibilizados para o usuário colaborador por meio de menus, listagens, pesquisa e opções de edição de informações.

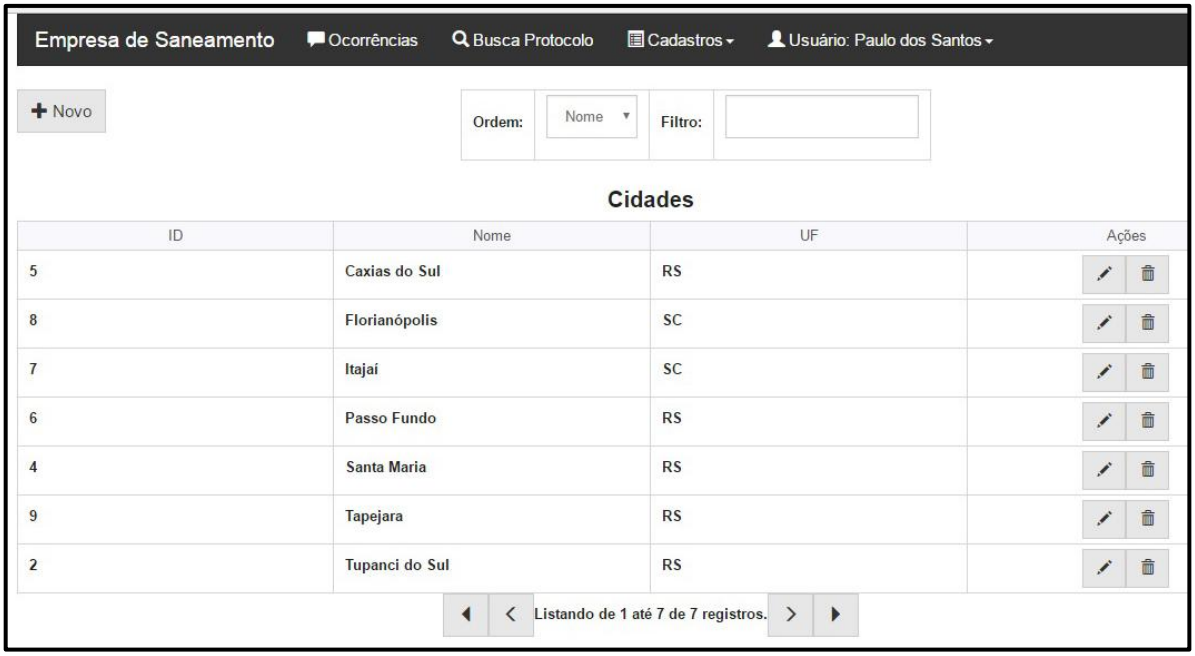
Figura 48: Interface para o usuário colaborador com listagem de ocorrências



Fonte: Do Autor

O desenvolvimento do *layout* da aplicação teve como objetivo atender ao requisito da modelagem que visava desenvolver uma aplicação com *layout* fluido. Desta forma, os elementos que compõe a interface foram extraídos das bibliotecas *PrimeFaces e BootsFaces*, atendem as necessidades requeridas para um *layout* responsivo. Por meio da propriedade class dos elementos div, foi definida a classe ui-fluid da biblioteca PrimeFaces para o dimensionamento dos quadros da interface, o que habilita cada elemento a receber diferentes estilos visuais, dependendo das dimensões da tela do dispositivo utilizado para o acesso. A figura 49 apresenta a renderização do *layout* para telas dimensionadas em 14 polegadas.

