



Desenvolvimento Web Ágil para a plataforma Java

Delano Medeiros Beder

Copyright © 2016 Delano Medeiros Beder

Esse material está licenciado sob a Licença *Creative Commons Atribuição-NãoComercial-CompartilhaIgual 3.0 Brasil*. Para obter uma cópia dessa licença, visite <http://creativecommons.org/licenses/by-nc-sa/3.0/br>.

Abril 2016



Sumário

1	Introdução	3
1.1	Modelo-Visão-Controlador (MVC)	4
1.2	Grails	5
1.2.1	Ambiente de Desenvolvimento	6
1.3	Considerações finais	8
2	Controle Bancário: Versão 1	9
2.1	Configuração da aplicação	11
2.1.1	Instalação de <i>plugins</i> e definição de dependências	11
2.1.2	Configuração do banco de dados	12
2.2	Implementando as primeiras funcionalidades	13
2.2.1	Classe de Domínio: Estado	14
2.2.2	Classe de Domínio: Cidade	16
2.2.3	Classe de Domínio: Endereco	17
2.2.4	Classe de Domínio: Banco	18
2.2.5	Classe de Domínio: Agencia	19
2.2.6	Classe de Domínio: Gerente	20
2.2.7	Classe de Domínio: CaixaEletronico	21
2.2.8	Classe de Domínio: Transacao	22
2.2.9	Classe de Domínio: ContaCliente	23
2.2.10	Classe de Domínio: Conta	24
2.2.11	Classe de Domínio: ContaCorrente	25
2.2.12	Classe de Domínio: ContaPoupanca	25
2.2.13	Classe de Domínio: Cliente	27
2.2.14	Classe de Domínio: ClienteFisico	28
2.2.15	Classe de Domínio: ClienteJuridico	28

2.3	Scaffolding	29
2.3.1	Scaffolding Dinâmico	29
2.3.2	Scaffolding Estático	30
2.3.3	Convenção na nomenclatura de URLs	32
2.3.4	Controlador: TransacaoController	32
2.3.5	Groovy Server Pages (GSPs)	34
2.4	Executando a aplicação	35
2.5	Considerações finais	41
3	Controle Bancário: Versão 2	43
3.1	Configuração da aplicação	43
3.2	Controle de Acesso	45
3.2.1	Classes de Domínio: Cliente, ClienteFisico, ClienteJuridico e Gerente	46
3.3	Internacionalização	46
3.4	Personalização dos templates utilizados no scaffolding	48
3.4.1	Template: Controller.groovy	49
3.4.2	Template: create.gsp	49
3.4.3	Template: index.gsp	52
3.4.4	Template: show.gsp	53
3.5	Controladores e Visões	54
3.5.1	Controlador: ContaController	54
3.5.2	Visão: conta/index.gsp	55
3.5.3	Controlador: ClienteController	56
3.5.4	Visão: cliente/index.gsp	56
3.5.5	Mapeamento URL	57
3.5.6	Controlador: MainController	58
3.5.7	Visão: main/index.gsp	58
3.5.8	Controladores: últimas alterações relacionadas ao controle de acesso	59
3.6	Melhorando o leiaute da aplicação: biblioteca de marca	60
3.7	Executando a aplicação	62
3.8	Considerações finais	67
4	Controle Bancário: Versão 3	69
4.1	Configuração da aplicação	70
4.2	Atribuição de papéis	71
4.2.1	Classe de domínio Gerente X Papel ROLE_GERENTE	71
4.2.2	Classe de domínio Cliente X Papel ROLE_CLIENTE	72
4.3	Controle de acesso: Transações	73
4.3.1	Controlador: MainController	73
4.3.2	Biblioteca de marcas: LoginTagLib	74
4.3.3	Controlador: SeleccionaContaController	75
4.3.4	Visão: seleccionaConta/index.gsp	75
4.3.5	Classe de Domínio: Transacao	76
4.3.6	Controlador: TransacaoController	76

4.3.7	Template transacao/_form.gsp	78
4.3.8	Executando a aplicação	79
4.4	Controle de acesso: Contas	80
4.4.1	Controlador: ContaCorrenteController	80
4.4.2	Template contaCorrente/_form.gsp	80
4.4.3	Controlador: ContaClienteController	82
4.4.4	Template contaCliente/_form.gsp	83
4.4.5	Controlador: ContaController	83
4.4.6	Executando a aplicação	84
4.5	Preenchimento automático de endereços	85
4.5.1	Templates endereco/_form.gsp & endereco/_address.gsp	85
4.5.2	Controlador: EnderecoController	86
4.6	Considerações finais	87
5	Controle Bancário: Versão 4	89
5.1	Configuração da aplicação	89
5.2	Design responsivo	92
5.2.1	Templates: create.gsp e edit.gsp	92
5.2.2	Template: index.gsp	94
5.2.3	Template: show.gsp	95
5.2.4	Biblioteca de marcas: LoginTagLib	95
5.2.5	Visões: geração automática pelo mecanismo de <i>Scaffolding</i>	96
5.3	Personalização dos atributos (Cliente Físico/Jurídico e Gerente)	97
5.3.1	Cliente Físico, Cliente Jurídico e Gerente	97
5.3.2	Cliente	100
5.4	Personalização das visões: Contas (corrente & poupança)	102
5.4.1	Visão: conta/index.gsp	103
5.5	Visões: main/index.gsp & selecionaConta/index.gsp	103
5.6	Personalização das visões: Transações	105
5.6.1	Visão: transacao/show.gsp	106
5.7	Extratos Bancários	108
5.7.1	Visões	110
5.8	Internacionalização - Mensagens I18n	113
5.9	Serviço <i>web</i> REST	114
5.10	Considerações finais	116
6	Considerações finais	117
6.1	Automação de testes	117
6.2	Estudos complementares	119
	Índice Remissivo	123



Lista de Códigos

2.1	build.gradle (configuração de <i>plugins</i>)	11
2.2	application.yml (configuração banco de dados)	12
2.3	Classe de domínio Estado	15
2.4	Classe de domínio Cidade	16
2.5	Classe de domínio Endereco	17
2.6	Classe de domínio Banco	18
2.7	Classe de domínio Agencia	19
2.8	Classe de domínio Gerente	20
2.9	Classe de domínio CaixaEletronico	21
2.10	Classe de domínio Transacao	22
2.11	Classe de domínio ContaCliente	23
2.12	Classe de domínio Conta	24
2.13	Classe de domínio ContaCorrente	25
2.14	Classe de domínio ContaPoupanca	25
2.15	Classe de domínio Cliente	27
2.16	Classe de domínio ClienteFisico	28
2.17	Classe de domínio ClienteJuridico	28
2.18	Controlador TransacaoController (1)	30
2.19	Controlador TransacaoController	33
2.20	Visão transacao/index.gsp	34
2.21	Bootstrap.groovy (1)	35
2.22	Bootstrap.groovy (2)	36
2.23	Bootstrap.groovy (3)	37
2.24	Bootstrap.groovy (4)	38
2.25	Bootstrap.groovy (5)	39
3.1	BuildConfig.groovy	44
3.2	Usuários: Clientes e Gerentes	46
3.3	Template Controller.groovy	50
3.4	Template create.gsp	51
3.5	Template index.gsp	52

3.6	<i>Template</i> show.gsp	53
3.7	Controlador ContaController	54
3.8	Visão conta/index.gsp	55
3.9	Controlador ClienteController	56
3.10	Visão cliente/index.gsp	57
3.11	URLMappings.groovy	57
3.12	Controlador MainController	58
3.13	Visão main/index.gsp	58
3.14	Controladores - últimas alterações relacionadas ao Controle de Acesso	59
3.15	Biblioteca de marca LoginTagLib	60
3.16	<i>Template</i> _footer.gsp	60
3.17	<i>Template</i> _header.gsp	60
3.18	Leiaute padrão grails-app/views/layouts/main.gsp	61
3.19	Arquivo main.css	61
3.20	Bootstrap.groovy (1)	62
3.21	Bootstrap.groovy (2)	63
3.22	Bootstrap.groovy (3)	64
3.23	Bootstrap.groovy (4)	65
3.24	Bootstrap.groovy (5)	66
4.1	BuildConfig.groovy	70
4.2	Controlador Gerente (ação save())	71
4.3	Controlador ClienteFisico (ação save())	72
4.4	Controlador ClienteJuridico (ação save())	72
4.5	Controlador MainController	73
4.6	Biblioteca de marca LoginTagLib	74
4.7	Controlador SelecionaContaController	75
4.8	Visão selecionaConta/index.gsp	75
4.9	Classe de domínio Transacao	76
4.10	Controlador TransacaoController	77
4.11	<i>Template</i> transacao/_form.gsp	78
4.12	Controlador ContaCorrenteController	81
4.13	<i>Template</i> contaCorrente/_form.gsp	81
4.14	Controlador ContaClienteController	82
4.15	<i>Template</i> contaCliente/_form.gsp	83
4.16	Controlador ContaController	83
4.17	<i>Template</i> endereco/_form.gsp	85
4.18	<i>Template</i> endereco/_address.gsp	86
4.19	Controlador EnderecoController	87
5.1	BuildConfig.groovy	90
5.2	Config.groovy	91
5.3	<i>Template</i> scaffolding/create.gsp	92
5.4	<i>Template</i> scaffolding/edit.gsp	93
5.5	<i>Template</i> scaffolding/index.gsp	94
5.6	<i>Template</i> scaffolding/show.gsp	95
5.7	Biblioteca de marca LoginTagLib	96
5.8	Visão clienteFisico/create.gsp	97
5.9	<i>Template</i> clienteFisico/_form.gsp	98
5.10	Visão clienteFisico/index.gsp	99

5.11	Visão cliente/index.gsp	100
5.12	Controlador ClienteController	101
5.13	Visão contaCorrente/create.gsp	102
5.14	<i>Template</i> contaCorrente/_form.gsp	102
5.15	Visão conta/index.gsp	103
5.16	Visão main/index.gsp	104
5.17	Visão selecionaConta/index.gsp	104
5.18	Visão transacao/create.gsp	105
5.19	<i>Template</i> transacao/_form.gsp	105
5.20	<i>Template</i> transacao/_form.gsp	107
5.21	Controlador ExtratoController	108
5.22	Classe Java Line	109
5.23	Visão extrato/index.gsp	110
5.24	<i>Template</i> extrato/_pdf.gsp	112
5.25	Visão extrato/chart.gsp	112
5.26	Controlador AgenciaController	114
5.27	Classe Java Info	115



Lista de Figuras

1.1	Padrão arquitetural MVC	4
1.2	Verificação da instalação do Java	6
1.3	Console do SGBD H2	8
2.1	Criação do Projeto ControleBancario.	9
2.2	Diagrama de classes UML	13
2.3	Criação da Classe de Domínio Estado.	14
2.4	GORM: Mapeamento de Hierarquia	26
2.5	Criação do controlador (<i>scaffolding</i> dinâmico).	29
2.6	Criação do controlador e das visões (<i>scaffolding</i> estático).	30
2.7	Scaffolding estático das classes de domínio.	31
2.8	Convenção na nomenclatura de URLs.	32
2.9	Execução da aplicação ControleBancario	39
2.10	Lista de transações bancárias	40
2.11	Criação de uma nova transação bancária	40
2.12	Criação de uma transação bancária com valores inválidos	41
3.1	I18n (arquivos de propriedades).	47
3.2	Mensagens I18n para a classe de Domínio Usuario	48
3.3	Máscaras de entrada: CEP, CNPJ e CPF	49
3.4	Visão main/index.gsp: diferentes visões	59
3.5	ControleBancario : Página de <i>Login</i>	67
4.1	Visão main/index.gsp : ROLE_CLIENTE	79
4.2	Lista de transações de uma conta do usuário <i>logado</i>	79
4.3	Visão main/index.gsp : ROLE_GERENTE	84
4.4	Lista de contas da agência do usuário <i>logado</i>	84
4.5	Visão endereco/create.gsp : Preenchimento automático de atributos	87
5.1	Instalação do wkhtmltopdf	91
5.2	Comando grails generate-views	96
5.3	Calendário dinâmico: <richui: dataChooser />	97

5.4	Funcionalidade <i>autocomplete</i> – Busca de clientes	101
5.5	Visão transacao/show.gsp : visualização em abas	106
5.6	Visão transacao/show.gsp : menu <i>accordion</i>	106
5.7	Extrato Bancário em formato HTML	111
5.8	Extrato Bancário em formato PDF	111
5.9	Movimentação financeira das contas bancárias	113
5.10	Mensagens internacionalizadas	113
5.11	Serviço <i>web</i> REST: Formato JSON	115
5.12	Serviço <i>web</i> REST: Formato XML	116



Lista de Tabelas

2.1	Projeto Grails: <i>Overview</i> dos Diretórios.	10
2.2	Regras de restrições.	15
3.1	Classes de domínio: geração dos Controladores e Visões.	54



1 — Introdução

É indiscutível que a web se tornou uma tecnologia essencial para negócios, comércio, educação, engenharia, entretenimento, finanças, governo, indústria, mídia, medicina, política, ciência e transporte, isso para citar apenas algumas áreas que têm impacto sobre nossa vida.

No entanto, à medida que aplicações web são integradas às estratégias de negócio, torna-se cada vez mais complexo o seu desenvolvimento. Apesar desse aumento de responsabilidade, o processo de engenharia de sistemas web ainda não é tão disciplinado quanto à Engenharia de Software tradicional. Muitas aplicações web continuam a ser construídas de uma maneira *ad hoc*, sem consideração com os princípios fundamentais da Engenharia de Software.

Dessa forma, é essencial que sistemas passem por um processo de engenharia, denominado Engenharia Web, de forma a construir e implantar uma solução eficaz e eficiente e que atenda às estratégias de negócio e às expectativas de seus usuários, pois como nos demais tipos de software, é necessário o completo entendimento do problema para se projetar uma solução eficaz que seja implementada e testada corretamente.

Nesse contexto, a Engenharia Web [1, 2] consiste em uma abordagem sistemática e ágil para o desenvolvimento de aplicações web. A agilidade implica em um enfoque de Engenharia de Software enxuto que incorpora ciclos de desenvolvimento rápidos. E cada ciclo resulta na implantação de um incremento da aplicação web. Ou seja, o desenvolvimento da aplicação web é realizado por meio da entrega de uma série de versões, chamadas de incrementos, que fornecem progressivamente mais funcionalidade para os clientes à medida que cada incremento é entregue.

Esse material apresenta o framework web Grails que é bem inerente à filosofia ágil. Na realidade, Grails, por si só não é ágil, pois nenhuma ferramenta por si só pode ser ágil. No entanto, ele se encaixa muito bem com a filosofia de desenvolvimento ágil que sustenta que a melhor forma de atender às necessidades dos clientes é por meio da colaboração de um grupo comprometido de pessoas, que se concentra na obtenção de resultados com rapidez, com o mínimo de sobrecarga possível [3, 4].

O framework Grails é inerente a muitas das boas práticas do desenvolvimento ágil, incluindo as seguintes:

- **Ser adaptável à mudança:** Graças ao seu mecanismo de *autoreloading* e natureza dinâmica, Grails promove a mudança e o desenvolvimento iterativo.
- **Entrega precoce do software funcional:** A simplicidade de Grails permite uma abordagem de desenvolvimento rápido de aplicações, aumentando as probabilidades de entrega

rápida. Além disso, Grails prega a automação dos testes unitários e de integração (ver Capítulo 6), conscientizando os desenvolvedores de sua importância.

- **Simplicidade é essencial:** Grails visa proporcionar simplicidade. Ou seja, Grails tem conceitos complexos como ORM (Object-Relational Mapping¹), porém ela encapsula esses conceitos em uma API simples. Mapeamento objeto-relacional é uma técnica de desenvolvimento de software que é utilizada com o objetivo de reduzir os problemas inerentes à utilização conjunta de banco de dados relacionais e o paradigma de desenvolvimento orientado a objetos. As tabelas do banco de dados são representadas através de classes e os registros de cada tabela são representados como instâncias das classes correspondentes.
- **Equipe entusiasmada, auto-organizada com o ambiente certo:** Desde que Grails permite aos desenvolvedores centrar-se principalmente na lógica de negócios necessários para resolver um problema específico – ao invés de aspectos relacionados à configuração de sua aplicação – a equipe está mais propensa a ter desenvolvedores entusiasmados.

1.1 Modelo-Visão-Controlador (MVC)

Desde que o padrão arquitetural MVC é o alicerce do framework de desenvolvimento Grails, torna-se de fundamental importância apresentar um *overview* dos principais conceitos relacionados a esse padrão arquitetural.

O modelo-visão-controlador (MVC) [5, 6] desacopla a interface com o usuário da funcionalidade e conteúdo da aplicação web. Uma representação esquemática da arquitetura MVC aparece na Figura 1.1.

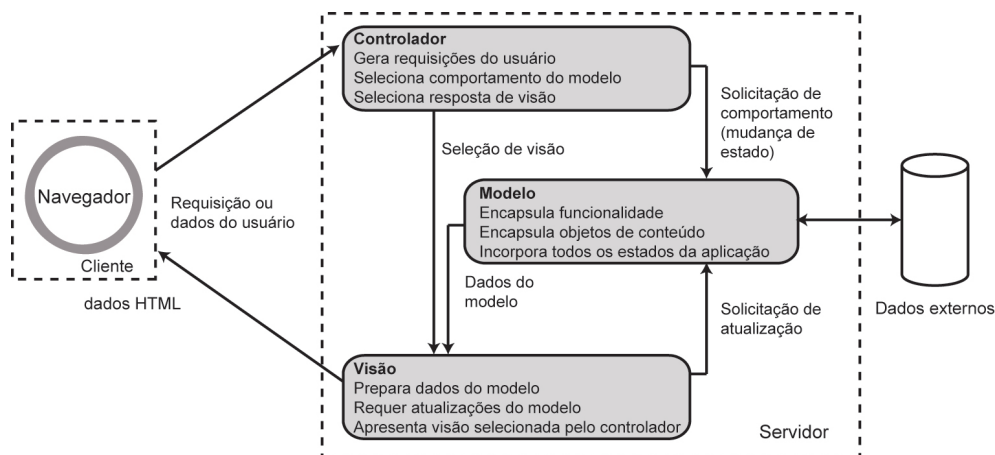


Figura 1.1: Padrão arquitetural MVC

Esse padrão arquitetural define três componentes (Modelo, Visão e Controlador) com características bem delineadas:

- O **Modelo** contém todo o conteúdo específico da aplicação e lógica de processamento, incluindo todos os objetos de conteúdo, acesso a dados externos e fontes de informação, e todas as funcionalidades de processamento que são específicas da aplicação. No caso de sistemas que utilizam bases de dados, o modelo mantém o estado persistente do negócio e somente ele pode acessar as bases de dados;

¹Mapeamento Objeto-Relacional

- A **Visão** contém todas as funcionalidades específicas da interface que habilita a apresentação de conteúdo e lógica de processamento; e
- O **Controlador** gerencia o acesso e a manipulação do modelo e da visão, bem como coordenar o fluxo de dados entre eles. Em uma aplicação web, o controlador monitora a interação com o usuário e, baseando-se nisso, recupera os dados do modelo e utiliza-os para atualizar ou construir a visão.

Conforme mostra a Figura 1.1, as solicitações ou dados do usuário são tratados pelo controlador que seleciona a visão apropriada com base na solicitação do usuário. Quando o tipo de solicitação é determinado, uma solicitação de comportamento é transmitida ao modelo, que implementa a funcionalidade ou recupera o conteúdo exigido para satisfazer a solicitação. O modelo pode acessar dados armazenados em um banco de dados corporativo, como parte de um armazenamento de dados local ou como uma coleção de arquivos independentes. Os dados devolvidos pelo modelo devem ser formatados e organizados pela visão apropriada e depois transmitidos do servidor da aplicação de volta para exibição no navegador presente na máquina do cliente [6].

1.2 Grails

Grails [7] é um framework web baseado no padrão arquitetural MVC [5] que utiliza a linguagem Groovy [8], executa sobre a máquina Virtual Java (JVM) e objetiva a alta produtividade no desenvolvimento de aplicações web. Ele combina os principais frameworks (Hibernate², Spring³, etc.) utilizados na plataforma Java e respeita o paradigma *Convention-over-configuration* (Convenção ao invés de Configuração).

Groovy. Groovy [8] é uma linguagem dinâmica, ágil para a plataforma Java inspirada em Python e Ruby que possui sua sintaxe semelhante à de aplicações desenvolvidas em Java. Apesar de poder ser usada como uma linguagem de *script*, ou seja, não gerar arquivos executáveis e não precisar ser compilada, Groovy não se limita a isso. Aplicações feitas nesta linguagem podem ser compiladas utilizando-se um compilador Java, gerando *bytecodes* Java (mesmo formato da compilação de uma aplicação escrita em Java), além disso, podem ser utilizadas em aplicações escritas puramente em Java.

A linguagem foi desenvolvida em 2004 por James Strachan. A sua sintaxe é extremamente parecida com a do Java, além disso, é possível “integrar” aplicações Java e Groovy de forma transparente. O Groovy, inclusive, simplifica a implementação por “adicionar” dinamicamente às suas classes os métodos de acesso (**get** e **set**), economizando tempo e esforço. O objetivo de Groovy é simplificar a sintaxe de Java para representar comportamentos dinâmicos como consultas a banco de dados, escritas e leituras de arquivos e geração de objetos em tempo de execução ao invés de compilação [8].

Convention Over Configuration (CoC). O CoC é um paradigma que visa a diminuir a quantidade de decisões que o desenvolvedor precisa tomar, tomando como “padrão” algo que é comumente usado (uma convenção). Se o padrão escolhido pelo framework for a que o desenvolvedor precisa, este não gasta tempo tendo que alterá-la. Entretanto, se ele necessita de algo diferente, fica livre para configurar da forma que desejar. No caso do Grails, ele assume diversas configurações, tais como as de banco de dados, as de localização do código-fonte, entre outras.

²<http://www.hibernate.org/>

³<http://springsource.org/>

Este tutorial apresenta o framework Grails apoiando o Desenvolvimento Web Ágil de Software. Uma aplicação denominada **ControleBancario** ilustra as diferentes etapas do processo de desenvolvimento:

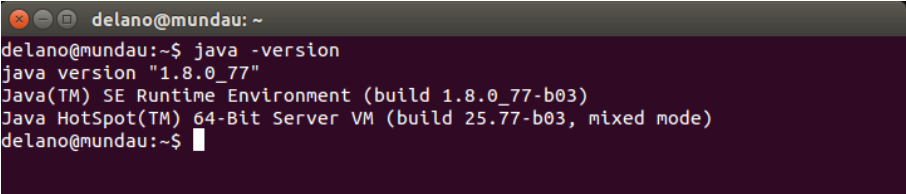
- O capítulo 2 descreve a implementação inicial das principais funcionalidades dessa aplicação;
- O capítulo 3 descreve a implementação das funcionalidades de autenticação de usuários no contexto dessa aplicação.
- O capítulo 4 descreve como pode ser realizada a personalização das funcionalidade presentes na aplicação ao adicionar aspectos relacionados à autorização do acesso às funcionalidades.
- O capítulo 5 descreve como pode ser realizada a personalização da interface da aplicação ao adicionar padrões de interface. Adicionalmente, esse capítulo descreve a implementação de um serviço *web* REST e de algumas funcionalidades AJAX na aplicação **ControleBancario**.
- Finalmente, o capítulo 6 apresenta o *overview* de mais algumas funcionalidades presentes em Grails que não foram abordadas nos capítulos anteriores.

1.2.1 Ambiente de Desenvolvimento

Esta seção apresenta alguns pré-requisitos⁴ do ambiente para apoiar o desenvolvimento. O atendimento desses pré-requisitos são fundamentais para a instalação e configuração do ambiente de desenvolvimento que apoiará a compilação e execução dos exemplos discutidos neste tutorial. A instalação do ambiente é simples e consiste de:

Instalação da linguagem Java. O Java Development Kit (JDK) – versão igual ou superior a 1.5 – será necessário para executar os exemplos apresentados nesse material. A última versão do JDK pode ser obtida em http://java.com/pt_BR/download/index.jsp (Esse material utiliza o JDK versão 1.8.0_77).

Dicas importantes: (1) A variável de ambiente **JAVA_HOME** precisa apontar para o diretório onde o JDK foi instalado. (2) Digite **java -version** em um terminal para verificar se o Java foi instalado corretamente (Figura 1.2).



```
delano@mundau: ~  
delano@mundau:~$ java -version  
java version "1.8.0_77"  
Java(TM) SE Runtime Environment (build 1.8.0_77-b03)  
Java HotSpot(TM) 64-Bit Server VM (build 25.77-b03, mixed mode)  
delano@mundau:~$
```

Figura 1.2: Verificação da instalação do Java

⁴Mais detalhes podem ser encontrados em: <http://www.itexto.net/devkico/?p=40>.

Instalação do Grails. Nesse tutorial adotou-se a versão 3.1.4 do Grails que pode ser obtida em <https://github.com/grails/grails-core/releases/download/v3.1.4/grails-3.1.4.zip>.

Após realizar o *download*, execute os seguintes passos:

1. Descompacte o Grails em um diretório e crie uma variável de ambiente **GRAILS_HOME** e faça-o apontar para o diretório onde o Grails foi descompactado; e
2. Adicione **GRAILS_HOME/bin** na variável de ambiente **PATH**.

Dica importante: Digite **grails -version** em um terminal para verificar se o Grails foi instalado corretamente e está pronto para uso. Para maiores informações sobre a instalação do Grails, consulte o endereço: <http://grails.org/Installation+Portuguese>.

Instalação de IDE. O IDE IntelliJ é utilizado para desenvolver as aplicações apresentadas nesse tutorial. Nesse tutorial foi adotada a versão 2016.1.1 do IDE IntelliJ que pode ser obtida em <https://www.jetbrains.com/idea>

Embora um IDE⁵ facilite o desenvolvimento, o Grails não exige a utilização de um IDE específico. Todos os comandos necessários ao desenvolvimento podem ser feitos em um terminal de comando. No entanto, assim como em outras linguagens de programação, o IDE torna ágil o processo de desenvolvimento ao integrar diferentes funcionalidades (edição, compilação, execução, etc.) e abstrair a sintaxe dos comandos necessários relacionados a essas atividades. Dessa forma, fica a critério do leitor a escolha do IDE mais adequado às suas necessidades. Uma lista de IDEs que dão apoio ao desenvolvimento de aplicações em Grails pode ser encontrada no seguinte link: <https://grails.org/wiki/IDE%20Integration>.

Banco de Dados. A instalação do Grails, na versão adotada, já incorpora uma cópia do H2⁶, um sistema de gerenciamento de banco de dados relacional totalmente implementado em Java que disponibiliza um console na URI `/dbconsole` (Figura 1.3).

O SGBD H2 é útil para aplicações de demonstração, mas em algum momento os desenvolvedores precisarão de um SGBD mais robusto tais como MySQL⁷, PostgreSQL⁸ e Oracle⁹. Desde que o GORM (*Grails Object-Relational Mapping*) é uma camada sobre o framework *hibernate*, qualquer banco de dados que possua um driver JDBC e um dialeto *hibernate* pode ser utilizado.

⁵Em inglês: *Integrated Development Environment*

⁶<http://www.h2database.com/html/main.html>

⁷<https://www.mysql.com/>

⁸<http://www.postgresql.org/>

⁹<http://www.oracle.com/br/database/overview/index.html>

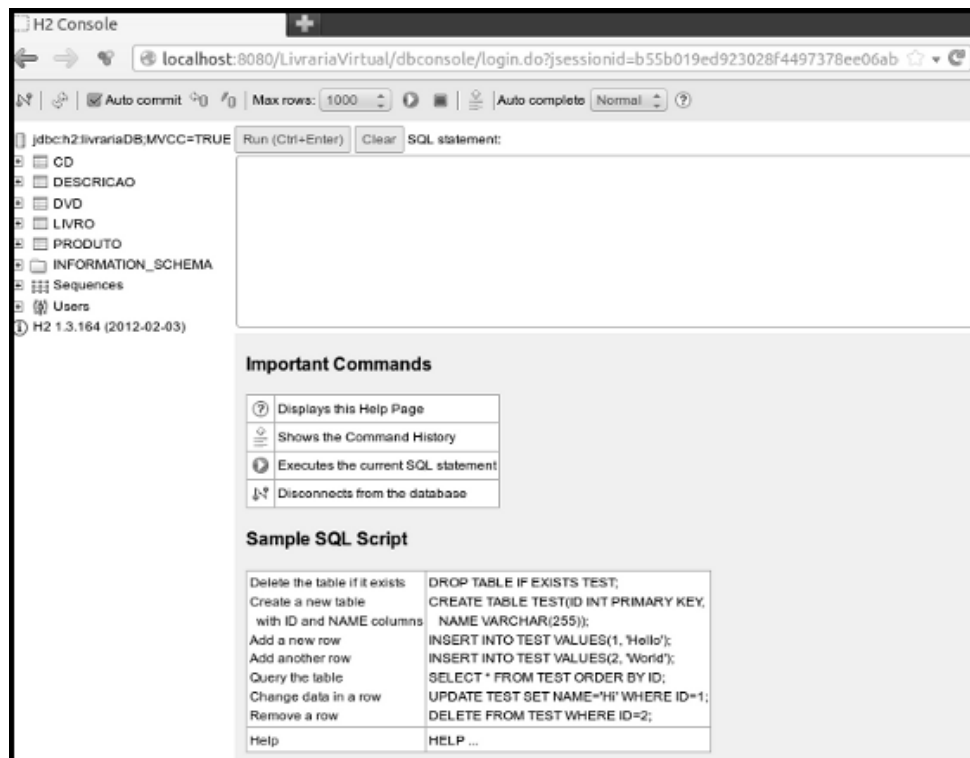


Figura 1.3: Console do SGBD H2

Mecanismo de *build*. Desde sua primeira versão, Grails sempre usou como mecanismo de *build*, a ferramenta Gant¹⁰, que é uma ferramenta bastante poderosa. No entanto, conforme o tempo foi passando novas opções foram surgindo, e desde a versão 3, Grails adota, como ferramenta de *build*, o Gradle¹¹ que vai além do Gant: trata-se de uma ferramenta de gerencia de projetos que lida desde a gestão de dependências, padronização de diretórios, ciclo de vida, construção e muito mais.

1.3 Considerações finais

Esse capítulo apresentou um *overview* das funcionalidades presentes em Grails. Dando continuidade ao desenvolvimento em Grails, o próximo capítulo apresenta a implementação inicial das principais funcionalidades da aplicação **ControleBancario**.

¹⁰<https://github.com/Gant/Gant>

¹¹<http://gradle.org/>



2 — Controle Bancário: Versão 1

Neste capítulo, será apresentado o processo de desenvolvimento da primeira versão da aplicação **ControleBancario**. Trata-se uma aplicação de gestão bancária cujo modelo de entidades é apresentado na Figura 2.2.

O primeiro passo é a criação de um projeto através da execução, em um terminal ou através da utilização de algum IDE, do comando **grails create-app ControleBancario**¹². No caso do IntelliJ IDE, a criação de um projeto segue os seguintes passos:

- No menu principal, selecione: **Create New Project** \Rightarrow **Grails**.
- Em nome do projeto, digite **ControleBancario** e clique em **Finish** (Figura 2.1). O IntelliJ IDE executa o comando **grails create-app**.

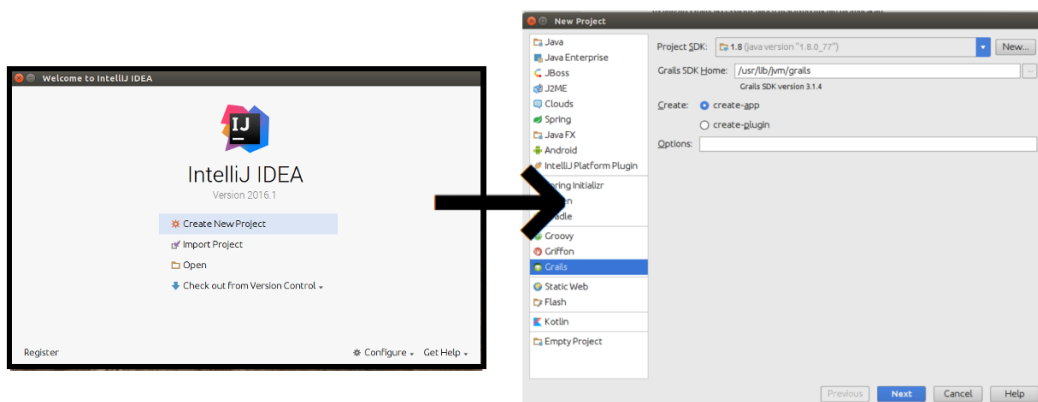


Figura 2.1: Criação do Projeto ControleBancario.

Caso esses passos sejam realizados com sucesso, o projeto da aplicação (hierarquia de diretórios) está criado. Ou seja, foi criado um conjunto de arquivos e diretórios para o projeto. Essa hierarquia de diretórios segue o paradigma *Convention Over Configuration*. Ou seja, os desenvolvedores seguem as convenções e já sabem *a priori* onde se encontram todos os elementos

¹²Para obter a lista completa de comandos Grails, executar o comando **grails help** em um terminal.

que compõem a aplicação em desenvolvimento. Um *overview* do conteúdo desses diretórios é apresentado na Tabela 2.1.

Diretório	Descrição
grails-app/domain	Onde se encontra o M do MVC. Ou seja, onde se encontram as classes de Domínio, ou modelos.
grails-app/controllers	Onde se encontra o C do MVC. Ou seja, onde se encontram os controladores.
grails-app/views	Onde se encontra o V do MVC. Ou seja, onde se encontram as visões (arquivos.gsp – Groovy Server Pages).
grails-app/taglib	Onde se encontram as bibliotecas de marcas (<i>taglibs</i>) criadas pelo usuário.
grails-app/services	Onde se encontram as classes utilizadas na camada de serviços (serviços web).
grails-app/i18n	Onde se encontram os arquivos relacionados à internacionalização.
grails-app/conf	Onde se encontram as configurações da aplicação, tais como a configuração do banco (application.yml), entre outros.
grails-app/init	Onde se encontra a classe BootStrap.groovy utilizada na inicialização de dados da aplicação, entre outros.
grails-app/assets	Esse diretório possui três diretórios (<i>images</i> , <i>javascript</i> e <i>stylesheets</i>) onde se encontram os <i>assets</i> utilizados na aplicação.
src/main/groovy	Onde se encontram outros códigos-fonte Java ou Groovy que não são modelos, controladores, visões ou serviços.
src/test/groovy	Onde se encontram os testes unitários da aplicação.
src/integration-test/groovy	Onde se encontram os testes de integração da aplicação.

Tabela 2.1: Projeto Grails: *Overview* dos Diretórios.

2.1 Configuração da aplicação

Considerando que o projeto foi criado, o próximo passo é configurar as dependências e instalar os *plugins* Grails necessários para o desenvolvimento da aplicação.

2.1.1 Instalação de *plugins* e definição de dependências

Na implementação das funcionalidades da aplicação **ControleBancario**, discutidas nesse capítulo, será utilizado o plugin Grails **br-validation** que auxilia a validação de campos CPF, CNPJ e CEP das entidades da aplicação.

Desde a versão 3 do Grails, a inserção de dependências de *plugins* é realizada no arquivo **build.gradle**. O conteúdo desse arquivo, relacionado à configuração de dependências de *plugins*, é apresentado no Código 2.1. Dessa forma, para instalar o plugin **br-validation** adicione o comando `compile "org.grails.plugins:br-validation:0.3"`, descrevendo a dependência, no arquivo **build.gradle** conforme apresentado na linha 13 do Código 2.1.

```
1 dependencies {  
2     compile "org.springframework.boot:spring-boot-starter-logging"  
3     compile "org.springframework.boot:spring-boot-autoconfigure"  
4     compile "org.grails:grails-core"  
5     compile "org.springframework.boot:spring-boot-starter-actuator"  
6     compile "org.springframework.boot:spring-boot-starter-tomcat"  
7     compile "org.grails:grails-dependencies"  
8     compile "org.grails:grails-web-boot"  
9     compile "org.grails.plugins:cache"  
10    compile "org.grails.plugins:scaffolding"  
11    compile "org.grails.plugins:hibernate4"  
12    compile "org.hibernate:hibernate-ehcache"  
13    compile "org.grails.plugins:br-validation:0.3"  
14    console "org.grails:grails-console"  
15    profile "org.grails.profiles:web:3.1.4"  
16    runtime "org.grails.plugins:asset-pipeline"  
17    runtime "com.h2database:h2"  
18    runtime "org.postgresql:postgresql:9.3-1101-jdbc41"  
19    testCompile "org.grails:grails-plugin-testing"  
20    testCompile "org.grails.plugins:geb"  
21    testRuntime "org.seleniumhq.selenium:selenium-htmlunit-driver:2.47.1"  
22    testRuntime "net.sourceforge.htmlunit:htmlunit:2.18"  
23 }
```

Código 2.1: **build.gradle** (configuração de *plugins*)

2.1.2 Configuração do banco de dados

O SGBD H2, provido pelo Grails, é adequado para aplicações de demonstração, mas em algum momento os desenvolvedores precisarão de um SGBD mais robusto, tais como **MySQL**, **Postgresql** ou **Oracle**. Com propósitos de ilustração, o SGBD **Postgresql** será utilizado na implementação da aplicação **ControleBancario**. No entanto, o leitor pode usar outro SGBD relacional.

Para o uso de outro SGBD, tais como **MySQL** ou **Oracle**, os passos para suas configurações são análogos aos apresentados a seguir para o SGBD **Postgresql**:

- Habilite o driver *JDBC* do SGBD **Postgresql**, conforme apresentado na linha 18 do Código 2.1; e
- Altere o arquivo **grails-app/conf/application.yml** para configurar o acesso (*driver*, *url*, *username* e *password*) ao banco de dados. O conteúdo desse arquivo, relacionado à configuração do banco de dados, é apresentado no Código 2.2.

```
dataSource :
  pooled: true
  jmxExport: true
  driverClassName: org.postgresql.Driver
  username: root
  password: root

environments:
  development:
    dataSource:
      dbCreate: create-drop
      url: jdbc:postgresql://localhost:5432/financeiro
  test:
    dataSource:
      dbCreate: update
      url: jdbc:postgresql://localhost:5432/financeiro
  production:
    dataSource:
      dbCreate: update
      url: jdbc:postgresql://localhost:5432/financeiro
      properties:
        jmxEnabled: true
        initialSize: 5
        maxActive: 50
        minIdle: 5
        maxIdle: 25
        maxWait: 10000
        maxAge: 600000
        timeBetweenEvictionRunsMillis: 5000
        minEvictableIdleTimeMillis: 60000
        validationQuery: SELECT 1
        validationQueryTimeout: 3
        validationInterval: 15000
        testOnBorrow: true
        testWhileIdle: true
        testOnReturn: false
        jdbcInterceptors: ConnectionState
        defaultTransactionIsolation: 2
```

Código 2.2: **application.yml** (configuração banco de dados)

2.2 Implementando as primeiras funcionalidades

Agora que o projeto foi criado e configurado, o próximo passo é implementar as primeiras funcionalidades da aplicação **ControleBancario**. É justificável iniciar pela implementação pelas operações de CRUD das entidades da aplicação. CRUD é o acrônimo para *Create*, *Read*, *Update* e *Delete*. Ou seja, as operações de criação, acesso, atualização e remoção das entidades da aplicação. Figura 2.2 apresenta a modelagem da entidades da aplicação **ControleBancario**.

