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 1

1.1 Modelo-Visão-Controlador (MVC) 2

1.2 Grails 3

1.2.1 Ambiente de Desenvolvimento 4

1.3 Considerações finais 6

2 Controle Bancário: Versão 1 7

2.1 Configuração da aplicação 9

2.1.1 Instalação de *plugins* e definição de dependências 9

2.1.2 Configuração do banco de dados 10

2.2 Implementando as primeiras funcionalidades 11

2.2.1 Classe de Domínio: Estado 12

2.2.2 Classe de Domínio: Cidade 14

2.2.3 Classe de Domínio: Endereco 15

2.2.4 Classe de Domínio: Banco 16

2.2.5 Classe de Domínio: Agencia 17

2.2.6 Classe de Domínio: Gerente 18

2.2.7 Classe de Domínio: CaixaEletronico 19

2.2.8 Classe de Domínio: Transacao 20

2.2.9 Classe de Domínio: ContaCliente 21

2.2.10 Classe de Domínio: Conta 22

2.2.11 Classe de Domínio: ContaCorrente 23

2.2.12 Classe de Domínio: ContaPoupanca 23

2.2.13 Classe de Domínio: Cliente 25

2.2.14 Classe de Domínio: ClienteFisico 26

2.2.15 Classe de Domínio: ClienteJuridico 26

2.3 Scaffolding **27**
2.3.1 Scaffolding Dinâmico 27
2.3.2 Scaffolding Estático 28
2.3.3 Convenção na nomenclatura de URLs 30
2.3.4 Controlador: TransacaoController 30
2.3.5 *Groovy Server Pages (GSPs)* 32
2.4 Executando a aplicação **33**
2.5 Considerações finais **39**

3 Controle Bancário: Versão 2 41
3.1 Configuração da aplicação **41**
3.2 Controle de Acesso **43**
3.2.1 Classes de Domínio: Cliente, ClienteFisico, ClienteJuridico e Gerente 44
3.3 Internacionalização **44**
3.4 Personalização dos templates utilizados no scaffolding **46**
3.4.1 Template: Controller.groovy 47
3.4.2 Template: create.gsp 47
3.4.3 Template: index.gsp 50
3.4.4 Template: show.gsp 51
3.5 Controladores e Visões **52**
3.5.1 Controlador: ContaController 52
3.5.2 Visão: conta/index.gsp 53
3.5.3 Controlador: ClienteController 54
3.5.4 Visão: cliente/index.gsp 54
3.5.5 Mapeamento URL 55
3.5.6 Controlador: MainController 56
3.5.7 Visão: main/index.gsp 56
3.5.8 Controladores: últimas alterações relacionadas ao controle de acesso . . . 57
3.6 Melhorando o leiaute da aplicação: biblioteca de marca **58**
3.7 Executando a aplicação **60**
3.8 Considerações finais **65**

4 Controle Bancário: Versão 3 67
4.1 Configuração da aplicação **68**
4.2 Atribuição de papéis **69**
4.2.1 Classe de domínio Gerente X Papel **ROLE_GERENTE** 69
4.2.2 Classe de domínio Cliente X Papel **ROLE_CLIENTE** 70
4.3 Controle de acesso: Transações **71**
4.3.1 Controlador: MainController 71
4.3.2 Biblioteca de marcas: LoginTagLib 72
4.3.3 Controlador: SelecionaContaController 73
4.3.4 Visão: selecionaConta/index.gsp 73
4.3.5 Classe de Domínio: Transacao 74
4.3.6 Controlador: TransacaoController 74

4.3.7 Template transacao/_fields.gsp 76
4.3.8 Executando a aplicação 77

4.4 Controle de acesso: Contas 78
4.4.1 Controlador: ContaCorrenteController 78
4.4.2 Template contaCorrente/_fields.gsp 79
4.4.3 Controlador: ContaClienteController 80
4.4.4 Template contaCliente/_fields.gsp 82
4.4.5 Controlador: ContaController 83
4.4.6 Executando a aplicação 83

4.5 Preenchimento automático de endereços 85
4.5.1 Template endereco/_address.gsp 85
4.5.2 Controlador: EnderecoController 87

4.6 Considerações finais 88

5 Considerações finais 89

5.1 Serviços Web 89
5.1.1 REST 89

5.2 Automação de testes 90
5.2.1 Grails: TDD e BDD 91
5.2.2 BDD: um exemplo prático 92

5.3 Estudos complementares 94

Índice Remissivo 101

Lista de Códigos

2.1 **build.gradle** (configuração de *plugins*) . . . 9
2.2 **application.yml (configuração banco de dados)** . . . 10
2.3 Classe de domínio **Estado** . . . 13
2.4 Classe de domínio **Cidade** . . . 14
2.5 Classe de domínio **Endereco** . . . 15
2.6 Classe de domínio **Banco** . . . 16
2.7 Classe de domínio **Agencia** . . . 17
2.8 Classe de domínio **Gerente** . . . 18
2.9 Classe de domínio **CaixaEletronico** . . . 19
2.10 Classe de domínio **Transacao** . . . 20
2.11 Classe de domínio **ContaCliente** . . . 21
2.12 Classe de domínio **Conta** . . . 22
2.13 Classe de domínio **ContaCorrente** . . . 23
2.14 Classe de domínio **ContaPoupanca** . . . 23
2.15 Classe de domínio **Cliente** . . . 25
2.16 Classe de domínio **ClienteFisico** . . . 26
2.17 Classe de domínio **ClienteJuridico** . . . 26
2.18 Controlador **TransacaoController (1)** . . . 28
2.19 Controlador **TransacaoController** . . . 31
2.20 Visão **transacao/index.gsp** . . . 32
2.21 **BootStrap.groovy (1)** . . . 33
2.22 **BootStrap.groovy (2)** . . . 34
2.23 **BootStrap.groovy (3)** . . . 35
2.24 **BootStrap.groovy (4)** . . . 36
2.25 **BootStrap.groovy (5)** . . . 37

3.1 **BuildConfig.groovy** . . . 42
3.2 Usuários: Clientes e Gerentes . . . 44
3.3 *Template* **Controller.groovy** . . . 48
3.4 *Template* **create.gsp** . . . 49
3.5 *Template* **index.gsp** . . . 50

3.6 *Template* **show.gsp** 51
3.7 Controlador **ContaController** 52
3.8 Visão **conta/index.gsp** 53
3.9 Controlador **ClienteController** 54
3.10 Visão **cliente/index.gsp** 55
3.11 **URLMappings.groovy** 55
3.12 Controlador **MainController** 56
3.13 Visão **main/index.gsp** 56
3.14 Controladores - últimas alterações relacionadas ao Controle de Acesso 57
3.15 Biblioteca de marca **LoginTagLib** 58
3.16 *Template* **_footer.gsp** 58
3.17 *Template* **_header.gsp** 58
3.18 Leiaute padrão **grails-app/views/layouts/main.gsp** 59
3.19 Arquivo **main.css** 59
3.20 **BootStrap.groovy (1)** 60
3.21 **BootStrap.groovy (2)** 61
3.22 **BootStrap.groovy (3)** 62
3.23 **BootStrap.groovy (4)** 63
3.24 **BootStrap.groovy (5)** 64

4.1 **build.gradle** 68
4.2 Controlador **Gerente** (ação **save()**) 69
4.3 Controlador **ClienteFisico** (ação **save()**) 70
4.4 Controlador **ClienteJuridico** (ação **save()**) 70
4.5 Controlador **MainController** 71
4.6 Biblioteca de marca **LoginTagLib** 72
4.7 Controlador **SelecionaContaController** 73
4.8 Visão **selecionaConta/index.gsp** 73
4.9 Classe de domínio **Transacao** 74
4.10 Controlador **TransacaoController** 75
4.11 *Template* **transacao/_fields.gsp** 76
4.12 **transacao/create.gsp** 77
4.13 Controlador **ContaCorrenteController** 79
4.14 *Template* **contaCorrente/_fields.gsp** 79
4.15 **contaCorrente/create.gsp** 80
4.16 Controlador **ContaClienteController** 81
4.17 *Template* **contaCliente/_fields.gsp** 82
4.18 **contaCliente/create.gsp** 82
4.19 Controlador **ContaController** 83
4.20 *Template* **endereco/_address.gsp** 85
4.21 **endereco/create.gsp** 86
4.22 Controlador **EnderecoController** 87

5.1 Classe de domínio – recurso REST 90

Lista de Figuras

1.1 Padrão arquitetural MVC 2
1.2 Verificação da instalação do Java 5
1.3 Console do SGBD H2 6

2.1 Criação do Projeto ControleBancario. 7
2.2 Diagrama de classes UML 11
2.3 Criação da Classe de Domínio Estado. 12
2.4 GORM: Mapeamento de Hierarquia 24
2.5 Criação do controlador (*scaffolding* dinâmico). 27
2.6 Criação do controlador e das visões (*scaffolding* estático). 28
2.7 Scaffolding estático das classes de domínio. 29
2.8 Convenção na nomenclatura de URLs. 30
2.9 Execução da aplicação **ControleBancario** 37
2.10 Lista de transações bancárias 38
2.11 Criação de uma nova transação bancária 38
2.12 Criação de uma transação bancária com valores inválidos 39

3.1 I18n (arquivos de propriedades). 45
3.2 Messagens I18n para a classe de Domínio Usuario 46
3.3 Máscaras de entrada: CEP, CNPJ e CPF 47
3.4 Visão main/index.gsp: diferentes visões 57
3.5 **ControleBancario**: Página de *Login* 65

4.1 Visão **main/index.gsp**: **ROLE_CLIENTE** 77
4.2 Lista de transações de uma conta do usuário *logado* 78
4.3 Visão **main/index.gsp**: **ROLE_GERENTE** 83
4.4 Lista de clientes 84
4.5 Lista de contas da agência do usuário *logado* 84
4.6 Visão **endereco/create.gsp**: Preenchimento automático de atributos 88

5.1 Especificação de testes para o controlador EnderecoController 92
5.2 Catálogo de Mídias. 95

Lista de Tabelas

2.1 Projeto Grails: *Overview* dos Diretórios. 8
2.2 Regras de restrições. 13

3.1 Classes de domínio: geração dos Controladores e Visões. 52

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 5), conscientizando os desenvolvedores de sua importância.
- **Simplicidade é essencial**: Grails visa proporcionar simplicidade. Ou seja, Grails tem conceitos complexos como ORM (Object-Relational Mapping[1]), 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 importancia 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.

