FACULDADE DE BALSAS
CURSO DE SISTEMAS DE INFORMAÇÃO

DESENVOLVIMENTO DE UM SISTEMA WEB PARA ENVIAR, ARMAZENAR E RECEBER DOCUMENTOS OFICIAIS

Fernando Ferreira Matos

BALSAS – MA
2013

FACULDADE DE BALSAS
CURSO DE SISTEMAS DE INFORMAÇÃO

Fernando Ferreira Matos

DESENVOLVIMENTO DE UM SISTEMA WEB PARA ENVIAR, ARMAZENAR E RECEBER DOCUMENTOS OFICIAIS

Trabalho de Conclusão de Curso apresentado como exigência parcial para obtenção do título de Bacharel em Sistemas de Informação à Faculdade de Balsas, sob a orientação do Professor Marcos David Souza Ramos.

Balsas - MA
2013

FACULDADE DE BALSAS

CURSO DE SISTEMAS DE INFORMAÇÃO

A Comissão Examinadora, abaixo assinada, aprova o Trabalho de Conclusão de Curso (TCC)

DESENVOLVIMENTO DE UM SISTEMA WEB PARA ENVIAR, ARMAZENAR E RECEBER DOCUMENTOS OFICIAIS

Elaborada por

Fernando Ferreira Matos

como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação

BANCA EXAMINADORA

_____

Prof. Marcos David Sousa Ramos

Prof. Orientador

_____

Prof. Msc. Jefferson Fontinele Da silva

Membro da Banca Examinadora

_____

Prof. Msc. Cleverton Marlon Possani

Membro da Banca Examinadora

AGRADECIMENTOS

Agradeço primeiramente a Deus, a meu orientador Marcos David Souza Ramos neste trabalho, a minha Família, a meus colegas e amigos.

RESUMO

Este trabalho visa mostrar o desenvolvimento do sistema web, para informatizar os processos de envio, armazenamento e recebimento de documentos oficiais utilizados pelos diversos setores da prefeitura municipal de Balsas-MA. Baseado em padrões Web Model-View-Controller (MVC) e alguns dos frameworks mais importantes para o desenvolvimento Web em Java como, Java Server Faces (JSF), Hibernate e Spring Security. O trabalho ainda expõe conceitos úteis sobre configurações de arquivos, anotações e CDI, necessários para o entendimento da estrutura e do desenvolvimento do mesmo. O trabalho tem como resultado final o desenvolvimento do sistema web, com a finalidade de automatizar os processos de envio de documentos oficiais utilizado na estrutura interna da prefeitura municipal de Balsas-MA.

Palavras-chaves: Desenvolvimento Web. Padrões. Frameworks.

ABSTRACT

This work aims to show the development of web system to computerize the process of sending, receiving and storage of official documents used by various sectors of the city hall Balsas-MA. Web-Based Model-View-Controller (MVC) and some of the most important frameworks for web development in java as Java Server Faces (JSF), Hibernate and Spring Security standards. the work also presents useful concepts about settings files, notes and CDI required for understanding the structure and development of the same. The work is the final result of the web system development, in order to automate the process of sending official documents used in the internal structure of the municipal government of Balsas-MA.

Keywords: Web Development. Standards. Frameworks.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
CDI	*Context and Depen- dency Injection*
DAO	*Data Access Object*
EJB	*Enterprise JavaBeans*
GUI	Componentes de Interfaces de Usuário
HTML	*Hypertext Markup Language*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*HyperText Transfer Protocol Secure*
IDE	*Integrated Development Enviroment*
J2EE	*Java 2 Platform Enterprise Edition*
JDBC	*Java Database Connectivity*
JDK	*Java Development Kit*
JSF	*Java Server Faces*
JSP	*Java Server Pages*
JSTL	*JavaServer Pages Standard Tag Library*
LDAP	*Lightweight Directory Access Protocol*
MVC	*Model-View-Controller*
OO	Orientada a Objetos
ORM	Mapeamento Objeto Relacional
PDF	*Portable Document Format* (Formato de Documento Portátil).
SQL	*Structured Query Language* (Linguagem de Consulta Estruturada)
URL	*Uniform Resource Locators* (Localizador Uniforme de Recursos)
XHTML	*Extensible Hypertext Markup Language*
XML	*Extensible Markup Language* (Linguagem de Marcação Estendida)

SUMÁRIO

1 INTRODUÇÃO

Sabe-se que a comunicação é responsável pelo sucesso ou fracasso em uma organização em relação à troca de informação, fazendo a passar de um ponto para outro, utilizando processos de emissão, transmissão e recepção de mensagens por meio de métodos ou sistemas convencionais (FERREIRA, 2005).

A Prefeitura Municipal de Balsas-MA vem passando por problemas relacionados à comunicação na transmissão de documentos oficiais, causado pela demora em que determinados documentos saiam do seu destino de origem e cheguem até seu destino final. A mesma é dividida fisicamente por setores e seus respectivos subsetores, todos localizados em locais diferentes, necessitando manter uma constante comunicação entre os mesmos, que é feita por troca de documentos oficiais. A prefeitura de Balsas-MA não possui um sistema de software que atenda às suas necessidades referentes à comunicação interna, na mesma existe uma grande quantidade de documentos oficiais armazenados, como ofícios, licitações e outros, sem contar que, novos documentos são gerados diariamente.

A **Figura1** é uma representação de como funcionar a comunicação interna da mesma, com foco no envio e recebimento de documentos oficiais.

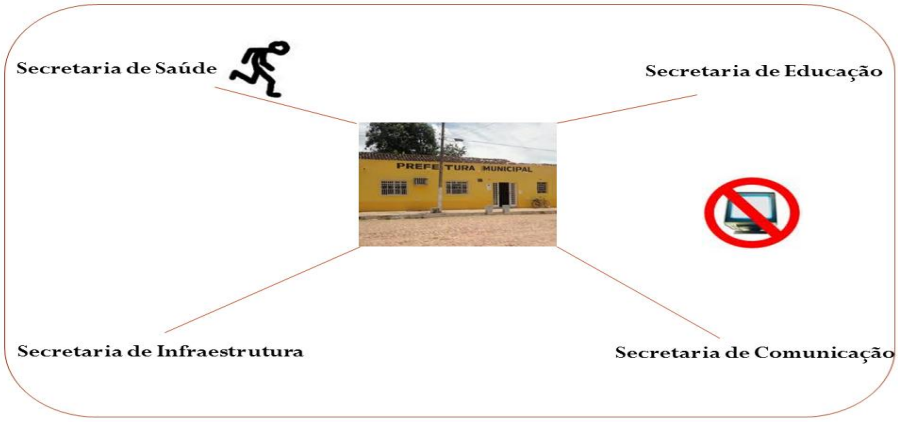


Figura 1. Representação da comunicação interna da Prefeitura Municipal de Balsas – MA

A Figura 1 representa o funcionamento da comunicação interna da prefeitura municipal de balsas, onde as secretarias de saúde, educação, infraestrutura e comunicação entre outras, são repartições da mesma e são localizadas e locais diferentes, onde necessita manter em comunicação com a prefeitura municipal. Se as secretarias desejam enviar documentos

oficiais para a prefeitura municipal de Balsas-MA, um funcionário tem que percorrer um caminho entre as mesmas até a Prefeitura Municipal de Balsas-MA, para realizar a entrega dos mesmos, causando assim: Demora no envio e no recebimento dos documentos oficiais; Riscos de acidentes de trabalho com funcionários na realização da entrega dos documentos oficiais em que tem que se locomover entre as ruas da cidade; Se o prefeito ou funcionários (a) competentes a receberem determinados documentos não estiverem presentes na cidade, eles (a) não terão como receber os documentos.

A necessidade de um sistema automatizado para a Prefeitura Municipal de Balsas-MA, surgiu como ponto de partida ao estudo e desenvolvimento de um sistema web em que foi observado que o mecanismo utilizado para enviar e receber documentos oficiais utilizados pela prefeitura municipal de Balsas-MA e seus setores contém problemas em relação à demora da execução no envio e recebimento de documentos oficiais.

O objetivo deste trabalho foi desenvolver um sistema Web que atenda as necessidades da prefeitura municipal de Balsas-MA em relação a troca de documentos oficiais entre seus setores. Espera-se que o software desenvolvido, quando for implantado, permita minimizar o tempo gasto nos envios e recebimentos dos documentos oficiais, permitindo assim uma rápida comunicação entre seus setores e a mesma.

1.1 Justificativa

A administração pública de Balsas – MA, passa hoje por problemas de comunicação entre os setores que a constitui e a própria prefeitura municipal, onde tudo o que é realizado é documentado e aprovado por secretarias por meio de seus respectivos supervisores ou secretários e pelo Prefeito, em que esses documentos precisam ser repassados para os setores que a constitui. Porém o envio ou a entrega desses documentos é feito manualmente através de um funcionário público, que se desloca de uma partição até outra para realizar a entregar dos mesmos, tornando assim um processo demorado, e que acaba dificultando o processo de organização da mesma.

O sistema web desenvolvido para a Prefeitura Municipal de Balsas-MA, propulsionar que os documentos sejam enviados com mais rapidez e segurança, visando melhorar a comunicação interna da mesma em relação à transmissão de documentos oficiais.

1.2 Objetivos

1.2.1 **Objetivo geral**:

Desenvolver um sistema web para a Prefeitura Municipal de Balsas-MA, que possa enviar, armazenar e receber documentos oficiais.

1.2.2 **Objetivos específicos**:

- Efetuar um levantamento bibliográfico das tecnologias envolvidas no desenvolvimento do sistema web;
- Fazer um levantamento dos requisitos do sistema a ser elaborado;
- Desenvolver o sistema proposto.

1.3 Metodologia

A pesquisa cientifica tem por objetivo a busca de conhecimento e permitir que o mesmo possa ser viável na resolução de determinado problema, com a finalidade de colocar em prática o conhecimento que foi adquirido, suficiente para realizar o desenvolvimento do sistema proposto.

O trabalho será desenvolvido para Prefeitura Municipal de Balsas-MA. No ano de 2013.

A metodologia do trabalho consistirá em:

- Pesquisas Bibliográficas: Serão realizados estudos em livros, tutoriais, publicações e artigos onde possa contribuir no aperfeiçoamento do conhecimento em relação às ferramentas que serão utilizadas para o desenvolvimento do sistema web;
- Será adotada a metodologia de desenvolvimento em Cascata;
- Reuniões com funcionários da prefeitura municipal de Balsas-MA: Com o propósito de aprimorar o conhecimento de como funciona o mecanismo de entrega dos documentos oficiais;
- Levantamento de requisitos: Serão levantados através de entrevista formal com os responsáveis pelas secretarias e setores da administração pública de Balsas-MA;
- Implementação do sistema;

1.4 Organização da monografia

Este trabalho está organizado da seguinte forma: Capítulo 2 Revisão Bibliográfica, Capítulo 3 Desenvolvimento e o Capítulo 4 Conclusão.

2. REVISÃO BIBLIOGRÁFICA

A revisão bibliográfica consistirá na apresentação de conceitos de diferentes autores sobre MVC que será utilizado para separação da aplicação, Hibernate para mapeamento objeto relacional, Spring Security para autenticação e autorização de usuários no sistema, também será apresentado conceitos sobre JSF, Prime Faces e engenharia de software.

2.1 Engenharia de Software

Engenharia de software é metodologia de desenvolvimento e manutenção de sistemas modulares, com as seguintes características: processo (roteiro) dinâmico, integrado e inteligente de soluções tecnológicas; adequação aos requisitos funcionais do negócio do cliente e seus respectivos procedimentos pertinentes; efetivação de padrões de qualidade, produtividade e efetividade em suas atividades e produtos; fundamentação na Tecnologia da informação disponível, viável, oportuna e personalizada; planejamento e gestão de atividades, recursos, custos e datas (REZENDE, 2005).

MAFFEO (1992 apud REZENDE, 2005, p.5.) considera-se que os objetivos primários da Engenharia de Software são o aprimoramento da qualidade dos produtos de software e o aumento da produtividade dos engenheiros de software, além do atendimento aos requisitos de eficácia e eficiência, ou seja, efetividade.

2.1.1 Metodologia de Desenvolvimento de Sistemas

Uma metodologia completa constitui-se de uma abordagem organizada para atingir um objetivo, por meio de passos preestabelecidos. É um roteiro, um processo dinâmico e iterativo para o desenvolvimento estruturado de projetos, sistemas ou software, visando a qualidade, produtividade e efetividade de projetos (REZENDE, 1997).

A metodologia deve auxiliar o desenvolvimento de projetos, sistemas ou software, de modo que os mesmos atendam de maneira adequada ás necessidades do cliente ou usuário, com os recursos disponíveis e dentro de um prazo ideal definido em conjunto com os envolvidos. Não deve limitar a criatividade profissional, mas deve ser um instrumento que determine um planejamento metódico, que harmonize e coordene as áreas envolvidas. O que limita a criatividade não é a metodologia, mas os requisitos de qualidade, produtividade e efetividade de um projeto (REZENDE, 2005).

2.1.2 Processo de Software

Um processo de software é um conjunto de atividades e resultados associados que geram um produto de software. Essas atividades são, em sua maioria, executadas por engenheiros de software (SOMMERVILLE, 2005).

Segundo REZENDE (2005):

> Os processos de software ou processos de desenvolvimento de software têm conceitos equivalentes a metodologia de desenvolvimento e manutenção de software, pois ambos são roteiros de elaboração de software que devem contemplar fases, subfases, produtos externados e pontos de avaliação de qualidade.

Um processo é um conjunto de passos parcialmente ordenados, constituídos por atividades, métodos, práticas e transformações, usado para atingir uma meta. Essa meta está associada a um ou mais resultados concretos finais, que são os produtos da execução do processo (FILHO, 2001).

Segundo SOMMERVILLE (2005), existem quatro atividades de processo fundamentais comuns a todos os processos de software. Essas atividades são:

1. Especificação do software: A funcionalidade do software e as restrições em sua operação devem ser definidas.
2. Desenvolvimento do software: O software deve ser produzido de modo que atenda as suas especificações.
3. Validação do software: O software tem de ser validado para garantir que ele faz o que o cliente deseja.
4. Evolução do software: O software deve evoluir para atender às necessidades mutáveis do cliente.

Diferentes processos de software organizam essas atividades de maneiras diversas e são descritos em diferentes níveis de detalhes. Os prazos das atividades variam, do mesmo modo que os resultados de cada atividades. Diferentes organizações podem utilizar processos diferentes para produzir o mesmo tipo de produto (SOMMERVILLE, 2005).

Os processos de software são complexos e, como todos os processos intelectuais, dependem de julgamento humano. Por causa da necessidade de utilizar o julgamento e a criatividade, tentativas de automatizar processos de software têm tido sucesso limitado (SOMMERVILLE, 2005).

Uma razão pela qual existe uma abordagem limitada para a automação de processos é a imensa diversidade dos processos de software. Não há um processo ideal, e diferentes organizações desenvolveram abordagens inteiramente diferentes para o desenvolvimento de software (SOMMERVILLE, 2005).

2.1.3 Modelos de Processo de Software

Um modelo de processo de software é uma representação abstrata de um processo de software. Cada modelo de processo representa um processo a partir de uma perspectiva particular, de uma maneira que proporciona apenas informações parciais sobre o processo (SOMMERVILLE, 2005).

2.1.4 Engenharia de Requisitos

O processo de engenharia de requisitos leva á produção de uma documentação de requisitos, que é a especificação para o sistema. Os requisitos geralmente são apresentados em dois níveis de detalhes nesse documento (SOMMERVILLE, 2005).

Engenharia de requisitos é um estágio particularmente importante do processo de software, uma vez que erros nesse estágio inevitavelmente produzem problemas posteriores no projeto e na implementação do sistema (SOMMERVILLE, 2005).

Existem quatro fases principais no processo de engenharia de requisitos:

1- Estudo de Viabilidade É feita uma estimativa para verificar se as necessidades dos usuários que foram identificadas podem ser satisfeitas com a utilização das atuais tecnologias de software e hardware. O estudo decidirá se o sistema proposto será viável, do ponto de vista comercial, e se poderá ser desenvolvido certas restrições orçamentárias existentes. Um estudo de viabilidade deve ser relativamente barato e rápido.

2- Levantamento e análise de requisitos Este é o processo de obter os requisitos do sistema pela observação de sistemas existentes, pela conversa com usuários e compradores em potencial, pela análise de tarefas assim por diante.

3- Especificação de requisitos É a atividade de traduzir as informações coletadas durante a atividade de análise em um documento que defina um conjunto de requisitos. Dois tipos de requisitos podem se incluídos nesse documento. Os requisitos dos usuários são declarações abstratas dos requisitos de sistema para

o cliente e os usuários finais do sistema; os requisitos do sistema são uma descrição mais detalhada da funcionalidade a ser fornecida.

4- Validação de requisitos Essa atividade verifica os requisitos quanto a sua pertinência, consistência e integridade. Durante esse processo, inevitavelmente são descobertos erros na documentação de requisitos. Os requisitos devem então ser modificados, a fim de corrigir esses problemas.

2.1.5 Modelos de Ciclo de Vida

Em engenharia de software, processos podem ser definidos para atividades como desenvolvimento, manutenção, aquisição e contratação de software. Podem-se definir subprocessos para cada um desses, um processo de desenvolvimento abrange subprocessos de determinação dos requisitos, análise, desenho, implementação e testes (FILHO, 2009).

No modelo de ciclo de vida em Cascata (Figura 2), os principais sub processo são executados em estrita sequência, o que permite demarcá-los com pontos de controles bem-definidos. Esses pontos de controle facilitam muito a gestão dos projetos, o que faz com que esse processo seja, em princípio, confiável e utilizável em projetos de qualquer escala. Por outro lado, se interpretado literalmente, é um processo rígido e burocrático, em que as atividades de requisitos, análise e desenho têm de ser muito bem dominada, pois, teoricamente, o processo não prevê a correção posterior de problemas nas fases anteriores. O modelo em cascata puro é de baixa visibilidade para o cliente, que só recebe o resultado final do projeto. Na prática, é sempre necessário permitir que, em fases posteriores, haja revisão e alteração de resultados das fases anteriores (FILHO, 2009).

