UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

UM ESTUDO SOBRE SISTEMAS WEB APLICADO AO GERENCIAMENTO DE INFORMAÇÕES DA CENTRALPREV

TRABALHO DE GRADUAÇÃO

Douglas Machado Monteiro

Santa Maria, RS, Brasil
2008

UM ESTUDO SOBRE SISTEMAS WEB APLICADO AO GERENCIAMENTO DE INFORMAÇÕES DA CENTRALPREV

por

Douglas Machado Monteiro

Trabalho de Graduação apresentado ao Curso de Ciência da Computação,
da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial
para a obtenção do grau de
Bacharel em Ciência da Computação

Orientadora: Profª. Oni Reasilvia Sichonany

Co-orientador: Fabio Machado Monteiro

Trabalho de Graduação N. 246

Santa Maria, RS, Brasil

2008

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

UM ESTUDO SOBRE SISTEMAS WEB APLICADO AO GERENCIAMENTO DE INFORMAÇÕES DA CENTRALPREV

elaborado por
Douglas Machado Monteiro

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

Comissão Examinadora:

____
Oni Reasilvia Sichonany, Me. (UFSM)
(Presidente/Orientadora)

____
Roseclea Duarte Medina, Dra. (UFSM)

____
Iria Brucker Roggia, Me. (UFSM)

Santa Maria, 31 de janeiro de 2008.

AGRADECIMENTOS

Agradeço à minha família pela força, pelo apoio e incentivo durante todos os momentos de minha vida e por não medirem esforços para que eu recebesse uma educação de qualidade.

Gostaria de agradecer também ao meu irmão, por ser meu co-orientador durante o desenvolvimento deste trabalho e quem sugeriu o tema do mesmo. Além dele, quero agradecer à minha orientadora, a professora Oni, pela ajuda, pelas correções e pelo apoio que me foi dado. Sem a ajuda deles, este trabalho não seria possível.

Agradeço também aos meus colegas de faculdade, principalmente àqueles que foram verdadeiros amigos durante esses quatro anos. Aproveito para destacar os colegas Miguel, Rodrigo e Valdir pelo companheirismo, quer seja para noites em claro estudando para provas ou fazendo trabalhos ou para jogar videogame. O Elias pela amizade, pelas risadas e pela parceria, mas também por me ajudar em todos os momentos que precisei. O Patrick e o Rodrigo pela parceria das noites de tequila e idas ao DCE. A Marília, a Liane e a Carol pelo apoio, pelos tira-dúvidas momentos antes das provas e pela amizade.

A todos aqueles, que mesmo que não tenham sido citados aqui, mas que tiveram participação importante no meu crescimento, tanto profissional quanto pessoal.

RESUMO

Trabalho de Conclusão de Curso
Curso de Ciência da Computação
Universidade Federal de Santa Maria

UM ESTUDO SOBRE SISTEMAS WEB APLICADO AO GERENCIAMENTO DE INFORMAÇÕES DA CENTRALPREV

AUTOR: DOUGLAS MACHADO MONTEIRO
ORIENTADORA: ONI REASILVIA SICHONANY
Data e Local da Defesa: Santa Maria, 31 de Janeiro de 2008.

A CentralPrev é uma empresa especializada em questões previdenciárias e empresariais, que trabalha com concessão e revisão de benefícios previdenciários. Possui unidades distribuídas em vários municípios, sendo que cada uma realiza suas atividades de forma autônoma, mas todas são gerenciadas por uma matriz situada em Porto Alegre.

Devido ao grande número de clientes, a CentralPrev possui muitas informações que precisam ser gerenciadas. Dados sobre processos de clientes e informações sobre os funcionários e o funcionamento dos escritórios devem ser trocados diariamente entre as unidades de atendimento. Tais informações devem ser protegidas e o tráfego entre as unidades precisa ser facilitado para melhorar a realização das atividades dos escritórios. Atualmente, a grande distância entre as unidades torna o processo de troca de informações lento e necessita de interação humana de ambos os lados.

Com o intuito de melhorar o gerenciamento de informações da CentralPrev, será desenvolvido um sistema informativo, que funcionará como um serviço *web*, mantendo uma base de dados acessada remotamente. Para facilitar o controle das atividades realizadas nos escritórios e garantir coerência das informações, uma base unificada de dados se faz necessária. Dessa forma, é preciso desenvolver um sistema que possa alimentar a base de dados e gerar informações a partir dela. Esta base de dados conterá informações importantes, que terão acesso restrito garantido através de um serviço local. O serviço local também será responsável por prover uma interface com o usuário mais rica para *Internet*, permitindo maior controle e interatividade.

Palavras-chave: Serviço *web*; Gerenciamento de informações, RIA.

ABSTRACT

Final Undergraduate Work
Computer Science
Universidade Federal de Santa Maria

A STUDY ABOUT WEB SYSTEMS APPLIED TO CENTRALPREV'S INFORMATION MANAGEMENT

AUTHOR: DOUGLAS MACHADO MONTEIRO
ADVISOR: ONI REASILVIA SICHONANY
Date and Location of Presentation: Santa Maria, January 31st, 2008.

CentralPrev *is an enterprise specialized in pensions and business questions, that works with grant and review of pension benefits. It has distributed units in several towns, and each one of them realizes their activities in a standalone way, but they are all managed by a matrix in* Porto Alegre.

Due to the large number of customers, CentralPrev *has much information that must be managed. Data on customers' processes and information on the officials and the functioning of the offices must be exchanged daily between units of care. This information must have restricted access and the traffic between units has to be facilitated to improve the performance of the office's activities. Currently, the vast distance between the units makes the process of exchanging information slow and requires human interaction on both sides.*

In order to improve the information management of CentralPrev*, will be developed an information system, which acts as a web service, maintaining a single database which can be accessed remotely. To facilitate the control of the activities carried out in offices and ensure consistency of information, a unified data base is needed. Thus, we must develop a system that can feed the database and generate information from it. The database will contain important information, which will have restricted access guaranteed by a local service. The local service will also be responsible for providing an interface with the user richer for the Internet, allowing greater control and interactivity.*

Key-words: Web Service; Information Management, RIA

LISTA DE FIGURAS

Figura 2.1 – *Container* EJB gerencia os *beans* em execução. Adaptado de MUNDOOO (2007). 19
Figura 2.2 – Subsistema EJB: Os vários clientes e *beans* (SRIGANESH, 2006). 23
Figura 2.3 - Código em XML. 29
Figura 3.1 - Divisão lógica do sistema. 35
Figura 3.2 - Divisão física. 36
Figura 3.3 – Acesso ao sistema através de páginas *web*. 37
Figura 3.4 – Acesso restrito a rede local. 38
Figura 3.5 – Regras XML no processo de montagem de telas. 39
Figura 3.6 – Validação de dados. 40
Figura 3.7 – Campos obrigatórios. 40
Figura 3.8 - XML no carregamento de uma *Combo Box* 41
Figura 3.9 - Etapas da certificação de usuário. 42
Figura 3.10 - Objeto de usuário. 43
Figura 3.11 – Exemplo de exceção. 45
Figura 3.12 – Execução de um aspecto. 45
Figura 3.13 – Exemplo de regras de validação. 46
Figura 3.14 - Exemplo de autenticação e autorização. 48
Figura 3.15 – Exemplo de filtros de busca. 49
Figura 3.16 - Montagem do objeto de tela. 49
Figura 3.17 - Montagem dos dados para inserção. 50
Figura 3.18 - Exemplo de caso utilizando *lazy fetching*. 51
Figura 3.19 - Exemplo de mapeamento com anotações. 52
Figura 3.20 – Código *Java* com *Hibernate Validator*. 53

SUMÁRIO

1 INTRODUÇÃO 10
2 REVISÃO BIBLIOGRÁFICA 12
2.1 A Linguagem utilizada 12
2.1.1 *Java* para desenvolvimento *web* 12
2.1.1.1 *Servlets* 12
2.1.1.2 *Java Server Pages* 13
2.1.1.3 *Model-View-Controller* 13
2.1.1.4 *Tomcat* 14
2.1.1.5 *Java Server Pages Tag Library* 15
2.2 Tecnologias utilizadas 15
2.2.1*Hibernate* 15
2.2.2 *Enterprise JavaBeans* 17
2.2.2.1 *Container* EJB 19
2.2.2.2 Tipos de EJB 21
2.2.2.3 Visão geral de um componente EJB 24
2.2.3 *Spring* 24
2.2.3.1 Módulos do *Spring* 26
2.2.3.2 XML 27
3 O SISTEMA DE GERENCIAMENTO DE INFORMAÇÕES 32
3.1 As necessidades da CentralPrev 32
3.1.1 Distribuição geográfica das unidades de atendimento 32
3.1.2 Controle administrativo e estatístico 33
3.1.3 Controle financeiro 33
3.2 Modelagem e Implementação 34
3.3 Camadas do Sistema 35
3.4 Camada de Apresentação (cliente) 37
3.4.1 Parte Visual (*View*) 39
3.4.2 Parte de Controle (*Controller*) 41
3.4.2.1 Controle de ações do usuário 42
3.4.2.2 Primeira Validação 44
3.4.2.3 Execução de aspectos 44

3.4.2.4 Acesso a base local 45
3.4.2.5 Regras de validação 46
3.4.3 Modelos de Dados (*Model*) 47
3.5 Camada de serviços 47
3.6 Camada de dados (Banco de dados) 50
4 CONCLUSÃO 54
REFERÊNCIAS BIBLIOGRÁFICAS 55

1 INTRODUÇÃO

A CentralPrev é uma empresa especializada em soluções previdenciárias e empresariais, que trabalha com concessão e revisão de benefícios previdenciários. Com grande atuação em todo o estado do Rio Grande do Sul, possui unidades distribuídas em mais de 20 municípios. Cada uma delas realiza suas atividades de forma autônoma, mas todas são gerenciadas por uma matriz baseada em Porto Alegre. Apesar de funcionarem de forma independente, os escritórios trocam informações constantemente.

Devido à grande distância entre as unidades, o processo de troca de informações se dá de forma lenta e necessita de interação humana de ambos os lados. Visto que todas as informações precisam ser centralizadas na matriz, isso acaba se tornando um gargalo, prejudicando o bom funcionamento do sistema.

Por ter um grande número de clientes, a CentralPrev possui muitas informações que precisam ser gerenciadas. Dados sobre processos de clientes e informações sobre os funcionários e o funcionamento dos escritórios devem ser trocados diariamente entre as unidades de atendimento. Tais informações necessitam ter acesso restrito e o tráfego entre as unidades precisa ser facilitado para melhorar a realização das atividades dos escritórios.

Segundo Adriana Beal (2007):

> "O principal benefício que a tecnologia da informação traz para as organizações é a sua capacidade de melhorar a qualidade e a disponibilidade de informações e conhecimentos importantes para a empresa, seus clientes e fornecedores. Os sistemas de informação mais modernos oferecem às empresas oportunidades sem precedentes para a melhoria dos processos internos e dos serviços prestados ao consumidor final."

O ambiente empresarial está mudando continuamente e cada vez mais dependente de informação e de toda a infra-estrutura tecnológica, que permite o gerenciamento de enormes quantidades de dados.

Buscando melhorar o gerenciamento das informações da CentralPrev, foi desenvolvido um sistema informatizado capaz de administrar suas atividades e informações. Para permitir o compartilhamento dessas informações, o sistema funciona como um serviço *web*, mantendo uma base de dados que pode ser acessada remotamente. Visando uma maior segurança, as unidades de atendimento possuem serviços locais que restringem o acesso. Estes serviços locais provêm acesso aos serviços remotos, bem como uma base de dados estáticos utilizados apenas para consulta, aumentando o desempenho. O serviço local também

é responsável por prover uma interface com o usuário mais rica para *Internet* (RIA), permitindo maior controle e interatividade. Interfaces ricas são aplicações *web* que possuem recursos e funcionalidades iguais aos demais aplicativos de *desktop* (MOORE, 2007). As RIAs transferem o processo necessário de interface de usuário para o cliente *web* gerenciando as informações no servidor de aplicação.

O objetivo geral deste projeto é através do estudo da Tecnologia da Informação (TI) desenvolver um sistema informatizado capaz de integrar os escritórios da CentralPrev, auxiliando na realização de suas tarefas.

Para alcançar este objetivo, existem alguns passos importantes que devem ser executados, dentre eles:

- Criar uma base unificada de dados;
- Facilitar o gerenciamento dos escritórios;
- Melhorar o atendimento ao cliente;
- Facilitar o controle de documentação.

No capítulo 2 será apresentada a revisão bibliográfica contendo as principais características e conceitos sobre as tecnologias utilizadas, o *Hibernate*, EJB e o *Spring*, bem como a linguagem Java que foi utilizada no desenvolvimento.

O capítulo 3 apresenta o Sistema *Web*. Primeiramente será exposto o problema que motivou a criação deste projeto. Posteriormente será apresentado o sistema *web* e como ele foi distribuído. Finalizando este capítulo serão expostas as partes que formam o sistema e quais as vantagens de utilizar esta solução.

O capítulo 4 apresenta a conclusão deste trabalho, bem como as perspectivas de trabalhos futuros.