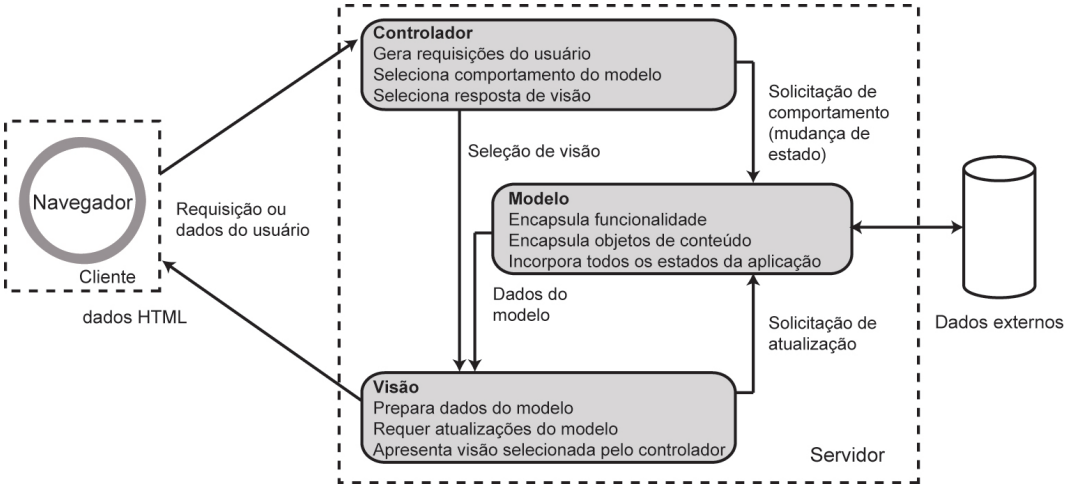


Figura 1.1: Padrão arquitetural MVC

[1] Mapeamento Objeto-Relacional

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;
- 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 (Hibernate[2], Spring[3], 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

[2] `http://www.hibernate.org/`
[3] `http://springsource.org/`

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.

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.
- Finalmente, o capítulo 5 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-requisitos[4] 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).

[4]Mais detalhes podem ser encontrados em: `http://www.itexto.net/devkico/?p=40`.

```
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

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 IDE[5] 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 H2[6], 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 MySQL[7], PostgreSQL[8] e Oracle[9]. 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.

[5] Em 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`

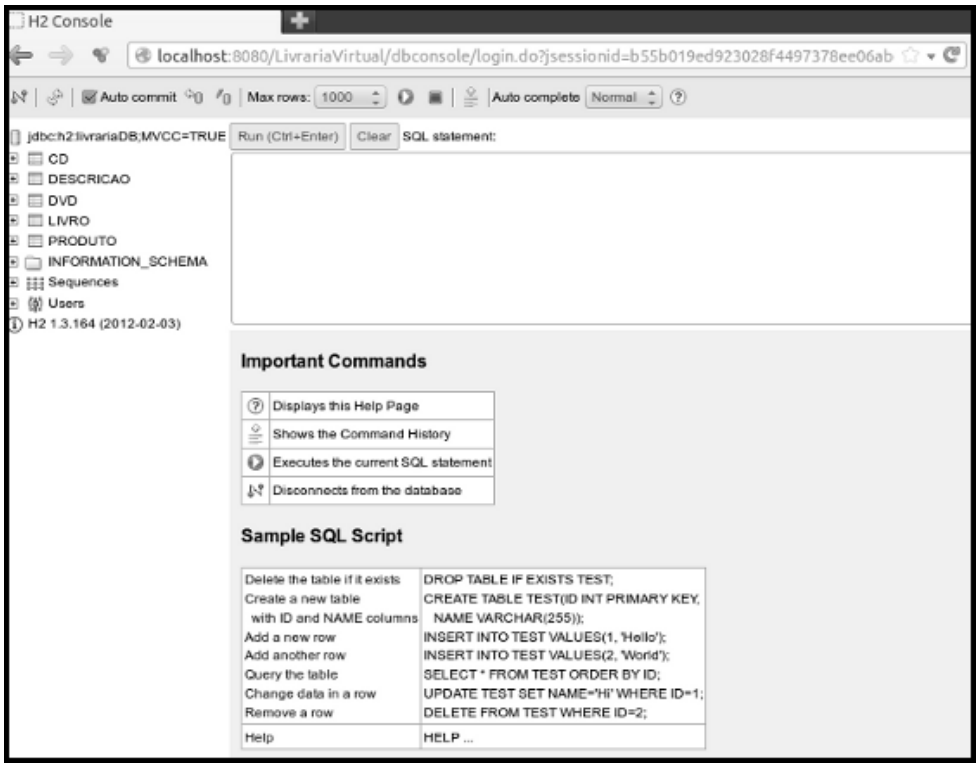


Figura 1.3: Console do SGBD H2

Mecanismo de *build*. Desde sua primeira versão, Grails sempre usou como mecanismo de *build*, a ferramenta Gant[10], 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 Gradle[11] 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** $\Longrightarrow$ **Grails**.
- Em nome do projeto, digite **ControleBancario** e clique em **Finish** (Figura 2.1). O Intellij IDE executa o comando **grails create-app**.

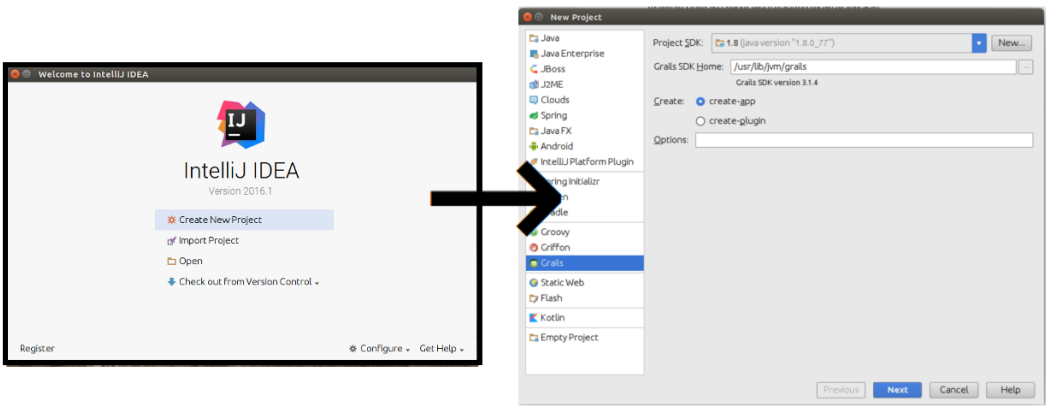


Figura 2.1: Criação do Projeto ControleBancario.

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

[12]Para 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.
grail-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 (*taglibs*) 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 (*images*, *javascript* e *stylesheets*) onde se encontram os *assets* 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.graddle** 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.

```yaml
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**.

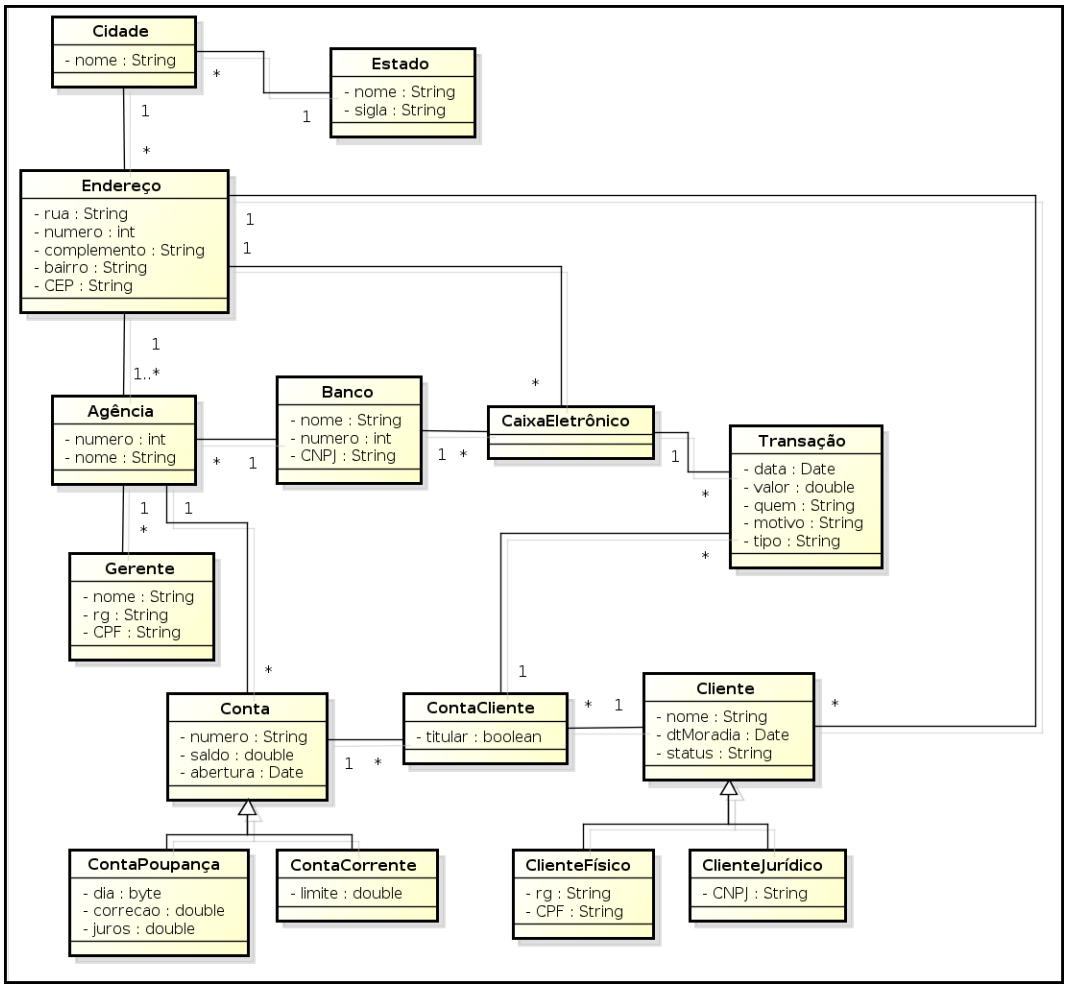


Figura 2.2: Diagrama de classes UML

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 Domínio[13] 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** $\Longrightarrow$ **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.

