UNVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM TECNOLOGIA
ESPECIALIZAÇÃO EM TECNOLOGIA JAVA

REBECA SUCH TOBIAS FRANCO

ESTUDO COMPARATIVO ENTRE FRAMEWORKS JAVA PARA DESENVOLVIMENTO DE APLICAÇÕES WEB: JSF 2.0, GRAILS E SPRING WEB MVC

MONOGRAFIA DE ESPECIALIZAÇÃO

CURITIBA - PR

2011

REBECA SUCH TOBIAS FRANCO

ESTUDO COMPARATIVO ENTRE FRAMEWORKS JAVA PARA DESENVOLVIMENTO DE APLICAÇÕES WEB: JSF 2.0, GRAILS E SPRING WEB MVC

Monografia de especialização apresentada ao Departamento Acadêmico de Informática da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do título de "Especialista em Tecnologia Java".

Orientador: Prof. MSc. Leandro Batista de Almeida.

Co-orientador: Prof. MSc. Maurício Correia Lemes Neto.

CURITIBA - PR

2011

AGRADECIMENTOS

Primeiramente a Deus, pela vida, saúde e inspiração que me permitiram escrever este trabalho.

Aos meus pais, Valdimir e Sônia, que sempre me incentivaram a estudar e lutar pelos meus objetivos.

Ao meu irmão, Gabriel, que compartilha do mesmo interesse que eu pela informática e me emprestou livros que me ajudaram no desenvolvimento deste trabalho.

Ao meu marido, Jonas, por sempre me apoiar em tudo o que eu faço e por compreender o tempo em que estive ocupada me dedicando a esta pós-graduação.

Aos colegas de curso, que tornaram os momentos de estudo mais agradáveis.

Aos professores, que transmitiram seu conhecimento e experiência, permitindo que nos tornássemos melhores profissionais.

A mente que se abre a uma nova ideia jamais voltará ao seu tamanho original. (Albert Einstein).

RESUMO

FRANCO, Rebeca S. T. Estudo Comparativo entre Frameworks Java para Desenvolvimento de Aplicações Web: JSF 2.0, Grails e Spring Web MVC. 2011. Monografia (Especialização em Tecnologia Java) – Programa de Pós-Graduação em Tecnologia, Universidade Tecnológica Federal do Paraná. Curitiba, 2011.

A plataforma Java é a mais utilizada para o desenvolvimento de aplicações web corporativas. O mercado exige que o software seja desenvolvido com o menor tempo e custo possível e com qualidade. Para isso tornou-se essencial a utilização de frameworks, pois eles aumentam a produtividade, maximizam o reuso e reduzem a possibilidade de erros. Atualmente existem diversos frameworks web disponíveis para a plataforma Java. Este trabalho apresenta alguns conceitos relacionados ao desenvolvimento web em Java e descreve três dos principais frameworks utilizados recentemente para essa finalidade: JSF 2.0, Spring Web MVC e Grails. Também realiza uma comparação entre eles. Através da implementação de um protótipo de um sistema financeiro são verificadas na prática as comparações teóricas apresentadas e verifica-se qual dos frameworks apresenta mais vantagens ao desenvolvedor. Conclui-se que Grails é o mais fácil de aprender e o mais simples de ser utilizado.

Palavras-chave: Java. Frameworks Web. JSF. Grails. Spring Web MVC.

ABSTRACT

FRANCO, Rebeca S. T. Comparative Study between Java Frameworks for Development of Web Applications: JSF 2.0, Grails and Spring Web MVC. 2011. Monografia (Especialização em Tecnologia Java) – Programa de Pós-Graduação em Tecnologia, Universidade Tecnológica Federal do Paraná. Curitiba, 2011.

The Java platform is the most used for the development of corporate web applications. The market demands that the software has to be developed with the least time and cost as possible and with quality and for that, the use of frameworks has become essential, because they increase productivity, maximize reuse and reduce the possibility of errors. Currently, there are many web frameworks available for Java. This paper presents some concepts related to web development in Java and describes three major frameworks presently used for this purpose: JSF 2.0, Spring Web MVC and Grails. It also makes a comparison between these frameworks. Through the implementation of a prototype of a financial system, it's possible to observe in practice the theoretical comparisons presented and define which framework has more advantages to the developer. It is concluded that Grails is the easiest to learn and the more simple to be used.

Keywords: Java. Web frameworks. JSF. Grails. Spring Web MVC.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE SIGLAS

Ajax	Asynchronous JavaScript and XML
API	Application Programming Interface
CoC	Convention over Configuration
CRUD	Create/Read/Update/Delete
DAO	Data Access Object
DI	Dependency Injection
DSL	Domain Specific Language
EJB	Enterprise JavaBeans
GDK	Groovy Development Kit
GORM	Grails Object Relation Mapping
GSP	Groovy Server Pages
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IoC	Inversion of Control
JDK	Java Development Kit
JPA	Java Persistence API
JSF	JavaServer Faces
JSON	JavaScript Object Notation
JSP	JavaServer Pages
JSR	Java Specification Request
JSTL	JavaServer Pages Standard Tag Library
JVM	Java Virtual Machine
LGPL	GNU Lesser General Public License
MVC	Model-View-Controller
ORM	Object-Relational Mapping
POJO	Plain Old Java Object
UI	User Interface

SUMÁRIO

1 INTRODUÇÃO

1.1 FRAMEWORKS WEB

O ambiente mais utilizado atualmente para o desenvolvimento de aplicações corporativas na plataforma Java é o ambiente web.

No início eram utilizados somente Servlets (classes Java que processam requisições dinamicamente e constroem respostas) e JSPs (*JavaServer Pages* – documentos textos executados como Servlets e apropriados para geração de páginas web), porém logo se percebeu que essa forma de trabalhar não era tão produtiva e organizada. Começaram então a serem utilizados alguns padrões, como o MVC (*Model-View-Controller*) e o *Front Controller* e tornou-se comum as empresas implementarem esses padrões e criarem soluções baseadas em miniframeworks caseiros. Mas ainda existia um grande retrabalho para reimplementar o padrão todo a cada projeto.

O Struts foi o primeiro framework de sucesso criado para a plataforma Java. Ele foi lançado em 2000, permitindo a criação de um controlador reutilizável entre os projetos e disponibilizando várias funcionalidades já prontas, aumentando significativamente a produtividade dos desenvolvedores. Com isso ele se tornou a principal solução MVC no mercado (CAELUM).

Porém, com a evolução da linguagem Java surgiram mais facilidades e o Struts passou a ser considerado um framework muito trabalhoso. Novos frameworks foram desenvolvidos para facilitar cada vez mais o processo de desenvolvimento de software, existindo atualmente inúmeras opções.

1.2 OBJETIVO GERAL

O objetivo deste trabalho é desenvolver um estudo sobre alguns dos frameworks web em Java mais utilizados atualmente no mercado: JSF 2.0, Spring Web MVC e Grails.

1.3 OBJETIVOS ESPECÍFICOS

Os objetivos específicos são descrever e comparar as características mais relevantes desses três frameworks, como implementação do padrão MVC, validação, suporte a Ajax, publicações disponíveis e vagas de emprego oferecidas, e verificar qual deles oferece mais vantagens ao desenvolvedor.

1.4 JUSTIFICATIVA

A utilização de frameworks para o desenvolvimento de aplicações web se tornou essencial nos últimos anos, pois eles fornecem muitos benefícios, como: ganho de produtividade, redução da possibilidade de erros, maior nível de abstração, compatibilidade e integração entre aplicações, redução de custos e maximização de reuso.

Porém, existem inúmeras opções de frameworks, tornando a escolha do melhor uma tarefa difícil. Esse trabalho irá auxiliar nessa decisão, pois irá comparar os frameworks web em Java mais relevantes em nossa região.

1.5 METODOLOGIA

Para o desenvolvimento desse trabalho foi realizado um levantamento bibliográfico sobre alguns conceitos de programação web em plataforma Java e sobre os frameworks JSF 2.0, Grails e Spring Web MVC para caracterização dos mesmos. Em seguida foi feita uma comparação entre esses frameworks. Também foi desenvolvido um protótipo de uma aplicação de controle financeiro em cada um dos frameworks para demonstrar melhor algumas de suas características e diferenças.

1.6 ORGANIZAÇÃO DO TRABALHO

O trabalho está organizado da seguinte maneira:

O capítulo 2 descreve alguns conceitos importantes utilizados na programação web em Java e caracteriza os frameworks JSF, Spring Web MVC e Grails.

O capítulo 3 faz uma comparação de várias características relevantes dos três frameworks e das linguagens Java e Groovy.

O capítulo 4 descreve a implementação do protótipo de um sistema de controle financeiro em cada um dos frameworks, enfatizando as diferenças entre cada um.

No capítulo 5 são apresentadas as considerações finais e conclusões obtidas neste trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 PADRÕES DE PROJETO

Padrões de projeto são soluções para problemas recorrentes no desenvolvimento de software e que podem ser reusadas por outros desenvolvedores.

Os componentes essenciais de um padrão de projeto são: nome, objetivo, problema, solução e consequências.

2.2 MVC

De acordo com Gonçalves (2007), *Model-View-Controller* (MVC) é um conceito de engenharia de software para desenvolvimento e *design*, que propõe a separação de uma aplicação em três partes distintas: modelo, visão e controle. O modelo é usado para definir e gerenciar o domínio da informação, a visão é responsável por exibir os dados ou informações da aplicação e o controle controla as duas camadas anteriores, exibindo a interface correta ou executando algum trabalho complementar para a aplicação.

O MVC é o padrão de arquitetura de software mais recomendado para aplicações interativas, conforme Singh (2002).

2.3 FRONT CONTROLLER

Front Controller (Controlador Frontal) é um padrão bastante utilizado em frameworks MVC, onde um único servlet é responsável por receber todas as requisições, delegar o processamento para os outros componentes da aplicação e devolver uma resposta ao usuário.

2.4 INVERSÃO DE CONTROLE

Conforme Machado (2008), Inversão de Controle (*Inversion of Control*, ou IoC) consiste na mudança do fluxo de controle de um programa, onde ao invés do programador determinar quando um procedimento será executado, ele apenas determina qual procedimento deve ser executado para um determinado evento. Isso propicia o reaproveitamento do código que contém o fluxo de controle, do *design* e do comportamento da aplicação, possibilitando que seja atingido um baixo acoplamento na arquitetura e um alto nível de testabilidade.

2.5 INJEÇÃO DE DEPENDÊNCIA

Injeção de Dependência (*Dependency Injection*, ou DI) é um padrão de projeto cujo objetivo é desacoplar os componentes de uma aplicação, instanciando-os externamente à classe e controlando suas instâncias através de um gerenciador. Consiste em uma implementação que faz uso de Inversão de Controle.

2.6 PROGRAMAÇÃO POR CONVENÇÃO

Programação por convenção, ou CoC (*Convention Over Configuration*), é um paradigma que visa diminuir a quantidade de decisões que o desenvolvedor precisa tomar, tomando como padrão algo que é comumente usado, uma convenção.

2.7 PARADIGMA DE ORIENTAÇÃO A ASPECTOS

Segundo Torsten (2011), o paradigma de orientação a aspectos consiste na decomposição do sistema em partes não entrelaçadas e espalhadas (fase de decomposição) e

em juntar essas partes novamente de forma significativa para obter o sistema desejado (processo de composição).

A programação orientada a aspectos possibilita que os interesses comuns do sistema sejam separados em aspectos, que encapsulam comportamentos que afetam múltiplas classes. Como consequência são obtidos benefícios como: melhor modularização, possibilidade de reutilização de grande parte dos módulos desenvolvidos e facilidade de manutenção e evolução do software.

2.8 JAVABEANS

De acordo com a especificação do JavaBeans (Sun Microsystems, 1997), “Java Bean é um componente reutilizável de software que pode ser manipulado visualmente com a ajuda de uma ferramenta de desenvolvimento".

As classes JavaBeans devem: implementar a interface *java.io.Serializable*; possuir um construtor sem argumentos; possibilitar o acesso às suas propriedades através de métodos “*get*” e “*set*”; possibilitar a inclusão de qualquer método de tratamento de eventos.

2.9 FRAMEWORK

Coad (1992) define um framework como um esqueleto de classes, objetos e relacionamentos agrupados para construir aplicações específicas.

Johnson e Woolf (1998) descrevem framework como um conjunto de classes abstratas e concretas que provê uma infraestrutura genérica de soluções para um conjunto de problemas.

Os frameworks permitem a reutilização de toda a arquitetura de um domínio específico, não somente de componentes isolados, diminuindo o tempo e o esforço exigidos na produção de software.

2.10 HIBERNATE

O Hibernate é um framework de persistência escrito na linguagem Java, com código aberto e distribuído com licença LGPL (*GNU Lesser General Public License*).

> Historicamente, o Hibernate facilitou o armazenamento e recuperação de objetos de domínio Java através do Mapeamento Objeto/Relacional. Hoje, o Hibernate é uma coleção de projetos relacionados que permitem aos desenvolvedores utilizarem modelos de domínio POJO em suas aplicações de forma a estender muito além do Mapeamento Objeto/Relacional. (JBoss Community).

2.11 FRAMEWORK JSF

Conforme Nunes e Magalhães (2010), JavaServer Faces (JSF) é uma especificação técnica do *Java Community Process* (JCP), publicada em 2004, com o objetivo de padronizar um framework para desenvolvimento da camada de apresentação em aplicações web.

JSF é um framework baseado em componentes. Com isso, para criar uma tabela, por exemplo, ao invés de criar um loop para adicionar linhas e colunas com tags HTML, é adicionado um componente tabela à página.

Os principais objetivos desse framework são maximizar a produtividade, minimizar a complexidade de manutenção, evoluir a experiência do usuário com uso de técnicas Ajax e proporcionar melhor integração com outras tecnologias web.

Conforme descrito no tutorial do Java 6 – *The Java EE 6 Tutorial* (Oracle, 2011), os principais componentes da tecnologia JavaServer Faces são:

- Uma API para representar componentes *UI (User Interface)* e gerenciar seu estado; manipulação de eventos, validação do lado servidor, e conversão de dados; definição de navegação de página; suporte à internacionalização e acessibilidade; e possibilidade de extensão para todas essas características.
- *Tag libraries* (bliblioteca de tags) para adicionar componentes às páginas web e conectar componentes a objetos do lado do servidor.

Uma típica aplicação JavaServer Faces inclui:

- Um conjunto de páginas web nos quais os componentes são definidos.
- Um conjunto de *tags* para adicionar componentes à página web.

- Um conjunto de *backing beans*, que são componentes *JavaBeans* que definem propriedades e funções para componentes de uma página.
- Um descritor de *deploy* web (arquivo web.xml).
- Opcionalmente, um ou mais arquivos de recursos de configuração da aplicação, como o arquivo *faces-config.xml*, os quais podem ser usados para definir uma página de regras de navegação e configurar *beans* e outros objetos e componentes customizados.
- Opcionalmente, um conjunto de objetos customizados, que podem incluir componentes customizados, validadores, conversores, ou *listeners*, criados pelo desenvolvedor da aplicação.
- Um conjunto de *tags* customizadas para representar objetos customizados na página.

A versão mais atual do JSF é a 2.0 (JSR - *Java Specification Requests* - 314), que introduziu soluções como gerenciamento de recursos, Facelets, suporte a Ajax, configuração com anotações e novos escopos.

JSF é uma especificação, portanto é necessário escolher uma implementação. A implementação de referência da Oracle é a Mojarra, mas existem várias outras, como: *MyFaces*, ADF (*Application Development Framework)* e *Shale*. Além disso, existem extensões de implementações, como o *ICEFaces*, que altera o comportamento padrão do JSF e possui vários componentes visuais. JSF também possui vários Kit Componentes, como *Apache Tomahawk, JBoss Ajax4JSF*, *JBoss RichFaces, Yahoo for JSF*, *DynaFaces*, *Jenia Faces*, *Web Galileo Faces*, *Mojarra UI*, *RestFaces*, *GMaps4JSF* e *PrimeFaces*.

JSF permite ainda integração com outras tecnologias e frameworks, como Spring, *Java Persistence API* (JPA), *Enterprise JavaBeans* (EJB) e *JBoss Seam*.

JSF é o framework Java mais utilizado para desenvolvimento de aplicações web. Alguns dos principais motivos para isso são: oferece um excelente conjunto de funcionalidades, possui bons materiais de estudo e exige pouco conhecimento inicial para construção de interfaces de usuários tradicionais.

2.12 FRAMEWORK SPRING WEB MVC

Spring é um framework leve com inversão de controle e orientado a aspectos. Spring Web MVC é um módulo robusto e flexível desse framework para desenvolvimento rápido de aplicações web utilizando o padrão de projeto MVC.