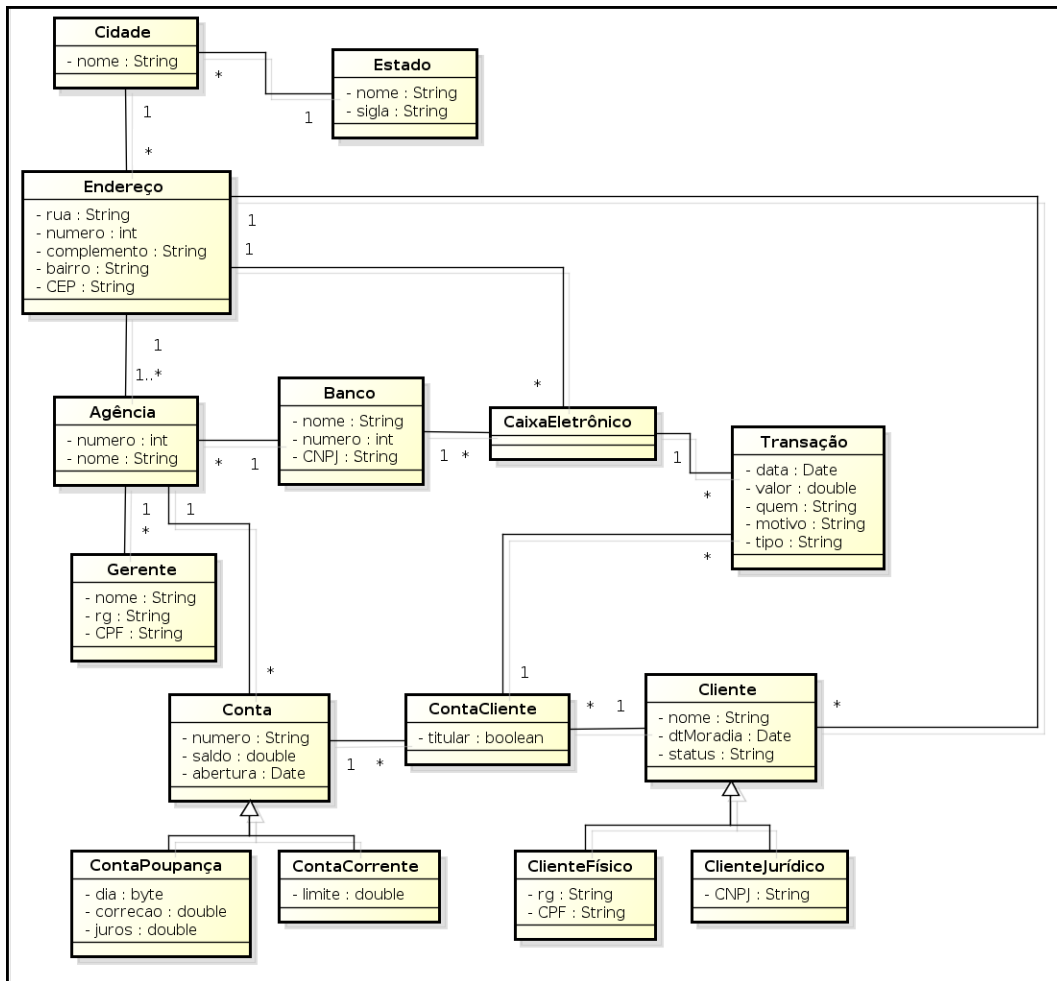


Figura 2.2: Diagrama de classes UML

ControleBancario. Essa aplicação controla bancos, cada um com diversas agências nas quais os clientes podem realizar transações em contas correntes ou poupanças. Um cliente é qualquer pessoa física ou jurídica que abre uma conta corrente ou poupança em uma ou mais agências de um banco operado por essa aplicação. Assim, as contas estão vinculadas às diferentes agências em diferentes endereços, de várias cidades em vários estados. Uma conta é associada sempre a um cliente titular e a zero ou mais outros clientes (segundo titular, terceiro titular, etc).

As transações realizadas pelo cliente podem ser de retirada de valor de suas contas, de depósito e de transferência de valores entre contas de um mesmo banco.

Levando em consideração o padrão MVC [5], as entidades, presentes na Figura 2.2, fazem parte do modelo (o M do MVC) da aplicação. Assim, é necessário criar uma classe de Domínio¹³ para cada entidade da aplicação **ControleBancario**.

2.2.1 Classe de Domínio: Estado

A primeira classe de domínio a ser implementada é a classe **Estado** que representa os estados brasileiros.

- Para criá-la no IDE IntelliJ, selecione **New** ⇒ **Grails Domain Class** (Figura 2.3).
- Digite **br.ufscar.dc.dsw.Estado** como o nome da classe de domínio e clique em **Finish**. O IDE IntelliJ executa o comando **grails create-domain-class**. A classe de domínio **Estado.groovy** é criada no diretório **grails-app/domain**.
- Abra a classe **Estado** e insira os atributos (**nome** e **sigla**) dessa classe conforme apresentado no Código 2.3.

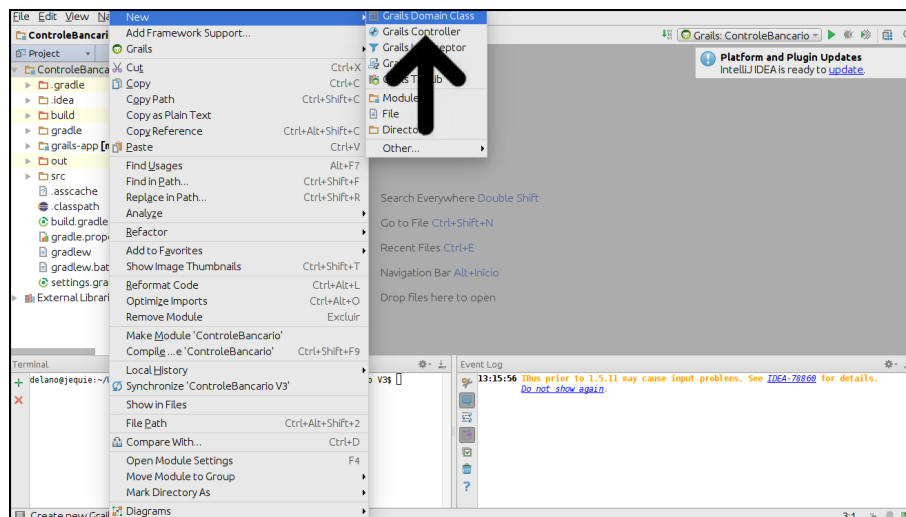


Figura 2.3: Criação da Classe de Domínio Estado.

Observações:

- O atributo **id** é gerado automaticamente pelo Grails. Logo, não é necessário incluí-lo na implementação da classe **Estado**;
- A classe de domínio **Estado** possui os seguintes atributos:
 - ◇ **nome** – que armazena o nome do estado;
 - ◇ **sigla** – que armazena a sigla do estado (por exemplo: **SP** para o estado de São Paulo);
- O método **toString()** retorna uma representação (por exemplo, o que é apresentada nas visões – páginas HTML) das instâncias das classes. No caso da implementação da classe de domínio **Estado**, o método **toString()** retorna a sigla do estado.

¹³Em Grails, os modelos são denominados de classes de domínio.

```

package br.ufscar.de.dsw

class Estado {

    static constraints = {
        nome (nullable: false, size: 1..20)
        sigla (nullable: false, size: 2..2)
    }

    String nome
    String sigla

    String toString() {
        return sigla
    }
}

```

Código 2.3: Classe de domínio **Estado**

Validação de Dados

O bloco **static constraints** permite que os desenvolvedores coloquem regras de validação nas classes de domínio. Por exemplo, é possível impor restrições sobre o tamanho máximo de um atributo String (por padrão, o tamanho é 255 caracteres). Além disso, é possível garantir que campos de texto (Strings) correspondem a um determinado padrão (como um endereço de e-mail ou URL). E por fim, é possível até mesmo tornar campos opcionais ou obrigatórios.

Além dessas validações, o bloco **static constraints** também possibilita definir a ordem em que os atributos de um modelo são apresentados nas visões associadas. O Código 2.3 descreve o uso do bloco **static constraints** para validar os atributos da classe **Estado** e definir a ordem em que os atributos dessa classe são apresentados. A Tabela 2.2 apresenta algumas restrições com exemplos de utilização e respectivas descrições.

Nome	Exemplo	Descrição
blank	nome(blank:false)	Coloque false se o valor da String não pode estar em branco.
email	e-mail(email:true)	Coloque true se a String necessitar ser um endereço de e-mail válido.
inList	sexo(inList:["F", "M"])	O valor deve estar contido na lista.
length	nome(length:5..15)	Usa uma faixa para restringir o tamanho de uma String ou array.
min	quantidade(min:0)	Define o valor mínimo.
matches	nome(matches:[a-zA-Z]/)	Verifica se corresponde a uma expressão regular fornecida.
max	quantidade(max:100)	Define o valor máximo.
nullable	preco(nullable:false)	Coloque false se o valor do atributo não poder ser nulo.
range	quantidade(range:5..15)	Valor deve estar dentro do intervalo especificado.
size	list(size:5..15)	Usa uma faixa para restringir o tamanho de uma coleção.
unique	nome(unique:true)	Defina como true se o valor do atributo não pode repetir.
url	homepage(url:true)	Defina como true se o valor da String precisar ser um endereço URL válido.

Tabela 2.2: Regras de restrições.

2.2.2 Classe de Domínio: Cidade

A próxima classe de domínio a ser criada é a classe **Cidade** que representa cidades brasileiras.

Crie, usando os mesmos passos da criação da classe de domínio **Estado** (Seção 2.2.1), a classe **Cidade** do pacote **br.ufscar.dc.dsw**. Abra a classe **Cidade**, insira os atributos e acrescente as suas validações, conforme apresentado no Código 2.4.

```
package br.ufscar.dc.dsw

class Cidade {

    static constraints = {
        nome (blank: false, size: 1..40)
        estado (nullable: false)
    }

    String nome
    Estado estado

    String toString() {
        StringBuilder sb = new StringBuilder()
        if (nome != null) {
            sb.append(nome)
            sb.append(" - ")
            sb.append(estados.sigla)
        }
        return sb.toString()
    }
}
```

Código 2.4: Classe de domínio **Cidade**

Observações:

- A classe de domínio **Cidade** possui os seguintes atributos:
 - ◇ **nome** – que armazena o nome da cidade;
 - ◇ **estado** – que armazena uma referência a uma instância da classe **Estado**. Esse atributo é um mapeamento unidirecional entre **Cidade** e **Estado**. Pelos requisitos da aplicação, não há necessidade de implementar um mapeamento bidirecional entre essas duas classes de domínio.

Uma importante característica do Grails é que programadores não precisam se preocupar em criar os métodos *getters* e *setters*. Eles são gerados pelo Groovy. Grails, por meio do mecanismo GORM (*Grails Object-Relational Mapping*), realiza um mapeamento automático entre modelos (classes de domínio) e tabelas em um SGBD.

Dessa forma, o Groovy gera outros métodos estáticos responsáveis pelas operações *CRUD* (*Create, Read, Update e Delete*):

- **Cidade.save()** armazena os dados na tabela Cidade (do SGBD).
- **Cidade.delete()** apaga os dados da tabela Cidade.
- **Cidade.list()** retorna uma lista de cidades.
- **Cidade.get()** retorna uma única instância da classe Cidade.

Todos esses e outros métodos estão disponíveis para os desenvolvedores. Note que **Cidade** não estende nenhuma classe pai nem implementa nenhuma interface. Graças aos recursos de metaprogramação Groovy, esses métodos simplesmente aparecem quando necessários. Apenas as classes presentes no diretório **grails-app/domain** (ou seja, classes de domínio) possuem esses métodos relacionados à persistência de dados: **save()**, **delete()**, **list()** e **get()**.

2.2.3 Classe de Domínio: Endereco

Crie, usando os mesmos passos da criação da classe de domínio **Estado** (Seção 2.2.1), a classe **Endereco** do pacote **br.ufscar.dc.dsw**. Abra a classe **Endereco**, insira os atributos e acrescente as suas validações, conforme apresentado no Código 2.5.

```
package br.ufscar.dc.dsw

class Endereco {

    static constraints = {
        CEP (blank: false, cep: true, size: 9..9)
        logradouro (blank: false, size: 1..40)
        numero (min: 0)
        complemento (nullable: true, size: 1..40)
        bairro (blank: false, size: 1..40)
        cidade (nullable: false)
    }

    String logradouro

    int numero

    String complemento

    String bairro

    String CEP

    Cidade cidade

    String toString() {
        return logradouro + ", " + numero +
            (complemento == null ? "" : " " + complemento) + ". " +
            bairro + " " + CEP + " " + cidade
    }
}
```

Código 2.5: Classe de domínio **Endereco**

Observações:

- A classe de domínio **Endereco** possui os seguintes atributos:
 - ◇ **logradouro** – que armazena o nome do logradouro;
 - ◇ **numero** – que armazena o número do endereço;
 - ◇ **complemento** – que armazena o complemento de endereço;
 - ◇ **bairro** – que armazena o bairro do endereço;
 - ◇ **CEP** – que armazena o CEP de endereço. A validação desse atributo utiliza a restrição **cep: true** definida pelo *plugin* **br-validation** instalado na Seção 2.1.1; e
 - ◇ **cidade** – que armazena uma referência a uma instância da classe **Cidade** (Seção 2.2.2). Esse atributo é um mapeamento unidirecional entre **Endereco** e **Cidade**. Pelos requisitos da aplicação, não há necessidade de implementar um mapeamento bidirecional entre essas duas classes de domínio.
- O método **toString()** retorna uma representação das instâncias das classes. No caso da implementação da classe de domínio **Endereco**, o método **toString()** retorna uma descrição textual do endereço (rua, número, complemento, bairro, CEP e cidade).

2.2.4 Classe de Domínio: Banco

A próxima classe de domínio a ser criada é a classe **Banco** que representa instituições bancárias. Crie, usando os mesmos passos da criação da classe de domínio **Estado** (Seção 2.2.1), a classe **Banco** do pacote **br.ufscar.dc.dsw**. Abra a classe **Banco**, insira os atributos e acrescente as suas validações, conforme apresentado no Código 2.6.

```
package br.ufscar.dc.dsw

class Banco {

    static hasMany = [agencias: Agencia, caixas: CaixaEletronico]

    static constraints = {
        numero (unique: true, min: 0)
        nome (blank: false, size: 1..20)
        CNPJ (blank: false, unique: true, cnpj: true, size: 18..18)
    }

    int numero
    String nome
    String CNPJ

    String toString() {
        return nome
    }
}
```

Código 2.6: Classe de domínio **Banco**

Observações:

- A classe de domínio **Banco** possui os seguintes atributos:
 - ◇ **numero** – que armazena o número (único) da instituição bancária;
 - ◇ **nome** – que armazena o nome da instituição bancária; e
 - ◇ **CNPJ** – que armazena o CNPJ da instituição bancária. A validação desse atributo utiliza a restrição **cnpj: true** definida pelo *plugin br-validation* instalado na Seção 2.1.1.
- O comando **static hasMany = [agencias: Agencia]** na classe de domínio **Banco** e o atributo **banco** na classe de domínio **Agencia** (Seção 2.2.5), foram utilizados em conjunto para implementar um mapeamento *um-para-muitos* entre essas classes.
- O comando **static hasMany = [caixas: CaixaEletronico]** na classe de domínio **Banco** e o atributo **banco** na classe de domínio **CaixaEletronico** (Seção 2.2.7), foram utilizados em conjunto para implementar um mapeamento *um-para-muitos* entre essas classes.
- O método **toString()** retorna uma representação das instâncias das classes. No caso da implementação da classe de domínio **Banco**, o método **toString()** retorna apenas o nome do banco.

2.2.5 Classe de Domínio: Agencia

A classe de domínio **Agencia** representa agências de instituições bancárias. Crie a classe **Agencia** do pacote **br.ufscar.dc.dsw**. Abra a classe **Agencia**, insira os atributos e acrescente as suas validações, conforme apresentado no Código 2.4.

```
package br.ufscar.dc.dsw

class Agencia {

    static hasMany = [gerentes: Gerente]

    static constraints = {
        banco (nullable: false)
        numero (blank: false, min: 0)
        nome (blank: false, size: 1..20)
        endereco (nullable: false)
    }

    int numero
    String nome
    Endereco endereco
    Banco banco

    String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append(numero)
        sb.append(" - ")
        sb.append(banco)
        return sb.toString();
    }
}
```

Código 2.7: Classe de domínio **Agencia**

Observações:

- A classe de domínio **Agencia** possui os seguintes atributos:
 - ◇ **numero** – que armazena o número da agência bancária;
 - ◇ **nome** – que armazena o nome da agência bancária;
 - ◇ **endereco** – que armazena uma referência a uma instância da classe de domínio **Endereco** (Seção 2.2.3). Esse atributo é um mapeamento unidirecional entre **Agencia** e **Endereco**. Pelos requisitos da aplicação, não há necessidade de implementar um mapeamento bidirecional entre essas duas classes de domínio; e
 - ◇ **banco** – que armazena uma referência a uma instância da classe de domínio **Banco** (Seção 2.2.4). Esse atributo representa a cardinalidade “*um*” do relacionamento *um-para-muitos* entre as classes de domínio **Banco** e **Agencia**.
- O comando **static hasMany = [gerentes: Gerente]** na classe de domínio **Agencia** e o atributo **agencia** na classe de domínio **Gerente** (Seção 2.2.6), foram utilizados em conjunto para implementar o mapeamento *um-para-muitos* entre as classes **Agencia** e **Gerente**.
- O método **toString()** retorna uma representação das instâncias das classes. No caso da implementação da classe de domínio **Agencia**, o método **toString()** retorna o número da agência concatenado com o nome do banco.

2.2.6 Classe de Domínio: Gerente

A classe de domínio **Gerente** representa gerentes de agências de instituições bancárias. Crie a classe **Gerente** do pacote **br.ufscar.dc.dsw**. Abra a classe **Gerente**, insira os atributos e acrescente as suas validações, conforme apresentado no Código 2.8.

```
package br.ufscar.dc.dsw

class Gerente {

    static constraints = {
        nome (blank: false, size: 1..30)
        rg (blank: false, size: 1..12)
        CPF (blank: false, unique: true, cpf: true, size: 14..14)
        agencia (nullable: false)
    }

    String nome
    String rg
    String CPF
    Agencia agencia

    String toString() {
        return nome + " " + CPF
    }
}
```

Código 2.8: Classe de domínio **Gerente**

Observações:

- A classe de domínio **Gerente** possui os seguintes atributos:
 - ◇ **nome** – que armazena o nome do gerente;
 - ◇ **rg** – que armazena o RG do gerente;
 - ◇ **CPF** – que armazena o CPF do gerente. A validação desse atributo utiliza a restrição **cpf: true** definida pelo *plugin* **br-validation** instalado na Seção 2.1.1; e
 - ◇ **agencia** – que armazena uma referência a uma instância da classe de domínio **Agencia** (Seção 2.2.5). Esse atributo representa a cardinalidade “*um*” do relacionamento *um-para-muitos* entre as classes de domínio **Agencia** e **Gerente**.
- O método **toString()** retorna uma representação das instâncias das classes. No caso da implementação da classe de domínio **Gerente**, o método **toString()** retorna o nome do gerente concatenado com seu respectivo CPF.

2.2.7 Classe de Domínio: CaixaEletronico

A classe de domínio **CaixaEletronico** representa caixas eletrônicos, pertencentes às instituições bancárias, onde transações bancárias (Seção 2.2.8) podem ser realizadas. Crie a classe **CaixaEletronico** do pacote **br.ufscar.dc.dsw**. Abra a classe **CaixaEletronico**, insira os atributos e acrescente as suas validações, conforme apresentado no Código 2.9.

```
package br.ufscar.dc.dsw

class CaixaEletronico {

    static hasMany = [transacoes: Transacao]

    static constraints = {
        banco (nullable: false)
        endereco (nullable: false)
    }

    Endereco endereco
    Banco banco

    String toString() {
        return banco.toString() + " - " + endereco.toString();
    }
}
```

Código 2.9: Classe de domínio **CaixaEletronico**

Observações:

- A classe de domínio **CaixaEletronico** possui os seguintes atributos:
 - ◇ **banco** – que armazena uma referência a uma instância da classe de domínio **Banco** (Seção 2.2.4). Ou seja, esse atributo representa a cardinalidade “*um*” do relacionamento *um-para-muitos* entre as classes de domínio **Banco** e **CaixaEletronico**; e
 - ◇ **endereco** – que armazena uma referência a uma instância da classe de domínio **Endereco** (Seção 2.2.3). Esse atributo é um mapeamento unidirecional entre **CaixaEletronico** e **Endereco**. Pelos requisitos da aplicação, não há necessidade de implementar um mapeamento bidirecional entre essas duas classes de domínio.
- O comando **static hasMany = [transacoes: Transacao]** na classe de domínio **CaixaEletronico** e o atributo **caixaEletronico** na classe de domínio **Transacao** (Seção 2.2.8), foram utilizados em conjunto para implementar o mapeamento *um-para-muitos* entre essas classes de domínio.
- O método **toString()** retorna uma representação das instâncias das classes. No caso da implementação da classe de domínio **CaixaEletronico**, o método **toString()** retorna o nome do banco concatenado com o endereço do caixa eletrônico.

2.2.8 Classe de Domínio: Transacao

A classe de domínio **Transacao** representa transações realizadas em contas bancárias vinculadas a clientes do banco. Pelos requisitos da aplicação, todas as transações bancárias são realizadas em um caixa eletrônico.

Crie a classe **Transacao** do pacote **br.ufscar.dc.dsw**. Abra a classe **Transacao**, insira os atributos e acrescente as suas validações, conforme apresentado no Código 2.10.

```
package br.ufscar.dc.dsw

class Transacao {

    public static final String CRÉDITO = "CRÉDITO"
    public static final String DÉBITO = "DÉBITO"

    static constraints = {
        contaCliente (nullable: false)
        caixaEletronico (nullable: false)
        valor (nullable: false, min: 0.1d)
        data (nullable: false)
        quem (nullable: false)
        motivo (nullable: false)
        tipo (nullable: false, inList: [CRÉDITO,DÉBITO])
    }

    Date data

    double valor

    String quem

    String motivo

    String tipo

    ContaCliente contaCliente

    CaixaEletronico caixaEletronico

    String toString() {
        return "[" + tipo + "] - " + motivo + " - R$ " + valor
    }
}
```

Código 2.10: Classe de domínio **Transacao**

Observações:

- A classe de domínio **Transacao** possui os seguintes atributos:
 - ◇ **data** – que armazena a data em que ocorreu a transação;
 - ◇ **valor** – que armazena o valor da transação;
 - ◇ **quem** – que armazena *quem* realizou a transação;
 - ◇ **motivo** – que armazena *o motivo* (saque, depósito, transferência) da transação;
 - ◇ **tipo** – que armazena o tipo de transação: **Crédito** ou **Débito**;
 - ◇ **contaCliente** – que armazena uma referência a uma instância da classe de domínio **ContaCliente** (Seção 2.2.9). Esse atributo representa a cardinalidade “*um*” do relacionamento *um-para-muitos* entre as classes de domínio **ContaCliente** e **Transacao**.
 - ◇ **caixaEletronico** – que armazena uma referência a uma instância da classe de domínio **CaixaEletronico** (Seção 2.2.7). Ou seja, esse atributo representa a cardinalidade “*um*” do relacionamento *um-para-muitos* entre as classes de domínio **CaixaEletronico** e **Transacao**.

2.2.9 Classe de Domínio: ContaCliente

A classe de domínio **ContaCliente** materializa o relacionamento *muitos-para-muitos* entre as classes de domínio **Conta** e **Cliente**. O relacionamento *muitos-para-muitos* entre **Conta** e **Cliente** é obtido ao implementar dois relacionamentos *um-para-muitos*: (i) **Conta** x **ContaCliente** e (ii) **Cliente** x **ContaCliente**.

Crie a classe **ContaCliente** do pacote **br.ufscar.dc.dsw**. Abra a classe **ContaCliente**, insira os atributos e acrescente as suas validações, conforme apresentado no Código 2.11.

```
package br.ufscar.dc.dsw

class ContaCliente {

    static hasMany = [transacoes: Transacao]

    static constraints = {
        cliente (nullable: false)
        conta (nullable: false, unique: 'cliente')
        titular (nullable: false)
    }

    boolean titular
    Conta conta
    Cliente cliente

    String toString() {
        return cliente.toString() + " X " + conta
    }
}
```

Código 2.11: Classe de domínio **ContaCliente**

Observações:

- A classe de domínio **ContaCliente** possui os seguintes atributos:
 - ◇ **titular** – que determina se o cliente é titular da conta;
 - ◇ **conta** – que armazena uma referência a uma instância da classe de domínio **Conta** (Seção 2.2.10). Esse atributo representa a cardinalidade “*um*” do relacionamento *um-para-muitos* entre as classes de domínio **Conta** e **ContaCliente**; e
 - ◇ **cliente** – que armazena uma referência a uma instância da classe de domínio **Cliente** (Seção 2.2.13). Esse atributo representa a cardinalidade “*um*” do relacionamento *um-para-muitos* entre as classes de domínio **Cliente** e **ContaCliente**.
- O comando **static hasMany = [transacoes: Transacao]** na classe de domínio **ContaCliente** e o atributo **conta** na classe de domínio **Transacao** (Seção 2.2.8), foram utilizados em conjunto para implementar um mapeamento *um-para-muitos* entre essas classes.
- O método **toString()** retorna uma representação das instâncias das classes. No caso da implementação da classe de domínio **ContaCliente**, o método **toString()** retorna a concatenação das representações da conta e do cliente associados à essa instância da classe **ContaCliente**.

2.2.10 Classe de Domínio: Conta

A classe de domínio **Conta** representa contas bancárias. Pelos requisitos da aplicação, a classe **Conta** é abstrata e portanto instâncias de **Conta** não podem ser criadas - apenas de suas subclasses: **ContaCorrente** (Seção 2.2.11) e **ContaPoupanca** (Seção 2.2.12).

Crie a classe **Conta** do pacote **br.ufscar.dc.dsw**. Abra a classe **Conta**, insira os atributos e acrescente as suas validações, conforme apresentado no Código 2.12.

```
package br.ufscar.dc.dsw

abstract class Conta {

    static hasMany = [contasCliente: ContaCliente]

    static constraints = {
        numero (blank: false)
        agencia (nullable: false)
        saldo (nullable: false, min: 0.0d)
        abertura (nullable: false)
    }

    static mapping = {
        tablePerHierarchy false
    }

    Agencia agencia

    String numero

    double saldo

    Date abertura

}
```

Código 2.12: Classe de domínio **Conta**

Observações:

- A classe de domínio abstrata **Conta** possui os seguintes atributos:
 - ◇ **agencia** – que armazena uma referência a uma instância da classe **Agencia** (Seção 2.2.5). Esse atributo é um mapeamento unidirecional entre **Conta** e **Agencia**. Pelos requisitos da aplicação, não há necessidade de implementar um mapeamento bidirecional entre essas duas classes de domínio;
 - ◇ **numero** – que armazena o número da conta bancária;
 - ◇ **saldo** – que armazena o saldo da conta bancária; e
 - ◇ **abertura** – que armazena a data de abertura da conta bancária;
- O comando **static hasMany = [contasCliente: ContaCliente]** na classe de domínio **Conta** e o atributo **conta** na classe de domínio **ContaCliente** (Seção 2.2.9), foram utilizados em conjunto para implementar um mapeamento *um-para-muitos* entre essas classes.

2.2.11 Classe de Domínio: ContaCorrente

A classe de domínio **ContaCorrente**, subclasse de **Conta**, representa contas correntes. O atributo **limite** representa o limite (cheque especial) que a instituição bancária disponibiliza ao cliente caso necessário. Crie a classe **ContaCorrente** do pacote **br.ufscar.dc.dsw**. Abra a classe **ContaCorrente**, insira os atributos e acrescente as suas validações, conforme apresentado no Código 2.13.

```
package br.ufscar.dc.dsw

class ContaCorrente extends Conta{

    static constraints = {
        numero (blank: false)
        agencia (nullable: false)
        saldo (nullable: false, min: 0.0d)
        limite (nullable: false, min: 0.0d)
        abertura (nullable: false)
    }

    double limite

    String toString() {
        return "(Conta Corrente) " + numero
    }
}
```

Código 2.13: Classe de domínio **ContaCorrente**

2.2.12 Classe de Domínio: ContaPoupanca

A classe de domínio **ContaPoupanca**, subclasse de **Conta**, representa contas de poupança, que também são chamadas de *cadernetas de poupança*. Crie a classe **ContaPoupanca** do pacote **br.ufscar.dc.dsw**. Abra a classe **ContaPoupanca**, insira os atributos e acrescente as suas validações, conforme apresentado no Código 2.14.

```
package br.ufscar.dc.dsw

class ContaPoupanca extends Conta{

    static constraints = {
        numero (blank: false)
        agencia (nullable: false)
        saldo (nullable: false, min: 0.0d)
        juros (min: 0.0d)
        correcao (min: 0.0d)
        dia (blank: false, range: 1..28)
        abertura (nullable: false)
    }

    byte dia
    double correcao
    double juros

    String toString() {
        return "(Poupança) " + numero
    }
}
```

Código 2.14: Classe de domínio **ContaPoupanca**

A classe de domínio **ContaPoupanca** possui os seguintes atributos: (i) **dia** que armazena o dia de aniversário da caderneta de poupança; (ii) **correcao** que armazena o valor de correção mensal da caderneta de poupança; e (iii) **juros** que armazena o valor dos juros de reajuste mensal da caderneta de poupança.

GORM: Relacionamento de herança

O mecanismo GORM (*Grails Object-Relational Mapping*) dá suporte à implementação de herança de classes de entidades abstratas e concretas. Ou seja, possibilita o mapeamento do modelo de objetos para o modelo de dados relacional e vice-versa

Por *default*, o mecanismo GORM utiliza a estratégia de mapeamento *table-per-hierarchy* (ver Figura 2.4(a)) que consiste em uma única tabela para toda a hierarquia. Essa tabela armazena os atributos da classe pai (**Conta**) bem como os atributos específicos de cada subclasse **ContaCorrente** e **ContaPoupanca** e contém ainda uma coluna discriminadora denominada **class**. Essa coluna mantém valores de marcação que informam ao framework *hibernate* qual subclasse instanciar durante a recuperação dos dados do SGBD.

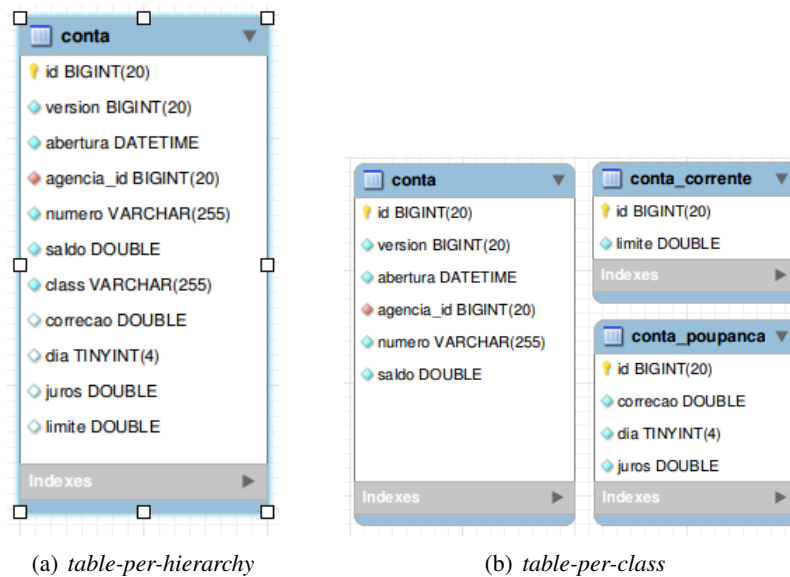


Figura 2.4: GORM: Mapeamento de Hierarquia

Porém, se o *default* da estratégia de mapeamento *table-per-hierarchy* não for a mais adequada para sua aplicação, o desenvolvedor pode utilizar um mapeamento alternativo – a estratégia *table-per-class* (Figura 2.4(b)). Nesse caso, é criada uma tabela para a classe pai assim como para cada classe filha. Em nosso exemplo, a estratégia *table-per-class* é habilitada ao incluir o trecho apresentado a seguir na classe **Conta** (pai da hierarquia).

```
static mapping = {
    tablePerHierarchy false
}
```

Pela Figura 2.4(b), pode-se observar que foi criada uma tabela que armazena os atributos específicos de cada classe: **Conta** e suas subclasses **ContaCorrente** e **ContaPoupanca**.

2.2.13 Classe de Domínio: **Cliente**

A classe de domínio **Cliente** representa clientes (correntistas) de instituições bancárias. Pelos requisitos da aplicação, a classe **Cliente** é abstrata e portanto instâncias de **Cliente** não podem ser criadas. Apenas de suas subclasses **ClienteFisico** e **ClienteJuridico** podem ser instanciadas.

Crie a classe **Cliente** do pacote **br.ufscar.dc.dsw**. Na classe **Cliente**, insira os atributos e acrescente as suas validações, conforme apresentado no Código 2.15.

```
package br.ufscar.dc.dsw

abstract class Cliente {

    static hasMany = [contasCliente: ContaCliente]

    public static final String ATIVO = "Ativo"
    public static final String INATIVO = "Inativo"

    static constraints = {
        nome (blank: false, size: 1..30)
        endereco (nullable: false)
        dtMoradia (blank: false)
        status (blank: false, inList: [ATIVO, INATIVO])
    }

    static mapping = {
        tablePerHierarchy false
    }

    String nome
    String status
    Endereco endereco
    Date dtMoradia
}
```

Código 2.15: Classe de domínio **Cliente**

Observações:

- A classe de domínio abstrata **Cliente** possui os seguintes atributos:
 - ◇ **nome** – que armazena o nome do cliente;
 - ◇ **status** – que armazena o *status* do cliente: **Ativo** ou **Inativo**;
 - ◇ **endereco** – que armazena uma referência a uma instância da classe **Endereco** (Seção 2.2.3). Esse atributo é um mapeamento unidirecional entre **Cliente** e **Endereco**. Pelos requisitos da aplicação, não há necessidade de implementar um mapeamento bidirecional entre essas duas classes de domínio; e
 - ◇ **dtMoradia** – que armazena desde quando o cliente reside no endereço armazenado no atributo **endereco**.
- O comando **static hasMany = [contasCliente: ContaCliente]** na classe **Cliente** e o atributo **cliente** na classe **ContaCliente**, foram utilizados em conjunto para implementar um mapeamento *um-para-muitos* entre essas classes.
- Análoga à classe **Conta**, a classe de domínio **Cliente** também adota a estratégia *table-per-class* no mapeamento objeto-relacional. Portanto, será criada uma tabela para a classe pai (**Cliente**) assim como para cada subclasses: **ClienteFisico** e **ClienteJuridico**.

2.2.14 Classe de Domínio: **ClienteFisico**

A classe de domínio **ClienteFisico**, subclasse de **Cliente**, representa clientes físicos de instituições bancárias. Os atributos **rg** e **CPF** armazenam respectivamente a identidade e o número do Cadastro de Pessoa Física desses clientes.

Crie a classe **ClienteFisico** do pacote **br.ufscar.dc.dsw**. Abra a classe **ClienteFisico**, insira os atributos e acrescente as suas validações, conforme apresentado no Código 2.16. Salienta-se que a validação do atributo **CPF** utiliza a restrição **cpf: true** definida pelo *plugin* **br-validation** instalado na Seção 2.1.1.

```
package br.ufscar.dc.dsw

class ClienteFisico extends Cliente{

    static constraints = {
        nome (blank: false, size: 1..30)
        rg (blank: false, size: 1..12)
        CPF (blank: false, unique:true, cpf: true, size: 14..14)
        endereco (nullable: false)
        dtMoradia (blank: false)
        status (blank: false, inList: [ATIVO, INATIVO])
    }

    String rg
    String CPF

    String toString() {
        return CPF
    }
}
```

Código 2.16: Classe de domínio **ClienteFisico**

2.2.15 Classe de Domínio: **ClienteJuridico**

A classe de domínio **ClienteJurídico**, subclasse de **Cliente**, representa clientes jurídicos de instituições bancárias. O atributo **CNPJ** armazena o número do Cadastro Nacional de Pessoas Jurídicas desses clientes.

Crie a classe **ClienteJuridico** do pacote **br.ufscar.dc.dsw**. Abra a classe **ClienteJuridico**, insira os atributos e acrescente as suas validações, conforme apresentado no Código 2.17. Salienta-se que a validação do atributo **CNPJ** utiliza a restrição **cnnpj: true** definida pelo *plugin* **br-validation** instalado na Seção 2.1.1.

```
package br.ufscar.dc.dsw

class ClienteJuridico extends Cliente{

    static constraints = {
        nome (blank: false, size: 1..30)
        CNPJ (blank: false, unique:true, cnnpj: true, size: 18..18)
        endereco (nullable: false)
        dtMoradia (blank: false)
        status (blank: false, inList: [ATIVO, INATIVO])
    }

    String CNPJ

    String toString() {
        return CNPJ
    }
}
```

Código 2.17: Classe de domínio **ClienteJuridico**

2.3 Scaffolding

Uma vez que as classes de domínio estão criadas, é preciso criar os controladores e as visões relacionados a essas classes de domínio. Na geração de tais artefatos de software será utilizada a abordagem *scaffolding* que é um termo cunhado pelo framework Rails e adotado pelo Grails para a geração dos artefatos (controladores, visões, etc.) que implementam as operações CRUD. Esse termo foi estendido e atualmente é possível criar código de autenticação e autorização, testes unitários, dentre outras operações. Em Grails, o *scaffolding* pode ser dinâmico ou estático. Discutiremos essas duas abordagens distintas nas próximas subseções.

2.3.1 Scaffolding Dinâmico

No *scaffolding* dinâmico, as visões são geradas em tempo de execução. Essa abordagem simplifica o desenvolvimento, pois nenhum código relacionado aos controladores e visões precisa ser desenvolvido. O *scaffolding* dinâmico pode servir a vários propósitos, por exemplo, é bastante útil na criação das interfaces de administração de uma aplicação web. No entanto, ele não é útil quando a equipe web deseja personalizar o sistema para seus propósitos, principalmente as visões geradas em tempo de execução.