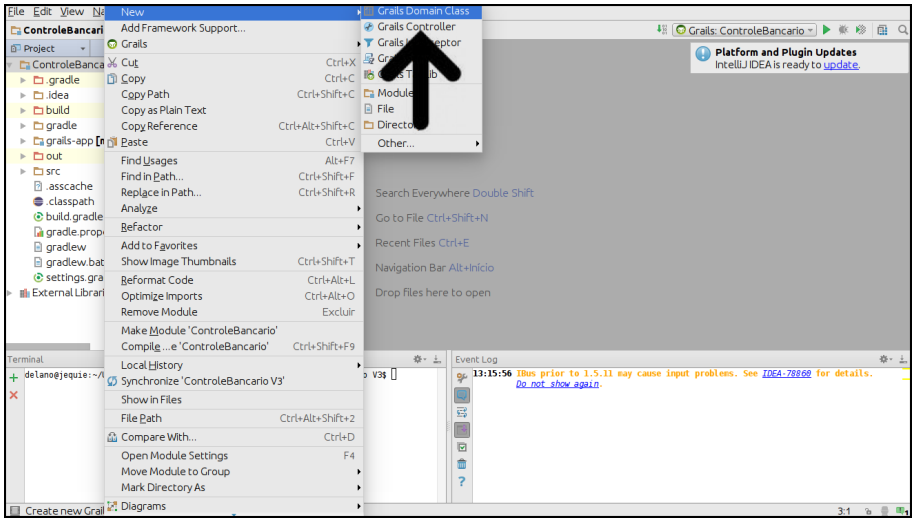


Figura 2.3: Criação da Classe de Domínio Estado.

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.

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

```
package br.ufscar.dc.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(estado.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**;
  - ◇ **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 **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 **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 **contaCliente** 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
    }
}
```

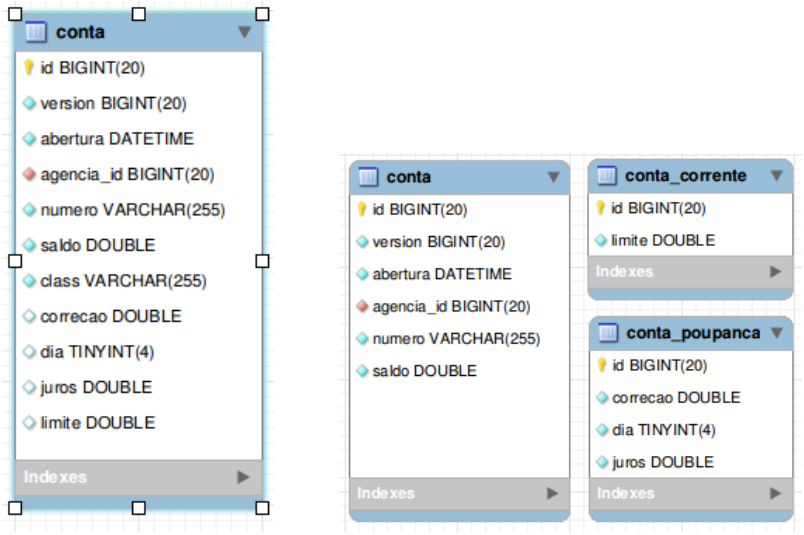
Código 2.14: Classe de domínio **ContaPoupanca**

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 contem 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.



(a) *table-per-hierarchy* (b) *table-per-class*

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 subclasse: **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 **cnpj: 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, cnpj: 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** $\Longrightarrow$ **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.

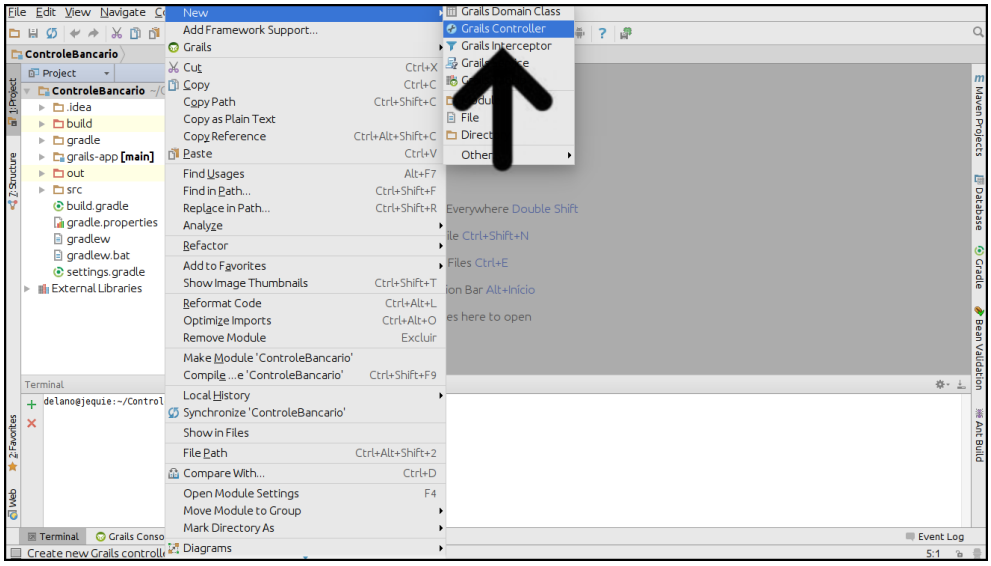


Figura 2.5: Criação do controlador (*scaffolding* dinâmico).

```
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 corresponde conjunto de *Groovy Server Pages (GSPs)* no diretório **grails-app/views/transacao** (Figura 2.6).

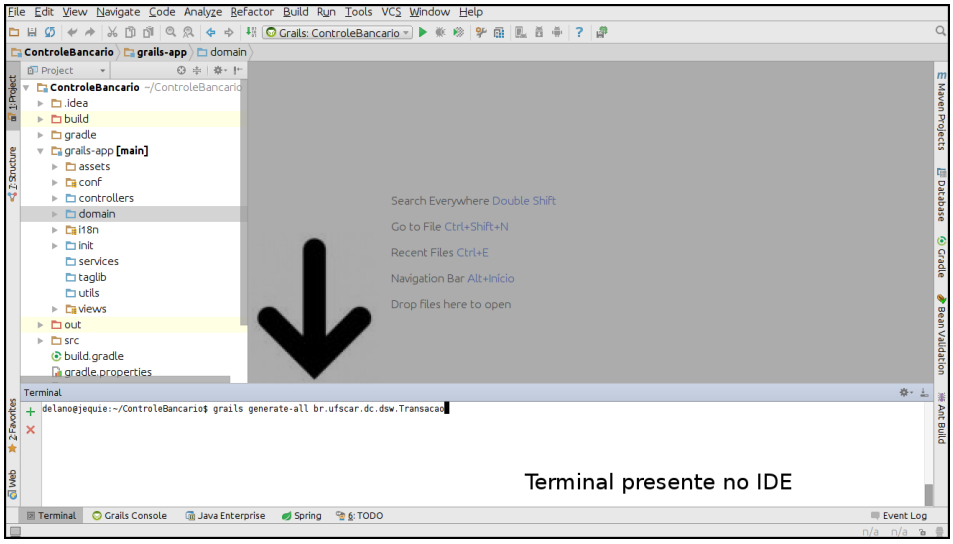


Figura 2.6: Criação do controlador e das visões (*scaffolding* estático).

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.

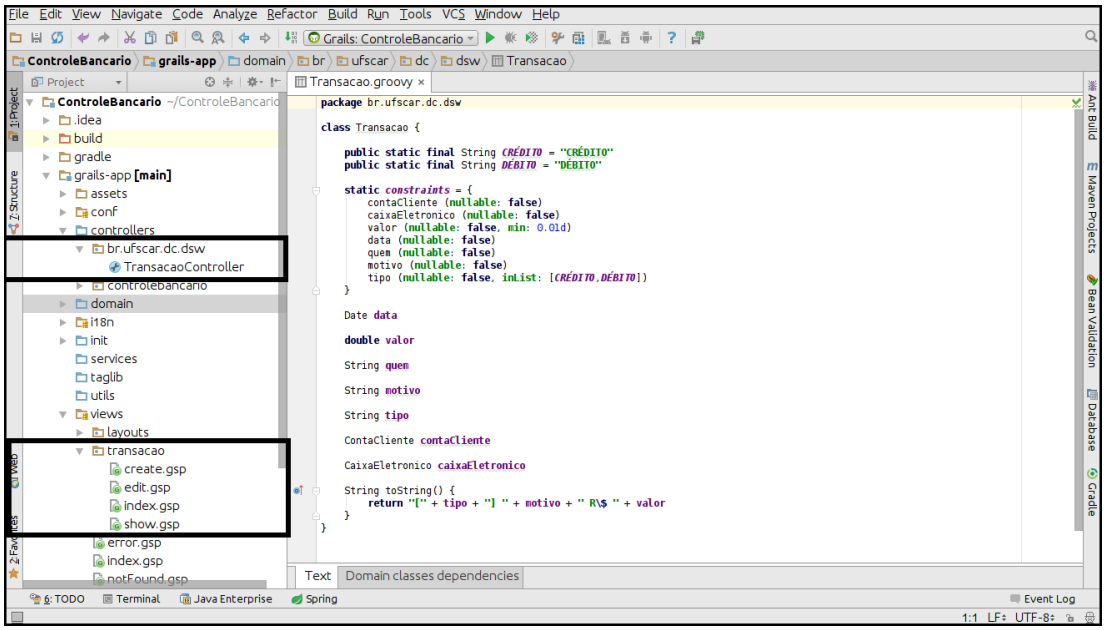


Figura 2.7: Scaffolding estático das classes de domínio.

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 hospedada na porta 8080 do servidor* **localhost**".

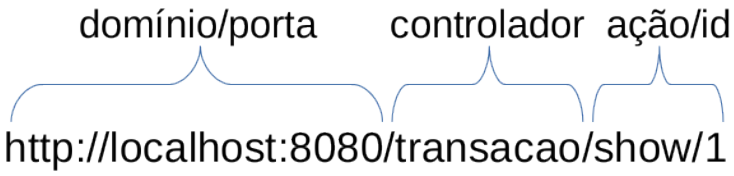


Figura 2.8: Convenção na nomenclatura de URLs.

A regra principal de roteamento 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 corriga 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