As principais características do Spring Web MVC, de acordo com sua documentação de referência (Johnson et al., 2011), são:

- Clara separação de papéis. Cada papel – controlador, validador, objeto de comando, objeto de formulário, objeto de modelo, *DispatcherServlet*, mapeamento de tratadores, resolvedores de visualização – podem ser definidos por um objeto especializado.
- Configuração simples e eficiente de classes de aplicações e frameworks como *JavaBeans*. Essa capacidade de configuração inclui fácil referência em diferentes contextos, como as dos controladores web para objetos de negócio e validadores.
- Adaptabilidade, não-intrusão e flexibilidade. Definição de qualquer assinatura de método de controle que for necessária, possivelmente usando uma das anotações de parâmetros (como *@RequestParam*, *@RequestHeader*, *@PathVariable*, etc.) para um determinado cenário.
- Reusabilidade do código de negócio, não sendo necessária duplicação. Usam-se objetos de negócio existentes como comandos ou objetos de formulários ao invés de replicá-los para estender uma base de classe particular do framework.
- Vínculos e validações customizáveis. Validação de tipos no nível da aplicação, que verificam o valor, localizam datas e números e permitem análise manual e conversão para objetos de negócio ao invés de somente objetos String de formulários.
- Mapeamento de tratadores e resolvedores de visualização customizáveis. Estratégias de mapeamento de tratadores e resolução de visualização variam de uma simples configuração baseada na URL até sofisticadas estratégias de resolução para este propósito. Spring é mais sofisticado que frameworks MVC que utilizam uma técnica particular.
- Modelo flexível de transferência. Modelo de transferência com mapeamento nome/valor suporta fácil integração com qualquer tecnologia de visão.

- Localidade customizável e resolução de temas, suporte para JSPs com ou sem *tag library* Spring, suporte para *JavaServer Pages Standard Tag Library* (JSTL), suporte para *Velocity* (engine baseado em Java para a construção de templates) sem necessidade de *plugins* extras, etc.
- Uma biblioteca de tags conhecida como *Spring tag library* que providencia suporte para características como vínculo de dados e temas. Essas *tags* customizadas permitem a máxima flexibilidade em termos de código de marcação.
- *JSP form tag library*, introduzido no Spring 2.0, que torna a escrita de formulários em páginas JSP muito mais fácil.
- *Beans* cujo ciclo de vida tem como escopo a solicitação atual HTTP ou a sessão HTTP. Essa não é uma característica do Spring MVC em si, mas sim do container *WebApplicationContext* que o Spring usa.

2.13 FRAMEWORK GRAILS

O framework Grails foi criado em 2006. Ele foi baseado nas ideias do *Ruby on Rails* e utiliza a linguagem Groovy, por isso seu nome inicial era *Groovy on Rails*, mas precisou ser trocado a pedido de David Heinemeier Hansson (o criador do *Ruby on Rails*). Seu foco é a alta produtividade e simples configuração.

Groovy é uma linguagem de *scripting*, orientada a objetos e dinamicamente tipada, que roda de forma nativa na *Java Virtual Machine* (JVM), o que permite que todo código Java pré-existente possa ser reaproveitado.

Segundo Subramaniam (2008) o Groovy traz o melhor dos dois mundos: uma linguagem flexível, de alta produtividade, ágil e dinâmica que funciona no rico framework da Plataforma Java.

Esse framework utiliza o princípio *Don't Repeat Yourself* (DRY) e reduz drasticamente a complexidade de construir aplicações web na plataforma Java.

Algumas das vantagens oferecidas pelo Grails são:

- Uma camada para mapeamento objeto-relacional fácil de usar construída no Hibernate.
- Uma tecnologia de visão expressiva, chamada *Groovy Server Pages* (GSP).

- Uma camada de controle construída no Spring MVC.
- Um ambiente em linha de comando construído no Groovy: Gant.
- Um container TomCat integrado que é configurado para recarregar quando necessário.
- Injeção de dependência com o container Spring embutido.
- Suporte à internacionalização (i18n) construída sobre o conceito *MessageSource* do núcleo do Spring.
- Uma camada de serviços transacionais construída na abstração de transações do Spring.

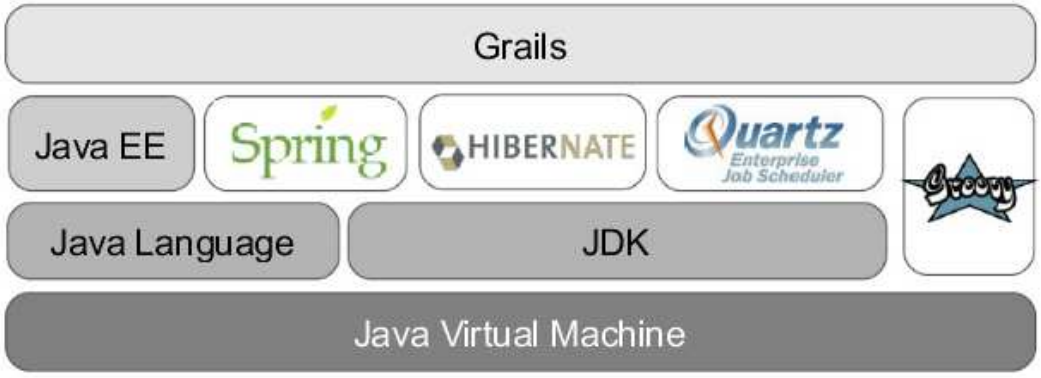

Figura 1 – Componentes do Grails

Fonte: Infonova (2008)

Outra grande vantagem oferecida pelo Grails é o fato de se tratar de um ambiente completo de desenvolvimento, dispensando a necessidade de baixar arquivos .jar da internet para poder trabalhar adequadamente.

Grails usa “convenção sobre configuração”, ou seja, o nome e a localização de arquivos são usados ao invés de configuração explícita. Para isso existe uma estrutura de diretório padrão:

- grails-app - diretório de nível superior para os arquivos fontes de Groovy
    - conf – arquivos de configuração.
    - controllers – controladores web.
    - domain – o domínio da aplicação.
    - i18n – suporte para internacionalização (i18n).

    - services – a camada de serviços.
    - taglib – *tag libraries*.
    - views – *Groovy Server Pages*.
- scripts – scripts Gant.
- src – arquivos fontes de apoio.
    - groovy – outros arquivos fontes Groovy
    - java – outros arquivos fontes Java
- test – testes de unidade e integração.

Grails possui uma propriedade chamada *Scaffolding* que permite que uma aplicação inteira seja gerada através das classes de domínio, incluindo as visões necessárias e as *actions* de controle para operações CRUD (*create*/*read*/*update*/*delete*).

A versão mais recente do Grails é a 1.3.7, que empacota o Groovy 1.7.8. Atualmente existem 623 *plugins* disponíveis para esse framework, que podem ser baixados através do endereço http://grails.org/plugin/category/all.

3 COMPARAÇÃO ENTRE OS FRAMEWORKS

Os frameworks JSF, Grails e Spring Web MVC possuem vários atributos em comum, como a arquitetura em MVC, utilização do padrão *Front Controller*, suporte a validação e conversão de dados, internacionalização e manipulação de erros. Neste capítulo serão comparados algumas de suas principais características.

3.1 IMPLEMENTAÇÃO DO PADRÃO MVC

3.1.1 JSF 2.0

No JSF a camada de controle é composta por:

- *FacesServlet* – recebe as requisições da WEB, redireciona-as para o modelo e remete uma resposta.
- Arquivos de configuração - realizam associações e mapeamentos de ações e definem as regras de navegação.
- Manipuladores de eventos – recebem os dados vindos da camada de visualização, acessam o modelo e devolvem o resultado ao *FacesServlet*.

A camada modelo é representada por objetos de negócio (Java Beans), que executam uma lógica de negócio ao receber os dados oriundos da camada de visualização.

A camada de visualização é representada por uma hierarquia de componentes (*component tree*), que possibilita a união de componentes para construir interfaces mais ricas e complexas.

Na versão 2.0 o padrão para montagem de *templates* é o Facelets (extensão xhtml), mas também podem ser utilizadas outras tecnologias como JSP e Clay.

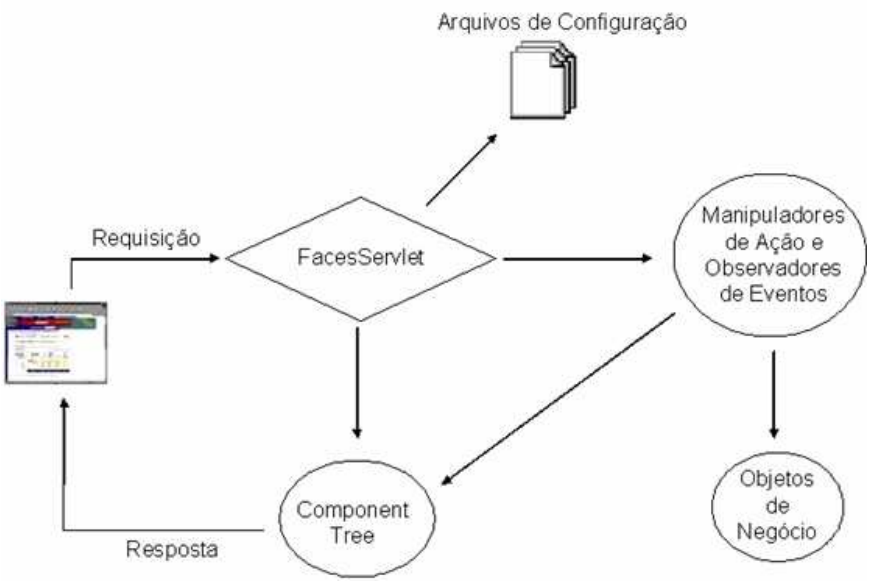

Figura 2 – Arquitetura JSF baseada no modelo MVC

Fonte: Pitanga

3.1.2 Spring Web MVC

No Spring Web MVC, a camada de controle é composta por:

- *DispatcherServlet* – processa todas as requisições do usuário e invoca elementos *Controller* apropriados.
- *HandlerMapping* – baseado na URL da requisição indica ao *DispatcherServlet* qual é o controlador a ser invocado.
- *HandlerAdapter* – responsável por delegar a requisição para mais processamentos.
- *ViewResolver* – interface que devolve ao *DispatcherServlet* qual visão deve ser buscada e instanciada, baseada em seu nome e localização.
- Controlador – processa os dados e executa alguma regra de negócio da aplicação. Para criar um controlador em Spring 3.0 basta usar a anotação *@Controller* na classe.

Muitos dos métodos das subclasses do *Controller* retornam um objeto *org.springframework.web.servlet.ModelandView*. Esse objeto encapsula o modelo (como um objeto *java.util.Map* que mantém JavaBeans que a interface *View* irá compor) e o nome da visão, possibilitando retornar ambos em um valor de retorno de um método para o *Dispatcher*.

A camada de visão do Spring é representada por várias tecnologias: JSP, JSTL, Velocity, FreeMarker, JasperReport, XSLT, Tiles, PDF e Excel.

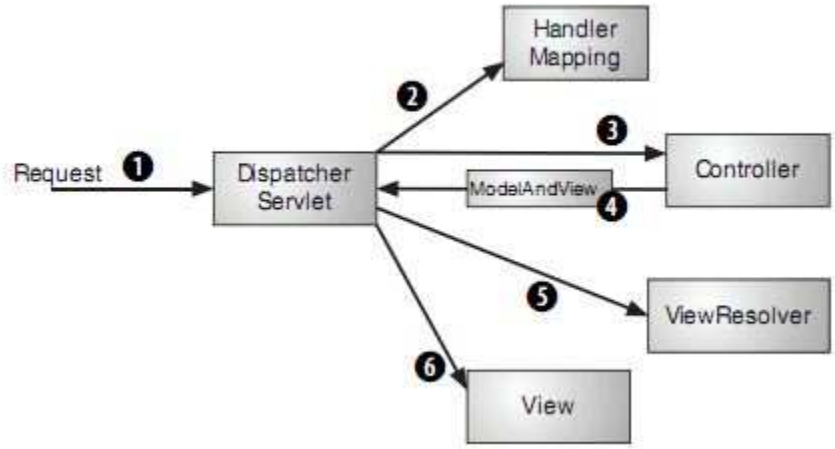

Figura 3 – Fluxo de informações no Spring Web MVC
Fonte: Saab (2011)

3.1.3 Grails

De acordo com Seifeddine (2009), o modelo do Grails é representado com classes de domínio assim como os outros frameworks apresentados, porém essas são automaticamente persistidas e podem até mesmo gerar o esquema de dados. Grails trata as classes de domínio como o componente central e mais importante da aplicação.

Grails utiliza o framework de persistência Hibernate, que provê uma solução de mapeamento objeto-relacional (ORM – *Object Relational Mapping*) para aplicações. Ao invés de construir sua própria solução ORM a partir do zero, Grails envolveu o Hibernate em uma API Groovy, chamada *Grails Object Relation Mapping* (GORM), que provê a linguagem *Domain Specific Language* (DSL) que usa convenção das próprias classes para construir o mapeamento, tornando desnecessária a utilização de arquivos externos, como XML, no Hibernate.

Os controladores basicamente manipulam as solicitações e preparam a resposta. Para criar um controlador em Grails basta criar uma classe cujo nome termine com Controller e salvá-la no diretório *grails-app/controllers/*.

Visões em Grails são tipicamente *Groovy Server Pages* (GSP) que são uma extensão de JSP, porém mais flexíveis e convenientes para se trabalhar, mas Grails suporta ambos os tipos. Cada visão GSP tem acesso a um modelo que é mapeado basicamente com chaves e valores que podem ser passados pelo controlador e usados na visão.

3.2 VALIDAÇÃO

3.2.1 JSF 2.0

JSF possui um amplo suporte para validação dos dados de entrada. É possível usar os validadores padrões do JSF ou métodos ou classes próprias para validação. Se o valor de um componente não for aceito, uma mensagem de erro é gerada para esse componente específico e o atributo da classe associado ao mesmo não é modificado.

As validações do JSF são todas feitas do lado do servidor. Caso seja necessário alguma validação do lado cliente é necessário que o desenvolvedor crie seus próprios validadores em JavaScript.

A interface que define o modelo de validação do JSF é a *Validator*. Os validadores padrões do JSF 2.0 são:

- *BeanValidator* – Delega a validação do *bean* para a *Bean Validation API* (JSR 303), que permite validar dados de forma ágil e rápida através do uso de anotações. Conforme Tosin (2010) essa API independe da camada da aplicação onde é usada e da forma de programar (pode ser usada em aplicações web e desktop) e não está atrelada ao mecanismo de persistência usado. Outra grande vantagem da *Bean Validation API* é possibilitar a validação de dados nas classes de domínio da aplicação, assim ao invés de verificar os dados nas diversas camadas o processo fica centralizado, pois os objetos destas classes normalmente trafegam entre as camadas da aplicação. A implementação de referência da JSR 303 é o *Hibernate Validator*, cujas anotações disponíveis estão descritas no Quadro 1.

- *DoubleRangeValidator* – Checa se o valor do componente correspondente está de acordo com o valor mínimo e máximo especificado.
- *LengthValidator* – Checa o número de caracteres da String que representa o valor do componente associado.
- *LongRangeValidator* – Checa se o valor do componente correspondente está de acordo com o valor mínimo e máximo especificado.
- *MethodExpressionValidator* – Envolve um *MethodExpression* e realiza a validação através da execução de um método em um objeto identificado pelo *MethodExpression*.
- *RegexValidator* – Checa o valor de acordo com uma expressão regular (que é uma propriedade *pattern*).
- *RequiredValidator* – Checa se o valor é vazio.

A exceção que é lançada quando o método validate() de uma classe de validação falha é a *ValidationException*.

A anotação @FacesValidator em uma classe a torna uma classe de validação.

Annotation	**Descrição**
@CreditCardNumber	Verifica se o elemento representa um número válido de cartão de crédito.
@Email	Verifica se a string contém um e-mail com formato válido.
@Length	Verifica se a string está entre o valor mínimo e máximo definidos.
@NotBlank	Verifica se a string não é vazia ou nula.
@NotEmpty	Verifica se a string, collection, map ou array não é nula ou vazia.
@Range	Verifica se o elemento está no padrão definido.
@ScriptAssert	Uma restrição de nível da classe, que avalia uma expressão de script contra o elemento anotado.
@URL	Verifica se a string é uma url válida.

Quadro 1 - Anotações do *Hibernate Validator*
Fonte: Adaptado de http://docs.jboss.org/hibernate/validator/4.1/api/.

3.2.2 Spring

No framework Spring pode-se criar uma classe para ser invocada pelo controlador para validar dados de formulários. Essa classe deve ser concreta e implementar a interface *org.springframework.validation.Validator*.

A interface *Validator* requer a implementação de dois métodos:

1. boolean supports (Class<?> clazz)

Esse método verifica se a classe de validação pode validar instâncias do parâmetro *clazz*. Sua implementação geralmente consiste em:

```
return NomeClasse.class.isAssignableFrom(classe);
```

onde NomeClasse é a classe ou superclasse da instância atual do objeto a ser validado.

2. void validate(Object target, Errors errors)

Esse método valida o objeto *target* fornecido, que deve ser uma classe em que o método *supports* retorne *true* (pode ser *null*). O parâmetro *errors* pode ser usado para fornecer qualquer resultado de validação de erros (não pode ser *null*).

Outra notável classe de validação é *org.springframework.validation.ValidationUtils*, que provê métodos convenientes para invocar um *Validator* e rejeitar campos vazios.

3.2.3 Grails

Para validação de propriedades das classes de domínio do Grails são declaradas *constraints* usando blocos de código estático Groovy. As *constraints* disponíveis na versão 1.3.7 do Grails estão representadas no Quadro 2.