- Para criar um controlador (empregando *scaffolding dinâmico*), relacionado à classe de domínio **Transacao**, no IDE IntelliJ: Selecione **New** ⇒ **Grails Controller** (Figura 2.5).
- Digite **br.ufscar.dc.dsw.Transacao** como o nome do controlador e clique em **Finish**. O IDE IntelliJ executa o comando **grails create-controller**. O controlador **TransacaoController.groovy** é criado no diretório **grails-app/controllers**.
- Abra o controlador **TransacaoController** e implemente-o conforme apresentado no Código 2.18.

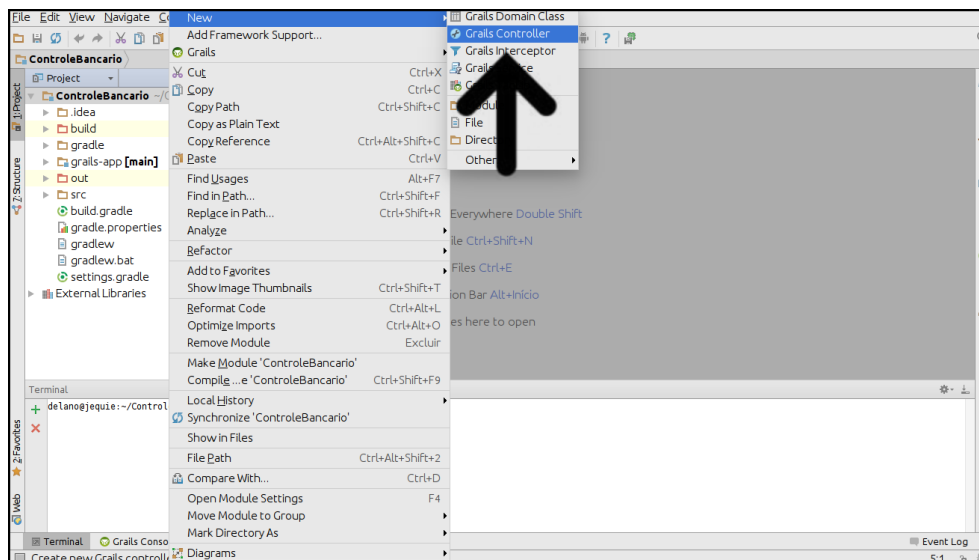


Figura 2.5: Criação do controlador (*scaffolding* dinâmico).

```
package br.ufscar.dc.dsw  
  
class TransacaoController {  
  
    static scaffold = true  
  
}
```

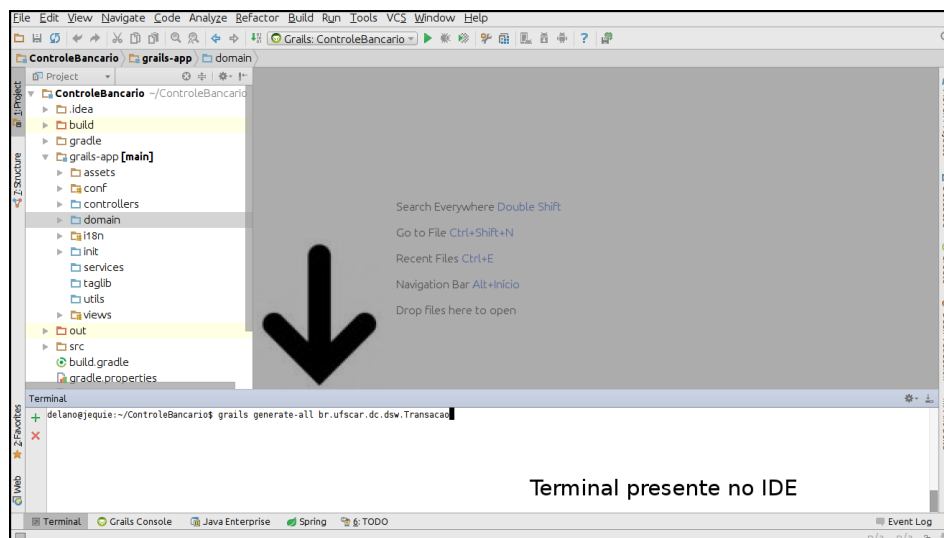
Código 2.18: Controlador **TransacaoController** (1)

2.3.2 Scaffolding Estático

O *scaffolding* estático produz, a partir de *templates*, o código dos controladores e visões que podem ser personalizados pela equipe web.

Levando em consideração esses aspectos, nesse tutorial adotou-se o *scaffolding* estático, pois algumas customizações nos controladores e visões serão necessárias no desenvolvimento da aplicação **ControleBancario**.

- Para criar um controlador e as visões (empregando *scaffolding* estático) relacionado à classe de domínio **Transacao**, no IDE IntelliJ: Abra o **Terminal** e execute o comando **grails generate-all br.ufscar.dc.dsw.Transacao**. O controlador **TransacaoController.groovy** é criado no diretório **grails-app/controllers** e o correspondente conjunto de *Groovy Server Pages* (GSPs) no diretório **grails-app/views/transacao** (Figura 2.6).

Figura 2.6: Criação do controlador e das visões (*scaffolding* estático).

Observação: Para cada método correspondente a uma ação em um controlador é criada uma correspondente visão (arquivo com extensão **.gsp**). Por exemplo, a ação **show()** tem o correspondente **show.gsp**, enquanto a ação **create()** tem o correspondente **create.gsp**.

Figura 2.7 ilustra os controladores e visões gerados pelo *scaffolding* da classe de domínio **Transacao**. Destaca-se os benefícios do paradigma *Convention Over Configuration* em ação em que nenhum arquivo XML é necessário para associar esses elementos. As classes de domínio estão associadas a controladores baseado em seus respectivos nomes (**Transacao.groovy** → **TransacaoController.groovy**). Conforme já mencionado, toda ação em um controlador é associada a uma visão baseada em seu nome (**index** → **index.gsp**). Desenvolvedores podem configurar para que o mapeamento seja feito de outra maneira. No entanto, na maioria das vezes, basta seguir a convenção e a aplicação funcionará corretamente sem maiores configurações.

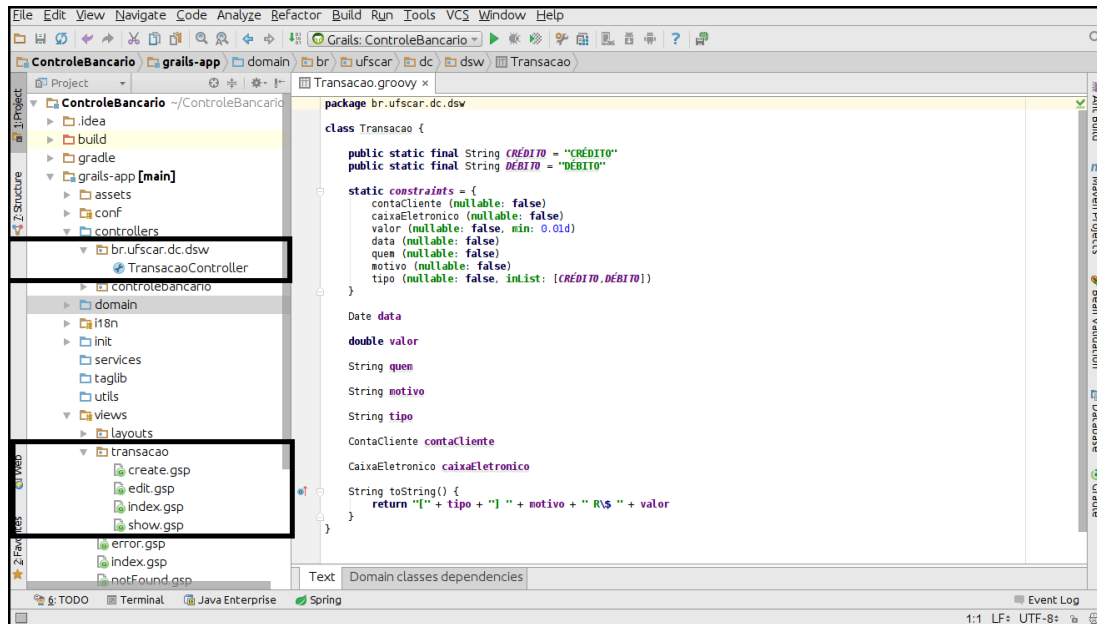


Figura 2.7: Scaffolding estático das classes de domínio.

! Um bom exercício consiste em utilizar o *scaffolding estático* para gerar o controlador e as visões das demais classes de domínio.

Lembrar que não se deve gerar os controladores e as visões para as classes **Cliente** e **Conta**. Essas classes são abstratas e não terão as operações de criação, acesso, atualização e remoção (CRUD).

2.3.3 Convenção na nomenclatura de URLs

Grails usa uma convenção (Figura 2.8) para automaticamente configurar o caminho para uma ação em particular. A URL a seguir pode ser entendida da seguinte forma:

- “Execute a ação **show()** do controlador **transacao** – um dos controladores da aplicação **ControleBancario** hospedada na porta 8080 do servidor **localhost**”.

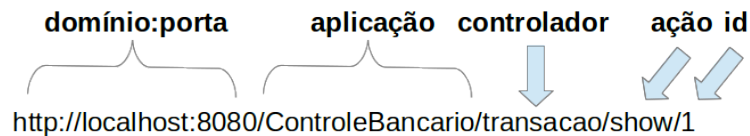


Figura 2.8: Convenção na nomenclatura de URLs.

A regra principal de roteamento no Grails forma URLs segundo o padrão **/controlador/ação/id**, onde **controlador** é o controlador (da arquitetura MVC) responsável por atender a requisição especificada por aquela URL, **ação** é um método dentro do controlador e **id** é um parâmetro opcional passado para identificar um objeto qualquer sobre o qual a ação será efetuada. A visão associada a essa ação geralmente é invocada (o controlador pode redirecionar para outra visão).

Por exemplo, **/transacao/edit/1** invoca a visão **transacao/edit.gsp**.

2.3.4 Controlador: TransacaoController

Para cada classes de entidade (classe de domínio) persistente no banco de dados tem-se um controlador. Por exemplo, durante o *scaffolding* estático, o controlador **TransacaoController** foi gerado com as seguintes ações (Código 2.19):

- A ação **index()** é responsável por retornar a lista de instâncias. Essa lista é repassada para visão **index.gsp** que a apresenta em uma página HTML;
- A ação **show()** é responsável por retornar os atributos de uma instância. Essa instância é repassada para a visão **show.gsp** que a apresenta em uma página HTML;
- A ação **create()** é responsável por criar uma instância que é repassada (retornada) para a visão **create.gsp** (uma página que contém um formulário HTML);
- Quando o formulário é submetido, a ação **save()** valida os dados e caso, tenha sucesso, grava a instância no banco de dados e redireciona para a ação **show()**. Por outro lado, se os dados são inválidos, a ação **save()** renderiza a visão **create.gsp** novamente para que o usuário corrija os erros encontrados na validação;
- A ação **edit()** é responsável por recuperar uma instância a ser atualizada posteriormente. A instância recuperada é repassada (retornada) para a visão **edit.gsp** (uma página que contém um formulário HTML);
- Quanto o formulário é submetido o método **update()** valida os dados e caso, tenha sucesso, atualiza a instância no banco de dados e redireciona para a ação **show()**. Por outro lado, se os dados são inválidos, a ação **update()** renderiza a visão **edit.gsp** novamente para que o usuário corrija os erros encontrados na validação; e

```

1  @Transactional(readOnly = true)
2  class TransacaoController {
3
4      static allowedMethods = [save: "POST", update: "PUT", delete: "DELETE"]
5
6      def index(Integer max) {
7          params.max = Math.min(max ?: 10, 100)
8          respond Transacao.list(params), model:[transacaoCount: Transacao.count()]
9      }
10
11     def show(Transacao transacao) {
12         respond transacao
13     }
14
15     def create() {
16         respond new Transacao(params)
17     }
18
19     @Transactional
20     def save(Transacao transacao) {
21         if (transacao == null) {
22             transactionStatus.setRollbackOnly()
23             notFound()
24             return
25         }
26         if (transacao.hasErrors()) {
27             transactionStatus.setRollbackOnly()
28             respond transacao.errors, view:'create'
29             return
30         }
31         transacao.save flush:true
32         request.withFormat {
33             form multipartForm {
34                 flash.message = message(code: 'default.created.message', args: [message(code: 'transacao.label',
35                                     default: 'Transacao'), transacao.id])
36                 redirect transacao
37             }
38             '*' { respond transacao, [status: CREATED] }
39         }
40
41     def edit(Transacao transacao) {
42         respond transacao
43     }
44
45     @Transactional
46     def update(Transacao transacao) {
47         if (transacao == null) {
48             transactionStatus.setRollbackOnly()
49             notFound()
50             return
51         }
52         if (transacao.hasErrors()) {
53             transactionStatus.setRollbackOnly()
54             respond transacao.errors, view:'edit'
55             return
56         }
57         transacao.save flush:true
58         request.withFormat {
59             form multipartForm {
60                 flash.message = message(code: 'default.updated.message', args: [message(code: 'transacao.label',
61                                     default: 'Transacao'), transacao.id])
62                 redirect transacao
63             }
64             '*' { respond transacao, [status: OK] }
65         }
66
67     @Transactional
68     def delete(Transacao transacao) {
69
70         if (transacao == null) {
71             transactionStatus.setRollbackOnly()
72             notFound()
73             return
74         }
75         transacao.delete flush:true
76         request.withFormat {
77             form multipartForm {
78                 flash.message = message(code: 'default.deleted.message', args: [message(code: 'transacao.label',
79                                     default: 'Transacao'), transacao.id])
80                 redirect action:"index", method:"GET"
81             }
82             '*' { render status: NO_CONTENT }
83         }
84
85     protected void notFound() {
86         request.withFormat {
87             form multipartForm {
88                 flash.message = message(code: 'default.not.found.message', args: [message(code: 'transacao.label',
89                                     default: 'Transacao'), params.id])
90                 redirect action:"index", method:"GET"
91             }
92             '*' { render status: NOT_FOUND }
93         }
94     }

```

Código 2.19: Controlador TransacaoController

- Por fim, a ação **delete()** remove uma instância do banco de dados através da invocação do método **delete (flush:true)** no objeto a ser removido.

Seguem alguns detalhes importantes:

Três Rs: Em Grails, ações normalmente terminam em uma das três formas, iniciadas com a letra R.

Redirecionamento – a ação solicita que o pedido seja atendido por uma outra ação.

Renderização – envia algum conteúdo (texto simples, XML, JSON, HTML, etc) para ser renderizado pelo navegador.

Retorno – o retorno geralmente é realizado de forma explícita. Outras vezes o retorno é implícito, como vemos no caso de **index()** que retorna uma lista de transações e o tamanho da lista de transações.

2.3.5 Groovy Server Pages (GSPs)

Durante o *scaffolding* estático, as visões relacionadas (**grails-app/views/transacao**) ao controlador **TransacaoController** foram criadas. Essas visões são *Groovy Server Pages (GSPs)* que podem ser definidas como um arquivo HTML básico que contém alguma *tags* Grails (elementos **<g: .. />**). Para desenvolvedores Java, as *tags* Grails são semelhantes às *tags* disponíveis na **JavaServer Pages Standard Tag Libraries (JSTL)**¹⁴

Uma característica importante de GSPs é o *data-binding* entre o modelo e as variáveis acessíveis pela visão durante a renderização das páginas HTML. Assim, um modelo é um mapa (chave, valor) que a visão utiliza. Por exemplo, o Código 2.20 (visão **index.gsp**) tem acesso a lista de transações (variável **transacaoList**) e o tamanho da lista (variável **transacaoCount**) que são retornados pela ação **index()** – Código 2.19, linha 8.

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="layout" content="main" />
    <g:set var="entityName" value="${ message(code: 'transacao.label', default: 'Transacao')}" />
    <title><g:message code="default.list.label" args="[entityName]" /></title>
  </head>
  <body>
    <a href="#list-transacao" class="skip" tabindex="-1"><g:message code="default.link.skip.label" default="Skip to content&hellip;" /></a>
    <div class="nav" role="navigation">
      <ul>
        <li><a class="home" href="${createLink(uri: '/')}"><g:message code="default.home.label"/></a></li>
        <li><g:link class="create" action="create"><g:message code="default.new.label" args="[entityName]" /></g:link></li>
      </ul>
    </div>
    <div id="list-transacao" class="content scaffold-list" role="main">
      <h1><g:message code="default.list.label" args="[entityName]" /></h1>
      <g:if test="${flash.message}">
        <div class="message" role="status">${flash.message}</div>
      </g:if>
      <g:table collection="${transacaoList}" />
      <div class="pagination">
        <g:paginate total="${transacaoCount ?: 0}" />
      </div>
    </div>
  </body>
</html>
```

Código 2.20: Visão **transacao/index.gsp**

O *data-binding* é realizado através da utilização da *GSP Expression Language (EL)* que facilita o acesso aos modelos através de uma sintaxe simples tais como **\${transacaoCount}** para uma variável simples ou **\${transacao.data}** para um atributo de uma instância de objeto.

¹⁴<http://www.oracle.com/technetwork/java/index-jsp-135995.html>

2.4 Executando a aplicação

Após gerar o CRUD das entidades, através do *scaffolding* estático ou do *scaffolding* dinâmico, a aplicação pode ser executada. Porém, antes de executar a aplicação as instâncias das entidades podem ser criadas. No caso, serão criados instâncias das entidades (**Estado**, **Cidade**, **Endereco**, etc) na classe **Bootstrap.groovy** que encontra-se no diretório **grails-app/init**. Essa classe é executada durante o *boot* da aplicação e serve, entre outros propósitos, para inicializar a aplicação por exemplo, criando algumas instâncias de objetos.

A implementação da classe **Bootstrap.groovy** é apresentado nos Códigos 2.21 a 2.25. O trecho apresentado no Código 2.21, popula instâncias das classes **Estado** e **Cidade**.

```
import br.ufscar.dc.dsw.Agency
import br.ufscar.dc.dsw.Banco
import br.ufscar.dc.dsw.CaixaEletronico
import br.ufscar.dc.dsw.Cidade
import br.ufscar.dc.dsw.Cliente
import br.ufscar.dc.dsw.ClienteFisico
import br.ufscar.dc.dsw.ClienteJuridico
import br.ufscar.dc.dsw.ContaCliente
import br.ufscar.dc.dsw.ContaCorrente
import br.ufscar.dc.dsw.ContaPoupanca
import br.ufscar.dc.dsw.Endereco
import br.ufscar.dc.dsw.Estado
import br.ufscar.dc.dsw.Gerente
import br.ufscar.dc.dsw.Transacao

class Bootstrap {

    def init = { servletContext ->

        def sp = new Estado(sigla: 'SP', nome: 'São Paulo')

        sp.save()
        if (sp.hasErrors()) {
            println sp.errors
        }

        println 'populando estados - ok'

        def sanca = new Cidade(nome: 'São Carlos', estado: sp)

        sanca.save()
        if (sanca.hasErrors()) {
            println sanca.errors
        }

        def sampa = new Cidade(nome: 'São Paulo', estado: sp)

        sampa.save()
        if (sampa.hasErrors()) {
            println sampa.errors
        }

        println 'populando cidades - ok'
    }
}
```

Código 2.21: **Bootstrap.groovy** (1)

O trecho apresentado no Código 2.22, popula instâncias das classes **Endereco**, **Banco** e **Agencia**.

```
def end1 = new Endereco(logradouro: 'R. Conde do Pinhal', numero: 1909,
    bairro: 'Centro', CEP: '13560-648', cidade: sanca)
end1.save()
if (end1.hasErrors()) {
    println end1.errors
}

def end2 = new Endereco(logradouro: 'R. Treze de Maio', numero: 1930,
    bairro: 'Centro', CEP: '13560-647', cidade: sanca)
end2.save()
if (end2.hasErrors()) {
    println end2.errors
}

def end3 = new Endereco(logradouro: 'R. Nilton Coelho de Andrade',
    numero: 772, bairro: 'Vila Maria', CEP: '03092-324', cidade: sampa)
end3.save()
if (end3.hasErrors()) {
    println end3.errors
}

def end4 = new Endereco(logradouro: 'R. Humberto Manelli', numero: 50,
    complemento: 'Apto 31', bairro: 'Jardim Gibertoni',
    CEP: '13562-420', cidade: sanca)
end4.save()
if (end4.hasErrors()) {
    println end4.errors
}

println 'populando endereços - ok'

def bb = new Banco(numero: 1, nome: 'Banco do Brasil',
    CNPJ: '00.000.000/0001-91')

bb.save()
if (bb.hasErrors()) {
    println bb.errors
}

def santander = new Banco(numero: 33, nome: 'Santander',
    CNPJ: '90.400.888/0001-42')

santander.save()
if (santander.hasErrors()) {
    println santander.errors
}

println 'populando bancos - ok'

def agencial = new Agencia(numero: 1888, nome: 'Conde do Pinhal',
    endereco: end1, banco: bb)

agencial.save()
if (agencial.hasErrors()) {
    println agencial.errors
}

def agencia2 = new Agencia(numero: 24, nome: 'Treze de Maio',
    endereco: end2, banco: santander)

agencia2.save()
if (agencia2.hasErrors()) {
    println agencia2.errors
}

println 'populando agências - ok'
```

Código 2.22: **BootStrap.groovy (2)**

O trecho apresentado no Código 2.23, popula instâncias das classes **Gerente**, **CaixaEletronico**, **ClienteFisico** e **ClienteJuridico**.

```
def gerente1 = new Gerente(nome: 'Carlos da Silva', rg: '1234 SSP/SP',
    CPF: '129.304.458-07', agencia: agencia1
)

gerente1.save()
if (gerente1.hasErrors()) {
    println gerente1.errors
}

def gerente2 = new Gerente(nome: 'Maria José', rg: '3467 SSP/RJ',
    CPF: '018.990.444-50', agencia: agencia2
)

gerente2.save()
if (gerente2.hasErrors()) {
    println gerente2.errors
}

println 'populando gerentes - ok'

def caixa1 = new CaixaEletronico(banco: bb, endereco: end1)

caixa1.save()
if (caixa1.hasErrors()) {
    println agencia1.errors
}

def caixa2 = new CaixaEletronico(banco: santander, endereco: end2)

caixa2.save()
if (caixa2.hasErrors()) {
    println agencia1.errors
}

println 'populando caixas eletrônicos - ok'

def cliFisico = new ClienteFisico(nome: 'Fulano de Tal',
    rg: '13567 SSP/SP', CPF: '018.990.444-50', endereco: end4,
    dtMoradia: new Date(), status: Cliente.ATIVO)

cliFisico.save()
if (cliFisico.hasErrors()) {
    println cliFisico.errors
}

println 'populando clientes físicos - ok'

def cliJuridico = new ClienteJuridico(nome: 'Viação Cometa S/A',
    CNPJ: '61.084.018/0001-03', endereco: end3,
    dtMoradia: new Date(), status: Cliente.ATIVO)

cliJuridico.save()
if (cliJuridico.hasErrors()) {
    println cliJuridico.errors
}

println 'populando clientes jurídicos - ok'
```

Código 2.23: **BootStrap.groovy (3)**

O trecho apresentado no Código 2.24, popula instâncias das classes **ContaCorrente**, **ContaPoupanca** e **Transacao**. Conforme pode-se observar, as instâncias da classe **Transacao** criadas representam depósitos e saques.

```
def corrente = new ContaCorrente(agencia: agencial,
    numero: '010414688', saldo: 1000.56d, limite: 500.00d,
    abertura: new Date()
)

corrente.save()
if (corrente.hasErrors()) {
    println corrente.errors
}

def contaCli1 = new ContaCliente(conta: corrente,
    cliente: cliJuridico, titular: true
)

contaCli1.save()
if (contaCli1.hasErrors()) {
    println contaCli1.errors
}

println 'populando contas correntes (associado ao cliente jurídico) - ok'

def poupanca = new ContaPoupanca(agencia: agencia2,
    numero: '261327', saldo: 10000.56d, juros: 0.50d,
    correcao: 1.20d, dia: 23, abertura: new Date()
)

poupanca.save()
if (poupanca.hasErrors()) {
    println poupanca.errors
}

def contaCli2 = new ContaCliente(conta: poupanca,
    cliente: cliFisico, titular: true
)

contaCli2.save()
if (contaCli2.hasErrors()) {
    println contaCli2.errors
}

println 'populando contas poupanças (associado ao cliente físico) - ok'

def deposito = new Transacao(contaCliente: contaCli2, caixaEletronico: caixa2,
    valor: 50d, data: new Date(), quem: 'Próprio', motivo: 'Depósito',
    tipo: Transacao.CRÉDITO
)

deposito.save()
if (deposito.hasErrors()) {
    println deposito.errors
}

println 'populando depósitos - ok'

def saque = new Transacao(contaCliente: contaCli1, caixaEletronico: caixa1,
    valor: 100d, data: new Date(), quem: 'Próprio', motivo: 'Saque',
    tipo: Transacao.DÉBITO
)

saque.save()
if (saque.hasErrors()) {
    println saque.errors
}

println 'populando saques - ok'
```

Código 2.24: **BootStrap.groovy** (4)

E por fim, o trecho apresentado no Código 2.25, popula instâncias da classe **Transacao**. Conforme pode-se observar, as instâncias da classe **Transacao** criadas representam transferências bancárias.

```
def transf1 = new Transacao(contaCliente: contaCli1,
    caixaEletronico: caixa2, valor: 25d, data: new Date(),
    quem: 'Próprio', motivo: 'Transferência', tipo: Transacao.DÉBITO)

def transf2 = new Transacao(contaCliente: contaCli2,
    caixaEletronico: caixa2, valor: 25d, data: new Date(),
    quem: 'Fulano de Tal', motivo: 'Transferência',
    tipo: Transacao.CRÉDITO)

transf1.save()
if (transf1.hasErrors()) {
    println transf1.errors
}

transf2.save()
if (transf2.hasErrors()) {
    println transf2.errors
}

println 'populando transferências - ok'
}

def destroy = {
}
}
```

Código 2.25: **BootStrap.groovy** (5)

Para executar a aplicação, clique no botão direito do mouse no ícone **Run: Grails** (Figura 2.9). A aplicação é implantada no servidor *Web*, como pode ser observado na janela **Run** do IDE IntelliJ.

- A aplicação pode ser acessada através da URL `http://localhost:8080`. Se o navegador não abrir automaticamente, cole a URL em um navegador e a aplicação será acessada. Os controladores da aplicação serão listados (Figura 2.9).

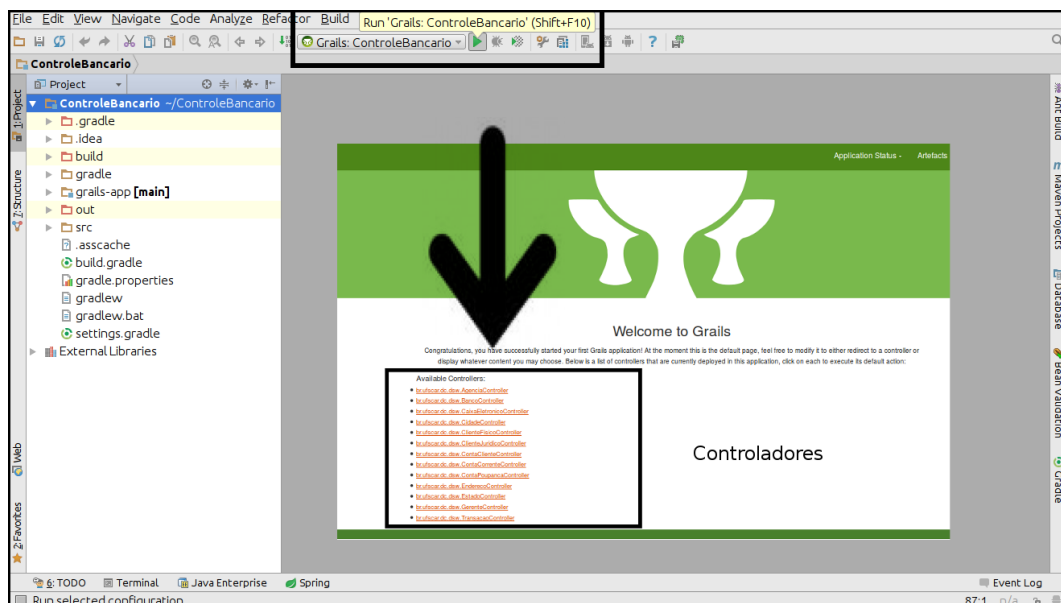


Figura 2.9: Execução da aplicação **ControleBancario**

- Ao clicar no link **br.ufscar.dc.dsw.TransacaoController**, as transações bancárias inseridas anteriormente (**BootStrap.groovy**) são apresentadas (Figura 2.10).

Conta Cliente	Caixa Eletronico	Valor	Data	Quem	Motivo	Tipo
018.990.444-50 X (Poupanca) 261327	Santander - R. Treze de Maio, 1930, Centro 13560-647 São Carlos - SP	50.0	09/04/2016 15:26:31 BRT	Próprio	Depósito	CRÉDITO
61.084.018.0001-03 X (Conta Corrente) 010414688	Banco do Brasil - R. Conde do Pinhal, 1909, Centro 13560-648 São Carlos - SP	100.0	09/04/2016 15:26:31 BRT	Próprio	Saque	DÉBITO
61.084.018.0001-03 X (Conta Corrente) 010414688	Santander - R. Treze de Maio, 1930, Centro 13560-647 São Carlos - SP	25.0	09/04/2016 15:26:31 BRT	Próprio	Transferência	DÉBITO
018.990.444-50 X (Poupanca) 261327	Santander - R. Treze de Maio, 1930, Centro 13560-647 São Carlos - SP	25.0	09/04/2016 15:26:31 BRT	Fulano de Tal	Transferência	CRÉDITO

Figura 2.10: Lista de transações bancárias

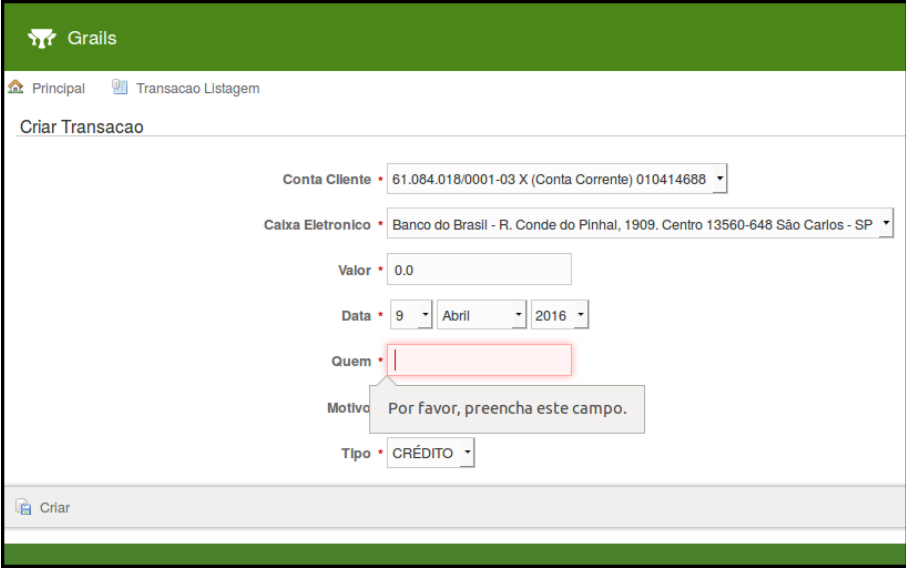
- Clique em **Novo Transacao** e crie uma nova transação bancária. Quando clicar em **Criar**, observe que você poderá **Editar** ou **Remover** a transação. Por fim, ao clicar em **Transacao Listagem**, a nova entrada é apresentada na lista de transações (Figura 2.11).

The diagram illustrates the process of creating a new transaction. It begins with the 'Criar Transacao' form, which includes fields for 'Conta Cliente', 'Caixa Eletronico', 'Valor', 'Data', 'Quem', 'Motivo', and 'Tipo'. An arrow points from this form to the 'Ver Transacao' confirmation screen, which displays the entered details. Another arrow points from the confirmation screen to the 'Transacao Listagem' table, where the new transaction is now listed alongside the previous ones.

Figura 2.11: Criação de uma nova transação bancária

Salienta-se que para a criação das outras entidades da aplicação (**Cliente**, **Conta** e outros), os passos são análogos.

- Clique em **Novo Transacao** e crie uma nova transação bancária com valores inválidos (Figura 2.12). Verifica-se que a transação não foi criada, pois as validações de dados foram violadas.



A interface Grails para criar uma transação bancária. O formulário "Criar Transacao" contém os seguintes campos:

- Conta Cliente: 61.084.018/0001-03 X (Conta Corrente) 010414688
- Caixa Eletronico: Banco do Brasil - R. Conde do Pinhal, 1909. Centro 13560-648 São Carlos - SP
- Valor: 0.0
- Data: 9 Abril 2016
- Quem: (campo em vermelho com mensagem de erro: "Por favor, preencha este campo.")
- Motivo: (campo vazio)
- Tipo: CRÉDITO

Um botão "Criar" está visível na base do formulário.

Figura 2.12: Criação de uma transação bancária com valores inválidos

- ! Sugere-se, como aprendizado, executar os demais controladores e verificar se as demais funcionalidades da aplicação encontram-se funcionando corretamente.

2.5 Considerações finais

Esse capítulo apresentou uma implementação parcial da aplicação **ControleBancario**. O código-fonte dessa aplicação (*ControleBancarioV1*) encontra-se disponível em um repositório *GitHub*¹⁵.

Dando continuidade ao desenvolvimento em Grails, o próximo capítulo apresenta a implementação de novas funcionalidades no contexto da aplicação **ControleBancario**.

¹⁵URL: <https://github.com/delanobeder/FG>



3 — Controle Bancário: Versão 2

Neste capítulo, dando continuidade ao desenvolvimento em Grails, será apresentado o processo de desenvolvimento da segunda versão da aplicação **ControleBancario**. Nessa versão são incorporadas as seguintes funcionalidades:

- Controle de acesso: autenticação e autorização de usuários;
- Internacionalização. Ou seja, personalizar o conteúdo apresentado nas visões (*.gsp) com base no *Locale* (idioma e região) dos usuários;
- Personalização dos *templates* utilizados pelo mecanismo de *scaffolding* na geração dos controladores e visões;
- Definição de máscaras de entrada para os atributos CEP, CNPJ, e CPF das classes de domínio; e
- Implementação de uma biblioteca de marca¹⁶ que apresenta informações relacionadas ao usuário logado.

3.1 Configuração da aplicação

(a) **Instalação de plugins.** Na implementação das funcionalidades da aplicação **ControleBancario**, discutidas nesse capítulo, será utilizado o plugin Grails **spring-security** que auxilia a autenticação dos usuários da aplicação. Conforme discutido anteriormente, para instalar o plugin **spring-security** adicione uma linha, descrevendo a dependência, no arquivo **build.gradle** conforme apresentado na linha 50 do Código 3.1.

¹⁶Em inglês: *tag library*.

(b) Instalação de bibliotecas Javascript. Na implementação das funcionalidades discutidas nesse capítulo, será utilizada a biblioteca: **jquery.maskedinput.min.js**¹⁷. Dessa forma, é necessário fazer o *download* desse arquivo e copiá-lo no diretório **grails-app/assets/javascripts**.

(c) Atualização das dependências. Por fim, é necessário atualizar as dependências (*plugins*) da aplicação **ControleBancario**. Para tal, no IDE IntelliJ: abra a janela **Terminal** e execute o comando **grails compile**. Esse comando compila o projeto atualizando as dependências e instala os *plugins*, caso necessário.

```

1  buildscript {
2      ext {
3          grailsVersion = project.grailsVersion
4      }
5      repositories {
6          mavenLocal()
7          maven { url "https://repo.grails.org/grails/core" }
8      }
9      dependencies {
10         classpath "org.grails:grails-gradle-plugin:$grailsVersion"
11         classpath "com.bertramlabs.plugins:asset-pipeline-gradle:2.5.0"
12         classpath "org.grails.plugins:hibernate4:5.0.2"
13     }
14 }
15 version "0.1"
16 group "controlebancario"
17 apply plugin: "eclipse"
18 apply plugin: "idea"
19 apply plugin: "war"
20 apply plugin: "org.grails.grails-web"
21 apply plugin: "org.grails.grails-gsp"
22 apply plugin: "asset-pipeline"
23 ext {
24     grailsVersion = project.grailsVersion
25     gradleWrapperVersion = project.gradleWrapperVersion
26 }
27 repositories {
28     mavenLocal()
29     maven { url "https://repo.grails.org/grails/core" }
30 }
31 dependencyManagement {
32     imports {
33         mavenBom "org.grails:grails-bom:$grailsVersion"
34     }
35     applyMavenExclusions false
36 }
37 dependencies {
38     compile "org.springframework.boot:spring-boot-starter-logging"
39     compile "org.springframework.boot:spring-boot-autoconfigure"
40     compile "org.grails:grails-core"
41     compile "org.springframework.boot:spring-boot-starter-actuator"
42     compile "org.springframework.boot:spring-boot-starter-tomcat"
43     compile "org.grails:grails-dependencies"
44     compile "org.grails:grails-web-boot"
45     compile "org.grails.plugins:cache"
46     compile "org.grails.plugins:scaffolding"
47     compile "org.grails.plugins:hibernate4"
48     compile "org.hibernate:hibernate-ehcache"
49     compile "org.grails.plugins:br-validation:0.3"
50     compile "org.grails.plugins:spring-security-core:3.0.4"
51     console "org.grails:grails-console"
52     profile "org.grails.profiles:web:3.1.4"
53     runtime "org.grails.plugins:asset-pipeline"
54     runtime "com.h2database:h2"
55     runtime "org.postgresql:postgresql:9.3-1101-jdbc41"
56     testCompile "org.grails:grails-plugin-testing"
57     testCompile "org.grails.plugins:geb"
58     testRuntime "org.seleniumhq.selenium:selenium-htmlunit-driver:2.47.1"
59     testRuntime "net.sourceforge.htmlunit:htmlunit:2.18"
60 }
61 task wrapper(type: Wrapper) {
62     gradleVersion = gradleWrapperVersion
63 }
64 assets {
65     minifyJs = true
66     minifyCss = true
67 }

```

Código 3.1: BuildConfig.groovy

¹⁷<http://digitalbush.com/projects/masked-input-plugin>

3.2 Controle de Acesso

Conforme dito, a segunda versão da aplicação **ControleBancario** utiliza o *plugin* **spring-security-core**¹⁸ que simplifica a integração do Spring Security¹⁹ em aplicações Grails.

Esse *plugin* define uma série de comandos. Entre esses podemos destacar o comando **s2-quickstart** que cria tanto as classes de domínio básicas tanto os controladores (e suas respectivas visões) necessários para lidar com a autenticação de usuários.

Portanto, o primeiro passo na implementação da funcionalidade de controle de acesso é a execução desse comando. Para tal, no IDE IntelliJ: abra a janela **Terminal** e execute o comando **grails s2-quickstart br.ufscar.dc.dsw Usuario Papel**. Esse comando cria os seguintes artefatos:

- **br.ufscar.dc.dsw.Usuario** – classe de domínio que representa os usuários autenticados.
- **br.ufscar.dc.dsw.Papel** – classe de domínio que representa os papéis que os usuários podem desempenhar. Cada papel possui permissões a ele associadas.
- **br.ufscar.dc.dsw.UsuarioPapel** – classe de domínio que representa o relacionamento muitos-para-muitos entre usuários e papéis. Ou seja, um usuário pode desempenhar vários papéis e um papel pode ser desempenhado por vários usuários.
- **LoginController** e **LogoutController** (e suas respectivas visões) que são responsáveis pelas operações de *login* e *logout* da aplicação.

A seguir, adicione o seguinte trecho na classe de domínio **Usuario** – pai da hierarquia de usuários da aplicação **ControleBancario**. Ou seja, a estratégia de mapeamento *table-per-hierarchy* (Seção 2.2.12) será desabilitada e, para essa hierarquia de classes, a estratégia *table-per-class* será utilizada. Assim, é criada uma tabela para a classe **Usuario** assim como para cada classe filha discutida nas próximas seções.