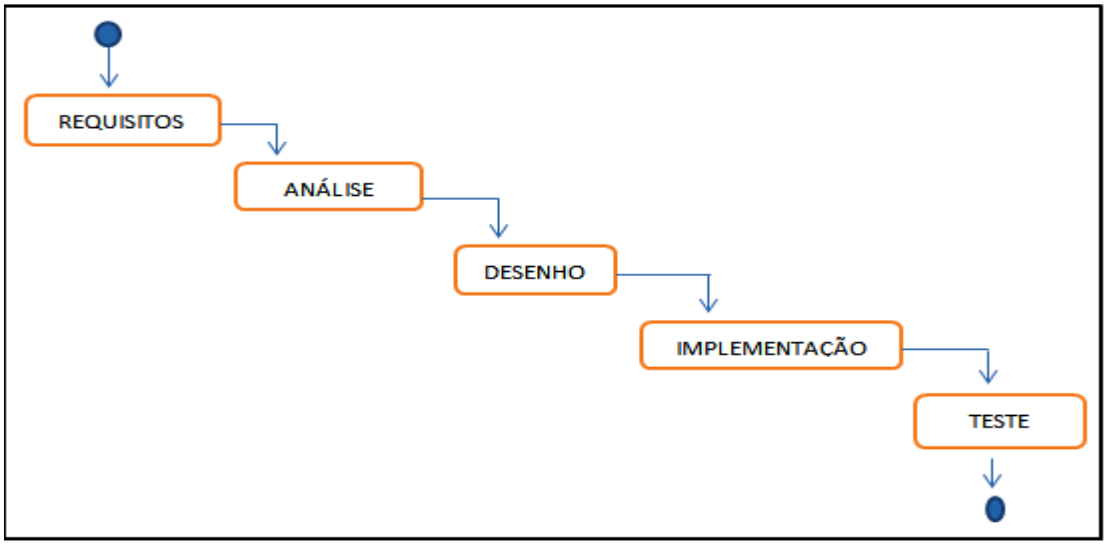


Figura 2: Ciclo de Vida em Cascata.

Requisitos:

Os problemas que os engenheiros de software têm para solucionar são, muitas vezes, imensamente complexos. Compreender a natureza dos problemas pode ser muito difícil, especificamente se o sistema for novo. Consequentemente, é difícil estabelecer com exatidão o que o sistema deve fazer. As descrições das funções e das restrições são os *requisitos* para o sistema; e o processo de descobrir, analisar, documentar e verificar essas funções e restrições é chamado de *engenharia de requisitos* (SOMMERVILLE, 2003).

Alguns dos problemas que surgem durante o processo de engenharia de requisitos são resultantes de falta de uma nítida separação entre esses diferentes níveis de descrição (SOMMERVILLE, 2003).

Os requisitos do usuário, os requisitos de sistema e a especificação de projeto de software podem ser definidos como se segue:

1- *Requisitos do usuário* são declarações, em linguagem natural e também em diagramas sobre as funções que o sistema deve fornecer e as restrições sob as quais deve operar.

2- *Requisitos de sistema* estabelecem detalhadamente as funções e as restrições de sistema. O documento de requisitos de sistema deve ser preciso. Ele pode servir como um contrato entre o comprador do sistema e o desenvolvedor do software.

3- *Especificação de projeto de software* é uma descrição abstrata do projeto de software.

Segundo Sommerville (2003), "os requisitos de sistema de software são, frequentemente, classificados como funcionais ou não funcionais".

1- Requisitos funcionais São declarações de funções que o sistema deve fornecer, como o sistema deve reagir a entradas especificas e como deve se comportar em determinadas situações.

2- Requisitos não funcionais São restrições sobre os serviços ou as funções oferecidas pelo sistema. Entre eles destacam-se restrições de tempo, restrições sobre o processo de desenvolvimento, padrões, entre outros.

Análise:

Fluxo cujo objetivo é detalhar, estruturar e validar os requisitos, de forma que estes possam ser usados como base para o planejamento detalhado (FILHO, 2009).

Desenho:

Fluxo cujo objetivo é formular um modelo do produto que sirva de base para implementação (FILHO, 2009).

Implementação:

Fluxo cujo objetivo é realizar o desenho em termos de componentes de códigos (FILHO, 2009).

Testes:

Fluxo cujo objetivo é verificar os resultados da implementação (FILHO, 2009).

2.2 Netbeans

O NetBeans é considerada uma das melhores IDEs open source do mercado. Desenvolvida pela Sun Microsystems e mantida pela comunidade, a cada nova versão vem se mostrando uma madura e consistente ferramenta para desenvolvimento Java. Em matéria de desenvolvimento Web, essa IDE é muito madura, sendo uma excelente alternativa para aqueles que desejam desenvolver aplicações Web de forma simples e rápida (GONÇALVES, 2007).

Segundo GONÇALVES (2007), SANTOS (2007):

> O NetBeans foi Desenvolvida pela Sun Microsystems e mantida pela comunidade, é considera uma das melhores IDEs open sourse do mercado. È um projeto que se dedica a prover ferramentas aplicáveis ao processo de desenvolvimento de software e que possa ser utilizado por desenvolvedores e demais usuários, entidades e corporações como base para a criação dos seus próprios produtos.

A comunidade desse projeto é um grupo de pessoas interessadas que residem nas mais diversas partes do mundo que contribuem entre si de diversas formas, como, postando na lista de discussões mensagens onde qualquer pessoa pode acessá-las para busca ajuda nos problemas enfrentados com o uso dessa ferramenta e também para compartilhar os sucessos obtidos com elas (SANTOS, 2007).

Segundo Santos (2007), “Qualquer pessoa pode obter gratuitamente o instalador do NetBeans no web site do projeto e está autorizada a fazer uso dele tanto para propósitos não comerciais quanto para fins comerciais”.

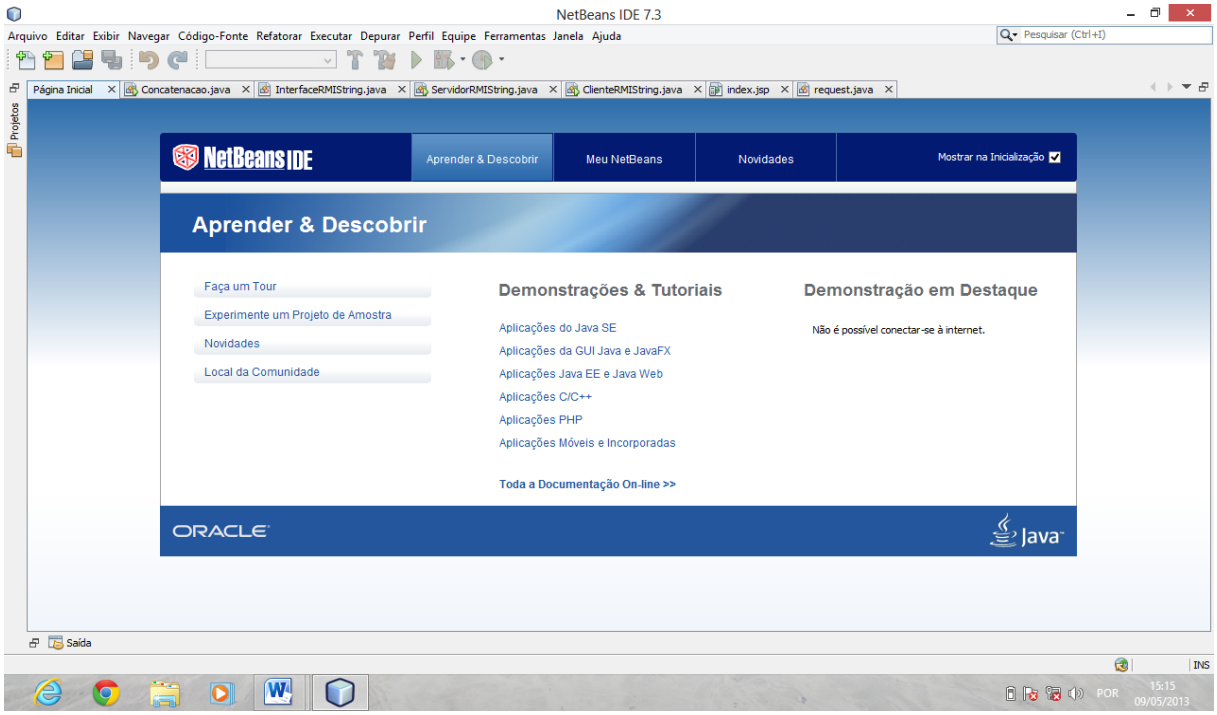


Figura 3- Ambiente de Desenvolvimento NetBeans IDE 7.3

Tela principal do NetBeans IDE 7.3. O mesmo será utilizado como ambiente de Desenvolvimento web para o sistema web proposto.

O NetBeans conta com um sistema de depuração, mostrando variáveis não declaradas, falhas de digitação, métodos inexistentes e ainda possibilitar importações de bibliotecas através de auxilio da ferramenta (GONÇALVES, 2007).

O ambiente NetBeans mostra como principal ferramenta de desenvolvimento com a tecnologia JAVA. Através dele o programador consegue mais recursos que podem auxiliar na produtividade para o desenvolvimento do seu software. Neste ambiente é possível montar uma arquitetura a nível de classes em três camadas, denominada MVC: Modelo-Visão e Controle.

2.3 MVC (Model-View-Controller)

Atualmente ainda é muito comum a prática de desenvolver aplicações web construída com código HTML e código servidor em uma mesma página, criando uma embaraçosa confusão. É exatamente isso que acontece em JDBC e JSTL, códigos de SQL, juntas a códigos de programação, e saídas de resultados ao usuário, tudo em um só local (GONÇALVES, 2007).

Esse tipo de desenvolvimento é conhecido como embutir a lógica de negócios ao resultado final. Essa prática e condenada pelos desenvolvedores atuais, principalmente os de

aplicações escritas em JSP. Graças a essa condenação, o paradigma do modelo MVC foi incorporado no desenvolvimento, criando assim dois modelos para desenvolvimento de aplicações escritas em Java: Model 1 (Modelo 1) e Model 2 (Modelo 2), baseados no paradigma MVC (GONÇALVES, 2007).

O Model 1:

A primeira arquitetura, conhecida como Model 1, é muito comum no desenvolvimento de aplicações Web, chamada de page-centric. Esta arquitetura fornece o modo mais fácil de reunir uma aplicação Web. Envolvendo simplesmente a construção de uma aplicação como um conjunto de páginas JSP (GONÇALVES, 2007)

O Model 2:

É uma arquitetura mais sofisticada que usa Servlets e páginas JSP. Foi batizada de Model 2 (Modelo 2), que está baseada em uma adaptação da arquitetura MVC. Nessa implementação, um Servlet é usado como um *Controlador*, recebendo pedidos do usuário, enquanto efetuando mudanças no *Modelo*, e fornecendo a *Apresentação* ao usuário (GONÇALVES, 2007).

Segundo GONÇALVES (2006):

> As Servlets são classes Java que são instanciadas e executadas em associação com servidores Web, atendendo requisições realizadas por meio de protocolo http. Os objetos Servlets quando acionados podem enviar a resposta na forma de uma página HTML ou qualquer outro conteúdo MIME.

Segundo GONÇALVES (2007):

> O MVC é um conceito (paradigma) de desenvolvimento e design que tenta separar uma aplicação em três partes distintas. Uma parte, a Model, está relacionada ao trabalho atual que a aplicação administrativa, outra parte, a View, está relacionada a exibir os dados ou informações dessa uma aplicação e a terceira parte, Controller, em coordenar os dois anteriores exibindo a interface correta ou executando algum trabalho que a aplicação precisa completar.

A visão é a camada responsável pela apresentação. Ela recebe o estado do modelo através do controlador. Os objetos dessa camada recebem os dados de entrada do usuário que serão repassados para o controlador (FOX, 2006).

A camada do controlador interliga a camada da visão à camada de negócio ou modelo, ou seja, é a fronteira entre ambas as camadas. No framework JSF quem faz o papel do controlador é o servlet FacesContex, que retira da solicitação do usuário os dados de entrada e interpreta o que eles significam para o modelo. Obriga o modelo a se atualizar e disponibiliza o novo estado do modelo para a visão (BASHAM et al, 2008).

O modelo é a camada que abriga a verdadeira lógica de negócio. Possui as regras para obtenção e atualização do estado e fornece ao controlador o acesso aos dados, pois é a única parte do sistema que se comunica com o banco de dados (BASHAM et al, 2008).

A ideia do padrão MVC é dividir uma aplicação em três camadas: modelo, visualização e controle. O modelo é responsável por representar os objetos de negócio, manter o estado da aplicação e fornecer ao controlador o acesso aos dados. A visualização representa a interface com o usuário, sendo responsável por definir a forma como os dados serão apresentados e encaminhar as ações dos usuários para o controlador. Já a camada de controle é responsável por fazer a ligação entre o modelo e a visualização, além de interpretar as ações do usuário e as traduzir para uma operação sobre o modelo, onde são realizadas mudanças e, então, gerar uma visualização apropriada.

- Model: O Model (modelo) é o objeto que representa os dados do programa. Maneja esses dados e controlam todas suas transformações. Esse modelo não tem conhecimento especifico do controlador (Controller) e das apresentações (View), nem sequer contém referência a eles. Portando, o Model são as classes que trabalham no armazenamento e busca de dados. Por exemplo, um cliente pode ser modelado em uma aplicação, e pode haver vários modos de criar novos clientes ou mudar informações de um relativo cliente.
- View: A View (apresentação) é o que maneja a apresentação visual dos dados apresentados pelo Model. Em resumo, é a responsável por apresentar os dados resultantes do Model ao usuário. Por exemplo, uma Apresentação poderá ser um local administrativo se logam em uma aplicação. Cada administrador poderá visualizar uma parte do sistema que o outro não vê.
- Controller: O Controller (controlador) é o objeto que responde as ordens executadas pelo usuário, atuando sobre os dados apresentados pelo modelo, decidindo como o modelo devera ser alterado ou devera ser revisto e qual apresentação devera ser exibida. Por exemplo, o controlador recebe um pedido para exibir uma lista de clientes interagindo com o Modelo e entregando uma apresentação onde esta lista poderá ser exibida.

2.3.1 Padrão MVC segundo JSF

No JSF, o controle é composto por um Servlet denominado FacesServlet, por arquivos de configuração e por um conjunto de manipuladores de ações e observadores de eventos. O FacesServlet é responsável por receber requisições da WEB, redirecioná-las para o modelo e então remeter uma resposta. Os arquivos de configuração são responsáveis por realizar associações e mapeamentos de ações e pela definição de regras de navegação. Os manipuladores de eventos são responsáveis por receber os dados vindos da camada de visualização, acessar o modelo, e então devolver o resultado para o FacesServlet (KITO, 2005).

Segundo KURNIAWAN (2002):

> Um servlet é uma classe Java que pode ser automaticamente carregada e executada por um Container. O container gerencia as instâncias dos servlets, além de prover os serviços de redes necessários para as requisições e respostas.

Com o padrão MVC será possível dividir a aplicação em três camadas: modelo, visualização e controle. Através desse padrão será possível fazer a manutenção do sistema com mais facilidade (GONÇALVES, 2007). O JSF basear-se nesse padrão de projeto, e uma de suas melhores vantagens e a clara separação entre a visualização e regras de negócio (modelo), o mesmo será utilizado como linguagem de programação web no desenvolvimento do trabalho.

2.4 JavaServer Faces

O Java Server Faces JSF é um framework utilizado para desenvolvimento de aplicações Web, que executam do lado do servidor, desenvolvidas pela Sun Microsystem. Essa tecnologia faz parte da plataforma J2EE (Java 2 Platform Enterprise Edition) que oferece um conjunto de tecnologias e soluções robusta para a Web.

Segundo Gonçalves (2006), “Um framework é uma estrutura de suporte definida, em que um outro projeto de software pode ser organizado e desenvolvido”.

Segundo Kito (2005):

> O JSF é executado do lado do servidor em um container Web padrão como, por exemplo, o Tomcat. Quando o usuário clica em um botão em uma aplicação swing2 ele vai disparar um evento que pode ser tratado diretamente no código do programa desktop. Em contraste, os navegadores Web não conhecem os componentes do JSF ou eventos, pois o browser só sabe interpretar HTML (HyperText Markup Language). Então quando um botão é clicado pelo usuário em um aplicativo Faces é gerada uma requisição que será enviada através do browser para o servidor pelo protocolo HTTP. O Faces é responsável por traduzir essa requisição em um evento que será processado pelo aplicativo no servidor. É responsável também por garantir que cada elemento da interface do usuário definidas no servidor seja exibida corretamente para o navegador.

Segundo GONÇALVES (2007):

> JSF (JavaServer Faces) é uma tecnologia do mundo Java EE e é desenhado para simplificar o desenvolvimento de aplicações web. JSF torna fácil o desenvolvimento através de componentes de interfaces de usuário (GUI) e conecta esses componentes a objetos de negócios. Também automatiza o processo de uso de JavaBeans e na navegação de páginas.

Java Server Faces é o framework para desenvolvimento de aplicação web padrão da Java EE. Ele é mantido pela Java Community Process JSR-314, que define o padrão para desenvolvimento de interfaces dentro do modelo orientado a componentes. Essa característica é, na maioria das vezes, o maior obstáculo para o aprendizado da maioria dos desenvolvedores, principalmente os que conhecem algum framework baseados em ações, como por exemplo, Struts (SOUSA, p.8-18).

O objetivo principal do JSF é tornar o desenvolvimento Web mais rápido e fácil. Permite aos desenvolvedores pensar em termos de componentes, eventos, backing beans ou Managed Beans. Managed Beans são Java Beans3 que podem ser usados para gerenciar melhor o comportamento dos componentes em uma página JSP. Na prática você pode criar backing beans diferentes para um ou mais componentes em uma página. O JSF permite pensar em estruturas de “alto nível” e suas interações, ao invés de requisição e resposta. A complexidade do desenvolvimento Web não é visualizada pelos programadores, permitindo que os desenvolvedores se concentrem mais na lógica de negócio da sua aplicação (KITO, 2005).