2 REVISÃO BIBLIOGRÁFICA

Este capítulo contextualiza a pesquisa bibliográfica apresentando as tecnologias, a linguagem utilizada e também as ferramentas de apoio. Inicialmente serão apresentadas as ferramentas que permitem o desenvolvimento de aplicações para a *Internet* em conjunto com a linguagem de programação escolhida. Logo após, será apresentado o *Hibernate*, um *framework* de acesso a banco de dados. Nas seções seguintes serão apresentados os *frameworks* EJB e *Spring*. EJB provê rápido e simplificado desenvolvimento de aplicações. *Spring* é responsável por fazer a interface entre a lógica de programação e o modelo de dados.

2.1 A Linguagem utilizada

2.1.1 *Java* para desenvolvimento *web*

No princípio, a *Internet* era uma dúzia de páginas estáticas contendo *sites* de pesquisa de diversas academias. A partir da necessidade de gerar conteúdo dinâmico, uma idéia bem simples hoje em dia, surgiram os primeiros programas de *Common Gateway Interface* (CGI).

Através de linguagens como C, PHP, *Delphi*, entre outras, foi possível gerar conteúdo que permite ao usuário acesso a diversas funcionalidades através de páginas HTML. Para melhorar o desempenho dessas páginas, existe o *servlet*, uma nova forma de trabalhar com requisições de clientes via *web* que economiza o tempo de processamento de uma chamada e a memória que seria gasta para tal processo, além de ser em Java e possuir todas as vantagens e facilidades da orientação a objeto.

A seguir, são descritas algumas das ferramentas e *frameworks* utilizados no desenvolvimento de programas *web*.

2.1.1.1 *Servlets*

Um *Servlet* funciona como um pequeno servidor que recebe chamadas de diversos clientes (GONÇALVES, 2006). Além disso, *servlets* são portáveis tanto quanto qualquer

programa escrito em Java, e aqueles que programam *servlets* não precisam mais se preocupar com a funcionalidade do servidor, o qual contém uma API[1] que abstrai e disponibiliza os recursos do servidor *web* de maneira simplificada para o programador.

2.1.1.2 *Java Server Pages*

Java Server Pages (JSP) funciona como uma especialização do *servlet* que permite que conteúdo dinâmico seja facilmente desenvolvido (FIELDS, 2000). Esta tecnologia permite ao desenvolver de páginas para *Internet* produzir aplicações que acessam banco de dados, manipulam arquivos e capturam informações em formulários ou sobre o visitante e o servidor. JSP nada mais é que um arquivo baseado em HTML que contém também código *Java*.

Apesar da grande vantagem em usar código *Java* junto com HTML, o código final pode ser pouco legível para outros programadores que venham a reutilizar ou modificar o mesmo. Para qualquer alteração necessária será preciso conhecimento em *Java* para saber onde e o que deve ser alterado.

Tendo em vista esse problema, uma abordagem mais interessante é utilizar as vantagens do *Servlet* e do JSP. O ideal é que o *Servlet* faça o acesso ao banco de dados e outros enquanto o JSP apenas apresente os resultados (GONÇALVES, 2006). Em outras palavras, o *Servlet* possui as regras de negócio e o JSP tem a lógica de apresentação. Dessa forma o código *Java* fica destinado ao *Servlet* e o código HTML ao JSP.

2.1.1.3 *Model-View-Controller*

A partir desse modelo, a separação permite que seja definida cada uma das partes da arquitetura. A parte responsável por apresentar resultados na página *web* é a Apresentação (*View*). O *Servlet* que gerencia quem deve executar as tarefas determinadas é a parte

---

[1]Nota: API, de *Application Programming Interface* é um conjunto de rotinas e padrões estabelecidos por um *software* para utilização de suas funcionalidades por programas aplicativos.

Controladora (*Controller*). As classes, que representam entidades e as que auxiliam no armazenamento e busca de dados, são chamadas de Modelo (*Model*).

Juntas, as três partes formam o padrão de arquitetura de *software* chamado *Model-View-Controller* (MVC) (WALLS, 2005). O MVC garante a separação das tarefas, de forma que alterações na interface não afetarão a manipulação de dados. Isso facilita a manutenção ou reescrita do código. MVC é usado em padrões de projeto de *software*, mas MVC abrange mais da arquitetura de uma aplicação do que é típico para um padrão de projeto (WIKIPEDIA, 2007).

Ainda que existam diferentes formas de MVC, o controle de fluxo geralmente funciona da seguinte maneira:

1. O usuário interage com a interface de alguma forma (por exemplo, ao apertar um botão);

2. O *Controller* manipula o evento da interface do usuário através de uma rotina pré-escrita;

3. O *Controller* acessa o *Model*, possivelmente atualizando-o de uma maneira apropriada, baseado na interação do usuário (por exemplo, atualizando os dados de cadastro do usuário);

4. Algumas implementações de *View* utilizam o *Model* para gerar uma interface apropriada (por exemplo, mostrando na tela os dados que foram alterados juntamente com uma confirmação). A *View* obtém seus próprios dados do *Model*. O *Model* não toma conhecimento direto da *View*;

5. A interface do usuário espera por próximas interações, que iniciarão o ciclo novamente.

2.1.1.4 *Tomcat*

Uma página criada com a tecnologia JSP, após instalada em um servidor de aplicação compatível com a tecnologia *Java Enterprise Edition* (J2EE) é transformada em um *servlet*.

Um exemplo de servidor compatível com a tecnologia JSP é o *Tomcat*. O *Tomcat* funciona como servidor de aplicações *Java* para *web*. É distribuído como *software* livre e desenvolvido como código aberto e é implementação de referência para as tecnologias *Java Servlet* e JSP (KURNIAWAN, 2004). Tecnicamente o *Tomcat* é um *container Web*, tendo a

capacidade de atuar também como servidor *web*/HTTP, ou pode funcionar integrado a um servidor *web* dedicado.

2.1.1.5 *Java Server Pages Tag Library*

Com o intuito de melhorar o código *Java* que vai ser escrito nas páginas JSP, foram desenvolvidas as *Java Server Pages Tag Library* (JSTL) (BAYERN, 2002). Ao encapsular a funcionalidade que as diversas páginas *web* precisam em *tags* simples, JSTL serve para padronizar o trabalho de programação em páginas JSP. JSTL também pode acessar banco de dados e escrever códigos SQL em páginas, embora essa prática não seja muito comum.

Ao utilizar JSTL podemos encontrar *tags* bastante similares aos comandos utilizados na programação *Java*, o que a torna bastante amigável ao usuário. Algumas das *tags*, familiares ao programador Java são:

- *c:catch* – bloco do tipo *try / catch*;
- *c:forEach* – laço de repetição do tipo *for*;
- *c:if* – expressão condicional *if*;
- *c:out* – saída;
- c:*import* – importa código de outras páginas JSP.

2.2 Tecnologias utilizadas

2.2.1 *Hibernate*

O *Hibernate* é uma solução *open-source* para Mapeamento Objeto / Relacional (ORM). ORM é uma técnica de mapeamento que consiste em mapear um modelo de Objetos para um modelo Relacional (usualmente representado por uma base de dados SQL) (BAUER, 2005). Foi criado em meados de 2001 por Gavin King e Christian Bauer, entre outros desenvolvedores (HIBERNATE, 2007). Desde então, o *Hibernate* vem se tornando um popular *framework* de persistência na comunidade *Java*.

O objetivo do *Hibernate* é diminuir a complexidade entre os programas *Java*, baseado no modelo orientado a objeto, que precisam trabalhar com um banco de dados do modelo

relacional (presente na maioria dos Sistemas Gerenciadores de Banco de Dados). Em especial, no desenvolvimento de consultas e atualizações dos dados (HEMRAJANI, 2006). A tecnologia relacional provê uma maneira de compartilhar dados entre aplicações diferentes ou tecnologias diferentes que formam uma mesma aplicação.

O *Hibernate* transforma os dados tabulares de um banco de dados em um grafo de objetos definido pelo desenvolvedor (BAUER, 2005). Com isto, o desenvolvedor se livra de escrever muito do código de acesso a banco de dados e de SQL, que ele escreveria não usando a ferramenta, acelerando a velocidade do seu desenvolvimento.

A ferramenta permite usar domínios de objetos (normalmente *Java Beans*) e de forma simples criar arquivos de mapeamento baseados em XML. Estes arquivos indicam quais campos (em um objeto) serão mapeados para suas respectivas colunas (na tabela) (GONÇALVES, 2006). O *Hibernate* possui uma linguagem de pesquisa chamada *Hibernate Query Language* (HQL). Esta linguagem permite escrever em comando SQL, mas também usar semânticas orientadas a objeto.

A HQL é um dialeto SQL para o *Hibernate*. Ela é uma linguagem de consulta que se parece muito com a SQL, mas a HQL é totalmente orientada a objeto, incluindo os paradigmas de herança, polimorfismo e encapsulamento.

No *Hibernate*, é possível escolher tanto usar a SQL quanto a HQL. Utilizar a HQL permite executar os pedidos SQL sobre as classes de persistência do *Java* ao invés de tabelas no banco de dados, separando assim, o desenvolvimento das regras de negócio do banco de dados. As classes de persistência não têm conhecimento nem dependência do mecanismo de persistência (HEMRAJANI, 2006).

Dessa forma, o *Hibernate* se torna uma solução não-intrusiva, pois não requer que se sigam muitas regras específicas e padrões ao escrever a lógica de negócios e as classes de persistência (MINTER, 2006). A integração com a maioria das aplicações existentes se dá de forma fácil e não requer mudanças no resto da aplicação.

O nível de transparência provido pelo *Hibernate* permite certo grau de portabilidade. Sem interfaces especiais, as classes persistentes não ficam presas a nenhuma solução em particular e a lógica de negócios é totalmente reutilizável em qualquer contexto da aplicação (BAUER, 2005).

No entanto, o *framework* não é uma boa opção para todos os tipos de aplicação. Sistemas que fazem uso extensivo de *stored procedures*[2], *triggers*[3] ou que implementam a maior parte da lógica da aplicação no banco de dados, contando com um modelo de objetos pobre (modelos que não possuem correspondência direta entre os objetos da aplicação e as tabelas do banco de dados) não vão se beneficiar com o uso do *Hibernate* (HIBERNATE, 2007). Ele é mais indicado para sistemas que contam com um modelo rico, onde a maior parte da lógica de negócios fica na própria aplicação *Java*, dependendo pouco de funções específicas do banco de dados.

2.2.2 *Enterprise JavaBeans*

Enterprise JavaBeans (EJB) (SRIGANESH, 2006) é um *framework* da plataforma J2EE, do tipo servidor, que é executado no *container*[4] para EJB do servidor de aplicação. Os principais objetivos da tecnologia EJB são fornecer rápido e simplificado desenvolvimento de aplicações *Java*, baseadas em componentes, distribuídas, transacionais, seguras e portáveis.

EJB é responsável por encapsular a lógica de negócios de uma aplicação. A idéia é que o desenvolvedor de EJBs se preocupe apenas com regras de negócio e não com serviços de *middleware* (segurança, transação, persistência, *cache*, *log*, entre outros) (MUNDOOO, 2007). Serviços de *middleware* são responsabilidade do servidor de aplicação. Esses serviços serão chamados pelo *container*.

Ao usar EJB é possível escrever aplicações escaláveis, confiáveis e seguras, sem ter que escrever um *framework* próprio e complexo. EJB permite fácil desenvolvimento de aplicações que executam no servidor (DEVELOPER, 2007). Pode-se rapidamente construir componentes para o servidor em *Java* ao reutilizar uma infra-estrutura distribuída já existente. EJB é desenvolvido para suportar reusabilidade e portabilidade de aplicações.

---

[2]Nota: *Stored procedures* são subrotinas utilizadas pelas aplicações para acessar um sistema de um banco de dados relacional.

[3]Nota: *Triggers* são códigos automaticamente executados em resposta a certos eventos em uma tabela em particular em um banco de dados.

[4]Nota: *Container* é um objeto que contém outros objetos. Estes objetos podem ser incluídos ou removidos dinamicamente, em tempo de execução.

Basicamente, EJB pode ser definido como um componente. Componente é uma entidade que contém a si própria, permitindo que ela seja reusada em uma aplicação similar ou em uma aplicação completamente diferente, enquanto as semânticas dos componentes estiverem em acordo (DEVX, 2007). Um componente deve ser empacotado com tudo que for necessário para que ele seja independente, reutilizável e exista fora de sua aplicação original. Uma aplicação de negócios ou um sistema pode ser feito para consistir com múltiplos e reutilizáveis componentes de *software*, cada um com certa funcionalidade.

O desenvolvedor de EJB deve descrever todas as regras de *middleware* (segurança, transação, persistência, entre outros) em arquivos descritores de implantação, que basicamente são arquivos XML que informam como um componente/método deve ser acessado. Quem escreve os descritores de implantação é o montador de aplicação ou *application assembler* (MARINESCU, 2002). Ele deve configurar os arquivos para o ambiente (servidor de aplicação, banco de dados, etc.) em que os EJBs serão implantados.