Exemplo de uma declaração de *constraint* em Groovy:

```
class Usuario {
        ...
      static constraints = {
        email (blank:false, email:true, unique:true)
        senha (blank:false, maxSize:12, minSize:6)
        nomeCompleto (blank:false, matches:"[a-zA-Z ]+", maxSize:50,
minSize:6)
        CPF (blank:false, matches:"[0-9]+", size:11..11)
        dataNascimento (blank:false, max:new Date())
      }
}
```

Além das *constraints* do Quadro 2 existem outras que afetam o *scaffolding*. Geralmente não é uma boa prática incluir informações de interface de usuário no domínio, mas é uma grande conveniência quando se usa a extensão do *scaffolding* do Grails. Essas *constraints* estão representadas no Quadro 3.

Constraint	Descrição	Exemplo
blank	Verifica se o valor da String não é branco.	login(blank:false)
creditCard	Verifica se o valor da String é um número de cartão de crédito válido.	numCartao(creditCard:true)
email	Verifica se o valor da String é um endereço de e-mail válido.	email(email:true)
inList	Verifica se o valor da String está dentro de um intervalo ou conjunto de valores limitados.	nome(inList:"Joe", "Fred", "Bob")
matches	Verifica se o valor da String está de acordo com uma expressão regular.	login(matches:"[a-zA-Z]+")
max	Assegura que um valor não excede o valor máximo informado.	idade(max:new Date()) preço(max:999F)
maxSize	Assegura que o tamanho de um valor não excede o valor máximo informado.	filhos(maxSize:25)
min	Assegura que um valor não é inferior ao valor mínimo informado.	idade(min:new Date()) preço(min:0F)
minSize	Assegura que o tamanho de um valor não é inferior ao valor mínimo informado.	filhos(minSize:25)
notEqual	Assegura que uma propriedade não é igual ao valor especificado.	login(notEqual:"Bob")
nullable	Permite que uma propriedade seja atribuída como null – o valor default é false para propriedade da classe de domínio.	idade(nullable:true)
range	Usa um intervalo Groovy para assegurar que o valor de uma propriedade ocorre dentro de um intervalo específico.	age(range:18..65)
scale	Atribui a escala desejada para números de ponto flutuante.	salary(scale:2)
size	Usa um intervalo Groovy para restringir uma coleção de números ou o tamanho de uma String.	filhos(size:5..15)
unique	Restringe uma propriedade como única no nível da base de dados.	login(unique:true)
url	Verifica se o valor de uma String é uma URL válida.	homePage(url:true)
validator	Adiciona uma validação customizada a um campo.	teste(validator: {return (x > y})

Quadro 2 - Constraints do Grails
Fonte: Adaptado de http://grails.org/doc/latest/ref/Constraints/Usage.html.

Constraint	Descrição
display	Boolean que determina se uma propriedade no scaffolding é vista. O valor default é true.
editable	Boolean que determina se uma propriedade pode ser editada nas visões do scaffolding. Se for false, no campo do formulário associado é exibido como somente leitura.
format	Especifica um formato de exibição para campos como datas. Por exemplo, 'yyyy-MM-dd'.
password	Boolean que indica se um campo deve ser mostrado como password. Somente funciona com campos que normalmente podem ser exibidos como um campo de texto.
widget	Controla qual componente é usado para mostrar a propriedade. Por exemplo, 'textArea' irá forçar o scaffolding a usar uma tag <textArea>.

Quadro 3 - Constraints do Grails que afetam o scaffolding
Fonte: Adaptado de http://grails.org/doc/latest/ref/Constraints/Usage.html.

3.3 LINGUAGEM JAVA X GROOVY

A linguagem Groovy tenta ser o mais natural possível para desenvolvedores Java, entretanto existem algumas diferenças básicas, conforme listado em sua documentação oficial:

- **Importações default.** Os seguintes pacotes e classes são importados por *default*, sendo desnecessário usar *import* explícito para utilizá-los:
  - java.io.*
  - java.lang.*
  - java.math.BigDecimal
  - java.math.BigInteger
  - java.net.*
  - java.util.*
  - groovy.lang.*
  - groovy.util.*
- **"==" significa sempre igualdade.** Em Java "==" significa igualdade para tipos primitivos e identidade para objetos. Em Groovy, para verificar identidade é utilizado o método "is". Para verificar se uma variável é nula usa-se ==, como em Java.
- **"in" é uma palavra chave, não pode ser usada como variável.**
- **Na declaração do conteúdo de arrays são utilizados colchetes ao invés de chaves.** Exemplo:

```
int[] a = [1,2,3]
```

- **Existem mais formas de escrever um *for loop*.** Exemplos:

```
for (int i=0; i < 10; i++){} //igual ao Java
for (i in 0..10){}
10.times{}
```

- **Ponto e vírgula são opcionais.** Apenas são necessários pontos e vírgulas para separar sentenças escritas em uma mesma linha.
- **A palavra-chave "*return*" é opcional**. O valor de retorno de um último comando de um método corresponde ao seu retorno, portanto é desnecessário escrever o "*return*".

- **Todo código que não estiver dentro de uma classe é considerado um script**. Um arquivo Groovy pode conter uma ou mais classes que não precisam ter o mesmo nome do arquivo que as contém. O arquivo pode conter código de scripting além das classes.
- **É possível utilizar a palavra chave "*this*" ao invés de métodos estáticos**.
- **Métodos e classes são públicos por default**.
- **Classes internas (*inner class*) não são suportadas no momento.** Na maioria dos casos são usadas *closures* para substituí-las.
- **A cláusula *throws* na assinatura de um método não é checada pelo compilador Groovy.** Em Groovy não há diferença entre exceções checadas (*checked*) e não checadas (*unchecked*).
- **Não ocorrem erros de compilação quando se usa membros não definidos ou são passados argumentos do tipo errado.** Ao invés disse é lançada a exceção *MissingMethodException* em tempo de execução, pois Groovy é uma linguagem dinâmica.
- **No Groovy não existem tipos primitivos, apenas objetos**. Os tipos primitivos do Java são convertidos para suas respectivas classes encapsuladoras de forma transparente para o programador.
- **Métodos *getters* e *setters* criados de forma dinâmica**. Na declaração de *beans* em Groovy não é necessário criar os códigos *getters* e *setters*, a menos que se queira sobrescrever um desses métodos de forma customizada.
- **Evolução das expressões booleanas**. Com essa evolução, mostrada no Quadro 4, as expressões ficam mais simples, conforme o exemplo:

```
    if (stringExemplo != null && !stringExemplo.isEmpty()) {...}
//código Java
    if (stringExemplo()) {…} //código Groovy
```

- **Instrução switch aceita qualquer objeto.** Os comandos "*case*" aceitam qualquer objeto que implemente o método *isCase()*, como detalhado no Quadro 5.

Tipo da Expressão	Condição para ser verdadeira
Boolean	Verdadeira (true)
Collection	Não vazia
Character	Valor diferente de zero
CharSequence	Tamanho > 0
Enumeration	Ter mais elementos para enumerar
Iterator	Ter próximo elemento
Number	Valor double diferente de zero
Map	Não vazio
Matcher	Ter ao menos uma correspondência
Object[]	Tamanho > 0
Qualquer outro tipo	Não nulo

Quadro 4 – Tratamento de expressões booleanas
Fonte: Adaptado de Seifeddine (2009).

Classe	Condição para ser verdadeira
Object	a == b
Class	a instanceof b
Collection	b.contains(a)
Range	b.contains(a)
Pattern	b matches in a?
String	a == b
Closure	b.call(a)

Quadro 5 – Classes que implementam isCase(b) na GDK para switch(a)
Fonte: Adaptado de Seifeddine (2009).

Além de apresentar algumas diferenças em relação à linguagem Java, Groovy possui alguns recursos extras, como:

- ***Closures***. Uma *closure* em Groovy é um pedaço de código que é definido para ser executado mais adiante. Exemplo:

```
def imprimirSoma = {a, b -> print a+b}
imprimirSoma(1,2) //imprime 3
```

  O símbolo "->" é opcional para definições de menos de dois parâmetros. Em *closures* de somente um parâmetro pode ser usado o "*it*':

```
def imprimir = {print it}
imprimir("Testando clausula Groovy!")
```

- Sintaxe nativa para listas e mapas.
- Suporte a *Groovy Markup* e *GPath*.
- Suporte nativo para expressões regulares.
- Interação de polimorfismo
- Tipos dinâmicos e estáticos são suportados, então é possível omitir tipos de declarações em métodos, campos e variáveis.

- É possível incluir expressões dentro de strings.
- Muitos métodos de ajuda foram incluídos na JDK.
- Sintaxe simples para escrita de beans e adição de eventos *listeners*.
- **Navegação segura usando o operador "?." para impedir que sejam lançadas exceções do tipo *NullPointerException*.** Exemplo:

```
Pessoa pessoa = Pessoa.find("Maria")
String cep = pessoa?.endereco?.cep
```

Neste caso a string CEP será nula se pessoa ou endereço forem nulos e não ocorrerá lançamento de *NullPointerException*.

- **Operador Elvis**. Consiste em um encurtamento do operador ternário Java. Exemplo:

```
String nomePai = usuario.nomePai != null ? usuario.nomePai : "Não consta"; //código Java
String nomePai = usuario.nomePai ?: "Não consta" //código Groovy
```

3.4 SUPORTE AO GROOVY

3.4.1 JSF 2.0

É possível utilizar a linguagem Groovy em aplicações JSF. Para isto basta mudar o sufixo dos códigos de *.java* para *.groovy* e utilizar um compilador Groovy para compila-los para um arquivo .class. JSF lida com arquivos *.class*, não importando como eles foram gerados.

3.4.2 Spring Web MVC

O Spring 3.0 suporta as linguagens dinâmicas JRuby, Groovy e BeanShell. Para definir beans de código dessas linguagens é utilizada a tag <lang:language/> na configuração

do XML. No caso do Groovy o elemento utilizado é <lang:groovy/>, conforme exemplo seguinte, disponível na Documentação de Referência do Spring 3.0 (Johnson et al., 2011).

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:lang="http://www.springframework.org/schema/lang"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/lang
http://www.springframework.org/schema/lang/spring-lang-3.0.xsd">
    <!-- this is the bean definition for the Groovy-backed Messenger
implementation -->
    <lang:groovy id="messenger" script-source="classpath:Messenger.groovy">
        <lang:property name="message" value="I Can Do The Frug" />
    </lang:groovy>
    <!-- an otherwise normal bean that will be injected by the Groovy-
backed Messenger -->
    <bean id="bookingService" class="x.y.DefaultBookingService">
        <property name="messenger" ref="messenger" />
    </bean>
</beans>
```

3.4.3 Grails

O framework Grails é totalmente baseado na linguagem Groovy.

3.5 SUPORTE A AJAX

O uso de *Asynchronous JavaScript and XML* (Ajax) atualmente é considerado essencial para desenvolver aplicações web mais dinâmicas, criativas e competitivas. A principal vantagem oferecida pelo Ajax é a redução de dados trafegados pela rede, evitando que o usuário precise aguardar a página ser recarregada a cada interação com o servidor.

3.5.1 JSF 2.0

JSF 2.0 possui suporte embutido a Ajax, com uma biblioteca JavaScript padrão. A tag <f:ajax> é utilizada para atribuir um comportamento Ajax a um componente. Exemplo:

```
<h:inputText value="#{usuarioBean.nome}">
        <f:ajax event="keyup"/>
</h:inputText>
```

Atributo	Descrição
disable	Desabilita a tag se o valor for true
event	O evento que dispara a requisição Ajax.
execute	Uma lista de componentes separados por espaço que o JSF executa no servidor durante a chamada Ajax. Palavras chaves válidas para o atributo *execute* são: *@this*, *@form*, *@all*, @none. O valor *default* é *@this.*
immediate	Se o valor for *true*, JSF processa as entradas no início do ciclo de vida.
onerror	Uma função JavaScript que JSF invoca na chamada Ajax que resulta em um erro.
onevent	Uma função JavaScript que JSF chama para eventos Ajax. Essa função será chamada três vezes durante o ciclo de vida de uma chamada Ajax com sucesso: *begin*, *complete*, *success.*
listener	Faz a ligação com um método "public void processAjaxBehavior (javax.faces.event.AjaxBehaviorEvent event) throws javax.faces.event.AbortProcessingException". Com isso é possível executar um código Java quando um evento qualquer é disparado.
render	Uma lista de componentes separada por espaço que JSF renderiza no cliente após a chamada Ajax retornar do servidor. Pode-se usar as mesmas palavras chaves do atributo *execute.* O valor default é *@none.*

Quadro 6 – Atributos suportados pela tag <f:ajax>
Fonte: Adaptado de Geary, Horstmann (2010).

3.5.2 Spring

Spring 3.0 simplificou muitas tarefas, como invocação de métodos assíncronos e chamadas Ajax. Essa versão do Spring provê suporte oficial para Ajax remoto com *JavaScript Object Notation* (JSON) como parte do Spring MVC. Conforme Donald (2010), isso inclui suporte para gerar respostas JSON e ligar requisições JSON usando o modelo de programação @Controller do Spring MVC.

Segundo Tarantelli (2002), JSON é um formato leve para intercâmbio de dados computacionais que se tornou uma alternativa para XML em Ajax, oferecendo como

vantagem o fato de ser muito mais fácil escrever um analisador JSON. Em JavaScript é possível analisar um arquivo JSON usando a função eval().

As Figuras 4 e 5 mostram um exemplo de um formulário que verifica se o nome preenchido pelo usuário em um campo está disponível e caso não esteja exibe uma mensagem de erro e permanece desativado até que seja informado um nome disponível.

De acordo com a documentação de referência do *Spring Web Flow* (Donald et al.), *Spring Javascript* provê algumas extensões simples do Spring MVC que fazem uso do *Tiles* para desenvolvimento em Ajax. A principal interface para integração das bibliotecas Ajax com o *Spring Web Flow* é a *org.springframework.js.AjaxHandler*. A *SpringJavascriptAjaxHandler* é a biblioteca padrão, que é capaz de detectar uma requisição Ajax submetida por uma API Spring JS do lado cliente e responder apropriadamente no caso em que um redirecionamento é solicitado. Para integrar uma biblioteca Ajax diferente, um *AjaxHandler* customizado pode ser injetado no *FlowHandlerAdapter* ou *FlowController*.

```
01 $(document).ready(function() {
02     // check name availability on focus lost
03     $('#name').blur(function() {
04         checkAvailability();
05     });
06 });
07
08 function checkAvailability() {
09     $.getJSON("account/availability", { name: $('#name').val() }, function(availability) {
10         if (availability.available) {
11             fieldValidated("name", { valid : true });
12         } else {
13             fieldValidated("name", { valid : false,
14                 message : $('#name').val() + " is not available, try " + availability.suggestions });
15         }
16     });
17 }
```

Figura 4 – Utilização de JSON com o padrão jQuery JavaScript para verificar disponibilidade de nome
Fonte: Donald (2010)

```
1 @RequestMapping(value="/availability", method=RequestMethod.GET)
2 public @ResponseBody AvailabilityStatus getAvailability(@RequestParam String name) {
3     for (Account a : accounts.values()) {
4         if (a.getName().equals(name)) {
5             return AvailabilityStatus.notAvailable(name);
6         }
7     }
8     return AvailabilityStatus.available();
9 }
```

Figura 5 – Controlador Spring que informa ao cliente se o nome do usuário está disponível
Fonte: Donald (2010)

3.5.3 Grails

Grails possui suporte a Ajax através das tags:

- *<g:remoteLink>* - cria um link HTML que executa uma requisição e opcionalmente define a resposta em um elemento. Seus atributos estão descritos no Quadro 7.
- *<g:formRemote>* - cria uma tag de formulário que utiliza uma uri remota para executar uma chamada ajax serializando os elementos do formulário, submetendo como uma requisição normal caso javascript não seja suportado. Seus atributos estão descritos no Quadro 8.
- *<g:submitToRemote>* - cria um botão que submete o formulário como uma chamada ajax serializando os campos em parâmetros. Seus atributos estão descritos no Quadro 9.

Para utilizar Ajax além de inserir uma dessas tags para fazer o POST, deve ser adicionada a biblioteca na tag *<head>* da página: *<g:javascript library="prototype"/>*. *Prototype* é a biblioteca padrão do Grails, mas podem ser utilizadas outras através da adição de *plugins*, como *Yahoo UI*, *Dojo Toolkit*, *jQuery* e *Google Web Toolkit*.

Atributo	Descrição
action (opcional)	Nome da *action* a ser usada no link, se não for especificada a action default será usado.
controller (opcional)	Nome do controlador a ser usado no link, se não especificado será usado o controlador atual.
id (opcional)	Id utilizado no link.
fragment (opcional)	O fragmento do link (âncora) a ser usado.
mapping (opcional)	O nome da *URL mapping* a ser usado para reescrever o link.
params (opcional)	Um *map* contendo os parâmetros da requisição.
update (opcional)	Um *map* contendo os elementos para atualização para estados de sucesso (*success*) ou falha (*failure*), ou uma string com o elemento para atualização nos casos em que falhas de eventos seriam ignoradas.
before (opcional)	A função javascript a ser chamada antes da chamada da função remota.
after (opcional)	A função javascript a ser chamada após a chamada função remota.
asynchronous (opcional)	Indica se a chamada deve ser feita de maneira assíncrona ou não (o padrão é true).
method (opcional)	O método a ser usado para executar a chamada (padrão para "post").

Quadro 7 – Atributos suportados pela tag <g:remoteLink>
Fonte: Adaptado de http://www.grails.org/doc/latest/ref/Tags/remoteLink.html.

Atributo	Descrição
name (obrigatório)	O nome do formulário. Este atributo será atribuído ao atributo id se não estiver presente, caso contrário, o valor desse atributo será omitido
url (obrigatório)	A url para submeter como um *map* (contendo valores para controlador, action, id e parâmetros) ou uma string URL.
id (opcional)	O id do formulário renderizado à saída. Se id não está definido, o valor do nome será atribuído.
action (opcional)	A ação a ser executada como um retorno, se não estiver especificada o padrão é a url.
controller (opcional)	Nome do controlador a ser usado no link, se não especificado será usado o controlador atual.
update (opcional)	Um *map* contendo os elementos para atualização para estados de sucesso (*success*) ou falha (*failure*), ou uma string com o elemento para atualização nos casos em que falhas de eventos seriam ignoradas.
before (opcional)	A função javascript a ser chamada antes da chamada da função remota.
after (opcional)	A função javascript a ser chamada após a chamada função remota.
asynchronous (opcional)	Indica se a chamada deve ser feita de maneira assíncrona ou não (o padrão é true).
method (opcional)	O método a ser usado para executar a chamada (padrão para “post”).

Quadro 8 – Atributos suportados pela tag <g:formRemote>
Fonte: Adaptado de http://www.grails.org/doc/latest/ref/Tags/formRemote.html.

Atributo	Descrição
url (opcional)	A url para submeter como um *map* (contendo valores para controlador, action, id e parâmetros) ou uma string URL.
update (opcional)	Um *map* contendo os elementos para atualização para estados de sucesso (*success*) ou falha (*failure*), ou uma string com o elemento para atualização nos casos em que falhas de eventos seriam ignoradas.
before (opcional)	A função javascript a ser chamada antes da chamada da função remota.
after (opcional)	A função javascript a ser chamada após a chamada função remota.
asynchronous (opcional)	Indica se a chamada deve ser feita de maneira assíncrona ou não (o padrão é true).
method (opcional)	O método a ser usado para executar a chamada (padrão para “post”).

Quadro 9 – Atributos suportados pela tag <g:submitToRemote>
Fonte: Adaptado de http://www.grails.org/doc/latest/ref/Tags/submitToRemote.html.

Exemplos:

```
<g:remoteLink action="listarUsuarios" id="1" update="success"
params="[sortBy:'nome',offset:offset]">Listar usuario 1</g:remoteLink>

<g:formRemote name="meuForm" on404="alert('não encontrado!')"
update="testeFormRemote" url="[ controller: 'usuario',
action:'listarUsuarios' ]">
        Usuario Id: <input name="id" type="text"></input>
</g:formRemote>
<div id="testeFormRemote"> Essa div é exibida com o resultado da ação
listarUsuarios</div>

<g:form action="listarUsuarios">
        Nome: <input name="nome" type="text"></input>
        <g:submitToRemote update="testeSubmitToRemote" />
</g:form>
<div id="testeSubmitToRemote">Essa div é exibida com o resultado da ação
listarUsuarios</div>
```

3.6 POPULARIDADE

Para determinar o interesse por cada um desses frameworks foi utilizada a ferramenta Google Trends (disponível no endereço http://www.google.com/trends), que verifica a frequência de pesquisas de termos. Foram utilizados os seguintes termos para pesquisar os frameworks:

- **JSF:** (javaserver faces) | (java server faces) | jsf
- **Grails:** (groovy on rails) | grails
- **Spring Web MVC:** (spring web mvc) | (spring web) | (spring mvc) – Não foi possível pesquisar pelo termo “spring” por ser muito comum.

Na Figura 6 pode-se observar que o JSF é, sem dúvidas, o framework mais popular.

Foi realizada uma nova pesquisa sem o JSF para comparar a popularidade dos frameworks Grails e Spring Web MVC. Como se pode notar na Figura 7, atualmente a popularidade de ambos os frameworks está muito próxima, sendo que o Grails está com uma pequena vantagem.

Figura 6 – Gráfico de buscas realizadas dos frameworks JSF, Grails e Spring Web MVC

Figura 7 – Gráfico de buscas realizadas dos frameworks Grails e Spring Web MVC

3.7 VAGAS DE EMPREGO

Foi realizada uma pesquisa de vagas disponíveis no site da Catho (www.catho.com.br) que exigiam conhecimento de um dos três frameworks apresentados, na data de 22 de junho de 2011. Das 270 vagas oferecidas, 90% exigiam conhecimento em JSF, conforme representado na Figura 8.

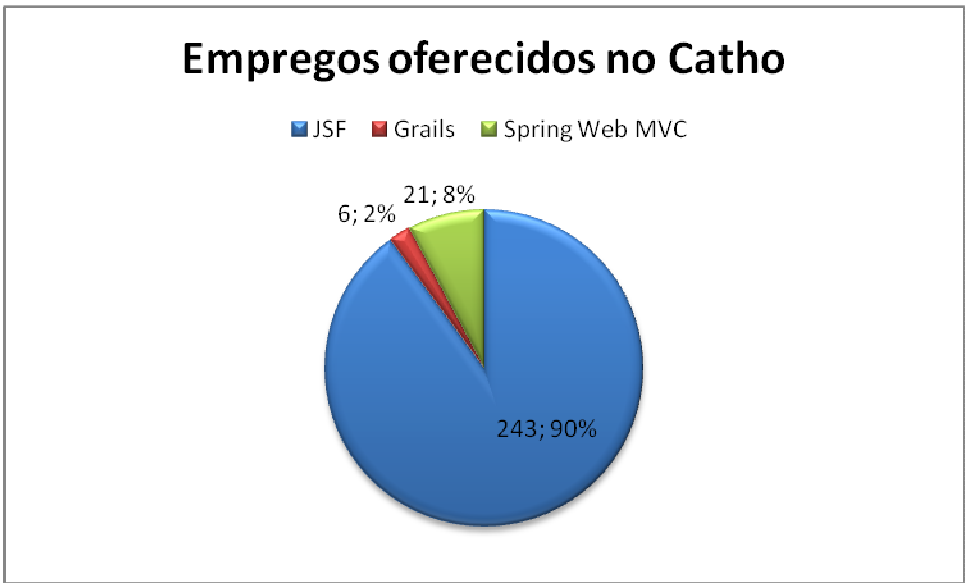

Figura 8 – Empregos disponíveis no site da Catho para cada framework

3.8 LIVROS DISPONÍVEIS

Para verificar a quantidade de livros disponíveis para cada framework foi realizada uma busca no site http://books.google.com. Foram pesquisados livros com idioma inglês, utilizando os seguintes critérios:

- **JSF:** intitle:”java server faces” | intitle:”javaserver faces” | intitle:”jsf” - foram filtrados os resultados que não se referiam ao framework jsf.
- **Grails:** intitle:grails – foram filtrados os resultados que não se referiam ao framework grails.
- **Spring Web MVC:** framework web mvc intitle:spring – foi preciso restringir a pesquisa, pois o termo “spring” é muito comum, tornando inviável filtrar os resultados manualmente.

Os resultados obtidos estão representados na Figura 9.

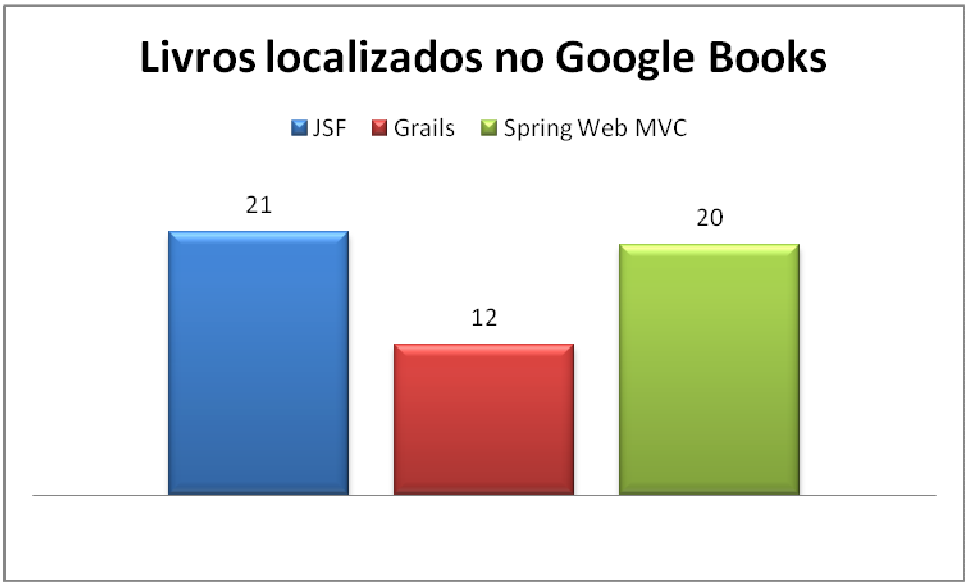

Figura 9 – Literatura disponível para cada framework

3.9 FÓRUNS DE DISCUSSÃO

Os frameworks Spring e Grails possuem fóruns brasileiros exclusivos para discutir assuntos sobre seus conteúdos, disponíveis respectivamente nos endereços http://www.springbrasil.com.br/forum e http://www.grailsbrasil.com.br. Para esclarecer dúvidas do JavaServer Faces o fórum mais popular é o GUJ, disponível em http://www.guj.com.br/forums/list.java.

3.10 ARTIGOS PUBLICADOS

Para comparar a quantidade de artigos publicados referentes a cada framework foi realizada uma pesquisa no site DevMedia, que é o maior site para desenvolvedores de softwares em língua portuguesa e está disponível no endereço http://www.devmedia.com.br.

Para o framework JSF foi pesquisada somente a versão 2.0, ainda assim o número de publicações foi bastante superior aos demais frameworks, conforme pode ser observado na Figura 10.

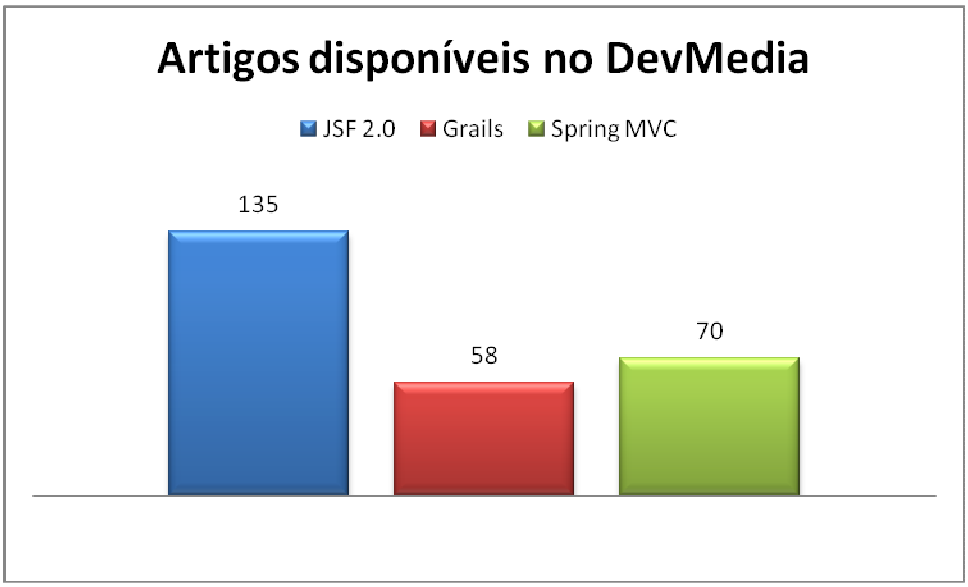

Figura 10 – Artigos disponíveis para cada framework no DevMedia

4 SISTEMA DE CONTROLE FINANCEIRO

Para verificar na prática algumas das características comparadas no capítulo anterior, foi desenvolvido um protótipo de um sistema de controle financeiro pessoal em cada um dos frameworks. Foram implementadas basicamente operações CRUD e validações.

As classes e atributos do sistema estão descritos no diagrama de classes da Figura 11. Para melhor compreensão do sistema, as classes estão descritas a seguir:

- Usuario: representa os usuários do sistema.
- Categoria: crédito e débito.
- Grupo: moradia, lazer, alimentação, educação...
- Conta: aluguel, financiamento de imóvel, ingresso de cinema, compras no supermercado, mensalidade do curso inglês...
- Lançamento: registro de débitos ou créditos de uma determinada conta realizados pelo usuário.

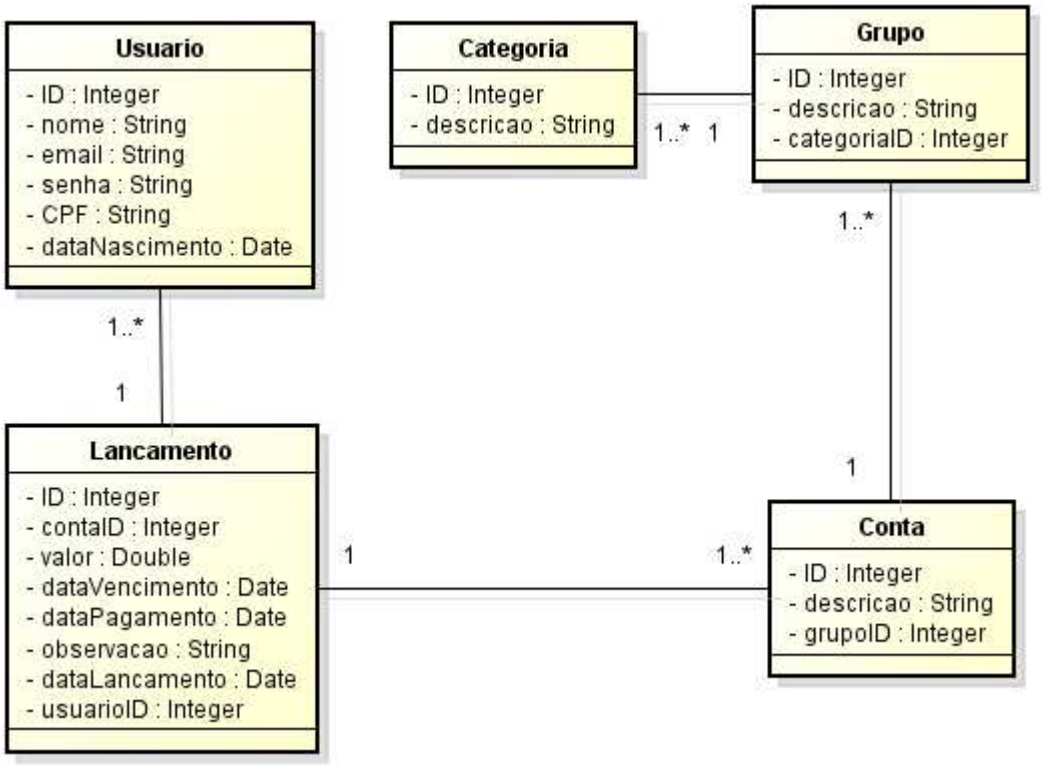

Figura 11 – Diagrama de Classes do Sistema de Controle Financeiro

Os casos de desse sistema estão representados na Figura 12.

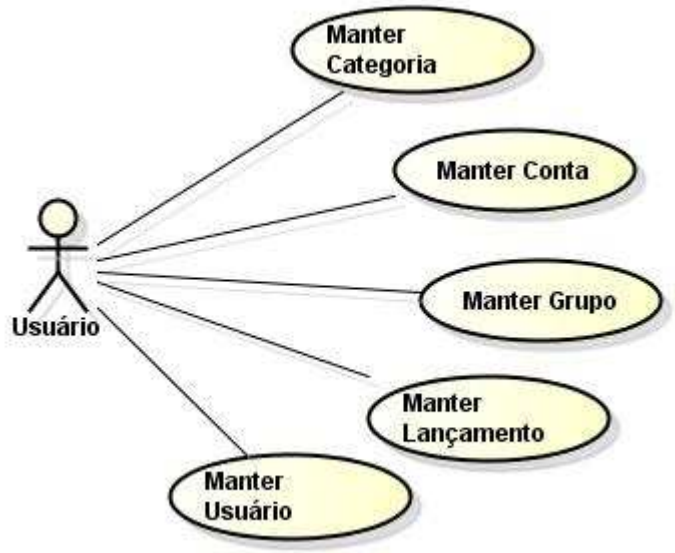

Figura 12 – Diagrama de Casos de Usos do Sistema de Controle Financeiro

A ferramenta NetBeans IDE 7.0 foi utilizada para desenvolvimento das três aplicações. No protótipo em Grails foi utilizada a linguagem Groovy e nos demais foi utilizada a linguagem Java. Para a base de dados foi utilizado o banco de dados PostgreSQL e o framework Hibernate.

Não houve preocupação em criar a mesma interface para todas as aplicações, pois o objetivo principal é avaliar a facilidade oferecida por cada tecnologia. Na aplicação em Grails, por exemplo, foi utilizada a interface padrão, criada de forma automática.

4.1 PROTÓTIPO DESENVOLVIDO EM JSF 2.0

Na aplicação em JSF 2.0 as classes de domínio são criadas como POJOs (*Plain Old Java Object*) com as anotações do Hibernate, conforme exemplificado na Figura 13. Para a comunicação com o banco de dados foram criadas classes DAO (*Data Access Object*), como representado na Figura 14. A *SessionFactory* utilizada pelo Hibernate na classe DAO é definida na classe HibernateUtil (Figura 15). A conexão com o banco de dados é configurada

no arquivo hibernate.cfg.xml. Esse arquivo deve conter ainda o mapeamento das classes, conforme Figura 16.

Categoria.java