```
@Transactional(readOnly = true)
class TransacaoController {

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

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

    def show(Transacao transacao) {
        respond transacao
    }

    def create() {
        respond new Transacao(params)
    }

    @Transactional
    def save(Transacao transacao) {
        if (transacao == null) {
            transactionStatus.setRollbackOnly()
            notFound()
            return
        }
        if (transacao.hasErrors()) {
            transactionStatus.setRollbackOnly()
            respond transacao.errors, view:'create'
            return
        }
        transacao.save flush:true
        request.withFormat {
            form multipartForm {
                flash.message = message(code: 'default.created.message', args: [message(code: 'transacao.label',
                                        default: 'Transacao'), transacao.id])
                redirect transacao
            }
            '*' { respond transacao, [status: CREATED] }
        }
    }

    def edit(Transacao transacao) {
        respond transacao
    }

    @Transactional
    def update(Transacao transacao) {
        if (transacao == null) {
            transactionStatus.setRollbackOnly()
            notFound()
            return
        }
        if (transacao.hasErrors()) {
            transactionStatus.setRollbackOnly()
            respond transacao.errors, view:'edit'
            return
        }
        transacao.save flush:true
        request.withFormat {
            form multipartForm {
                flash.message = message(code: 'default.updated.message', args: [message(code: 'transacao.label',
                                        default: 'Transacao'), transacao.id])
                redirect transacao
            }
            '*'{ respond transacao, [status: OK] }
        }
    }

    @Transactional
    def delete(Transacao transacao) {

        if (transacao == null) {
            transactionStatus.setRollbackOnly()
            notFound()
            return
        }
        transacao.delete flush:true
        request.withFormat {
            form multipartForm {
                flash.message = message(code: 'default.deleted.message', args: [message(code: 'transacao.label',
                                        default: 'Transacao'), transacao.id])
                redirect action:"index", method:"GET"
            }
            '*'{ render status: NO_CONTENT }
        }
    }

    protected void notFound() {
        request.withFormat {
            form multipartForm {
                flash.message = message(code: 'default.not.found.message', args: [message(code: 'transacao.label',
                                        default: 'Transacao'), params.id])
                redirect action: "index", method: "GET"
            }
            '*'{ render status: NOT_FOUND }
        }
    }
}
```

Código 2.19: Controlador **TransacaoController**

usuário corriga os erros encontrados na validação; e

- 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 contêm 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>
            <f: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

[14] `http://www.oracle.com/technetwork/java/index-jsp-135995.html`

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.

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.Agencia
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 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 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: agencia1,
            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 depositos - 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).

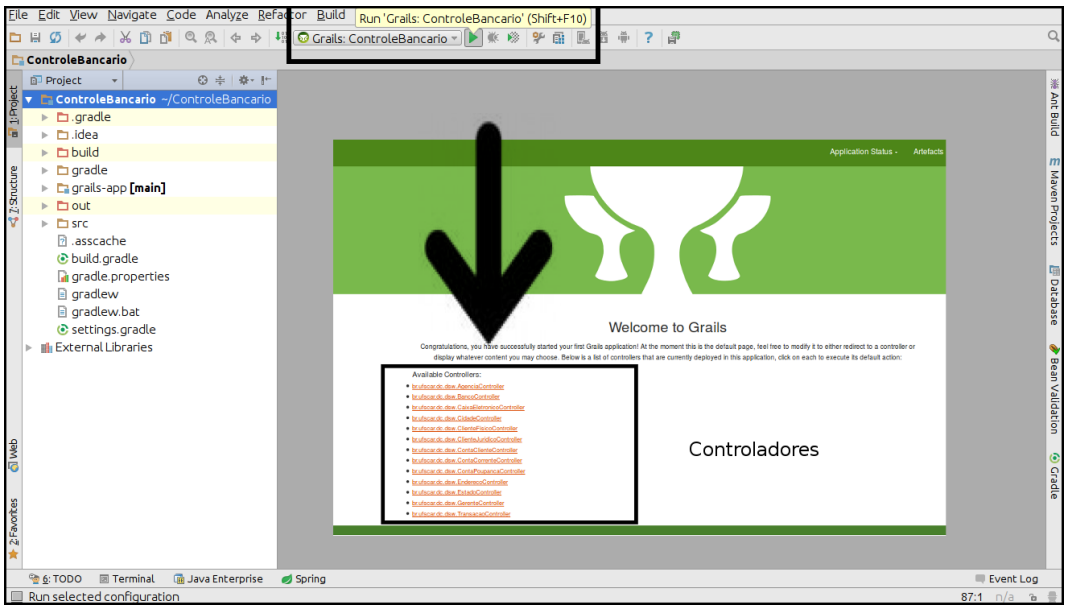


Figura 2.9: Execução da aplicação **ControleBancario**

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

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).

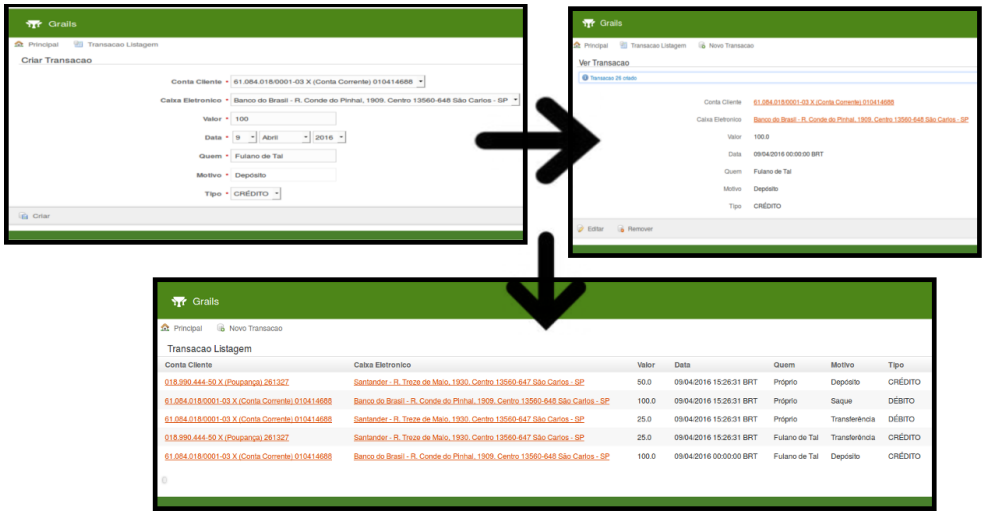


Figura 2.11: Criação de uma nova transação bancária

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). Verfica-se que a transação não foi criada, pois as validações de dados foram violadas.

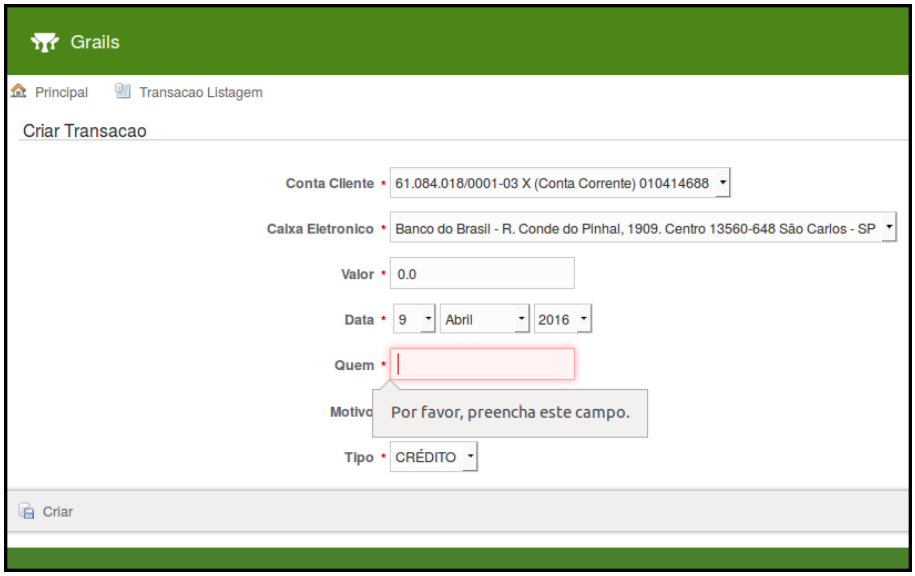


Figura 2.12: Criação de uma transação bancária com valores inválidos

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**.

[15] URL: `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 marca[16] 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.

[16] Em 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.

```
buildscript {
    ext {
        grailsVersion = project.grailsVersion
    }
    repositories {
        mavenLocal()
        maven { url "https://repo.grails.org/grails/core" }
    }
    dependencies {
        classpath "org.grails:grails-gradle-plugin:$grailsVersion"
        classpath "com.bertramlabs.plugins:asset-pipeline-gradle:2.5.0"
        classpath "org.grails.plugins:hibernate4:5.0.2"
    }
}
version "0.1"
group "controlebancario"
apply plugin:"eclipse"
apply plugin:"idea"
apply plugin:"war"
apply plugin:"org.grails.grails-web"
apply plugin:"org.grails.grails-gsp"
apply plugin:"asset-pipeline"
ext {
    grailsVersion = project.grailsVersion
    gradleWrapperVersion = project.gradleWrapperVersion
}
repositories {
    mavenLocal()
    maven { url "https://repo.grails.org/grails/core" }
}
dependencyManagement {
    imports {
        mavenBom "org.grails:grails-bom:$grailsVersion"
    }
    applyMavenExclusions false
}
dependencies {
    compile "org.springframework.boot:spring-boot-starter-logging"
    compile "org.springframework.boot:spring-boot-autoconfigure"
    compile "org.grails:grails-core"
    compile "org.springframework.boot:spring-boot-starter-actuator"
    compile "org.springframework.boot:spring-boot-starter-tomcat"
    compile "org.grails:grails-dependencies"
    compile "org.grails:grails-web-boot"
    compile "org.grails.plugins:cache"
    compile "org.grails.plugins:scaffolding"
    compile "org.grails.plugins:hibernate4"
    compile "org.hibernate:hibernate-ehcache"
    compile "org.grails.plugins:br-validation:0.3"
    compile "org.grails.plugins:spring-security-core:3.0.4"
    console "org.grails:grails-console"
    profile "org.grails.profiles:web:3.1.4"
    runtime "org.grails.plugins:asset-pipeline"
    runtime "com.h2database:h2"
    runtime "org.postgresql:postgresql:9.3-1101-jdbc41"
    testCompile "org.grails:grails-plugin-testing"
    testCompile "org.grails.plugins:geb"
    testRuntime "org.seleniumhq.selenium:selenium-htmlunit-driver:2.47.1"
    testRuntime "net.sourceforge.htmlunit:htmlunit:2.18"
}
task wrapper(type: Wrapper) {
    gradleVersion = gradleWrapperVersion
}
assets {
    minifyJs = true
    minifyCss = true
}
```

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 Security[19] 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 papeis 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).

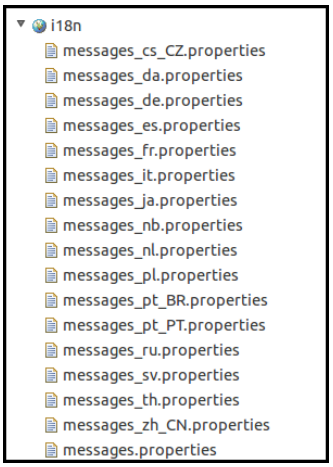


Figura 3.1: I18n (arquivos de propriedades).

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 incluido 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: Messagens I18n para a classe de Domínio Usuario

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.

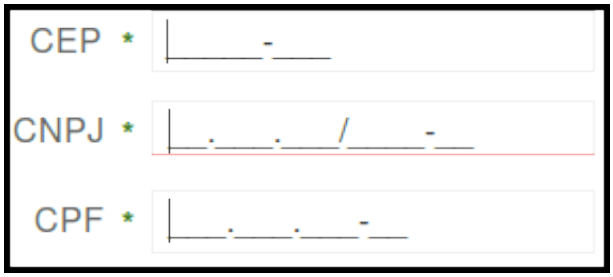


Figura 3.3: Máscaras de entrada: CEP, CNPJ e CPF

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 digito (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 digitos 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/`