Fisicamente (SRIGANESH, 2006), EJB é na verdade, dois em um:

- Especificação: Após o lançamento da versão 3.0 a especificação ficou dividida em três documentos. A especificação deixa as regras de empenhamento entre os componentes e os servidores de aplicação. Garante que o comportamento do código da aplicação seja do tipo “escreva uma vez, execute em qualquer lugar”.
- Um conjunto de interfaces *Java*: Componentes e servidores de aplicação devem estar de acordo com essas interfaces. Desde que todos os componentes sejam escritos para essas interfaces, o servidor de aplicação poderá gerenciar qualquer desses componentes EJB.

Atualmente ele encontra-se na versão 3.0 e seu futuro é definido conjuntamente entre grandes empresas, como também por uma vasta comunidade de desenvolvedores numa rede mundial de colaboração.

A grande mudança entre a versão 2.1 e a versão 3.0 é a introdução de anotações *Java*. As anotações facilitam o desenvolvimento, diminuindo a quantidade de código e o uso de arquivos de configuração XML.

2.2.2.1 *Container* EJB

O conceito de *container* na arquitetura J2EE define um conjunto de componentes que atendem uma especificação. Existem diversos tipos de *container* como *Applet Container*, *Web Container*, EJB *Container*, os quais um componente para executar nestes ambientes tem que atender as especificações descritas nos mesmos (PANKAJ, 2003).

O *container* EJB é responsável por gerenciar os EJBs. A responsabilidade mais importante do EJB é prover um ambiente seguro, transacional e distribuído no qual os EJBs possam executar. No entanto, nem os *beans*[5] nem os clientes que chamam esses *beans* precisam gerar código voltado explicitamente às APIs do container EJB para utilizá-lo. Ao invés disso, o *container* toma conhecimento dessa necessidade através da especificação de informações de configuração em XML, dentro do *deployment descriptor* (descritor de implantação) ou dentro do código do *bean* usando *deployment annotations*[6] (KURNIAWAN, 2002). Em essência, um *container* EJB age de forma “invisível” entre o cliente e o *bean*, conforme a figura 2.1 mostra. O *container* provê serviços implicitamente ao *bean* (MUNDOOO, 2007). Alguns desses serviços disponíveis serão descritos a seguir.

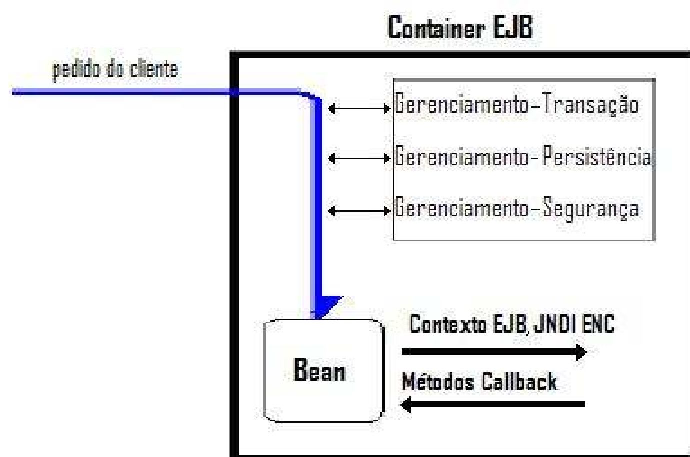

Figura 2.1 – *Container* EJB gerencia os *beans* em execução. Adaptado de MUNDOOO (2007).

---

[5]Nota: Nesse contexto, EJB e *beans* possuem o mesmo significado.

[6]Nota: *Annotation* é uma maneira de adicionar metadados ao código Java para que esteja disponível ao programador em tempo de execução. É usado como alternativa ao XML.

Gerenciamento de Transação: permite realizar operações determinísticas ao configurar atributos dos *beans*. O *container* EJB provê serviço de transação, uma implementação baixo-nível de gerenciamento de transação e coordenação (KODALI, 2006). O serviço é exposto através da *Java Transaction API* (JTA). JTA é uma interface alto-nível para controle de transações (SRIGANESH, 2006).

Segurança: é a principal consideração para implantações multi-camadas. A plataforma *Java Standard Edition* (Java SE) provê um ambiente seguro que autentica e autoriza o acesso ao código *Java* (SRIGANESH, 2006). EJB adiciona a isso a noção de segurança transparente, tal que os acessos aos métodos dos *beans* estão seguros ao configurar atributos de segurança ao invés de ser necessário gerar mais código para isso (PANKAJ, 2003).

Gerenciamento de Recurso e Ciclo de vida: o *container* EJB gerencia recursos, tais como *threads, sockets,* e conexões de banco de dados para os *beans.* De fato, o *container* gerencia o ciclo de vida dos *beans* (KODALI, 2006). O *container* realiza as tarefas:

- Cria e destrói as instâncias dos *beans*;
- Torna-os passivos ao serializá-los a um armazenamento secundário (quando necessário);
- Ativa-os lendo seu estado serializado, entre outras.

Além disso, o *container* tem a habilidade de reutilizar instâncias dos *beans* como e quando quiser.

Acessibilidade remota: clientes localizados em uma máquina virtual *Java* remota podem invocar métodos em um *bean*. O *container* torna isso possível sem serem necessárias mais linhas de código e converte os *beans* em objetos distribuídos, para servir aos clientes remotos (SRIGANESH, 2006).

Suporte a pedidos concorrentes: o *container* também serve pedidos concorrentes de clientes, provendo suporte ao gerenciamento de *threads.* Dessa forma, pode manter um *pool* de instâncias do *bean* (mantém múltiplas instâncias do *bean*) para servir concorrentemente aos pedidos dos clientes (SRIGANESH, 2006). Se múltiplos clientes chamam métodos de uma instância do *bean*, o *container* pode também serializar os pedidos, permitindo que somente um cliente acesse a instância do *bean* por vez (DEVX, 2007). Quando isso acontece, outros clientes são roteados para diferentes instâncias do *bean* ou são forçados a esperar até a instância original estar disponível.

Clustering e balanceamento de carga: Apesar de não fazer parte da especificação, a maioria dos *containers* EJB vem equipado com suporte a *clustering* e balanceamento de carga

(SRIGANESH, 2006). Obviamente essa é uma vantagem valiosa para qualquer implantação que precisa trabalhar com um grande número de requisições de forma segura contra falhas e escalável (KODALI, 2006). Ao mesmo tempo, porque esses são essencialmente serviços fora do padrão, sua configuração varia de *container* para *container*.

Um *bean* depende do *container* para qualquer serviço que necessite. Para isto ele interage com o *container* através de três mecanismos:

- Métodos de *CallBack*: Todo EJB implementa uma interface a qual define vários métodos, chamados métodos de *callback* (MUNDOOO, 2007). Cada método *callback* alerta o *bean* de um diferente evento no seu ciclo de vida e o *container* invocará estes métodos para notificar o *bean* quando o mesmo deva ser criado, removido, persistido na base de dados, etc.;
- *EJBContext Interface*: Todo EJB obtém um objeto de contexto, o qual é uma referência direta para o *container*. A interface provê métodos para interagir com o *container*, tanto que o *bean* pode requisitar a informação sobre o seu ambiente como a identificação do cliente ou o status de uma transação ou pode obter referências remotas para ele mesmo (KODALI, 2006);
- JNDI (*Java Naming and Directory Interface*): Todo Bean automaticamente tem acesso a um sistema especial chamado *Environment Naming Context* (ENC). O ENC é gerenciado pelo *container* e acessa os *beans* usando JNDI. O JNDI ENC permite que um *bean* acesse recursos como conexões JDBC, outros *beans*, e propriedades específicas para aquele *bean* (MUNDOOO, 2007). Isto é feito internamente, porém a programação é a mesma que de um cliente comum.

2.2.2.2 Tipos de EJB

Um *container* EJB mantém três tipos de *beans*:

- *Session Beans* (Beans de Sessão) – que pode ser *stateful* ou *stateless*;
- *Message Driven Beans (MDBs) – Beans* orientados a mensagens;
- *Entity Beans.*

2.2.2.2.1 *Session Beans*

Um *bean* de sessão *stateless* é uma coleção de serviços cada qual sendo representado por um método específico (DEVELOPER, 2007). Quando uma aplicação cliente, que pode ser até outro EJB, invoca um método em um *bean* de sessão *stateless* o método é executado e retorna o resultado sem nenhum compromisso de se manter o estado do Objeto entre as chamadas para este mesmo EJB que está no *pool* (DEVX, 2007). *Stateless* costumam ser de propósitos gerais ou reusáveis como um serviço de *software*.

Um bean de sessão *stateful* é uma extensão da aplicação cliente. Quando um cliente obtém uma conexão com o EJB *stateful* o seu estado é mantido entre as chamadas do mesmo cliente até o cliente remover esta conexão (DEVELOPER, 2007). Isto significa que o estado dos atributos de classe do *stateful* permanece inalterado enquanto o cliente mantém a conexão. Esta característica do *stateful* se chama estado conversacional porque representa uma conversação contínua entre o mesmo e o cliente (DEVX, 2007). Eles representam uma lógica que pode ser capturada uma aplicação cliente entre chamadas e temos como exemplo uma cesta de pedidos de livros em um site como o *Amazon* (DEVELOPER, 2007).

2.2.2.2.2 *Message Driven Beans*

Antes da versão 2.0 da especificação EJB a maioria dos consumidores de mensagens JMS (*Java Message Service*) foram construídos como simples programas *Java* fora do *container* J2EE (DEVELOPER, 2007). Existia um programa *Java* que era ativado e conectado a uma destinação JMS que ficava esperando as mensagens. Embora a construção de consumidores de mensagens por meio de aplicações *Java* era uma das melhores soluções para o problema esta solução gerava vários outros problemas e o maior era a escalabilidade (MUNDOOO, 2007). Quando o número de mensagens na fila aumentava consideravelmente, haveria a necessidade de se executar um programa *Java multi-thread* para promover a escalabilidade da leitura destas mensagens e se este volume variasse muito, ficaria difícil fazer um algoritmo que na medida exata consumisse as mensagens sem perda de performance com muitas *threads* e sem perda destas mensagens no caso de poucas *threads* (DEVX, 2007).

Basicamente, *beans* orientados a mensagens processam alguma lógica de negócios usando mensagens JMS enviadas para uma destinação particular, ou seja, consomem mensagens JMS através da tecnologia EJB (MUNDOOO, 2007).

A grande diferença entre *beans* orientados a mensagens e *beans* de sessão ou de entidade é que os *beans* orientados a mensagens são completamente escondidos do cliente. O único meio de clientes comunicarem com os *beans* orientados a mensagens é enviando uma mensagem para um destinatário JMS (DEVX, 2007).

2.2.2.2.3 *Entity Beans*

São objetos de dados, isto é, objetos *Java* que guardam informações do banco de dados (DEVELOPER, 2007). A figura 2.2 ilustra os vários tipos de clientes em uma aplicação EJB em conjunto com os diferentes tipos de EJBs.

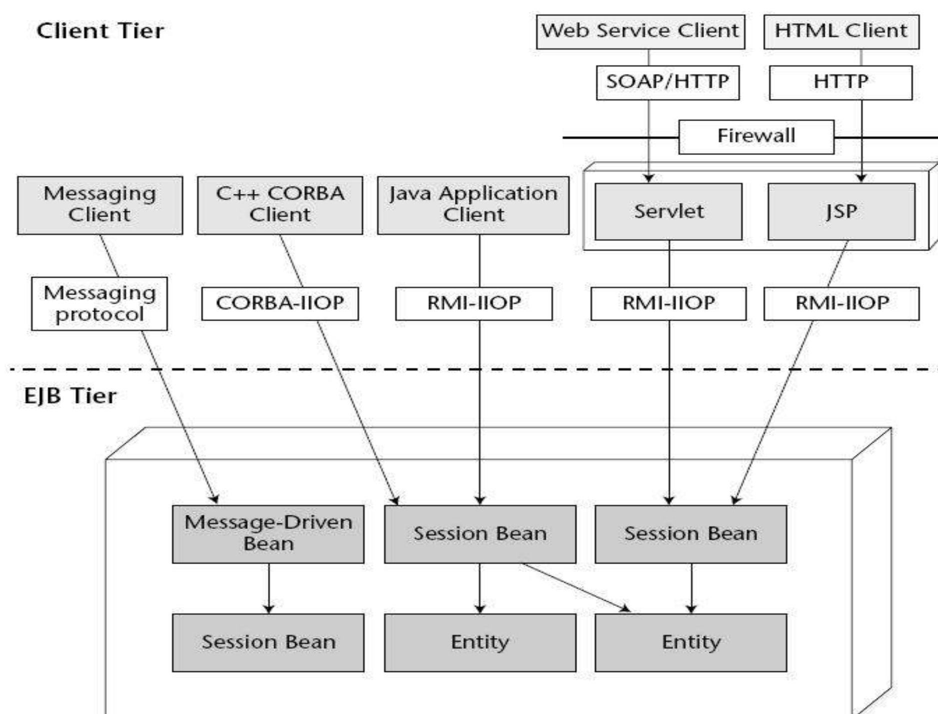

Figura 2.2 – Subsistema EJB: Os vários clientes e *beans* (SRIGANESH, 2006).