```
1  package modelo;
2
3  import java.io.Serializable;
4  import javax.persistence.Entity;
5  import javax.persistence.GeneratedValue;
6  import javax.persistence.GenerationType;
7  import javax.persistence.Id;
8  import javax.persistence.SequenceGenerator;
9  import javax.persistence.Table;
10
11 @Entity
12 @Table(name="categoria")
13 @SequenceGenerator(name = "SEQ_ID", sequenceName = "SEQ_ID")
14 public class Categoria implements Serializable{
15
16     @Id
17     @GeneratedValue(strategy = GenerationType.AUTO, generator = "SEQ_ID")
18     private Integer id;
19     private String descricao;
20
21     public Integer getId() {
22         return id;
23     }
24     public void setId(Integer id) {
25         this.id = id;
26     }
27     public String getDescricao() {
28         return descricao;
29     }
30     public void setDescricao(String descricao) {
31         this.descricao = descricao;
32     }
33
34 }
```

Figura 13 – Classe de domínio Categoria do protótipo em JSF 2.0

```
package dao;

import modelo.Categoria;
import java.util.List;
import org.hibernate.Criteria;
import org.hibernate.Session;
import org.hibernate.Transaction;
import utils.HibernateUtil;

public class CategoriaDaoImp implements CategoriaDao{

    @Override
    public void salvar(Categoria categoria){
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        Transaction t = session.beginTransaction();
        session.save(categoria);
        t.commit();
    }

    @Override
    public Categoria getCategoria(Integer id) {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        return (Categoria) session.load(Categoria.class, id);
    }

    @Override
    public List<Categoria> listar(){
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        Transaction t = session.beginTransaction();
        Criteria select = session.createCriteria(Categoria.class);
        List lista = select.list();
        t.commit();
        return lista;
    }

    @Override
    public void remover(Categoria categoria) {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        Transaction t = session.beginTransaction();
        session.delete(categoria);
        t.commit();
    }

    @Override
    public void atualizar(Categoria categoria) {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        Transaction t = session.beginTransaction();
        session.update(categoria);
        t.commit();
    }

}
```

Figura 14 – Classe CategoriaDaoImp do protótipo em JSF 2.0

```java
package utils;

import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.SessionFactory;

public class HibernateUtil {
    private static final SessionFactory sessionFactory;

    static {
        try {
            sessionFactory = new AnnotationConfiguration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            System.err.println("Falha na criação da SessionFactory! " + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

Figura 15 – Classe HibernateUtil do protótipo em JSF 2.0

HibernateUtil.java | hibernate.cfg.xml | web.xml

Projeto | XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.current_session_context_class">
        org.hibernate.context.ThreadLocalSessionContext
     </property>
    <property name="hibernate.dialect">
        org.hibernate.dialect.PostgreSQLDialect
    </property>
    <property name="hibernate.connection.driver_class">
        org.postgresql.Driver
    </property>
    <property name="hibernate.connection.url">
        jdbc:postgresql://localhost:5432/controleFinanceiro
    </property>
    <property name="hibernate.connection.username">
        postgres
    </property>
    <property name="hibernate.connection.password">
        postgres
    </property>
    <property name="hibernate.show_sql">
        true
    </property>
    <mapping class="modelo.Categoria"/>
    <mapping class="modelo.Conta"/>
    <mapping class="modelo.Grupo"/>
    <mapping class="modelo.Lancamento"/>
    <mapping class="modelo.Usuario"/>
  </session-factory>
</hibernate-configuration>
```

Figura 16 – Arquivo hibernate.cfg.xml do protótipo em JSF 2.0

As classes controladoras são criadas através da anotação @ ManagedBean e contêm os métodos necessários para realizar a comunicação da camada de visão com o banco de dados, realizando as operações requisitadas nas páginas JSF e encaminhando o usuário para a página adequada. As Figura 17 e 18 representam uma dessas classes criadas no protótipo.

```
package controle;
import dao.CategoriaDao;
import dao.CategoriaDaoImp;
import java.io.IOException;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import javax.faces.context.FacesContext;
import javax.faces.event.ActionEvent;
import javax.faces.model.DataModel;
import javax.faces.model.ListDataModel;
import javax.faces.model.SelectItem;
import modelo.Categoria;
import org.primefaces.context.RequestContext;
@ManagedBean
@SessionScoped
public class CategoriaBean implements Serializable {
    private Categoria categoria;
    private DataModel listaCategorias;
    private ArrayList<SelectItem> itens;
    private boolean alteracao;
    public CategoriaBean(){
        categoria = new Categoria();
    }
    public Categoria getCategoria() {
        return categoria;
    }
    public void setCategoria(Categoria categoria) {
        this.categoria = categoria;
    }
    public DataModel getListaCategorias() {
        List<Categoria> lista = new CategoriaDaoImp().listar();
        listaCategorias = new ListDataModel(lista);
        return listaCategorias;
    }
```

Figura 17 – Classe CategoriaBean do protótipo em JSF 2.0 (parte 1)

```
    public ArrayList<SelectItem> getItens() {
        List<Categoria> categorias = new CategoriaDaoImp().listar();
        itens = new ArrayList<SelectItem>();
        SelectItem item = new SelectItem();
        item.setLabel("");
        item.setValue(0+"");
        itens.add(0, item);
        for (Categoria c: categorias) {
            item = new SelectItem();
            item.setLabel(c.getDescricao());
            item.setValue(c.getId()+"");
            itens.add(item);
        }
        return itens;
    }
    public void prepararAdicionar(ActionEvent actionEvent){
        categoria = new Categoria();
        alteracao = false;
    }
    public void adicionar(ActionEvent actionEvent){
        CategoriaDao dao = new CategoriaDaoImp();
        dao.salvar(categoria);
    }
    public void prepararAlterar(ActionEvent actionEvent){
        categoria = (Categoria)(listaCategorias.getRowData());
        alteracao = true;
    }
    public void alterar(ActionEvent actionEvent){
        CategoriaDao dao = new CategoriaDaoImp();
        dao.atualizar(categoria);
    }
    public void excluir(){
        Categoria cat = (Categoria)(listaCategorias.getRowData());
        CategoriaDao dao = new CategoriaDaoImp();
        dao.remover(cat);
        try {
            FacesContext.getCurrentInstance().getExternalContext().redirect("gerenciaCategoria.xhtml");
        } catch (IOException ex) {
            Logger.getLogger(CategoriaBean.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
    public void salvar(ActionEvent actionEvent){
        RequestContext requestContext = RequestContext.getCurrentInstance();
        requestContext.addCallbackParam("sucesso", true);
        if (alteracao){
            alterar(actionEvent);
        } else adicionar(actionEvent);
    }
    public void gerenciarCategoria(){
        try {
            FacesContext.getCurrentInstance().getExternalContext().redirect("gerenciaCategoria.xhtml");
        } catch (IOException ex) {
            Logger.getLogger(CategoriaBean.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

}
```

Figura 18 – Classe CategoriaBean do protótipo em JSF 2.0 (parte 2)

A visão é implementada através de páginas xhtml. Foi utilizada a biblioteca de componentes JSF *PrimeFaces* para criar interfaces mais elegantes (Figura 19). Essa biblioteca está disponível no endereço http://www.primefaces.org/downloads.html, basta adicioná-la às bibliotecas do projeto e declará-la na página (xmlns:p="http://primefaces.prime.com.tr/ui").

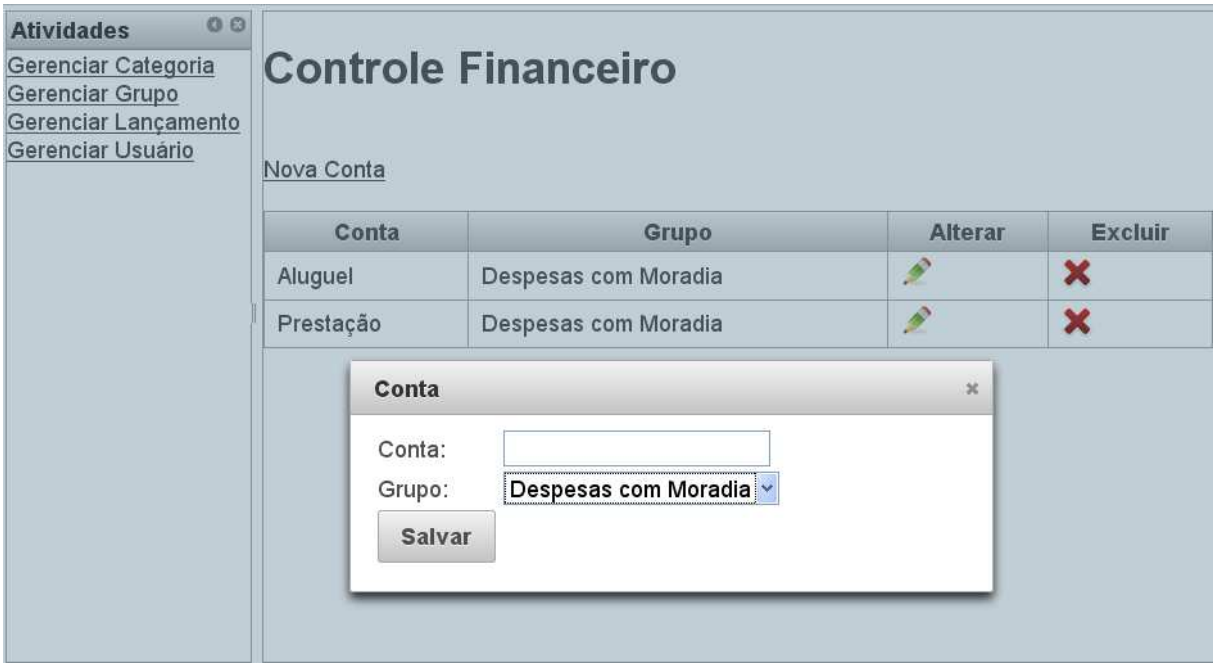

Figura 19– Interface do protótipo em JSF 2.0 criada com PrimeFaces

O arquivo web.xml do protótipo foi configurado como mostrado na Figura 20.

A validação no framework JSF pode ser feita na própria página, conforme exemplificado no código seguinte, ou através de anotações do *Hibernate Validator*, como mostrado nas Figuras 21 e 22.

```
<h:inputText id="desc" value="#{categoriaBean.categoria.descricao}"
required="true" requiredMessage="o campo descrição é obrigatório."/>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" 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/web-app_3_0.xsd">
    <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>
    <context-param>
        <param-name>primefaces.THEME</param-name>
        <param-value>aristo</param-value>
    </context-param>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>gerenciaLancamentos.xhtml</welcome-file>
    </welcome-file-list>
</web-app>
```

Figura 20 – Arquivo web.xml do protótipo em JSF 2.0

```java
@Id
@GeneratedValue(strategy = GenerationType.AUTO, generator = "SEQ_ID")
private Integer id;
@Pattern (regexp="[a-zA-Z ]+", message="Foram informados caracteres inválidos no campo nome.")
@Length (min=6, max=50, message="O nome deve conter de 6 a 50 caracteres.")
private String nome;
@NotEmpty (message="O campo email é obrigatório.")
@Email (message="O email informado não está no formato adequado.")
private String email;
@Length (min=6, max=12, message="A senha deve conter de 6 a 12 caracteres.")
private String senha;
@Pattern (regexp="[0-9]+", message="O cpf deve conter somente dígitos.")
@Length (min=11, max=11, message="O cpf deve conter 11 digitos.")
private String cpf;
@Temporal(javax.persistence.TemporalType.DATE)
@Past (message="A data de nascimento deve ser inferior à data atual")
@NotNull (message="O campo data é obrigatório.")
@Column (name="data_nascimento")
private Date dataNascimento;
```

Figura 21 – Validação dos atributos da classe Usuario utilizando anotações do Hibernate Validator

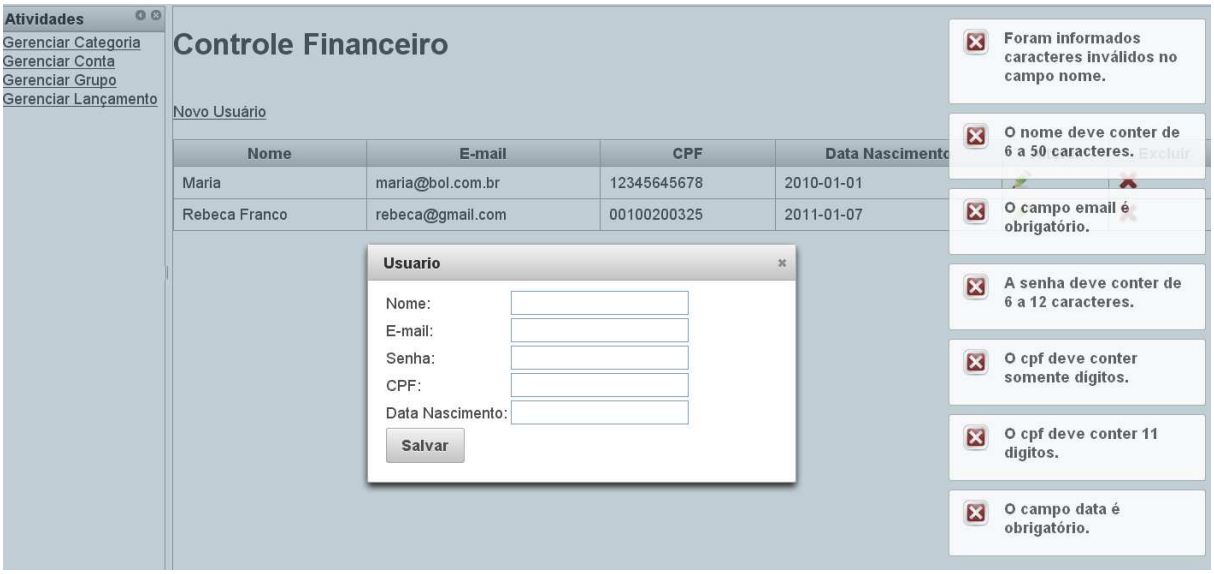

Figura 22 – Mensagens exibidas pelo *Hibernate Validator*

O Apêndice A contém o código fonte da página xhtml da Figura 22.

4.2 PROTÓTIPO DESENVOLVIDO EM SPRING WEB MVC

Na aplicação em Spring Web MVC as classes de domínio (entidades) são criadas da mesma maneira que na aplicação em JSF. Também foi utilizada a validação do *Hibernate Validator*, portanto essas classes ficaram exatamente iguais às do protótipo anterior.

Para o acesso aos dados foram criadas classes DAOs conforme a Figura 23.

As classes de controle tratam as requisições do cliente, controlando o fluxo a ser percorrido na chamada de uma determinada visão. A Figura 24 mostra a implementação da classe CategoriaController. As variáveis inclusas no ModelMap podem ser acessadas pela visão da seguinte maneira, por exemplo:

```
<c:forEach items="${categorias}" var="categoria">
```

As propriedades de conexão com o banco de dados e do Hibernate foram definidas no arquivo config.properties, conforme Figura 25.

O arquivo applicationContext.xml, exibido na Figura 26, contém mapeamentos Spring que configuram o contexto mínimo da aplicação, como *hibernate session factory bean*, *data source*, *transaction manager*, etc.

Para definir o caminhos das visões e a tecnologia utilizada no projeto Spring Web MVC é utilizado o arquivo dispatcher-servlet.xml (que trata-se de um Spring *DispatcherServlet*, que pode ter qualquer outro nome que será seguido do "-servlet" no arquivo xml), conforme exemplificado na Figura 27. Esse servlet é definido no arquivo web.xml, juntamente com os demais servlets utilizados no projeto (Figura 28).

Como boa prática recomendada pelo Spring, as páginas devem ficar dentro do diretório WEB-INF para que não possam ser acessadas diretamente através da URL. Os Apêndices B, C e D contêm algumas das páginas criadas neste protótipo.