A arquitetura JSF favorece o surgimento de ferramentas RAD (Desenvolvimento rápido de aplicações) através da arquitetura de componentes, da infraestrutura da aplicação e do conjunto de componentes padrões. Para a IDE Eclipse, há diversos plug-ins, como o WTP, o Jboss Tools e MyEclipse. Já o NetBeans oferece um suporte avançado com recursos de *drag-and-drop* para montagem de páginas, além de oferecer componentes mais avançados que os

padrões, cabe ressaltar que estes recursos estão atrelados a um modelo de desenvolvimento que o usuário deve seguir, caso contrário os recursos são bem limitados (SOUSA, p.8-18).

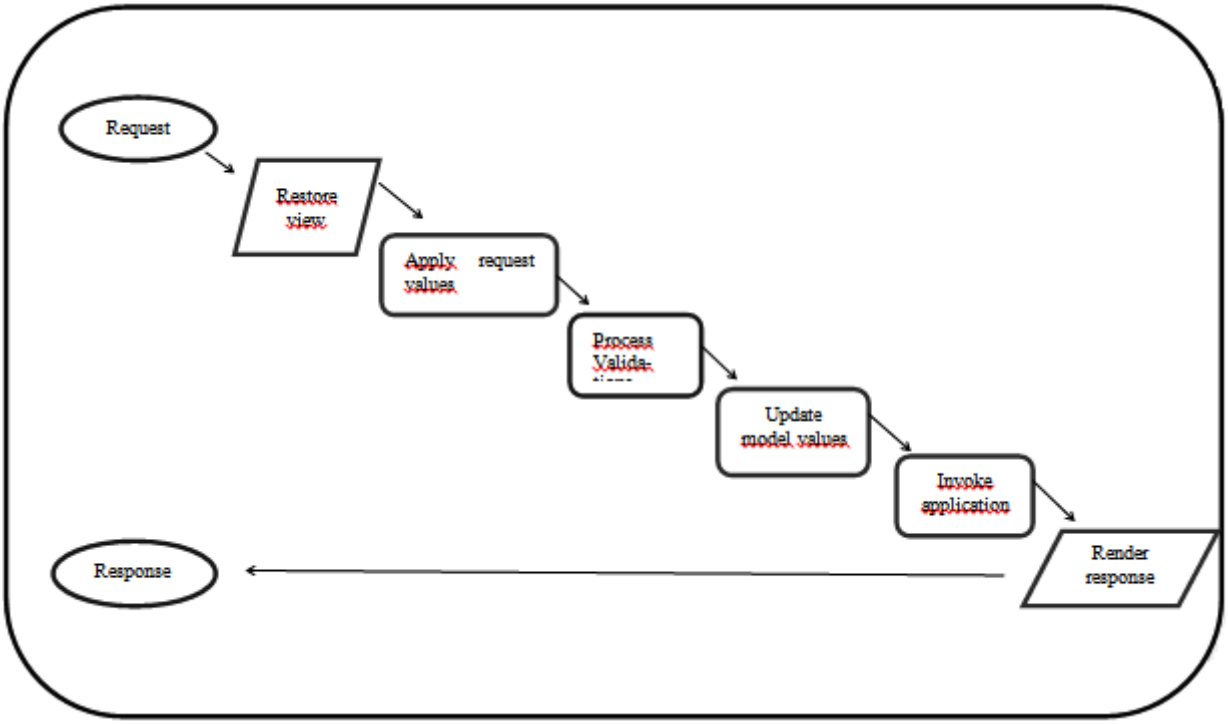
Os componentes JSF são orientados a eventos, ou seja, é possível processar eventos gerados pelo cliente, como um clique no botão ou alteração do valor de um campo de texto. Por padrão, há um conjunto de componentes básicos para manipulação de formulários, mas o *Faces* oferece uma infraestrutura para criação de novos componentes. Dentre os componentes de terceiros, os mais conhecidos são os Jboss RichFaces e Apache Tomahawk (SOUSA, p.8-18).

O framework JSF é responsável por interagir com o usuário (cliente), e fornece ferramentas para criar uma apresentação visual, a aplicação lógica e a lógica de negócios de uma aplicação Web. Porém, o escopo de JSF é restringindo à camadas de apresentação. A persistência de banco de dados e outras conexões de back-end estão fora do escopo de JSF (GONÇALVES, 2007).

2.4.1 Ciclo de vida do JSF

Toda requisição realizada dentro do contexto JSF passa por um processo de seis fases, conhecidas como ciclo de vida JSF. Em suma, durante esse processo é restaurada a árvore de componentes, os valores são lidos, convertidos e validados; eventos são executados e uma resposta é gerada para o usuário. Os eventos são executados, na grande maioria, após uma dessas seis fases. Numa requisição podemos passar por todas essas fases ou por nenhuma, dependendo do tipo de pedido, de erros que ocorrem durante as validações, conversões e do tipo de resposta (SOUSA, p.8-18).

A **Figura 4** mostra o fluxo de execução de uma requisição gerada pelo cliente pelo ciclo de vida JSF. O processo inicia-se assim que a requisição é recebida pelo *servlet* do JSF. É importante lembra que JSF é construído em cima da API do Servlet.



. **Figura 4.** Ciclo de vida JSF

Restore View

A requisição é recebida pelo **FacesController**, que extrair a *View ID* usada para identificar a página JSP associada a ela. Uma *View* é a representação de todos os componentes que compõem uma determinada página. Uma vez que a página está em mãos, é realizada uma tentativa de restauração desta *View* que, geralmente, é restaurada com base em um campo oculto, ou baseada na sessão de usuário. A árvore de componentes da *View* é obtida através de duas formas: *Initial View* e *Postback*, que são executados de maneiras distintas (SOUSA, p.8-18).

A *Initial View* ocorre quando a página é acessada pela primeira vez ou quando a página é acessada via HTTP GET. O contexto JSF cria árvore de componentes, com base na *View* acessada, e a mesma é gravada na propriedade **viewRoot** do objeto **FacesContext**. Esse objeto contém todas as informações relacionadas à requisição que são necessárias para manipular o estado dos componentes GUI de uma página de uma determinada requisição. Como não há nenhum valor de entrada a ser processado, o fluxo avança para a última fase, a *Render Response* (SOUSA, p.8-18).

Já o *Postback* ocorre quando o usuário interage com o formulário da página, seja alterando um valor de campo de texto ou clicando num link ou botão. Nesses casos, a requisição é

enviada para a mesma página da qual foi submetido o formulário. Como a *View* já existe, ela precisa ser restaurada; nesse caso, o JSF utiliza as informações restauradas para reconstrução do estado da *View*. Uma vez reconstruído o estado, o fluxo passa para a próxima fase do ciclo de vida (SOUSA, p.8-18).

Apply Request Values

Também conhecido como processo de decodificação, a fase *Apply Request Values* é responsável por atribuir aos componentes o valor submetido através de parâmetros enviados no *request*. Todo componente que aceita a entrada de valores possui uma propriedade que recebe o valor original enviado pelo usuário. Esse valor é obtido com base do ID do componente. Um componente pode ser renderizado mais de uma vez, geralmente em virtude de uma interação. Para garantir um ID único para cada componente, o JSF adiciona como prefixo o ID do componente pai, o ID resultante é conhecido como *Client Id*. Por exemplo, o *Client Id* de uma caixa de texto com ID "nome" dentro de um formulário com ID "cadastro" será "cadastro:nome" (SOUSA, p.8-18).

Todo componente possui uma propriedade **immediate** que indica se o valor deve ser convertido e validado nesta fase. Já em componentes do tipo comando, como um *link* ou botão, essa propriedade é usada para ignorar todos os valores do formulário. A utilização dessa opção será para botões cancelar ou para *link* que aceitam valores de um controle específico. Concluída a decodificação, é verificado se houve algum erro de validação ou conversão. Caso exista algum erro, o ciclo avança para a fase *Render Response*; caso contrário, o ciclo avança para a próxima fase (SOUSA, p.8-18).

Process Validation

Nesta fase é assegurado que todos os valores enviados são válidos. A validação é desempenhada diretamente pelo componente e também pode ser delegada para um ou mais validadores. Antes disso, o valor submetido pelo usuário precisa ser convertido, o que é feito por meio de um conversor padrão ou através de um conversor específico (SOUSA, p.8-18).

Uma vez validado o valor enviado pelo usuário, o mesmo é verificado com o valor do componente; caso seja diferente, é gerado um evento de alteração de valor. Nesse ponto, eventos *value-change* e qualquer outro evento associado a esta fase são executados pelos *listeners* apropriados. Caso todos os valores submetidos sejam válidos, a execução passa para a

próxima fase do ciclo; em caso de erros, a página é renderizadas com as mensagens de validação (SOUSA, p.8-18).

Update Model Values

Nesta fase temos os valores enviados atualizados, convertidos nos tipos de dados desejados, validados e associados aos objetos de modelo e aos *Backing Beans*. E tudo isso sem qualquer código de aplicação, essa é a grande vantagem de se usar Faces: o desenvolvedor foca apenas na implementação das lógicas de negócio, e as tarefas repetitivas e tediosas são executadas automaticamente. Completada essa fase, a execução para a próxima fase (SOUSA, p.8-18).

Invoke Application

Essa fase é responsável por chamar qualquer evento ou ação registrados para a requisição. Nesse momento, a aplicação tem o estado necessário para executar todos os eventos e lógicas de negócios da aplicação. Há dois tipos de métodos que podem ser chamados neste momento: *action handlers* e *event listeners* (SOUSA, p.8-18).

Action handlers são usadas para controle de paginação. Nesse contexto, dependendo da lógica de negócio processada, é definida a qual página o usuário deve ser redirecionado. Um *action handlers* é um método de um *Backing Bean* sem argumentos que retorna uma string que determinará a página de destino. Por ser um método sem argumentos, o *action handler* não contém a informação de qual componente chamou a ação (SOUSA, p.8-18).

Event listener não tem retorno e recebe como parâmetro um objeto do tipo **ActionEvent**, que contém informações relacionadas ao evento, como o componente que o originou, a **phaseId** relacionada ao ciclo de vida JSF. Por não ter retorno, este tipo de ação é ideal para executar lógica de negócios quando o usuário clica em um botão e um campo é atualizado, ou selecionando uma opção de uma caixa de seleção, que determina a exibição de valores adicionais. Os métodos *listener* podem ser criados dentro de *Backing Beans* ou em classes separadas, mas há uma vantagem em se criar classes separadas que é a possibilidade de reutilização em mais de uma página (SOUSA, p.8-18).

Render Response

Esta é a ultima fase do ciclo de vida JSF, todo o processamento no nível de *framework* e no nível da aplicação já foi realizado. Essa última fase tem dois objetivos: o primeiro é gerar e enviar a resposta para o usuário e o segundo é salvar o estado da *View* para ser restaurada no próximo *request*, caso a página venha requisitá-la novamente (SOUSA, p.8-18).

Durante a renderização dos componentes, os conversores são novamente chamados para transformar o objeto em uma String para ser visualizada. Depois de completada a fase, o *container web* transmite fisicamente a resposta para o usuário, a qual é exibida no navegador (SOUSA, p.8-18).

2.4.2 Páginas HTML

Uma página web é um documento adequado para *Word Wide Web* que pode ser acessado por um navegador web e apresentado em algum dispositivo de saída como tela de computador, celular, TV ou similares. Em geral. E declarada usando HTML ou XHTML. Pode ser estática ou ser gerada dinamicamente no servidor web. É composta por componentes visuais tais como caixas de combinação (*comboboxes*) (NUNES; MAGALHÂES, p. 22-36).

Em JSF declaramos uma página web dinâmica usando Facelets ou JSP. Esta declaração é representada em memória como um conjunto de componentes visuais e não visuais chamado árvore de componentes. Uma das responsabilidades da árvore de componentes é renderizar a página web (NUNES; MAGALHÂES, p. 22-36).

O processamento da árvore de componentes inicia quando um evento de um componente de ação é enviado ao servidor. Por exemplo, o clique de um botão. Este processamento é composto por seis fases: identificação ou criação da árvore (1), recuperação dos parâmetros da requisição (2), conversão e validação dos parâmetros (3), atualização de valores nos controladores de página (4), invocação da ação (5), e rederização da resposta (6), (NUNES; MAGALHÂES, p. 24-35).

Um recurso interessante das páginas JSF é que o desenvolvedor pode desenhá-las ainda sem controladores associados e já ter a noção aproximada da identificação visual final. Em outras palavras, não é necessário fazer páginas em HTML primeiro e depois converter em XHTML do JSF. As páginas podem ser desenhadas com os componentes que serão usados na aplicação, mais ainda sem classes Java associadas (NUNES; MAGALHÂES, p. 24-35).

2.4.3 Navegação entre páginas

Conceitualmente, o sistema de navegação JSF é semelhante ao sistema de navegação do Struts. Só que em JSF não temos as tão conhecidas “ActionForward”, mas, por outro lado, temos o *navigation handler*, que é o coração do sistema de navegação. E é ele que determina qual será a próxima página a ser carregada. A base de trabalho de um *navigation handler* é a resposta de *action events*. E a resposta de uma *action event* está associada com um *action method* que executa uma determinada lógica de negócio e retorna uma determinada String como resultado ou pode ser um valor fixo na propriedade *action* de um link ou um botão. As regras de navegação são definidas no arquivo de configuração (SOUSA, p.14-26).

2.4.4 Componentes

Em páginas JSF declaramos os componentes visuais e não visuais que formarão a árvore de componentes. O conceito de componente criado no JSF é muito rico: classes Java definem comportamentos e atributos de componentes que podem ser renderizados não só em HTML, como também em outras linguagens de marcação (NUNES; MAGALHÂES, p. 24-35).

Ainda na renderização HTML, a infraestrutura para componentes em JSF possibilitou que o conjunto básico de componentes visuais fosse significativamente ampliado por bibliotecas de componentes de terceiros, como Apache Tomahawk, Jboss Richfaces, Oracle ADF Faces e IceSoft IceFaces (NUNES; MAGALHÂES, p. 24-35).

Para o desenvolvedor isto significa que funcionalidades interativas desejadas na web e bastaknte conhecidas em aplicações desktop estão disponíveis sem que este desenvolvedor precise dominar HTML, Javascript e CSS (NUNES; MAGALHÂES, p. 24-35).

Entretanto, em uma aplicação não é rara a situação em que precisamos criar componentes compostos: combinar vários componentes prontos em uma área para reuso. Uma área de login, uma área de botões de ação, um menu com busca integrada e uma caixa de texto combinada com suas respectivas mensagens de erro e informação são alguns exemplos (NUNES; MAGALHÂES, p. 24-35).

No JSF o conceito de componentes compostos foi incorporado a partir das ideias de templating do Facelets. Entretanto, evoluído a partir dos maduros recursos existentes na solução Tag Files do JSP e dos próprios conceitos de componentes já disponíveis na especifica-

ção. Para criar um componente composto JSF basta criar um arquivo XHTML e publicá-lo na pasta de recursos (NUNES; MAGALHÂES, p. 24-35).

Managed Beans (Controladores de página)

Controladores de páginas têm como responsabilidade disponibilizar dados e métodos que possam ser referenciados por uma página web para entrada de dados, apresentação de dados ou tratamento de eventos. Em JSF os controladores são chamados de *Managed Beans* (SOUSA, p.14-26).

Existem dois tipos de controladores: aqueles que não referenciam as classes de componentes no código Java e o que referenciam. Os primeiros são chamados *backing beans* e a referência à classe de componente é chamada vínculo a componentes ou *Component Binding.* Os controladores de página que não referenciam componentes são chamados POJOs e apenas utilizam vínculo a dados (*Data Binding*) e vínculo a métodos (*Method Binding*). Nos controladores de página JSF a configuração do *Managed Bean* e feita usando a anotação @**ManagedBean** e alguma das anotações de escopo (SOUSA, p.14-26).

O JSF será utilizado como linguagem de programação para o desenvolvimento do sistema Web utilizando como ambiente de desenvolvimento o NetBeans, junto com o Hibernate para mapeamento objeto/relacional.

2.5 Hibernate

Hibernate é uma ferramenta para mapeamento objeto/relacional para ambiente Java. O termo mapeamento objeto/relacional (ORM) refere-se à técnica de mapeamento de uma representação de dados em um modelo de objetos para um modelo de dados relacionais, baseado em um esquema E/R. O hibernate não cuida somente do mapeamento das classes Java para tabelas do banco de dados (e dos tipos de dados Java para os tipos de dados SQL), mas também provê facilidades para consultar e retorna os dados da consulta, e pode reduzir significativamente o tempo de desenvolvimento em contrapartida ao tempo gasto pelas operações manuais dos dados feitas com SQL e JDBC (PRIMO; SANTOS, p.46-57).

Segundo Gonçalves (2007):

> O Hibernate é um framework que se relaciona com o banco de dados, onde esse relacionamento é conhecido como mapeamento objeto/relacional para Java, deixando o desenvolvedor livre para se concentrar em problemas da lógica do negócio. Sua simplicidade em configuração, dá ao desenvolvedor algumas regras para que sejam seguidas como padrões de desenvolvimento ao escrever sua lógica de negócio e suas classes persistentes. De resto, o Hibernate se integra suavemente ao seu sistema se comunicando com o banco de dados como se fosse diretamente feito por sua aplicação. Uma mudança de banco de dados, nesse caso, não se torna traumática, alterando apenas um ou outro detalhe na configuração do Hibernate.

O Hibernate é uma ferramenta de consulta e persistência objeto/relacional de alta performance. Uma das soluções ORM mais flexíveis e poderosas do mercado, ele faz o mapeamento de classes Java para tabelas de banco de dados e de tipos de dados Java para tipos de dados SQL (PRIMO; SANTOS, p.46-57).

Ele fornece consultas e facilidades para retorno dos dados que reduzem significativamente o tempo de desenvolvimento. A meta do projeto do Hibernate é aliviar os desenvolvedores de 95% das tarefas comuns de programação relacionada à persistência, como a codificação manual com SQL e a API JDBC. O Hibernate gera o SQL para a aplicação, não necessitando o tratamento dos "*result sets*" (comuns nas conexões manuais com JDBC), faz a conversão entre registros e objetos e permite que sua aplicação seja portável para qualquer banco de dados SQL. (PRIMO; SANTOS, p.46-57).