2.2.2.3 Visão geral de um componente EJB

Com relação ao projeto geral, a melhor forma de entendê-lo é verificar sua implementação. Para implementar um *bean, é* necessário definir duas interfaces e uma ou duas classes:

- *Remote Interface*:

A interface remota de um EJB define os métodos de negócio, ou seja, os métodos presentes para o mundo externo que irá utilizar o *bean* (DEVELOPER, 2007);

- *Home Interface*:

A interface *home* define o ciclo de vida do *bean* com métodos de criação, remoção ou localização de *beans* (MUNDOOO, 2007);

- *Bean Class*:

A classe é responsável por implementar os métodos de negócios do *bean* (DEVELOPER, 2007);

- *Primary Key*:

A chave primária é uma classe muito simples que provê um ponteiro para a base de dados. Somente *Entity Beans* necessitam de uma chave primária (DEVX, 2007).

2.2.3 *Spring*

Spring é um *framework open- source* criado por Rod Johnson e descrito em seu livro *Expert One-on-One: J2EE Design and Development* (JOHNSON, 2005). Foi criado para endereçar a complexidade do desenvolvimento de aplicações empresariais. Spring torna possível utilizar *Java Beans* para conseguir desenvolver aplicações que não eram possíveis com EJBs (WALLS, 2005). No entanto, a utilidade do *Spring* não se limita ao desenvolvimento do lado do servidor. Qualquer aplicação *Java* pode se beneficiar do *Spring* em termos de simplicidade, testabilidade e baixo acoplamento (HEMRAJANI, 2006).

Também se pode definir *Spring* como um *framework* “leve” para inversão de controle e um *container* orientado a aspectos (WALLS, 2005), ou seja:

Leve - em termos de tamanho e de sobrecarga. Todo o *framework* pode ser distribuído em um simples arquivo JAR com tamanho médio de um *megabyte*. A sobrecarga de processamento requerido pelo *Spring* é mínima (TATE, 2005). Além disso, *Spring* é não-intrusivo: objetos em uma aplicação com *Spring* tipicamente não possuem dependência com

as classes específicas do *Spring* (WALLS, 2005);

Inversão de controle – *Spring* promove baixo acoplamento através da técnica conhecida como Inversão de Controle (IoC) (WALLS, 2005). Quando IoC é aplicado, passivamente suas dependências são passadas aos objetos, ao invés de criar ou procurar por objetos dependentes por si próprios, isto é, um objeto não procura por dependências em um *container*, o *container* dá as dependências para o objeto durante a instanciação sem esperar por um pedido (JOHNSON, 2005);

Orientado a aspectos - *Spring* vem com suporte para programação orientada a aspectos que permite desenvolvimento coeso ao separar a lógica de negócios da aplicação dos serviços do sistema, como por exemplo, o gerenciamento de transações (HEMRAJANI, 2006). Os objetos da aplicação fazem o que devem fazer, de acordo com a lógica de negócios e nada mais. Eles não são responsáveis por (ou tem conhecimento de) outros interesses do sistema, tais como *login* ou suporte transacional (TATE, 2005);

Container – no sentido de conter e gerenciar o ciclo de vida e configuração de objetos da aplicação (TATE, 2005). É possível configurar como cada um dos *beans* deve ser criado – uma instância ou uma nova instância para cada vez que for necessário – baseado em um protótipo configurável e como eles devem se associar. *Spring*, no entanto, não deve ser confundido com os *containers* EJB (SAM-BODDEN, 2006);

Framework – torna possível configurar e compor aplicações complexas com componentes simples (TATE, 2005). Em *Spring*, objetos da aplicação são compostos declarativamente, tipicamente em um arquivo XML. *Spring* também provê funcionalidades para a infra-estrutura (gerenciamento de transação, integração do *framework* de persistência, etc.), deixando a lógica da aplicação a cargo do desenvolvedor (WALLS, 2005).

Todos esses atributos permitem que seja escrito código limpo, mais gerenciável e fácil de testar. Eles também ajustam uma variedade de sub-*frameworks* dentro do *Spring* (WALLS, 2005).

Com *Spring* os *beans* dependem dos colaboradores através das interfaces (HEMRAJANI, 2006). Desde que não haja dependências específicas de implementação, aplicações *Spring* são bastante desacopladas, testáveis e fáceis de manter (SAM-BODDEN, 2006). E porque o *container Spring* é responsável por resolver as dependências, o serviço ativo de procura, que é envolvido no EJB está agora fora de uso e o custo de programar interfaces é minimizado (WALLS, 2005). Tudo o que é preciso é criar classes que se comunicam através de interfaces e o *Spring* cuida do resto.

2.2.3.1 Módulos do *Spring*

O *framework Spring* é feito de sete módulos bem definidos, descritos a seguir. Como um todo, esses módulos provêem o necessário para desenvolver aplicações empresariais. Todos os módulos são feitos sobre o *container* núcleo. O *container* define como os *beans* são criados, configurados e gerenciados (JOHNSON, 2005).

Container – provê a funcionalidade fundamental do *framework*. Nesse módulo se encontra a *BeanFactory*, o coração de qualquer aplicação baseada em Spring (WALLS, 2005). A *BeanFactory* é uma implementação da *factory* padrão que usa IoC para separar a configuração da sua aplicação e especificações de dependência do verdadeiro código da aplicação (HEMRAJANI, 2006);

Contexto da Aplicação – a *BeanFactory* torna o *Spring* um *container*, mas o módulo de contexto é o que torna *Spring* um *framework* (JOHNSON, 2005). Esse módulo estende o conceito da *BeanFactory*, adicionando algumas funcionalidades . Esse módulo também supre muitos serviços, tais como *e-mail*, integração com EJB, entre outros (WALLS, 2005).

Programação Orientada a Aspectos (AOP) – *Spring* provê suporte para programação orientada a aspectos em seu módulo AOP. Esse módulo serve como base para desenvolver seus próprios aspectos para uma aplicação com *Spring* (WALLS, 2005);

Abstração *Java Database Connectivity*[7] (JDBC) e *Data Access Object*[8] (DAO) – trabalhar com JDBC freqüentemente resulta em código padrão e repetitivo para estabelecer uma conexão, criar uma declaração, processar um resultado e então fechar a conexão. O módulo abstrai todo esse código para manter o código do banco de dados simples e claro, e prevenir problemas que resultam de uma falha em fechar recursos do banco de dados (JOHNSON, 2005). Além disso, este módulo constrói uma camada de exceções no topo das mensagens de erro dados pelos servidores dos bancos de dados, tornando mais simples sua compreensão (HEMRAJANI, 2006);

---

[7]Nota: JDBC é um conjunto de classes e interfaces Java que envia instruções SQL para bancos de dados relacionais

[8]Nota: DAO é um padrão para persistência de dados que permite separar regras de negócio das regras de acesso a banco de dados.

Integração com mapeamento Objeto/Relacional (ORM) – *Spring* provê esse módulo para os desenvolvedores que preferem usar ferramentas para mapeamento objeto/relacional ao invés de JDBC. Essa não é uma solução própria para ORM, mas provê integração com os *frameworks* populares para ORM, dentre eles o *Hibernate* (HEMRAJANI, 2006);

Spring Web – o módulo de contexto *web* trabalha sobre o módulo de contexto da aplicação, provendo um contexto apropriado para aplicações *web*. Além disso, esse módulo suporta várias tarefas orientadas a *web* (TATE, 2005);

Framework MVC – *Spring* possui um *framework* MVC completo para desenvolver aplicações *web*. Apesar de *Spring* poder se integrar facilmente com outros *frameworks* MVC, como o *Struts*, o *framework* MVC usa IoC para prover uma separação clara da lógica de controle dos objetos de negócios (WALLS, 2005). Também permite ligar declarativamente parâmetros de requisição para seus objetos de negócios (JOHNSON, 2005).

2.2.3.2 XML

XML (*eXtensible Markup Language*) é uma recomendação da W3C (consórcio de empresas de tecnologia responsável por desenvolver padrões de criação e interpretação de conteúdos para a Web) para gerar linguagens de marcação para necessidades especiais (W3ORG, 2007).

A XML foi desenvolvida para permitir a criação de marcações de documentos, com o intuito de facilitar o intercâmbio de dados (HEITLINGER, 2001). Ela possibilita uma maior flexibilização na descrição dos conteúdos e a sua grande contribuição é permitir a definição de marcadores sem preocupação com a apresentação (DEITEL, 2003).

É capaz de descrever diversos tipos de dados. Seu propósito principal é a facilidade de compartilhamento de informações através da *Internet* (WIKIPEDIA, 2008). Entre linguagens baseadas em XML incluem-se XHTML (formato para páginas *web*), MathML (formato para expressões matemáticas), entre outros (INFOWESTER, 2007).

Devido à insatisfação com os formatos existentes até o momento, o *World Wide Web Consortium* (W3C) trabalhou no desenvolvimento de uma linguagem de marcação que combinasse a flexibilidade da SGML com a simplicidade da HTML (W3ORG, 2007). O objetivo do projeto era criar uma linguagem que pudesse ser lida por *software*, e integrar-se

com as demais linguagens existentes. Sua filosofia (WIKIPEDIA, 2008) tem como princípios importantes:

- Separar o conteúdo da formatação;
- Ser simples e legível para humanos e para computadores;
- Permitir a criação ilimitada de *tags*;
- Criação de arquivos para validação de estrutura (DTDs);
- Interligação de bancos de dados distintos;
- Concentração na estrutura da informação, e não na sua aparência.

O XML é um bom formato para criar documentos com dados organizados de forma hierárquica, como em documentos de texto formatados ou bancos de dados (HEITLINGER, 2001). Pela sua portabilidade, um banco de dados pode através de uma aplicação, escrever em um arquivo XML, enquanto outro banco distinto pode ler estes mesmos dados (DEITEL, 2003).

Dentre algumas das vantagens de XML, é interessante citar:

- XML não é uma linguagem de programação, o que faz com que não seja necessário ser um programador para poder aprendê-la (W3ORG, 2007). XML é um conjunto de regras para projetar formatos de texto que permitem estruturar dados;
- Como XML tem influência do HTML, XML usa marcadores e atributos. No entanto, HTML especifica o que cada um deles significa enquanto XML apenas delimita os trechos de dados, deixando a interpretação à cargo da aplicação que os lê (DEITEL, 2003).

A figura 2.3 apresenta um exemplo que demonstra a flexibilidade do XML, sendo usada para descrever um *Curriculum Vitae*:

```
<?xml version="1.0" encoding="UTF-8"?>
<curriculo>
   <InformacaoPessoal>
      <DataNascimento>23-07-68</DataNascimento>
      <Nomecompleto>...</Nomecompleto>
      <Contatos>
         <Morada>
              <Rua>R.Topazio</Rua>
              <Num>111</Num>
              <Cidade>nome_cidade</Cidade>
              <Pais>nome_pais</Pais>
         </Morada>
         <Telefone>9999-9999</Telefone>
         <CorreioEletronico>email@email.com</CorreioEletronico>
      </Contatos>
      <Nacionalidade>brasileiro</Nacionalidade>
      <Sexo>M</Sexo>
   </InformacaoPessoal>
   <objetivo>Atuar na area de TI</objetivo>
   <Experiencia>
      <Cargo>Suporte tecnico</Cargo>
      <Empregador>Empresa, Cidade - Estado</Empregador>
   </Experiencia>
   <Formacao>Superior Completo</Formacao>
      </curriculo>
```

Figura 2.3 - Código em XML.

2.2.3.2.1 XML e o mapeamento para Sistemas de Bancos de Dados

A *Web* tem se tornado um canal importante de troca de informações entre os indivíduos ou organizações, assim sendo observam-se cada vez mais estudos nas áreas voltadas para a *Internet* (W3ORG, 2007). Tendo isso em vista, O W3C propôs a XML como alternativa para permitir o intercâmbio de documentos pela *Web*.

Verifica-se também a preocupação de grupos de pesquisa na área de banco de dados em relação à integridade, armazenamento, representação dos dados, restrições e vários outros aspectos relacionados à XML (INFOWESTER, 2007). Na área de banco de dados existe o que se pode chamar de gerações de sistemas de bancos de dados, sendo que a geração dos sistemas gerenciadores de bancos de dados relacionais (SGBDRs) é a que predomina comercialmente ainda hoje (W3ORG, 2007).

No entanto, o desenvolvimento dos sistemas gerenciadores de bancos de dados objeto-relacionais (SGBDOR) está crescendo em setores em que os SGBDRs não atendem de forma adequada (INFOWESTER, 2007). Isso abrange sistemas onde são necessários novos tipos de dados, tais como dados complexos que aparecem em projetos de engenharia, projetos de arquitetura, entre outros (W3ORG, 2007).

Diversos SGBDs possuem extensões para transferência entre documentos XML e suas próprias estruturas de dados. Eles são usados por aplicações centradas nos dados, exceto quando o banco de dados suporta o armazenamento nativo em XML (HEITLINGER, 2001). No caso em que o banco de dados não suporta o armazenamento de dados nativo, deve-se mapear as estruturas XML para o esquema do banco de dados (DEITEL, 2003).

Para um efetivo armazenamento de documentos XML, é necessário conhecer o mapeamento do documento nos modelos relacional, objeto-relacional e orientado a objetos, tendo em vista as diversas possibilidades, impondo-se desta forma, restrições de um ambiente para outro (DEITEL, 2003). Uma aplicação prática é transferir os documentos XML para o banco de dados, levando-se em consideração o mapeamento proposto sem a intervenção do usuário (HEITLINGER, 2001).