```
static mapping = {
    password column: 'password'
    tablePerHierarchy false
}
```

Por fim, adicione o seguinte trecho no arquivo **conf/application.groovy**. Esse comando habilita que os comandos HTTP POST e GET sejam utilizados para invocar o controlador **LogoutController** que é responsável pela operação de *logout* da aplicação. Por *default*, apenas o comando POST pode ser utilizado para invocar o controlador **LogoutController**.

```
grails.plugin.springsecurity.logout.postOnly = false
```

¹⁸<http://grails.org/plugin/spring-security-core>

¹⁹<http://static.springsource.org/spring-security/site/index.html>

3.2.1 Classes de Domínio: Cliente, ClienteFisico, ClienteJuridico e Gerente

Os clientes e os gerentes serão usuários da aplicação **ControleBancario**. Ou seja, as classes de domínio **Cliente** e **Gerente** são subclasses da classe **Usuario** definida na seção anterior. Desde que as classes **ClienteFisico** e **ClienteJuridico** são subclasses de **Cliente**, estas também serão subclasses de **Usuario**.

A classe **Usuario** define dois atributos **username** e **password** que são responsáveis pelo armazenamento do *login* e senha dos usuários da aplicação. Código 3.2 mostra as alterações na implementação das classes **Cliente**, **ClienteFisico**, **ClienteJuridico** e **Gerente**. Pode-se observar que na implementação dessas classes foram incluídas restrições relacionadas aos atributos **username** e **password** definidos pela classe pai **Usuario**.

```
abstract class Cliente extends Usuario {  
    static constraints = {  
        username (blank: false, unique: true)  
        password (password: true, blank: false)  
        // demais restrições  
    }  
    // atributos e métodos da classe  
}  
  
class ClienteFisico extends Cliente {  
    static constraints = {  
        username (blank: false, unique: true)  
        password (password: true, blank: false)  
        // demais restrições  
    }  
    // atributos e métodos da classe  
}  
  
class ClienteJuridico extends Cliente {  
    static constraints = {  
        username (blank: false, unique: true)  
        password (password: true, blank: false)  
        // demais restrições  
    }  
    // atributos e métodos da classe  
}  
  
class Gerente extends Usuario {  
    static constraints = {  
        username (blank: false, unique: true)  
        password (password: true, blank: false)  
        // demais restrições  
    }  
    // atributos e métodos da classe  
}
```

Código 3.2: Usuários: Clientes e Gerentes

3.3 Internacionalização

Grails apoia a internacionalização (i18n). Ou seja, com o Grails é possível personalizar o conteúdo que aparece em qualquer visão (*.gsp) com base no *Locale* dos usuários.

Para tirar proveito do suporte a internacionalização em Grails, a equipe de desenvolvimento tem que criar *message bundles* – uma para cada idioma que a equipe deseja internacionalizar. *Message bundles* em Grails estão localizados dentro do diretório **i18n** e são simples arquivos de propriedades Java/Groovy. Por padrão, Grails procura em *messages.properties* para mensagens, a menos que o usuário tenha especificado um *Locale* em específico. A equipe de desenvolvimento pode criar novas *message bundles* simplesmente criando novos arquivos de propriedades que terminam com a localidade que está interessada. Por exemplo, *messages_pt_BR.properties* para o Português do Brasil (Figura 3.1).

Um objeto *Locale* representa uma região geográfica específica, político ou cultural. Uma operação que requer uma localidade para executar a sua tarefa é denominada de sensível e usa o objeto *Locale* para prover a informação mais apropriada aos usuários. Por exemplo, exibir o preço de uma mercadoria é uma operação sensível a localidade – o número deve ser formatado de acordo com os costumes e convenções do país de origem do usuário, região ou cultura.

O objeto *Locale* é composto por: (i) um código do idioma ou (ii) um código do idioma e um código de país. Por exemplo, **en** é o código para o idioma Inglês (não importando o país ou região geográfica) enquanto **pt_BR** e **pt_PT** são dois *Locales* que compartilham o mesmo idioma: o primeiro é o código para o Português do Brasil e o segundo é o código para o Português usado em Portugal (Figura 3.1).

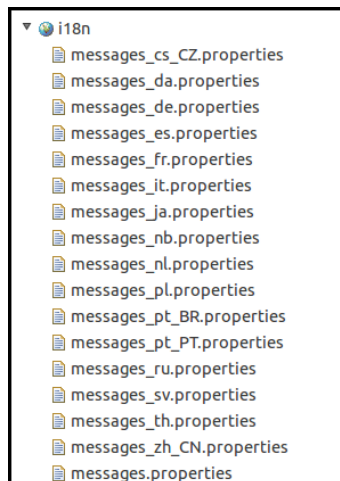


Figura 3.1: I18n (arquivos de propriedades).

As visões geradas no *scaffolding* de nossa aplicação estão parcialmente internacionalizados. Ou seja, pode-se observar que essas visões incluem a *tag* **<g:message code>**, onde *code* é uma referência a algum termo a ser internacionalizado. O nó **i18n** contém um conjunto de arquivos de propriedades (termos a serem traduzidos para diferentes línguas) – Figura 3.1. É importante salientar que esses arquivos são gerados automaticamente durante a execução dos comandos do Grails (**create-app**, **create-controller**, **generate-all**, etc).

As visões são os locais mais comuns para o uso de mensagens internacionalizadas. Para acessar as mensagens personalizadas nas visões, a equipe de desenvolvimento basta utilizar a seguinte *tag* nas visões (*.gsp) desenvolvidas:

```
<g:message code="welcome.greeting" />
```

Se a equipe tiver incluído uma chave no arquivo `messages.properties` (com sufixo do *Locale* apropriado), tal como ilustrada a seguir, então Grails apresenta a mensagem personalizada:

```
welcome.greeting = Good Morning. My name is Bob.
```

Note que em algumas ocasiões é necessário passar argumentos para a mensagem. Isso também é possível com a mesma *tag*:

```
<g:message code="welcome.greeting" args="${ ['Night','Bill'] }" />
```

No entanto, é necessário utilizar os parâmetros de posicionamento na mensagem.

```
welcome.greeting = Good {0}. My name is {1}.
```

Note que o esforço da equipe Web para internacionalizar sua aplicação consiste em determinar os termos a serem traduzidos, que serão inseridos em um arquivo de propriedades, e depois realizar

a tradução para as diferentes línguas. As traduções serão armazenadas em arquivos cujos nomes terminam com o *Locale* (língua e região) desejado.

Figura 3.2 apresenta algumas mensagens relacionadas às classes de domínio **Agencia** que foram traduzidas e copiadas para o arquivo `messages_pt_BR.properties` (*Message Bundle* para o Português do Brasil). Pode-se observar que esse arquivo contém traduções para o nome da classe assim como para o nome de cada um de seus atributos. Além disso, foram adicionadas algumas mensagens relacionadas à página de *login* que foi gerada automaticamente pela execução do comando **s2-quickstart**.

```
# Agencia - mensagens
agencia.label = Agência
agencia.numero.label = Número
agencia.nome.label = Nome
agencia.endereco.label = Endereço
agencia.banco.label = Banco
agencia.gerentes.label = Gerentes

# Página de Login - mensagens
springSecurity.login.header = Login
springSecurity.login.remember.me.label = Lembre-me
springSecurity.login.button = Login
springSecurity.login.username.label = Login
springSecurity.login.password.label = Password
```

Figura 3.2: Mensagens I18n para a classe de Domínio Usuário



Um bom exercício consiste em internacionalizar as mensagens relacionadas às demais classes de domínio da aplicação **ControleBancario**.

3.4 Personalização dos templates utilizados no scaffolding

Nessa seção será apresentado o processo de personalização dos *templates* utilizados pelo mecanismo de *scaffolding* na geração dos controladores e visões. No contexto da aplicação **ControleBancario**, essas personalizações tem como objetivo gerar os controladores e visões com funcionalidades relacionadas ao controle de acesso já incorporadas. Além disso, a personalização é realizada de tal forma que as visões **create.gsp** e **edit.gsp**, geradas pelo *scaffolding*, já incorporam as máscaras de entrada para os atributos CEP, CNPJ e CPF.

Portanto, o primeiro passo no processo de personalização dos templates consiste na execução do comando **install-templates**. Para tal, no IDE IntelliJ: abra a janela **Terminal** e execute o comando **grails install-templates**. Esse comando copia os *templates* usadas nas atividades de geração de código para o diretório **src/main/templates/scaffolding**. Esse diretório contém:

- Os *templates* utilizados pelos comandos `create-*` (**create-domain-class**, **create-controller**, etc);
- Os *templates* utilizados pelos comandos `generate-*` (**generate-all**, **generate-controller**, **generate-views**, etc). No contexto desse tutorial, apenas serão personalizados os *templates* dessa categoria;

3.4.1 Template: Controller.groovy

O *plugin* **spring-security** permite a utilização da anotação **@Secured** para aplicar regras de controle de acesso aos controladores (e suas respectivas ações). A anotação pode ser definida a nível de uma ação específica, que significa que os papéis especificados são necessários no acesso à aquela ação ou a nível de classe, que significa que os papéis especificados são necessários no acesso a todas as ações do controlador.

O *template* **Controller.groovy** é utilizado na geração dos controladores. No contexto da aplicação **ControleBancario**, esse *template* será alterado de tal forma que o acesso a ação **show()**, dos controladores da aplicação, será restrito aos usuários que desempenham os respectivos papéis: **ROLE_ADMIN**, **ROLE_CLIENTE** e **ROLE_GERENTE** (Código 3.3, linha 14). As demais ações dos controladores serão restritas ao papel **ROLE_ADMIN** (Código 3.3, linha 7). Salienta-se que esses papéis serão criados no *Bootstrap* da aplicação (Seção 3.7).

3.4.2 Template: create.gsp

O *template* **create.gsp** é utilizado na geração das visões **create()** associadas a cada um dos controladores da aplicação. No contexto da aplicação **ControleBancario**, esse *template* será alterado de tal forma que serão definidas máscaras de entrada (Código 3.4, linhas 7-16) para os atributos CEP, CNPJ e CPF das classes de domínio.

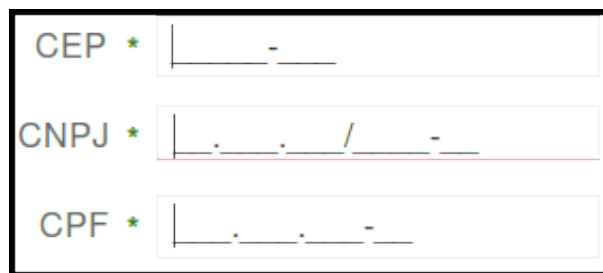
A imagem mostra três campos de entrada de texto empilhados verticalmente. O primeiro campo é rotulado 'CEP *' e contém a máscara '____-____'. O segundo campo é rotulado 'CNPJ *' e contém a máscara '____.____.____/____-____'. O terceiro campo é rotulado 'CPF *' e contém a máscara '____.____.____-____'. Os campos são de cor cinza clara com bordas arredondadas.

Figura 3.3: Máscaras de entrada: CEP, CNPJ e CPF

Figura 3.3 apresenta um exemplo das máscaras de entrada dos atributos CEP, CNPJ e CPF da aplicação **ControleBancario**. Na definição dessas máscaras de entrada, foi utilizado o *plugin* **jQuery Masked Input**²⁰ que permite construir máscaras em campos HTML com as seguintes regras:

- **a** - Representa um caractere alfabético (A-Z, a-z)
- **9** - Representa um dígito (0-9)
- ***** - Representa um caractere alfanumérico (A-Z, a-z ,0-9)

Assim, a máscara **999.999.999-99** define um CPF composto por 11 dígitos separados pelos caracteres: ponto (.) e traço (-).

! Análogo ao **create.gsp**, o *template* **edit.gsp** também necessita ser alterado para definir as máscaras de entrada para os atributos CEP, CNPJ e CPF das classes de domínio. Então, fica como exercício para o leitor realizar tal alteração.

²⁰<http://digitalbush.com/projects/masked-input-plugin/>

```

1  <%=packageName ? "package ${packageName}" : ''%>
2
3  import static org.springframework.http.HttpStatus.*
4  import grails.transaction.Transactional
5  import org.springframework.security.access.annotation.Secured
6  @Transactional(readOnly = true)
7  @Secured ('ROLE_ADMIN')
8  class ${className} Controller {
9      static allowedMethods = [save: "POST", update: "PUT", delete: "DELETE"]
10     def index(Integer max) {
11         params.max = Math.min(max ?: 10, 100)
12         respond ${className}.list(params), model:[${propertyName}Count: ${className}.count()]
13     }
14     @Secured ([ 'ROLE_ADMIN', 'ROLE_CLIENTE', 'ROLE_GERENTE' ])
15     def show(${className} ${propertyName}) {
16         respond ${propertyName}
17     }
18     def create() {
19         respond new ${className}(params)
20     }
21     @Transactional
22     def save(${className} ${propertyName}) {
23         if (${propertyName} == null) {
24             transactionStatus.setRollbackOnly()
25             notFound()
26             return
27         }
28         if (${propertyName}.hasErrors()) {
29             transactionStatus.setRollbackOnly()
30             respond ${propertyName}.errors, view: 'create'
31             return
32         }
33         ${propertyName}.save flush:true
34         request.withFormat {
35             form multipartForm {
36                 flash.message = message(code: 'default.created.message', args: [message(code: '${propertyName}.label',
37                                     default: '${className}')], ${propertyName}.id))
38                 redirect ${propertyName}
39             }
40             '*' { respond ${propertyName}, [status: CREATED] }
41         }
42     }
43     def edit(${className} ${propertyName}) {
44         respond ${propertyName}
45     }
46     @Transactional
47     def update(${className} ${propertyName}) {
48         if (${propertyName} == null) {
49             transactionStatus.setRollbackOnly()
50             notFound()
51             return
52         }
53         if (${propertyName}.hasErrors()) {
54             transactionStatus.setRollbackOnly()
55             respond ${propertyName}.errors, view: 'edit'
56             return
57         }
58         ${propertyName}.save flush:true
59         request.withFormat {
60             form multipartForm {
61                 flash.message = message(code: 'default.updated.message', args: [message(code: '${propertyName}.label',
62                                     default: '${className}')], ${propertyName}.id))
63                 redirect ${propertyName}
64             }
65             '*' { respond ${propertyName}, [status: OK] }
66         }
67     }
68     @Transactional
69     def delete(${className} ${propertyName}) {
70         if (${propertyName} == null) {
71             transactionStatus.setRollbackOnly()
72             notFound()
73             return
74         }
75         ${propertyName}.delete flush:true
76         request.withFormat {
77             form multipartForm {
78                 flash.message = message(code: 'default.deleted.message', args: [message(code: '${propertyName}.label',
79                                     default: '${className}')], ${propertyName}.id))
80                 redirect action: "index", method: "GET"
81             }
82             '*' { render status: NO_CONTENT }
83         }
84     }
85     protected void notFound() {
86         request.withFormat {
87             form multipartForm {
88                 flash.message = message(code: 'default.not.found.message', args: [message(code: '${propertyName}.label',
89                                     default: '${className}'), params.id])
90                 redirect action: "index", method: "GET"
91             }
92             '*' { render status: NOT_FOUND }
93         }
94     }
95 }

```

Código 3.3: *Template Controller.groovy*

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta name="layout" content="main" />
5     <g:set var="entityName" value="\${message(code: '${propertyName}.label', default: '${className}')}"/>
6     <title><g:message code="default.create.label" args="[entityName]" /></title>
7     <asset:javascript src="jquery-2.2.0.min.js" />
8     <asset:javascript src="jquery.maskedinput.min.js" />
9     <g:javascript>
10      var JQuery = jQuery.noConflict()
11      JQuery(document).ready(function(){
12        JQuery("#CPF").mask("999.999.999-99");
13        JQuery("#CNPJ").mask("99.999.999/9999-99");
14        JQuery("#CEP").mask("99999-999");
15      });
16    </g:javascript>
17  </head>
18  <body>
19    <a href="#create-${propertyName}" class="skip" tabindex="-1"><g:message code="default.link.skip.label"
20      default="Skip to content&hellip;"/></a>
21
22    <div class="nav" role="navigation">
23      <ul>
24        <li><a class="home" href="\${createLink(uri: '/')}"><g:message code="default.home.label"/></a></li>
25        <li><g:link class="list" action="index">
26          <g:message code="default.list.label" args="[entityName]" />
27          </g:link></li>
28        </ul>
29      </div>
30      <div id="create-${propertyName}" class="content scaffold-create" role="main">
31        <h1><g:message code="default.create.label" args="[entityName]" /></h1>
32        <g:if test="\${flash.message}">
33          <div class="message" role="status">\${flash.message}</div>
34        </g:if>
35        <g:hasErrors bean="\${this.${propertyName}}">
36          <ul class="errors" role="alert">
37            <g:eachError bean="\${this.${propertyName}}" var="error">
38              <li><g:if test="\${error in org.springframework.validation.FieldError}">
39                data-field-id="\${error.field}" </g:if><g:message error="\${error}"/></li>
40            </g:eachError>
41          </ul>
42        </g:hasErrors>
43        <g:form action="save">
44          <fieldset class="form">
45            <f:all bean="\${propertyName}"/>
46          </fieldset>
47          <fieldset class="buttons">
48            <g:submitButton name="create" class="save"
49              value="\${message(code: 'default.button.create.label', default: 'Create')}"/>
50          </fieldset>
51        </g:form>
52      </div>
53    </body>
54  </html>

```

Código 3.4: *Template create.gsp*

3.4.3 Template: index.gsp

O *template* **index.gsp** é utilizado na geração das visões **index** associadas a cada um dos controladores da aplicação. No contexto da aplicação **ControleBancario**, esse *template* será alterado (Código 3.5, linhas 14-16) de tal forma que o *link* para a operação de criação de entidades (ação **create()**) apenas será apresentada se o usuário encontra-se autenticado e possui papel necessário para executar essa ação.

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta name="layout" content="main" />
5     <g:set var="entityName" value="\${message(code: '${propertyName}.label', default: '${className}')}"/>
6     <title><g:message code="default.list.label" args="[entityName]" /></title>
7   </head>
8   <body>
9     <a href="#list-${propertyName}" class="skip" tabindex="-1"><g:message code="default.link.skip.label"
10                                     default="Skip to content&hellip;"/></a>
11
12     <div class="nav" role="navigation">
13       <ul>
14         <li><a class="home" href="\${createLink(uri: '/')}"><g:message code="default.home.label"/></a></li>
15         <sec:access controller="${propertyName}" action='create'>
16           <li><g:link class="create" action="create"><g:message code="default.new.label"
17                                     args="[entityName]" /></g:link></li>
18         </sec:access>
19       </ul>
20     </div>
21     <div id="list-${propertyName}" class="content scaffold-list" role="main">
22       <h1><g:message code="default.list.label" args="[entityName]" /></h1>
23       <g:if test="\${flash.message}">
24         <div class="message" role="status">\${flash.message}</div>
25       </g:if>
26       <f:table collection="\${${propertyName}List}" />
27
28       <div class="pagination">
29         <g:paginate total="\${${propertyName}Count ?: 0}" />
30       </div>
31     </div>
32 </body>
</html>

```

Código 3.5: *Template* **index.gsp**

O *plugin* **spring-security** define algumas *tags* GSP que permite a exibição condicional de *links* de acesso a ações de controladores. Ou seja, aquele *link* apenas será apresentado em uma visão se o usuário encontra-se autenticado e possui papel necessário para executar a ação.

Por exemplo, a *tag* GSP **<sec: ifLoggedIn>** apresentada abaixo, apenas renderizaria a mensagem *Bemvindo!*, se o usuário encontra-se autenticado.

```

<sec:ifLoggedIn>
  Bemvindo!
</sec:ifLoggedIn>

```

Como outro exemplo, a *tag* GSP **<sec: access>** abaixo, apenas renderizaria o *link* para a ação **create** do controlador **transacao**, se o usuário encontra-se autenticado e possui papel necessário para executar a ação.

```

<sec:access controller='transacao' action='create'>
  <g:link controller='transacao' action='create'>Cria Transação</g:link>
</sec:access>

```

Para maiores informações sobre as *tags* GSP, definidas pelo *plugin* **spring-security**, o leitor pode consultar o seguinte endereço: <http://grails.org/plugin/spring-security-core>.

3.4.4 Template: show.gsp

O *template* **show.gsp** é utilizado na geração das visões **show** associadas a cada um dos controladores da aplicação. No contexto da aplicação **ControleBancario**, esse *template* será alterado para refletir as seguintes funcionalidades:

- O **link** para a operação de criação de entidades (ação **create**) apenas será apresentada se o usuário encontra-se autenticado e possui papel necessário para executar essa ação (Código 3.6, linhas 16-19).
- O botão **edit**, responsável pela edição de entidades, apenas será apresentada se o usuário encontra-se autenticado e possui papel necessário para executar essa ação (Código 3.6, linhas 30-33).
- O botão **delete**, responsável pela remoção de entidades, apenas será apresentada se o usuário encontra-se autenticado e possui papel necessário para executar essa ação (Código 3.6, linhas 34-38).

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta name="layout" content="main" />
5     <g:set var="entityName" value="\${message(code: '${propertyName}.label', default: '${className}')}" />
6     <title><g:message code="default.show.label" args="[entityName]" /></title>
7   </head>
8   <body>
9     <a href="#show-${propertyName}" class="skip" tabindex="-1"><g:message code="default.link.skip.label"
10       default="Skip to content&hellip;" /></a>
11     <div class="nav" role="navigation">
12       <ul>
13         <li><a class="home" href="\${createLink(uri: '/')}"><g:message code="default.home.label" /></a></li>
14         <li><g:link class="list" action="index"><g:message code="default.list.label"
15           args="[entityName]" /></g:link></li>
16         <sec:access controller="${propertyName}" action="create">
17           <li><g:link class="create" action="create"><g:message code="default.new.label"
18             args="[entityName]" /></g:link></li>
19         </sec:access>
20       </ul>
21     </div>
22     <div id="show-${propertyName}" class="content scaffold-show" role="main">
23       <h1><g:message code="default.show.label" args="[entityName]" /></h1>
24       <g:if test="\${flash.message}">
25         <div class="message" role="status">\${flash.message}</div>
26       </g:if>
27       <f:display bean="\${propertyName}" />
28       <g:form resource="\${this.${propertyName}}" method="DELETE">
29         <fieldset class="buttons">
30           <sec:access controller="${propertyName}" action='edit'>
31             <g:link class="edit" action="edit" resource="\${this.${propertyName}}">
32               <g:message code="default.button.edit.label" default="Edit" /></g:link>
33           </sec:access>
34           <sec:access controller="${propertyName}" action='delete'>
35             <input class="delete" type="submit" value="\${message(code: 'default.button.delete.label',
36               default: 'Delete')})" onclick="return confirm('${message(code:
37                 'default.button.delete.confirm.message', default: 'Are you sure?')}');" />
38           </sec:access>
39         </fieldset>
40       </g:form>
41     </div>
42   </body>
43 </html>

```

Código 3.6: *Template show.gsp*

3.5 Controladores e Visões

Após realizar as alterações discutidas na seção anterior, é necessário executar o comando **generate-all** para que as alterações nos *templates*, discutidos na seção anterior, sejam refletidos nos controladores e visões da aplicação **ControleBancario**. No IDE IntelliJ: abra a janela **Terminal** e execute o comando **grails generate-all br.ufscar.dc.dsw.Agencia**. Repita a execução desse comando para as classes de domínio listadas na Tabela 3.1.

Agencia	Banco	CaixaEletronico	Cidade	ClienteFisico
ClienteJuridico	ContaCliente	ContaCorrente	ContaPoupanca	Endereco
Estado	Gerente	Transacao		

Tabela 3.1: Classes de domínio: geração dos Controladores e Visões.

3.5.1 Controlador: ContaController

O próximo passo consiste na definição do **ContaController** associado à classe de domínio **Conta**. Esse controlador implementa operações que uniformiza o acesso às instâncias das subclasses da classe abstrata **Conta**. Por exemplo, a ação **index()** é responsável por listar contas bancárias (não importando se elas são contas correntes ou contas poupanças).

Para criar um controlador, relacionado à classe de domínio **Conta**, no IDE IntelliJ: abra a janela **Terminal** e execute o comando **grails create-controller br.ufscar.dc.dsw.Conta**. Abra o controlador **ContaController** e implemente-o conforme apresentado no Código 3.7.

```
package br.ufscar.dc.dsw

import static org.springframework.http.HttpStatus.*
import grails.transaction.Transactional
import org.springframework.security.access.annotation.Secured

@Secured('ROLE_GERENTE')
class ContaController {

    static allowedMethods = [save: "POST", update: "PUT", delete: "DELETE"]

    def index(Integer max) {
        params.max = Math.min(max ?: 10, 100)
        respond Conta.list(params), model:[list: Conta.list(params), contaInstanceCount: Conta.count()]
    }

    @Secured(['ROLE_ADMIN', 'ROLE_CLIENTE', 'ROLE_GERENTE'])
    def show() {
        Conta instance = Conta.get(params.id)
        if (instance instanceof ContaCorrente) {
            forward controller: 'contaCorrente', action: "show"
        } else {
            forward controller: 'contaPoupanca', action: "show"
        }
    }
}
```

Código 3.7: Controlador **ContaController**

- A ação **index()** é restrita aos usuários que desempenham o papel **ROLE_GERENTE**; e
- A ação **show()** é restrita aos usuários que desempenham os papéis: **ROLE_ADMIN**, **ROLE_CLIENTE** e **ROLE_GERENTE**. Além disso, essa ação verifica se a instância é uma conta corrente ou conta poupança e direciona para o controlador mais apropriado: **ContaCorrenteController** ou **ContaPoupancaController**.

3.5.2 Visão: conta/index.gsp

Relembrando a discussão da Seção 2.3.2, para cada método correspondente a uma ação em um controlador é criada uma correspondente visão (arquivo com extensão **.gsp**). Assim, a ação **index**, de **ContaController**, tem o correspondente **index.gsp**.

A visão **index.gsp** apresenta uma lista de contas bancárias (não importando se elas são contas correntes ou contas poupanças). A implementação dessa visão encontra-se apresentada no Código 3.8.

```
<%@ page import="br.ufscar.dc.dsw.Conta" %>
<!DOCTYPE html>
<html>
  <head>
    <meta name="layout" content="main">
    <g:set var="entityName" value="${message(code: 'conta.label', default: 'Conta')}" />
    <title><g:message code="default.list.label" args="[entityName]" /></title>
  </head>
  <body>
    <a href="#list-conta" class="skip" tabindex="-1"><g:message code="default.link.skip.label"
      default="Skip to content&hellip;" /></a>

    <div class="nav" role="navigation">
      <ul>
        <li><a class="home" href="${createLink(uri: '/')}"><g:message code="default.home.label" /></a></li>
      </ul>
    </div>
    <div id="list-conta" class="content scaffold-list" role="main">
      <h1><g:message code="default.list.label" args="[entityName]" /></h1>
      <g:if test="${flash.message}">
        <div class="message" role="status">${flash.message}</div>
      </g:if>
      <table>
        <thead>
          <tr>
            <g:sortableColumn property="numero" title="${message(code: 'conta.numero.label', default: 'Numero')}" />
            <th><g:message code="conta.agencia.label" default="Agencia" /></th>
            <g:sortableColumn property="saldo" title="${message(code: 'conta.saldo.label', default: 'Saldo')}" />
            <g:sortableColumn property="abertura" title="${message(code: 'conta.abertura.label',
              default: 'Abertura')}" />
          </tr>
        </thead>
        <tbody>
          <g:each in="${list}" status="i" var="contaInstance">
            <tr class="${(i % 2) == 0 ? 'even' : 'odd'}">
              <td><g:link action="show" controller="${contaInstance.class}" id="${contaInstance.id}">
                ${fieldValue(bean: contaInstance, field: "numero")}</g:link></td>
              <td>${fieldValue(bean: contaInstance, field: "agencia")}</td>
              <td>${fieldValue(bean: contaInstance, field: "saldo")}</td>
              <td><g:formatDate date="${contaInstance.abertura}" /></td>
            </tr>
          </g:each>
        </tbody>
      </table>
      <div class="pagination">
        <g:paginate total="${contaInstanceCount ?: 0}" />
      </div>
    </div>
  </body>
</html>
```

Código 3.8: Visão **conta/index.gsp**

Conforme pode-se observar, essa visão constrói uma tabela HTML com os atributos (número, agência, saldo e data de abertura) das contas bancárias retornadas. É importante salientar que é através da variável **list**, retornada pela ação **index()**, que essa visão tem acesso aos valores dos atributos das contas bancárias.

3.5.3 Controlador: **ClienteController**

Análogo ao **ContaControlador**, o **ClienteController**, associado à classe de domínio **Cliente**, implementa operações que uniformiza o acesso às instâncias das subclasses da classe abstrata **Cliente**. Por exemplo, a ação **index()** é responsável por listar clientes (não importando se eles são clientes físicos ou jurídicos).

Para criar um controlador, relacionado à classe de domínio **Conta**, no IDE IntelliJ: abra a janela **Terminal** e execute o comando **grails create-controller br.ufscar.dc.dsw.Cliente**. Abra o controlador **ClienteController** e implemente-o conforme apresentado no Código 3.9.

```
package br.ufscar.dc.dsw

import static org.springframework.http.HttpStatus.*
import grails.transaction.Transactional
import org.springframework.security.access.annotation.Secured

@Secured('ROLE_GERENTE')
class ClienteController {

    static allowedMethods = [save: "POST", update: "PUT", delete: "DELETE"]
    def index(Integer max) {
        params.max = Math.min(max ?: 10, 100)
        respond Cliente.list(params), model:[list: Cliente.list(params), clienteInstanceCount: Cliente.count()]
    }

    @Secured(['ROLE_ADMIN', 'ROLE_CLIENTE', 'ROLE_GERENTE'])
    def show() {
        Cliente instance = Cliente.get(params.id)
        if (instance instanceof ClienteFisico) {
            forward controller: 'clienteFisico', action: "show"
        } else {
            forward controller: 'clienteJuridico', action: "show"
        }
    }
}
```

Código 3.9: Controlador **ClienteController**

- A ação **index()** é restrita aos usuários que desempenham o papel **ROLE_GERENTE**; e
- A ação **show()** é restrita aos usuários que desempenham os respectivos papéis: **ROLE_ADMIN**, **ROLE_CLIENTE** e **ROLE_GERENTE**. Além disso, essa ação verifica se a instância é um cliente físico ou cliente jurídico e direciona para o controlador mais apropriado: **ClienteFisicoController** ou **ClienteJuridicoController**.

3.5.4 Visão: **cliente/index.gsp**

A visão **index.gsp** apresenta uma lista de clientes (não importando se eles são clientes físicos ou jurídicos). A implementação dessa visão encontra-se apresentada no Código 3.10.

- Conforme pode-se observar, essa visão constrói uma tabela HTML com os atributos (nome, endereço, data de moradia e status) dos clientes retornados. É importante salientar que é através da variável **list**, retornada pela ação **index()**, que essa visão tem acesso aos valores dos atributos dos clientes.

```

<%@ page import="br.ufscar.dc.dsw.Cliente" %>
<!DOCTYPE html>
<html>
<head>
<meta name="layout" content="main">
<g:set var="entityName" value="${message(code: 'cliente.label', default: 'Cliente')}" />
<title><g:message code="default.list.label" args="[entityName]" /></title>
</head>
<body>
<a href="#list-cliente" class="skip" tabindex="-1"><g:message code="default.link.skip.label" default="Skip to content&hellip;" /></a>
<div class="nav" role="navigation">
<ul>
<li><a class="home" href="${createLink(uri: '/')}"><g:message code="default.home.label" /></a></li>
</ul>
</div>
<div id="list-cliente" class="content scaffold-list" role="main">
<h1><g:message code="default.list.label" args="[entityName]" /></h1>
<g:if test="${flash.message}">
<div class="message" role="status"><g:flash.message /></div>
</g:if>
<table>
<thead>
<tr>
<g:sortableColumn property="nome" title="${message(code: 'cliente.nome.label', default: 'Nome')}" />
<th><g:message code="cliente.endereco.label" default="Endereco" /></th>
<g:sortableColumn property="dtMoradia" title="${message(code: 'cliente.dtMoradia.label', default: 'Dt Moradia')}" />
<g:sortableColumn property="status" title="${message(code: 'cliente.status.label', default: 'Status')}" />
</tr>
</thead>
<tbody>
<g:each in="${list}" status="i" var="clienteInstance">
<tr class="${(i % 2) == 0 ? 'even' : 'odd'}">
<td><g:link action="show" controller="${clienteInstance.class}" id="${clienteInstance.id}">${fieldValue(bean: clienteInstance, field: "nome")}</g:link></td>
<td>${fieldValue(bean: clienteInstance, field: "endereco")}</td>
<td><g:formatDate date="${clienteInstance.dtMoradia}" /></td>
<td>${fieldValue(bean: clienteInstance, field: "status")}</td>
</tr>
</g:each>
</tbody>
</table>
<div class="pagination">
<g:paginate total="${clienteInstanceCount ?: 0}" />
</div>
</div>
</body>
</html>

```

Código 3.10: Visão cliente/index.gsp

3.5.5 Mapeamento URL

Pela convenção, ao executar a aplicação, a página principal é a que lista todos os controladores da aplicação. Porém é possível alterar o controlador **URLMappings** – arquivo **grails-app/controllers/controlebancario/URLMapping.groovy** para definir outro controlador/ação padrão. Código 3.11 define, como a página principal, a ação **index()** do controlador **MainController** (Seção 3.5.6).

```

class UrlMappings {
    static mappings = {
        "/$controller/$action?/$id?(.$format)?"{
            constraints {
                // apply constraints here
            }
        }
        "/"(controller: "main")
        "500"(view: '/error')
    }
}

```

Código 3.11: URLMappings.groovy

3.5.6 Controlador: MainController

O **MainController** consiste no controlador principal da aplicação **ControleBancario**. Para criar o controlador **MainController** no IDE IntelliJ: abra a janela **Terminal** e execute o comando **grails create-controller br.ufscar.dc.dsw.MainController**. Abra o controlador **MainController** e implemente-o conforme apresentado no Código 3.12.

```
package br.ufscar.dc.dsw

import org.springframework.security.access.annotation.Secured

@Secured({ 'ROLE_ADMIN', 'ROLE_CLIENTE', 'ROLE_GERENTE' })
class MainController {

    def index() { }
}
```

Código 3.12: Controlador **MainController**

- A ação **index()** é restrita aos usuários que desempenham os respectivos papéis: **ROLE_ADMIN**, **ROLE_CLIENTE** e **ROLE_GERENTE**.

3.5.7 Visão: main/index.gsp

A visão **main/index.gsp** consiste na visão principal da aplicação **ControleBancario**. A implementação dessa visão encontra-se apresentada no Código 3.13.

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta name="layout" content="main">
5 <g:javascript library="jquery" />
6 </head>
7 <body>
8 <div id="status" role="complementary">
9 <h1>Opções</h1>
10 <table>
11 <g:each var="c" in="${grailsApplication.controllerClasses.sort { it.fullName } }">
12 <g:set var="name" value="${c.logicalPropertyName}" />
13 <g:if test="${name != 'logout' && name != 'login' && name != 'main'}">
14 <sec:access url='${createLink(controller: c.logicalPropertyName, base: "/")}'>
15 <tr><td>
16 <g:link controller="${c.logicalPropertyName}">${c.naturalName.replace(" Controller", "")}</g:link>
17 </td></tr>
18 </sec:access>
19 </g:if>
20 </g:each>
21 </table>
22 </div>
23 </body>
24 </html>
```

Código 3.13: Visão **main/index.gsp**

Pode-se observar, pelas linhas 14-18, que a visão apenas apresenta os controladores para qual o usuário autenticado tem permissão de acessá-lo. Assim, para os usuários que desempenham diferentes papéis, a página principal da aplicação **ControleBancario** torna-se diferente.

Figura 3.4(a) apresenta a página principal conforme acessada por um usuário que desempenha o papel **ROLE_ADMIN** enquanto Figura 3.4(b) apresenta a página principal conforme acessada por usuário que desempenha o papel **ROLE_GERENTE**.



(a) Visão: Papel Administrador



(b) Visão: Papel Gerente

Figura 3.4: Visão main/index.gsp: diferentes visões

3.5.8 Controladores: últimas alterações relacionadas ao controle de acesso

Por fim, atualize os controladores conforme o Código 3.14. Pelo código, observa-se que:

- Os controladores **ClienteFisicoController**, **ClienteJuridicoController**, **ContaClienteController**, **ContaCorrenteController**, **ContaPoupancaController** e **EnderecoController** serão apenas acessados por usuários que desempenham o papel de gerente; e
- O controlador **TransacaoController** será apenas acessado por usuários que desempenham o papel de cliente.