Arquitetura

O projeto Hibernate é composto por vários pacotes Java. Cada pacote tem uma funcionalidade específica e alguns deles só são disponíveis a partir da versão 5.0 do Java SE e do Java EE. A Figura 5 apresenta um quadro ilustrativo.

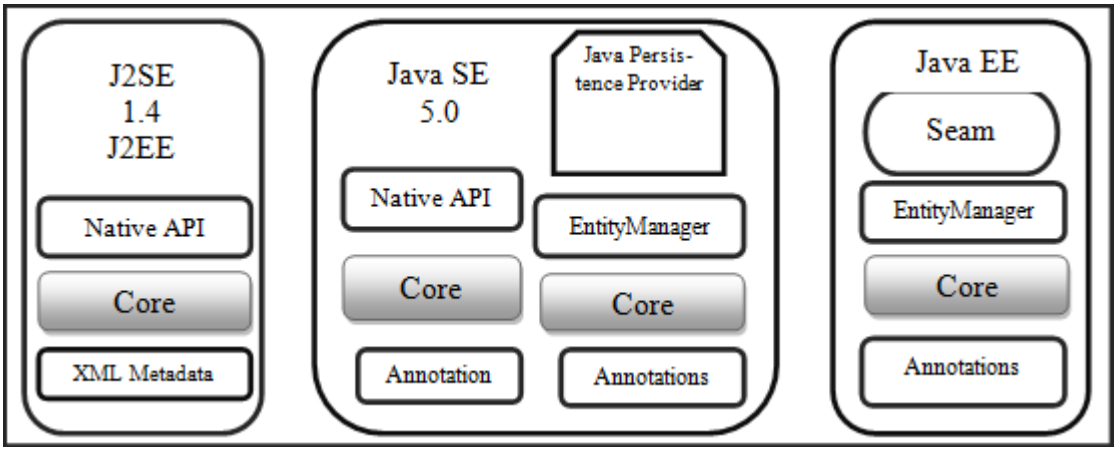


Figura 5: Arquitetura do Hibernate para plataforma Java.

Os pacotes apresentados na Figura 5 são obrigatórios para o desenvolvimento utilizando o Hibernate. O único pacote que deve estar sempre presente independente da plataforma Java utilizada é o *Core*. Além desses pacotes existem outros que agregam outras funcionalidades ao framework e que podem ser adicionados mediante a necessidade do desenvolvedor da aplicação, são eles: Hibernate Shard, Hibernate Validator, Hibernate Search e Hibernate Tools (PRIMO; SANTOS, p.46-57). A Tabela 1 apresenta uma lista e uma descrição resumida de todos os pacotes.

Pacote	***Descrição***
Hibernate Core	Hibernate para Java, APIs nativas e metadados XML
Hibernate Annotation	Mapeia as classes com anotações do JDK 5.0
Hibernate EntityManager	API de persistência Java padrão para Java SE e Java EE
Hibernate Shards	Framework para particionamento horizontal de dados
Hibernate Validator	API para anotação e validação da integridade dos dados
Hibernate Search	Integração do Hibernate com o Lucene para indexação e consulta de dados.
Hibernate Tools	Ferramentas de desenvolvimento para Eclipse e Ant.
Jboss Seam	Framework para aplicação JSF, Ajax e EJB 3.0/Java EE 5.0.

Tabela 1. Lista dos pacotes Hibernate e uma descrição resumida de cada um deles

Hibernate Core

O Hibernate Core está presente em todas as possíveis arquiteturas Java onde se possa utilizar o Hibernate. O Hibernate provê *transparent persistence* (persistência transparente); o único requisito para uma classe ser persistente é ter declarado um construtor *default*, ou seja, sem argumentos. Ele oferece ainda opções de consulta sofisticadas, seja com SQL diretamente, ou uma linguagem de query orientada a objeto e independente de SGBD – HQL (Hibernate Query Language), ou ainda, com uma API de queries “Criteria” muito útil para gerar queries dinamicamente (PRIMO; SANTOS, p.46-57).

Outras características do Hibernate Core:

- Modelo de Programação Natural – Hibernate suporta o idioma OO natural: herança, polimorfismo, composição e coleções Java;
- Suporte para modelo de objetos com granularidade fina – uma rica variedade de mapeamento para coleções e objetos dependentes;
- Sem aumento no tempo de construção do *bytecode* – Não acontece a geração de código extra ou de *bytecode* no processo de *build* da aplicação.

Hibernate Annotations

O Hibernate, como toda ferramenta para mapeamento objeto/relacional, requer metadados que governem as transformações dos dados de uma representação para outra (e vice-versa). Estes metadados eram tradicionalmente fornecidos por arquivos XML, mas a partir da versão 3.2 do Hibernate, pode-se usar as anotações do JDK 5.0 para fazer o mapeamento (PRIMO; SANTOS, p.46-57).

Hibernate EntityManager

A especificação EJB 3.0 reconhece a vantagem e o sucesso do mapeamento objeto/relacional feito de forma transparente, estabelecendo uma API padrão inspirada no Hibernate. O Hibernate EntityManager, junto com o Hibernate Annotations, implementam as interfaces e as regras do ciclo de vida definidos pela nova especificação EJB 3, utilizando por baixo a maturidade do Hibernate Core (PRIMO; SANTOS, p.46-57).

O Entity Manager é o objeto principal da API EntityManager. Ele é usado para acessar um banco de dados de uma determinada unidade de persistência. Uma unidade de persistência define o banco de dados e as respectivas classes persistentes para as quais o Entity Manager provê suporte para criação, atualização, remoção e consultas. Uma aplicação poderá definir várias unidades persistentes, uma para cada banco de dados que acessar (PRIMO; SANTOS, p.46-57).

2.6 Spring Security

O Spring Security surgiu da necessidade de melhorar o suporte à segurança oferecido pela especificação Java EE. O framework centraliza a configuração em um único XML, dispensando configurações do container e tornando a aplicação web um arquivo WAR auto contido (ZANINI, p.28-38).

Segundo Jamacedo (2013), “Spring Security é um framework responsável por cuidar da autenticação e autorização de usuários em aplicações web”.

Para começar a utilizá-lo basta adicionar seus JARs ao classpath, configurar um filtro e um listener no web.xml e criar um application contexto (XML de configuração). O XML centraliza todas as configurações de autenticação e autorização. As tags <intercept-url> definem quais roles podem acessar cada grupo de URLs. A tag <authentication-provider> define a fonte de dados para as informações de usuários (banco de dados, arquivos de propriedades, LDAP, etc.) (ZANINI, ano).

Quando necessário, é possível utilizar os eventos publicados pelo framework a cada sucesso ou falha na autenticação ou autorização. Ouvir os eventos permite criar complexos casos de gerenciamento de usuários. O Spring Security ainda oferece integração com a API de servlets, taglibs para facilitar a codificação de JSPs, suporte à HTTPS, segurança em métodos com uso de anotações e suporte a autenticação com LDAP ou certificados X509 (ZANINI, p.28-38).

Com o Spring Security é possível criar um mecanismo de autenticação e autorização para aplicações web em questão de minutos. O framework foca facilitar a implementação dos casos de uso mais frequentes, porém oferece valiosos pontos de extensão para requisitos mais complexos. Por fim, disponibilizar suporte a inúmeros diferentes tipos de autenticação e integração com as mais usadas tecnologias na área de segurança (ZANINI, p.28-38).

O Spring Security e aplicável a qualquer aplicação web que necessite restringir seus recursos para diferentes tipos de usuários, bem como assegurar que se autentiquem de forma prática e segura (ZANINI, p.28-38).

2.7 PrimeFaces

A biblioteca do PrimeFaces é composta por cerca de 100 componentes de interface todos estes personalizados e de código aberto, e implementa tecnologia AJAX, gráficos ani-

mados JavaScript entre outros. A utilização do PrimeFaces em projetos proporciona uma infinita gama de possibilidades para criação de layouts já que em seu ShowCase a cerca de 30 temas diferentes que podem ser facilmente adicionados ao seu projeto (Santos, et.al. 2010).

Segundo GONÇALVES (2007):

> AJAX é a sigla de **Asynchronous JavaScript and XML**, não é uma tecnologia e sim o uso de tecnologias incorporadas que tem as principais o Javascript e o XML, onde juntos são capazes de torna o navegador mais interativo, utilizando-se de solicitações assíncronas de informações. É sim um conjunto de tecnologia; cada uma com sua forma de exibir e interagir com o usuário.

O primeFaces será usado na construção das interfaces, ela é uma biblioteca que define componentes JSF.

3. DESENVOLVIMENTO

A partir de agora será dado início ao processo de desenvolvimento do projeto. Colocando em prática todos os conhecimentos adquiridos, através dos estudos realizados sobre as diversas ferramentas que serão utilizadas, bem como um maior aprofundamento do conteúdo no decorrer do desenvolvimento, onde serão mostrados os passos importantes para o desenvolvimento e funcionamento do mesmo.

3.1 Estrutura do Projeto

O projeto sysPrefeitura apresentado na Figura 6, teve início com as configurações dos arquivos XML, que se encontra no diretório WEB-INF, são eles, os arquivos applicationContext.xml e web.xml. No projeto ainda foram criadas quatro pastas:

1. **Resources:** Contém as pastas css e imagens.
2. **Sistema:** Contém a página index.xhtml que só poderá ser acessadas por usuários logado no sistema.
3. **Templates:** Contém uma página que define o layout da página em topo, conteúdo e rodapé.
4. **Páginas Web:** Contém as páginas index.xhtml, que terá o formulário de login, a página login_falhou.xhtml e acesso_negado.xhtml, que será acionadas caso os dados do usuário esteja incorretos.

O projeto ainda conta com o pacote de código fonte que contém os pacotes Default, com.sysprefeitura.beans, com.sysprefeitura.model.daos, com.sysprefeitura.model.entidades e com.sysprefeitura.util.

O pacote com.sysprefeitura.beans possui a classe UsuarioBean. O JavaBeans são classes que possuem o construtor sem argumentos e métodos de acesso get e set. (GONÇALVES, 2007).

O pacote com.sysprefeitura.daos possui a classe UsuarioDAO. O padrão DAO (Data Access Object) é o padrão mais utilizado para acesso a dados contido em um banco de dados. Ele fornece uma interface independente, na qual você pode usar para persistir objetos de dados. A ideia é colocar todas as funcionalidades encontradas no desenvolvimento de acesso e trabalho com dados em um só local, tornando simples sua manutenção (GONÇALVES, 2007).

Tipicamente um DAO inclui métodos para inserir, selecionar, atualizar e excluir objetos de um banco de dados. Dependendo de como é implementado o padrão DAO, pode se ter um DAO para cada classe de objetos em uma aplicação ou poderá ter um único DAO que é responsável por todos os objetos (GONÇALVES, 2007).

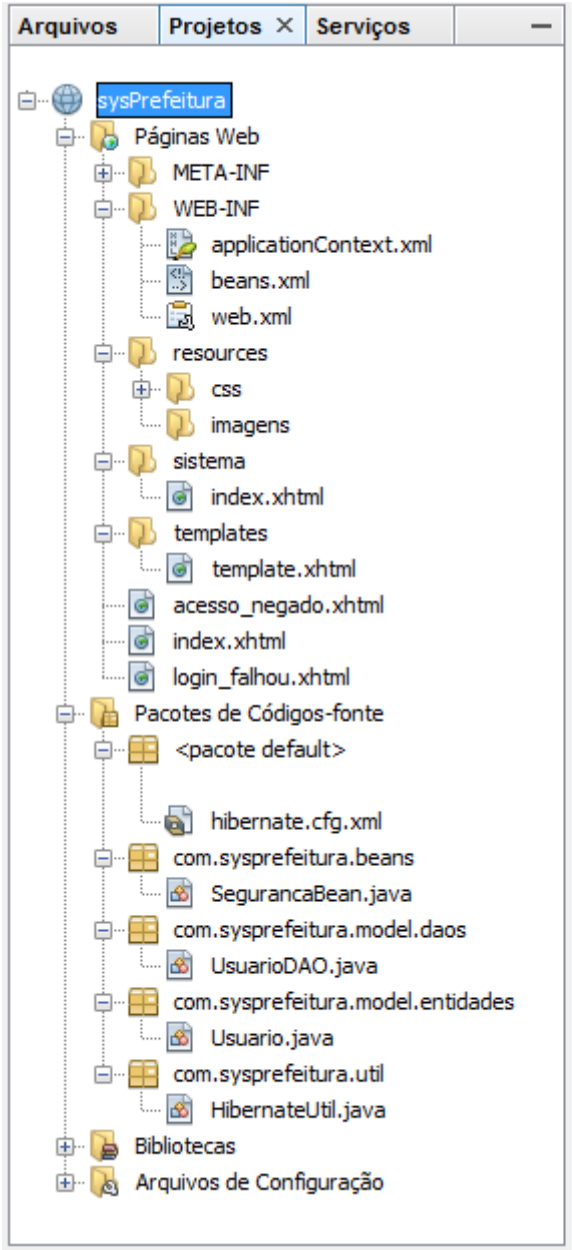


Figura 6: Projeto sysPrefeitura

3.2 Configuração do arquivo web.xml

O diretório *WEB-INF* possui o arquivo *web.xml*, que é o *Deployment Descriptor* da aplicação. Ele contém informações de configuração como parâmetros de inicialização, mapeamento de *Servlets* entre outros (PEREIRA, p.52-60).

Para facilitar o entendimento do arquivo *web.xml*, o mesmo foi dividido em varias partes separadas por comentários. A Figura 6 apresenta a estrutura do projeto e onde se encontra o arquivo *web.xml*.

Na primeira parte da configuração do arquivo *web.xml* foi configurado o JSF apresentado na Figura 7, incluído o *servlet Faces Servlet*, que será responsável pelas renderizações feitas pelo JSF.

```
<context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
</context-param>
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.xhtml</url-pattern>
</servlet-mapping>
```

Figura 7: Configuração do JSF.

Na segunda parte temos a configuração do *Spring* apresentado na Figura 8, onde está sendo mapeado um *listener* (***ContextLoaderListener***) que é responsável por inicializar o contexto do *Spring* na aplicação. Esse contexto será utilizado pelo *Spring* para fabricar beans, injetar as dependências, publicar eventos, entre outros (PEREIRA, p.52-60).

```
<!-- Spring -->
  <listener>
    <listenerclass>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
<!-- Spring -->
```

A Figura8**:** Apresenta a configuração do Spring**.**

Na terceira parte temos a configuração do Spring Security apresentado na Figura 9, que descrever um filtro HTTP para interceptar todas as URLs acessadas e conferir suas permissões de acesso. Por isso, o filtro é aplicado com o url-pattern "/*".

```
<!-- Spring Security -->
   <filter>
         <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
   </filter>

     <filter-mapping>
         <filter-name>springSecurityFilterChain</filter-name>
         <url-pattern>/*</url-pattern>
         <dispatcher>FORWARD</dispatcher>
         <dispatcher>REQUEST</dispatcher>
     </filter-mapping>
```

A Figura9**:** Apresenta a configuração do Spring Security.

E na última parte temos a configuração do Prime Faces apresentado na Figura 10, utilizando o tema delta. Caso o desenvolvedor pretenda mudar o tema da aplicação, basta só trocar o nome "delta" na configuração, pelo nome do novo tema.

```
<context-param>
    <param-name>primefaces.THEME</param-name>
    <param-value>delta</param-value>
</context-param>
```

A Figura10**:** Apresenta a configuração do Prime Faces**.**

3.3 Configuração do Spring Security

As configurações do *Spring* são muitas extensas e se dividem em configurações gerais, do banco de dados e de segurança (PEREIRA, p.52-60).

As configurações do *Spring Security* apresentada na Figura 11, 12 e 13, são feitas no arquivo *applicationContext.xml* dentro do pacote WEB-INF, nesse arquivo criamos as regras de segurança da aplicação.

```
<http auto-config="true" access-denied-page="/acesso_negado.xhtml">
          <intercept-url pattern="/sistema/**" access="ROLE_ADMIN"/>
          <intercept-url pattern="/admin/**" access="ROLE_SUPER"/>

          <form-login login-page="/index.xhtml" default-target-
url="/sistema/index.xhtml" authentication-failure-url="/login_falhou.xhtml" />
          <http-basic/>
          <logout invalidate-session="true" logout-success-url="/index.xhtml"/>
     </http>
```

A Figura11: Configuração do Spring Security-applicationContext.xml**.**

A tag <*http*> é a tag root de todas as configurações de seguranças da aplicação, seu atributo *auto-config* marcado com *true* configura o framework para utilizar autenticação HTTP-Basic, cria um formulário de login de autenticação padrão para configurar o *logout* (PEREIRA, p.52-60).

A tag <*intercept-url*> é usada para determinar o conjunto de padrões de URL que a aplicação está interessada em controlar o acesso e a respectiva restrição. O atributo *pattern* descreve o caminho que será feito o controle e o atributo *access* define a restrição. Desta forma todas as páginas que estive dentro da pasta sistema só serão acessadas por usuário autenticado (PEREIRA, p.52-60).

O objetivo da tag <*form-login*> é configurar o caminho da página de login (atributo *login-page*), bem como a página que o usuário será redirecionado caso a login falhe (atributo *authentication-failure-url*) (PEREIRA, p.52-60).

Ainda no mesmo arquivo de configuração iremos mapear as informações do banco de dados: usuário, senha, url e driver para que o Spring crie e gerencie o *dataSource*.

```
<b:bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <b:property name="url" value="jdbc:mysql://localhost:3306/projeto" />
        <b:property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <b:property name="username" value="root" />
        <b:property name="password" value="" />
</b:bean>
```

A Figura12: Configuração do Spring Security-applicationContext.xml.

Por fim, na tag <*authentication-manager*> definimos que o *authentication-provider* que o *Spring Security* irá executar vai ser o *dataSource*.

```
<authentication-manager>
    <authentication-provider>
        <jdbc-user-service data-source-ref="dataSource"
         users-by-username-query="SELECT login AS username, senha AS pas-
sword, 'true' as enable FROM usuarios WHERE login=?"
          authorities-by-username-query="SELECT login AS username, permis-
sao AS authority FROM usuarios WHERE login=?" />
    </authentication-provider>
</authentication-manager>
```

A Figura13: Configuração do Spring Security-applicationContext.xml.