Os documentos XML podem ou não possuir um esquema associado. Esses esquemas associados definem as normas da estrutura do documento e existem várias representações destes esquemas, tais como o XML *Schema*[9] e o DTD (*Document Type Definition*). Um documento XML é considerado válido se ele está em conformidade com o DTD a ele associado (INFOWESTER, 2007).

Existem três focos fundamentais de pesquisa sobre XML: consulta, armazenamento e publicação. Assumindo que se conhece a definição do tipo de documento XML, deve-se definir o armazenamento. Os modelos orientados a objetos (incluindo o modelo objeto-relacional) são mais próximos ao XML e podem facilmente lidar com coleções aninhadas e sua ordem (listas) (HEITLINGER, 2001). Para o caso dos sistemas de bancos de dados relacionais, as coleções precisam ser simuladas aumentando-se o número de tabelas e/ou o número de atributos (DEITEL, 2003).

A transferência de documentos XML para o banco de dados envolve o mapeamento do esquema do documento XML para o esquema do banco de dados (DEITEL, 2003). O mapeamento pode ser feito através de um conjunto de regras que, aplicadas à estrutura do documento, fornecem uma tradução do documento para classes e/ou tabelas definindo desta forma uma linguagem de mapeamento (INFOWESTER, 2007).

---

[9] Nota: XML *Schema* é uma linguagem baseada no formato XML para definição de regras de validação (esquemas) em documentos no formato XML.

Diversos sistemas gerenciadores de bancos de dados já fornecem suporte ao XML, tais como o Oracle e o DB2. Deve ser mencionado que alguns bancos de dados objeto-relacionais possuem um repositório de objetos XML, como o DB2 (INFOWESTER, 2007).

3 O SISTEMA DE GERENCIAMENTO DE INFORMAÇÕES

Neste capítulo, será apresentado o Sistema de Gerenciamento de Informações da CentralPrev (SGICP) que será responsável por gerenciar as atividades realizadas pelas unidades de atendimento e na matriz da CentralPrev.

Para apresentar o SGICP, inicialmente serão expostas as necessidades da CentralPrev que serão tratadas por este sistema. Posteriormente, serão apresentadas as soluções adotadas para viabilizar o desenvolvimento e atender as necessidades da CentralPrev.

3.1 As necessidades da CentralPrev

Nesta seção serão abordadas as dificuldades encontradas pela CentralPrev em realizar suas atividades diárias. Para trazer maior qualidade e mais agilidade à realização de suas tarefas, todos os processos realizados em suas unidades de atendimentos foram analisados. Durante essa análise foram encontradas as seguintes possíveis melhorias:

3.1.1 Distribuição geográfica das unidades de atendimento

Como toda empresa que procura expandir sua área de atuação, a CentralPrev criou várias unidades de atendimento em mais de 20 cidades do estado do Rio Grande do Sul e uma no Distrito Federal. Desta forma, a CentralPrev pode realizar suas atividades de assessoria em direito previdenciário e empresarial para clientes que estão localizados nas cidades e regiões onde possui unidades de atendimento.

Embora a distribuição das unidades de atendimento permita uma maior área de atuação, isto acaba por dificultar o controle de suas atividades. Um escritório de advocacia gera muita documentação durante a realização de suas atividades, dificultando a troca de informações entre os escritórios. Durante a realização de todo um processo a comunicação entre escritórios é freqüente, enviando e recebendo informações sobre o andamento do processo e documentações relevantes.

Muito freqüentemente um processo muda de alçada. Por exemplo: Um cliente procura a unidade de Santa Maria e em um determinado momento, o processo será julgado em Porto Alegre. Para acompanhar o andamento do processo, eventualmente algumas tarefas são delegadas a um advogado da matriz. Essa troca de informações seria grandemente facilitada por um sistema de gerenciamento.

Ter escritórios distribuídos geograficamente gera dificuldades no acompanhamento da realização das tarefas, bem como a troca de informações. Para facilitar o controle das atividades realizadas nos escritórios e garantir coerência das informações, uma base unificada de dados se faz necessária. Para tal, é preciso desenvolver um sistema que possa alimentar essa base de dados e gerar informações a partir dela. Esta base de dados conterá informações muito importantes que devem ter seu acesso restrito, para tanto, um serviço local (presente nas unidades de atendimento) garantirá que as informações somente serão acessadas dentro das instalações.

Ainda citando o exemplo anterior, durante o andamento em Porto Alegre, a unidade de Santa Maria necessita receber informações sobre o andamento para prosseguir com suas atividades e para informar o cliente.

3.1.2 Controle administrativo e estatístico

Como qualquer empresa, a CentralPrev precisa manter um controle administrativo de suas unidades de atendimento. Dentro dessas atividades está o controle de funcionários e colaboradores, controle de atividades exercidas, agendas dos advogados, controle de arquivo, entre outros. Estas tarefas, embora exercidas em unidades diferentes, precisam ser acessadas pela matriz para que possam ser gerenciadas de uma forma centralizada.

A análise estatística das atividades também é uma necessidade. Dados sobre andamento de processos e casos procedentes e improcedentes são informações muito importantes para a compreensão do andamento das tarefas.

3.1.3 Controle financeiro

O setor financeiro da empresa que fica localizado em Santa Maria é responsável por controlar as finanças de todas as unidades. Para poder fazer a contabilidade de todas as

unidades, o setor financeiro necessita reunir as informações sobre as receitas e despesas de cada unidade. Por não possuir subsetor financeiro em cada uma das unidades, a tarefa de reunir as informações necessárias é de responsabilidade do setor financeiro.

A possibilidade de registrar as informações através de um sistema remoto permitiria maior agilidade na realização da contabilidade das unidades individualmente e da empresa como um todo. Ainda será necessário reunir a documentação necessária para comprovar as despesas e as receitas para concluir a contabilidade da empresa.

3.2 Modelagem e Implementação

A modelagem do sistema será apresentada ao longo das próximas sessões. Conforme as camadas do sistema forem detalhadas, será exibida a sua modelagem, visando melhor compreensão das idéias.

O desenvolvimento deste projeto, visou à implementação de todo o sistema web. No entanto, ao término deste trabalho de conclusão, foram adicionadas, de acordo com as necessidades encontradas, as seguintes funcionalidades ao sistema:

- Infra-estrutura – O núcleo de todo o sistema *web*, sendo responsável por suportar e garantir o funcionamento do sistema. É a infra-estrutura que garante a comunicação entre as camadas do sistema (que serão descritas na próxima seção). Todos os módulos desenvolvidos funcionam sobre esta parte, utilizando seus métodos para desempenharem suas atividades. Novos módulos podem ser construídos ao definir novas regras de negócio que garantam seu funcionamento;
- Módulo de cadastro de clientes – Cada um dos clientes da CentralPrev precisa ser cadastrado junto ao sistema, para que seja possível manter um registro de clientes. Além disso, é imprescindível garantir que suas informações sejam consistentes e que seja possível acessar estes dados rapidamente. Este módulo permite que os escritórios criem novos cadastros de clientes, bem como possam consultar cadastros existentes, independente de qual escritório criou este registro;
- Módulo de cadastro de funcionários – Desenvolvido a partir de razões semelhantes às do cadastro de clientes. É importante manter uma lista de todos os funcionários que trabalham para a empresa, bem como informar de qual escritório eles fazem parte. Estes dados devem estar disponíveis para todos os escritórios;

- Módulo de processos – Freqüentemente, processos mudam de jurisdição. Por isso, as informações sobre estes processos precisam ser passadas para o escritório que ficará encarregado. Neste módulo, cada unidade da empresa deve registrar os processos pelos quais é responsável. Junto desse registro, estarão anexados arquivos essenciais para complementar as informações. Dessa forma, todos os processos ficam disponíveis para todos os escritórios, facilitando a troca desses dados;
- Módulo de atividades – Este módulo serve para agendar todas as atividades que devem ser realizadas pelas unidades diariamente. É necessário que estas atividades estejam devidamente registradas para controlar e garantir que os prazos sejam cumpridos.

3.3 Camadas do Sistema

Para solucionar as necessidades da CentralPrev, o sistema foi divido em três camadas lógicas. A camada de apresentação contém as páginas e sua lógica de controle, enquanto a camada de serviço possui todas as lógicas de negócio da aplicação. A camada de dados contém as configurações e validações de acesso a dados.

A figura 3.1 exemplifica a divisão lógica do sistema.

Figura 3.1 - Divisão lógica do sistema.

A divisão física do sistema foi feita em duas camadas. A primeira camada contém um servidor de páginas que provê a interface de usuários e a lógica de controle da camada de aplicação. A segunda camada é um servidor de aplicação que possui a lógica de negócios acessível através de serviços e a validação de dados e controle de acesso ao banco de dados.

A figura 3.2 exemplifica a divisão física do sistema.

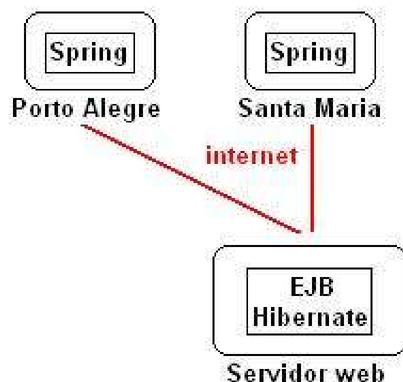

Figura 3.2 - Divisão física.

Em cada uma das unidades de atendimento deve ser instalado um servidor de páginas junto com um *container servlet*. Este servidor disponibilizará para toda a unidade páginas que servirão como camada de apresentação. Este servidor estará acessível apenas na *intranet* de cada unidade de atendimento. Com isto, o acesso a camada de apresentação externo as dependências da unidade estará indisponível.

Para disponibilizar os serviços *web*, deve ser contratado um servidor de páginas com suporte ao servidor de aplicação. Este servidor de aplicação conterá os serviços EJB que contem a lógica de negócios do sistema.

Ainda no servidor de páginas contratado, deve ser instalado o *framework Hibernate* que será responsável por configurar o acesso ao banco de dados e validar os dados.

Conforme mencionado anteriormente, a divisão lógica do sistema foi feita em três partes. A seguir, serão mais bem detalhadas essas partes.

3.4 Camada de Apresentação (cliente)

Ao utilizar um sistema *web* como solução, ao invés de utilizar aplicações *desktop*, o sistema terá como central um servidor. Portanto, é necessário que os escritórios tenham acesso a este servidor de alguma forma.

Para servir de interface com o usuário foi decidido que a forma mais simples e fácil de disponibilizar esta interface seria através de páginas *web*. Assim, cada escritório deve acessar essas páginas para utilizar o sistema. O conjunto destas páginas forma a camada de apresentação do SGICP. A figura 3.3 mostra como as redes locais possuem acesso ao sistema. Cada LAN representa um dos escritórios da CentralPrev.

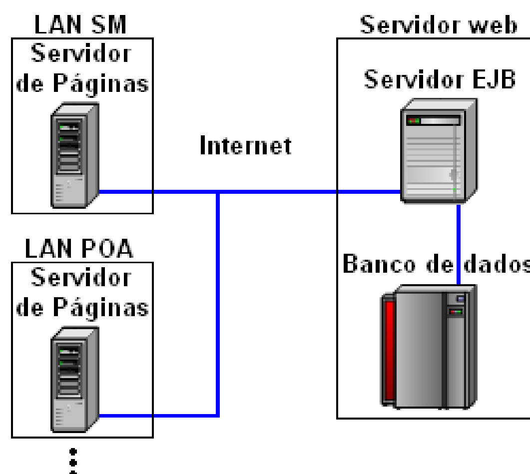

Figura 3.3 – Acesso ao sistema através de páginas *web*.

Durante o desenvolvimento da camada de apresentação, foram levadas em consideração duas importantes questões: segurança e desempenho do sistema. Para tal, optou-se por instalar servidores de páginas *Tomcat*, que possuem *container* para *servlets*.

Os computadores dos escritórios estão conectados em rede, através de uma LAN. A camada de apresentação provê segurança ao garantir que os acessos ao sistema poderão ser somente de dentro desta rede. Se as páginas fossem disponibilizadas através de um servidor de páginas, seria possível que essas fossem acessadas por qualquer computador com acesso a *Internet*, ao passo que restringir os acessos somente à LAN garante que seu conteúdo não será acessado por outros, como está representado pela figura 3.4.

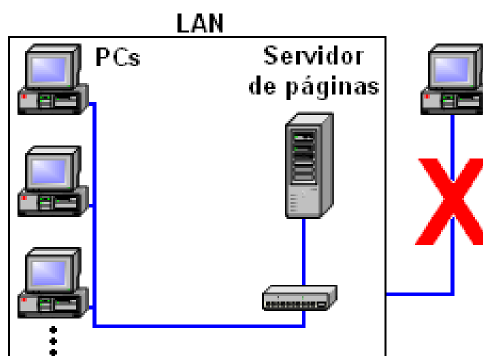

Figura 3.4 – Acesso restrito a rede local.

Utilizar páginas *web* também traz benefícios para o desempenho do sistema. Isto se dá pelo fato de que navegar por páginas, ao invés de utilizar uma aplicação local, requer menos poder de processamento dos computadores do escritório. Além disso, um servidor de páginas demanda menos processamento do que um servidor de aplicações para desempenhar sua função.