```
@Secured('ROLE_GERENTE')
class ClienteFisicoController {
    // Implementação do controlador ClienteFisico
}

@Secured('ROLE_GERENTE')
class ClienteJuridicoController {
    // Implementação do controlador ClienteJuridico
}

@Secured('ROLE_GERENTE')
class ContaClienteController {
    // Implementação do controlador ClienteCliente
}

@Secured('ROLE_GERENTE')
class ContaCorrenteController {
    // Implementação do controlador ContaCorrente
}

@Secured('ROLE_GERENTE')
class ContaPoupancaController {
    // Implementação do controlador ContaPoupanca
}

@Secured('ROLE_GERENTE')
class EnderecoController {
    // Implementação do controlador Endereco
}

@Secured('ROLE_CLIENTE')
class TransacaoController {
    // Implementação do controlador Transacao
}
```

Código 3.14: Controladores - últimas alterações relacionadas ao Controle de Acesso

3.6 Melhorando o leiaute da aplicação: biblioteca de marca

Com o objetivo de melhorar o leiaute da aplicação, essa seção apresenta a implementação de uma biblioteca de marca (*taglib*) que é utilizada em todas as visões da aplicação **ControleBancario**. Os passos são descritos a seguir:

- Para criar uma biblioteca de marca, no IDE GGTS: Selecione **Grails Tools** \Rightarrow **Create TagLib**. Digite **Login** como o nome da biblioteca de marca e clique em **Finish**. Abra a biblioteca de marcas **LoginTagLib** e implemente-o conforme apresentado no Código 3.15. Pelo código-fonte, observa-se que essa classe imprime o nome do papel desempenhado pelo usuário *logado*.

```
class LoginTagLib {
    def springSecurityService
    def loginControl = {
        if (springSecurityService.isLoggedIn()) {
            def usuario = springSecurityService.getCurrentUser()
            def authority = usuario.getAuthorities()[0].getAuthority()
            def papel
            if (authority.equals('ROLE_ADMIN')) {
                papel = "Administrador"
            } else if (authority.equals('ROLE_CLIENTE')) {
                papel = "Cliente"
            } else {
                papel = "Gerente"
            }
            out << "<span style=\\"padding-right:50px\\">" << papel << "</span>"
            out << "<span style=\\"padding-right:25px\\">"
            out << " " [${ link(controller: "logout")("Logout") }] " "
            out << "</span>"
        }
    }
}
```

Código 3.15: Biblioteca de marca **LoginTagLib**

- Inclua o *template _footer* (grails-app/views/layouts/_footer.gsp) com o conteúdo apresentado na Listagem 3.16. Esse *template* será o rodapé presente em todas as visões da aplicação.

```
<div id="footer">
    <hr/> &copy; DC - UFSCar
</div>
```

Código 3.16: *Template _footer.gsp*

- Incluir o *template _header* (grails-app/views/layouts/_header.gsp) com o conteúdo apresentado na Listagem 3.17. Esse *template* será o cabeçalho presente em todas as visões da aplicação. Observa-se que esse *template* utiliza a biblioteca de marcas **LoginTagLib** definida anteriormente.

```
<div id="header">
    <div id = "loginHeader">
        </br>
        <g:loginControl />
    </div>
</div>
```

Código 3.17: *Template _header.gsp*

- No leiaute padrão, utilizado por todas as visões (arquivo `grails-app/views/layouts/main.gsp`), realize as alterações conforme apresentadas pelo Código 3.18. Pelo código-fonte, observa-se que:

- O nome da aplicação (**ControleBancario**) é inserido (linha 29);
- O *template header*, definido anteriormente, é utilizado (linha 40); e
- O *template footer*, definido anteriormente, é utilizado (linha 42).

```

1 <!doctype html>
2 <html lang="en" class="no-js">
3 <head>
4   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
5   <meta http-equiv="X-UA-Compatible" content="IE=edge"/>
6   <title>
7     <g:layoutTitle default="Grails"/>
8   </title>
9   <meta name="viewport" content="width=device-width, initial-scale=1"/>
10  <asset:stylesheet src="application.css"/>
11  <g:layoutHead/>
12 </head>
13 <body>
14   <div class="navbar navbar-default navbar-static-top" role="navigation">
15     <div class="container">
16       <div class="navbar-header">
17         <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
18           <span class="sr-only">Toggle navigation</span>
19           <span class="icon-bar"></span>
20           <span class="icon-bar"></span>
21           <span class="icon-bar"></span>
22         </button>
23         <a class="navbar-brand" href="/#">
24           <i class="fa grails-icon">
25             <asset:image src="grails-cupsonly-logo-white.svg"/>
26           </i>Controle Bancario
27         </a>
28       </div>
29       <div class="navbar-collapse collapse" aria-expanded="false" style="height: 0.8px;">
30         <ul class="nav navbar-nav navbar-right">
31           <g:message code="page.nav" />
32         </ul>
33       </div>
34     </div>
35   </div>
36   <g:render template="/layouts/header" />
37   <g:layoutBody/>
38   <g:render template="/layouts/footer" />
39   <div class="footer" role="contentinfo"></div>
40   <div id="spinner" class="spinner" style="display:none;">
41     <g:message code="spinner.alt" default="Loading&hellip;" />
42   </div>
43   <asset:javascript src="application.js"/>
44 </body>
45 </html>

```

Código 3.18: Leiaute padrão `grails-app/views/layouts/main.gsp`

- Por fim, adicione as linhas no arquivo `grails-app/assets/stylesheets/main.css` conforme apresentado no Código 3.19.

```

#footer {
  font-size: 0.75em;
  font-style: italic;
  padding: 2em 1em 2em 1em;
  margin-bottom: 1em;
  margin-top: 1em;
  clear: both;
}

#loginHeader {
  float: right;
}

```

Código 3.19: Arquivo `main.css`

3.7 Executando a aplicação

Conforme discutido anteriormente, antes de executar a aplicação, instâncias das entidades serão criadas. No caso, serão criados instâncias das entidades (**Estado**, **Cidade**, **Endereco**, etc) na classe **Bootstrap.groovy** que encontra-se no diretório **grails-app/init**. Essa classe é executada durante o *boot* da aplicação e serve, entre outros propósitos, para inicializar a aplicação por exemplo, criando algumas instâncias de objetos.

A implementação da classe **Bootstrap.groovy** é apresentado nos Códigos 3.20 a 3.24. O trecho apresentado no Código 3.20, popula instâncias das classes **Usuario**, **Estado** e **Cidade**. Observa-se que o usuário criado é associado ao papel **ROLE_ADMIN**. Ou seja, o usuário criado desempenha o papel de administrador da aplicação **ControleBancario**.

```
import br.ufscar.dc.dsw.Agency
import br.ufscar.dc.dsw.Banco
import br.ufscar.dc.dsw.CaixaEletronico
import br.ufscar.dc.dsw.Cidade
import br.ufscar.dc.dsw.Cliente
import br.ufscar.dc.dsw.ClienteFisico
import br.ufscar.dc.dsw.ClienteJuridico
import br.ufscar.dc.dsw.ContaCliente
import br.ufscar.dc.dsw.ContaCorrente
import br.ufscar.dc.dsw.ContaPoupanca
import br.ufscar.dc.dsw.Endereco
import br.ufscar.dc.dsw.Estado
import br.ufscar.dc.dsw.Gerente
import br.ufscar.dc.dsw.Papel
import br.ufscar.dc.dsw.Transacao
import br.ufscar.dc.dsw.Usuario
import br.ufscar.dc.dsw.UsuarioPapel

class Bootstrap {

    def init = { servletContext ->

        def adminPapel = Papel.findByAuthority("ROLE_ADMIN") ?:
        new Papel(authority: "ROLE_ADMIN").save()

        def admin = new Usuario(
            username: "admin",
            password: "admin",
            nome: "Administrador",
            enabled : true
        )

        admin.save()
        if (admin.hasErrors()) {
            println admin.errors
        }
        UsuarioPapel.create(admin, adminPapel)

        println 'populando usuário admin - ok'

        def sp = new Estado(sigla: 'SP', nome: 'São Paulo')

        sp.save()
        if (sp.hasErrors()) {
            println sp.errors
        }

        println 'populando estados - ok'

        def sanca = new Cidade(nome: 'São Carlos', estado: sp)

        sanca.save()
        if (sanca.hasErrors()) {
            println sanca.errors
        }

        def sampa = new Cidade(nome: 'São Paulo', estado: sp)

        sampa.save()
        if (sampa.hasErrors()) {
            println sampa.errors
        }

        println 'populando cidades - ok'
    }
}
```

Código 3.20: **Bootstrap.groovy** (1)

O trecho apresentado no Código 3.21, popula instâncias das classes **Endereco**, **Banco** e **Agencia**.

```
def end1 = new Endereco(logradouro: 'R. Conde do Pinhal', numero: 1909,
    bairro: 'Centro', CEP: '13560-648', cidade: sanca)
end1.save()
if (end1.hasErrors()) {
    println end1.errors
}

def end2 = new Endereco(logradouro: 'R. Treze de Maio', numero: 1930,
    bairro: 'Centro', CEP: '13560-647', cidade: sanca)
end2.save()
if (end2.hasErrors()) {
    println end2.errors
}

def end3 = new Endereco(logradouro: 'R. Nilton Coelho de Andrade',
    numero: 772, bairro: 'Vila Maria', CEP: '03092-324', cidade: sampa)
end3.save()
if (end3.hasErrors()) {
    println end3.errors
}

def end4 = new Endereco(logradouro: 'R. Humberto Manelli', numero: 50,
    complemento: 'Apto 31', bairro: 'Jardim Gibertoni',
    CEP: '13562-420', cidade: sanca)
end4.save()
if (end4.hasErrors()) {
    println end4.errors
}

println 'populando endereços - ok'

def bb = new Banco(numero: 1, nome: 'Banco do Brasil',
    CNPJ: '00.000.000/0001-91')
bb.save()
if (bb.hasErrors()) {
    println bb.errors
}

def santander = new Banco(numero: 33, nome: 'Santander',
    CNPJ: '90.400.888/0001-42')
santander.save()
if (santander.hasErrors()) {
    println santander.errors
}

println 'populando bancos - ok'

def agencia1 = new Agencia(numero: 1888, nome: 'Conde do Pinhal',
    endereco: end1, banco: bb)
agencia1.save()
if (agencia1.hasErrors()) {
    println agencia1.errors
}

def agencia2 = new Agencia(numero: 24, nome: 'Treze de Maio',
    endereco: end2, banco: santander)
agencia2.save()
if (agencia2.hasErrors()) {
    println agencia2.errors
}

println 'populando agências - ok'
```

Código 3.21: **BootStrap.groovy (2)**

O trecho apresentado no Código 3.22, popula instâncias das classes **Gerente**, **ClienteFisico** e **ClienteJuridico**. Observa-se que os clientes criados são associados ao papel **ROLE_CLIENTE** enquanto os gerentes criados são associados ao papel **ROLE_GERENTE**.

```
def gerentePapel = Papel.findByAuthority("ROLE_GERENTE")?:
new Papel(authority: "ROLE_GERENTE").save()

def gerente1 = new Gerente(username: 'carlos', password: 'carlos',
    enabled: true, nome: 'Carlos da Silva', rg: '1234 SSP/SP',
    CPF: '129.304.458-07', agencia: agencia1
)
gerente1.save()
if (gerente1.hasErrors()) {
    println gerente1.errors
}

UsuarioPapel.create(gerente1, gerentePapel)

def gerente2 = new Gerente(username: "joao", password: "joao",
    enabled: true, nome: 'João Maria', rg: '3467 SSP/RJ',
    CPF: '018.990.444-50', agencia: agencia2
)
gerente2.save()
if (gerente2.hasErrors()) {
    println gerente2.errors
}

UsuarioPapel.create(gerente2, gerentePapel)

println 'populando gerentes - ok'

def clientePapel = Papel.findByAuthority("ROLE_CLIENTE")?:
new Papel(authority: "ROLE_CLIENTE").save()

def cliFisico = new ClienteFisico(username: 'maria', password: 'maria',
    enabled: true, nome: 'Maria da Silva',
    rg: '13567 SSP/SP', CPF: '018.990.444-50', endereco: end4,
    dtMoradia: new Date(), status: Cliente.ATIVO)
cliFisico.save()
if (cliFisico.hasErrors()) {
    println cliFisico.errors
}

UsuarioPapel.create(cliFisico, clientePapel)

def cliFisico2 = new ClienteFisico(username: 'pedro', password: 'pedro',
    enabled: false, nome: 'Pedro Soares',
    rg: '13567 SSP/SP', CPF: '784.232.889-78', endereco: end4,
    dtMoradia: new Date(), status: Cliente.ATIVO)
cliFisico2.save()
if (cliFisico2.hasErrors()) {
    println cliFisico2.errors
}

UsuarioPapel.create(cliFisico2, clientePapel)

println 'populando clientes fisicos - ok'

def cliJuridico = new ClienteJuridico(username: 'cometa',
    password: 'cometa', enabled: true, nome: 'Viação Cometa S/A',
    CNPJ: '61.084.018/0001-03', endereco: end3,
    dtMoradia: new Date(), status: Cliente.ATIVO)
cliJuridico.save()
if (cliJuridico.hasErrors()) {
    println cliJuridico.errors
}

UsuarioPapel.create(cliJuridico, clientePapel)

println 'populando clientes jurídicos - ok'
```

Código 3.22: **BootStrap.groovy (3)**

O trecho apresentado no Código 3.23, popula instâncias das classes **CaixaEletronico**, **ContaCorrente** e **ContaPoupanca**.

```
def caixa1 = new CaixaEletronico(banco: bb, endereco: end1)
caixa1.save()
if (caixa1.hasErrors()) {
    println agencia1.errors
}

def caixa2 = new CaixaEletronico(banco: santander, endereco: end2)
caixa2.save()
if (caixa2.hasErrors()) {
    println agencia1.errors
}

println 'populando caixas eletrônicos - ok'

def corrente = new ContaCorrente(agencia: agencia1,
    numero: '010414688', saldo: 1000.56d, limite: 500.00d,
    abertura: new Date()
)

corrente.save()
if (corrente.hasErrors()) {
    println corrente.errors
}

println 'populando contas correntes - ok'

def poupanca = new ContaPoupanca(agencia: agencia2,
    numero: '261327', saldo: 10000.56d, juros: 0.50d,
    correcao: 1.20d, dia: 23, abertura: new Date()
)

poupanca.save()
if (poupanca.hasErrors()) {
    println poupanca.errors
}

println 'populando contas poupanças - ok'

def contaCli1 = new ContaCliente(conta: corrente,
    cliente: cliJuridico, titular: true
)

contaCli1.save()
if (contaCli1.hasErrors()) {
    println contaCli1.errors
}

def contaCli2 = new ContaCliente(conta: poupanca,
    cliente: cliFisico, titular: true
)

contaCli2.save()
if (contaCli2.hasErrors()) {
    println contaCli2.errors
}

println 'relacionando contas x clientes - ok'
```

Código 3.23: **BootStrap.groovy (4)**

E por fim, o trecho apresentado no Código 3.24, popula instâncias da classe **Transacao**: depósitos, saques e transferências.

```
def deposito = new Transacao(contaCliente: contaCli2, caixaEletronico: caixa2,
    valor: 50d, data: new Date(), quem: 'Próprio', motivo: 'Depósito',
    tipo: Transacao.CRÉDITO
)

deposito.save()
if (deposito.hasErrors()) {
    println deposito.errors
}

println 'populando depósitos - ok'

def saque = new Transacao(contaCliente: contaCli1, caixaEletronico: caixa1,
    valor: 100d, data: new Date(), quem: 'Próprio', motivo: 'Saque',
    tipo: Transacao.DÉBITO)

saque.save()
if (saque.hasErrors()) {
    println saque.errors
}

println 'populando saques - ok'

def transf1 = new Transacao(contaCliente: contaCli1,
    caixaEletronico: caixa2, valor: 25d, data: new Date(),
    quem: 'Próprio', motivo: 'Transferência', tipo: Transacao.DÉBITO)

def transf2 = new Transacao(contaCliente: contaCli2,
    caixaEletronico: caixa2, valor: 25d, data: new Date(),
    quem: 'Fulano de Tal', motivo: 'Transferência',
    tipo: Transacao.CRÉDITO)

transf1.save()
if (transf1.hasErrors()) {
    println transf1.errors
}

transf2.save()
if (transf2.hasErrors()) {
    println transf2.errors
}

println 'populando transferências - ok'
}

def destroy = {
}
}
```

Código 3.24: **BootStrap.groovy (5)**

Para executar a aplicação, clique no botão direito do mouse no ícone **Run: Grails**. A aplicação é implantada no servidor *Web*, como pode ser observado na janela **Run** do IDE IntelliJ.

- A aplicação pode ser acessada através da URL `http://localhost:8080`. Se o navegador não abrir automaticamente, cole a URL em um navegador e a aplicação será acessada. A página de *Login* será apresentada (Figura 3.5).
- Conforme discutido, três papéis são definidos: **ROLE_ADMIN**, **ROLE_CLIENTE** e **ROLE_GERENTE**. Dessa forma, a página principal (`main/index.gsp`) da aplicação **ControleBancario** torna-se diferente.
- Figura 3.4(a) apresenta a página principal conforme acessada por um usuário que desempenha o papel **ROLE_ADMIN** (por exemplo, usuário: “admin”, senha: “admin”) enquanto Figura 3.4(b) apresenta a página principal conforme acessada por usuário que desempenha o papel **ROLE_GERENTE** (por exemplo, usuário: “joão”, senha: “joão”). Salienta-se que esses usuários foram criados durante o *Bootstrap* da aplicação.



A imagem mostra a interface de login de uma aplicação web. No topo, há uma barra verde com o ícone de um prédio e o texto "Controle Bancario". Abaixo, centralizado, há um formulário branco com o título "Página de Acesso". O formulário contém os campos "Usuário:" e "Senha:" com caixas de entrada adjacentes. Abaixo desses campos, há uma caixa de seleção desativada e o texto "Lembre-me". No final do formulário, há um botão "Login". Na base da página, uma barra cinza contém o texto de copyright: "© Departamento de Computação - Universidade Federal de São Carlos".

Figura 3.5: **ControleBancario**: Página de *Login*

3.8 Considerações finais

Esse capítulo apresentou a segunda versão da implementação da aplicação **ControleBancario**. O código-fonte dessa aplicação (`ControleBancarioV2`) encontra-se disponível em um repositório *GitHub*²¹.

Dando continuidade ao desenvolvimento em Grails, o próximo capítulo apresenta a implementação de novas funcionalidades no contexto da aplicação **ControleBancario**.

²¹URL: <https://github.com/delanobeder/FG>



4 — Controle Bancário: Versão 3

Neste capítulo, dando continuidade ao desenvolvimento em Grails, será apresentado o processo de desenvolvimento da terceira versão da aplicação **ControleBancario**. Nessa versão são incorporadas as seguintes funcionalidades:

- Sintonia fina no controle de acesso:
 - Atribuição de papéis na criação das instâncias da classe de domínio **Gerente** e das instâncias das subclasses da classe **Cliente**;
 - Na segunda versão da aplicação **ControleBancario**, um cliente (físico ou jurídico) pode acessar todas as transações feitas em contas (corrente ou poupança). Na versão, discutida nesse capítulo, clientes apenas terão acesso às transações realizadas nas contas (corrente ou poupança) a eles associadas;
 - Se um cliente possui mais de uma conta (corrente ou poupança), será solicitado que ele escolha a conta (corrente ou poupança) que ele deseja acessar;
 - Analogamente, na segunda versão da aplicação **ControleBancario**, um gerente pode acessar qualquer conta (corrente ou poupança). Na versão, discutida nesse capítulo, gerentes apenas terão acesso às contas pertencentes à agência em que ele trabalha.
- Alteração do controlador **Main**, definido no Capítulo 4, para refletir as mudanças relacionadas ao controle de acesso;
- Alteração da biblioteca de marcas *LoginTagLibrary*, definida no Capítulo 4, para refletir as mudanças relacionadas ao controle de acesso;
- Acesso a um serviço *web* que, dado um CEP como parâmetro, retorna as demais informações de um endereço (logradouro, bairro, cidade, etc). Essas informações serão utilizadas no preenchimento automático dos atributos da classe de domínio **Endereco**.

4.1 Configuração da aplicação

Instalação de *plugins*. Na implementação das funcionalidades da aplicação **ControleBancario**, discutidas nesse capítulo, será utilizado o *plugin* Grails **rest** que adiciona funcionalidades de clientes REST. Ou seja, ao utilizarmos esse *plugin* será possível acessar serviços web REST.

```

1  grails.project.dependency.resolution = {
2      // inherit Grails' default dependencies
3      inherits("global") {
4          // specify dependency exclusions here; for example, uncomment this to disable ehcache:
5          // excludes 'ehcache'
6      }
7      log "error" // log level of Ivy resolver, either 'error', 'warn', 'info', 'debug' or 'verbose'
8      checksums true // Whether to verify checksums on resolve
9      legacyResolve false // whether to do a secondary resolve on plugin installation, not advised and
10                        // here for backwards compatibility
11
12      repositories {
13          inherits true // Whether to inherit repository definitions from plugins
14
15          grailsPlugins()
16          grailsHome()
17          mavenLocal()
18          grailsCentral()
19          mavenCentral()
20          // uncomment these (or add new ones) to enable remote dependency resolution from public Maven repositories
21          mavenRepo "http://repository.codehaus.org"
22          // mavenRepo "http://download.java.net/maven/2/"
23          // mavenRepo "http://repository.jboss.com/maven2/"
24          mavenRepo "http://repo.spring.io/milestone/"
25      }
26
27      dependencies {
28          // specify dependencies here under either 'build', 'compile', 'runtime', 'test' or 'provided' scopes e.g.
29          // runtime 'mysql:mysql-connector-java:5.1.27'
30          runtime 'org.postgresql:postgresql:9.3-1100-jdbc41'
31      }
32
33      plugins {
34          // plugins for the build system only
35          build ":tomcat:7.0.50"
36
37          // plugins for the compile step
38          compile ":scaffolding:2.0.1"
39          compile ':cache:1.1.1'
40
41          // plugins needed at runtime but not for compilation
42          runtime ":hibernate:3.6.10.7" // or ":hibernate4:4.1.11.6"
43          runtime ":database-migration:1.3.8"
44          runtime ":jquery:1.10.2.2"
45          runtime ":resources:1.2.1"
46          // Uncomment these (or add new ones) to enable additional resources capabilities
47          // runtime ":zipper-resources:1.0.1"
48          // runtime ":cached-resources:1.1"
49          // runtime ":yui-minify-resources:0.1.5"
50
51          compile ":br-validation:0.3"
52          compile ":spring-security-core:2.0-RC2"
53          compile ":rest:0.8"
54      }
55  }

```

Código 4.1: BuildConfig.groovy

Para instalar o *plugin rest* adicione uma linha, descrevendo a dependência, no arquivo **BuildConfig.groovy** conforme apresentado na linha 53 do Código 4.1. Porém, desde que o código fonte desse *plugin* encontra-se no repositório da *Codehaus*, é necessário também configurar o endereço desse repositório conforme apresentado na linha 21 do Código 4.1.

4.2 Atribuição de papéis

Na segunda versão da aplicação **ControleBancario**, discutida no Capítulo 3, os papéis são atribuídos durante o *Bootstrap* da aplicação (Código 3.22). Nessa terceira versão da aplicação **ControleBancario**, a atribuição de papéis seguirá a abordagem discutida nas próximas seções.

4.2.1 Classe de domínio Gerente X Papel ROLE_GERENTE

Na terceira versão da aplicação **ControleBancario**, o papel **ROLE_GERENTE** será atribuído automaticamente às todas instâncias da classe de domínio **Gerente**.

Código 4.2 mostra a reimplementação da ação **save()** do controlador **GerenteController** com o objetivo de atribuir automaticamente o papel **ROLE_GERENTE** a todas as instâncias da classe de domínio **Gerente**. Conforme pode-se observar, logo após a operação de **save** na instância da classe de domínio **Gerente** (linha 27), essa instância é associada ao papel **ROLE_GERENTE** (linhas 29-31).

```
1 package br.ufscar.dc.dsw
2
3 import static org.springframework.http.HttpStatus.*
4 import grails.transaction.Transactional
5 import org.springframework.security.access.annotation.Secured
6
7 @Secured('ROLE_ADMIN')
8 @Transactional(readOnly = true)
9 class GerenteController {
10
11     static allowedMethods = [save: "POST", update: "PUT", delete: "DELETE"]
12
13     // Demais ações/métodos do controlador GerenteController
14
15     @Transactional
16     def save(Gerente gerenteInstance) {
17         if (gerenteInstance == null) {
18             notFound()
19             return
20         }
21
22         if (gerenteInstance.hasErrors()) {
23             respond gerenteInstance.errors, view: 'create'
24             return
25         }
26
27         gerenteInstance.save flush:true
28
29         def gerentePapel = Papel.findByAuthority("ROLE_GERENTE")
30
31         UsuarioPapel.create(gerenteInstance, gerentePapel)
32
33         request.withFormat {
34             form {
35                 flash.message = message(code: 'default.created.message',
36                                         args: [message(code: 'gerenteInstance.label',
37                                                         default: 'Gerente'), gerenteInstance.id])
38                 redirect gerenteInstance
39             }
40             '*' { respond gerenteInstance, [status: CREATED] }
41         }
42     }
43 }
```

Código 4.2: Controlador **Gerente** (ação **save()**)

4.2.2 Classe de domínio Cliente X Papel ROLE_CLIENTE

Analógo ao discutido na seção anterior, o papel **ROLE_CLIENTE** será atribuído automaticamente às todas instâncias das subclasses da classe de domínio abstrata **Cliente**.

```
class ClienteFisicoController {
  // Demais ações/atributos/métodos do controlador ClienteFisicoController

  @Transactional
  def save(ClienteFisico clienteFisicoInstance) {
    if (clienteFisicoInstance == null) {
      notFound()
      return
    }
    if (clienteFisicoInstance.hasErrors()) {
      respond clienteFisicoInstance.errors, view: 'create'
      return
    }

    clienteFisicoInstance.enabled = false
    clienteFisicoInstance.save flush:true
    def clientePapel = Papel.findByAuthority("ROLE_CLIENTE")
    UsuarioPapel.create(clienteFisicoInstance, clientePapel)

    request.withFormat {
      form {
        flash.message = message(code: 'default.created.message', args: [message(code: '
          clienteFisicoInstance.label', default: 'ClienteFisico'), clienteFisicoInstance.id])
        redirect clienteFisicoInstance
      }
      '*' { respond clienteFisicoInstance, [status: CREATED] }
    }
  }
}
```

Código 4.3: Controlador **ClienteFisico** (ação **save()**)

Códigos 4.3 e 4.4 apresentam a reimplementação da ação **save()** dos controladores **ClienteFisicoController** e **ClienteJuridicoController** com o objetivo de atribuir automaticamente o papel **ROLE_CLIENTE** a todas as instâncias da classe de domínio **ClienteFisico** e **ClienteJuridico**.

```
class ClienteJuridicoController {
  // Demais ações/atributos/métodos do controlador ClienteJuridicoController

  @Transactional
  def save(ClienteJuridico clienteJuridicoInstance) {
    if (clienteJuridicoInstance == null) {
      notFound()
      return
    }
    if (clienteJuridicoInstance.hasErrors()) {
      respond clienteJuridicoInstance.errors, view: 'create'
      return
    }

    clienteJuridicoInstance.enabled = false
    clienteJuridicoInstance.save flush:true
    def clientePapel = Papel.findByAuthority("ROLE_CLIENTE")
    UsuarioPapel.create(clienteJuridicoInstance, clientePapel)

    request.withFormat {
      form {
        flash.message = message(code: 'default.created.message', args: [message(code: '
          clienteJuridicoInstance.label', default: 'ClienteJuridico'), clienteJuridicoInstance.id])
        redirect clienteJuridicoInstance
      }
      '*' { respond clienteJuridicoInstance, [status: CREATED] }
    }
  }
}
```

Código 4.4: Controlador **ClienteJuridico** (ação **save()**)

4.3 Controle de acesso: Transações

Conforme discutido no Capítulo 3, o acesso às transações é restrito aos usuários que desempenham o papel **ROLE_CLIENTE**. Porém essa abordagem não é suficiente pois um cliente pode acessar todas as transações realizadas em contas (corrente ou poupança) independentemente se essa conta está associada ou não a esse cliente.

Na versão da aplicação **ControleBancario**, discutida nesse capítulo, clientes apenas terão acesso às transações realizadas nas contas a ele associadas.

4.3.1 Controlador: MainController

Conforme discutido, o controlador **MainController**, em conjunto com a visão **index** associada, consiste na página principal da aplicação **ControleBancario**.

Código 4.5 apresenta a reimplementação da ação **index()** do controlador **MainController** com o objetivo de refletir as mudanças relacionadas a nova abordagem de controle de acesso discutida nesse capítulo.

```
1 package br.ufscar.dc.dsw
2
3 import org.springframework.security.access.annotation.Secured
4
5 @Secured({ 'ROLE_ADMIN', 'ROLE_CLIENTE', 'ROLE_GERENTE' })
6 class MainController {
7
8     def springSecurityService
9
10    def index() {
11
12        def usuario = springSecurityService.getCurrentUser()
13        def authority = usuario.getAuthorities()[0].getAuthority()
14
15        if (authority.equals('ROLE_GERENTE')) {
16            if (!session.agencia) {
17                session.agencia = usuario.agencia
18                session.papel = "Gerente"
19            }
20        } else if (authority.equals('ROLE_CLIENTE')) {
21
22            if (!session.papel) {
23                session.papel = "Cliente"
24            }
25
26            if (!session.cliente) {
27                session.cliente = usuario
28            }
29
30            if (!session.contaCliente) {
31                if (session.cliente.contasCliente.size() == 1) {
32                    session.contaCliente = session.cliente.contasCliente[0]
33                } else {
34                    redirect(controller: 'selecionaConta')
35                }
36            }
37        } else {
38            if (!session.papel) {
39                session.papel = "Administrador"
40            }
41        }
42    }
43 }
```

Código 4.5: Controlador **MainController**

Quando um usuário acessa uma aplicação *web*, ele estabelece uma sessão com o servidor. Uma variável de sessão existe desde o instante de sua criação até que ela expire por inatividade, seja voluntariamente (*logout* da aplicação) ou finalizado pela aplicação. Dessa forma, uma variável de sessão é visível a todos os controladores e visões enquanto não expirar por inatividade.

Conforme pode-se observar, quatro variáveis de sessão são utilizadas:

- A variável de sessão **papel** armazena o nome do papel do usuário *logado* (linhas 18, 23 e 39);
- A variável de sessão **agencia** armazena a agência em que trabalha o usuário *logado*, caso este desempenhe o papel **ROLE_GERENTE** (linha 17);
- A variável de sessão **cliente** armazena o usuário *logado*, caso este desempenhe o papel **ROLE_CLIENTE** (linha 27); e
- A variável de sessão **contaCliente** armazena a conta cliente (instância da classe de domínio **ContaCliente**) que o usuário *logado*, caso este desempenhe o papel **ROLE_CLIENTE**, está acessando no momento. Se o cliente possuir apenas uma conta associada, este já será armazenado na variável de sessão **contaCliente** (linha 32). Caso contrário, há um redirecionamento para o controlador **SelecionaContaController** (linha 34).

4.3.2 Biblioteca de marcas: LoginTagLib

No Código 4.6 tem-se a biblioteca de marcas **LoginTagLib** que foi reimplementada para refletir as alterações discutidas nesse capítulo.

Conforme pode-se observar essa classe imprime o nome do papel desempenhado pelo usuário *logado* (variável de sessão **papel**, linha 23). Além disso, essa classe imprime a conta que o usuário *logado* está acessando, considerando que esse desempenhe o papel **ROLE_CLIENTE** (variável de sessão **contaCliente**, linha 15).

Por fim, essa classe imprime a agência em que o usuário *logado* trabalha, caso esse desempenhe o papel **ROLE_GERENTE** (variável de sessão **agencia**, linha 19).

```

1  class LoginTagLib {
2      def springSecurityService
3      def loginControl = {
4          if (springSecurityService.isLoggedIn()) {
5              def usuario = springSecurityService.getCurrentUser()
6              def authority = usuario.getAuthorities()[0].getAuthority()
7              def papel
8
9              def span = "<span style=\"text-align:center;padding-left:25px;padding-right:25px\">"
10
11              StringBuilder sb = new StringBuilder();
12
13              if (session.contaCliente) {
14                  sb.append("Conta: ")
15                  sb.append(session.contaCliente.conta + " [")
16                  sb.append(session.contaCliente.conta.agencia + "]")
17                  sb.append(span)
18              } else if (session.agencia) {
19                  sb.append("Agência: ")
20                  sb.append(session.agencia)
21              }
22
23              out << span
24              out << session.papel
25              out << "</span>"
26              out << span
27              out << sb
28              out << "</span>"
29              out << "<span style=\"padding-right:25px\">"
30              out << "" [{ link(controller: "logout") {"Logout"} }] ""
31              out << "</span>"
32          }
33      }
34  }

```

Código 4.6: Biblioteca de marca **LoginTagLib**

4.3.3 Controlador: **SelecionaContaController**

Caso um cliente possua mais de uma conta (corrente ou poupança), ele deverá escolher que conta (corrente ou poupança) ele deseja acessar. O controlador **SelecionaContaController** é responsável por essa nova funcionalidade. A implementação desse controlador encontra-se apresentada no Código 4.7.

- A ação **index()** simplesmente invoca implicitamente a visão **index.gsp** que será discutida na próxima seção; e
- A ação **selected()** é responsável por: (1) receber, como parâmetro, a conta escolhida pelo usuário *logado* e armazenar essa conta na variável de sessão **contaCliente** e (2) redirecionar a requisição para o controlador **MainController**.

```
package br.ufscar.dc.dsw
import org.springframework.security.access.annotation.Secured

@Secured('ROLE_CLIENTE')
class SelecionaContaController {

    def index() { }

    def selected() {
        session.contaCliente = ContaCliente.get(params.conta)
        redirect(controller: 'main')
    }
}
```

Código 4.7: Controlador **SelecionaContaController**

4.3.4 Visão: **selecionaConta/index.gsp**

Relembrando a discussão da Seção 2.3.2, para cada método correspondente a uma ação em um controlador é criada uma correspondente visão (arquivo com extensão **.gsp**). Assim, a ação **index()**, de **SelecionaContaController**, tem o correspondente **index.gsp**.

A visão **index.gsp** cria um formulário HTML com um campo de seleção que contém as contas clientes associadas ao cliente *logado* (variável de sessão **cliente**). É importante salientar que a submissão do formulário invoca a ação **selected()** do controlador **SelecionaContaController** (Seção 4.3.3). A implementação dessa visão encontra-se apresentada no Código 4.8.

```
<%@ page import="br.ufscar.dc.dsw.ContaCliente" %>
<!DOCTYPE html>
<html>
    <head>
        <meta name="layout" content="main">
    </head>
    <body>
        <div id="status" role="complementary">
            <h1>Selecione Conta:</h1>
            <g:form url="[action: 'selected']" >
                <g:set var="contas" value="{ContaCliente.findAll("from ContaCliente as contaCliente where
                    contaCliente.cliente = ?", [session.cliente])}" />
                <g:select name="conta" from="{contas}"
                    optionKey="id" optionValue="conta">
                    </g:select>
                </br>
                </br>
                <fieldset class="buttons">
                    <g:submitButton name="OK" class="save" value="OK" />
                </fieldset>
            </g:form>
        </div>
    </body>
</html>
```

Código 4.8: Visão **selecionaConta/index.gsp**

4.3.5 Classe de Domínio: Transacao

Código 4.9 mostra a reimplementação da classe de domínio **Transacao** com o objetivo de refletir as mudanças discutidas nesse capítulo. Por questão de brevidade, apenas serão apresentadas as mudanças realizadas nessa classe de domínio. Conforme pode-se observar, foram definidos dois métodos (**getValorReal()** e **getValorAnterior()**) que serão utilizados pelas ações **save()**, **update()** e **delete()** do controlador **TransacaoController**.

```
class Transacao {  
    // Demais atributos/métodos da classe de domínio Transacao  
  
    static transients = ['valorAnterior', 'valorReal']  
  
    double getValorReal() { // Retorna valor negativo caso transação seja um débito  
        return (tipo == DÉBITO) ? -valor : valor;  
    }  
  
    double getValorAnterior() {  
        def ant = this.getPersistentValue('valor') // Retorna o valor do atributo 'valor' antes de ser persistido  
        def tipo = this.getPersistentValue('tipo') // Retorna o valor do atributo 'tipo' antes de ser persistido  
        if (tipo == Transacao.DÉBITO) {  
            ant = -ant // Retorna valor negativo caso transação seja um débito  
        }  
        return ant  
    }  
}
```

Código 4.9: Classe de domínio **Transacao**

4.3.6 Controlador: TransacaoController

Código 4.10 mostra a reimplementação do controlador **TransacaoController** com o objetivo de refletir as mudanças relacionadas a nova abordagem de controle de acesso discutida nesse capítulo. Por questão de brevidade, apenas serão apresentadas as mudanças realizadas nesse controlador.

Relembrando a discussão da Seção 2.3.4, a ação **index()** é responsável por retornar a lista de instâncias da classe de domínio **Transacao**. No caso da implementação apresentada nesse capítulo, a lista é composta apenas pelas transações que foram realizadas na conta escolhida anteriormente pelo usuário *logado* (variável de sessão **contaCliente**).

A ação **create()** é responsável por criar uma instância da classe de domínio **Transacao** que é repassada (retornada) para a visão **create.gsp** (uma página que contém um formulário HTML). Quando o formulário é submetido, a ação **save()** valida os dados e caso, tenha sucesso, grava a instância no banco de dados e redireciona para a ação **show()**. Por outro lado, se os dados são inválidos, a ação **save()** renderiza a visão **create.gsp** novamente para que o usuário corrija os erros encontrados na validação. No caso da implementação apresentada nesse capítulo, a transação criada é associada à conta escolhida anteriormente pelo usuário *logado* (variável de sessão **contaCliente**).

Analogamente, a ação **edit()** é responsável por recuperar uma instância a ser atualizada posteriormente. A instância recuperada é repassada (retornada) para a visão **edit.gsp** (uma página que contém um formulário HTML). Quando o formulário é submetido o método **update()** valida os dados e caso, tenha sucesso, atualiza a instância no banco de dados e redireciona para a ação **show()**. Por outro lado, se os dados são inválidos, a ação **update()** renderiza a visão **edit.gsp** novamente para que o usuário corrija os erros encontrados na validação. É importante salientar que as ações **save()** e **update()** atualizam o saldo da conta para refletir as transações criadas e/ou atualizadas.