3.4 Configuração do Hibernate

A configuração do hibernate apresentado na Figura 14, foi realizado em uma arquivo dentro do pacote de Códigos-fonte no pacote <pacote default>, esse arquivo é o *hibernate.cfg.xml*. Nesse arquivo podem ser descritas várias unidades de persistência, diferenciadas pelo nome, e cada qual com as seguintes propriedades: driver de banco de dados, usuário e senha de acesso, url de acesso ao banco e dialeto utilizado. Existe um dialeto especifico para cada banco de dados (PRIMO; SANTOS, p.46-57).

As configurações aqui estão divididas em elementos <*property*/>, onde cada um contém um atributo chamado *name*, indicando o que ele faz.

```
<?xml version="1.0" encoding="UTF-8"?>
 2 <!DOCTYPE hibernate-configuration PUBLIC
 "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
 "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>

    <session-factory>
<!-- O dialeto usado pelo Hibernate para conversar com o banco -->
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
<!-- Configuração do driver do banco de dados -->
   <propertyname="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
<!-- A URL de conexão ao banco de dados -->
<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/projeto</property>
<!-- Nome do Usuário -->
        <property name="hibernate.connection.username">root</property>
<!-- Tabelas são geradas automaticamente pelo hibernate -->
        <property name="hibernate.hbm2ddl.auto">update</property>
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>
        <mapping class="com.sysprefeitura.model.entidades.Usuario"/>
        <mapping class="com.sysprefeitura.model.entidades.Documento"/>
        <mapping class="com.sysprefeitura.model.entidades.Secretaria"/>
    </session-factory>
</hibernate-configuration>
```

Figura14**:** Arquivo hibernate.cfg.xml

Em sequência temos cada valor do atributo *name* listado por ordem de aparição na configuração do Hibernate.

- **Hibernate.dialect**: o dialeto no qual o Hibernate deverá utilizar para se comunicar com o banco de dados. Para configuração do hibernate foi utilizado o dialeto **org.hibernate.dialect.MySQLDialect,** pois na aplicação foi utilizado o banco de dados MySQL, e o valor que define a forma de geração foi o ***update*** (GONÇALVES, 2007).

- **Hibernate.connection.driver_class**: nome da classe do drive JDBC do banco de dados que está sendo utilizado. No caso a configuração está utilizando o driver do MySQL (GONÇALVES, 2007).
- **Hibernate.connection.url**: a URL de conexão específica do banco de dados que está sendo utilizado (GONÇALVES, 2007).
- **Hibernate.connection.username**: o nome de usuário com o qual o Hibernate deve se conectar ao banco de dados. No caso, pela configuração é root (GONÇALVES, 2007).
- **Hibernate.hbm2ddl.auto**: Esta propriedade define que as tabelas do banco de dados serão geradas automaticamente pelo Hibernate. Existem dois valores que define a forma de geração: o valor ***create-drop*** informa que o banco é sempre recriado na inicialização e o valor ***update*** denota que o banco é atualizado apenas se houver alteração no modelo, ou seja, após a inclusão de novos mapeamentos ou modificação em mapeamentos existentes (PRIMO; SANTOS, p.46-57).

Depois de ser criada a classe usuário para geração da tabela, e necessário mapear a mesma, adicionando na configuração do hibernate em < *mapping*> seguindo do nome *class*, o caminho exato da classe que será mapeada.

3.5 Anotações

O objetivo das anotações é possibilitar a declaração de metadados nos nossos objetos, isto é, configurações de uma classe podem ficar dentro dela, em vez de em um XML.

Toda classe Java que será persistida em um banco de dados através do Hibernate é considerada uma entidade e é declarada utilizando a anotação *@Entity* (acima da declaração da classe) (PRIMO; SANTOS, p.46-57). Na Figura15, temos um exemplo da declaração da anotação *@Entity*.

```
@Entity
@Table(name="usuarios")
public class Usuario implements Serializable{
```

Figura15**:** Declaração da anotação @Entity

A anotação *@Table* é definida em nível de classe e permite que você descreva o nome da tabela que será utilizada para o mapeamento da entidade. Se nenhum *@Table* for declara-

da, os valores padrões serão utilizados: no caso o próprio nome da classe (PRIMO; SANTOS, p.46-57).

O *@Table* elemento também contém um esquema e um catálogo de atributos, e eles precisam ser definidos. Você também pode definir restrições de unicidade para a tabela usando o *@UniqueConstraint* anotação em conjunto com *@Table* (PRIMO; SANTOS, p.46-57).

A anotação *@Id* permite que o usuário defina qual propriedade é a chave primária da sua entidade. A propriedade pode ser preenchida pela própria aplicação ou ser gerada pelo Hibernate. É possível definir a estratégia para geração graças à anotação *@GeneratedValue* (PRIMO; SANTOS, p.46-57). Na Figura16, temos a declaração da anotação *@Entity* e *@GeneratedValue.*

```
@Id
@GeneratedValue
private Long id;   // A id será gerada pelo banco
```

Figura16: Declaração da anotação @Entity e @GeneratedValue.

São quatro tipos possíveis de anotação *@GeneratedValue***.**

1) TABLE – utiliza uma tabela com a informação dos últimos Ids para geração dos próximos;
2) IDENTITY – utilizado quando você possui bancos que não suportam *sequence*, mas suportam colunas identidade;
3) SEQUENCE – utilizando uma sequence para gerar a chave primária;
4) AUTO – utiliza uma das estratégias acima de acordo com o banco de dados.

Associação Um-para-Um (OneToOne)

É possível associar entidades através de um relacionamento um-para-um usando a anotação *@OneToOne* apresentado na Figura 17. Existem três casos para utilização deste tipo de relacionamento: ou as entidades associadas compartilham os mesmos valores da chave primaria; ou uma chave estrangeira é armazenada por uma das entidades ou uma tabela associativa é usada para armazenar o link entre duas entidades (PRIMO; SANTOS, p.46-57).

```
@OneToOne
private Secretaria secretaria;
private String permissao;
private String nome;
```

Figura 17: Declaração da anotação *@OneToOne*.

A anotação *@Column* permite mapear as colunas de uma tabela. No código abaixo, Figura 18, está sendo mapeada a coluna login e senha.

```
@Column (unique = true)
private String login;
private String senha;
```

Figura 18: Declaração da anotação @Column.

3.6 Construção da tabela Usuarios

A primeira tabela a ser criada foi a de usuarios, para autenticação e autorização no banco de dados com *Spring Security*. O primeiro passo foi á criação do pacote "com.sysprefeitura.model.entidade" no projeto sysPrefeitura, como foi explicado anteriormente. Nesse pacote foi criada a Classe Usuario, essa será mapeada com anotações e usará o Hibernate para manipulação dos objetos em banco de dados. A tabela usuário é responsável por guarda o id, login, senha, nome, permissao e secretaria_id. A Figura 19 apresenta a criação da classe usuário para ser persistida.

```
package com.sysprefeitura.model.entidades;

import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.Table;
 /* *A anotação @Entity indica que a classe é uma classe persistente e a anotação @Table(name="usuarios") denota a tabela onde os dados serão persistidos. **/

@Entity
@Table(name="usuarios")
 public class Usuario implements Serializable{

/* *A anotação @Id define qual propriedade é a chave primária da entidade * */
@Id
@GeneratedValue   /* * A anotação @GeneratedValue Define a estratégia para a geração de Id, nesse caso será gerada automaticamente pelo banco de dado. * */

/* Para realizar o mapeamento e necessário que para cada campo da tabela seja criada uma variável do tipo correspondente */

 private Long id;

@OneToOne        /* Associa entidades através de um relacionamento um-para-um. */
```

```
private Secretaria secretaria;
private String permissao;
private String nome;

@Column (unique = true) /* Irá mapear as colunas login e senha da tabela usuarios. */
private String login;
private String senha;
 /* O mapeamento exige que para cada variável criada existam os métodos get e set */
```

Figura 19**:** Classe Usuario.java

3.7 UsuarioDAO

Para adicionar ao sistema comportamentos de gravar um cadastro foi adicionado ao pacote "com.sysprefeitura.model.daos", a Classe UsuarioDAO, que irá fazer acesso aos dados no banco.

Foi criado nessa classe o método, gravar (), apresentado na Figura 20, tem a incumbência de armazenar novos cadastros no banco de dados.

```
@RequestScoped
public class UsuarioDAO implements Serializable {

    public boolean gravar(Usuario u) {
        Session session = HibernateUtil.getSessionFactory().openSession();
        boolean res = false;
        try {
            session.beginTransaction();
            session.merge(u);
            session.getTransaction().commit();
            res = true;
        } catch (Exception e) {
            e.printStackTrace();
            session.getTransaction().rollback();
        } finally {
            session.close();
        }
        return res;
    }
```

Figura 20. Método gravar.

A classe HibernateUtil traz a sessão através do método getSessionFactory(), iniciando assim uma sessão, através do objeto *Session* criado nessa classe. Esse método utiliza um objeto do tipo *Transaction*. Quando utilizado em conjunto com um objeto *Session*, permite transações em um banco de dado relacional. Para gravar os dados vindo do objeto Usuario, o método *merge ()* do objeto *Session* é utilizado. A transação é efetuada através do método *commit ()*. Em caso de erro, a cláusula *catch* chama o método *rollback ()* para desfazer a transação. Depois de utilizado, o objeto Session é fechado através da cláusula *finally*, através do método *close ()*.

3.8 Spring Security

Autenticação

A autenticação é a verificação das credenciais (nome de usuário e senha) da tentativa de conexão. Esse processo consiste no envio de credenciais do cliente para o servidor em um formulário de texto simples ou criptografado usando um protocolo de autenticação. O Spring Security possui várias formas de realizar a autenticação. Por exemplo, podemos utilizar o formulário de login que é gerado automaticamente pelo framework ou contruir um próprio formulário personalizado. Os usuários da aplicação que serão autenticados podem ser definidos diretamente no arquivo XML de configurações do framework ou em um banco de dados (PEREIRA, p.52-60).

Autorização

Autorização é utilizada para verificar se determinado usuário previamente autenticado possui permissão para usar, manipular ou executar o recurso em questão. É a concessão de uso para determinados tipos de serviço, dada a um usuário previamente autenticado(PEREIRA, p.52-60).

O Spring Security possui uma abordagem declarativa para a segurança, baseada em roles (papéis). Por exemplo, uma aplicação de uma pousada poderia ter dois roles: um ADMIN, que possui permissão para cadastrar acomodações e reservá-las, e um COMUM, que possui permissão apenas para reservá-las. Dessa forma os usuários que forem utilizar essa aplicação teriam que possuir algum desses roles (PEREIRA, p.52-60).

O projeto sysPrefeitura possuir dois roles: um ROLE_SUPER, destinado ao administrador do sistema, que possuir permissão para cadastrar e excluir usuários e para cadastrar e excluir setores, e um ROLE_ADMIN, destinado ao usuário, que possuir permissão para enviar, excluir e baixar documentos oficiais.

OBS: Para dar continuação ao projeto foi primeiro necessário criar a tabela usuários com hibernate para autenticação e autorização no banco de dados utilizando Spring Security, como foi mostrado acima. A tabela usuário é responsável por guarda o id, login, senha, nome, permissao e secretaria_id.

3.9 Página de Login (index.xhtml)

A Figura 21 apresenta o formulário da página index.html de login, onde temos os campos de login, senha e um botão "Acesse sua Conta" que redireciona a aplicação para uma área restrita (*/sistema/index.xhtml*), caso senha e login estiverem corretos. O formulário ainda contém o atributo *required* com o valor *true* para que o mesmo passe a ser obrigatório em seu preenchimento. Para que uma mensagem apareça, caso o usuário não preencha um campo, está sendo utilizado o atributo *requiredMessage*, que receber a mensagem que será mostrada ao usuário. O atributo *value* está sendo usado para dar o nome ao campo, no caso e-mail e senha. O atributo *action* está definindo o local para onde serão enviados os dados do formulário.

```
<ui:define name="conteudo">
        <h:form id="form" prependId="false">
            <h:panelGrid columns="1" cellpadding="3">

                <p:inputText id="j_username" required="true"
                 styleClass="campo_form" requiredMessage="Informe seu e-mail!"
                 validatorMessage="E-mail inválido!">

                <f:validateRegex
                pattern="^[_A-Za-z0-9-\+]+(\.[_A-Za-z0-9-]+)*@[A-Za-z0-9-]+(\.[A-
Za-z0-9]+)*(\.[A-Za-z]{2,})$" />
                </p:inputText>
                <p:watermark for="j_username" value="E-mail" />
                <p:password id="j_password" required="true"
                   styleClass="campo_form" requiredMessage="Informe sua senha!" />
                <p:watermark for="j_password" value="Senha" />
            </h:panelGrid>

                <p:commandButton styleClass="botao" value="Acesse sua conta"
                  action="#{segurancaBean.doLogin}" ajax="false" update=":alerta" />
        </h:form>
    </ui:define>
</ui:composition>
```

Figura 21**:** Formulário de login (index.xhtml)

Entendendo o processo de autenticação

A página *index.xhtml*, mostrada na Figura 21, contém o formulário com os campos *j_username* e *j_password* que armazenarão as credenciais do usuário. Esses dados serão sub-

metidos para a classe SegurancaBean mostrado na Figura 22, quando o usuário clicar no botão "**Acesse sua conta".**

Essa classe é responsável por receber o pedido de autenticação e verificar se as credenciais do usuário são válidas. Para realizar esse processo ela invoca o método doLogin () dentro da classe SegurancaBean.

```
@Named
@RequestScoped
public class SegurancaBean implements Serializable {

        public String doLogin() throws IOException, ServletException
    {
        ExternalContext context = FacesContext.getCurrentInstance()
                .getExternalContext();
        RequestDispatcher dispatcher = ((ServletRequest) context.getRequest())
                 .getRequestDispatcher("/j_spring_security_check");
        dispatcher.forward((ServletRequest) context.getRequest(),
                (ServletResponse) context.getResponse());

        FacesContext.getCurrentInstance().responseComplete();

        return null;
    }
}
```

Figura 22. Classe SegurancaBean

Caso não seja encontrado o usuário no banco de dados, o framework interrompe o fluxo de autenticação e redireciona o usuário para a página de usuário/senha incorreto. Se existir um usuário com o *j_username* e *j_password* informado o usuário terá acesso ao sistema, ou seja, a página *index.xhtml* dentro da pasta sistema. A Figura 23 mostra a página de login.

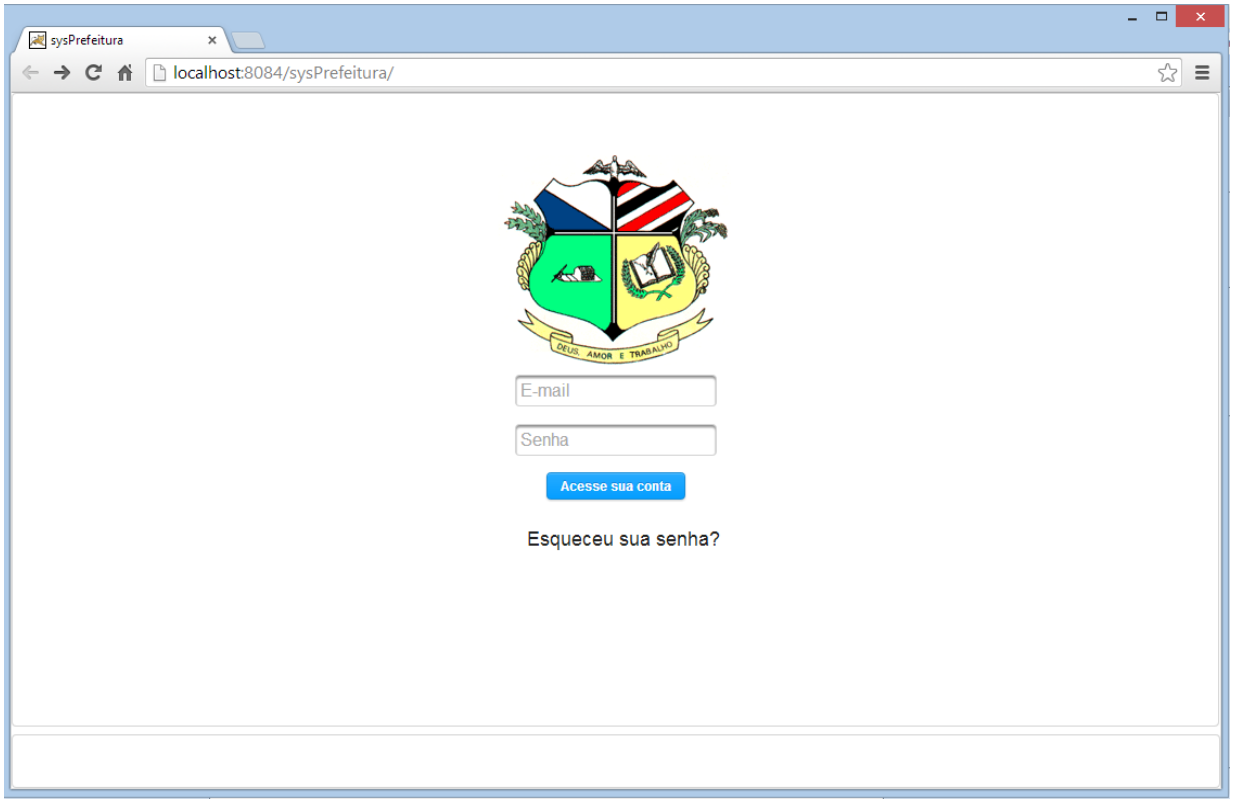


Figura 23. Página de Login (index.xhtml)

Deste modo foi concluída as configurações estruturais da aplicação que consistiram em criar o projeto, configurar os arquivos XML para integrar o *Spring*, *Spring Security*, *JSF* e *PrimeFaces*, configuração do hibernate, criação da tabela usuário no banco de dados para autenticação e autorização no sistema utilizando *Spring Security* e a criação do formulário da página de login, bem como a criação das classes UsuarioDAO e SegurancaBean.