```
<%=packageName ? "package ${packageName}" : ''%>

import static org.springframework.http.HttpStatus.*
import grails.transaction.Transactional
import org.springframework.security.access.annotation.Secured
@Transactional(readOnly = true)
@Secured ('ROLE_ADMIN')
class ${className}Controller {
    static allowedMethods = [save: "POST", update: "PUT", delete: "DELETE"]
    def index(Integer max) {
        params.max = Math.min(max ?: 10, 100)
        respond ${className}.list(params), model:[${propertyName}Count: ${className}.count()]
    }
    @Secured (['ROLE_ADMIN', 'ROLE_CLIENTE' , 'ROLE_GERENTE'])
    def show(${className} ${propertyName}) {
        respond ${propertyName}
    }
    def create() {
        respond new ${className}(params)
    }
    @Transactional
    def save(${className} ${propertyName}) {
        if (${propertyName} == null) {
            transactionStatus.setRollbackOnly()
            notFound()
            return
        }
        if (${propertyName}.hasErrors()) {
            transactionStatus.setRollbackOnly()
            respond ${propertyName}.errors, view:'create'
            return
        }
        ${propertyName}.save flush:true
        request.withFormat {
            form multipartForm {
            flash.message = message(code: 'default.created.message', args: [message(code: '${propertyName}.label',
                                    default: '${className}'), ${propertyName}.id])
                redirect ${propertyName}
            }
            '*' { respond ${propertyName}, [status: CREATED] }
        }
    }
    def edit(${className} ${propertyName}) {
        respond ${propertyName}
    }
    @Transactional
    def update(${className} ${propertyName}) {
        if (${propertyName} == null) {
            transactionStatus.setRollbackOnly()
            notFound()
            return
        }
        if (${propertyName}.hasErrors()) {
            transactionStatus.setRollbackOnly()
            respond ${propertyName}.errors, view:'edit'
            return
        }
        ${propertyName}.save flush:true
        request.withFormat {
            form multipartForm {
            flash.message = message(code: 'default.updated.message', args: [message(code: '${propertyName}.label',
                                    default: '${className}'), ${propertyName}.id])
                redirect ${propertyName}
            }
            '*'{ respond ${propertyName}, [status: OK] }
        }
    }
    @Transactional
    def delete(${className} ${propertyName}) {
        if (${propertyName} == null) {
            transactionStatus.setRollbackOnly()
            notFound()
            return
        }
        ${propertyName}.delete flush:true
        request.withFormat {
            form multipartForm {
            flash.message = message(code:'default.deleted.message', args: [message(code: '${propertyName}.label',
                                    default: '${className}'), ${propertyName}.id])
                redirect action:"index", method:"GET"
            }
            '*'{ render status: NO_CONTENT }
        }
    }
    protected void notFound() {
        request.withFormat {
          form multipartForm {
          flash.message = message(code: 'default.not.found.message', args: [message(code: '${propertyName}.label',
                                  default: '${className}'), params.id])
                redirect action: "index", method: "GET"
            }
            '*'{ render status: NOT_FOUND }
        }
    }
}
```

Código 3.3: *Template* **Controller.groovy**

```
<!DOCTYPE html>
<html>
    <head>
        <meta name="layout" content="main" />
        <g:set var="entityName" value="\${message(code: '${propertyName}.label', default: '${className}')}" />
        <title><g:message code="default.create.label" args="[entityName]" /></title>
        <asset:javascript src="jquery-2.2.0.min.js" />
        <asset:javascript src="jquery.maskedinput.min.js" />
        <g:javascript>
            var JQuery = jQuery.noConflict()
            JQuery(document).ready(function(){
                JQuery("#CPF").mask("999.999.999-99");
                JQuery("#CNPJ").mask("99.999.999/9999-99");
                JQuery("#CEP").mask("99999-999");
            });
        </g:javascript>
    </head>
    <body>
        <a href="#create-${propertyName}" class="skip" tabindex="-1"><g:message code="default.link.skip.label"
                                                                        default="Skip to content&hellip;"/></a>
        <div class="nav" role="navigation">
            <ul>
                <li><a class="home" href="\${createLink(uri: '/')}"><g:message code="default.home.label"/></a></li>
                <li><g:link class="list" action="index">
                      <g:message code="default.list.label" args="[entityName]" />
                    </g:link></li>
            </ul>
        </div>
        <div id="create-${propertyName}" class="content scaffold-create" role="main">
            <h1><g:message code="default.create.label" args="[entityName]" /></h1>
            <g:if test="\${flash.message}">
            <div class="message" role="status">\${flash.message}</div>
            </g:if>
            <g:hasErrors bean="\${this.${propertyName}}">
            <ul class="errors" role="alert">
                <g:eachError bean="\${this.${propertyName}}" var="error">
                <li <g:if test="\${error in org.springframework.validation.FieldError}">
                data-field-id="\${error.field}" </g:if><g:message error="\${error}"/></li>
                </g:eachError>
            </ul>
            </g:hasErrors>
            <g:form action="save">
                <fieldset class="form">
                    <f:all bean="${propertyName}"/>
                </fieldset>
                <fieldset class="buttons">
                    <g:submitButton name="create" class="save"
                    value="\${message(code: 'default.button.create.label', default: 'Create')}" />
                </fieldset>
            </g:form>
        </div>
    </body>
</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-${propertyName}" class="skip" tabindex="-1"><g:message code="default.link.skip.label"
10                                                                  default="Skip to content&hellip;"/></a>
11         <div class="nav" role="navigation">
12             <ul>
13                 <li><a class="home" href="\${createLink(uri: '/')}"><g:message code="default.home.label"/></a></li>
14                 <sec:access controller="${propertyName}" action='create'>
15                     <li><g:link class="create" action="create"><g:message code="default.new.label"
16                                                             args="[entityName]" /></g:link></li>
17                 </sec:access>
18             </ul>
19         </div>
20         <div id="list-${propertyName}" class="content scaffold-list" role="main">
21             <h1><g:message code="default.list.label" args="[entityName]" /></h1>
22             <g:if test="\${flash.message}">
23                 <div class="message" role="status">\${flash.message}</div>
24             </g:if>
25             <f:table collection="\${${propertyName}List}" />
26
27             <div class="pagination">
28                 <g:paginate total="\${${propertyName}Count ?: 0}" />
29             </div>
30         </div>
31     </body>
32 </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), contaCount: 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="conta">
          <tr class="${(i % 2) == 0 ? 'even' : 'odd'}">
            <td><g:link action="show" controller="${conta.class}" id="${conta.id}">
                                                 ${fieldValue(bean: conta, field: "numero")}</g:link></td>
            <td>${fieldValue(bean: conta, field: "agencia")}</td>
            <td>${fieldValue(bean: conta, field: "saldo")}</td>
            <td><g:formatDate date="${conta.abertura}" /></td>
          </tr>
        </g:each>
        </tbody>
      </table>
      <div class="pagination">
        <g:paginate total="${contaCount ?: 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), clienteCount: 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="${flash.message}">
        <div class="message" role="status">${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="cliente">
          <tr class="${(i % 2) == 0 ? 'even' : 'odd'}">
            <td><g:link action="show" controller="${cliente.class}" id="${cliente.id}">${fieldValue(bean: cliente,
                  field: "nome")}</g:link></td>
            <td>${fieldValue(bean: cliente, field: "endereco")}</td>
            <td><g:formatDate date="${cliente.dtMoradia}" /></td>
            <td>${fieldValue(bean: cliente, field: "status")}</td>
          </tr>
        </g:each>
        </tbody>
      </table>
      <div class="pagination">
        <g:paginate total="${clienteCount ?: 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