```

class TransacaoController {

    // Demais ações/atributos/métodos do controlador TransacaoController

    def index(Integer max) {
        params.max = Math.min(max ?: 10, 100)
        def results = Transacao.findAllByContaCliente(session.contaCliente, params)
        respond results, model:[transacaoInstanceCount: Transacao.count()]
    }

    @Transactional
    def save(Transacao transacaoInstance) {
        if (transacaoInstance == null) {
            notFound()
            return
        }
        if (transacaoInstance.hasErrors()) {
            respond transacaoInstance.errors, view: 'create'
            return
        }
        def conta = transacaoInstance.contaCliente.conta
        conta.saldo += transacaoInstance.getValorReal()
        transacaoInstance.save flush:true
        conta.save flush:true

        request.withFormat {
            form {
                flash.message = message(code: 'default.created.message', args: [message(code: 'transacaoInstance.label', default: 'Transacao'), transacaoInstance.id])
                redirect transacaoInstance
            }
            '*' { respond transacaoInstance, [status: CREATED] }
        }
    }

    def edit(Transacao transacaoInstance) {
        if (transacaoInstance != null && transacaoInstance.contaCliente.id != session.contaCliente.id) {
            flash.message = message(code: 'springSecurity.denied.message', args: [message(code: 'Transacao.label', default: 'Transacao'), transacaoInstance.id])
            redirect action: "index"
        }
        respond transacaoInstance
    }

    @Transactional
    def update(Transacao transacaoInstance) {
        if (transacaoInstance == null) {
            notFound()
            return
        }
        if (transacaoInstance.hasErrors()) {
            respond transacaoInstance.errors, view: 'edit'
            return
        }
        def conta = transacaoInstance.contaCliente.conta
        conta.saldo += (transacaoInstance.getValorReal() - transacaoInstance.getValorAnterior())
        transacaoInstance.save flush:true
        conta.save flush:true

        request.withFormat {
            form {
                flash.message = message(code: 'default.updated.message', args: [message(code: 'Transacao.label', default: 'Transacao'), transacaoInstance.id])
                redirect transacaoInstance
            }
            '*' { respond transacaoInstance, [status: OK] }
        }
    }

    @Transactional
    def delete(Transacao transacaoInstance) {
        if (transacaoInstance.contaCliente.id != session.contaCliente.id) {
            flash.message = message(code: 'springSecurity.denied.message', args: [message(code: 'Transacao.label', default: 'Transacao'), transacaoInstance.id])
            redirect action: "index", method: "GET"
            return
        }
        if (transacaoInstance == null) {
            notFound()
            return
        }
        def conta = transacaoInstance.contaCliente.conta
        conta.saldo -= transacaoInstance.getValorReal()
        transacaoInstance.delete flush:true
        conta.save flush:true

        request.withFormat {
            form {
                flash.message = message(code: 'default.deleted.message', args: [message(code: 'Transacao.label', default: 'Transacao'), transacaoInstance.id])
                redirect action: "index", method: "GET"
            }
            '*' { render status: NO_CONTENT }
        }
    }
}

```

Código 4.10: Controlador **TransacaoController**

Relembrando a discussão da Seção 2.3.3, Grails usa uma convenção para automaticamente configurar o caminho para uma ação em particular. Por exemplo, a URL `http://localhost:8080/ControleBancario/transacao/edit/1` executaria a ação `edit()` na instância da classe de domínio **Transacao** cujo `id` é igual a 1.

Dessa forma, através da digitação de uma URL no navegador *web*, um cliente *logado* pode editar ou remover indevidamente (maliciosamente) uma transação não associada a nenhuma de suas contas. Nesse contexto, as ações `edit()` e `delete()` foram alteradas com o objetivo de prevenir essas atualizações indevidas. Por fim, a ação `delete()` também atualiza o saldo da conta associada (crédito ou débito) para refletir a operação de remoção da transação.

4.3.7 Template `transacao/_form.gsp`

O *template* `_form.gsp`, utilizado tanto pela visão `create.gsp` quanto pela visão `edit.gsp`, representa os campos que devem ser preenchidos/alterados durante a criação/edição de instâncias das classes de domínio.

Código 4.11 mostra as mudanças realizadas no *template* `transacao/_form.gsp` para refletir às novas funcionalidades discutidas nesse capítulo. Por questão de brevidade, apenas serão apresentadas as mudanças realizadas nesse arquivo.

```

1 <%@ page import="br.ufscar.dc.dsw.CaixaEletronico" %>
2 <%@ page import="br.ufscar.dc.dsw.Transacao" %>
3
4 <div class="fieldcontain ${hasErrors(bean: transacaoInstance, field: 'contaCliente', 'error')} required">
5   <label for="contaCliente">
6     <g:message code="transacao.contaCliente.label" default="Conta Cliente" />
7     <span class="required-indicator">*</span>
8   </label>
9
10  <g:select id="contaCliente" name="contaCliente.id" from="${session.contaCliente}" optionKey="id"
11    required="" value="${transacaoInstance?.contaCliente?.id}" class="many-to-one" />
12 </div>
13
14 <div class="fieldcontain ${hasErrors(bean: transacaoInstance, field: 'caixaEletronico', 'error')} ">
15   <label for="caixaEletronico">
16     <g:message code="transacao.caixaEletronico.label" default="Caixa Eletronico" />
17     <span class="required-indicator">*</span>
18   </label>
19
20   <g:set var="caixas" value="${CaixaEletronico.findAll('from CaixaEletronico as caixa where caixa.banco = ?',
21     [session.contaCliente.conta.agencia.banco])}" />
22   <g:select id="caixaEletronico" name="caixaEletronico.id" from="${caixas}" optionKey="id"
23     value="${transacaoInstance?.caixaEletronico?.id}" class="many-to-one" />
24 </div>
25
26 <%= Nenhuma alteração nos demais campos =%>

```

Código 4.11: *Template* `transacao/_form.gsp`

Basicamente foram realizadas duas alterações no *template* `transacao/_form.gsp`. A primeira consiste em alterar o atributo **from** (linha 10) do campo de seleção de tal forma que a transação criada/editada sempre será associada à conta escolhida anteriormente pelo usuário (variável sessão `contaCliente`).

A segunda alteração consiste em modificar o segundo campo de seleção (linha 22) de tal forma que o usuário *logado* apenas possa selecionar caixas eletrônicos pertencentes ao banco que mantém a conta (variável de sessão `contaCliente`) sendo acessada no momento. É importante salientar que o atributo **from** desse campo de seleção é a variável `caixas` que armazena uma lista de instâncias da classe de domínio **CaixaEletronico** obtida através da execução de uma consulta `findAll()` (linhas 20-21) parametrizada. O parâmetro dessa consulta é o banco que mantém a conta sendo acessada no momento.

4.3.8 Executando a aplicação

Após realizar as alterações discutidas nessa seção, sugere-se que a aplicação **ControleBancario** seja executada.

Figura 4.1 apresenta a página principal conforme acessada por um usuário *logado* que desempenha o papel **ROLE_CLIENTE**.



Figura 4.1: Visão **main/index.gsp**: **ROLE_CLIENTE**

Conforme pode-se observar o usuário *logado* tem duas opções disponíveis (dois controladores da aplicação **ControleBancario**):

- **SelecionaConta** caso deseje escolher que conta (corrente ou poupança) ele deseja acessar;
- **Transacao** caso deseje acessar/atualizar a lista de transações realizadas na conta que está sendo acessada no momento. Figura 4.2 apresenta a lista de transações associadas a uma conta do usuário *logado*.

Conta Cliente	Caixa Eletronico	Valor	Data	Quem	Motivo
018.990.444-50 X (Poupança) 261327	Santander - R. Treze de Maio, 1930. Centro 13560-647 São Carlos - SP	50	17/06/2014 09:09:18 BRT	Próprio	Depósito
018.990.444-50 X (Poupança) 261327	Santander - R. Treze de Maio, 1930. Centro 13560-647 São Carlos - SP	25	17/06/2014 09:09:18 BRT	Fulano de Tal	Transferência

Figura 4.2: Lista de transações de uma conta do usuário *logado*

4.4 Controle de acesso: Contas

Conforme discutido no Capítulo 3, o acesso às contas é restrito aos usuários que desempenham o papel **ROLE_GERENTE**. Porém essa abordagem não é suficiente pois um gerente pode acessar todas as contas (corrente ou poupança) independentemente se essa conta pertence ou não a sua agência.

Na versão da aplicação **ControleBancario**, discutida nesse capítulo, gerentes apenas terão acesso às contas pertencentes à agência em que ele trabalha.

4.4.1 Controlador: ContaCorrenteController

Código 4.12 apresenta a reimplementação do controlador **ContaCorrenteController** com o objetivo de refletir as mudanças relacionadas a nova abordagem de controle de acesso discutida nesse capítulo.

Relembrando a discussão da Seção 2.3.4, a ação **index()** é responsável por retornar a lista de instâncias da classe de domínio **ContaCorrente**. No caso da implementação apresentada nesse capítulo, a lista é composta apenas pelas contas correntes que pertencem à agência em que o usuário *logado* trabalha (variável de sessão **agencia**).

Por fim, relembrando a discussão da Seção 4.3.6, é possível que gerentes *logados* acessem de forma indevida (maliciosa) uma conta não associada à agência em que trabalha. Nesse contexto, as ações **edit()** e **delete()** foram alteradas com o objetivo de prevenir essas atualizações indevidas.

! Análogo ao controlador **ContaCorrenteController**, o controlador **ContaPoupancaController** também necessita ser alterado para refletir as mudanças relacionadas a nova abordagem de controle de acesso discutida nesse capítulo. Fica como exercício para o leitor realizar tal alteração.

4.4.2 Template contaCorrente/_form.gsp

O *template* **_form.gsp**, utilizado tanto pela visão **create.gsp** quanto pela visão **edit.gsp**, representa os campos que devem ser preenchidos/alterados durante a criação/edição de instâncias das classes de domínio.

Código 4.13 apresenta as mudanças realizadas no *template* **contaCorrente/_form.gsp** para refletir às novas funcionalidades discutidas nesse capítulo. Por questão de brevidade, apenas serão apresentadas as mudanças realizadas nesse arquivo.

Basicamente foi realizada apenas uma alteração no *template* **contaCorrente/_form.gsp** e consiste em alterar atributo **from** do campo de seleção de tal forma que a conta corrente criada/editada sempre será associada à agência (variável de sessão **agencia**) em que trabalha o usuário *logado*.

! Análogo ao *template* **contaCorrente/_form.gsp**, o *template* **contaPoupanca/_form.gsp** também precisa ser alterado com o intuito de desabilitar o campo de seleção (**Agência**). Fica como exercício para o leitor realizar tal alteração.

```

class ContaCorrenteController {

    // Demais ações/atributos/métodos do controlador ContaCorrenteController

    def index(Integer max) {
        params.max = Math.min(max ?: 10, 100)

        def results = ContaCorrente.findAllByAgencia(session.agencia, params)

        respond results, model:[contaCorrenteInstanceCount: ContaCorrente.count()]
    }

    def edit(ContaCorrente contaCorrenteInstance) {

        if (contaCorrenteInstance != null && contaCorrenteInstance.agencia.id != session.agencia.id) {
            flash.message = message(code: 'springSecurity.denied.message', args: [message(code: 'transacaoInstance.label', default: 'Transacao'), contaCorrenteInstance.id])
            redirect action: "index"
        }

        respond contaCorrenteInstance
    }

    @Transactional
    def delete(ContaCorrente contaCorrenteInstance) {

        if (contaCorrenteInstance.agencia.id != session.agencia.id) {
            flash.message = message(code: 'springSecurity.denied.message', args: [message(code: 'transacaoInstance.label', default: 'Transacao'), contaCorrenteInstance.id])
            redirect action: "index"
            return
        }

        if (contaCorrenteInstance == null) {
            notFound()
            return
        }

        contaCorrenteInstance.delete flush:true

        request.withFormat {
            form {
                flash.message = message(code: 'default.deleted.message', args: [message(code: 'ContaCorrente.label', default: 'ContaCorrente'), contaCorrenteInstance.id])
                redirect action: "index", method: "GET"
            }
            '*' { render status: NO_CONTENT }
        }
    }
}

```

Código 4.12: Controlador **ContaCorrenteController**

```

<@ page import="br.ufscar.dc.dsw.ContaCorrente" %>

<div class="fieldcontain ${hasErrors(bean: contaCorrenteInstance, field: 'agencia', 'error')} required">
    <label for="agencia">
        <g:message code="contaCorrente.agencia.label" default="Agencia" />
        <span class="required-indicator">*</span>
    </label>
    <g:select id="agencia" name="agencia.id" from="${session.agencia}" optionKey="id" required="true" value="${contaCorrenteInstance?.agencia?.id}" disabled class="many-to-one"/>
</div>

<%— Nenhuma alteração nos demais campos —%>

```

Código 4.13: *Template* **contaCorrente/_form.gsp**

4.4.3 Controlador: ContaClienteController

Código 4.14 apresenta a reimplementação do controlador **ContaClienteController** com o objetivo de refletir as mudanças relacionadas a nova abordagem de controle de acesso discutida nesse capítulo.

A ação **index()** é responsável por retornar a lista de instâncias da classe de domínio **ContaCliente** que materializa o relacionamento *muitos-para-muitos* entre as classes de domínio **Conta** e **Cliente**. No caso da implementação apresentada nessa seção, a lista é composta apenas pelas instâncias que estão associadas a contas que pertencem à agência em que o usuário *logado* trabalha (variável de sessão **agencia**).

A ação **save()** valida os dados e caso, tenha sucesso, grava a instância no banco de dados. No caso da implementação apresentada nessa seção, a ação **save()** também habilita o cliente (torna-se um usuário da aplicação) caso esteja desabilitado. No contexto da aplicação **ControleBancario**, um cliente apenas torna-se um usuário (pode realizar a operação de *login*) da aplicação caso tenha uma conta associada.

Além disso, conforme discutido anteriormente, é possível que gerentes *logados* acessem de forma indevida (maliciosa) instâncias dessa classe de domínio. Nesse contexto, as ações **edit()** e **delete()** foram alteradas com o objetivo de prevenir essas atualizações indevidas.

```
class ContaClienteController {
  // Demais ações/atributos/métodos do controlador ContaClienteController

  def index(Integer max) {
    params.max = Math.min(max ?: 10, 100)

    def results = ContaCliente.findAll("from ContaCliente as cc where cc.conta.agencia = :agencia",
      [agencia: session.agencia])

    respond results, model:[contaClienteInstanceCount: ContaCliente.count()]
  }

  @Transactional
  def save(ContaCliente contaClienteInstance) {
    if (contaClienteInstance == null) {
      notFound()
      return
    }

    if (contaClienteInstance.hasErrors()) {
      respond contaClienteInstance.errors, view: 'create'
      return
    }

    contaClienteInstance.save flush:true

    def clienteInstance = contaClienteInstance.cliente

    if (!clienteInstance.enabled) {
      clienteInstance.enabled = true
      clienteInstance.save flush:true
    }

    request.withFormat {
      form {
        flash.message = message(code: 'default.created.message', args: [message(code: 'contaClienteInstance'
          .label', default: 'ContaCliente'), contaClienteInstance.id])
        redirect contaClienteInstance
      }
      '*' { respond contaClienteInstance, [status: CREATED] }
    }
  }
}
```

Código 4.14: Controlador **ContaClienteController**

4.4.4 Template contaCliente/_form.gsp

Código 4.15 apresenta as mudanças realizadas no *template* **transacao/_form.gsp** para refletir às novas funcionalidades discutidas nesse capítulo. Por questão de brevidade, apenas serão apresentadas as mudanças realizadas nesse arquivo.

```

1 <%@ page import="br.ufscar.dc.dsw.Conta" %>
2 <%@ page import="br.ufscar.dc.dsw.ContaCliente" %>
3
4 <div class="fieldcontain" ${hasErrors(bean: contaClienteInstance, field: 'cliente', 'error')} required">
5   <label for="cliente">
6     <g:message code="contaCliente.cliente.label" default="Cliente" />
7     <span class="required-indicator">*</span>
8   </label>
9   <g:select id="cliente" name="cliente.id" from="${br.ufscar.dc.dsw.Cliente.list()}" optionKey="id" required=""
10     value="${contaClienteInstance?.cliente?.id}" disabled="${contaClienteInstance?.cliente?.id != null}"
11     class="many-to-one"/>
12 </div>
13
14 <div class="fieldcontain" ${hasErrors(bean: contaClienteInstance, field: 'conta', 'error')} required">
15   <label for="conta">
16     <g:message code="contaCliente.conta.label" default="Conta" />
17     <span class="required-indicator">*</span>
18   </label>
19   <g:set var="contas" value="${Conta.findAll('from Conta as conta where conta.agencia = ?', [session.agencia])}" />
20   <g:select id="conta" name="conta.id" from="${contas}" optionKey="id" value="${contaClienteInstance?.conta?.id}"
21     required="" disabled="${contaClienteInstance?.conta?.id != null}" class="many-to-one"/>
22 </div>
23
24 <%= Nenhuma alteração nos demais campos =%>

```

Código 4.15: Template **contaCliente/_form.gsp**

Basicamente foram realizadas duas alterações no *template* **contaCliente/_form.gsp**. A primeira consiste em desabilitar os campos de seleção nas operações de edição (linhas 10 e 21) quando o relacionamento *muitos-para-muitos* entre as classes de domínio **Conta** e **Cliente** já foi estabelecido. Ou seja, na edição apenas os demais atributos podem ser atualizados. A segunda alteração consiste em modificar o segundo campo de seleção (linha 20) de tal forma que o usuário *logado* apenas possa selecionar contas (corrente ou poupança) pertencentes à agência em que trabalha (variável de sessão **agencia**).

4.4.5 Controlador: ContaController

Código 4.16 apresenta a reimplementação do controlador **ContaController** com o objetivo de refletir as mudanças relacionadas a nova abordagem de controle de acesso discutida nesse capítulo.

Relembrando a discussão da Seção 2.3.4, a ação **index()** é responsável por retornar a lista de instâncias da classe de domínio **Conta**. No caso da implementação apresentada nesse capítulo, a lista é composta apenas pelas contas que pertencem à agência em que o usuário *logado* trabalha (variável de sessão **agencia**).

```

class ContaController {

    // Demais ações/atributos/métodos do controlador ContaController

    def index(Integer max) {
        params.max = Math.min(max ?: 10, 100)

        def results = Conta.findAllByAgencia(session.agencia, params)

        respond results, model:[list: results, contaInstanceCount: Conta.count()]
    }
}

```

Código 4.16: Controlador **ContaController**

4.4.6 Executando a aplicação

Figura 4.3 apresenta a página principal conforme acessada por um usuário *logado* que desempenha o papel **ROLE_GERENTE**.



Figura 4.3: Visão **main/index.gsp**: **ROLE_GERENTE**

Conforme pode-se observar o usuário *logado* tem oito opções disponíveis (controladores da aplicação **ControleBancario**):

- **Cliente**, **ClienteFísico** e **ClienteJurídico** caso deseje acessar/atualizar a lista de clientes;
- **Conta**, **ContaCorrente** e **ContaPoupança** caso deseje acessar/atualizar a lista de contas da agência do usuário *logado*. Figura 4.4 apresenta a lista de transações associadas a uma conta do usuário *logado*;
- **ContaCliente** caso deseje acessar/atualizar a lista de relacionamentos entre clientes e contas da agência do usuário *logado*;
- **Endereco** caso deseje acessar/atualizar a lista de endereços cadastrados.

Agencia	Numero	Saldo	Abertura
1888 - Banco do Brasil	010414688	1.000,56	17/06/2014 19:36:36 BRT

Figura 4.4: Lista de contas da agência do usuário *logado*

4.5 Preenchimento automático de endereços

Essa seção tem como objetivo incorporar, na aplicação **ControleBancario**, algumas funcionalidades AJAX relacionadas ao acesso a um serviço *web*. Em especial, essa seção apresenta a implementação da funcionalidade de preenchimento automático dos atributos da classe de domínio **Endereco**.

Ou seja, dado o atributo CEP, os demais atributos dessa classe de domínio serão preenchidos automaticamente. Para prover essa funcionalidade, será acessado um serviço *web* que, dado um CEP como parâmetro, retorna as demais informações de um endereço (logradouro, bairro, cidade, etc).

4.5.1 Templates `endereco/_form.gsp` & `endereco/_address.gsp`

O primeiro passo na implementação da funcionalidade de preenchimento automático de endereços consiste em alterar o *template* `endereco/_form.gsp` conforme apresentado no Código 4.17.

Basicamente foram realizadas duas alterações no *template* `endereco/_form.gsp`. A primeira consiste em alterar o campo de texto CEP (linhas 8-10) de tal forma que o tratamento ao evento Javascript *onblur*²² consiste em invocar, passando como parâmetro o conteúdo do campo de texto CEP, a ação `addressByCEP()` do controlador **EnderecoController**.

```

1 <%@ page import="br.ufscar.dc.dsw.Endereco" %>
2
3 <div class="fieldcontain" ${hasErrors(bean: enderecoInstance, field: 'CEP', 'error')} required">
4   <label for="CEP">
5     <g:message code="endereco.CEP.label" default="CEP" />
6     <span class="required-indicator">*</span>
7   </label>
8   <g:textField name="CEP" maxlength="9" required="" value="${enderecoInstance?.CEP}"
9     onblur="{remoteFunction(action: 'addressByCEP', update: {success: 'addressContainer'},
10      params: '\`CEP=\` + this.value ')}"/>
11 </div>
12
13 <div id="addressContainer">
14   <g:render template="address" />
15 </div>

```

Código 4.17: *Template* `endereco/_form.gsp`

A segunda alteração consiste em inserir o *template* `endereco/_address.gsp` (Código 4.18) que contem os demais atributos da classe de domínio **Endereco**. O *template* `endereco/_address.gsp` será atualizado em resposta ao retorno da invocação da ação `addressByCEP()` do controlador **EnderecoController**. Ou seja, as informações retornadas pela ação `addressByCEP()` serão utilizados para o preenchimento automático dos demais atributos da classe de domínio **Endereco**.

É importante salientar que o *template* `endereco/_address.gsp` apenas será atualizado pois encontra-se no escopo de uma *tag* `div` cujo `id` (`addressContainer`) é igual ao valor do atributo `update/success` de `remoteFunction()`.

Uma segunda observação é que os campos **logradouro**, **bairro** e **cidade** no *template* `endereco/_address.gsp` estão desabilitados (não podem ser alterados). Esses valores serão preenchidos automaticamente pelas informações retornadas pela ação `addressByCEP()`.

²²O evento *onblur* ocorre quando um objeto perde o foco.

```

<@ page import="br.ufscar.dc.dsw.Endereco" %>

<div class="fieldcontain" ${hasErrors(bean: enderecoInstance, field: 'logradouro', 'error')} required">
  <label for="logradouro">
    <g:message code="endereco.logradouro.label" default="Logradouro" />
    <span class="required-indicator">*</span>
  </label>
  <g:textField name="logradouro" maxlength="30" required="" disabled value="${enderecoInstance?.logradouro}" />
</div>

<div class="fieldcontain" ${hasErrors(bean: enderecoInstance, field: 'numero', 'error')} required">
  <label for="numero">
    <g:message code="endereco.numero.label" default="Numero" />
    <span class="required-indicator">*</span>
  </label>
  <g:field name="numero" type="number" min="0" value="${enderecoInstance.numero}" required="" />
</div>

<div class="fieldcontain" ${hasErrors(bean: enderecoInstance, field: 'complemento', 'error')} ">
  <label for="complemento">
    <g:message code="endereco.complemento.label" default="Complemento" />
  </label>
  <g:textField name="complemento" maxlength="20" value="${enderecoInstance?.complemento}" />
</div>

<div class="fieldcontain" ${hasErrors(bean: enderecoInstance, field: 'bairro', 'error')} required">
  <label for="bairro">
    <g:message code="endereco.bairro.label" default="Bairro" />
    <span class="required-indicator">*</span>
  </label>
  <g:textField name="bairro" maxlength="20" required="" disabled value="${enderecoInstance?.bairro}" />
</div>

<div class="fieldcontain" ${hasErrors(bean: enderecoInstance, field: 'cidade', 'error')} required">
  <label for="cidade">
    <g:message code="endereco.cidade.label" default="Cidade" />
    <span class="required-indicator">*</span>
  </label>
  <g:select id="cidade" name="cidade.id" from="${br.ufscar.dc.dsw.Cidade.list()}" optionKey="id" required=""
    disabled value="${enderecoInstance?.cidade?.id}" class="many-to-one" />
</div>

```

Código 4.18: *Template endereco/_address.gsp*

4.5.2 Controlador: EnderecoController

Código 4.19 apresenta a implementação da ação **addressByCEP()** que, conforme discutido anteriormente, é invocado pelo *template endereco/_form.gsp*.

Essa ação utiliza o *plugin* que provê funcionalidades para o acesso a serviços *web*. Ou seja, com a utilização das funcionalidades providas por esse *plugin* é possível acessar serviços *web*.

Conforme pode-se observar essa ação invoca um serviço *web* que, dado um CEP como parâmetro, retorna as demais informações de um endereço (logradouro, bairro, cidade, etc).

O serviço *web* retorna o endereço em formato JSON²³ com as seguintes informações:

- **uf** que armazena a sigla da unidade federativa/estado;
- **cidade** que armazena o nome da cidade;
- **bairro** que armazena o nome do bairro;
- **tipo_logradouro** que armazena o tipo do logradouro (rua, avenida, etc); e
- **logradouro** que armazena o nome do logradouro.

Tomando como base as informações retornadas pelo serviço *web*, a ação **addressByCEP()** cria uma instância da classe de domínio **Endereco** e retorna essa instância para ser renderizada pelo *template endereco/_address.gsp*.

²³JSON, acrônimo para *JavaScript Object Notation*, é um formato leve para intercâmbio de dados computacionais.

```

class EnderecoController {

  // Demais ações/atributos/métodos do controlador EnderecoController

  def addressByCEP() {
    def html

    try {
      withHttp(uri: "http://cep.republicavirtual.com.br/") {
        html = get(path: 'web_cep.php',
                    query: [cep:params.CEP, formato:'json'])

        params.estado = Estado.findBySigla(html.uf)
        params.cidade = Cidade.findByNomeAndEstado(html.cidade, params.estado)
        params.bairro = html.bairro
        params.logradouro = html.tipo_logradouro + " " + html.logradouro
      }

      render template: 'address', model: [enderecoInstance: new Endereco(params)]
    } catch (Exception e) {
      println e
    }
  }
}

```

Código 4.19: Controlador **EnderecoController**

Figura 4.5 apresenta a página de cadastro de um endereço (visão **endereco/create.gsp**) em que os atributos **logradouro**, **bairro** e **cidade** foram preenchidos automaticamente pelas informações retornadas pela ação **addressByCEP()**.

Figura 4.5: Visão **endereco/create.gsp**: Preenchimento automático de atributos

4.6 Considerações finais

Esse capítulo apresentou a terceira versão da implementação da aplicação **ControleBancario**. O código-fonte dessa aplicação (**ControleBancarioV3.zip**) encontra-se disponível no *Moodle* do curso, localizado no endereço: <http://moodle.latosensu.dc.ufscar.br>. Seguindo os passos do tutorial apresentado obtém-se esse mesmo código da aplicação **ControleBancario**.

Dando continuidade ao desenvolvimento em Grails, o próximo capítulo apresenta a implementação de novas funcionalidades no contexto da aplicação **ControleBancario**.



5 — Controle Bancário: Versão 4

Neste capítulo, dando continuidade ao desenvolvimento em Grails, será apresentado o processo de desenvolvimento da quarta versão da aplicação **ControleBancario**. Nessa versão são incorporadas as seguintes funcionalidades:

- Personalização dos *templates* utilizados pelo mecanismo de *scaffolding* na geração das visões com o objetivo que estas apresentem leiautes mais responsivos. Para tal, serão utilizados dois *plugins*: **richui** e **pure-css**;
- Conforme discutido anteriormente, as classes de domínio **Cliente** e **Gerente** são subclasses da classe **Usuario** que provê as funcionalidades relacionadas ao controle de acesso. Dessa forma, eles herdam alguns atributos que são específicos às funcionalidades de controle de acesso (**accountExpired**, **accountLocked**, etc). Visando “despoluir” (removendo esses atributos) será realizada a personalização dos atributos, a serem exibidos nas visões, das instâncias dessas classes;
- Implementação da funcionalidade de visualização e impressão de extratos bancários associados às contas bancárias. Para tal, serão utilizados os *plugins* **wkhtmltopdf** e **google-visualization**;
- Implementação de um serviço *web* REST que retorna uma lista (em formato JSON ou XML) de agências de um determinado banco;

5.1 Configuração da aplicação

Instalação de *plugins*. Na implementação das funcionalidades da aplicação **ControleBancario**, discutidas nesse capítulo, serão utilizados os *plugins* Grails **richui**, **pure-css**, **wkhtmltopdf** e **google-visualization**. Conforme discutido anteriormente, para instalar esses *plugins*, adicione as linhas, descrevendo as dependências, no arquivo **BuildConfig.groovy** conforme apresentado nas linhas 54-57 do Código 5.1.

```

1  grails.project.dependency.resolution = {
2      // inherit Grails' default dependencies
3      inherits("global") {
4          // specify dependency exclusions here; for example, uncomment this to disable ehcache:
5          // excludes 'ehcache'
6      }
7      log "error" // log level of Ivy resolver, either 'error', 'warn', 'info', 'debug' or 'verbose'
8      checksums true // Whether to verify checksums on resolve
9      legacyResolve false // whether to do a secondary resolve on plugin installation, not advised and
10                          // here for backwards compatibility
11
12      repositories {
13          inherits true // Whether to inherit repository definitions from plugins
14
15          grailsPlugins()
16          grailsHome()
17          mavenLocal()
18          grailsCentral()
19          mavenCentral()
20          // uncomment these (or add new ones) to enable remote dependency resolution from public Maven repositories
21          mavenRepo "http://repository.codehaus.org"
22          //mavenRepo "http://download.java.net/maven/2/"
23          //mavenRepo "http://repository.jboss.com/maven2/"
24          mavenRepo "http://repo.spring.io/milestone/"
25      }
26
27      dependencies {
28          // specify dependencies here under either 'build', 'compile', 'runtime', 'test' or 'provided' scopes e.g.
29          // runtime 'mysql:mysql-connector-java:5.1.27'
30          runtime 'org.postgresql:postgresql:9.3-1100-jdbc41'
31      }
32
33      plugins {
34          // plugins for the build system only
35          build "tomcat:7.0.50"
36
37          // plugins for the compile step
38          compile "scaffolding:2.0.1"
39          compile "cache:1.1.1"
40
41          // plugins needed at runtime but not for compilation
42          runtime "hibernate:3.6.10.7" // or "hibernate4:4.1.11.6"
43          runtime "database-migration:1.3.8"
44          runtime "jquery:1.10.2.2"
45          runtime "resources:1.2.1"
46          // Uncomment these (or add new ones) to enable additional resources capabilities
47          runtime "zipped-resources:1.0.1"
48          runtime "cached-resources:1.1"
49          runtime "yui-minify-resources:0.1.5"
50
51          compile "br-validation:0.3"
52          compile "spring-security-core:2.0-RC2"
53          compile "rest:0.8"
54          compile "richui:0.8"
55          compile "pure-css:0.4.2"
56          compile "wkhtmltopdf:0.1.7"
57          compile "google-visualization:0.7"
58      }
59  }

```

Código 5.1: BuildConfig.groovy

Instalação do wkhtmltopdf. Na versão, discutida nesse capítulo, extratos bancários, associados às contas bancárias, poderão ser convertidos para o formato pdf. Para tal, será necessária a instalação do **wkhtmltopdf**, que consiste em uma ferramenta *open-source* que converte arquivos **html** em arquivos **pdf**.

A instalação no sistema operacional linux Ubuntu pode ser realizada através do comando **sudo apt-get install wkhtmltopdf -q**, conforme apresentada na Figura 5.1.



Para os demais sistemas operacionais, sugere-se ao leitor que verifique as instruções de instalação disponíveis em: <http://wkhtmltopdf.org>.

```
delano@jequie: ~
delano@jequie:~$ sudo apt-get install wkhtmltopdf -q
Lendo listas de pacotes...
Construindo árvore de dependências...
Lendo informação de estado...
Os NOVOS pacotes a seguir serão instalados:
 wkhtmltopdf
0 pacotes atualizados, 1 pacotes novos instalados, 0 a serem removidos e 11 não
atualizados.
É preciso baixar 0 B/104 kB de arquivos.
Depois desta operação, 303 kB adicionais de espaço em disco serão usados.
Selecionando pacote wkhtmltopdf previamente não selecionado.
(Lendo banco de dados ... 379541 ficheiros e directórios actualmente instalados.
)
Desempacotando wkhtmltopdf (de .../wkhtmltopdf_0.9.9-3_amd64.deb) ...
Processando gatilhos para man-db ...
Configurando wkhtmltopdf (0.9.9-3) ...
delano@jequie:~$
```

Figura 5.1: Instalação do **wkhtmltopdf**

Configuração de *plugins*. Após a instalação dos *plugins* é necessário incluir algumas configurações no arquivo **conf/Config.groovy** (Código 5.2). Por questão de brevidade, apenas serão apresentadas as mudanças realizadas nesse arquivo.

- O primeiro bloco adiciona o formato **pdf** na lista de *MimeTypes* manipulados pela aplicação;
- O segundo bloco configura a maneira em que os *scriptlets* (trechos de código) são tratadas nas visões. Essa alteração é necessária para o bom funcionamento da geração de arquivos **pdf**;
- Por fim, o terceiro bloco indica onde encontra-se instalada a ferramenta **wkhtmltopdf**.

```
grails.mime.types = [ // the first one is the default format
  all:          '*', // 'all' maps to '*' or the first available format in withFormat
  atom:         'application/atom+xml',
  css:         'text/css',
  csv:         'text/csv',
  pdf:         'application/x-pdf',
  form:        'application/x-www-form-urlencoded',
  html:        ['text/html', 'application/xhtml+xml'],
  js:          'text/javascript',
  json:        ['application/json', 'text/json'],
  multipartForm: 'multipart/form-data',
  rss:         'application/rss+xml',
  text:        'text/plain',
  hal:         ['application/hal+json', 'application/hal+xml'],
  xml:         ['text/xml', 'application/xml']
]

grails {
  views {
    gsp {
      encoding = 'UTF-8'
      htmlcodec = 'xml' // use xml escaping instead of HTML4 escaping
      codecs {
        expression = 'html' // escapes values inside ${}
        scriptlet = 'none' // escapes output from scriptlets in GSPs
        taglib = 'none' // escapes output from taglibs
        staticparts = 'none' // escapes output from static template parts
      }
    }
    // escapes all not-encoded output at final stage of outputting
    filteringCodecForContentType {
      // 'text/html' = 'html'
    }
  }
}

grails.plugin.wkhtmltopdf.binary = "/usr/bin/wkhtmltopdf"
```

Código 5.2: **Config.groovy**

5.2 Design responsivo

Com o objetivo que as visões do sistema **ControleBancario** apresentem leiautes mais responsivos, essa seção descreve a personalização dos *templates* utilizados pelo mecanismo de *scaffolding* na geração de visões. É importante salientar que será utilizada, através do *plugin pure-css* instalado anteriormente, a biblioteca CSS **purecss**²⁴ que auxilia no *design* de leiautes responsivos.

5.2.1 Templates: create.gsp e edit.gsp

Conforme discutido, o *template create.gsp* é utilizado na geração das visões **create** associadas a cada um dos controladores da aplicação. No contexto desse capítulo, esse *template* será alterado com o objetivo de utilizar algumas das funcionalidades providas pela biblioteca CSS **purecss**.

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta name="layout" content="main">
5 <g:set var="entityName" value="\${ message(code: '${ domainClass.propertyName }.label ', default: '${ className }')}" />
6 <title><g:message code="default.create.label" args="[entityName]" /></title>
7 <g:javascript src="jquery-1.8.3.min.js"/>
8 <g:javascript src="jquery.maskedinput.min.js"/>
9 <g:javascript>
10     var JQuery = jQuery.noConflict()
11     JQuery(document).ready(function(){
12         JQuery("#CPF").mask("999.999.999-99");
13         JQuery("#CNPJ").mask("99.999.999/9999-99");
14         JQuery("#CEP").mask("99999-999");
15     });
16 </g:javascript>
17 <r:require module="pure-all" />
18 </head>
19 <body>
20 <a href="#create-#{domainClass.propertyName}" class="skip" tabindex="1">
21 <g:message code="default.link.skip.label" default="Skip to content&hellip;" /></a>
22 <div class="pure-menu pure-menu-open pure-menu-horizontal">
23 <ul>
24 <li><a class="home" href="\${ createLink(uri: '/')}"><g:message code="default.home.label" /></a></li>
25 <li><g:link class="list" action="index"><g:message code="default.list.label" args="[entityName]" />
26 <g:link></li>
27 <li><g:link controller="logout">Logout</g:link></li>
28 </ul>
29 </div>
30 <div id="create-#{domainClass.propertyName}" class="content scaffold-create" role="main">
31 <h1><g:message code="default.create.label" args="[entityName]" /></h1>
32 <g:if test="\${ flash.message}">
33 <div class="message" role="status">\${ flash.message}</div>
34 </g:if>
35 <g:hasErrors bean="\${ ${ propertyName }}">
36 <ul class="errors" role="alert">
37 <g:eachError bean="\${ ${ propertyName }}" var="error">
38 <li><g:if test="\${ error in org.springframework.validation.FieldError}">
39 data-field-id="\${ error.field}"</g:if>
40 <g:message error="\${ error}" /></li>
41 </g:eachError>
42 </ul>
43 </g:hasErrors>
44 <g:form class="pure-form pure-form-aligned" url="[resource:${ propertyName }, action: 'save']">
45 <%= multiPart ? ' enctype="multipart/form-data"' : '' %>
46 <fieldset class="form">
47 <g:render template="form"/>
48 </fieldset>
49 <fieldset class="buttons">
50 <g:submitButton name="create" class="save"
51 value="\${ message(code: 'default.button.create.label ', default: 'Create')}" />
52 </fieldset>
53 </g:form>
54 </div>
55 </body>
56 </html>

```

Código 5.3: *Template scaffolding/create.gsp*

²⁴<http://purecss.io>

Código 5.3 mostra as alterações realizadas no *template create.gsp*:

- A linha 7 adiciona a folha de estilo (CSS), definida pela biblioteca **purecss**, nas visões **create** geradas pelo *scaffolding*;
- As linhas 22 e 44 utilizam os estilos definidos pela biblioteca **purecss** para adicionar nas visões, geradas pelo *scaffolding*, dois componentes HTML responsivos: um menu horizontal e um formulário;
- Por fim, a linha 27 adiciona uma opção ao menu horizontal, responsável pela operação de *logout* da aplicação.