3.10 CDI

A CDI é uma especificação Java, cujo nome completo é "*Context and Depen- dency Injection for Java EE"* (CORDEIRO, p. 1-208).

Para configurar o CDI é necessário copiar todos os arquivos *weld-servlet.jar* que foram baixados para a pasta do projeto, no caso foi copiado para sysprefeitura/lib/Weld_CDI. A Figura 24 abaixo apresenta todos os arquivos *Weld* que foi adicionado no projeto.

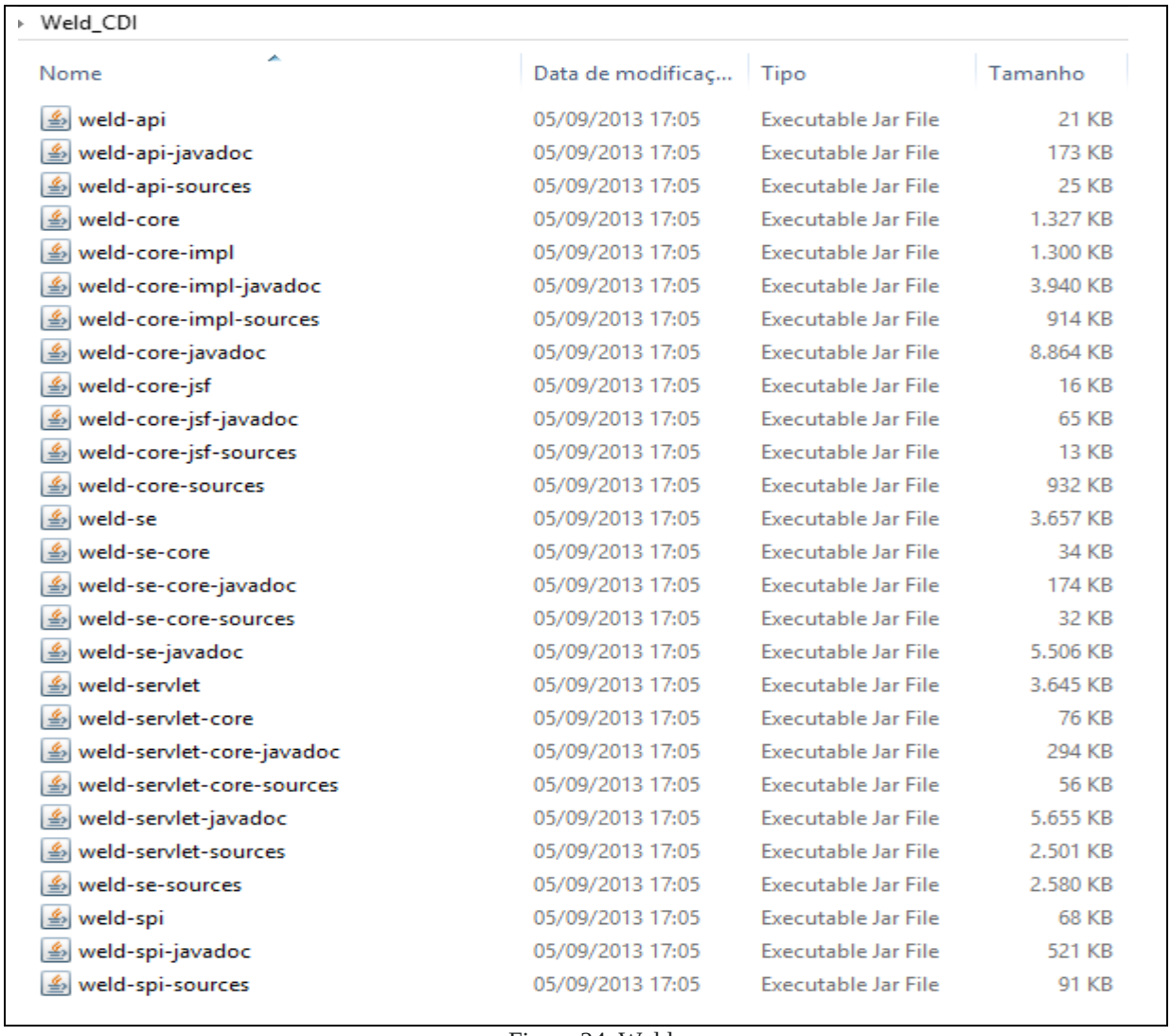


Figura 24. Weld

Segundo Cordeiro (2013), “Um pacote CDI São aquelas classes utilitárias que nós já temos prontas dentro de um *jar*, em que podemos injetar instância delas dentro da nossa aplicação colocando um arquivo chamado *beans.xml* dentro da pasta WEB-INF da nossa aplicação”. A Figura 25 apresenta a configuração do arquivo *beans.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
</beans>
```

Figura 25: Arquivo beans.xml

Arquivo de configuração do Weld no Tomcat no arquivo Context.xml na Figura 26

```
<?xml version="1.0" encoding="UTF-8"?>
<Context antiJARLocking="true" path="/sysPrefeitura">

 <Resource name="BeanManager"
 auth="Container"
 type="javax.enterprise.inject.spi.BeanManager"
 factory="org.jboss.weld.resources.ManagerObjectFactory"/>

</Context>
```

Figura 26: Configuração do CDI no arquivo Context.xml

Em seguida é necessário adicionar no arquivo *web.xml* o *listener* do *Weld.* A Figura 27 apresenta o arquivo *web.xml* com o *listener do Weld.*

```
<listener>
<listener-class>org.jboss.weld.environment.servlet.Listener</listener-class>
</listener>
```

Figura 27: Configuração do CDI no arquivo web.xml

A CDI é uma API para injeção de dependências e contextos. Em *Seam* e *Spring*, as dependências funcionam, principalmente, nomeando *beans* e os vinculando aos seus pontos de injeção pelos nomes. A atribuição principal da anotação *@Named* é definir o *bean* com o objetivo de resolver instruções EL na aplicação, normalmente por meio dos resolvedores JSF EL (CORDEIRO, p. 1-208).

Escopo de requisição com @RequestScoped O escopo request

É o mais comum em uma aplicação *web*, pois toda interação entre o *browser* e o servidor é um *request*. Esse escopo define um ciclo de vida que se inicia no momento em que a requisição do usuário chega no servidor, e dura todo o processamento e também todo o processo de geração da resposta para o usuário. Somente quando a resposta para o usuário é terminada que o *request* termina. Parece simples, mas é preciso ficarmos atentos principalmente a essa última parte: a geração da resposta para o usuário. Não adianta acompanharmos a execução passo a passo dentro da lógica da aplicação e achar que está tudo certo, pois se algo ocorrer depois do processamento, mas antes da geração da resposta terminar, ainda podemos ter problema, pois o ciclo como um todo é que importa (CORDEIRO, p. 1-208). Uma forma de tentarmos ilustrar o funcionamento desse escopo é através da Figura 28 a seguir:

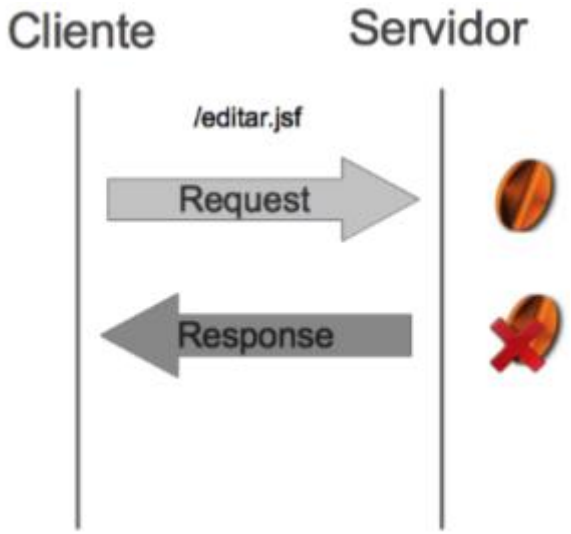


Figura 28: Ilustração do Escopo Request

A Figura 28 exemplificamos a chamada da URL /editar.jsf. Onde o *bean* é criado no momento em que a requisição chega no servidor, e termina no momento em que sai dele (CORDEIRO, p. 1-208).

Escopo de sessão como @SessionScoped

O processo de criação e descarte de um *bean* não muda se o escopo é *request* ou *session*, o que muda é quando o processo ocorre. Enquanto o *request* compreende cada invocação ao servidor feita pelo cliente, a sessão engloba todas as requisições de um mesmo cliente. Para facilitar, é só imaginar o processo de autenticação em uma aplicação web. Enquanto tivermos com o browser aberto, estaremos logados na aplicação, isso porque as informações de autenticação costumam ficar no escopo de sessão. Cada vez que acessamos uma aplicação pela primeira vez, comum determinado navegador, é criada uma nova sessão. Mas isso não cria automaticamente os *beans* de escopo sessão, estes geralmente são criados quando utilizados pela primeira vez, e duram enquanto a sessão existir. Enquanto conseguimos perceber pelo console que, no escopo *request*, os objetos são recriados a cada solicitação; como escopo sessão, só perceberemos essa criação uma vez para cada browser utilizado. Geralmente a sessão é mantida através de um *cookie* ou então algum parâmetro na url que é gerado pelo servidor, assim conseguimos abrir mais de uma sessão se usarmos *browsers* diferente sou a chamada sessão anônima suportada pelos navegadores mais modernos. Para testar a criação sessões, é possível fechar o *browser* e abrir novamente pra criar uma nova sessão, mas isso não significa que a antiga será fechada. Geralmente o que ocorre quando um usuário fecha o *browser* e usar alguma opção "sair" da aplicação, é que a sessão ficará inativa no servidor, ocupando recursos desnecessariamente, até que o time out seja alcançado. Isso geralmente ocorre após trinta mi-

nutos de inatividade (CORDEIRO, p. 1-208). A seguir a Figura 29 ilustrar esse funcionamento.

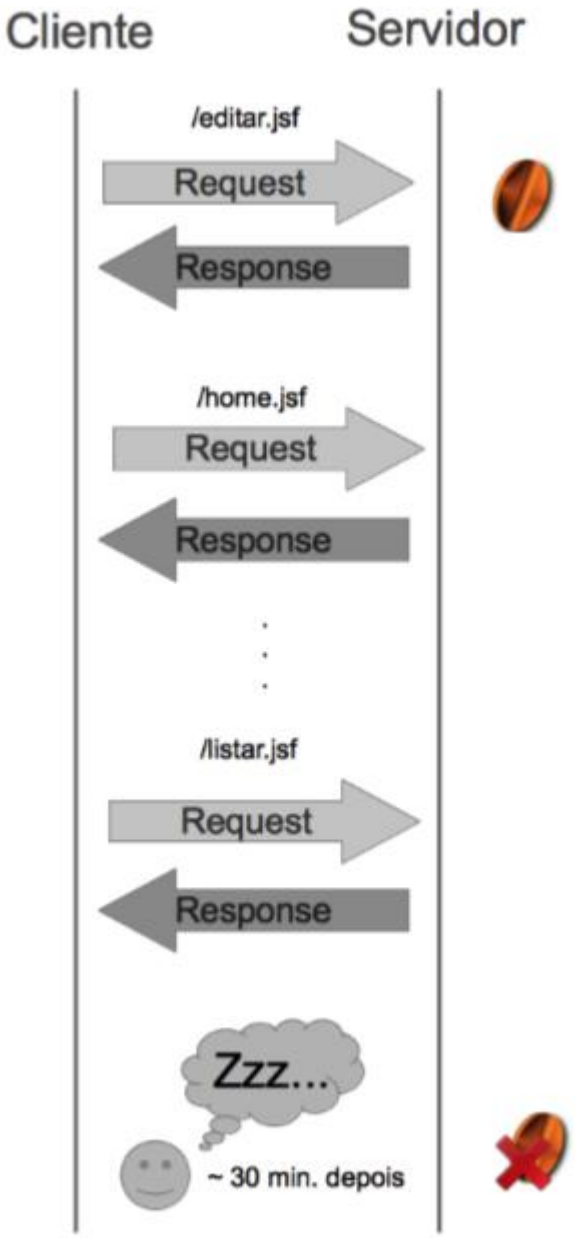


Figura 29: Session

O escopo de sessão serve para armazenar informações que devem estar em memória durante toda a navegação do usuário na aplicação. Outros exemplos podem ser as permissões do usuário, informações de perfil de acesso, nome, hora do login, entre outras que possam ser interessantes dependendo da aplicação (CORDEIRO, p. 1-208).

Anotação @Inject e @PostContruct

A anotação *@Inject* é usada para marcar o local onde será feita a injeção. Para solicitar a injeção de uma dependência basta utilizarmos a anotação *@javax.inject.Inject*. Dessa forma a CDI procura em seu contexto uma classe candidata a suprir essa dependência, e como por padrão toda classe dentro de um pacote CDI é candidata a ser injetada, não precisamos fazer nenhuma configuração nas classes (CORDEIRO, p. 1-208).

No código abaixo Figura 30 está sendo apresentada a classe CadastroBean sem injeção, onde o método salvar apresenta uma instancia da classe UsuarioDAO.

```
@Named
@RequestScoped
public class CadastroBean implements Serializable {

    private Usuario usuario = new Usuario();

    public String salvar(){
        UsuarioDAO dao = new UsuarioDAO();
        if(dao.gravar(usuario)){
            FacesUtils.showFacesMessage("Usuário cadastrado com sucesso!", 2);
            return "index";
        }
        FacesUtils.showFacesMessage("Erro ao tentar cadastrar o usuário!", 1);
        return null;
    }
```

Figura 30: Sem injeção de Dependência

Já no código abaixo na Figura 31 está usando injeção, onde está injetando a classe UsuarioDAO na classe CadastroBean pela anotação *@Inject*, observa-se que no método salvar não precisa mais instanciar a classe UsuarioDAO, como foi instanciado no código acima.

```
@Named
@RequestScoped
public class CadastroBean implements Serializable {

 @Inject
 UsuarioDAO dao;

 private Usuario usuario = new Usuario();

 public String salvar(){
     if(dao.gravar(usuario)){
 FacesUtils.showFacesMessage("Usuário cadastrado com sucesso!", 2);
          return "index";
     }
 FacesUtils.showFacesMessage("Erro ao tentar cadastrar o usuário!", 1);
     return null;
 }
```

Figura 31: Com Injeção de Dependência.

A anotação *@PostConstruct* apresentada na Figura 32, é usada para que nossos beans saibam quando estão prontos ,com todas as dependências satisfeitas, e assim possam fazer algo.

```
@Named(value = "sessionBean")
@SessionScoped
public class SessionBean implements Serializable {

    @Inject
    UsuarioDAO dao;

    Usuario usuario;

    @PostConstruct
    private void init() {
        this.usuario = new Usuario();

        String login = SecurityContextHolder.getContext().getAuthentication()
                .getName();
        this.usuario = dao.buscarPorLogin(login);


    }
```

Figura 32: Anotação @PostConstruct

Para dar continuidade ao desenvolvimento do sistema, no pacote com.sysprefeitura.model.daos foi adicionado mais uma classe, a classe DocumentoDAO. E no pacote sysprefeitura.model.entidades foi adicionado mais uma entidade, a classe Documento será responsável por gerar a tabela documentos no banco de dados e no pacote sysprefeitura.Beans foi criado as classes DocumentoBean e documentoDownloadBean. Essas classes serão acessadas pela página index.xhtml, que está em uma área restrita dentro da pasta sistema. A mesma só poderá ser acessada por usuários que estiverem logados no sistema. As listagens abaixo apresenta o formulário da página index, onde foi separada em partes diferentes por comentários para melhor entendimento.

A primeira parte apresenta a tag do PrimeFaces <p:dataTable/>, que criar uma tabela na página e recebe todos os documentos pertencentes ao usuário logado no sistema através do método listaDocumentos(), da classe documentoBean, no atributo *value*. A Figura 33 apresenta a criação da tabela e na Figura 34 esta sendo apresentado seu método.

```
<p:dataTable id="dataTable" var="d"
value="#{documentoBean.listaDocumentos}"
    paginator="true" rows="10"
    paginatorTemplate="{CurrentPageReport}  {FirstPageLink}
    {PreviousPageLink} {PageLinks} {NextPageLink}
    {LastPageLink} {RowsPerPageDropdown}"
    rowsPerPageTemplate="5,10,15">
```

Figura 33. Criação da tabela.

```
public List<Documento> getListaDocumentos() {

    listaDocumentos = dao.listarTodosSecretaria(sb.getUsuario());
    return listaDocumentos;
}

public void setListaDocumentos(List<Documento> listaDocumentos) {
    this.listaDocumentos = listaDocumentos;
}
```

Figura 34. Método da tabela.

Na mesma possui uma coluna para a data do documento que é criada através da tag <p:columm headerText> que recebe o nome "Data" como o nome da coluna. A tag <h:outputText value="#{d.data}"> acessa a classe Documento no pacote syspefei-tua.model.entidades e receber a data atual e em seguida na tag <f:convertDateTime> a data e convertida e mostrada em dia, mês e ano. A Figura 35 abaixo mostra a criação da coluna Data e na Figura 36 mostra o código da classe Documento que é responsável por retorna a data.

```
<p:column headerText="Data" >
    <center>
        <h:outputText value="#{d.data}" >
            <f:convertDateTime pattern="dd/MM/yyyy" />
        </h:outputText>
    </center>
</p:column>
```

Figura 35. Criação da coluna Data.

```
@Temporal(TemporalType.DATE)
private Date data;

public Date getData() {
    return data;
}

public void setData(Date data) {
    this.data = data;
}
```

Figura 36. Código da classe Documento responsável por retorna á data.

As anotações que contém @Temporal são tipos baseados em informações armazena-das relativas ao tempo. O uso dessa anotação inclui um atributo chamado de TemporalType com um valor enumerado. Esses valores são: DATE, TIME e TIMESTAMP para representar os tipos de Java.sql (GONÇALVES, 2007). Nesse caso foi utilizado o valor DATE.

A tag <p:column> , mostrado na Figura 37 criar a coluna Assunto e receber o nome do assunto do documento digitado pelo usuário que estão armazenados na tabela documentos.