```
package com.springmvc.dao;

import com.springmvc.modelo.Categoria;
import java.util.List;
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.orm.hibernate3.HibernateTemplate;
import org.springframework.stereotype.Repository;

@Repository
public class CategoriaDaoImp implements CategoriaDao{

    private HibernateTemplate hibernateTemplate;

    @Autowired
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.hibernateTemplate = new HibernateTemplate(sessionFactory);
    }

    @Override
    public void salvar(Categoria categoria){
        hibernateTemplate.saveOrUpdate(categoria);
    }

    @Override
    public Categoria getCategoria(Integer id) {
       return (Categoria) hibernateTemplate.find("from Categoria where id = "+id).get(0);
    }

    @Override
    public List<Categoria> listar(){
        return hibernateTemplate.loadAll(Categoria.class);
    }

    @Override
    public void remover(Categoria categoria) {
        hibernateTemplate.delete(categoria);
    }

}
```

Figura 23 – Classe CategoriaDaoImp do protótipo em Spring Web MVC

```java
package com.springmvc.controle;
import com.springmvc.dao.CategoriaDao;
import com.springmvc.modelo.Categoria;
import javax.validation.Valid;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
@Controller
@RequestMapping("/categoria/*")
public class CategoriaController {
    @Autowired
    private CategoriaDao categoriaDao;
    @RequestMapping(value = "/listar", method = RequestMethod.GET)
    public String listar(ModelMap modelMap) {
        modelMap.addAttribute("categorias", categoriaDao.listar());
        return "listarCategorias";
    }
    @RequestMapping(value = "/categoria/{id}", method = RequestMethod.DELETE)
    public String apagar(@PathVariable("id") Integer id) {
        categoriaDao.remover(categoriaDao.getCategoria(id));
        return "redirect:/categoria/listar";
    }
    @RequestMapping(value = "/criar", method = RequestMethod.POST)
    public String criar(@Valid Categoria categoria, BindingResult result) {
        if (result.hasErrors()) return "criarCategoria";
        categoriaDao.salvar(categoria);
        return "redirect:/categoria/listar";
    }
    @RequestMapping(method = RequestMethod.PUT)
    public String atualizar(@Valid Categoria categoria, BindingResult result) {
        if (result.hasErrors()) return "alterarCategoria";
        categoriaDao.salvar(categoria);
        return "redirect:/categoria/listar";
    }
    @RequestMapping(value = "/form", method = RequestMethod.GET)
    public String form(ModelMap modelMap) {
        modelMap.addAttribute("categoria", new Categoria());
        return "criarCategoria";
    }
    @RequestMapping(value = "/{id}/form", method = RequestMethod.GET)
    public String atualizarForm(@PathVariable("id") Integer id, ModelMap modelMap) {
        modelMap.addAttribute("categoria", categoriaDao.getCategoria(id));
        return "alterarCategoria";
    }
}
```

Figura 24 – Classe CategoriaController do protótipo em Spring Web MVC

```
#################### Configurações JDBC ############################
jdbc.driverClassName=org.postgresql.Driver
jdbc.url=jdbc:postgresql://localhost:5432/controleFinanceiro
jdbc.username=postgres
jdbc.password=postgres

#################### Configurações Hibernate ######################
hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
hibernate.show_sql=true
hibernate.generate_statistics=true
```

Figura 25 – Arquivo config.properties do protótipo em Spring Web MVC

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd
       http://www.springframework.org/schema/mvc
       http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
    <context:component-scan base-package="com.springmvc"/>
    <mvc:annotation-driven />
    <bean id="propertyConfigurer"
          class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"
          p:location="/WEB-INF/config.properties" />
    <bean id="dataSource"
          class="org.springframework.jdbc.datasource.DriverManagerDataSource"
          p:driverClassName="${jdbc.driverClassName}"
          p:url="${jdbc.url}"
          p:username="${jdbc.username}"
          p:password="${jdbc.password}" />
   <bean id="sessionFactory"
         class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
         <property name="dataSource" ref="dataSource"></property>
         <property name="hibernateProperties">
             <props>
                <prop key="hibernate.dialect">${hibernate.dialect}</prop>
                <prop key="hibernate.show_sql">${hibernate.show_sql}</prop>
             </props>
         </property>
        <property name="packagesToScan">
            <list>
                <value>com.springmvc.modelo</value>
            </list>
        </property>
   </bean>
   <bean id="transactionManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory"><ref local="sessionFactory"/></property>
   </bean>
   <tx:annotation-driven transaction-manager="transactionManager"/>
</beans>
```

Figura 26 – Arquivo applicationContext.xml do protótipo em Spring Web MVC

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <bean id="viewResolver"
         class="org.springframework.web.servlet.view.InternalResourceViewResolver"
         p:prefix="/WEB-INF/jsp/"
         p:suffix=".jsp" />

</beans>
```

Figura 27 – Arquivo dispatcher-servlet.xml do protótipo em Spring Web MVC

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" 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/web-app_3_0.xsd">
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/applicationContext.xml</param-value>
    </context-param>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>2</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
    <servlet>
        <servlet-name>default</servlet-name>
        <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>default</servlet-name>
        <url-pattern>/static/*</url-pattern>
    </servlet-mapping>
    <!-- Habilitar o suporte REST do Spring 3.0 -->
    <filter>
        <filter-name>httpMethodFilter</filter-name>
        <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
    </filter>
    <!-- Permitir inclusão de um campo oculto para PUT e DELETE -->
    <filter-mapping>
        <filter-name>httpMethodFilter</filter-name>
        <servlet-name>dispatcher</servlet-name>
    </filter-mapping>
    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

Figura 28 – Arquivo web.xml do protótipo em Spring Web MVC

4.3 PROTÓTIPO DESENVOLVIDO EM GRAILS

Para desenvolver essa aplicação em Grails inicialmente foram criadas as classes de domínio, com o mapeamento e validação dos atributos, conforme Figura 29. Não é necessário criar atributos *id*, pois estes são gerados automaticamente pelo Grails.

Usuario.groovy

```
1   class Usuario {
2
3       String email
4       String senha
5       String nomeCompleto
6       String CPF
7       Date dataNascimento
8       static hasMany = [lancamento:Lancamento]
9
10      static constraints = {
11          email (blank:false, email:true, unique:true)
12          senha (blank:false, maxSize:12, minSize:6, password:true)
13          nomeCompleto (blank:false, matches:"[a-zA-Z ]+", maxSize:50, minSize:6)
14          CPF (blank:false, matches:"[0-9]+", size:11..11)
15          dataNascimento (blank:false, max:new Date())
16      }
17
18  }
19
```

Figura 29 – Classe de domínio Usuario do protótipo em Grails

As classes de controle e as páginas de visualização foram criadas pelo Grails, através da seleção das classes de domínio, clique do mouse com o botão direito e clique na opção “Gerar tudo”, como mostrado na Figura 30. Para cada classe são geradas as páginas de cada operação CRUD, nomeadas como *create.gsp*, *edit.gsp*, *list.gsp* e *show.gsp* (conforme exemplificado nos Apêndices E, F, G e H) e armazenadas em uma pasta com o nome da classe, dentro do diretório *view* (exibido como Visualizações e Layouts no Netbeans). Os controladores são gerados com o nome da classe adicionado da palavra *Controller*, dentro da pasta *controllers* (exibida como Controladores no Netbeans).

A página inicial que foi gerada automaticamente está representada na Figura 31. Ela contém um link para cada um dos controladores do sistema.

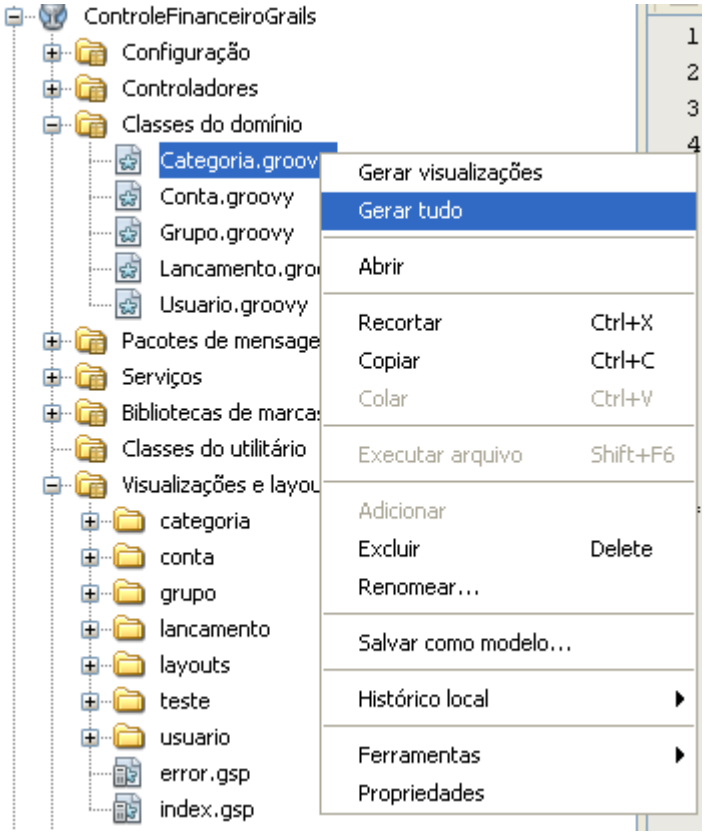

Figura 30 – Opção para gerar controles e visões do sistema automaticamente em Grails

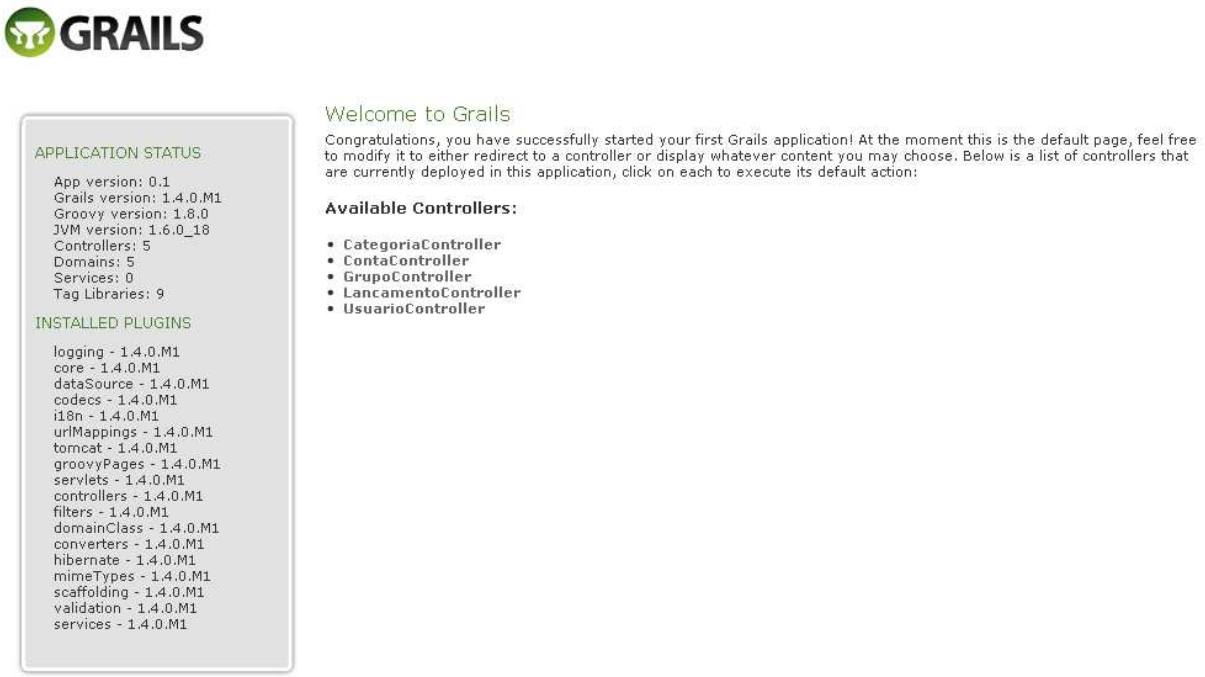

Figura 31 – Página inicial do sistema gerada automaticamente pelo framework Grails

Clicando em um dos controladores, UsuarioController, por exemplo, é aberta uma página que lista os usuários cadastrados e apresenta link para inclusão de usuário (Figura 32). Nesse exemplo não é desejável que a senha seja exibida na listagem, para isso basta editar a página *list.gsp* dentro da pasta usuario e remover as linhas de código que exibem esse atributo. Clicando no Id de um usuário listado, é aberta uma página com as informações desse usuário e opções para editá-lo ou excluí-lo. Os campos do formulário de inclusão e edição já são criados de acordo com o tipo (o campo senha não irá exibir os caracteres digitados e o de data de nascimento já traz as *comboboxs* para escolha do dia, mês e ano, por exemplo) e com a validação definidos na classe de domínio, como exemplificado na Figura 33.

Figura 32 – Página de listagem de usuário gerada automaticamente pelo framework Grails

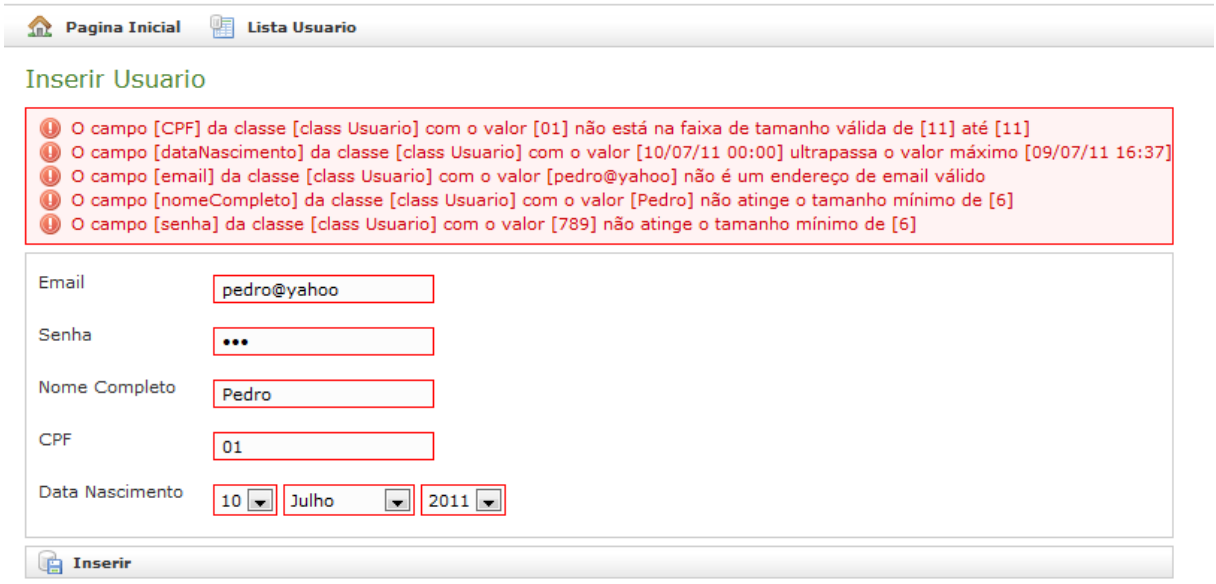

Figura 33 – Página de inclusão de usuário gerada automaticamente pelo framework Grails

Nas classes que possuem relacionamento, o combobox da página exibe o id, para que exiba outro campo é necessário incluir o atributo optionValue indicando a informação a ser recuperada da classe, conforme código seguinte, que tem como resultado o conteúdo do campo Grupo exibido na Figuras 34.

```
<g:select name="grupo.id" from="${Grupo.list()}" optionKey="id"
value="${contaInstance?.grupo?.id}" optionValue="${{it.descricao}}"  />
```

Figura 34 – Exibição da descrição do grupo na combobox

Até a versão 1.4.0.M1 (a mais recente até o momento), o arquivo de propriedades de mensagens (*messages_pt_BR.properties*, dentro da pasta i18n, exibida como Pacotes de Mensagens no NetBeans) ainda não vem com todas as mensagens utilizadas no sistema, por isso as tela são geradas com várias palavras em inglês. Para que isso não ocorra, basta copiar as demais mensagens do arquivo *messages.properties* para o *messages_pt_BR.properties* e traduzi-las.

As páginas e controladores gerados automaticamente podem ser alterados para customização do sistema. Anteriormente, se uma página era alterada e precisasse ser gerada novamente devido a alguma modificação na classe era necessário refazer as alterações de layout. A partir da versão 1.4.0.M1 se tornou possível a customização dos templates gerados pelo Grails. Para isso deve-se executar o comando "*grails install-templates*" na pasta onde se encontra o projeto. Esse comando irá criar uma pasta *templates* no diretório src, permitindo que o usuário altere o que desejar. Dessa forma, por exemplo, as páginas geradas pelo *scaffolding* seguirão o padrão definido pelo usuário.

Inicialmente os dados inseridos no sistema são armazenados em memória. Para integrar a aplicação com o PostgreSQL é necessário adicionar o *driver* da versão do banco de dados utilizada na pasta lib (exibida no NetBeans como Bibliotecas) e alterar no arquivo DataSource.groovy (dentro da pasta conf, exibida como Configuração no NetBeans) os atributos *driverClassName*, *username*, *password* e *url*, conforme exemplificado na Figura 35. Como o Grails utiliza o GORM, que encapsula o Hibernate, por *default* as operações executadas pelo banco não são exibidas. Caso o desenvolvedor deseje visualizá-las basta acrescentar a linha "*logSql = true*". As tabelas podem ser geradas pela aplicação se o valor do atributo *dbCreate* for definido como *create* ou *create-drop*. O script da tabela gerada para a classe Usuario está exibido na Figura 36.

Na classe groovy é possível atribuir valores a propriedades de uma classe a partir de algum evento do banco de dados. Nesse protótipo foram utilizados os métodos *beforeInsert()* e *beforeUpdate()* para definir as datas de inserção e alteração de lançamentos, conforme exemplificado na Figura 37.

DataSource.groovy

```
dataSource {
    pooled = true
    logSql = true
    driverClassName = "org.postgresql.Driver"
    username = "postgres"
    password = "postgres"
}
hibernate {
    cache.use_second_level_cache = true
    cache.use_query_cache = true
    cache.provider_class = 'net.sf.ehcache.hibernate.EhCacheProvider'
}
// environment specific settings
environments {
    development {
        dataSource {
            dbCreate = "create-drop" // one of 'create', 'create-drop','update'
            url = "jdbc:postgresql://localhost:5432/controleFinanceiroGrails"
        }
    }
    test {
        dataSource {
            dbCreate = "update"
            url = "jdbc:postgresql://localhost:5432/controleFinanceiroGrails"
        }
    }
    production {
        dataSource {
            dbCreate = "update"
            url = "jdbc:postgresql://localhost:5432/controleFinanceiroGrails"
        }
    }
}
```

Figura 35 – Configuração do arquivo dataSource para integração com o PostgreSQL

Painel SQL