Análogo ao **create.gsp**, o *template edit.gsp* também necessita ser alterado com o objetivo de utilizar as funcionalidades providas pela biblioteca CSS **purecss**. Código 5.4 mostra as alterações realizadas no *template edit.gsp*. Conforme pode-se observar, as alterações são similares às realizadas no *template create.gsp*.

```

1 <%=packageName%>
2 <!DOCTYPE html>
3 <html>
4   <head>
5     <meta name="layout" content="main">
6     <g:set var="entityName" value="\${ message(code: '${ domainClass.propertyName }.label ', default: '${ className }) }'" />
7     <title><g:message code="default.edit.label" args="[entityName]" /></title>
8     <g:javascript src="jquery-1.8.3.min.js"/>
9     <g:javascript src="jquery.maskedinput.min.js"/>
10    <g:javascript>
11      var JQuery = jQuery.noConflict()
12      JQuery(document).ready(function(){
13        JQuery("#CPF").mask("999.999.999-99");
14        JQuery("#CNPJ").mask("99.999.999/9999-99");
15        JQuery("#CEP").mask("99999-999");
16      });
17    </g:javascript>
18    <r:require module="pure-all" />
19  </head>
20  <body>
21    <a href="#edit-\${domainClass.propertyName}" class="skip" tabindex="-1">
22    <g:message code="default.link.skip.label" default="Skip to content&hellip;" /></a>
23    <div class="pure-menu pure-menu-open pure-menu-horizontal">
24      <ul>
25        <li><a class="home" href="\${ createLink(uri: '/')}"><g:message code="default.home.label" /></a></li>
26        <li><g:link class="list" action="index"><g:message code="default.list.label" args="[entityName]" />
27          </g:link></li>
28        <li><g:link class="create" action="create"><g:message code="default.new.label" args="[entityName]" />
29          </g:link></li>
30        <li><g:link controller="logout">Logout</g:link></li>
31      </ul>
32    </div>
33    <div id="edit-\${domainClass.propertyName}" class="content scaffold-edit" role="main">
34    <h1><g:message code="default.edit.label" args="[entityName]" /></h1>
35    <g:if test="\${ flash.message}">
36      <div class="message" role="status">\${ flash.message}</div>
37    </g:if>
38    <g:hasErrors bean="\${ ${ propertyName }}">
39      <ul class="errors" role="alert">
40        <g:eachError bean="\${ ${ propertyName }}" var="error">
41          <li><g:if test="\${ error in org.springframework.validation.FieldError}">
42            data-field-id="\${ error.field}"</g:if><g:message error="\${ error}" /></li>
43        </g:eachError>
44      </ul>
45    </g:hasErrors>
46    <g:form class="pure-form pure-form-aligned" url="[resource:${propertyName}, action:'update']" method="PUT">
47      <%= multiPart ? ' enctype="multipart/form-data" : '' %>>
48      <g:hiddenField name="version" value="\${ ${ propertyName }.version}" />
49      <fieldset class="form">
50        <g:render template="form" />
51      </fieldset>
52      <fieldset class="buttons">
53        <g:actionSubmit class="save" action="update" value="\${ message(code: 'default.button.update.label ',
54          default: 'Update')}" />
55      </fieldset>
56    </g:form>
57  </div>
58 </body>
59 </html>

```

Código 5.4: *Template scaffolding/edit.gsp*

5.2.2 Template: index.gsp

Conforme discutido, o *template* **index.gsp** é utilizado na geração das visões **index** associadas a cada um dos controladores da aplicação. No contexto desse capítulo, esse *template* será alterado com o objetivo de utilizar as funcionalidades, providas pela biblioteca CSS **purecss**, responsáveis pela obtenção de leiautes mais responsivos. Por questão de brevidade, apenas serão apresentadas as mudanças realizadas nesse *template*.

```

1  <% import grails.persistence.Event %>
2  <%=packageName%>
3  <!DOCTYPE html>
4  <html>
5    <head>
6      <meta name="layout" content="main">
7      <g:set var="entityName" value="\${ message( code: '${ domainClass.propertyName }.label ', default: '${ className } ' ) }" />
8      <title><g:message code="default.list.label" args="[entityName]" /></title>
9      <r:require module="pure-all" />
10   </head>
11   <body>
12     <a href="#list -\${ domainClass.propertyName }" class="skip" tabindex="-1">
13       <g:message code="default.link.skip.label" default="Skip to content&hellip;" /></a>
14     <div class="pure-menu pure-menu-open pure-menu-horizontal">
15       <ul>
16         <li><a class="home" href="\${ createLink( uri: '/' ) }"><g:message code="default.home.label" /></a></li>
17         <li>
18           <sec:access controller="\${ domainClass.propertyName }" action='create'>
19             <g:link class="create" action="create"><g:message code="default.new.label" args="[entityName]" />
20             </g:link>
21           </sec:access>
22         </li>
23         <li><g:link controller="logout">Logout</g:link></li>
24       </ul>
25     </div>
26     <div id="list -\${ domainClass.propertyName }" class="content scaffold-list" role="main">
27       <h1><g:message code="default.list.label" args="[entityName]" /></h1>
28       <g:if test="\${ flash.message }">
29         <div class="message" role="status">\${ flash.message }</div>
30       </g:if>
31       <table class="pure-table pure-table-bordered">
32
33         <!-- Definição da tabela — Removido por questões de brevidade -->
34
35       </table>
36       <div class="pagination">
37         <g:paginate total="\${ {propertyName} Count ?: 0 }" />
38       </div>
39     </div>
40   </body>
41 </html>

```

Código 5.5: *Template scaffolding/index.gsp*

Código 5.5 mostra as alterações realizadas no *template* **index.gsp**:

- A linha 9 adiciona a folha de estilo (CSS), definida pela biblioteca **purecss**, nas visões **index** geradas pelo *scaffolding*;
- As linhas 14 e 31 utilizam os estilos definidos pela biblioteca **purecss** para adicionar nas visões, geradas pelo *scaffolding*, dois componentes HTML responsivos: um menu horizontal e uma tabela; e
- Por fim, a linha 23 adiciona uma opção ao menu horizontal, responsável pela operação de *logout* da aplicação.

5.2.3 Template: show.gsp

Conforme discutido, o *template show.gsp* é utilizado na geração das visões **show** associadas a cada um dos controladores da aplicação. No contexto desse capítulo, esse *template* será alterado com o objetivo de utilizar as funcionalidades, providas pela biblioteca CSS **purecss**, responsáveis pela obtenção de leiautes mais responsivos. Por questão de brevidade, apenas serão apresentadas as mudanças realizadas nesse *template*.

```

1 <% import grails.persistence.Event %>
2 <%=packageName%>
3 <!--DOCTYPE html-->
4 <html>
5   <head>
6     <meta name="layout" content="main">
7     <g:set var="entityName" value="\${ message( code: '${ domainClass.propertyName }.label ', default: '${ className } ' ) }"/>
8     <title><g:message code="default.show.label" args="[entityName]" /></title>
9     <r:require module="pure-all" />
10  </head>
11  <body>
12    <a href="#show-${domainClass.propertyName}" class="skip" tabindex="-1"><g:message
13      code="default.link.skip.label" default="Skip to content&hellip;" /></a>
14    <div class="pure-menu pure-menu-open pure-menu-horizontal">
15      <ul>
16        <li><a class="home" href="\${ createLink( uri: '/' ) }"><g:message code="default.home.label" /></a></li>
17        <li><g:link class="list" action="index"><g:message code="default.list.label" args="[entityName]" /></g:link>
18        </li><li>
19          <sec:access controller="${domainClass.propertyName}" action='create'>
20            <g:link class="create" action="create"><g:message code="default.new.label" args="[entityName]" /></g:link>
21          </sec:access>
22        </li>
23        <li><g:link controller="logout">Logout</g:link></li>
24      </ul>
25    </div>
26
27    <!-- Definição das demais tags — Removido por questões de brevidade -->
28
29  </body>
30 </html>

```

Código 5.6: *Template scaffolding/show.gsp*

Código 5.6 mostra as alterações realizadas no *template show.gsp*:

- A linha 9 adiciona a folha de estilo (CSS), definida pela biblioteca **purecss**, nas visões **index** geradas pelo *scaffolding*;
- A linha 14 utiliza os estilos definidos pela biblioteca **purecss** para adicionar nas visões, geradas pelo *scaffolding*, um componente HTML responsivo: um menu horizontal;
- Por fim, a linha 23 adiciona uma opção ao menu horizontal, responsável pela operação de *logout* da aplicação.

5.2.4 Biblioteca de marcas: LoginTagLib

Conforme discutido anteriormente, os *templates* adicionaram uma opção ao menu horizontal, responsável pela operação de *logout* da aplicação. Dessa forma, Código 5.7 apresenta a biblioteca de marcas **LoginTagLib** que foi reimplementada para refletir as alterações discutidas nesse capítulo: remoção do *link* para o controlador **Logout** — essa opção agora encontra-se presente no menu horizontal de todas as visões geradas pelo *scaffolding*.

```

1  class LoginTagLib {
2      def springSecurityService
3      def loginControl = {
4          if (springSecurityService.isLoggedIn()) {
5              def usuario = springSecurityService.getCurrentUser()
6              def authority = usuario.getAuthorities()[0].getAuthority()
7              def papel
8
9              def span = "<span style=\\"text-align:center;padding-left:25px;padding-right:25px\\">"
10
11              StringBuilder sb = new StringBuilder();
12
13              if (session.contaCliente) {
14                  sb.append("Conta: ")
15                  sb.append(session.contaCliente.conta)
16                  sb.append(" [")
17                  sb.append(session.contaCliente.conta.agencia)
18                  sb.append("]")
19                  sb.append(span)
20              } else if (session.agencia) {
21                  sb.append("Agência: ")
22                  sb.append(session.agencia)
23              }
24
25              out << span
26              out << session.papel
27              out << "</span>"
28              out << span
29              out << sb
30              out << "</span>"
31          }
32      }
33  }

```

Código 5.7: Biblioteca de marca **LoginTagLib**

5.2.5 Visões: geração automática pelo mecanismo de *Scaffolding*

Após realizar as alterações discutidas nas seções anteriores, é necessário executar o comando **generate-views** para que as alterações nos *templates*, discutidos na seções anteriores, sejam refletidos nas visões da aplicação **ControleBancario**. No IDE GGTS: Selecione **Grails Tools** ⇒ **Grails Command Wizard**. Digite **generate-views** como o nome do comando a ser executado e clique em **Next**. Digite o nome da classe de domínio como o parâmetro do comando e clique em **Next**. Tabela 3.1 lista o nome das classes de domínio que necessitam que as visões sejam geradas novamente.

Observação: Os *templates* **_form.gsp**, associados a cada uma das classes de domínio, não precisam ser gerados novamente. Ou seja, digite a opção **n** para o *template* **_form.gsp** e **y** para as demais visões. Figura 5.2 apresenta a geração das visões associadas a classe de domínio **br.ufscar.dc.dsw.Agencia**.

```

Console  [Markers] [Progress]
generate-views br.ufscar.dc.dsw.Agencia - TERMINATED
.....
...
[Generating views for domain class br.ufscar.dc.dsw.Agencia
File /grails-app/views/agencia/index.gsp already exists. Overwrite?[y,n,a] y
File /grails-app/views/agencia/edit.gsp already exists. Overwrite?[y,n,a] y
File /grails-app/views/agencia/show.gsp already exists. Overwrite?[y,n,a] y
File /grails-app/views/agencia/_form.gsp already exists. Overwrite?[y,n,a] n
File /grails-app/views/agencia/create.gsp already exists. Overwrite?[y,n,a] y
[Finished generation for domain class br.ufscar.dc.dsw.Agencia

```

Figura 5.2: Comando **grails generate-views**

5.3 Personalização dos atributos (Cliente Físico/Jurídico e Gerente)

Conforme discutido as classes de domínio **Cliente** e **Gerente** são subclasses da classe **Usuario** que provê as funcionalidades relacionadas ao controle de acesso. Dessa forma, eles herdam alguns atributos que são específicos às funcionalidades de controle de acesso (**accountExpired**, **accountLocked**, etc). Visando “despoluir” (removendo esses atributos) as visões, relacionadas a essas classes de domínio, serão personalizadas conforme será discutido nessa seção.

Em adição, essa seção discute a utilização da *tag* **dateChooser**, definida pelo *plugin* **richui**, na personalização da entrada de atributos que representam datas (ver Figura 5.3).

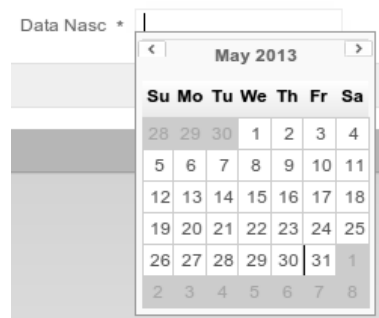


Figura 5.3: Calendário dinâmico: `<richui: dateChooser />`

5.3.1 Cliente Físico, Cliente Jurídico e Gerente

Código 5.8 apresenta a alteração (linha 16) realizada na visão **clienteFisico/create.gsp** com o objetivo de permitir a utilização da *tag* **dateChooser** na entrada de atributos que representam datas. Por questão de brevidade, apenas será apresentada a única mudança realizada nesse arquivo.

```

1 <html>
2 <head>
3   <meta name="layout" content="main">
4   <g:set var="entityName" value="${message(code: 'clienteFisico.label', default: 'ClienteFisico')}"/>
5   <title><g:message code="default.create.label" args="[entityName]" /></title>
6   <g:javascript src="jquery-1.8.3.min.js"/>
7   <g:javascript src="jquery.maskedinput.min.js"/>
8   <g:javascript>
9     var JQuery = jQuery.noConflict()
10    JQuery(document).ready(function(){
11      JQuery("#CPF").mask("999.999.999-99");
12      JQuery("#CNPJ").mask("99.999.999/9999-99");
13      JQuery("#CEP").mask("99999-999");
14    });
15  </g:javascript>
16  <resource:dateChooser />
17  <r:require module="pure-all" />
18 </head>
19 <body>
20
21 <!-- Definição das demais tags — Removido por questões de brevidade -->
22
23 </body>
24 </html>

```

Código 5.8: Visão **clienteFisico/create.gsp**



Análogo à visão **clienteFisico/create.gsp**, as visões **clienteFisico/edit.gsp**, **clienteJuridico/create.gsp** e **clienteJuridico/edit.gsp** também necessitam ser alteradas para permitir a utilização da *tag* **dateChooser** na entrada de atributos que representam datas. Fica como exercício para o leitor realizar tais alterações.

Código 5.9 apresenta o *template* **clienteFisico/_form.gsp** que foi reimplementado para refletir as alterações discutidas nesse capítulo: remoção dos atributos **accountExpired** e **accountLocked**. Além disso, esse *template* utiliza a *tag* **dateChooser** (linha 31) definido pelo *plugin* **richui** instalado anteriormente.

```

1 <@page import="br.ufscar.dc.dsw.ClienteFisico" %>
2 <div class="fieldcontain" ${hasErrors(bean: clienteFisicoInstance, field: 'username', 'error')} required">
3   <label for="username">
4     <g:message code="clienteFisico.username.label" default="Username" />
5   </label>
6   <g:textField name="username" required="" value="${clienteFisicoInstance?.username}"/>
7 </div>
8 <div class="fieldcontain" ${hasErrors(bean: clienteFisicoInstance, field: 'password', 'error')} required">
9   <label for="password">
10    <g:message code="clienteFisico.password.label" default="Password" />
11  </label>
12  <g:field type="password" name="password" required="" value="${clienteFisicoInstance?.password}"/>
13 </div>
14 <div class="fieldcontain" ${hasErrors(bean: clienteFisicoInstance, field: 'nome', 'error')} required">
15   <label for="nome">
16     <g:message code="clienteFisico.nome.label" default="Nome" />
17   </label>
18   <g:textField name="nome" maxlength="30" required="" value="${clienteFisicoInstance?.nome}"/>
19 </div>
20 <div class="fieldcontain" ${hasErrors(bean: clienteFisicoInstance, field: 'endereco', 'error')} required">
21   <label for="endereco">
22     <g:message code="clienteFisico.endereco.label" default="Endereco" />
23   </label>
24   <g:select id="endereco" name="endereco.id" from="${br.ufscar.dc.dsw.Endereco.list()}" optionKey="id"
25     required="" value="${clienteFisicoInstance?.endereco?.id}" class="many-to-one"/>
26 </div>
27 <div class="fieldcontain" ${hasErrors(bean: clienteFisicoInstance, field: 'dtMoradia', 'error')} required">
28   <label for="dtMoradia">
29     <g:message code="clienteFisico.dtMoradia.label" default="Dt Moradia" />
30   </label>
31   <richui:dateChooser name="dtMoradia" format="dd/MM/yyyy" value="${clienteFisicoInstance?.dtMoradia}"/>
32 </div>
33 <div class="fieldcontain" ${hasErrors(bean: clienteFisicoInstance, field: 'status', 'error')} required">
34   <label for="status">
35     <g:message code="clienteFisico.status.label" default="Status" />
36   </label>
37   <g:select name="status" from="${clienteFisicoInstance.constraints.status.inList}" required=""
38     value="${clienteFisicoInstance?.status}" valueMessagePrefix="clienteFisico.status"/>
39 </div>
40 <div class="fieldcontain" ${hasErrors(bean: clienteFisicoInstance, field: 'rg', 'error')} required">
41   <label for="rg">
42     <g:message code="clienteFisico.rg.label" default="Rg" />
43   </label>
44   <g:textField name="rg" maxlength="12" required="" value="${clienteFisicoInstance?.rg}"/>
45 </div>
46 <div class="fieldcontain" ${hasErrors(bean: clienteFisicoInstance, field: 'CPF', 'error')} required">
47   <label for="CPF">
48     <g:message code="clienteFisico.CPF.label" default="CPF" />
49   </label>
50   <g:textField name="CPF" maxlength="14" required="" value="${clienteFisicoInstance?.CPF}"/>
51 </div>
52 <div class="fieldcontain" ${hasErrors(bean: clienteFisicoInstance, field: 'contasCliente', 'error')} ">
53   <label for="contasCliente">
54     <g:message code="clienteFisico.contasCliente.label" default="Contas Cliente" />
55   </label>
56 </div>
57 <ul class="one-to-many">
58 <g:each in="${clienteFisicoInstance?.contasCliente?}" var="c">
59   <li><g:link controller="contaCliente" action="show" id="${c.id}">${c?.encodeAsHTML()}</g:link></li>
60 </g:each>
61 <li class="add">
62   <g:link controller="contaCliente" action="create" params=["clienteFisico.id": clienteFisicoInstance?.id]>
63     ${message(code: 'default.add.label', args: [message(code: 'contaCliente.label', default: 'ContaCliente')])}
64   </g:link>
65 </li>
66 </ul>
67 </div>
68

```

Código 5.9: *Template* **clienteFisico/_form.gsp**



Análogo ao *template* **clienteFisico/_form.gsp**, os *templates* **clienteJuridico/_form.gsp** e **gerente/_form.gsp** também necessitam ser alteradas. Fica como exercício para o leitor realizar tais alterações.

Código 5.10 apresenta a visão **clienteFisico/index.gsp** que foi reimplementada para refletir as alterações discutidas nesse capítulo: a personalização dos atributos a serem apresentados nas visões.

```

1  <@page import="br.ufscar.dc.dsw.ClienteFisico" %>
2  <!DOCTYPE html>
3  <html>
4  <head>
5    <meta name="layout" content="main">
6    <g:set var="entityName" value="${message(code: 'clienteFisico.label', default: 'ClienteFisico')}" />
7    <title><g:message code="default.list.label" args="[entityName]" /></title>
8    <r:require module="pure-all" />
9  </head>
10 <body>
11   <a href="#list-clienteFisico" class="skip" tabindex="-1">
12     <g:message code="default.link.skip.label" default="Skip to content&hellip;" /></a>
13   <div class="pure-menu pure-menu-open pure-menu-horizontal">
14     <ul>
15       <li><a class="home" href="${createLink(uri: '/')}"><g:message code="default.home.label" /></a></li>
16       <li><sec:access controller="clienteFisico" action='create'>
17         <g:link class="create" action="create"><g:message code="default.new.label" args="[entityName]" />
18         </g:link>
19       </sec:access>
20     </li>
21     <li><g:link controller="logout">Logout</g:link></li>
22   </ul>
23 </div>
24 <div id="list-clienteFisico" class="content scaffold-list" role="main">
25   <h1><g:message code="default.list.label" args="[entityName]" /></h1>
26   <g:if test="${flash.message}">
27     <div class="message" role="status"><g:flash.message /></div>
28   </g:if>
29   <table class="pure-table pure-table-bordered">
30     <thead>
31       <tr>
32         <th><g:sortableColumn property="nome" title="${message(code: 'clienteFisico.nome.label', default: 'Nome')}" />
33         <th><g:message code="clienteFisico.endereco.label" default="Endereco" /></th>
34         <th><g:sortableColumn property="dtMoradia" title="${message(code: 'clienteFisico.dtMoradia.label',
35           default: 'Dt Moradia')}" />
36         <th><g:sortableColumn property="status" title="${message(code: 'clienteFisico.status.label',
37           default: 'Status')}" />
38       </tr>
39     </thead>
40     <tbody>
41       <g:each in="${clienteFisicoInstanceList}" status="i" var="clienteFisicoInstance">
42         <tr class="${(i % 2) == 0 ? 'even' : 'odd'}">
43           <td><g:link action="show" id="${clienteFisicoInstance.id}">
44             ${fieldValue(bean: clienteFisicoInstance, field: "nome")}</g:link></td>
45           <td>${fieldValue(bean: clienteFisicoInstance, field: "endereco")}</td>
46           <td><g:formatDate date="${clienteFisicoInstance.dtMoradia}" /></td>
47           <td>${fieldValue(bean: clienteFisicoInstance, field: "status")}</td>
48         </tr>
49       </g:each>
50     </tbody>
51   </table>
52   <div class="pagination">
53     <g:paginate total="${clienteFisicoInstanceCount ?: 0}" />
54   </div>
55 </div>
56 </body>
57 </html>

```

Código 5.10: Visão **clienteFisico/index.gsp**



Análogo à visão **clienteFisico/index.gsp**, as visões **clienteFisico/show.gsp**, **clienteJuridico/index.gsp**, **clienteJuridico/show.gsp**, **gerente/index.gsp** e **gerente/show.gsp** também necessitam ser alterados com o objetivo de personalizar os atributos a serem apresentados. Fica como exercício para o leitor realizar tais alterações.

5.3.2 Cliente

Conforme discutido anteriormente, a classe de domínio **Cliente** é abstrata. Portanto, as visões associadas a essa classe de domínio não são geradas automaticamente pelo mecanismo de *scaffolding*. A visão **cliente/index.gsp** apresenta uma lista de clientes. Código 5.11 apresenta a reimplementação dessa visão com o objetivo da obtenção de um leiaute mais responsivo: menu horizontal (linha 14), formulário (linha 25) e tabela (linha 37).

Adicionalmente, essa visão disponibiliza a funcionalidade *autocomplete*. Além de ser visualmente mais agradável, este recurso faz com que apenas apareçam as opções de acordo com o que é digitado em um campo texto. Para a aplicação **ControleBancario**, a funcionalidade *autocomplete* foi utilizada objetivando auxiliar a busca de clientes (físicos ou jurídicos). A funcionalidade *autocomplete* é implementada pela *tag autocomplete* (linhas 30-32) do *plugin richui* instalado anteriormente.

```

1 <%@ page import="br.ufscar.dc.dsw.Cliente" %>
2 <!DOCTYPE html>
3 <html>
4 <head>
5 <meta name="layout" content="main">
6 <g:set var="entityName" value="${message(code: 'cliente.label', default: 'Cliente')}"/>
7 <title><g:message code="default.list.label" args="[entityName]" /></title>
8 <resource:autocomplete skin="default" />
9 <r:require module="pure-all" />
10 </head>
11 <body>
12 <a href="#list-cliente" class="skip" tabindex="-1">
13 <g:message code="default.link.skip.label" default="Skip to content&hellip;"/></a>
14 <div class="pure-menu pure-menu-open pure-menu-horizontal">
15 <ul>
16 <li><a class="home" href="${createLink(uri: '/')}"><g:message code="default.home.label"/></a></li>
17 <li><g:link controller="logout">Logout</g:link></li>
18 </ul>
19 </div>
20 <div id="list-cliente" class="content scaffold-list" role="main">
21 <h1><g:message code="default.list.label" args="[entityName]" /></h1>
22 <g:if test="${flash.message}">
23 <div class="message" role="status">${flash.message}</div>
24 </g:if>
25 <form class="pure-form">
26 <table>
27 <thead>
28 <th><g:message code="search.label"/></th>
29 <th>
30 <richui:autocomplete name="search"
31 onItemSelect="document.location.href = '${createLinkTo(dir: 'cliente/show')}' + id;"
32 class="pure-input-rounded" action="${createLinkTo('dir: 'cliente/searchAJAX')}" />
33 </th>
34 </thead>
35 </table>
36 </form>
37 <table class="pure-table pure-table-bordered">
38 <thead>
39 <tr><g:sortableColumn property="nome" title="${message(code: 'cliente.nome.label', default: 'Nome')}" />
40 <th><g:message code="cliente.endereco.label" default="Endereco" /></th>
41 <g:sortableColumn property="dtMoradia" title="${message(code: 'cliente.dtMoradia.label',
42 default: 'Dt Moradia')}" />
43 <g:sortableColumn property="status" title="${message(code: 'cliente.status.label',
44 default: 'Status')}" /></tr>
45 </thead>
46 <tbody>
47 <g:each in="${list}" status="i" var="clienteInstance">
48 <tr class="${(i % 2) == 0 ? 'even' : 'odd'}">
49 <td><g:link action="show" id="${clienteInstance.id}">
50 ${fieldValue(bean: clienteInstance, field: "nome")}</g:link></td>
51 <td>${fieldValue(bean: clienteInstance, field: "endereco")}</td>
52 <td><g:formatDate date="${clienteInstance.dtMoradia}" /></td>
53 <td>${fieldValue(bean: clienteInstance, field: "status")}</td></tr>
54 </g:each>
55 </tbody>
56 </table>
57 <div class="pagination">
58 <g:paginate total="${clienteInstanceCount ?: 0}" />
59 </div>
60 </div>
61 </body>
62 </html>

```

Código 5.11: Visão **cliente/index.gsp**

Conforme pode-se observar, a tag **<richui:autoComplete>** (linhas 30-32) invoca assincronamente (sempre que ocorre uma edição no campo de texto **search**) a ação **searchAJAX** do controlador **ClienteController** que realiza uma busca nos clientes baseando-se no que encontra-se digitado no campo texto e retorna as opções possíveis. A implementação da ação **searchAJAX** do controlador **ClienteController** é apresentada no Código 5.12.

```
class ClienteController {
    // Demais ações/atributos/métodos do controlador ClienteController

    def searchAJAX () {
        // busca clientes
        def clientes = Cliente.findAllByNameLike("%${params.query}%")
        // cria resposta XML
        render(contentType: "text/xml") { // retorna os clientes encontrados
            results() {
                clientes.each {
                    cliente -> result() {
                        name(cliente.nome)
                        id (cliente.id)
                    }
                }
            }
        }
    }
}
```

Código 5.12: Controlador **ClienteController**

Figura 5.4 (A) ilustra a situação onde o usuário digitou 'e'. Nesse caso, os dois clientes estão presentes nas opções disponíveis desde que a sentença 'e' encontra-se presente nos nomes dos dois clientes – Pedro Soares e Viação Cometa S/A.

Figura 5.4: Funcionalidade *autocomplete* – Busca de clientes

Figura 5.4 (B) ilustra a situação onde o usuário digitou 'ed'. Nesse caso, apenas um cliente está presente nas opções disponíveis desde que a sentença 'ed' encontra-se presente no nome de apenas um cliente – Pedro Soares. Ao selecionar essa opção, a aplicação apresenta as informações desse cliente (invoca a ação **show** do controlador **Cliente**).

5.4 Personalização das visões: Contas (corrente & poupança)

Essa seção discute a utilização da *tag* **dateChooser**, definida pelo *plugin richui*, na personalização da entrada de atributos (que representam datas) das classes de domínio que representam contas bancárias.

Código 5.13 apresenta a alteração (linha 16) realizada na visão **contaCorrente/create.gsp** com o objetivo de permitir a utilização da *tag* **dateChooser** na entrada de atributos que representam datas. Por questão de brevidade, apenas será apresentada a única mudança realizada nesse arquivo.

```

1 <html>
2 <head>
3   <meta name="layout" content="main">
4   <g:set var="entityName" value="${message(code: 'contaCorrente.label', default: 'ContaCorrente')}"/>
5   <title><g:message code="default.create.label" args="[entityName]" /></title>
6   <g:javascript src="jquery-1.8.3.min.js"/>
7   <g:javascript src="jquery.maskedinput.min.js"/>
8   <g:javascript>
9     var JQuery = jQuery.noConflict()
10    JQuery(document).ready(function(){
11      JQuery("#CPF").mask("999.999.999-99");
12      JQuery("#CNPJ").mask("99.999.999/9999-99");
13      JQuery("#CEP").mask("99999-999");
14    });
15  </g:javascript>
16  <resource:dateChooser />
17  <r:require module="pure-all" />
18 </head>
19 <body>
20
21 <!-- Definição das demais tags — Removido por questões de brevidade -->
22
23 </body>
24 </html>

```

Código 5.13: Visão **contaCorrente/create.gsp**



Análogo à visão **contaCorrente/create.gsp**, as visões **contaCorrente/edit.gsp**, **contaPoupanca/create.gsp** e **contaPoupanca/edit.gsp** também necessitam ser alteradas para permitir a utilização da *tag* **dateChooser** na entrada de atributos que representam datas. Fica como exercício para o leitor realizar tais alterações.

Código 5.14 mostra o *template* **contaCorrente/_form.gsp** que foi reimplementado para refletir as alterações discutidas nesse capítulo: utilização da *tag* **dateChooser**, definida pelo *plugin richui*, na entrada de valores para o atributo **abertura** da classe de domínio **ContaCorrente**.

```

<%@ page import="br.ufscar.dc.dsw.ContaCorrente" %>
<!-- Definição das demais tags — Removido por questões de brevidade -->
<div class="fieldcontain" ${hasErrors(bean: contaCorrenteInstance, field: 'abertura', 'error')} required">
  <label for="abertura">
    <g:message code="contaCorrente.abertura.label" default="Abertura" />
    <span class="required-indicator">*</span>
  </label>
  <richui:dateChooser name="abertura" format="dd/MM/yyyy" value="${contaCorrenteInstance?.abertura}"/>
</div>
<!-- Definição das demais tags — Removido por questões de brevidade -->

```

Código 5.14: *Template* **contaCorrente/_form.gsp**



Análogo ao *template* **contaCorrente/_form.gsp**, o *template* **contaPoupanca/_form.gsp** também necessita ser alterado. Fica como exercício para o leitor realizar tais alterações.

5.4.1 Visão: conta/index.gsp

Conforme discutido anteriormente, a classe de domínio **Conta** é abstrata. Portanto, as visões associadas a essa classe de domínio não são geradas automaticamente pelo mecanismo de *scaffolding*. A visão **conta/index.gsp** apresenta uma lista de contas bancárias (não importando se estas são contas correntes ou poupanças). Código 5.15 apresenta a reimplementação dessa visão com o objetivo da obtenção de um leiaute mais responsivo: menu horizontal (linha 13) e tabela (linha 24). Em complemento, a linha 16 adiciona uma opção ao menu horizontal, responsável pela operação de *logout* da aplicação.

```

1 <%@ page import="br.ufscar.dc.dsw.Conta" %>
2 <!DOCTYPE html>
3 <html>
4 <head>
5 <meta name="layout" content="main">
6 <g:set var="entityName" value="${message(code: 'conta.label', default: 'Conta')}" />
7 <title><g:message code="default.list.label" args="[entityName]" /></title>
8 <r:require module="pure-all" />
9 </head>
10 <body>
11 <a href="#list-Conta" class="skip" tabindex="-1">
12 <g:message code="default.link.skip.label" default="Skip to content&hellip;" /></a>
13 <div class="pure-menu pure-menu-open pure-menu-horizontal">
14 <ul>
15 <li><a class="home" href="${createLink(uri: '/')}"><g:message code="default.home.label" /></a></li>
16 <li><g:link controller="logout">Logout</g:link></li>
17 </ul>
18 </div>
19 <div id="list-Conta" class="content scaffold-list" role="main">
20 <h1><g:message code="default.list.label" args="[entityName]" /></h1>
21 <g:if test="${flash.message}">
22 <div class="message" role="status">${flash.message}</div>
23 </g:if>
24 <table class="pure-table pure-table-bordered">
25 <thead>
26 <tr>
27 <th><g:message code="conta.agencia.label" default="Agencia" /></th>
28 <g:sortableColumn property="numero" title="${message(code: 'conta.numero.label', default: 'Numero')}" />
29 <g:sortableColumn property="saldo" title="${message(code: 'conta.saldo.label', default: 'Saldo')}" />
30 <g:sortableColumn property="abertura" title="${message(code: 'conta.abertura.label',
31 <g:formatDate date="${contaInstance.abertura}" /></td>
32 </tr>
33 </thead>
34 <tbody>
35 <g:each in="${list}" status="i" var="contaInstance">
36 <tr class="${(i % 2) == 0 ? 'even' : 'odd'}">
37 <td><g:link action="show" id="${contaInstance.id}">
38 <g:link>${fieldValue(bean: contaInstance, field: 'agencia')}</g:link></td>
39 <td>${fieldValue(bean: contaInstance, field: 'numero')}</td>
40 <td>${fieldValue(bean: contaInstance, field: 'saldo')}</td>
41 <td><g:formatDate date="${contaInstance.abertura}" /></td>
42 </tr>
43 </g:each>
44 </tbody>
45 </table>
46 <div class="pagination">
47 <g:paginate total="${contaInstanceCount ?: 0}" />
48 </div>
49 </div>
50 </body>
51 </html>

```

Código 5.15: Visão **conta/index.gsp**

5.5 Visões: main/index.gsp & selecionaConta/index.gsp

Relembrando a discussão do Capítulo 3, a visão **main/index.gsp** consiste na visão principal da aplicação **ControleBancario**. Código 5.16 apresenta a reimplementação dessa visão com o objetivo da obtenção de um leiaute mais responsivo: menu horizontal (linha 9) e menu vertical (linha 14). Em complemento, a linha 11 adiciona uma opção ao menu horizontal, responsável pela operação de *logout* da aplicação.

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta name="layout" content="main">
5   <g:javascript library="jquery" />
6   <r:require module="pure-all" />
7 </head>
8 <body>
9   <div class="pure-menu pure-menu-open pure-menu-horizontal">
10    <ul><li><a class="home" href="${createLink(uri: '/')}"><g:message code="default.home.label"/></a></li>
11      <li><g:link controller="logout">Logout</g:link></li></ul>
12    </div>
13    <h2><g:message code="main.options"/></h2>
14    <div class="pure-menu pure-menu-open">
15      <ul>
16        <g:each var="c" in="${grailsApplication.controllerClasses.sort { it.fullName }}">
17          <g:set var="name" value="${c.logicalPropertyName}" />
18          <g:if test="${name != 'logout' && name != 'login' && name != 'main'}">
19            <sec:access url='${createLink(controller: c.logicalPropertyName, base: "/")}'>
20              <li><g:link controller="${c.logicalPropertyName}">${c.naturalName.replace("Controller","")}</g:link></li>
21            </sec:access>
22          </g:if>
23        </g:each>
24      </ul>
25    </div>
26  </body>
27 </html>

```

Código 5.16: Visão **main/index.gsp**

Relembrando a discussão do Capítulo 4, se o cliente *logado* possui mais de uma conta (corrente ou poupança), a ele será solicitado escolher que conta ele deseja acessar. Nesse contexto, a visão **selecionaConta/index.gsp** apresenta um campo de seleção que contém as contas associadas ao cliente *logado* (variável de sessão **cliente**). Código 5.17 apresenta a reimplementação dessa visão com o objetivo da obtenção de um leiaute mais responsivo: menu horizontal (linha 9) e formulário (linha 15). Em complemento, a linha 12 adiciona uma opção ao menu horizontal, responsável pela operação de *logout* da aplicação.

```

1 <%@ page import="br.ufscar.dc.dsw.ContaCliente" %>
2 <!DOCTYPE html>
3 <html>
4   <head>
5     <meta name="layout" content="main">
6     <r:require module="pure-all" />
7   </head>
8   <body>
9     <div class="pure-menu pure-menu-open pure-menu-horizontal">
10      <ul>
11        <li><a class="home" href="${createLink(uri: '/')}"><g:message code="default.home.label"/></a></li>
12        <li><g:link controller="logout">Logout</g:link></li>
13      </ul>
14    </div>
15    <g:form class="pure-form pure-form-aligned" url="[action: 'selected']">
16      <fieldset class="form">
17        <g:set var="contas" value="${ContaCliente.findAll("from ContaCliente as cc where cc.cliente = ?",
18          [session.cliente])}" />
19        <g:select name="conta" from="${contas}" optionKey="id" optionValue="conta">
20          </g:select>
21        </fieldset>
22        <fieldset class="buttons">
23          <g:submitButton name="OK" class="save" value="OK" />
24        </fieldset>
25      </g:form>
26    </body>
27 </html>

```

Código 5.17: Visão **selecionaConta/index.gsp**

5.6 Personalização das visões: Transações

Essa seção discute a utilização da *tag* **dateChooser**, definida pelo *plugin richui*, na personalização da entrada de atributos (que representam datas) das classes de domínio que representam transações.

Código 5.18 apresenta a alteração (linha 16) realizada na visão **transacao/create.gsp** com o objetivo de permitir a utilização da *tag* **dateChooser** na entrada de atributos que representam datas. Por questão de brevidade, apenas será apresentada a única mudança realizada nesse arquivo.

```

1 <html>
2 <head>
3   <meta name="layout" content="main">
4   <g:set var="entityName" value="${message(code: 'transacao.label', default: 'Transacao')}"/>
5   <title><g:message code="default.create.label" args="[entityName]" /></title>
6   <g:javascript src="jquery-1.8.3.min.js"/>
7   <g:javascript src="jquery.maskedinput.min.js"/>
8   <g:javascript>
9     var JQuery = jQuery.noConflict()
10    JQuery(document).ready(function(){
11      JQuery("#CPF").mask("999.999.999-99");
12      JQuery("#CNPJ").mask("99.999.999/9999-99");
13      JQuery("#CEP").mask("99999-999");
14    });
15  </g:javascript>
16  <resource:dateChooser />
17  <r:require module="pure-all" />
18 </head>
19 <body>
20
21 <!-- Definição das demais tags — Removido por questões de brevidade -->
22
23 </body>
24 </html>

```

Código 5.18: Visão **transacao/create.gsp**



Análogo à visão **transacao/create.gsp**, a visão **transacao/edit.gsp** também necessita ser alterada para permitir a utilização da *tag* **dateChooser** na entrada de atributos que representam datas. Fica como exercício para o leitor realizar tais alterações.

Código 5.19 apresenta o *template* **transacao/_form.gsp** que foi reimplementado para refletir as alterações discutidas nesse capítulo: utilização da *tag* **dateChooser**, definida pelo *plugin richui*, na entrada de valores para o atributo **data** da classe de domínio **Transacao**.