```
<p:column headerText="Assunto">
    <h:outputText value="#{d.assunto}" />
</p:column>
```

Figura 37. Criação da coluna Assunto.

Em seguida e criada mais uma coluna que recebe o nome de todas as secretarias que receberam os documentos. A tag <h:outputText> tem um atributo *value* que recebe a classe documentoBean que busca o método secretariasNome, esse método recebe o nome da secretaria para onde o documento foi enviado, caso for enviado pra mais de uma secretaria os nomes das mesma será separado por uma virgula, porém uma virgula era sempre inserida no final do nome da ultima secretaria. Para resolver esse problema foi utilizado um return, que conta toda a String e apaga dois caracteres do fim da String, e em seguida foi adicionado um ponto no fim da String. Figura 38 criação da coluna destinatário e Figura 39 seu método.

```
<p:column headerText="Destinatário">
<h:outputText value="#{documentoBean.secretariasNome(d.secretarias)}" />
</p:column>
```

Figura 38. Criação da coluna Destinatário.

```
public String secretariasNome(List<Secretaria> list) {
    String texto = "";
    for (Secretaria s : list) {
        texto += s.getNome() + ", ";
    }
    return texto.substring(0, texto.length() - 2) + ".";
}
```

Figura 39. Método secretariaNome.

Na coluna Documentos, para cada documento listado é criado um botão <p: commandButton> que receber o valor “baixar” que chama a ação selecionarDocumento(d) na classe documentoBean. A Figura 40 mostra a criação da coluna na tabela e na Figura 41 mostra o código correspondente ao método selecionarDocumento(d).

```
<p:column headerText="Documentos">
<p:commandButton value="baixar"
actionListener="#{documentoBean.selecionarDocumento(d)}" ajax="false">
            <p:fileDownload value="#{documentoDownloadBean.file}" />
</p:commandButton>
</p:column>
```

Figura 40. Criação da Coluna Documentos

```
public void selecionarDocumento(Documento d) {
    doc.setDocumento(d);
}

public Documento getDocumento() {
    return documento;
}
```

Figura 41. Método selecionar

Em seguida na Figura 42 temos a classe documentoDownloadBean e o método file.

```
public StreamedContent getFile() {
    return file;
}

public void setFile(StreamedContent file) {
    this.file = file;
}
```

Figura 42: Classe DocumentoDownloadBean

3.11 Formulário para Manda Arquivo

Para manda um arquivo é necessário acessa a página documento_form.xhtml que contém o formulário com os campos para serem preenchidos.

A tag <h:outputLabel> receber o atributo value que coloca o nome "para" na página, em seguida na tag <p:selectManyCheckbox> mostrará todas as secretaria cadastradas no banco de dados através do método secretarias(). A tag <f:selectItems> permite que seja selecionada as secretarias para que seja enviado algum arquivo pra mesma. Figura 43 apresenta o formulário da página.

```
<h:form id="formCadDoc" enctype="multipart/form-data">

        <h:panelGrid columns="2" cellspacing="5">
            <h:outputLabel for="para" value="Para: " />
            <p:selectManyCheckbox      value="#{documentoBean.documento.secretarias}"
converter="secretariaConverter"
                        required="true" requiredMessage="Selecione  uma Secretaria!" >
              <f:selectItems var="sec" value="#{documentoBean.listaSecretarias}"
                          itemLabel="#{sec.nome}" itemValue="#{sec}" />
            </p:selectManyCheckbox>
```

Figura 43. Página do formúlario

Para que o usuário pudesse adicionar um nome para o documento que ele queira enviar foi adiciona no formulário um campo com o nome assunto que receberá o nome digitado pelo usuário. Em seguida foi criado um botão que Permite procurar um arquivo através do método

arquivo (). Figura 44 tem o formulário da página com a criação da coluna assunto e seu botão procura arquivo.

```
<h:outputLabel for="assunto" value="Assunto: " />
<p:inputText id="assunto" value="#{documentoBean.documento.assunto}"/>
<h:inputFile value="#{documentoBean.arquivo}" />
```

Figura 44. Página do formulário coluna assunto.

Para que os arquivos fosse salvos no banco de dados foi criado o método salvar(), da classe documentoBean. A Figura 45 abaixo apresenta o código no formulário que aciona a classe documentoBean e seu método salvar, e na Figura 46 apresenta o método que salva o arquivo no banco, esse método na hora que o usuário tenta enviar o arquivo ele faz uma comparação entre o arquivo escolhido pelo usuário com os tipos de arquivos que o sistema suporta, se o arquivo tiver extensão diferente de pdf, docx e doc o sistema mostrara uma mensagem de “Não é permitido esse tipo de arquivo”, caso contrario o arquivo será salvo com sucesso no banco de dados e uma mensagem será mostrada para o usuário de “arquivo enviado com sucesso”.

```
<p:commandButton styleClass="botao" value="Enviar"
        action="#{documentoBean.salvar()}" ajax="false" update=":alerta" />
```

Figura 45:Botão Enviar.