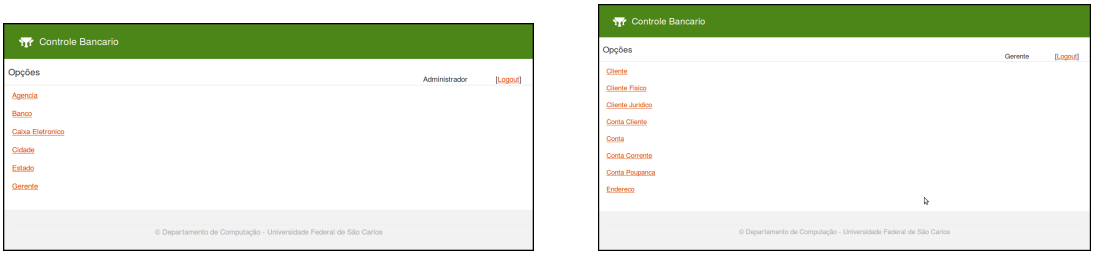
A visão **main/index.gsp** consiste na visão principal da aplicação **ControleBancario**. A implementação dessa visão encontra-se apresentada no Código 3.13.

```
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** $\Longrightarrow$ **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 leaiute 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:pageProperty name="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.Agencia
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 físicos - 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.

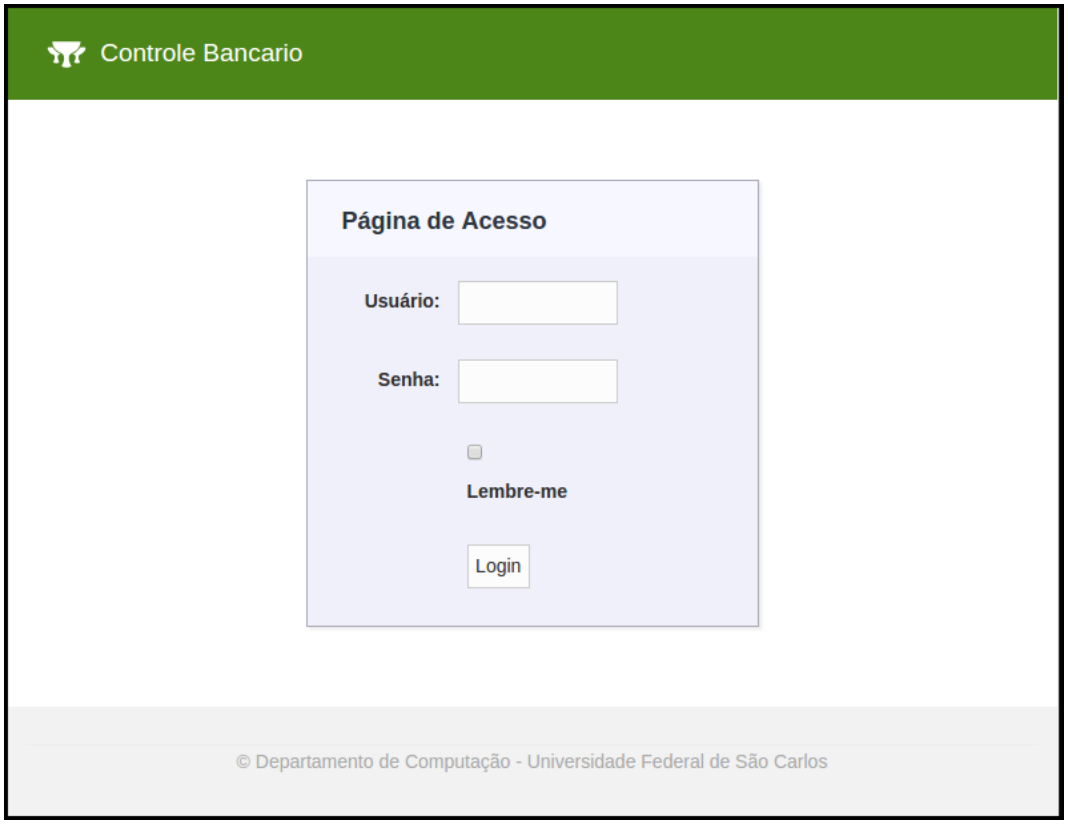


Figura 3.5: **ControleBancario**: Página de *Login*

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**.

[21] URL: `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 um *plugin* Grails que adiciona funcionalidades de clientes REST. Ou seja, ao utilizarmos esse *plugin* será possível acessar serviços web REST.

Conforme mencionado, 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 4.1.

Dessa forma, para instalar o plugin **grails-datastore-rest-client** adicione o comando `compile "org.grails:grails-datastore-rest-client:5.0.0.RC2"`, descrevendo a dependência, no arquivo **build.graddle** conforme apresentado na linha 15 do Código 4.1.

```
1  repositories {
2      mavenLocal()
3      maven { url "https://repo.grails.org/grails/core" }
4  }
5
6  dependencyManagement {
7      imports {
8          mavenBom "org.grails:grails-bom:$grailsVersion"
9      }
10     applyMavenExclusions false
11 }
12
13 dependencies {
14     compile "org.springframework.boot:spring-boot-starter-logging"
15     compile "org.springframework.boot:spring-boot-autoconfigure"
16     compile "org.grails:grails-core"
17     compile "org.springframework.boot:spring-boot-starter-actuator"
18     compile "org.springframework.boot:spring-boot-starter-tomcat"
19     compile "org.grails:grails-dependencies"
20     compile "org.grails:grails-web-boot"
21     compile "org.grails.plugins:cache"
22     compile "org.grails.plugins:scaffolding"
23     compile "org.grails.plugins:hibernate4"
24     compile "org.hibernate:hibernate-ehcache"
25     compile "org.grails.plugins:br-validation:0.3"
26     compile "org.grails.plugins:spring-security-core:3.0.4"
27     compile "org.grails:grails-datastore-rest-client:5.0.0.RC2"
28     compile "org.grails.plugins:ajax-tags:1.0.0"
29     console "org.grails:grails-console"
30     profile "org.grails.profiles:web:3.1.4"
31     runtime "org.grails.plugins:asset-pipeline"
32     runtime "com.h2database:h2"
33     runtime "org.postgresql:postgresql:9.3-1101-jdbc41"
34     testCompile "org.grails:grails-plugin-testing"
35     testCompile "org.grails.plugins:geb"
36     testRuntime "org.seleniumhq.selenium:selenium-htmlunit-driver:2.47.1"
37     testRuntime "net.sourceforge.htmlunit:htmlunit:2.18"
38 }
```

Código 4.1: **build.gradle**

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 29), essa instância é associada ao papel **ROLE_GERENTE** (linhas 31-33).

```
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 gerente) {
17         if (gerente == null) {
18             transactionStatus.setRollbackOnly()
19             notFound()
20             return
21         }
22
23         if (gerente.hasErrors()) {
24             transactionStatus.setRollbackOnly()
25             respond gerente.errors, view:'create'
26             return
27         }
28
29         gerente.save flush:true
30
31         def gerentePapel = Papel.findByAuthority("ROLE_GERENTE")
32
33         UsuarioPapel.create(gerente, gerentePapel)
34
35         request.withFormat {
36             form {
37                 flash.message = message(code: 'default.created.message',
38                                         args: [message(code: 'gerente.label',
39                                         default: 'Gerente'), gerente.id])
40                 redirect gerente
41             }
42             '*' { respond gerente, [status: CREATED] }
43         }
44     }
45 }
```

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 clienteFisico) {
        if (clienteFisico == null) {
            transactionStatus.setRollbackOnly()
            notFound()
            return
        }
        if (clienteFisico.hasErrors()) {
            transactionStatus.setRollbackOnly()
            respond clienteFisico.errors, view:'create'
            return
        }
        clienteFisico.enabled = false
        clienteFisico.save flush:true
        def clientePapel = Papel.findByAuthority("ROLE_CLIENTE")
        UsuarioPapel.create(clienteFisico, clientePapel)
        request.withFormat {
            form {
                flash.message = message(code:'default.created.message', args: [message(code:'clienteFisico.label',
                                        default: 'ClienteFisico'), clienteFisico.id])
                redirect clienteFisico
            }
            '*' { respond clienteFisico, [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 clienteJuridico) {
        if (clienteJuridico == null) {
            transactionStatus.setRollbackOnly()
            notFound()
            return
        }
        if (clienteJuridico.hasErrors()) {
            transactionStatus.setRollbackOnly()
            respond clienteJuridico.errors, view:'create'
            return
        }
        clienteJuridico.enabled = false
        clienteJuridico.save flush:true
        def clientePapel = Papel.findByAuthority("ROLE_CLIENTE")
        UsuarioPapel.create(clienteJuridico, clientePapel)
        request.withFormat {
            form {
                flash.message = message(code:'default.created.message',args: [message(code:'clienteJuridico.label',
                                        default: 'ClienteJuridico'), clienteJuridicoInstance.id])
                redirect clienteJuridico
            }
            '*' { respond clienteJuridico, [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 userName = springSecurityService.authentication.principal.getUsername()
14         def authority = usuario.getAuthorities()[0].getAuthority()
15
16         if (authority.equals('ROLE_GERENTE')) {
17             def gerente = Gerente.findByUsername(userName)
18             if (!session.agencia) {
19                 session.agencia = gerente.agencia
20             }
21         } else if (authority.equals('ROLE_CLIENTE')) {
22
23             session.cliente = Cliente.findByUsername(userName)
24
25             if (!session.conta) {
26                 if (session.cliente.contasCliente.size() == 1) {
27                     def contaCliente = session.cliente.contasCliente[0]
28                     session.contaCliente = contaCliente
29                     session.conta = contaCliente.conta
30                     session.agencia = session.conta.agencia
31                 } else {
32                     redirect(controller:'selecionaConta')
33                 }
34             }
35         }
36     }
37 }
```

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 **agencia** armazena a agência em que trabalha o usuário *logado*, caso este desempenhe o papel **ROLE_GERENTE** (linha 19);
- A variável de sessão **cliente** armazena o usuário *logado*, caso este desempenhe o papel **ROLE_CLIENTE** (linha 23);
- 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. As variáveis de sessão **conta** e **agencia** armazenam a conta (e a respectiva agência) associada à conta cliente armazenada na variável de sessão **contaCliente**;
- É importante observar que se o cliente possuir apenas uma conta associada, este já será armazenado na variável de sessão **contaCliente** (linha 28). Caso contrário, há um redirecionamento para o controlador **SelecionaContaController** (linha 32).