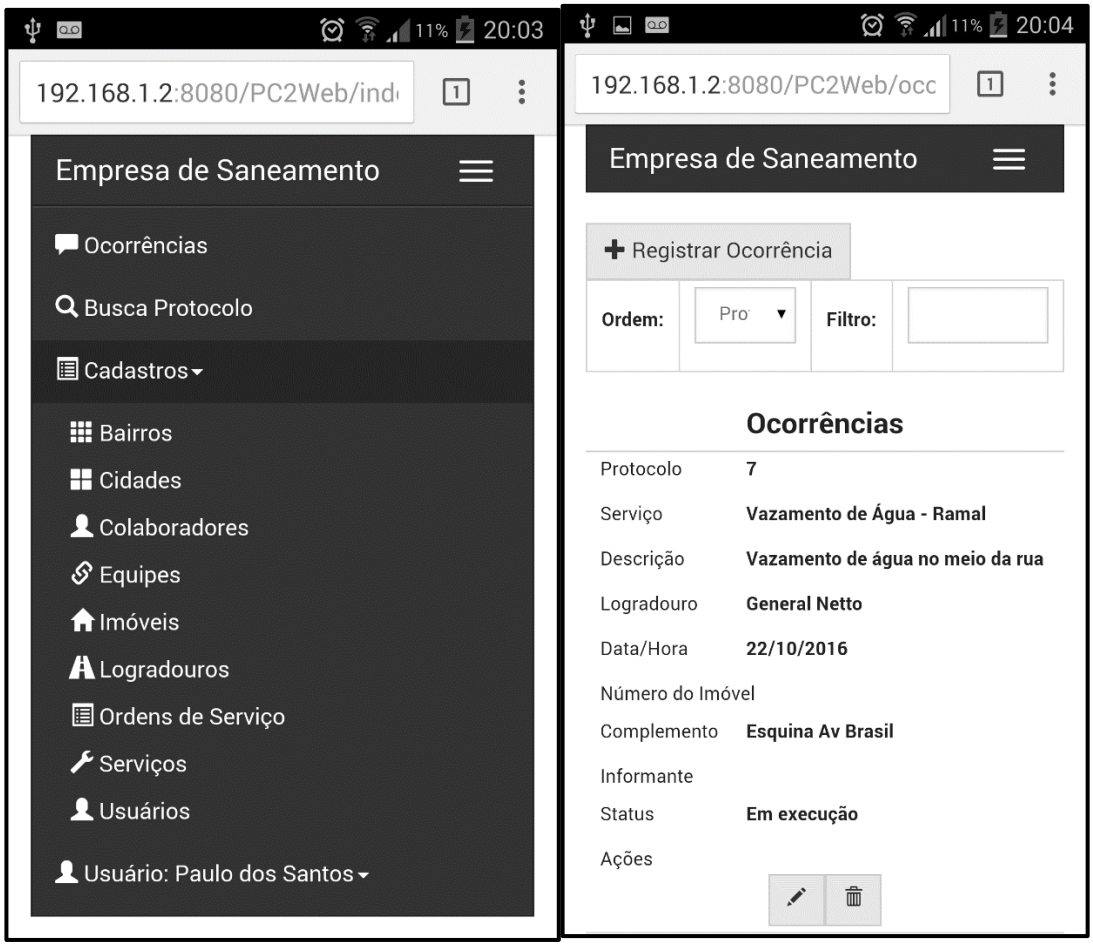
Figura 49: *Layout* para dispositivo com tela de 14 polegadas



Fonte: Do Autor

Na figura 50, apresenta-se a renderização do *layout* para sua utilização em um dispositivo com tamanho de tela de 5 polegadas, dimensões utilizadas no projeto para o teste do *layout* em dispositivos móveis. Na imagem a esquerda, pode ser visualizado o menu da aplicação, que antes localizava-se na área superior da tela e agora, quando acionado, acaba por ocupar grande parte da tela do aparelho. À direita, pode ser visualizado um dos formulários de listagem de informações da aplicação, no caso listagem de Ocorrências, com *layout* adaptado ao tamanho do dispositivo.

Figura 50: Menu e listagem para dispositivo com tela de 5 polegadas



Fonte: Do Autor

O desenvolvimento da aplicação buscou atender a todos os requisitos levantados e visou satisfazer a todas as regras de negócio pesquisadas para esta área de atuação. Cada área do sistema possui funcionalidades que auxiliam na integridade dos dados persistidos e na comunicação através de mensagens disparadas por eventos que acontecem no sistema de forma automática ou de forma manual.