```

<%@ page import="br.ufscar.dc.dsw.CaixaEletronico" %>
<%@ page import="br.ufscar.dc.dsw.Transacao" %>

<!-- Definição das demais tags — Removido por questões de brevidade -->

<div class="fieldcontain" ${hasErrors(bean: transacaoInstance, field: 'data', 'error')} required">
  <label for="data">
    <g:message code="transacao.data.label" default="Data" />
    <span class="required-indicator">*</span>
  </label>
  <richui:dateChooser name="data" format="dd/MM/yyyy" value="${transacaoInstance?.data}"/>
</div>

<!-- Definição das demais tags — Removido por questões de brevidade -->

```

Código 5.19: *Template* **transacao/_form.gsp**

5.6.1 Visão: `transacao/show.gsp`

Conforme discutido, a visão `transacao/show.gsp` é responsável pela apresentação das informações das transações. Código 5.20 apresenta a reimplementação dessa visão com o objetivo de melhorar sua estética. Basicamente foram realizadas duas alterações. A primeira consiste em utilizar visualizações em abas (Figura 5.5). Para tal foi utilizado o componente `tabView` do *plugin richui*. A segunda consiste em utilizar menus *accordion*. Analogamente, foi utilizado o componente `accordion` do *plugin richui*. Conforme pode-se observar:

- As tags `<resource:tabView>` e `<resource:accordion>` (linhas 9 e 10) que habilitam a utilização desses componentes na visão `show`;
- A tag `<richui:tabLabel>` (linhas 31-38) define a legenda de cada aba. Figura 5.5 apresenta a visualização em abas da visão `transacao/show.gsp`;

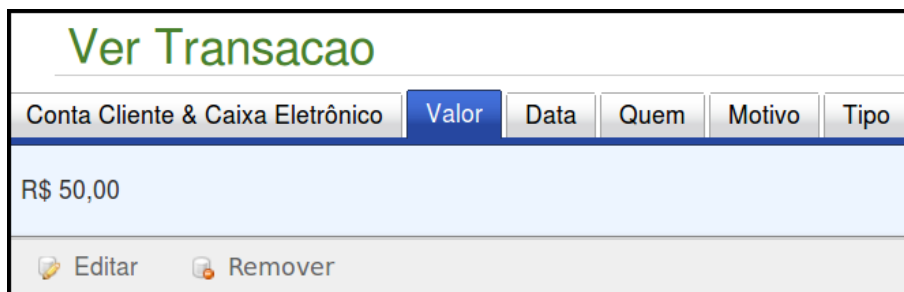


Figura 5.5: Visão `transacao/show.gsp`: visualização em abas

- A tag `<richui:tabContent>` (linhas 39-81) define o conteúdo de cada aba. Ao selecionar uma aba, o conteúdo da aba é apresentado. Por exemplo, O conteúdo da aba **Valor** é apresentado na Figura 5.5.
- No caso da aba **Conta Cliente & Caixa Eletrônico**, foi utilizado um menu *accordion*. O menu *accordion* é definido pelas tags `<richui:accordion>` e `<richui:accordionItem>` (linhas 41-54). Figura 5.6 apresenta as duas opções disponíveis da aba **Conta Cliente & Caixa Eletrônico**.



Figura 5.6: Visão `transacao/show.gsp`: menu *accordion*

```

1 <%@ page import="br.ufscar.dc.dsw.Transacao" %>
2 <!DOCTYPE html>
3 <html>
4 <head>
5 <meta name="layout" content="main">
6 <g:set var="entityName" value="{message(code: 'transacao.label', default: 'Transacao')}" />
7 <title><g:message code="default.show.label" args="[entityName]" /></title>
8 <r:require module="pure-all" />
9 <resource:tabView />
10 <resource:accordion skin="default" />
11 </head>
12 <body>
13 <a href="#show-transacao" class="skip" tabindex="1">
14 <g:message code="default.link.skip.label" default="Skip to content&hellip;" /></a>
15 <div class="pure-menu pure-menu-open pure-menu-horizontal">
16 <ul>
17 <li><a class="home" href="{createLink(uri: '/')}"><g:message code="default.home.label" /></a></li>
18 <li><g:link class="list" action="index"><g:message code="default.list.label" args="[entityName]" /></g:link>
19 </li><li><sec:access controller="transacao" action="create">
20 <g:link class="create" action="create"><g:message code="default.new.label" args="[entityName]" /></g:link>
21 </sec:access></li>
22 <li><g:link controller="logout">Logout</g:link></li>
23 </ul>
24 </div>
25 <div id="show-transacao" class="content scaffold-show" role="main">
26 <h1><g:message code="default.show.label" args="[entityName]" /></h1>
27 <g:if test="{flash.message}">
28 <div class="message" role="status"><g:flash.message /></div>
29 </g:if>
30 <richui:tabView id="tabView">
31 <richui:tabLabels>
32 <richui:tabLabel selected="true" title="{message(code: 'transacao.main')}" />
33 <richui:tabLabel title="{message(code: 'transacao.value')}" />
34 <richui:tabLabel title="{message(code: 'transacao.data')}" />
35 <richui:tabLabel title="{message(code: 'transacao.quem')}" />
36 <richui:tabLabel title="{message(code: 'transacao.motivo')}" />
37 <richui:tabLabel title="{message(code: 'transacao.tipo')}" />
38 </richui:tabLabels>
39 <richui:tabContents>
40 <richui:tabContent>
41 <richui:accordion>
42 <richui:accordionItem id="1" caption="Conta Cliente">
43 <g:if test="{transacaoInstance?.contaCliente}">
44 <g:link controller="contaCliente" action="show" id="{transacaoInstance?.contaCliente?.id}">
45 <g:transacaoInstance?.contaCliente?.encodeAsHTML() /></g:link>
46 </g:if>
47 </richui:accordionItem>
48 <richui:accordionItem caption="Caixa Eletrônico">
49 <g:if test="{transacaoInstance?.caixaEletronico}">
50 <g:link controller="caixaEletronico" action="show" id="{transacaoInstance?.caixaEletronico?.id}">
51 <g:transacaoInstance?.caixaEletronico?.encodeAsHTML() /></g:link>
52 </g:if>
53 </richui:accordionItem>
54 </richui:accordion>
55 </richui:tabContent>
56 <richui:tabContent>
57 <g:if test="{transacaoInstance?.valor}">
58 <g:formatNumber number="{transacaoInstance?.valor}" type="currency" />
59 </g:if>
60 </richui:tabContent>
61 <richui:tabContent>
62 <g:if test="{transacaoInstance?.data}">
63 <g:formatDate date="{transacaoInstance?.data}" type="datetime" style="LONG" timeStyle="SHORT" />
64 </g:if>
65 </richui:tabContent>
66 <richui:tabContent>
67 <g:if test="{transacaoInstance?.quem}">
68 <g:fieldValue bean="{transacaoInstance}" field="quem" />
69 </g:if>
70 </richui:tabContent>
71 <richui:tabContent>
72 <g:if test="{transacaoInstance?.motivo}">
73 <g:fieldValue bean="{transacaoInstance}" field="motivo" />
74 </g:if>
75 </richui:tabContent>
76 <richui:tabContent>
77 <g:if test="{transacaoInstance?.tipo}">
78 <g:fieldValue bean="{transacaoInstance}" field="tipo" /></span>
79 </g:if>
80 </richui:tabContent>
81 </richui:tabContents>
82 </richui:tabView>
83 <g:form url="{resource:transacaoInstance, action: 'delete'}" method="DELETE">
84 <fieldset class="buttons">
85 <sec:access controller="transacao" action="edit">
86 <g:link class="edit" action="edit" resource="{transacaoInstance}">
87 <g:message code="default.button.edit.label" default="Edit" /></g:link>
88 </sec:access>
89 <sec:access controller="transacao" action="delete">
90 <g:actionSubmit class="delete" action="delete" value="{message(code: 'default.button.delete.label',
91 default: 'Delete')}" onclick="return confirm('{message(code: 'default.button.delete.confirm.message',
92 default: 'Are you sure?')}');" />
93 </sec:access>
94 </fieldset>
95 </g:form>
96 </div>
97 </body>
98 </html>

```

Código 5.20: Template transacao/_form.gsp

5.7 Extratos Bancários

O controlador **ExtratoController** é responsável pela implementação da funcionalidade de visualização e impressão de extratos bancários associados às contas bancárias. A implementação desse controlador encontra-se apresentada no Código 5.21.

```

1 package br.ufscar.dc.dsw
2 import java.text.SimpleDateFormat
3 import org.springframework.security.access.annotation.Secured
4
5 @Secured('ROLE_CLIENTE')
6 class ExtratoController {
7
8     def index() {
9         def linhas = getLinhas()
10        if (params?.format && params.format == "pdf") {
11            render( filename:"extrato.pdf",
12                  view:"/extrato/_pdf",
13                  model:[lines: linhas, contaCliente:session.contaCliente],
14                  marginLeft:20,
15                  marginTop:35,
16                  marginBottom:20,
17                  marginRight:20,
18                  headerSpacing:10,
19            )
20        }
21        model:[lines: linhas]
22    }
23
24    def chart = {
25        def columns = [['date', 'Dia'], ['number', 'Saldo (R$)']]
26        def lines = []
27        def linhas = getLinhas()
28        def anterior = null
29        SimpleDateFormat formato = new SimpleDateFormat("dd/MM/yyyy")
30
31        linhas.reverse().each {
32            def atual = formato.format(it.data)
33            if (!atual.equals(anterior)) {
34                lines.add([it.data, it.saldo])
35            }
36            anterior = it.data
37        }
38
39        ["columns": columns, "lines": lines]
40    }
41
42    private List<Line> getLinhas() {
43        def conta = Conta.get(session.contaCliente.conta.id)
44        def saldo = conta.saldo
45        def results = Transacao.findAllByContaCliente(session.contaCliente, [sort:"data"])
46        results.each {
47            saldo -= it.getValorReal()
48        }
49
50        List<Line> linhas = new ArrayList<Line>();
51        Line linha = new Line(session.contaCliente.conta.abertura, "ABERTURA", saldo)
52        linhas.add(linha)
53
54        results.each {
55            saldo += it.getValorReal()
56            linha = new Line(it.data, it.tipo, saldo, it.valor, it.motivo)
57            linhas.add(linha)
58        }
59
60        linha = new Line(new Date(), "SALDO ATUAL", saldo)
61        linhas.add(linha)
62        return linhas
63    }
64 }

```

Código 5.21: Controlador **ExtratoController**

- A ação **index()** (linhas 7-21) simplesmente invoca implicitamente a visão **index.gsp** (Código 5.23) que representa extratos bancários em formato HTML. Adicionalmente, essa ação renderiza, utilizando o *plugin wkhtmltopdf*, o arquivo **extrato.pdf** baseado no *template extrato/_pdf.gsp* (Código 5.24). Ou seja, essa ação é também responsável por gerar os arquivos que representam extratos bancários em formato PDF;

- A ação **chart()** (linhas 23-38) simplesmente invoca implicitamente a visão **chart.gsp** (Código 5.25) que será discutida na Seção 5.7.1;
- É importante salientar que as ações (e respectivas visões) utilizam o método privado **getLinhas()** (linhas 40-58) retorna uma lista de instâncias da classe **Line**. Conforme pode-se observar, a lista de instâncias da classe **Line** é construída através da iteração da lista de transações (ordenadas pela data) associadas às contas bancárias;
- A classe Java **Line**²⁵ (Código 5.22) armazena as informações (atributos: **data**, **tipo**, **motivo**, **valor** e **saldo**) relacionadas às transações e representa cada linha do extrato bancário a ser visualizado e/ou impresso.

```
package br.ufscar.dc.dsw;

import java.util.Date;

public class Line implements Comparable<Line>{

    private final Date data;
    private final String tipo;
    private final String motivo;
    private final Double valor;
    private final Double saldo;

    public Line(Date data, String tipo, Double saldo, Double valor, String motivo) {
        this.data = data;
        this.tipo = tipo;
        this.saldo = saldo;
        this.valor = valor;
        this.motivo = motivo;
    }

    public Line(Date data, String tipo, double saldo) {
        this(data, tipo, saldo, null, null);
    }

    public Date getData() {
        return data;
    }

    public String getTipo() {
        return tipo;
    }

    public String getMotivo() {
        return motivo;
    }

    public Double getValor() {
        return valor;
    }

    public Double getSaldo() {
        return saldo;
    }

    @Override
    public int compareTo(Line o) {
        return this.data.compareTo(o.data);
    }

    @Override
    public String toString() {
        return data.toString() + " - " + valor + " - " + saldo;
    }
}
```

Código 5.22: Classe Java **Line**

²⁵Arquivo: `src/br/ufscar/dc/dsw/Line.java`

5.7.1 Visões

Relembrando a discussão da Seção 2.3.2, para cada método correspondente a uma ação em um controlador é criada uma correspondente visão (arquivo com extensão **.gsp**). Assim, a ação **index()**, de **ExtratoController**, tem o correspondente **index.gsp** que representa extratos bancários em formato HTML. A implementação dessa visão encontra-se apresentada no Código 5.23.

Figura 5.7 apresenta um exemplo de um extrato bancário representado pela visão **extrato/index.gsp**. É importante salientar que essa visão já apresenta um leiaute mais responsivo: menu horizontal (linha 14) e tabela (linha 26).

```

1 <@ page import="org.grails.plugins.google.visualization.data.Cell;
2   org.grails.plugins.google.visualization.util.DateUtil" %>
3 <@ page import="br.ufscar.dc.dsw.Transacao" %>
4 <!DOCTYPE html>
5 <html>
6 <head>
7   <meta name="layout" content="main">
8   <title><g:message code="extrato.statement" /></title>
9   <r:require module="pure-all" />
10 </head>
11 <body>
12   <a href="#list-transacao" class="skip" tabindex="-1">
13     <g:message code="default.link.skip.label" default="Skip to content&hellip;" /></a>
14   <div class="pure-menu pure-menu-open pure-menu-horizontal">
15     <ul>
16       <li><a class="home" href="{createLink(uri: '/')}"><g:message code="default.home.label" /></a></li>
17       <li><g:link action="chart"><g:message code="extrato.chart" default="Chart" /></g:link></li>
18       <li><g:link controller="logout">Logout</g:link></li>
19     </ul>
20   </div>
21   <div id="list-transacao" class="content scaffold-list" role="main">
22     <h1><g:message code="extrato.statement" /></h1>
23     <g:if test="{flash.message}">
24       <div class="message" role="status"><g:flash.message></div>
25     </g:if>
26     <table class="pure-table pure-table-bordered">
27       <thead>
28         <tr>
29           <th><g:message code="transacao.data" /></th>
30           <th><g:message code="transacao.tipo" /></th>
31           <th><g:message code="transacao.motivo" /></th>
32           <th><g:message code="extrato.valor" /></th>
33           <th><g:message code="extrato.saldo" /></th>
34         </tr>
35       </thead>
36       <tbody>
37         <g:each in="{lines}" status="i" var="linha">
38           <tr class="{(i % 2) == 0 ? 'even' : 'odd'}">
39             <td><g:formatDate date="{linha.data}" type="date" style="SHORT" /></td>
40             <td><g:fieldValue(bean: linha, field: "tipo")></td>
41             <td><g:fieldValue(bean: linha, field: "motivo")></td>
42             <td><g:formatNumber number="{linha.valor}" type="currency" /></td>
43             <td><g:formatNumber number="{linha.saldo}" type="currency" /></td>
44           </tr>
45         </g:each>
46       </tbody>
47     </table>
48     <center>
49       <h4>
50         <g:message code="extrato.saveaspdf" />
51         <g:link action="index.pdf">
52           
53         </g:link>
54       </h4>
55     </center>
56   </div>
57 </body>
58 </html>

```

Código 5.23: Visão **extrato/index.gsp**

Em complemento, essa visão adiciona um *link* (linha 51-53) que possibilita salvar o extrato em formato PDF. Conforme pode-se observar que esse *link* invoca a ação **index** do controlador **ExtratoController** passando o parâmetro **format** com o valor igual a **pdf**. Nesse caso, o controlador **ExtratoController** renderiza, utilizando o *plugin* **wkhtmltopdf**, o arquivo **extrato.pdf** baseado no *template* **extrato/_pdf.gsp**.


 Controle Bancário				
Principal	Line Chart	Logout	Cliente	Conta: (Poupança) 261327 [24 - Santander]
Extrato				
Data	Tipo	Motivo	Valor	Saldo
11/06/14	ABERTURA			R\$ 1.000,56
12/06/14	CRÉDITO	Depósito	R\$ 50,00	R\$ 1.050,56
14/06/14	CRÉDITO	Transferência	R\$ 25,00	R\$ 1.075,56
06/07/14	SALDO ATUAL			R\$ 1.075,56
PDF 				
© Departamento de Computação - Universidade Federal de São Carlos				

Figura 5.7: Extrato Bancário em formato HTML

Conforme pode-se observar, o *template* `extrato/_pdf.gsp` (Código 5.24) define uma página HTML que será convertida em um arquivo PDF. Figura 5.8 apresenta um exemplo de um extrato bancário, em formato PDF, gerado pela ação `index()`.

EXTRATO				
Nome: Maria da Silva				
CPF: 018.990.444-50				
Conta: 261327 (Conta Poupança)				
Agência: 24				
Santander - CNPJ: 90.400.888/0001-42				
Data	Tipo	Motivo	Valor	Saldo
11/06/14	ABERTURA			R\$ 1.000,56
12/06/14	CRÉDITO	Depósito	R\$ 50,00	R\$ 1.050,56
14/06/14	CRÉDITO	Transferência	R\$ 25,00	R\$ 1.075,56
06/07/14	SALDO ATUAL			R\$ 1.075,56

Figura 5.8: Extrato Bancário em formato PDF

```

<%@ page import="br.ufscar.dc.dsw.ContaCorrente" %>
<%@ page import="br.ufscar.dc.dsw.ClienteFisico" %>
<hr/>
<center>
<h1>EXTRATO</h1>
</center>
<hr/>
<h4>Nome: ${contaCliente.cliente.nome}</h4>
<h4>
    ${contaCliente.cliente instanceof ClienteFisico ? "CPF: " + contaCliente.cliente.CPF :
    "CNPJ: " + contaCliente.cliente.CNPJ}
</h4>
<hr/>
<h4>Conta: ${contaCliente.conta.numero}
    (${contaCliente.conta instanceof ContaCorrente ? "Conta Corrente" : "Conta Poupança"})</h4>
<h4>Agência: ${contaCliente.conta.agencia.numero}</h4>
<h4>${contaCliente.conta.agencia.banco} – CNPJ: ${contaCliente.conta.agencia.banco.CNPJ}</h4>
<hr/>
<br/>
<br/>
<table style="text-align: left; margin-left: auto; margin-right: auto;" class="pure-table pure-table-bordered"
    border="2">
    <thead>
    <tr>
    <th style="width: 200px; background-color: rgb(204, 255, 255);">Data</th>
    <th style="width: 200px; background-color: rgb(204, 255, 255);">Tipo</th>
    <th style="width: 200px; background-color: rgb(204, 255, 255);">Motivo</th>
    <th style="width: 200px; background-color: rgb(204, 255, 255);">Valor</th>
    <th style="width: 200px; background-color: rgb(204, 255, 255);">Saldo</th>
    </tr>
    </thead>
    <tbody>
    <g:each in="${lines}" status="i" var="linha">
    <tr class="${(i % 2) == 0 ? 'even' : 'odd'}">
    <td><g:formatDate date="${linha.data}" type="date"/></td>
    <td>${fieldValue(bean: linha, field: "tipo")}</td>
    <td>${fieldValue(bean: linha, field: "motivo")}</td>
    <td><g:formatNumber number="${linha.valor}" type="currency"/></td>
    <td><g:formatNumber number="${linha.saldo}" type="currency"/></td>
    </tr>
    </g:each>
    </tbody>
</table>
<br/>
<br/>
<hr/>
<center>
<h6>&copy; Departamento de Computação – Universidade Federal de São Carlos</h6>
<h6><g:formatDate date="${new Date()}" type="datetime" style="LONG"/></h6>
</center>

```

Código 5.24: *Template extrato/_pdf.gsp*

A visão **extrato/chart.gsp** renderiza, utilizando o *plugin google-visualization*, um gráfico de linhas que representa a movimentação financeira das contas bancárias. A implementação dessa visão encontra-se apresentada no Código 5.25.

Conforme pode-se observar essa visão utiliza a tag **<gvisualization:lineCoreChart>** para gerar um gráfico de linhas com os valores (**columns** e **lines**) retornados pela ação **chart()**.

```

<%@ page import="org.grails.plugins.google.visualization.data.Cell;
    org.grails.plugins.google.visualization.util.DateUtil" %>
<html>
<head>
<title>Movimentação Financeira</title>
<meta name="layout" content="main" />
<gvisualization:apiImport/>
<r:require module="pure-all" />
</head>
<body>
<div class="pure-menu pure-menu-open pure-menu-horizontal">
<ul>
<li><a class="home" href="${createLink(uri: '/')}"><g:message code="default.home.label"/></a></li>
<li><g:link controller="extrato"><g:message code="extrato.statement"/></g:link></li>
<li><g:link controller="logout">Logout</g:link></li>
</ul>
</div>
<div id="list-transacao" class="content scaffold-list" role="main">
<div id="linechart"></div>
<gvisualization:lineCoreChart elementId="linechart" width="{800}" height="{300}"
    title="Movimentação Financeira" columns="${columns}" data="${lines}" />
</div>
</body>
</html>

```

Código 5.25: Visão **extrato/chart.gsp**

Figura 5.9 apresenta um exemplo de um gráfico renderizado pela ação **chart.gsp**. Esse gráfico ilustra a movimentação financeira (créditos e débitos) de uma conta bancária.

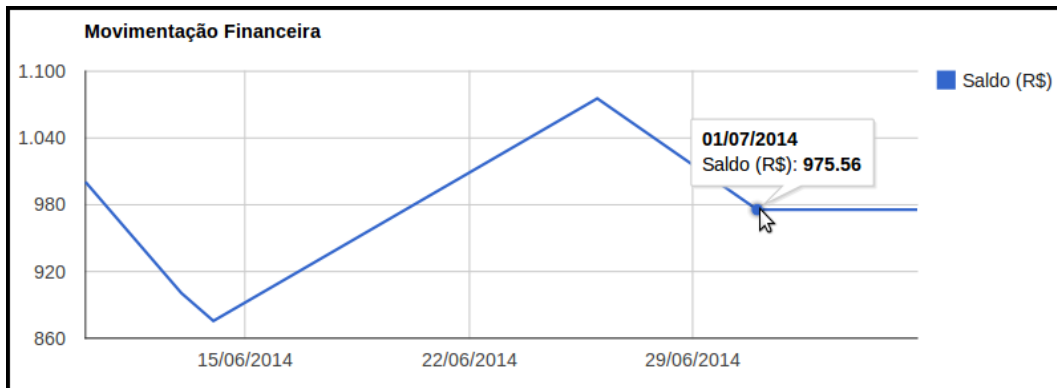


Figura 5.9: Movimentação financeira das contas bancárias

5.8 Internacionalização - Mensagens i18n

Conforme pode-se observar pelas visões apresentadas/discutidas nesse capítulo, a tag **<g:message>** é utilizada para inserir rótulos internacionalizados nessas visões. Dessa forma, é necessário adicionar uma tradução, para cada idioma desejado, desses rótulos nos arquivos presentes no diretório **i18n**.

Figura 5.10 apresenta a tradução desses rótulos que foram inseridos no arquivo **messages_pt_BR.properties** (*Message Bundle* para o Português do Brasil).

```
# Visão cliente/index.gsp
search.label=Busca de Cliente

# Visão main/index.gsp
main.options = Opções

# Visões - Controlador Transacao
transacao.main = Conta Cliente & Caixa Eletrônico
transacao.value = Valor
transacao.data = Data
transacao.quem = Quem
transacao.motivo = Motivo
transacao.tipo = Tipo

# Visões - Controlador Extrato
extrato.statement = Extrato
extrato.chart = Line Chart
extrato.valor = Valor
extrato.saldo = Saldo
extrato.saveaspdf = PDF
```

Figura 5.10: Mensagens internacionalizadas

5.9 Serviço *web* REST

REST não é uma tecnologia em si, pode ser considerada mais como um padrão arquitetural. O padrão arquitetural REST é muito simples e envolve apenas o uso simples de XML ou JSON como meio de comunicação, em conjunto com padrões na definição de URLs que são “representações” do sistema subjacente, e métodos HTTP, como GET, PUT, POST e DELETE. Cada método HTTP é mapeado para um tipo de ação do controlador. Por exemplo, o método GET é mapeado para operações de recuperação, o método PUT é mapeado para operações de criação, o método POST é mapeado para operações de atualização e assim por diante. Neste sentido REST se encaixa muito bem com as operações CRUD.

Essa seção descreve a implementação de um serviço *web* REST que retorna uma lista (em formato JSON ou XML) de agências de um determinado banco (parâmetro **numero**). O controlador **AgenciaController** é a escolha mais óbvia para ser o responsável por prover essa nova funcionalidade. Dessa forma, Código 5.26 apresenta a implementação da ação **list()**, do controlador **AgenciaController**, responsável pela implementação do serviço *web* REST. Por questão de brevidade, apenas serão apresentadas as mudanças realizadas nesse controlador.

```
import grails.converters.JSON
import grails.converters.XML

// Demais imports do controlador AgenciaController

class AgenciaController {

    // Demais ações/atributos/métodos do controlador AgenciaController

    @Secured('IS_AUTHENTICATED_ANONYMOUSLY')
    def list() {

        def banco = Banco.findAllByNumero(params.numero)
        def agencias = Agencia.findAllByBanco(banco)
        def estado = Estado.findBySigla(params.estado)
        def cidade = Cidade.findByNomeAndEstado(params.cidade, estado)

        List<Info> lista = new ArrayList<Info>()

        agencias.each {

            if (!params.cidade || it.endereco.cidade == cidade) {
                Info info = new Info(it.banco.nome, it.numero,
                                     it.nome, it.endereco.toString())

                lista.add(info)
            }
        }

        withFormat {
            json { render lista as JSON }
            xml { render lista as XML }
        }
    }
}
```

Código 5.26: Controlador **AgenciaController**

- A anotação **@Secured('IS_AUTHENTICATED_ANONYMOUSLY')** indica que esse serviço *web* é público e pode ser acessado por qualquer usuário e/ou aplicação.
- É importante salientar que a ação **list()** retorna uma lista de instâncias da classe **Info**. Conforme pode-se observar, a lista de instâncias da classe **Info** é construída através da iteração da lista de agências associadas a um determinado banco (parâmetro **numero**). Opcionalmente, a ação **list()** pode filtrar essa lista visando apenas retornar as agências situadas em uma determinada cidade (parâmetros **cidade** e **estado**);

- A classe Java **Info**²⁶ (Código 5.27) armazena as informações (atributos: **numero**, **nome**, **endereço** e **banco**) relacionadas às agências bancárias.
- Por fim, a ação **list()** constrói a saída ao renderizar a lista de instâncias da classe **Info** (em formato JSON ou XML). A conversão para os formatos JSON ou XML é realizada através da utilização das classes **JSON** e **XML** do pacote **grails.converters**.

```
package br.ufscar.dc.dsw;

public class Info {

    private final int numero;

    private final String nome;

    private final String endereco;

    private final String banco;

    public Info(String banco, int numero, String nome, String endereco) {
        this.banco = banco;
        this.numero = numero;
        this.nome = nome;
        this.endereco = endereco;
    }

    public String getBanco() {
        return banco;
    }

    public String getEndereco() {
        return endereco;
    }

    public String getNome() {
        return nome;
    }

    public int getNumero() {
        return numero;
    }

}
```

Código 5.27: Classe Java **Info**

Figura 5.11 ilustra o acesso desse serviço *web* através da URL `http://localhost:8080/ControleBancarioV4/agencia/list.json?numero=1`. É importante salientar que essa URL define a ação a ser acessada: **list()** do controlador **AgenciaController**. Adicionalmente, essa URL define o valor do parâmetro **numero** e o formato da resposta (formato JSON).

Figura 5.11: Serviço *web* REST: Formato JSON

²⁶Arquivo: `src/br/ufscar/dc/dsw/Info.java`

Analogamente, Figura 5.12 ilustra o acesso desse serviço *web* através da URL `http://localhost:8080/ControleBancarioV4/agencia/list.xml?numero=1`. Novamente essa URL define a ação a ser acessada, o valor do parâmetro **numero** e o formato da resposta (formato XML).

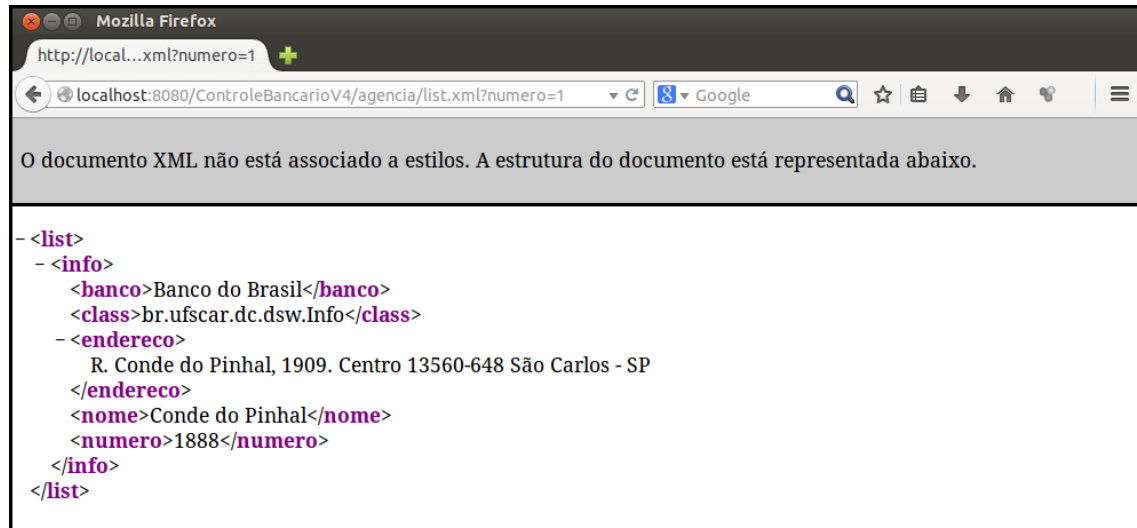


Figura 5.12: Serviço *web* REST: Formato XML

5.10 Considerações finais

Esse capítulo apresentou a quarta versão da implementação da aplicação **ControleBancario**. O código-fonte dessa aplicação (`ControleBancarioV4.zip`) encontra-se disponível no *Moodle* do curso, localizado no endereço: `http://moodle.latosensu.dc.ufscar.br`. Seguindo os passos do tutorial apresentado obtém-se esse mesmo código da aplicação **ControleBancario**.



6 — Considerações finais

Este material apresentou as principais funcionalidades do framework Grails através da descrição dos aspectos relacionados ao desenvolvimento da aplicação **ControleBancario**.

Para finalizar esse material, esse capítulo apresenta o *overview* de mais algumas funcionalidades presentes em Grails que não foram abordadas nos capítulos anteriores.

6.1 Automação de testes

Essa seção apresenta algumas características de Grails que facilitam a automação de testes. Relembrando: testes automatizados te dão segurança no momento da manutenção. No entanto, antes do início da discussão sobre tais características, é fundamental relembrar a diferença entre testes unitários e de integração.

Testes unitários levam em consideração a unidade a ser verificada isoladamente. Não há conexões com bancos de dados ou qualquer outro tipo de componente: a unidade deve ser vista como um elemento isolado (não se comunica com ninguém).

Testes de integração, por sua vez, levam em consideração, como o próprio nome já diz a integração da unidade a ser testada com componentes externos, como por exemplo, bancos de dados ou outros serviços de natureza diversa. Testes de integração são, portanto, mais caros do ponto de vista computacional, visto que é necessário iniciar a aplicação para que estes possam ser executados.

Todos os testes se encontram no diretório **test/unit** (ou **integration**) presente na raiz do projeto Grails. Toda vez que é criada uma classe de domínio, controlador (e outros artefatos), testes automaticamente são incluídos no diretório **test/unit**. Da mesma forma que o GORM é fortemente baseado no framework *hibernate*, o arcabouço de testes presente em Grails é fortemente baseado no *Junit*²⁷.

Há três maneiras de se criar estes testes: (1) O Grails os cria automaticamente; (2) A classe de teste é criada manualmente pelos desenvolvedores; e (3) A classe de teste é criada usando o comando **grails create-unit-test**.

²⁷<http://junit.org/>

Assim como diversos aspectos do Grails, aqui é necessário ater-se à algumas convenções. Toda classe de teste possui o sufixo **Tests** em seu nome. Sendo assim, os testes unitários para a classe de domínio **Usuario**, por exemplo, ficariam em **test/unit/UsuarioTests.groovy**.

O comando **grails create-unit-test** ou **grails create-integration-test** deve receber o nome do teste unitário ou de integração a ser gerado. Não é necessário incluir o **Tests** no final do arquivo, Grails o inclui automaticamente.

Testando classes de domínio. Ao lidar com linguagens dinâmicas como Groovy é frequente a necessidade de lidar com o seguinte problema: como testar uma classe que contém métodos e atributos que só serão injetados em tempo de execução? Funções como **save()**, **validate()** ou **constraints** só funcionam após injetados pelo framework.

Uma possibilidade é escrever testes de integração. O problema é que leva tempo até a aplicação ser iniciada – o que provavelmente irá reduzir a sua produtividade. O ideal é executar testes unitários, que por sua própria natureza são ordens de magnitude mais rápidas. A solução para o problema é usar *mock objects*.

Para ilustrar, tenha como base esta classe de domínio:

```
class Usuario {
    String nome
    String login
    static constraints = {
        nome(nullable:false, blank:false, maxSize:128, unique:true)
        login(nullable:false, blank:false, maxSize:16, unique:true)
    }
}
```

O teste unitário encontra-se na classe a seguir:

```
import grails.test.*
class UsuarioTests extends GrailsUnitTestCase {
    protected void setUp() { super.setUp() }
    protected void tearDown() { super.tearDown() }
    void testConstraints() {
        mockDomain Usuario
        def usuario = new Usuario()
        assertFalse usuario.validate() // usuario nao validado pois
                                        // nao incluiu campos obrigatorios
        def usuarioOK = new Usuario(nome:"Maria", login: "maria")
        assertTrue usuarioOK.validate()
    }
}
```

Vale a pena salientar o seguinte ponto: mesmo se tratando de um teste unitário, o teste exercita métodos que só existem em tempo de execução: no caso, o **validate**. Para isto, foi utilizado o método **mockDomain**, herdado de **GrailsUnitTestCase**. Este injeta na classe de domínio todos os métodos que uma classe deste tipo deve ter – como exemplos, os métodos de validação, **save()**, **delete()**, etc. Assim é possível testar facilmente a validação.

No caso de testes de integração, obviamente você não precisa do método **mockDomain**, pois as classes já estão prontas.

Testes unitários para controladores. É muito comum encontrarmos projetos nos quais apenas classes de domínio são testadas. Para testar seus controladores, você deve criar um teste tal como faria normalmente. A diferença é que este teste não estenderá a classe **GrailsUnitTestCase**, e sim **ControllerUnitTestCase**.

Executando os testes. Com a aplicação funcionando, o próximo passo é executar seus testes. Para isto, deve-se usar o comando **grails test-app**, que executará todos os seus testes.

Para executar apenas alguns testes, basta passar como parâmetro os nomes dos testes excluindo o sufixo **Tests**, como exemplo **grails test-app Usuario**.

Para executar apenas testes unitários, execute **grails test-app unit** e para executar apenas os testes de integração, execute **grails test-app integration**.

Executados os seus testes, será criado o diretório **target/test-reports** em seu projeto, contendo o relatório de execução dos testes.

6.2 Estudos complementares

Para estudos complementares sobre o framework Grails que foi abordado nesse material, o leitor interessado pode consultar as seguintes referências:

- BROWN, J.S.; ROCHER, G. *The Definitive Guide to Grails 2*. New York, USA: Apress, 2013.
- SMITH, G.; LEDBROOK, P. *Grails in Action*. Greenwich, CT, USA: Manning Publications Co., 2009.
- JUDD, C.M.; NUSAIRAT, J.F.; SHINGLER, J. *Groovy and Grails – From Novice to Professional*. New York, USA: Apress, 2008.
- KÖENIG, D. et al. *Groovy in Action*. Greenwich, CT, USA: Manning Publications Co., 2007.
- VISHAL, L.; JUDD, C.M.; NUSAIRAT, J.F.; SHINGLER, J. *Beginning Groovy, Grails and Griffon*. New York, USA: Apress, 2013.



Referências Bibliográficas

- [1] BEDER, D. M. *Engenharia Web : Uma Abordagem Sistemática para o Desenvolvimento de Aplicações Web*. São Carlos: EdUFSCar, 2012. (Coleção UAB - UFSCar). ISBN 978-85-7600-290-1.
- [2] PRESSMAN, R.; LOWE, D. *Engenharia Web*. Rio de Janeiro: LTC, 2009.
- [3] BECK, K.; ANDRES, C. *Extreme Programming Explained: Embrace Change*. 2a. ed. Boston: Addison-Wesley, 2009.
- [4] SCHWABER, K. *Agile Project Management with SCRUM*. Washington: Microsoft Press, 2004.
- [5] KRASNER, G.; POPE, S. A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, v. 1, n. 3, p. 26–49, 1988.
- [6] PRESSMAN, R. *Engenharia de Software: Uma Abordagem Profissional*. 7a. ed. São Paulo: Bookman, 2011.
- [7] SMITH, G.; LEDBROOK, P. *Grails in Action*. Greenwich, CT, USA: Manning Publications Co., 2009.
- [8] KÖENIG, D. et al. *Groovy in Action*. Greenwich, CT, USA: Manning Publications Co., 2007.



Índice Remissivo

A

Ambiente de Desenvolvimento 6

B

Banco de Dados 7, 12

Bootstrap 35

C

Comando

grails create-integration-test 118

grails create-unit-test 117

grails test-app 119

Comandos

grails create-controller 29

grails create-app 9

grails compile 44

grails create-controller 54, 56, 58

grails create-domain-class 14

grails generate-all 30, 54

grails generate-views 96

grails install-plugin 11

grails install-templates 48

grails run-app 39, 66

grails s2-quickstart 45

Convenção

versus Configuração 5, 10, 31

Nomenclatura URL 32

E

Engenharia Web 3

G

GORM 7, 16

Relacionamento de herança 26

I

Internacionalização - I18n 46, 113

M

Modelo-Visão-Controlador (MVC) 4

Controlador 29, 32

Modelo 14

Visão 29, 34

P

Plugins

br-validation 11

google-visualization 89

pure-css 89, 92

rest	70
richui	89, 102
spring-security	43, 45, 49, 52
wkhtmltopdf	89

S

Scaffolding	29
Dinâmico	29
Estático	30
Serviços <i>web</i> REST	114
Cliente	86

U

URL	
Mapeamento	57

V

Validação de Dados	15
--------------------------	----