```
    public String salvar() {
        try {
            String tipo = arquivo.getContentType();
            if (tipo.equals("application/pdf")) {
                tipo = "pdf";
            } else if (tipo.equals("application/vnd.openxmlformats-
officedocument.wordprocessingml.document")) {
                tipo = "docx";
            } else if (tipo.equals("application/octet-stream")) {
                tipo = "doc";
            } else {
                FacesUtils.showFacesMessage("Não é permitido esse tipo de aquivo! "+tipo, 1);
                return null;    }
             documento.setTipo(tipo);
            InputStream is = arquivo.getInputStream();
                byte[] bytes = IOUtils.toByteArray(is);
            documento.setArquivo(bytes);
            documento.setData(new Date());
            if (dao.gravar(documento)) {
                documento = new Documento();
                FacesUtils.showFacesMessage("Arquivo enviado com sucesso!", 2);
                return "index";
            } else {
                FacesUtils.showFacesMessage("Erro ao tentar enviar o arquivo!", 1);
                return null;  }
        } catch (IOException e) {
                FacesUtils.showFacesMessage("Erro ao tentar enviar o arquivo!", 1);
            return null;       }  }
```

Figura 46. Método salvar.

É por fim, o sistema usará a classe FacesUtils presente no pacote com.sysprefeitura.util que contém um método showFacesMessage, responsável por mostra mensagem no sistema. Essa mensagem e apresentada através do switch case, ou seja, caso contrário. Para o caso 1 foi associado á mensagens de erro e ao caso 2 para mensagens de informação. O código acima mostra como essa classe trabalhar.

A Figura 47 apresenta a página que mostrará os documentos recebidos pelo usuário logado no sistema e na Figura 48 apresenta a página para o envio de documentos.

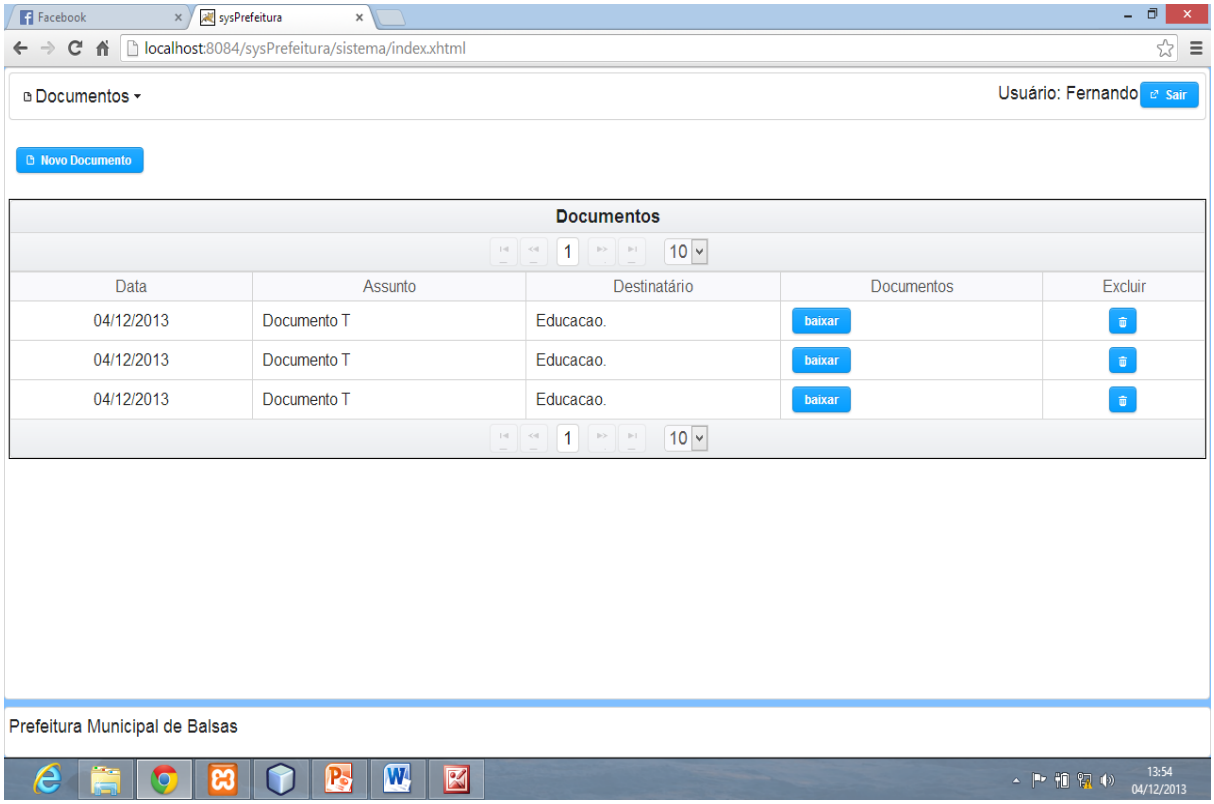


Figura 47: Página que mostrar os documentos

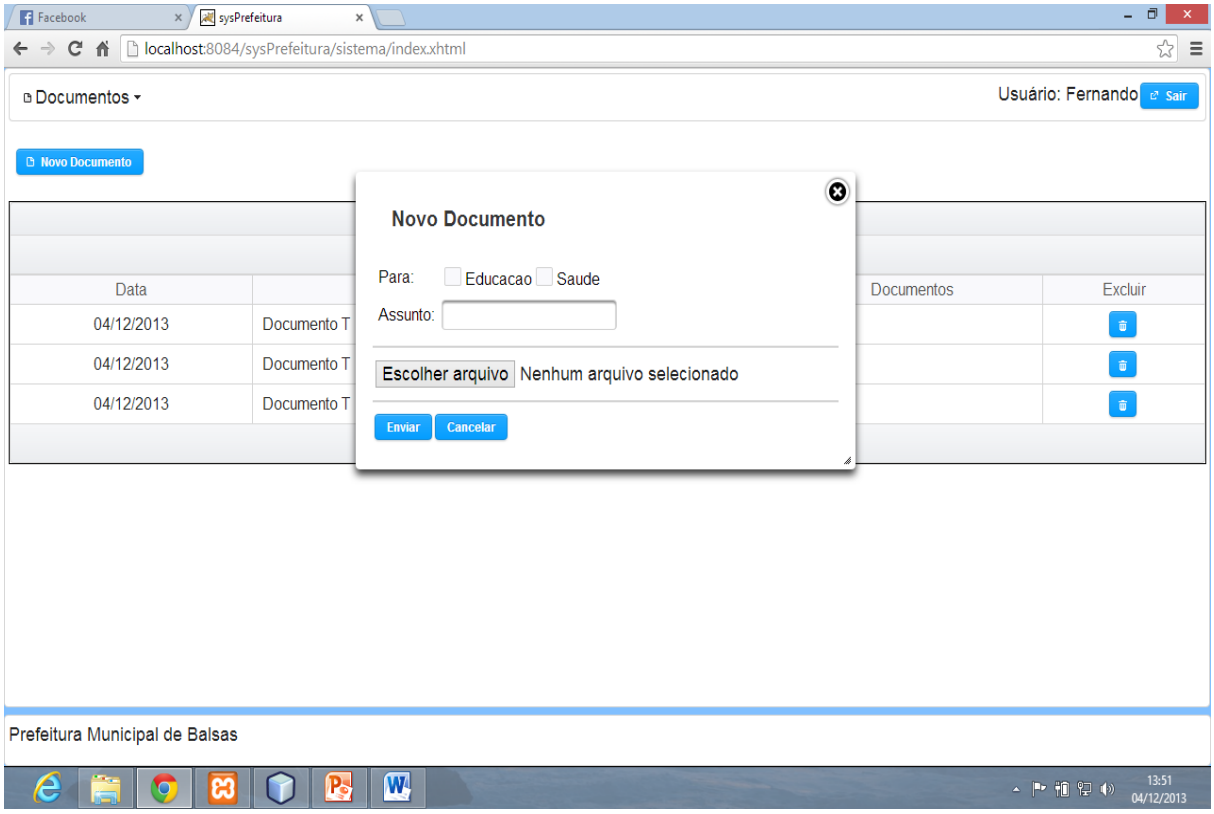


Figura 48: Página para enviar documento.

4. CONCLUSÃO

4.1 Considerações Finais

Com o sistema desenvolvido por este trabalho, espera-se que o mesmo quando implantado na prefeitura municipal de Balsas-MA consiga realizar de forma mais rápida a comunicação feita por troca de documentos oficiais.

Essa solução foi concebida com a realização dos estudos e análise feitos em diversos frameworks, softwares e padrões utilizados para a separação da aplicação, que possibilitou para o projeto uma solução viável para o seu desenvolvimento.

As tecnologias baseadas na Web permitira aos usuários a oportunidade de enviar, armazenar e receber documentos oficiais de qualquer partição da estrutura interna da mesma que esteja cadastrada no sistema, a partir de qualquer navegador Web.

O propósito deste trabalho foi apresentar as ferramentas utilizadas para o desenvolvimento do sistema proposto, bem com mostrar passo a passo o desenvolvimento do mesmo.

4.2 Trabalhos Futuro

Como trabalho futuro será implementado no sistema o MD5, que é um tipo de "impressão digital" de um arquivo, usado para determinar se um arquivo baixado de um FTP não se alterou ao chegar ao micro, utilizando-se de um checksum de 128 bits. O checksum é uma soma matemática baseadas no conteúdo dos dados em processamento, para verificar correção.

5 REFERÊNCIAS BIBLIOGRÁFICAS

BASHAM, B.; SIERRA K.; BATES B**. Head first servlets & jsp,** 2 Edition**.** O'REILLY.2008.

CORDEIRO, Gilliard. **CDI Integre as dependências e contextos do seu código Java**. Casa do Código, São Paulo, p. 1- 208.

FERREIRA, A. (Ed). Aurélio Júnior: português. Editora Positivo Ltda, 2005.

FILHO, W. **Engenharia de Software fundamentos, métodos e padrões.** LTC – Livros Técnicos e Científicos Editora S.A., 2005.

FOX, C**. Introduction to software engineering design: processes, principles, and patterns with uml 2.** 1 st ed. Boston: Pearson, 2006.

GONÇALVES, E. **Desenvolvendo Aplicações Web com JSP Servlets, JavaServer Faces, Hibernate, EJB3 Persistence e Ajax.** Editora Ciência Moderna Ltda., 2007.

GONÇALVES, E. **Dominando NetBeans.** Editora Ciência Moderna Ltda., 2006.

JAMACEDO 2013 Jamacedo. Disponível em: http://jamacedo.com/2011/01/crud-jsf-2-parte-3-seguna-com-spring-security-3/, Acessado em 04 de abril de 2013.

KITO, D. JAVASERVER FACES IN ACTION. Mann Foreword by Ed Burns: MANNING. 2005.

KURNIAWAN, B. Java para a Web com Servlet, JSP e EJB. Rio de Janeiro: Editora Ciência Moderna Ltda., 2002.

NUNES, Vinny; MAGALHÃES, Eder. Aprendendo JSF 2.0 com ScrumToys. **Java Magazine**, Rio de Janeiro, p.22-36.

PEREIRA, Tiago. Spring Security 3, JSF 2 e JPA 2. **Java Magazine,** Rio de Janeiro, p. 52-60.

PRIMO, Izalmo; SANTOS, Samuel. Desenvolvendo com Hibernate. **Java Magazine,** Fortaleza, p.46-57.

REZENDE, D. **Engenharia de software e sistemas de informação.** BRASPORT LIVROS E Multimídia Ltda., 2005.

SANTOS, R. **Java na Web.** Axcel Books do Brasil Editora Ltda., 2007.

SOMMERVILLE, I. **Engenharia de Software.** Pearson Education do Brasil., 2005.

SOUSA, Marcos. Desenvolvendo com JavaServer Faces - Parte 1. **Java Magazine**, Rio de Janeiro, p.8-18.

SOUSA, Marcos. Desenvolvendo com JavaServer Faces - Parte 2. **Java Magazine**, Rio de Janeiro, p.14-26.

ZANINI, Michel. Spring Security. **Java Magazine**, Rio de Janeiro, p.28-38.

APÊNDICE A

DOCUMENTOS DE REQUISITOS

SUMÁRIO

1 IDENTIFICAÇÃO DOS REQUISITOS

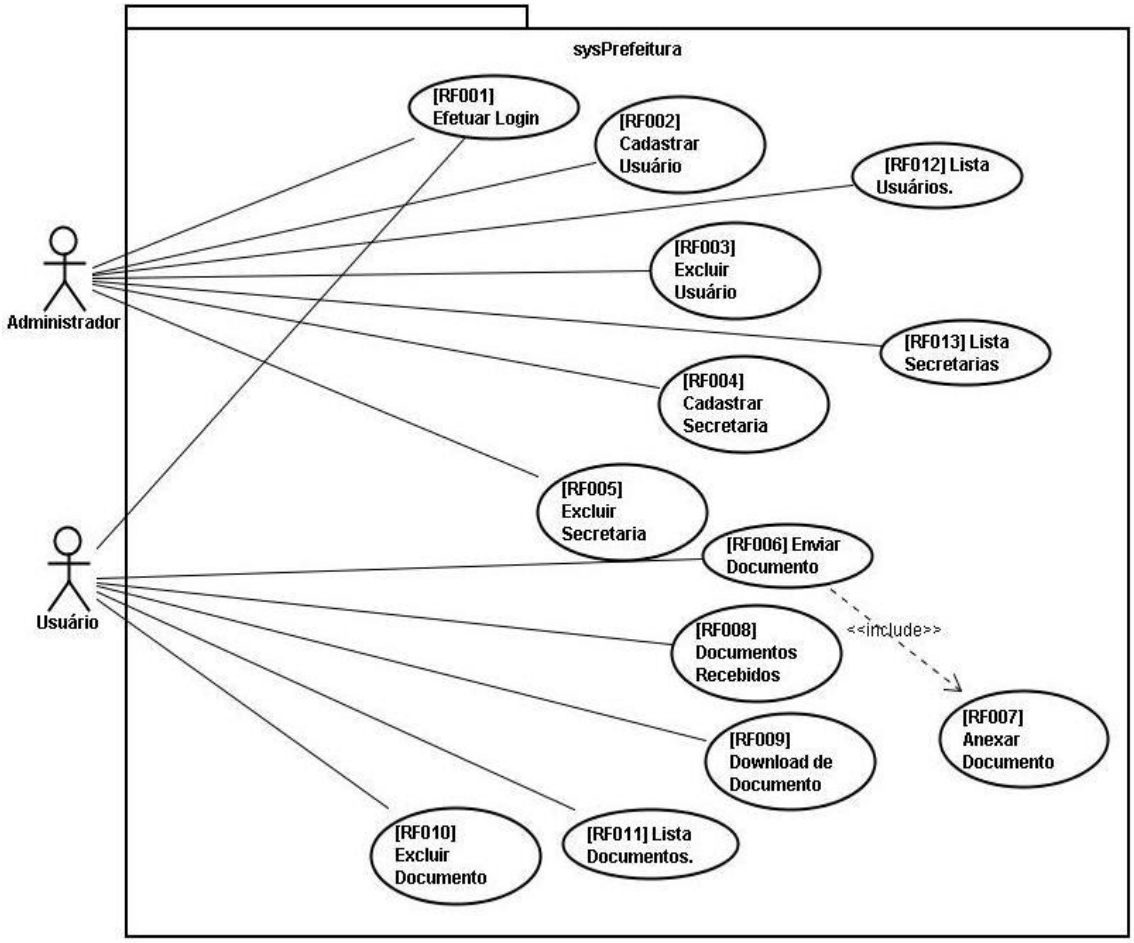
1.1 Finalidade

Tem por finalidade descrever e especificar os requisitos que devem ser atendidos pelo sistema de forma a atender as necessidades dos usuários, bem como definir o sistema a ser feito, para o desenvolvedor do mesmo.

1.2 Casos de Uso

[RF001] Login de Usuários.
[RF002] Cadastrar Usuário .
[RF003] Excluir Usuário.
[RF004] Cadastrar Secretaria.
[RF005] Excluir Secretaria.
[RF006] Enviar Documentos
[RF007] Anexar Documentos.
[RF008] Receber Documentos.
[RF009] Download de Documentos.
[RF010] Excluir Documentos.
[RF011] Lista Documentos.
[RF012] Lista Usuários.
[RF013] Lista Secretarias

1.3 Diagrama de Caso de Uso



1.4 Descrição dos Casos de Uso

[RF001] Efetuar Login

Ator: Administrado e usuário.

Descrição do caso de uso: Entrada: Este caso de uso permite que o usuário acesse o sistema.

Pré-condições: Somente o administrador do sistema e os usuários cadastrados terão acesso ao mesmo.

Processo: O sistema verificar se o administrado/usuário está cadastrado no sistema, se for terão acesso ao mesmo.

Saída: Mensagem de confirmação bem sucedida do Login efetuado com sucesso.

Pós-Condição: Não há pós-condição.

Fluxo Principal

1. O caso de uso se inicia quando o administrado/usuário tem a necessidade de logar no sistema.
2. O sistema exibirá uma tela de login de usuários.
3. O administrado/usuário insere na tela exibida seu e-mail e senha.
4. O sistema verifica as informações de login [FA001].
5. O sistema apresenta uma tela indicando que o usuário está logado no sistema
6. O caso de uso se encerra.

Fluxo Alternativo

[FA001] E-mail e/ou senha errados.

1. O sistema verifica se o e-mail e senha são validos.
2. O sistema exibe uma mensagem erro informando que o e-mail e/ou senha estão incorretos.
3. O caso de uso retorna ao passo 3 do fluxo principal.

[RF002] Cadastrar Usuário

Ator: Administrador.

Descrição do caso de uso: Este caso de uso permite que o administrado cadastre usuários no sistema.

Entradas: O administrador acessa o sistema com seu login e senha e insere nos campos o nome do usuário, senha, e-mail e o perfil para o usuário que será cadastrado.

Pré-condições: Somente o administrador do sistema poderá cadastrar usuários.

Processo: O cadastro será incluído no banco de dados.

Saída: Mensagem de confirmação bem sucedido do cadastro caso tenha sido efetuado com sucesso, senão, mensagem de erro.

Pós-Condição: Usuário cadastrado.

Fluxo Principal

1. O caso de uso se inicia quando o administrador do sistema tem a necessidade de efetuar o cadastro de usuário.
2. O administrador loga no sistema.
3. O administrador insere os dados dos usuários no sistema. [FA001]
4. O administrador confirma os dados do cadastro.
5. O caso de uso se encerra.

Fluxo Alternativo

[FA001] Usuário já cadastrado

1. O sistema verifica se o usuário já está cadastrado.
2. O sistema exibe uma mensagem de erro informando que o usuário inserido já existe.
3. O caso de uso retorna ao passo 3 do fluxo principal.

[RF003] Excluir Usuário

Ator: Administrador

Descrição do caso de uso: Este caso de uso permite que o administrador exclua usuários do sistema.

Entradas: O administrador acessa o sistema com seu login e senha e excluir o usuário cadastrado.

Pré-condições: Somente o administrador do sistema poderá excluir o usuário cadastrado.

Processo: O cadastro será excluído no banco de dados.

Saída: Mensagem de confirmação bem sucedido da alteração do cadastro caso tenha sido efetuado com sucesso, senão, mensagem de erro.

Pós-Condição: Não há pós-condição.

Fluxo Principal

1. O caso de uso se inicia quando o administrador do sistema tem a necessidade de excluir o usuário cadastrado.
2. O administrador loga no sistema.
3. O administrador escolher o do usuário que será excluído.
4. O administrador excluir os dados do cadastro do usuário.
5. O caso de uso se encerra.

[RF004] Cadastrar Secretaria

Ator: Administrador.

Descrição do caso de uso: Este caso de uso permite cadastrar secretarias.

Entradas: O administrador acessa o sistema com seu login e senha e insere no campo o nome da Secretaria que será cadastrada.

Pré-condições: Somente o administrador do sistema poderá cadastrar uma secretaria.

Processo: O cadastro será incluído no banco de dados.

Saída: Mensagem de confirmação bem sucedido do cadastro caso tenha sido efetuado com sucesso, senão, mensagem de erro.

Pós-Condição: Não há pós-condição.

Fluxo Principal

1. O caso de uso se inicia quando o administrador do sistema tem a necessidade de efetuar o cadastro de uma secretaria.
2. O administrador logar no sistema.
3. O administrador insere o nome da secretaria no campo.
4. O administrador confirma o dado do cadastro.
5. O caso de uso se encerra.

[RF005] Excluir Secretaria

Ator: Administrador

Descrição do caso de uso: Este caso de uso permite excluir secretarias cadastradas no sistema.

Entradas: O administrador acessa o sistema com seu login e senha e excluir a secretaria cadastrada.

Pré-condições: Somente o administrador do sistema poderá excluir a secretaria cadastrada.

Processo: O cadastro será excluído no banco de dados.

Saída: Mensagem de confirmação bem sucedido da alteração do cadastro caso tenha sido efetuado com sucesso, senão, mensagem de erro.

Pós-Condição: Não há pós-condição.

Fluxo Principal

1. O caso de uso se inicia quando o administrador do sistema tem a necessidade de excluir a secretaria cadastrada.
2. O administrador logar no sistema.
3. O administrador escolher a secretaria que será excluída.
4. O administrador excluir a secretaria cadastrada.
5. O caso de uso se encerra.

[RF006] Enviar Documento

Ator: Usuário.

Descrição do caso de uso: Este caso de uso permite enviar documentos para setores da estrutura interna da mesma.

Entradas: Deve receber como entrada o nome do setor para onde será enviado o documento, juntamente com o nome do assunto do que o documento se trata e o documento anexado [RF007].

Pré-condições: O usuário deve tá logado no sistema.

Saída: Mensagem de confirmação bem sucedido do envio do documento, caso tenha sido efetuado com sucesso, senão, mensagem de erro.

Pós-Condição: Não há pós-condição.

Fluxo Principal

1. O caso de uso se inicia quando o usuário tem a necessidade de enviar documentos oficiais.
2. O sistema exibirá uma tela contendo a opção documentos.
3. O usuário escolherá a opção novo documento, o sistema redirecionara para uma tela de envio de documentos.
4. O usuário entrada o nome do setor para onde será enviado o documento, juntamente com o nome do assunto do que se trata o documento.
5. O usuário anexar o documento para enviar [RF007].
6. O usuário confirma os dados e envia o documento.
7. O caso de uso se encerra.

[RF007] Anexar Documento

Ator: Usuário.

Descrição do caso de uso: Este caso de uso permite anexar documentos oficiais para que sejam enviados.

Entradas e pré-condições: Deve receber como entrada o caminho absoluto para anexar um documento no sistema.

Saídas e pós-condição: Documento anexado.

Fluxo Principal

1. O caso de uso se inicia quando o usuário precisa anexar um documento.
2. O sistema exibe na tela a opção para anexar documentos.
3. O usuário escolhe o documento que será anexado.
4. O caso de uso se encerra.

[RF008] Receber Documento

Ator: Usuário.

Descrição do caso de uso: Este caso de uso permite que usuários recebam documentos oficiais que foram enviados dos setores da estrutura interna da mesma.

Pré-condições: Documentos enviados.

Pós-condição: Documentos recebidos.

Fluxo Principal

1. O caso de uso se inicia quando o usuário envia um documento para uma dos setores cadastrados [RF006].
2. O sistema exibe na tela os documentos recebidos.
3. O usuário poderá baixar ou excluir os documentos [RF009], [RF010].
4. O caso de uso se encerra.

[RF009] Download de Documento

Ator: Usuário.

Descrição do caso de uso: Este caso de uso permite que usuários faça download dos documentos oficiais que foram recebidos dos setores da estrutura interna da mesma.

Pré-condições: Documentos enviados.

Pós-condição: Documentos recebidos.

Saída: Download realizado com sucesso.

Fluxo Principal

1. O caso de uso se inicia quando o usuário deseja fazer download do documento oficial.
2. O usuário escolher o documento.
3. O usuário faz o download do documento.
4. O caso de uso se encerra.

[RF010] Excluir Documento

Ator: Usuário.

Descrição do caso de uso: Este caso de uso permite que usuários excluam documentos oficiais que foram recebidos dos setores da estrutura interna da mesma.

Pré-condições: Documentos enviados.

Pós-condição: Documentos recebidos.

Saída: Documento excluído com sucesso.

Fluxo Principal

1. O caso de uso se inicia quando o usuário deseja exclui um documento oficial.
2. O usuário escolher o documento.
3. O usuário excluir o documento.
4. O caso de uso se encerra.

[RF011] Lista Documentos

Ator: Usuário.

Descrição do caso de uso: Este caso de uso permite que o usuário liste todos os documentos oficiais que foram recebidos dos setores da estrutura interna da mesma.

Pré-condições: Documentos enviados.

Pós-condição: Documentos recebidos.

Saída: Documentos listados.

Fluxo Principal

1. O caso de uso se inicia quando o usuário deseja listar os documentos oficiais.
2. O usuário escolher em documento a opção lista.
3. O sistema listará os documentos e mostrará na tela.
4. O caso de uso se encerra.

[RF012] Lista Usuários

Ator: Administrador.

Descrição do caso de uso: Este caso de uso permite que o administrador do sistema liste todos os usuários cadastrados no mesmo.

Pré-condições: Não há pós-condição.

Pós-condição: Não há pós-condição.

Saída: Usuários listados.

Fluxo Principal

1. O caso de uso se inicia quando o administrador do sistema deseja listar todos os usuarios cadastrados no mesmo.
2. O administrado escolher em usuário a opção lista.
3. O sistema listará os usuários e mostrará na tela.
4. O caso de uso se encerra.

[RF013] Lista Secretarias

Ator: Administrador.

Descrição do caso de uso: Este caso de uso permite que o administrador do sistema liste todas as secretarias cadastradas no mesmo.

Pré-condições: Não há pós-condição.

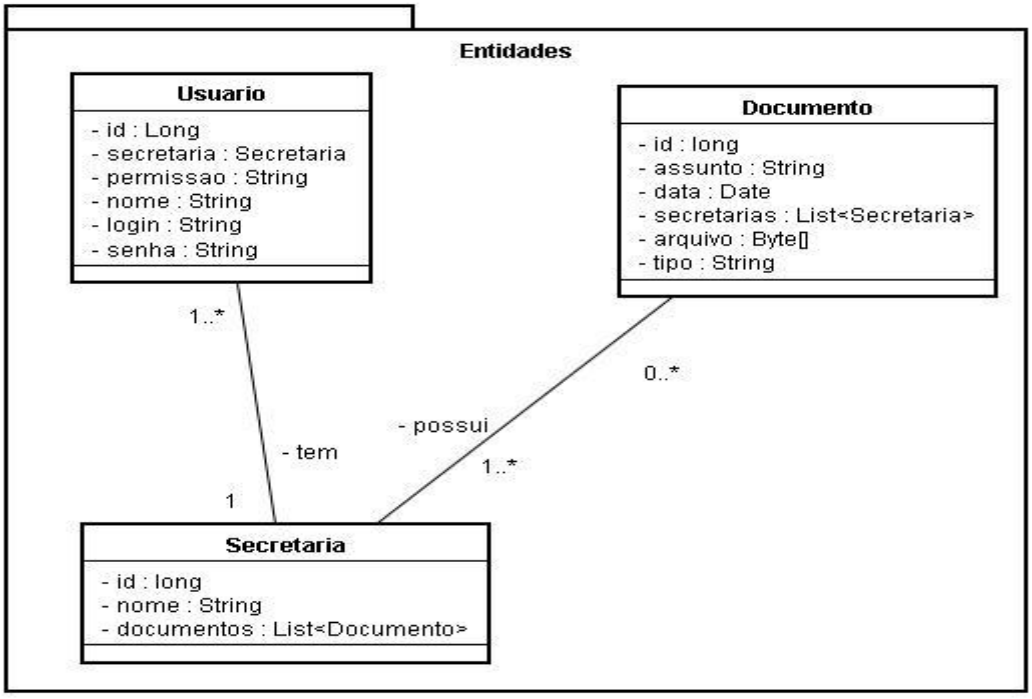
Pós-condição: Não há pós-condição.

Saída: Secretarias listadas.

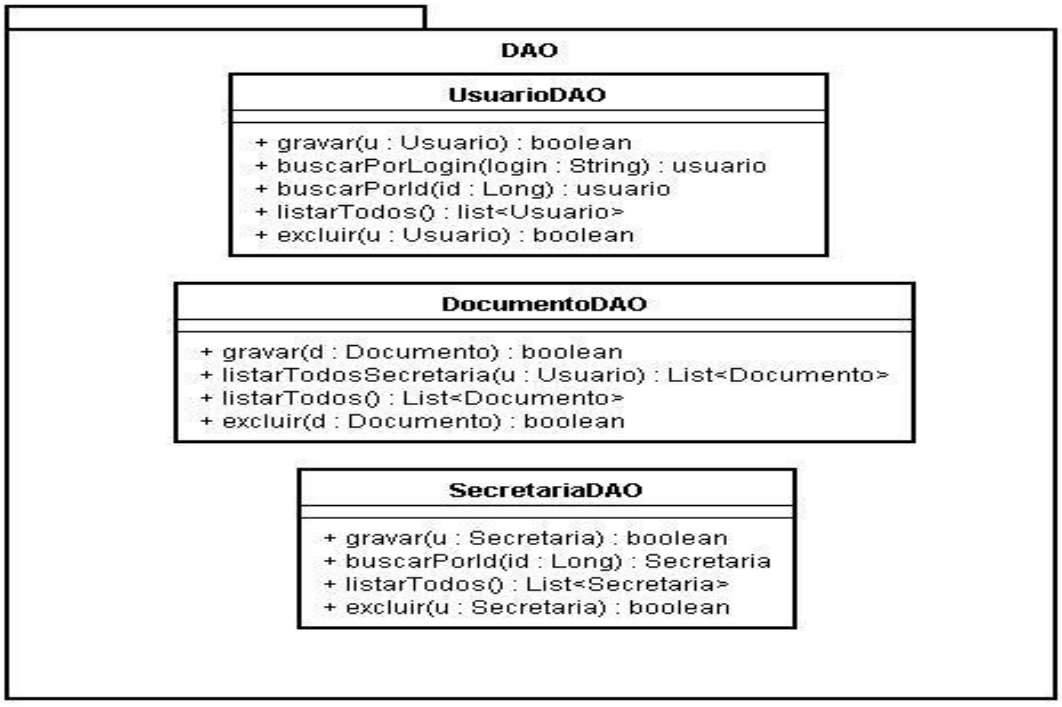
Fluxo Principal

1. O caso de uso se inicia quando o administrador do sistema deseja listar todas as secretarias cadastradas no mesmo.
2. O administrado escolher em secretaria a opção lista.
3. O sistema listará as secretarias e mostrará na tela.
4. O caso de uso se encerra.

1.5 Identificação das Entidades e Atributos do Sistema



1.6 Identificação das Classes DAO e seus métodos para acesso ao Banco de Dados



Clemilda Izaias Santos
Bibliotecária CRB 13/626

Matos, Fernando Ferreira

Desenvolvimento de um sistema web para enviar, armazenar e receber documentos oficiais / Fernando Ferreira Matos. – Balsas, 2013.

68f.:il.

Monografia (Graduação em Sistemas de Informação) – Curso de Sistemas de Informação, Faculdade de Balsas - UNIBALSAS / Balsas, 2013.

1. Desenvolvimento web. 2. Frameworks. 3. Padrões. I. Título.

CDU 004 (812.1Balsas) (02)
M425d