```
-- Table: usuario

-- DROP TABLE usuario;

CREATE TABLE usuario
(
  id bigint NOT NULL,
  "version" bigint NOT NULL,
  cpf character varying(11) NOT NULL,
  data_nascimento timestamp without time zone NOT NULL,
  email character varying(255) NOT NULL,
  nome_completo character varying(50) NOT NULL,
  senha character varying(12) NOT NULL,
  CONSTRAINT usuario_pkey PRIMARY KEY (id),
  CONSTRAINT usuario_email_key UNIQUE (email)
)
WITH (
  OIDS=FALSE
);
ALTER TABLE usuario OWNER TO postgres;
```

Figura 36 – Script gerado automaticamente para criação da tabela usuário

list.gsp | Lancamento.groovy

```
1   class Lancamento {
2
3       Double valor
4       Date dataVencimento
5       Date dataPagamento
6       String observacao
7       Date dataLancamento
8       Date dataAlteracao
9
10      static belongsTo = [usuario:Usuario, conta:Conta]
11
12      static constraints = {
13          valor (min:0.01d)
14          dataPagamento (max:new Date())
15          dataLancamento (display:false, nullable:true)
16          dataAlteracao (display:false, nullable:true)
17      }
18
19      def beforeInsert(){
20          dataLancamento = new Date()
21      }
22
23      def beforeUpdate(){
24          dataAlteracao = new Date()
25      }
26
27  }
```

Figura 37 – Utilização dos métodos beforeInsert() e beforeUpdate()

5 CONSIDERAÇÕES FINAIS

A utilização dos frameworks facilita muito o desenvolvimento do software, permitindo que sejam construídas aplicações muito mais robustas, com maior qualidade e em menor tempo, o que é essencial para a exigência e competitividade existentes no mercado de trabalho.

JSF 2.0, Spring Web MVC e Grails, conforme demonstrado nesse trabalho, são frameworks fáceis de ser utilizados e que oferecem muitos recursos ao desenvolvedor.

JSF é definido como o framework padrão pela Oracle e pela JCP. É construído sob o conceito de componentes, que são a base para a construção da interface com o usuário, e dirigido a eventos. Tem como principais objetivos a facilidade de utilização. Possui vários pontos de extensão (como conversores, validadores e *listeners*) e componentes prontos. Dos frameworks apresentados é o mais utilizado no mercado.

Spring Web MVC é um módulo do projeto Spring Framework, cujas principais vantagens são o uso do IoC do Spring e a facilidade de renderização de diversos tipos de saída, como JSP, Tiles, Velocity, Freemaker, Excel e PDF.

O framework Grails é uma oportunidade de melhorar a produtividade mantendo a plataforma Java, apenas migrando para a linguagem Groovy. Essa linguagem possui sintaxe reduzida e clara e maior poder funcional. Por ser muito semelhante à Java, é facilmente aprendida por estes desenvolvedores. O uso dos frameworks Hibernate e Spring torna-se muito mais simples com Grails, pois não é necessária nenhuma configuração para utilizá-los. As maiores vantagens desse framework são simplificar o aprendizado e reduzir a complexidade de implementação, reduzindo o tempo de desenvolvimento consideravelmente.

Os três frameworks são excelentes. Eles possuem muitas características em comum e também algumas diferenças, resumidas no Quadro 10. A escolha de qual deles utilizar dependerá muito do tipo de aplicação a ser desenvolvida. Dependendo do projeto pode ser interessante até mesmo combinar o uso de mais de um framework.

No desenvolvimento dos protótipos foi possível verificar algumas dificuldades que poderão ser enfrentadas pelos desenvolvedores:

- Spring Web MVC – a configuração de arquivos é bastante trabalhosa (são necessários vários arquivos XML, com cabeçalhos bem extensos), possibilitando a ocorrência de erros cujas causas são difíceis de serem detectadas.

- JSF – para os desenvolvedores habituados a trabalhar com frameworks baseados em ações pode ser um pouco difícil mudar o pensamento para desenvolvimento baseado em componentes.

Grails foi o que se mostrou mais simples e fácil de ser utilizado, exigindo menos esforço e tempo de programação para atingir o mesmo resultado dos demais frameworks. As principais justificativas para isso são: as classes Groovy serem bem mais simples e compactas do que as classes Java e não ser necessário configurar arquivos XML e nem acrescentar arquivos jar (o único necessário foi o driver do PostgreSQL), o que torna o aprendizado muito mais fácil, pois evita grande parte dos erros e dificuldades causados por essas etapas no desenvolvimento com JSF ou Spring Web MVC.

A Tabela 1 demonstra a diferença entre quantidade de linhas necessárias para implementação do protótipo em cada um dos frameworks. Não foram comparadas as quantidades de linhas das páginas porque não foi utilizado o mesmo padrão, o que prejudicaria a análise do resultado.

Como sugestão de trabalhos futuros estão concluir o desenvolvimento dos protótipos, adicionando novas funcionalidades para possibilitar a utilização de todos os recursos demonstrados nesse trabalho e incluir mais frameworks na comparação, como VRaptor, Mentawai e Wicket.

Características	**JSF 2.0**	**Spring Web MVC**	**Grails**
Ajax	Automático	Automático	Automático
Annotations	Sim	Sim	Sim
Baseado em	Componentes (*component-based*)	Ação (*action-based*)	Ação (*action-based*)
CoC	Configuração	Configuração	Convenção
Curva de Aprendizado	Curta	Média	Muito curta
DRY	Sim	Sim	Sim
Free	Sim	Sim	Sim
Front-Controller	Sim	Sim	Sim
IoC	Sim	Sim	Sim
Linguagem	Java	Java	Groovy
MVC	Sim	Sim	Sim
Scaffolding	Classes de domínio podem ser geradas a partir de tabelas e páginas JSF geradas a partir das classes de domínio.	O scaffolding pode ocorrer a partir de uma base de dados, Entidades JPA ou Java Beans.	Classes de controle, páginas GSP e tabelas podem ser geradas a partir das classes de domínio.
Suporte a JSF	-	Sim	Sim
Suporte a Spring	Sim	É um módulo do Spring	Sim
Usa XML	Sim	Sim	Não
Validação	Sim	Sim	Sim

Quadro 10 – Comparação entre os frameworks

Tabela 1 – Linhas de código do protótipo de cada um dos frameworks

Tipo de arquivo	JSF 2.0	Spring Web MVC	Grails
Classes de Controle	394	215	315
Classes DAO	233	165	0
Classes de Domínio	227	227	47
Arquivos XML	79	126	0
Total	**933**	**733**	**362**

REFERÊNCIAS

CAELUM, Ensino e Inovação. **FJ-21 Java para desenvolvimento web**. Disponível em: <http:// www.caelum.com.br/apostilas/>. Acesso em: 16 abr. 2011.

COAD, Peter. **Object-oriented patterns**. Communications of the ACM, V. 35, nº 9, p. 152-159, 1992.

CODEHAUS FOUNDATION. **Groovy. An agile dynamic language for the Java Platform**. 2011. Disponível em: <http://groovy.codehaus.org/>. Acesso em: 10 maio 2011.

DONALD, Keith. **Ajax simplifications in Spring 3.0**. Springsource, 2010. Disponível em: <http://blog.springsource.com/2010/01/25/ajax-simplifications-in-spring-3-0/>. Acesso em: 31 maio 2011.

DONALD, Keith et al. **Spring Web Flow Reference Guide**. Disponível em: <http://static.springsource.org/spring-webflow/docs/2.0.x/reference/html/index.html>. Acesso em: 01 jun. 2011.

GEARY, David; HORSTMANN, Cay. **Core JavaServer Faces**. Ann Arbor, Michigan: Prentice Hall, 2010.

GONCALVES, Edson. **Desenvolvendo aplicações web com JSP, Servlets, Java Server Faces, Hibernate, EJB3 Persistence e Ajax**. Rio de Janeiro: Editora Ciência Moderna Ltda., 2007.

HEMRAJANI, Anil. **Agile Java development with Spring, Hibernate and Eclipse**. Sams Publishing, 2006.

INFONOVA. **Agile web development with Groovy & Grails.** Bearing Point, Inc., 2008. Disponível em: < http://pt.scribd.com/doc/14801783/Agile-Web-Development-with-Groovy-Grails>. Acesso em: 09 maio 2011.

JBOSS COMMUNITY. **Hibernate**. Disponível em: <http://www.hibernate.org/>. Acesso em: 12 jun. 2011.

JOHNSON, Ralph E.; WOOLF, B. Type Object. In: “Martin, R.C.; Riehle, D.; Buschmann, F. **Pattern languages of Program Design 3**. Reading-MA, Addison-Wesley, 1998”, p. 47-65.

JOHNSON, Rod et al. **Reference Documentation** – Spring Framework 3.0. 2011. Disponível em: <http://static.springsource.org/spring/docs/3.1.0.M1/spring-framework-reference/pdf/spring-framework-reference.pdf>. Acesso em: 07 maio 2011.

MACHACEK, Jan; VUKOTIC, Aleksa; CHAKRABORTY, Anyrvan; DITT, Jessica. **Pro Spring 2.5**. Rio de Janeiro: Editora Ciência Moderna Ltda., 2009.

MACHADO, Erich. **Inversão de controle**. Blog da Locaweb, 2008. Disponível em: <http://blog.locaweb.com.br/ tecnologia/inversao-de-controle/>. Acesso em: 14 maio 2011.

NUNES, Vinny; Magalhães, Eder. **Aprendendo JSF 2.0 com ScrumToys**. Java Magazine, Edição 78, Ano VII (2010). Disponível em: <http://www.devmedia.com.br/post-16559-JSF-2-0.html>. Acesso em: 28 abr. 2011.

ORACLE**. The Java EE 6 Tutorial**. 2011. Disponível em: <http://download.oracle.com/javaee/6/tutorial/doc/ bnaph.html>. Acesso em: 07 maio 2011.

PITANGA, Talita. **JavaServer Faces: A mais nova tecnologia Java para desenvolvimento WEB.** Disponível em: <http://www.guj.com.br/content/articles/jsf/jsf.pdf>. Acesso em: 30 abr. 2011.

SAAB, Felipe N. **Spring Framework Parte 5 – Spring Web MVC**. Java Simples, 2011. Disponível em: < http://www.javasimples.com.br/spring-2/spring-framework-parte-5-spring-web-mvc/>. Acesso em: 30 abr. 2011.

SEIFEDDINE, Mohamed. **Introduction to Groovy and Grails**. 2009. Disponível em: < http://www.opensourceconnections.com/wp-content/uploads/2010/01/Introduction_to_ Groovy_and_Grails.pdf>. Acesso em: 17 abr. 2011.

SINGH, I. et al. **Designing Enterprise Applications with the J2EE Platform**. Second Edition. 2nd ed. New Jersey, USA: Addison-Wesley, 2002.

SUBRAMANIAM, Venkat. **Programming Groovy**. Dynamic Productivity for the Java Developer. The Pragmatic Programmers, 2008.

SUN MICROSYSTEMS. **JavaBeans**. 1997. Disponível em: <http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html>. Acesso em: 01 jun. 2011.

TARANTELLI, Renato. **JSON**. World of Bit, 2002. Disponível em: <http://worldofbit.com/wob/programacao/json.html>. Acesso em: 22 out. 2011.

TORSTEN. **Programação orientada a aspectos**. 2011. Disponível em: <http://www.inf.pucminas.br/professores/torsten/aulas/aula06.html>. Acesso em: 15 maio 2011.

TOSIN, Carlos E. S. **Conhecendo o Hibernate Validator**. Java Magazine, Edição 83, Ano VII, p110, 2010.

WEISSMANN, Henrique L. **Grails: um guia rápido e indireto**. Disponível em: <http://www. itexto.net/devkico/?page_id=220>. Acesso em: 16 abr. 2011.

APÊNDICES

APÊNDICE A – Página gerenciaUsuario.xhtml do protótipo em JSF 2.0

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:p="http://primefaces.prime.com.tr/ui">
    <h:head>
        <title>Controle Financeiro</title>
    </h:head>
    <h:body>
        <p:growl id="aviso" life="10000"/>
        <p:layout fullPage="true">
            <p:layoutUnit position="left" width="200" header="Atividades" resizable="true" closable="true"
collapsible="true">
                <h:form prependId="false">
                    <p:commandLink value="Gerenciar Categoria" action="GerenciarCategoria"
actionListener="#{categoriaBean.gerenciarCategoria}"/>
                    <br></br>
                    <p:commandLink value="Gerenciar Conta" action="GerenciarConta"
actionListener="#{contaBean.gerenciarConta}"/>
                    <br></br>
                    <p:commandLink value="Gerenciar Grupo" action="GerenciarGrupo"
actionListener="#{grupoBean.gerenciarGrupo}"/>
                    <br></br>
                    <p:commandLink value="Gerenciar Lançamento" action="GerenciarLancamento"
actionListener="#{lancamentoBean.gerenciarLancamento}"/>
```

```
            </h:form>
        </p:layoutUnit>
        <p:layoutUnit position="center">
            <h1>Controle Financeiro</h1>
            <br/>
            <h:form prependId="false">
                <p:commandLink value="Novo Usuário" actionListener="#{usuarioBean.prepararAdicionar}"
update="infosUsuario" oncomplete="dialogUsuario.show()"/>
                <br/><br/>
                <p:dataTable id="tabela" var="usuario" value="#{usuarioBean.listaUsuarios}">
                    <p:column>
                        <f:facet name="header">
                            <h:outputText value="Nome"/>
                        </f:facet>
                        <h:outputText value="#{usuario.nome}" />
                    </p:column>
                    <p:column>
                        <f:facet name="header">
                            <h:outputText value="E-mail"/>
                        </f:facet>
                        <h:outputText value="#{usuario.email}" />
                    </p:column>
                    <p:column>
                        <f:facet name="header">
                            <h:outputText value="CPF"/>
                        </f:facet>
                        <h:outputText value="#{usuario.cpf}" />
                    </p:column>
                    <p:column>
                        <f:facet name="header">
                            <h:outputText value="Data Nascimento"/>
                        </f:facet>
                        <h:outputText value="#{usuario.dataNascimento}" />
                    </p:column>
                    <p:column>
                        <f:facet name="header">
                            <h:outputText value="Alterar"/>
                        </f:facet>
                        <p:commandLink actionListener="#{usuarioBean.prepararAlterar}" update="infosUsuario"
oncomplete="dialogUsuario.show()">
```

```
                                <h:graphicImage url="/images/editar.png" styleClass="imagem" style="cursor:pointer;
border-width:0px;"/>
                            </p:commandLink>
                        </p:column>
                        <p:column>
                            <f:facet name="header">
                                <h:outputText value="Excluir"/>
                            </f:facet>
                            <p:commandLink action="#{usuarioBean.excluir}">
                                <h:graphicImage url="/images/delete.png" styleClass="imagem" style="cursor:pointer;
border-width:0px;"/>
                            </p:commandLink>
                        </p:column>
                    </p:dataTable>
                </h:form>
            </p:layoutUnit>
        </p:layout>

        <p:dialog header="Usuario" widgetVar="dialogUsuario"  resizable="false" modal="true" showEffect="slide"
width="500">
            <h:form prependId="false">
                <h:panelGrid id="infosUsuario" columns="2" style="margin-bottom:10px">
                    <h:outputLabel for="nome" value="Nome:" />
                    <h:inputText id="nome" value="#{usuarioBean.usuario.nome}"/>
                    <h:outputLabel for="email" value="E-mail:" />
                    <h:inputText id="email" value="#{usuarioBean.usuario.email}"/>
                    <h:outputLabel for="senha" value="Senha:" />
                    <h:inputSecret id="senha" value="#{usuarioBean.usuario.senha}"/>
                    <h:outputLabel for="cpf" value="CPF:" />
                    <h:inputText id="cpf" value="#{usuarioBean.usuario.cpf}"/>
                    <h:outputLabel for="nasc" value="Data Nascimento:" />
                    <h:inputText id="nasc" value="#{usuarioBean.usuario.dataNascimento}">
                        <f:convertDateTime type="date" dateStyle="short" pattern="yyyy/MM/dd"/>
                    </h:inputText>
                    <p:commandButton update="aviso, tabela" oncomplete="sucesso(xhr, status, args)"
actionListener="#{usuarioBean.salvar(actionEvent)}" value="Salvar"/>
                </h:panelGrid>
            </h:form>
        </p:dialog>
```

```
        <script type="text/javascript">
            function sucesso (xhr, status, args){
                if(args.sucesso == true) {
                    dialogCategoria.hide();
                }
            }
        </script>

    </h:body>
</html>
```

APÊNDICE B – Página listarCategorias.jsp do protótipo em Spring Web MVC

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions"%>

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
        <style media="screen">
            @import url("<c:url value="/static/styles/style.css"/>");
        </style>
        <title>Lista de Categorias</title>
    </head>