No SGICP, é preciso que apenas um computador tenha maior poder de processamento, sendo este o servidor local. Em um sistema tradicional, é necessário que todos os computadores que utilizam a aplicação sejam poderosos para garantir que o sistema desenvolva sua função com o desempenho necessário. Além disso, essa escolha diminui o custo com equipamento, já que os computadores não precisam ser avançados para navegar em páginas.

As possíveis atualizações no servidor não serão passadas para todas as máquinas do sistema, mas sim apenas ao servidor local. Isso será discutido no capítulo 4.

As páginas *web* do servidor foram desenvolvidas utilizando a tecnologia *JavaServer Pages*. Por ser baseada na linguagem *Java*, JSP possui portabilidade de plataforma (FIELDS, 2000), isto significa que sua execução é independente de sistema operacional, podendo ser executado em diversos sistemas, como *Windows* ou *Linux*, por exemplo. Com JSP é possível desenvolver aplicações para a internet que acessem bancos de dados e também manipulem informações a partir da interface de usuário.

Para trazer maior desempenho e melhoria na qualidade da geração de código fonte, foi empregado o modelo de desenvolvimento *Model-view-controller*. MVC permite separar os dados (que ficam localizados na *model*) da interface (*view*). Assim, qualquer alteração na interface não afetará a forma como os dados são manipulados, e da mesma forma, é possível

reorganizar os dados sem que seja necessário alterar a interface do usuário. Para que isso aconteça, existe um componente chamado *controller*, que separa o acesso aos dados da lógica de negócio da apresentação, bem como a interação com o usuário (WIKIPEDIA, 2007).

Para essa implementação, foi utilizado o *framework Spring*. O *container* do *Spring* permitirá o controle de toda a camada de apresentação, através da utilização de seus módulos.

A camada de apresentação é subdividida em três partes, sendo estas: parte visual, parte de controle e modelos de dados.

3.4.1 Parte Visual (*View*)

A parte visual representa a *view* no modelo MVC, sendo responsável pelo exterior da aplicação. Aqui é realizada a montagem das telas que serão exibidas. Nesta parte estão definidas as regras que declaram como devem ser detalhados os componentes da página, como por exemplo, cabeçalhos, menus e títulos. A figura 3.5 representa como as regras agem sobre as páginas. Na figura, o quadro que envolve as regras XML está destacado para representar que somente as regras necessárias para aquela página serão carregadas.

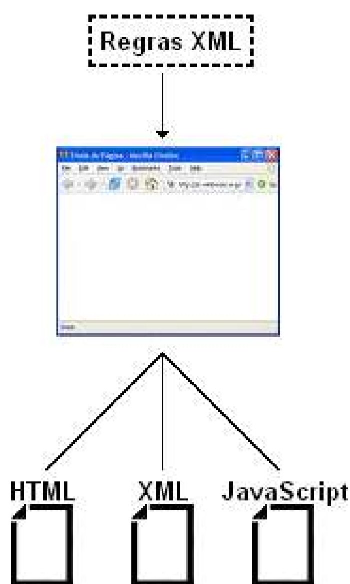

Figura 3.5 – Regras XML no processo de montagem de telas.

Uma importante tarefa desta parte da camada de apresentação é construir máscaras de validação de dados, responsável por controlar a entrada de dados. Um exemplo é a mascara de validação de telefone, que aceita apenas números e valida a quantidade mínima de caracteres. Outro exemplo de validação de dados pode ser visto na figura 3.6. Ao pesquisar por uma pessoa no sistema, esta pode ser feita através do código da pessoa, ou por seu nome. O campo código possui um número limitado de caracteres, sendo que estes devem ser apenas números.

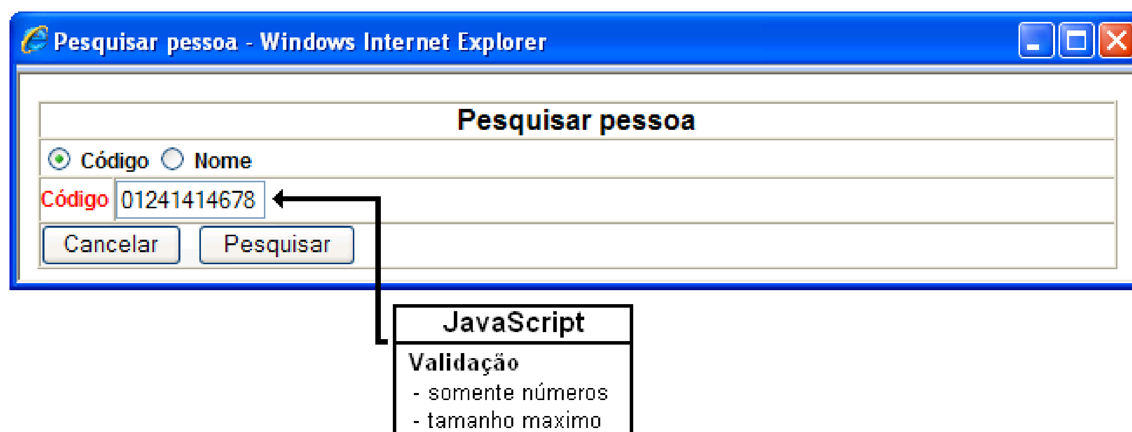

Figura 3.6 – Validação de dados.

Outra informação visual que é passada para o usuário e fica a cargo da *view* é o destaque dos campos obrigatórios. Esses campos devem ser preenchidos obrigatoriamente em uma determinada tela para que os dados dos objetos possam ser informados corretamente. Exemplos de campos obrigatórios podem ser vistos na figura 3.7.

Cadastro de usuário	
Nome	Douglas Monteiro
Tipo	Funcionário
Login	douglas
Senha	••••••••
Permissão	Administrador

XML
Campos obrigatórios
- login
- senha
- permissão

Figura 3.7 – Campos obrigatórios.

Embora a parte visual seja responsável por máscaras e campos obrigatórios, ela não verifica as informações passadas nesses campos. Os dados serão revalidados na parte de controle, que será apresentada mais adiante. A parte visual apenas define como os campos obrigatórios serão visualizados na tela e como serão padronizadas as máscaras que porventura aparecerem nas demais telas da aplicação.

Para o preenchimento de informações que já são pré-definidas, como sexo e cidade, por exemplo, serão utilizadas páginas XML. Essas páginas contêm os dados que a aplicação precisa no momento que as telas forem exibidas. Dessa forma, a aplicação não terá que se conectar ao servidor, neste momento, para buscar essas informações. Após a aplicação verificar as páginas XML, os dados irão popular as *Combo Box* presentes nas páginas *web*, como a figura 3.8 exemplifica.

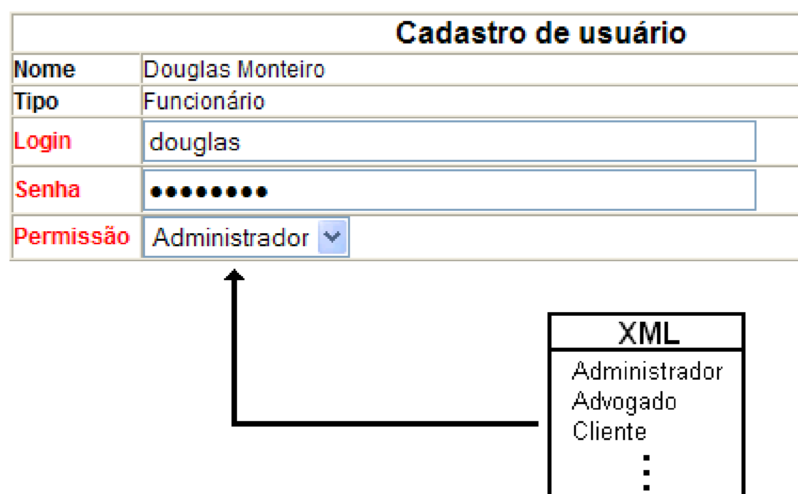

Figura 3.8 - XML no carregamento de uma *Combo Box*

3.4.2 Parte de Controle (*Controller*)

Esta parte da camada de apresentação representa o *controller* no modelo de desenvolvimento MVC. O *controller* é encarregado de validar as informações fornecidas à parte de apresentação, como os dados de entrada, campos obrigatórios e máscaras.

Além disso, a parte de controle também é responsável por comandar as ações que acontecem na camada de apresentação. No *controller* são definidas as regras de validação e o carregamento das páginas XML.

Outra característica da parte de controle é a utilização de inversão de controle. IoC torna a implementação da lógica de controle das páginas mais fácil. A lógica de restrição de acesso a páginas é um exemplo.

Dentre as funções do *controller*, pode-se destacar: controle de ações do usuário, primeira validação, execução de aspectos, acesso a base local e regras de validação.

3.4.2.1 Controle de ações do usuário

Assim que o usuário acessa a aplicação, o *controller* procura por uma conta de usuário no contexto do *container* do servidor de páginas. Logo na primeira vez, o contexto não possui nenhuma conta de usuário, portanto é necessário que o servidor de páginas procure a conta de usuário junto ao serviço *web.*

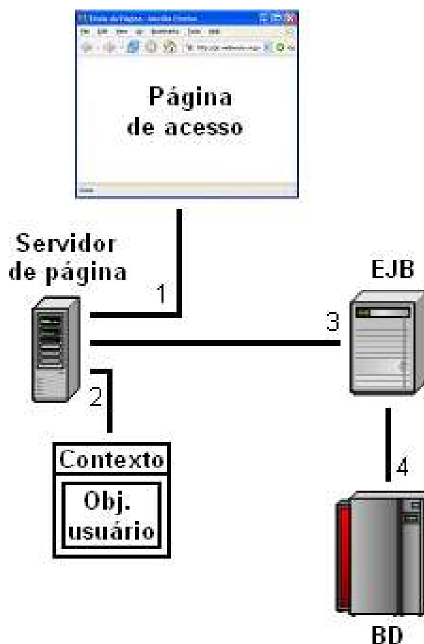

Figura 3.9 - Etapas da certificação de usuário.

Quando o usuário deseja efetuar *login*, ele informa seu nome de usuário e senha. Essas informações são enviadas ao serviço *web* que será responsável pela validação dos dados (passo 1 na figura 3.9). A próxima etapa é a consulta ao contexto. É preciso verificar a existência de uma conta que corresponda às informações fornecidas pelo usuário (passo 2 na figura 3.9). Caso essa conta não esteja no contexto, o serviço precisa construir o objeto do usuário, que será utilizado pela aplicação. O objeto é criado para que a camada de aplicação possa trabalhar com as informações referentes ao usuário, bem como controlar as ações do mesmo. Portanto, o servidor de páginas precisa fazer uma chamada ao serviço (passo 3 na figura 3.9). O serviço busca as informações do usuário no banco de dados (passo 4 na figura 3.9) e retornar um objeto de usuário. Na próxima vez que esse mesmo usuário efetuar *login* no sistema, seu objeto de usuário estará no contexto.

Após a confirmação do *login* do usuário, seus privilégios são determinados a partir das informações retornadas pelo objeto correspondente a ele. Cada usuário possui um tipo diferenciado de conta, que contém níveis de permissão diferentes. Além das permissões, o objeto do usuário também define as restrições do usuário, ou seja, o seu espaço de ação. A figura 3.10 representa um objeto de usuário.

Obj usuário

- nome: Douglas Monteiro
- código: 0001
- acesso:
  - permissão: cadastro_usuario
  - restrição: total
⋮

Figura 3.10 - Objeto de usuário.

Ao delimitar as permissões do usuário, pode-se garantir que as informações serão visualizadas apenas por pessoal autorizado. Essas permissões determinam quais páginas o usuário pode acessar.

As restrições determinam se o usuário pode manipular um grupo de informações ou não. Ao usuário pode ser permitida a visualização de uma página, mas não necessariamente a alteração. É de extrema importância que as informações permaneçam corretas e que somente pessoal autorizado possa alterar essas informações.

3.4.2.2 Primeira Validação

Para sustentar o bom funcionamento do sistema, é necessário que algumas das informações sejam obrigatórias, como por exemplo, o nome de um cliente ou o seu número de identidade. Portanto, é necessário garantir que esses dados recebidos estejam corretos.

No entanto, apenas se certificar do envio de dados pode não ser suficiente. Também é preciso garantir que as informações se encaixem nos critérios estabelecidos para os determinados tipos necessários. Por exemplo, ao enviar um número de telefone deve se certificar que ele contenha apenas números.

Também é importante enfatizar que verificar se os dados estão corretos antes de enviá-los ao serviço, evita que informações erradas sejam passadas para a próxima camada. Se o servidor fosse o único responsável pela validação dos dados, o tráfego de dados seria maior, pois cada vez que as informações passadas estivessem erradas, seria necessário que estas informações fossem reenviadas. Se o erro for detectado ainda na camada de aplicação, o custo para reenviar dados é significativamente menor.

3.4.2.3 Execução de aspectos

Durante o desenvolvimento do sistema, pôde-se notar que existem algumas regras que se aplicam a vários casos. Utilizando programação orientada a aspectos (AOP) é possível elaborar aspectos que executam serviços do sistema que são comuns a todas as regras de controle da camada de apresentação. Cada uma dessas regras pode ser aplicada como um aspecto.