6 CONSIDERAÇÕES FINAIS

O presente estudo teve como objetivo o desenvolvimento de uma aplicação de gestão de ocorrências operacionais para empresas de saneamento e o público em geral. Esta aplicação visou instituir um canal de comunicação alternativo às formas de comunicação convencionais utilizadas atualmente pelas empresas de saneamento. De semelhante modo, o projeto buscou contribuir com a redução do desperdício de recursos naturais como a água, que figura no contexto mundial como um dos recursos propensos a escassez em algumas décadas. O uso do software visou contribuir com a redução do custo operacional que envolve tratar água e esgoto devido à capacidade da aplicação em possibilitar maior agilidade na gestão das ocorrências informadas, permitindo que as empresas promovam mais rapidamente ações para atender às demandas registradas.

O desenvolvimento da aplicação buscou interação com os métodos atuais de gestão de informações utilizados pelas empresas, e por este motivo optou-se pelo desenvolvimento de uma aplicação web em virtude de muitas empresas possuírem sites institucionais. Ressalta-se o trabalho envolvido em todas as etapas do projeto, que iniciou-se com uma coleta de dados e levantamento de requisitos por meio de pesquisas e entrevistas. Posteriormente, a implementação do projeto abrangeu a construção da modelagem do sistema e findou-se com o desenvolvimento de uma aplicação direcionada a atender áreas variadas, com mecanismos de inteligência de operações, validação e persistência de dados. Para que fosse possível desenvolver uma aplicação com recursos variados, foi necessário o estudo e a utilização de tecnologias como o Java EE, as bibliotecas de validação e persistência de dados, a JPA e as bibliotecas de componentes de interface.

O estudo apresentou resultados compatíveis com o esperado, onde a aplicação abrangeu os requisitos apontados e demonstrou capacidade de atender aos objetivos do projeto. Em se tratando de um tema com grande relevância, a execução de novas pesquisas pode contribuir ainda mais com o desenvolvimento de soluções contra as agressões ao meio ambiente e que aplicações ainda melhores podem vir a ser desenvolvidas. Como sugestão para trabalhos futuros, cita-se a agregação do software desenvolvido a tecnologias de automação como por exemplo, as desenvolvidas a partir sistemas embarcados, que poderiam monitorar a pressão interna de um cano para detectar um possível vazamento. A implantação da

aplicação em um ambiente real de produção poderia permitir a qualificação dos recursos implementados no sistema a partir de situações reais. Contudo, a estrutura computacional da empresa entrevistada não era compatível com a arquitetura da aplicação desenvolvida e a empresa optou por não promover a execução do sistema em ambiente de produção.

7 REFERÊNCIAS

ARNOLD, Ken. GOSLING, James. **A Linguagem de Programação Java.** Bookman, 2009.

BEZERRA JR, David de Almeida. **Engenharia de Software: diagrama de atividades.** Disponível em <http://diariodainformatica.blogspot.com.br/2012/03/artigo-engenharia-de-software-diagrama.html>. Acesso em 17 de maio de 2016.

BOOTSFACES. **Dashboard**. Disponível em <http://showcase.bootsfaces.net/Examples/javax.faces.resource/dashboard.jpg.jsf>. Acesso em 21 de maio de 2016.

BOOTSFACES. **Integration with PrimeFaces**. Disponível em < http://showcase.bootsfaces.net/integration/PrimeFaces.jsf>. Acesso em 21 de maio de 2016.

BOOTSFACES. **Mobile First!** Disponível em < http://showcase.bootsfaces.net/layout/mobile.jsf>. Acesso em 21 de maio de 2016.

BOOTSFACES. **Quick Start**. Disponível em < http://www.bootsfaces.net/quickstart.jsf>. Acesso em 21 de maio de 2016.