    <body>
        <div>
            <div id="menu">
```

```
            <%@ include file="menu.jsp" %>
        </div>
        <div id="main">
            <div>
                <c:if test="${not empty categorias}">
                    <table width="600px">
                        <tr>
                            <td><b>Id</b></td>
                            <td><b>Descrição</b></td>
                            <td><b>Alterar</b></td>
                            <td><b>Excluir</b></td>
                        </tr>
                        <c:forEach items="${categorias}" var="categoria">
                            <c:url var="url" value="/categoria/${categoria.id}" />
                        <tr>
                            <td>${categoria.id}</td>
                            <td>${categoria.descricao}</td>
                            <td>
                                <form:form action="${url}/form" method="GET">
                                    <input alt="Alterar Categoria" src="<c:url
value="/static/images/editar.png"/>" title="Alterar Categoria" type="image" value="Alterar Categoria"/>
                                </form:form>
                            </td>
                            <td>
                                <form:form action="${url}" method="DELETE">
                                    <input alt="Excluir Categoria" src="<c:url
value="/static/images/delete.png"/>" title="Excluir Contato" type="image" value="Excluir Categoria"/>
                                </form:form>
                            </td>
                        </tr>
                        </c:forEach>
                    </table>
                </c:if>
                <c:if test="${empty categorias}">Não há categorias cadastradas.</c:if>
            </div>
        </div>
    </div>
</body>
</html>
```

APÊNDICE C – Página criarCategoria.jsp do protótipo em Spring Web MVC

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
        <style type="text/css" media="screen">
            @import url("<c:url value="/static/styles/style.css"/>");
        </style>
        <title>Criar Categoria</title>
    </head>
    <body>
        <div id="wrap">
            <div id="menu">
                <%@ include file="menu.jsp" %>
            </div>
            <div id="main">
                <div id="body">
                    <c:url var="url" value="/categoria/criar" />
                    <form:form action="${url}" method="POST" modelAttribute="categoria">
                        <div>
                            <label for="descricao">Descricao:</label>
                            <form:errors path="descricao" cssClass="errors"/><br />
                            <form:input cssStyle="width:250px" maxlength="20" path="descricao" size="20"/>
                        </div>
                        <br/>
                        <div class="submit">
                            <input id="criar" type="submit" value="Criar Categoria"/>
                        </div>
                    </form:form>
                </div>
```

```
            </div>
        </div>
    </body>
</html>
```

APÊNDICE D – Página alterarCategoria.jsp do protótipo em Spring Web MVC

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
        <style type="text/css" media="screen">
            @import url("<c:url value="/static/styles/style.css"/>");
        </style>
        <title>Atualizar Categoria</title>
    </head>
    <body>
        <div id="wrap">
            <div id="menu">
                <%@ include file="menu.jsp" %>
            </div>
            <div id="main">
                <div id="body">
                    <c:url var="url" value="/categoria/${categoria.id}" />
                    <form:form action="${url}" method="PUT" modelAttribute="categoria">
                        <div>
                            <label for="descricao">Descrição:</label>
                            <form:input cssStyle="width:250px" maxlength="30" path="descricao" size="30"/>
                        </div>
```

```
                        <br/>
                        <div class="submit">
                            <input id="atualizar" type="submit" value="Atualizar Categoria"/>
                        </div>
                        <form:hidden path="id"/>
                    </form:form>
                </div>
            </div>
        </div>
    </body>
</html>
```

APÊNDICE E – Página create.gsp para a classe Usuario do protótipo em Grails

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
        <meta name="layout" content="main" />
        <g:set var="entityName" value="${message(code: 'usuario.label', default: 'Usuario')}" />
        <title><g:message code="default.create.label" args="[entityName]" /></title>
    </head>
    <body>
        <div class="nav">
            <span class="menuButton"><a class="home" href="${createLink(uri: '/')}"><g:message
code="default.home.label"/></a></span>
            <span class="menuButton"><g:link class="list" action="list"><g:message code="default.list.label"
args="[entityName]" /></g:link></span>
        </div>
        <div class="body">
            <h1><g:message code="default.create.label" args="[entityName]" /></h1>
            <g:if test="${flash.message}">
            <div class="message">${flash.message}</div>
```

```
            </g:if>
            <g:hasErrors bean="${usuarioInstance}">
            <div class="errors">
                <g:renderErrors bean="${usuarioInstance}" as="list" />
            </div>
            </g:hasErrors>
            <g:form action="save" >
                <div class="dialog">
                    <table>
                        <tbody>

                            <tr class="prop">
                                <td valign="top" class="name">
                                    <label for="email"><g:message code="usuario.email.label" default="Email"
/></label>
                                </td>
                                <td valign="top" class="value ${hasErrors(bean: usuarioInstance, field: 'email',
'errors')}">
                                    <g:textField name="email" value="${usuarioInstance?.email}" />
                                </td>
                            </tr>

                            <tr class="prop">
                                <td valign="top" class="name">
                                    <label for="senha"><g:message code="usuario.senha.label" default="Senha"
/></label>
                                </td>
                                <td valign="top" class="value ${hasErrors(bean: usuarioInstance, field: 'senha',
'errors')}">
                                    <g:passwordField name="senha" maxlength="12" value="${usuarioInstance?.senha}"
/>
                                </td>
                            </tr>

                            <tr class="prop">
                                <td valign="top" class="name">
                                    <label for="nomeCompleto"><g:message code="usuario.nomeCompleto.label"
default="Nome Completo" /></label>
                                </td>
```

```
                                <td valign="top" class="value ${hasErrors(bean: usuarioInstance, field:
'nomeCompleto', 'errors')}">
                                    <g:textField name="nomeCompleto" maxlength="50"
value="${usuarioInstance?.nomeCompleto}" />
                                </td>
                            </tr>

                            <tr class="prop">
                                <td valign="top" class="name">
                                    <label for="CPF"><g:message code="usuario.CPF.label" default="CPF" /></label>
                                </td>
                                <td valign="top" class="value ${hasErrors(bean: usuarioInstance, field: 'CPF',
'errors')}">
                                    <g:textField name="CPF" maxlength="11" value="${usuarioInstance?.CPF}" />
                                </td>
                            </tr>

                            <tr class="prop">
                                <td valign="top" class="name">
                                    <label for="dataNascimento"><g:message code="usuario.dataNascimento.label"
default="Data Nascimento" /></label>
                                </td>
                                <td valign="top" class="value ${hasErrors(bean: usuarioInstance, field:
'dataNascimento', 'errors')}">
                                    <g:datePicker name="dataNascimento" precision="day"
value="${usuarioInstance?.dataNascimento}"  />
                                </td>
                            </tr>

                        </tbody>
                    </table>
                </div>
                <div class="buttons">
                    <span class="button"><g:submitButton name="create" class="save" value="${message(code:
'default.button.create.label', default: 'Create')}" /></span>
                </div>
            </g:form>
        </div>
    </body>
</html>
```

APÊNDICE F – Página edit.gsp para a classe Usuario do protótipo em Grails

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
        <meta name="layout" content="main" />
        <g:set var="entityName" value="${message(code: 'usuario.label', default: 'Usuario')}" />
        <title><g:message code="default.edit.label" args="[entityName]" /></title>
    </head>
    <body>
        <div class="nav">
            <span class="menuButton"><a class="home" href="${createLink(uri: '/')}"><g:message
code="default.home.label"/></a></span>
            <span class="menuButton"><g:link class="list" action="list"><g:message code="default.list.label"
args="[entityName]" /></g:link></span>
            <span class="menuButton"><g:link class="create" action="create"><g:message code="default.new.label"
args="[entityName]" /></g:link></span>
        </div>
        <div class="body">
            <h1><g:message code="default.edit.label" args="[entityName]" /></h1>
            <g:if test="${flash.message}">
            <div class="message">${flash.message}</div>
            </g:if>
            <g:hasErrors bean="${usuarioInstance}">
            <div class="errors">
                <g:renderErrors bean="${usuarioInstance}" as="list" />
            </div>
            </g:hasErrors>
            <g:form method="post" >
                <g:hiddenField name="id" value="${usuarioInstance?.id}" />
                <g:hiddenField name="version" value="${usuarioInstance?.version}" />
                <div class="dialog">
                    <table>
                        <tbody>
```

```
                            <tr class="prop">
                                <td valign="top" class="name">
                                  <label for="email"><g:message code="usuario.email.label" default="Email"
/></label>
                                </td>
                                <td valign="top" class="value ${hasErrors(bean: usuarioInstance, field: 'email',
'errors')}">
                                    <g:textField name="email" value="${usuarioInstance?.email}" />
                                </td>
                            </tr>

                            <tr class="prop">
                                <td valign="top" class="name">
                                  <label for="senha"><g:message code="usuario.senha.label" default="Senha"
/></label>
                                </td>
                                <td valign="top" class="value ${hasErrors(bean: usuarioInstance, field: 'senha',
'errors')}">
                                    <g:passwordField name="senha" maxlength="12" value="${usuarioInstance?.senha}"
/>
                                </td>
                            </tr>

                            <tr class="prop">
                                <td valign="top" class="name">
                                  <label for="nomeCompleto"><g:message code="usuario.nomeCompleto.label"
default="Nome Completo" /></label>
                                </td>
                                <td valign="top" class="value ${hasErrors(bean: usuarioInstance, field:
'nomeCompleto', 'errors')}">
                                    <g:textField name="nomeCompleto" maxlength="50"
value="${usuarioInstance?.nomeCompleto}" />
                                </td>
                            </tr>

                            <tr class="prop">
                                <td valign="top" class="name">
                                  <label for="CPF"><g:message code="usuario.CPF.label" default="CPF" /></label>
                                </td>
```

```
                                <td valign="top" class="value ${hasErrors(bean: usuarioInstance, field: 'CPF',
'errors')}">
                                    <g:textField name="CPF" maxlength="11" value="${usuarioInstance?.CPF}" />
                                </td>
                            </tr>

                            <tr class="prop">
                                <td valign="top" class="name">
                                  <label for="dataNascimento"><g:message code="usuario.dataNascimento.label"
default="Data Nascimento" /></label>
                                </td>
                                <td valign="top" class="value ${hasErrors(bean: usuarioInstance, field:
'dataNascimento', 'errors')}">
                                    <g:datePicker name="dataNascimento" precision="day"
value="${usuarioInstance?.dataNascimento}"  />
                                </td>
                            </tr>

                            <tr class="prop">
                                <td valign="top" class="name">
                                  <label for="lancamento"><g:message code="usuario.lancamento.label"
default="Lancamento" /></label>
                                </td>
                                <td valign="top" class="value ${hasErrors(bean: usuarioInstance, field:
'lancamento', 'errors')}">

<ul>
<g:each in="${usuarioInstance?.lancamento?}" var="l">
    <li><g:link controller="lancamento" action="show" id="${l.id}">${l?.encodeAsHTML()}</g:link></li>
</g:each>
</ul>
<g:link controller="lancamento" action="create" params="['usuario.id': usuarioInstance?.id]">${message(code:
'default.add.label', args: [message(code: 'lancamento.label', default: 'Lancamento')])}</g:link>

                                </td>
                            </tr>

                        </tbody>
                    </table>
                </div>
```

```
                <div class="buttons">
                    <span class="button"><g:actionSubmit class="save" action="update" value="${message(code:
'default.button.update.label', default: 'Update')}" /></span>
                    <span class="button"><g:actionSubmit class="delete" action="delete" value="${message(code:
'default.button.delete.label', default: 'Delete')}" onclick="return confirm('${message(code:
'default.button.delete.confirm.message', default: 'Are you sure?')}');" /></span>
                </div>
            </g:form>
        </div>
    </body>
</html>
```

APÊNDICE G – Página list.gsp para a classe Usuario do protótipo em Grails

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
        <meta name="layout" content="main" />
        <g:set var="entityName" value="${message(code: 'usuario.label', default: 'Usuario')}" />
        <title><g:message code="default.list.label" args="[entityName]" /></title>
    </head>
    <body>
        <div class="nav">
            <span class="menuButton"><a class="home" href="${createLink(uri: '/')}"><g:message
code="default.home.label"/></a></span>
            <span class="menuButton"><g:link class="create" action="create"><g:message code="default.new.label"
args="[entityName]" /></g:link></span>
        </div>
        <div class="body">
            <h1><g:message code="default.list.label" args="[entityName]" /></h1>
            <g:if test="${flash.message}">
```

```
            <div class="message">${flash.message}</div>
            </g:if>
            <div class="list">
                <table>
                    <thead>
                        <tr>

                            <g:sortableColumn property="id" title="${message(code: 'usuario.id.label', default:
'Id')}" />

                            <g:sortableColumn property="email" title="${message(code: 'usuario.email.label',
default: 'Email')}" />

                            <g:sortableColumn property="nomeCompleto" title="${message(code:
'usuario.nomeCompleto.label', default: 'Nome Completo')}" />

                            <g:sortableColumn property="CPF" title="${message(code: 'usuario.CPF.label', default:
'CPF')}" />

                            <g:sortableColumn property="dataNascimento" title="${message(code:
'usuario.dataNascimento.label', default: 'Data Nascimento')}" />

                        </tr>
                    </thead>
                    <tbody>
                    <g:each in="${usuarioInstanceList}" status="i" var="usuarioInstance">
                        <tr class="${(i % 2) == 0 ? 'odd' : 'even'}">

                            <td><g:link action="show" id="${usuarioInstance.id}">${fieldValue(bean: usuarioInstance,
field: "id")}</g:link></td>

                            <td>${fieldValue(bean: usuarioInstance, field: "email")}</td>

                            <td>${fieldValue(bean: usuarioInstance, field: "nomeCompleto")}</td>

                            <td>${fieldValue(bean: usuarioInstance, field: "CPF")}</td>

                            <td><g:formatDate date="${usuarioInstance.dataNascimento}" /></td>

                        </tr>
```

```
                    </g:each>
                    </tbody>
                </table>
            </div>
            <div class="paginateButtons">
                <g:paginate total="${usuarioInstanceTotal}" />
            </div>
        </div>
    </body>
</html>
```

APÊNDICE H – Página show.gsp para a classe Usuario do protótipo em Grails

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
        <meta name="layout" content="main" />
        <g:set var="entityName" value="${message(code: 'usuario.label', default: 'Usuario')}" />
        <title><g:message code="default.show.label" args="[entityName]" /></title>
    </head>
    <body>
        <div class="nav">
            <span class="menuButton"><a class="home" href="${createLink(uri: '/')}"><g:message
code="default.home.label"/></a></span>
            <span class="menuButton"><g:link class="list" action="list"><g:message code="default.list.label"
args="[entityName]" /></g:link></span>
            <span class="menuButton"><g:link class="create" action="create"><g:message code="default.new.label"
args="[entityName]" /></g:link></span>
        </div>
        <div class="body">
            <h1><g:message code="default.show.label" args="[entityName]" /></h1>
```

```
            <g:if test="${flash.message}">
            <div class="message">${flash.message}</div>
            </g:if>
            <div class="dialog">
                <table>
                    <tbody>

                        <tr class="prop">
                            <td valign="top" class="name"><g:message code="usuario.id.label" default="Id" /></td>

                            <td valign="top" class="value">${fieldValue(bean: usuarioInstance, field: "id")}</td>

                        </tr>

                        <tr class="prop">
                            <td valign="top" class="name"><g:message code="usuario.email.label" default="Email"
/></td>

                            <td valign="top" class="value">${fieldValue(bean: usuarioInstance, field: "email")}</td>

                        </tr>

                        <tr class="prop">
                            <td valign="top" class="name"><g:message code="usuario.senha.label" default="Senha"
/></td>

                            <td valign="top" class="value">${fieldValue(bean: usuarioInstance, field: "senha")}</td>

                        </tr>

                        <tr class="prop">
                            <td valign="top" class="name"><g:message code="usuario.nomeCompleto.label" default="Nome
Completo" /></td>

                            <td valign="top" class="value">${fieldValue(bean: usuarioInstance, field:
"nomeCompleto")}</td>

                        </tr>

                        <tr class="prop">
```

```
                            <td valign="top" class="name"><g:message code="usuario.CPF.label" default="CPF" /></td>

                            <td valign="top" class="value">${fieldValue(bean: usuarioInstance, field: "CPF")}</td>

                        </tr>

                        <tr class="prop">
                            <td valign="top" class="name"><g:message code="usuario.dataNascimento.label"
default="Data Nascimento" /></td>

                            <td valign="top" class="value"><g:formatDate date="${usuarioInstance?.dataNascimento}"
/></td>

                        </tr>

                        <tr class="prop">
                            <td valign="top" class="name"><g:message code="usuario.lancamento.label"
default="Lancamento" /></td>

                            <td valign="top" style="text-align: left;" class="value">
                                <ul>
                                <g:each in="${usuarioInstance.lancamento}" var="l">
                                    <li><g:link controller="lancamento" action="show"
id="${l.id}">${l?.encodeAsHTML()}</g:link></li>
                                </g:each>
                                </ul>
                            </td>

                        </tr>

                    </tbody>
                </table>
            </div>
            <div class="buttons">
                <g:form>
                    <g:hiddenField name="id" value="${usuarioInstance?.id}" />
                    <span class="button"><g:actionSubmit class="edit" action="edit" value="${message(code:
'default.button.edit.label', default: 'Edit')}" /></span>
```

```
                    <span class="button"><g:actionSubmit class="delete" action="delete" value="${message(code:
'default.button.delete.label', default: 'Delete')}" onclick="return confirm('${message(code:
'default.button.delete.confirm.message', default: 'Are you sure?')}');" /></span>
                </g:form>
            </div>
        </div>
    </body>
</html>
```