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* (linha 30). Além disso, essa classe imprime a conta (e a respectiva agência) que o usuário *logado* está acessando, considerando que esse desempenhe o papel **ROLE_CLIENTE** (variáveis de sessão **conta** e **agencia**, linhas 14–22). Por fim, essa classe imprime a agência em que o usuário *logado* trabalha, caso esse desempenhe o papel **ROLE_GERENTE** (variável de sessão **agencia**, linhas 23–27).

```
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              def span = "<span style=\"text-align:center;padding-left:25px;padding-right:25px\">"
9              StringBuilder sb = new StringBuilder();
10             if (authority.equals('ROLE_ADMIN')) {
11                 papel = "Administrador"
12             } else if (authority.equals('ROLE_CLIENTE')) {
13                 papel = "Cliente"
14                 if (session.conta) {
15                     sb.append("Conta: ")
16                     sb.append(session.conta)
17                     sb.append(" [")
18                     session.agencia.attach()
19                     sb.append(session.agencia)
20                     sb.append("]")
21                     sb.append(span)
22                 }
23             } else {
24                 papel = "Gerente"
25                 sb.append("Agência: ")
26                 sb.append(session.agencia)
27             }
28             out << span
29             out << papel
30             out << "</span>"
31             out << span
32             out << sb
33             out << "</span>"
34             out << "<span style=\"padding-right:25px\">"
35             out << """ [${link(controller: "logout"){"Logout"}}]"""
36             out << "</span>"
37         }
38     }
39 }
```

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() {
        def contaCliente = ContaCliente.get(params.conta)
        session.contaCliente = contaCliente
        session.conta = contaCliente.conta
        session.agencia = session.conta.agencia
        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**.

```
<%@ 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**

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.

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()**. Esses dois métodos 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 contem 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 corriga 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). 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 corriga 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:[transacaoCount: Transacao.count()]
    }

    @Transactional
    def save(Transacao transacao) {
        if (transacao == null) {
            transactionStatus.setRollbackOnly()
            notFound()
            return
        }
        if (transacao.hasErrors()) {
            transactionStatus.setRollbackOnly()
            respond transacao.errors, view:'create'
            return
        }
        def conta = transacao.contaCliente.conta
        conta.saldo += transacao.getValorReal()
        transacao.save flush:true
        conta.save flush: true
        request.withFormat {
            form multipartForm {
                flash.message = message(code: 'default.created.message', args:[message(code: 'transacao.label',
                                        default: 'Transacao'), transacao.id])
                redirect transacao
            }
            '*' { respond transacao, [status: CREATED] }
        }
    }

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

    @Transactional
    def update(Transacao transacao) {
        if (transacao == null) {
            transactionStatus.setRollbackOnly()
            notFound()
            return
        }
        if (transacao.hasErrors()) {
            transactionStatus.setRollbackOnly()
            respond transacao.errors, view:'edit'
            return
        }
        def conta = transacao.contaCliente.conta
        conta.saldo += (transacao.getValorReal() - transacao.getValorAnterior())
        transacao.save flush:true
        conta.save flush: true
        request.withFormat {
            form multipartForm {
                flash.message = message(code: 'default.updated.message', args: [message(code: 'transacao.label',
                                        default: 'Transacao'), transacao.id])
                redirect transacao
            }
            '*'{ respond transacao, [status: OK] }
        }
    }

    @Transactional
    def delete(Transacao transacao) {
        if (transacao.contaCliente.id != session.contaCliente.id) {
            flash.message = message(code: 'springSecurity.denied.message', args: [message(code: 'transacao.label',
                                    default: 'Transacao'), transacao.id])
            redirect action: "index", method: "GET"
            return
        }
        if (transacao == null) {
            transactionStatus.setRollbackOnly()
            notFound()
            return
        }
        def conta = transacao.contaCliente.conta;
        conta.saldo -= transacao.getValorReal()
        transacao.delete flush:true
        conta.save flush:true
        request.withFormat {
            form multipartForm {
                flash.message = message(code: 'default.deleted.message', args: [message(code: 'transacao.label',
                                        default: 'Transacao'), transacao.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/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/_fields.gsp

Essa seção apresenta o *template* **transacao/_fields.gsp** (Código 4.11) que será utilizado pela visões **create.gsp** e **edit.gsp** e que contém personalizações na entrada dos atributos **contaCliente** e **caixaEletronico** da classe de domínio **Transacao**. Essas personalizações tem como objetivo refletir as novas funcionalidades discutidas nesse capítulo.

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

Código 4.11: *Template* **transacao/_fields.gsp**

Conforme pode-se observar, a transação criada/editada sempre será associada à conta escolhida anteriormente pelo usuário (variável sessão **contaCliente**). Ou seja, o atributo **from** (linha 9) do campo de seleção é limitado ao valor armazenado na variável de sessão **contaCliente**.

De maneira análoga, o segundo campo de seleção (linhas 21–22) apenas possibilita que o usuário selecione 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 19–20) parametrizada. O parâmetro dessa consulta é o banco que mantém a conta sendo acessada no momento.

Código 4.12 mostra a reimplementação da visão **transacao/create.gsp**. Por questão de brevidade, apenas serão apresentadas as mudanças realizadas nessa visão. Pode-se observar que o *template* **_fields.gsp** é renderizado (linha 17) no contexto do formulário HTML definido pela *tag* **g:form** (linhas 15–25). Dessa forma, os atributos **contaCliente** e **caixaEletronico** são incluídos no formulário através da renderização do *template* **_fields.gsp** enquanto os demais são incluídos através da *tag* **f:all** (linha 19).

```
<div id="create-transacao" class="content scaffold-create" role="main">
    <h1><g:message code="default.create.label" args="[entityName]"/></h1>
    <g:if test="${flash.message}">
        <div class="message" role="status">${flash.message}</div>
    </g:if>
    <g:hasErrors bean="${this.transacao}">
        <ul class="errors" role="alert">
            <g:eachError bean="${this.transacao}" var="error">
                <li <g:if test="${error in org.springframework.validation.FieldError}">
                            data-field-id="${error.field}"</g:if><g:message
                        error="${error}"/></li>
            </g:eachError>
        </ul>
    </g:hasErrors>
    <g:form action="save">
        <fieldset class="form">
            <g:render template="fields"/>

            <f:all bean="transacao" except="contaCliente, caixaEletronico"/>
        </fieldset>
        <fieldset class="buttons">
            <g:submitButton name="create" class="save"
                            value="${message(code: 'default.button.create.label', default: 'Create')}"/>
        </fieldset>
    </g:form>
</div>
```

Código 4.12: **transacao/create.gsp**

Análogo à visão **transacao/create.gsp**, a visão **transacao/edit.gsp** também necessita ser alterada para utilizar o *template* **_fields.gsp**. Fica como exercício para o leitor realizar tal alteração.

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**.

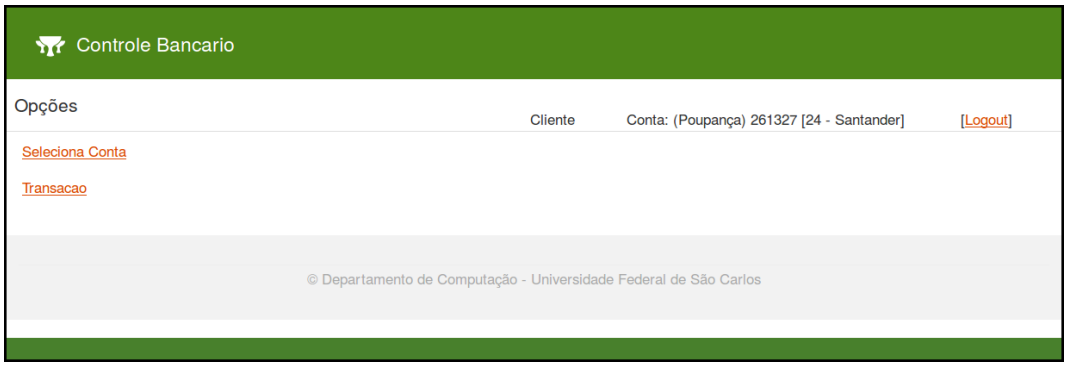


Figura 4.1: Visão **main/index.gsp**: **ROLE_CLIENTE**

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*.

Controle Bancario

Principal Novo Transacao

Cliente Conta: (Poupança) 261327 [24 - Santander] [Logout]

Transacao Listagem

Conta Cliente	Caixa Eletronico	Valor	Data	Quem	Motivo	Tipo
018.990.444-50 X (Poupança) 261327	Santander - R. Treze de Maio, 1930. Centro 13560-647 São Carlos - SP	50.0	21/04/2016 21:11:05 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	21/04/2016 21:11:05 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	21/04/2016 21:11:06 BRT	Próprio	Transferência	DÉBITO
018.990.444-50 X (Poupança) 261327	Santander - R. Treze de Maio, 1930. Centro 13560-647 São Carlos - SP	25.0	21/04/2016 21:11:06 BRT	Fulano de Tal	Transferência	CRÉDITO

© Departamento de Computação - Universidade Federal de São Carlos

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 4.13 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 **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.

```
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:[contaCorrenteCount: ContaCorrente.count()]
    }

    def edit(ContaCorrente contaCorrente) {

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

        respond contaCorrente
    }

    @Transactional
    def delete(ContaCorrente contaCorrente) {

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

        if (contaCorrente == null) {
            notFound()
            transactionStatus.setRollbackOnly()
            return
        }

        contaCorrente.delete flush:true

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

Código 4.13: Controlador **ContaCorrenteController**

4.4.2 Template contaCorrente/_fields.gsp

Essa seção apresenta o *template* **contaCorrente/_fields.gsp** (Código 4.14) que será utilizado pela visões **create.gsp** e **edit.gsp** e que contém a personalização na entrada do atributo **agencia** da classe de domínio **ContaCorrente**.

Conforme pode-se observar, a conta corrente criada/editada sempre será associada à agência (variável de sessão **agencia**) em que trabalha o usuário *logado*.

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

Código 4.14: *Template* **contaCorrente/_fields.gsp**

Código 4.15 mostra a reimplementação da visão **contaCorrente/create.gsp**. Por questão de brevidade, apenas serão apresentadas as mudanças realizadas nessa visão. Pode-se observar que o *template* **_fields.gsp** é renderizado (linha 17) no contexto do formulário HTML definido pela *tag* **g:form** (linhas 15–25). Dessa forma, o atributo **agencia** é incluído no formulário através da renderização do *template* **_fields.gsp** enquanto os demais são incluídos através da *tag* **f:all** (linha 19).

```
1  <div id="create-contaCorrente" class="content scaffold-create" role="main">
2      <h1><g:message code="default.create.label" args="[entityName]"/></h1>
3      <g:if test="${flash.message}">
4          <div class="message" role="status">${flash.message}</div>
5      </g:if>
6      <g:hasErrors bean="${this.contaCorrente}">
7          <ul class="errors" role="alert">
8              <g:eachError bean="${this.contaCorrente}" var="error">
9                  <li <g:if test="${error in org.springframework.validation.FieldError}">
10                             data-field-id="${error.field}"</g:if><g:message
11                     error="${error}"/></li>
12             </g:eachError>
13         </ul>
14     </g:hasErrors>
15     <g:form action="save">
16         <fieldset class="form">
17             <g:render template="fields"/>
18
19             <f:all bean="contaCorrente" except="agencia"/>
20         </fieldset>
21         <fieldset class="buttons">
22             <g:submitButton name="create" class="save"
23                             value="${message(code: 'default.button.create.label', default: 'Create')}"/>
24         </fieldset>
25     </g:form>
26 </div>
```

Código 4.15: **contaCorrente/create.gsp**

> ! Análogo à visão **contaCorrente/create.gsp**, a visão **contaCorrente/edit.gsp** também necessita ser alterada para utilizar o *template* **_fields.gsp**.
>
> Análogo ao *template* **contaCorrente/_fields.gsp**, o *template* **contaPoupanca/_fields.gsp** (e consequentemente as visões **contaPoupanca/create.gsp** e **contaPoupanca/edit.gsp**) também precisa ser definido com o intuito de personalizar a entrada do atributo **agencia** da classe de domínio **ContaPoupanca**. Fica como exercício para o leitor realizar tais alterações.