BROWN, Tiffany B. BUTTERS, Kerry. PANDA, Sandeep. **Jump Start HTML5**. SitePoint, 2014.

CASAN. **Agualert é mais um canal de comunicação com a CASAN em Porto Belo e Bombinhas**. Disponível em < http://www.casan.com.br/noticia/index/url/agualert-e-mais-um-canal-de-comunicacao-com-a-casan-em-porto-belo-e-bombinhas#0>. Acesso em 10 de abril de 2016.

CASAN. **CASAN lança aplicativo para região de Porto Belo e Bombinhas**. Disponível em < http://www.casan.com.br/noticia/index/url/casan-lanca-aplicativo-para-regiao-de-porto-belo-e-bombinhas#0>. Acesso em 10 de abril de 2016.

GEARY, David. HORSTMANN, Cay S. **Core JavaServer Faces.** Pearson Education, 2010.

GONCALVES, Antonio. **Beginning Java EE 7**. Apress, 2013.

GUEDES, Gilleanes T. A. **UML 2 : uma abordagem prática**. São Paulo : Novatec Editora, 2009.

HEFFELFINGER, David R**. Java EE 7 Development with NetBeans 8.** Packt Publishing LTDA, 2015.

IBM Knowledge Center. **EJB 3 Architecture**. Disponível em < http://www.ibm.com/support/knowledgecenter/SSRTLW_9.1.1/com.ibm.javaee.doc/images/ejb3architecture.gif>. Acesso em 15 de maio de 2016.

JONNA, Sudheer. **Learning PrimeFaces Extensions Development**. Packt Publishing, 2014.

KROSING, Hannu. ROYBAL, Kirk. MLODGENSKI, Jim. **PostgreSQL Server Programming**. Packt Publishing, 2013.

MELO, Ana Cristina. **Desenvolvendo Aplicações com UML 2.2: do conceitual à implementação**. 3ª ed. Rio de Janeiro: Brasport, 2010.

NAÇÕES UNIDAS. **Assembleia Geral da ONU reconhece saneamento como direito humano distinto do direito à água potável.** Disponível em <https://nacoesunidas.org/assembleia-geral-da-onu-reconhece-saneamento-como-direito-humano-distinto-do-direito-a-agua-potavel/>. Acesso em 03 de abril de 2016.

ORACLE. **"Hello World!" for the NetBeans IDE**. Disponível em < https://docs.oracle.com/javase/tutorial/getStarted/cupojava/netbeans.html>. Acesso em 15 de maio de 2016.

ORACLE. **About the Java Technology**. Disponível em < https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html>. Acesso em 13 de maio de 2016.

PETERSON, Clarissa. **Learning Responsive Web Design: a begginer's guide**. O'Reilly, 2014.

POSTGRESQL. **About.** Disponível em <http://www.postgresql.org/about/>. Acesso em 21 de maio de 2016.

POSTGRESQL. **History.** Disponível em <http://www.postgresql.org/about/history/>. Acesso em 21 de maio de 2016.

POWERS, David. **Beginning CSS3**. Apress, 2012.

PRIMEFACES. **Why PrimeFaces**. Disponível em < http://www.primefaces.org/whyprimefaces>. Acesso em 21 de maio de 2016.

PRIMEFACES. **PrimeFaces ShowCase**. Disponível em < http://www.primefaces.org/showcase/index.xhtml>. Acesso em 21 de maio de 2016.

REDMONK. **The RedMonk Programming Language Rankings: January 2016**. Disponível em <http://redmonk.com/sogrady/2016/02/19/language-rankings-1-16/>. Acesso em 22 de maio de 2016.

SILVA, Maurício Samy. **Fundamentos de HTML5 e CSS3.** Novatec, 2015.

W3C. **HTML Responsive Web Design**. Disponível em <http://www.w3schools.com/html/html_responsive.asp>. Acesso em 17 de abril de 2016.

ZEMEL, Tárcio. **Web Design Responsivo: Páginas adaptáveis para todos os dispositivos.** Casa do Código, 2012.