Os aspectos ficam localizados no *container* e são controlados automaticamente por ele. Dessa forma é necessário somente que os aspectos sejam programados e o *container* fica responsável pela execução dessas regras. Por exemplo, o tratamento de exceções pode ser considerado um aspecto. A figura 3.11 ilustra uma situação onde ocorre o tratamento de exceções.

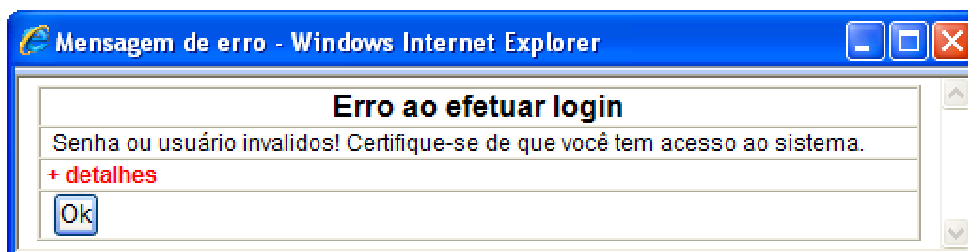

Figura 3.11 – Exemplo de exceção.

Em caso de falha ao tentar efetuar *login*, é necessário que exista uma regra que determine como o sistema deve se portar nesta situação. Quando isto acontecer, cabe ao *container* detectar a exceção através do aspecto e chamar o método necessário para tratar a exceção. O *container* tem conhecimento de onde se localizam todos os métodos que devem ser observados pelos aspectos, bem como todos os aspectos que devem ser utilizados. A figura 3.12 representa o momento em que uma exceção é detectada.

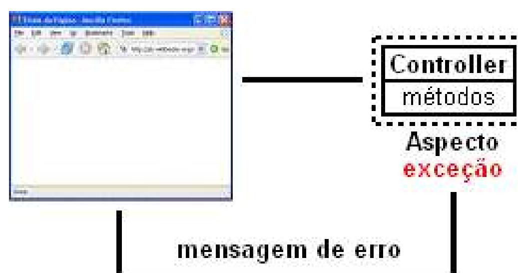

Figura 3.12 – Execução de um aspecto.

3.4.2.4 Acesso a base local

Em algumas telas da camada de apresentação é necessário que sejam apresentadas listas de dados, como listas de municípios, estados civis, entre outros. Essas listas se fazem necessárias para que o usuário possa ter opções antes de tomar alguma decisão.

Para evitar que a aplicação tenha que fazer uma conexão ao banco de dados somente para preencher essas listas, foi utilizada uma base de dados local. Essa base de dados é acessada pelo *controller* para que este, através de IoC, possa passar as informações das listas para as telas da *view*.

A base de dados foi construída utilizando XML, o que permite que ela seja acessada de forma rápida. Esta base contém apenas dados para consulta, o que reforça o fato de não ser necessário que essas informações sejam armazenadas em uma base de dados remota.

3.4.2.5 Regras de validação

Para que a camada de apresentação possa manipular o objeto do usuário corretamente, o *controller* precisa validá-lo primeiro. O *controller* encontra o objeto que deve ser validado, carrega um arquivo XML contendo as regras referentes aquele objeto (por exemplo, quais ações devem ser executadas para um determinado tipo de conta) e busca o conjunto de métodos para fazer a validação daquele objeto. Após isso, a parte de controle executa a validação.

Cada modelo representa uma tela da aplicação, sendo assim, para que as regras sejam definidas para cada modelo, existe um arquivo contendo as regras de validação daquele modelo. Assim, pode-se dizer que as regras estão fortemente ligadas ao modelo de dados de tela, já que cada modelo possui seu próprio arquivo de regras, como está representado pela figura 3.13.

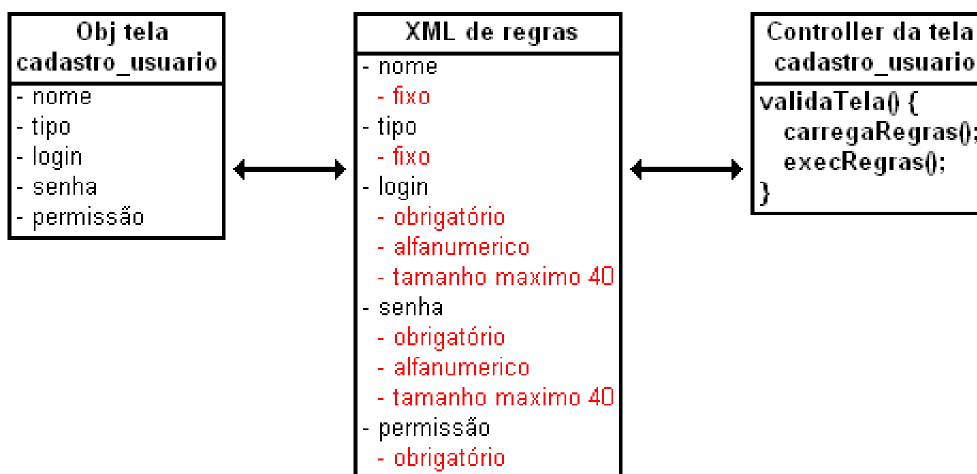

Figura 3.13 – Exemplo de regras de validação.

Separar as regras dos modelos é vantajoso, pois diminui o tamanho dos dados que irão trafegar pela rede. Além disso, essas regras só se aplicam a camada de apresentação, portanto não há a necessidade de passá-las para as demais camadas do sistema.

3.4.3 Modelos de Dados (*Model*)

Para que a camada de apresentação possa utilizar as informações que se encontram no banco de dados, estas informações precisam ser transformadas pela camada de serviços. Assim, o serviço cria objetos com estes dados, para que a camada de apresentação possa utilizá-los.

Os objetos são modelados para as páginas da camada de apresentação. Estes objetos não refletem colunas de tabelas, mas sim as informações necessárias para a geração das páginas. Cada objeto é modelado de forma que melhor corresponda com a página a ser exibida.

Quando for necessário que as informações sejam armazenadas, é preciso que os objetos de tela sejam enviados ao serviço. O serviço é responsável por transformar os objetos de tela em objetos do *Hibernate*, para que possam ser persistidos na base de dados. Isso se torna possível porque a camada de serviços possui o mesmo modelo de dados da camada de apresentação consigo. Dessa forma é mais fácil controlar as regras dos objetos, como por exemplo, se os campos estão preenchidos.

Os objetos do modelo de dados funcionam como *beans.* Por isso, são menores e contêm somente informações relevantes para as demais camadas do sistema. Conforme mencionado anteriormente, por serem de menor tamanho trafegam facilmente pela *Internet*.

3.5 Camada de serviços

A camada de serviços se situa entre a camada de aplicação e a camada de dados, sendo responsável por fazer a comunicação entre elas. Através do *framework* EJB, foram desenvolvidos serviços que permitem que a camada de serviços controle a lógica de negócios. Além das regras de negócios, foram aplicadas autenticações e autorizações de acesso de

código, para garantir segurança transparente ao acesso de métodos dos serviços. A figura 3.14 mostra como este processo está estruturado.

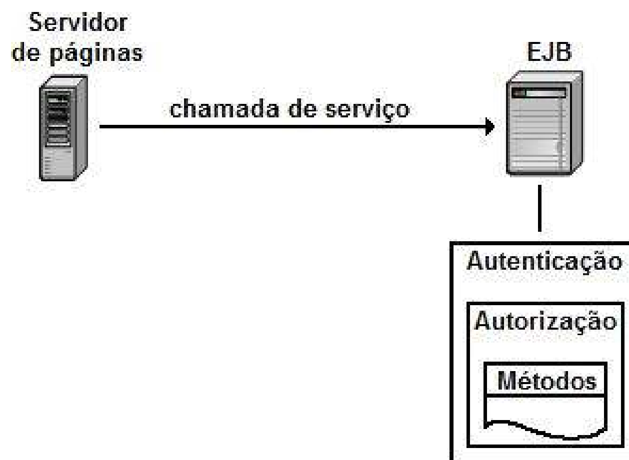

Figura 3.14 - Exemplo de autenticação e autorização.

Para que o usuário possa visualizar as informações referentes ao sistema, a camada de aplicação necessita dos objetos de tela. São eles que contêm todos os dados referentes à tela que está sendo exibida. Como essas informações se encontram na camada de dados, cabe a camada de serviços buscar essas informações nas tabelas do banco de dados e montar um objeto de tela que seja coerente com o que se deseja naquele momento.

Quando o *controller* identifica a necessidade da aplicação de utilizar um objeto de tela, se for necessário fazer uma requisição ao banco de dados, ele envia ao EJB um filtro contendo as chaves de busca. Os filtros servem para que o banco de dados possa localizar as informações que serão utilizadas para montar um objeto de tela, como a figura 3.15 exemplifica.

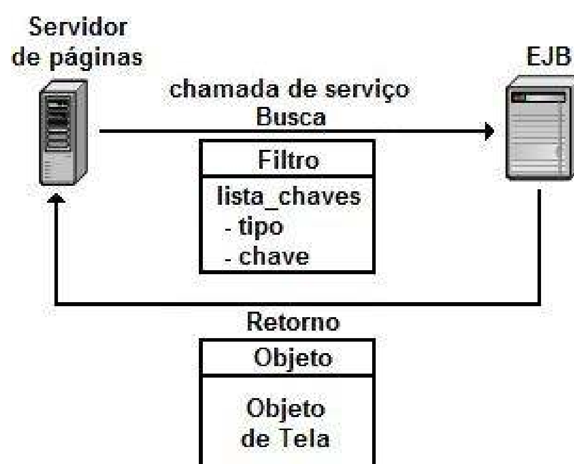

Figura 3.15 – Exemplo de filtros de busca.

Após a camada de serviços receber os dados, ela é responsável por montar o objeto de tela. Assim que o objeto estiver pronto, a camada de serviços utiliza os validadores para verificar se o objeto possui todos os atributos e se estes são válidos. A figura 3.16 mostra como ocorre este processo.

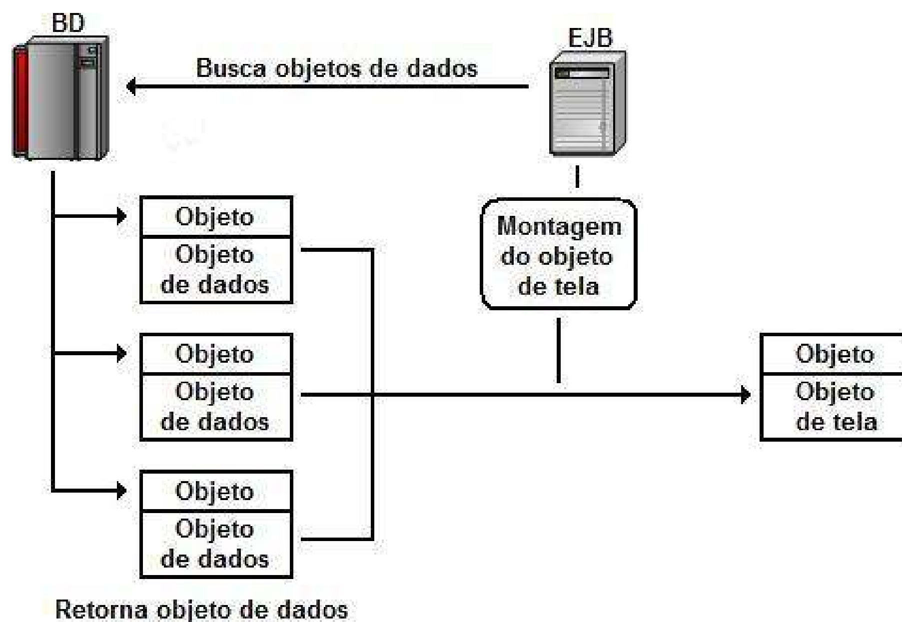

Figura 3.16 - Montagem do objeto de tela.

Outra funcionalidade da camada de serviços é repassar as informações vindas da camada de apresentação para o banco de dados. Para que isto possa ocorrer, a camada de serviços possui uma cópia do modelo de dados da camada de apresentação. Assim, os objetos

de tela podem ser validados pela camada de serviços antes que alguma alteração no banco de dados seja feita. Uma vez que todas as informações do objeto estejam presentes e sejam válidas, a alteração no banco de dados pode ocorrer. A figura 3.17 demonstra o processo de validação e inserção das informações no banco de dados.

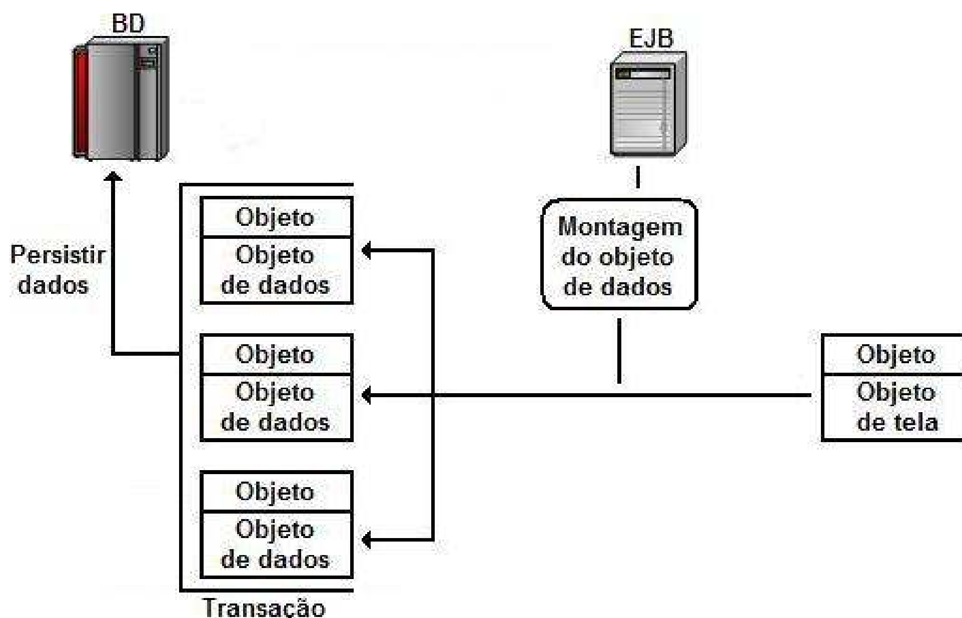

Figura 3.17 - Montagem dos dados para inserção.

3.6 Camada de dados (Banco de dados)

A camada de dados é responsável pela persistência das informações no banco de dados do sistema. Todos os tipos de dados que trafegam pelo sistema e que precisam ser reutilizados devem ser armazenados no banco de dados, para que o sistema possa reutilizá-los sempre que necessário. De outra forma, seria necessário que todas as informações fossem passadas ao sistema sempre que uma nova ação fosse executada por este.

O *framework Hibernate* é o responsável pelo mapeamento objeto relacional, isto é, ele mapeia o banco de dados para um modelo de objeto. Este mapeamento diminui a complexidade para acessar o banco de dados e transformar informações tabulares em objetos.

Ao fazer uma consulta ao banco de dados, eventualmente existem muitas referências a outras tabelas, o que pode não ser necessário no momento. Para tal, o *Hibernate* possui uma

estratégia chamada *lazy fetching* (BAUER, 2005), que consiste em inicializar os objetos ou coleções somente quando eles forem necessários. Isto diminui o tráfego de dados, melhorando o desempenho do sistema. A figura 3.18 mostra um exemplo onde *lazy fetching* foi utilizado. O objeto cadastro_usuario faz uma referência ao objeto pessoa. No entanto, cadastro_usuario não utiliza todas as informações do objeto pessoa. Somente as informações necessárias serão referenciadas

Figura 3.18 - Exemplo de caso utilizando *lazy fetching*.

Outra importante tarefa que será realizada pelo *Hibernate* será a validação de dados no momento da persistência. Esta tarefa será realizada com base em anotações (do *framework* do *Hibernate*) feitas nos modelos de dados. Dessa forma a validação do objeto que deverá ser persistido fica a cargo do Hibernate retirando esta atribuição da camada de negócio.

Anteriormente, o mapeamento com *Hibernate* era feito a partir de um conjunto de configurações em arquivos XML. Para que o *Hibernate* soubesse como carregar e armazenar objetos de classes persistentes, eram usados arquivos de mapeamento XML. Dessa forma, eram informadas quais tabelas do banco de dados se referiam à quais classes persistentes e quais colunas na tabela são referentes à quais atributos da classe (FERNANDES, 2008). Com o surgimento das anotações, a partir do *Java* SE 5.0, é possível que classes *Java* sejam mapeadas mais facilmente.

As anotações podem ser definidas como metadados que aparecem no código fonte e marcam partes de objetos de forma que agreguem algum significado especial (FERNANDES, 2008). A partir das anotações, não é mais necessário que as configurações sejam definidas em

arquivos XML, através do uso de um conjunto de anotações no código fonte das classes mapeadas.

Anotações permitem que seja informado ao *Hibernate* qual tabela no banco de dados representa determinada classe *Java*, bem como quais atributos correspondem as colunas dessa tabela. A figura 3.19 mostra um exemplo de como utilizar anotações junto ao código *Java*.

```
package br.com.centralprev.hibernate.anotacoes.dominio;

import org.hibernate.annotations.*;
import javax.persistence.*;

//Anotação que informa que a classe mapeada é persistente
@Entity
//Informando nome e esquema da tabela mapeada
@Table(name="usuario", schema="anotacoes")
public class Usuario {
//Definição da chave primária
@Id
//Definição do mecanismo de definição da chave primária
@GeneratedValue(strategy = GenerationType.SEQUENCE)
//Informa o nome da coluna mapeada para o atributo
@Column(name="id_usuario")
private long idUsuario;
@ManyToOne(fetch = FetchType.EAGER)
@JoinColumn(name="id_pessoa", insertable=true, updatable=true)
@Fetch(FetchMode.JOIN)
@Cascade(CascadeType.SAVE_UPDATE)
private Pessoa pessoa;
private String tipo;
private String login;
private String senha;
private int permissao;
public void Usuario(){}
//Métodos getters e setters
//...
}
```

Figura 3.19 - Exemplo de mapeamento com anotações.

Para garantir que o sistema funcione corretamente, é necessário que as anotações sejam validadas. Isto significa que se deve verificar se as anotações estão de acordo com a representação das tabelas no banco de dados.

Hibernate provê uma forma de validação, o *Hibernate Validator*. O *Validator* permite expressar restrições de domínio de forma única, garantindo que as regras definidas serão cumpridas na camada de dados do sistema (HIBERNATE, 2007).

Assim como anotações são formas bastante convenientes e elegantes de especificar restrições no modelo de domínio da implementação, o *Hibernate Validator* vem com um conjunto de validações comuns, tais como *@NotNull*, *@Max*, entre outros. Além disso, é possível criar regras de validação próprias. A figura 3.20 mostra um trecho do código apresentado pela figura 3.19, utilizando *Hibernate Validator*. No exemplo é possível perceber a existência de campos obrigatórios e alguns destes são limitados por tamanho.

```
private Pessoa pessoa;
@NotNull(message="Tipo de usuário é obrigatório")
private String tipo;
@NotNull(message="Login de usuário é obrigatório")
@Length(min=3, max=20, message="Login de usuário deve ter entre 3 e 20 caracteres")
private String login;
@NotNull(message="Senha de usuário é obrigatório")
@Length(min=7, max=20, message="Senha de usuário deve ter entre 3 e 20 caracteres")
private String senha;
@NotNull(message="Permissão de usuario é obrigatório")
private int permissao;
public void Usuario(){}
//Métodos getters e setters
//...
}
```

Figura 3.20 – Código *Java* com *Hibernate Validator*.

4 CONCLUSÃO

Este trabalho apresentou uma implementação de um sistema *web* como proposta de solução para o problema com gerenciamento de informações apresentado pela CentralPrev. Com o auxílio de diferentes tecnologias e ferramentas foi desenvolvido um sistema capaz de controlar as informações da empresa.

Até o momento da conclusão deste trabalho, foi desenvolvida a infra-estrutura e os seguintes módulos do sistema: módulos de cadastro de funcionários e clientes, módulo de processos e o módulo de atividades.

A infra-estrutura pode ser definida como um conjunto de elementos que provê suporte ao sistema. Aqui foram desenvolvidas as funcionalidades básicas do sistema, como métodos de comunicação entre camadas, chamadas básicas, entre outros.

Os módulos de cadastro são responsáveis por gerenciar as listas de funcionários e clientes da empresa. Esses módulos mantêm uma base de dados consistente, ao mesmo tempo em que permitem criar novas informações de forma rápida e fácil. Já os módulos de processos e atividades servem para controlar o andamento da empresa. Além de gerenciar, estes módulos possuem funcionalidades específicas para garantir coerência nas informações e facilitar o processo de troca de informações entre os escritórios.

Atualmente, o sistema encontra-se em fase de ampliação. Alguns módulos ainda precisam ser desenvolvidos, o que adicionará ainda mais funcionalidades ao sistema. Além disso, será instalado o servidor de páginas no escritório de Porto Alegre, para fins de teste. A partir dos resultados desses testes, será possível identificar se há a necessidade de alterações no sistema.

Para trabalhos futuros, ficam destacadas algumas funcionalidades do sistema que não foram projetadas, sendo elas: atualização de servidores e segurança de acesso. A cada vez que houver atualizações no sistema, é preciso que estas atualizações sejam repassadas aos escritórios. Até então não foi planejado como isto deve se acontecer. Já na parte de segurança, deve-se estudar uma forma de permitir que ocorram acessos remotos ao sistema, sendo que por enquanto só é permitido acesso de dentro das redes locais.

REFERÊNCIAS BIBLIOGRÁFICAS

BAUER C.; KING G. **Hibernate in Action**. Editora Manning Greenwich. 2005.

BAYERN S. **JSTL in Action.** Manning Publications Co. 2002.

BEAL A. **Manual de Tecnologia da Informação**. Disponível em: <http://www.2beal.org/adriana/>. Acessado em 29 de agosto 2007.

DEITEL. **XML Como Programar.** Editora Bookman. 2003.

DEVELOPER. **Introduction to EJBs.** Disponível em: <http://www.developer.com/java/article.php/1434371>. Acessado em 27 de Novembro de 2007.

DEVX. **An Introduction to Java Object Persistence with EJB.** Disponível em: <http://www.devx.com/Java/Article/22441/0/page/1>. Acessado em 27 de Novembro de 2007.

FERNANDES R.; LIMA G. **Hibernate com Anotações.** Disponível em: <ftp://raphaela:web@users.dca.ufrn.br/UnP2007/Hibernate_Anotacoes.pdf> Acessado em 14 de janeiro de 2008.

FIELDS D.; KOLB M. **Desenvolvendo na Web com Java Server Pages.** Editora Ciência Moderna. 2000.

GONÇALVES E. **Desenvolvendo Aplicações Web com JSP, Servlets, JSF, Hibernate, EJB 3, Persistence e AJAX.** Primeira Edição. Editora Ciência Moderna. 2006.

HEITLINGER P. **O Guia Prático da XML.** Editora Centro Atlântico. 2001.

HEMRAJANI A. **Desenvolvimento Ágil em Java com Spring, Hibernate e Eclipse.** Primeira Edição. Editora Pearson. 2006.

HIBERNATE. **Página oficial do Hibernate.** Disponível em: <http://www.hibernate.org/>. Acessado em 29 de agosto de 2007.

INFOWESTER. **Linguagem XML.** Disponível em: <http://www.infowester.com/lingxml.php>. Acessado em 18 de novembro de 2007.

JOHNSON R.et al. **Professional Java Development with the Spring Framework.** Editora Wrox Press. 2005.

KODALI R. R.; WETHERBEE J. R.; ZADROZNY P. **Beginning EJB 3 Application Development: From Novice to Professional.** Primeira Edição. Editora Apress. 2006.

KURNIAWAN B.; DECK P. **Como Tomcat Funciona.** Primeira Edição. Editora Ciência Moderna. 2004.

KURNIAWAN B. **Java for the Web with Servlets, JSP, and EJB: A Developer's Guide to J2EE Solutions.** Primeira Edição. Editora Pearson. 2002.

MARINESCU F. **EJB Design Patterns: Advanced Patterns, Processes, and Idioms.** Primeira Edição. Editora John Wiley & Sons. 2002.

MINTER D.; LINWOOD J. **Beginning Hibernate: From Novice to Professional.** Terceira Edição. Editora Apress. 2006.

MOORE D.; BUDD R.; BENSON E. **Professional Rich Internet Applications: AJAX and Beyond (Programmer to Programmer).** Editora Wrox. 2007.

MUNDOOO. **Fundamentos de Enterprise Java Beans.** Disponível em: <http://www.mundooo.com.br/php/mooartigos.php?pa=showpage&pid=6>. Acessado em 27 de Novembro de 2007.

PANKAJ K. **J2EE Security for Servlets, EJBs and Web Services.** Primeira Edição. Editora Pearson. 2003.

SAM-BODDEN B. **Desenvolvendo em POJOs com Hibernate, JBoss, Spring e Tapestry.** Editora Alta Books. 2006.

SRIGANESH R.P.; BROSE G.; SILVERMAN M. **Mastering Enterprise JavaBeans 3.0.** Editora Wiley. 2006.

SUHRING S. **MySQL A Bíblia.** Editora Campus. 2002.

TATE B. A.; GEHTLAND J. **Spring: A Developer's Notebook.** Primeira Edição. Editora O'Reilly. 2005.

W3ORG. **XML in 10 points.** Disponível em: <http://www.w3.org/XML/1999/XML-in-10-points.html>. Acessado em 18 de novembro de 2007.

WALLS C.; BREIDENBACH R. **Spring in Action**. Manning Publications Co. 2005.

WIKIPEDIA. **Model-View-Controller - Artigo Wikipedia.** Disponível em: <http://pt.wikipedia.org/wiki/MVC>. Acessado em 9 de novembro de 2007.

WIKIPEDIA. **XML - Artigo Wikipedia.** Disponível em: <http://pt.wikipedia.org/wiki/XML>. Acessado em 5 de janeiro de 2008.