4.4.3 Controlador: ContaClienteController

Código 4.16 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:[contaClienteCount: ContaCliente.count()]
    }

    @Transactional
    def save(ContaCliente contaCliente) {
        if (contaCliente == null) {
            transactionStatus.setRollbackOnly()
            notFound()
            return
        }

        if (contaCliente.hasErrors()) {
            transactionStatus.setRollbackOnly()
            respond contaCliente.errors, view:'create'
            return
        }

        contaCliente.save flush:true
        def cliente = contaCliente.cliente
        if (!cliente.enabled) {
            cliente.enabled = true
            cliente.save flush:true
        }

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

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

    @Transactional
    def delete(ContaCliente contaCliente) {
        if (contaCliente != null && contaCliente.conta.agencia.id != session.agencia.id) {
            flash.message = message(code: 'springSecurity.denied.message', args: [message(code:
                    'contaCliente.label', default: 'ContaCliente'), contaCliente.id])
            redirect action: "index"
        }

        if (contaCliente == null) {
            transactionStatus.setRollbackOnly()
            notFound()
            return
        }
        contaCliente.delete flush: true

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

}
```

Código 4.16: Controlador **ContaClienteController**

4.4.4 Template contaCliente/_fields.gsp

Essa seção apresenta o *template* **contaCliente/_fields.gsp** (Código 4.17) que será utilizado pela visões **create.gsp** e **edit.gsp** e que contém personalizações na entrada dos atributos **cliente** e **conta** da classe de domínio **ContaCliente**.

```
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"
10                required="" value="${contaClienteInstance?.cliente?.id}"
11                disabled="${contaClienteInstance?.cliente?.id != null}" class="many-to-one"/>
12 </div>
13
14 <div class="fieldcontain ${hasErrors(bean: contaClienteInstance, field: 'conta', 'error')} required">
15     <label for="conta">
16         <g:message code="contaCliente.conta.label" default="Conta"/>
17         <span class="required-indicator">*</span>
18     </label>
19     <g:set var="contas"
20            value="${Conta.findAll("from Conta as conta where conta.agencia = ?", [session.agencia])}"/>
21     <g:select id="conta" name="conta.id" from="${contas}" optionKey="id" required=""
22                value="${contaClienteInstance?.conta?.id}"
23                disabled="${contaClienteInstance?.conta?.id != null}" class="many-to-one"/>
24 </div>
```

Código 4.17: *Template* **contaCliente/_fields.gsp**

Conforme pode-se observar, os campos de seleção dos atributos **cliente** e **conta** são desabilitados nas operações de edição (linhas 11 e 23) – 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. Além disso, o segundo campo de seleção (linha 21) apenas possibilita que o usuário *logado* apenas possa selecionar contas (corrente ou poupança) pertencentes à agência em que trabalha (variável de sessão **agencia**).

Código 4.18 mostra a reimplementação da visão **contaCliente/create.gsp**. Por questão de brevidade, apenas serão apresentadas as mudanças realizadas nessa visão. Pode-se observar que o *template* **_fields.gsp** é renderizado (linha 3) no contexto do formulário HTML definido pela *tag* **g:form**. Dessa forma, os atributos **cliente** e **conta** são incluídos no formulário através da renderização do *template* **_fields.gsp** enquanto os demais são incluídos através da *tag* **f:all** (linha 5).

```
1  <g:form action="save">
2      <fieldset class="form">
3          <g:render template="fields"/>
4
5          <f:all bean="contaCliente" except="cliente, conta"/>
6      </fieldset>
7
8      <fieldset class="buttons">
9          <g:submitButton name="create" class="save"
10                         value="${message(code: 'default.button.create.label', default: 'Create')}"/>
11     </fieldset>
12 </g:form>
```

Código 4.18: **contaCliente/create.gsp**

! Análogo à visão **contaCliente/create.gsp**, a visão **contaCliente/edit.gsp** também necessita ser alterada para utilizar o *template* **_fields.gsp**. Fica como exercício para o leitor realizar tal alteração.

4.4.5 Controlador: ContaController

Código 4.19 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()]
    }

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

Código 4.19: 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**, **ClienteFisico** e **ClienteJuridico** caso deseje acessar/atualizar a lista de clientes. Figura 4.4 apresenta a lista de clientes cadastrados na aplicação **ControleBancario**.

Controle Bancario

Principal

Gerente Agência: 1888 - Banco do Brasil [Logout]

Cliente Listagem

Nome	Endereco	Dt Moradia	Status
Maria da Silva	R. Humberto Manelli, 50 Apto 31. Jardim Gibertoni 13562-420 São Carlos - SP	22/04/2016 13:39:43 BRT	Ativo
Pedro Soares	R. Humberto Manelli, 50 Apto 31. Jardim Gibertoni 13562-420 São Carlos - SP	22/04/2016 13:39:43 BRT	Ativo
Viação Cometa S/A	R. Nilton Coelho de Andrade, 772. Vila Maria 03092-324 São Paulo - SP	22/04/2016 13:39:44 BRT	Ativo

© Departamento de Computação - Universidade Federal de São Carlos

Figura 4.4: Lista de clientes

- **ContaCliente** caso deseje acessar/atualizar a lista de relacionamentos entre clientes e contas da agência do usuário *logado*;

- **Conta**, **ContaCorrente** e **ContaPoupança** caso deseje acessar/atualizar a lista de contas da agência do usuário *logado*. Figura 4.5 apresenta a lista de contas da agência do usuário *logado*.

Controle Bancario

Principal

Gerente Agência: 1888 - Banco do Brasil [Logout]

Conta Listagem

Numero	Agencia	Saldo	Abertura
010414688	1888 - Banco do Brasil	1000.56	22/04/2016 13:39:44 BRT

© Departamento de Computação - Universidade Federal de São Carlos

Figura 4.5: Lista de contas da agência do usuário *logado*

- **Endereco** caso deseje acessar/atualizar a lista de endereços cadastrados.

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 desse 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 Template endereco/_address.gsp

O primeiro passo na implementação da funcionalidade de preenchimento automático de endereços consiste em definir o *template* **endereco/_address.gsp** conforme apresentado no Código 4.20. Esse *template* será utilizado pelas visões **create.gsp** e **edit.gsp** e contém todos os campos (exceto o campo CEP discutido a seguir nessa seção) responsáveis pela entrada dos valores dos atributos da classe de domínio **Endereco**.

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

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

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

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

    </label>
    <g:textField name="complemento" maxlength="20" value="${endereco?.complemento}"/>
</div>

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

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

Código 4.20: *Template* **endereco/_address.gsp**

Código 4.21 mostra a reimplementação da visão **endereco/create.gsp**. Por questão de brevidade, apenas serão apresentadas as mudanças realizadas nessa visão. Pode-se observar que o *template* **_address.gsp** é renderizado (linha 15) no contexto do formulário HTML definido pela *tag* **g:form**.

No contexto do formulário foi definido também, o campo de texto CEP (linhas 22-24) 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** (Seção 4.5.2).

```
1  <div id="create-endereco" class="content scaffold-create" role="main">
2      <h1><g:message code="default.create.label" args="[entityName]" /></h1>
3      <g:if test="${flash.message}">
4         <div class="message" role="status">${flash.message}</div>
5      </g:if>
6      <g:hasErrors bean="${this.endereco}">
7          <ul class="errors" role="alert">
8              <g:eachError bean="${this.endereco}" var="error">
9                  <li <g:if test="${error in org.springframework.validation.FieldError}">
10                                 data-field-id="${error.field}"</g:if><g:message error="${error}"/></li>
11             </g:eachError>
12         </ul>
13     </g:hasErrors>
14     <g:form action="save">
15        <fieldset class="form">
16
17           <div class="fieldcontain ${hasErrors(bean: endereco, field: 'CEP', 'error')} required">
18                 <label for="CEP">
19                     <g:message code="endereco.CEP.label" default="CEP" />
20                     <span class="required-indicator">*</span>
21                  </label>
22                  <g:textField name="CEP" maxlength="9" required="" value="${endereco?.CEP}"
23                           onblur="${remoteFunction(action: 'addressByCEP', update: [success: 'addressContainer'],
24                                   params: '\'cep=\' + this.value', asynchronous: false)}"/>
25           </div>
26
27           <div id="addressContainer">
28                 <g:render template="address" bean="${endereco}"/>
29           </div>
30
31         </fieldset>
32         <fieldset class="buttons">
33              <g:submitButton name="create" class="save"
34                              value="${message(code: 'default.button.create.label', default: 'Create')}" />
35         </fieldset>
36     </g:form>
37 </div>
```

Código 4.21: **endereco/create.gsp**

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 prenchimento 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** (Código 4.21, linhas 23 e 27).

Análogo à visão **endereco/create.gsp**, a visão **endereco/edit.gsp** também necessita ser alterada para utilizar o *template* **_address.gsp**. Fica como exercício para o leitor realizar tal alteração.

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

4.5.2 Controlador: EnderecoController

O segundo passo na implementação da funcionalidade de preenchimento automático de endereços consiste em implementar a ação **addressByCEP()** que, conforme discutido anteriormente, é invocado pelo *template* **endereco/_address.gsp** (Código 4.22).

Essa ação utiliza as funcionalidades providas pelo *plugin* **grails-datastore-rest-client** que possibilita o acesso a serviços *web* REST. Ou seja, conforme pode-se observar, a ação **addressByCEP ()** invoca um serviço *web* REST que, dado um CEP como parâmetro, retorna as demais informações de um endereço (logradouro, bairro, cidade, etc).

O serviço *web* REST retorna o endereço em formato JSON[23] 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.

```
class EnderecoController {

    // Demais ações/atributos/métodos do controlador EnderecoController

    def addressByCEP() {

        String restUrl="http://cep.republicavirtual.com.br/web_cep.php?cep="+params.cep+"&formato=json"

        RestBuilder rBuilder = new RestBuilder()
        RestResponse rResponse = rBuilder.get(restUrl)

        def html = rResponse?.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: [endereco: new Endereco(params)]
    }
}
```

Código 4.22: Controlador **EnderecoController**

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

Figura 4.6 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()**.

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

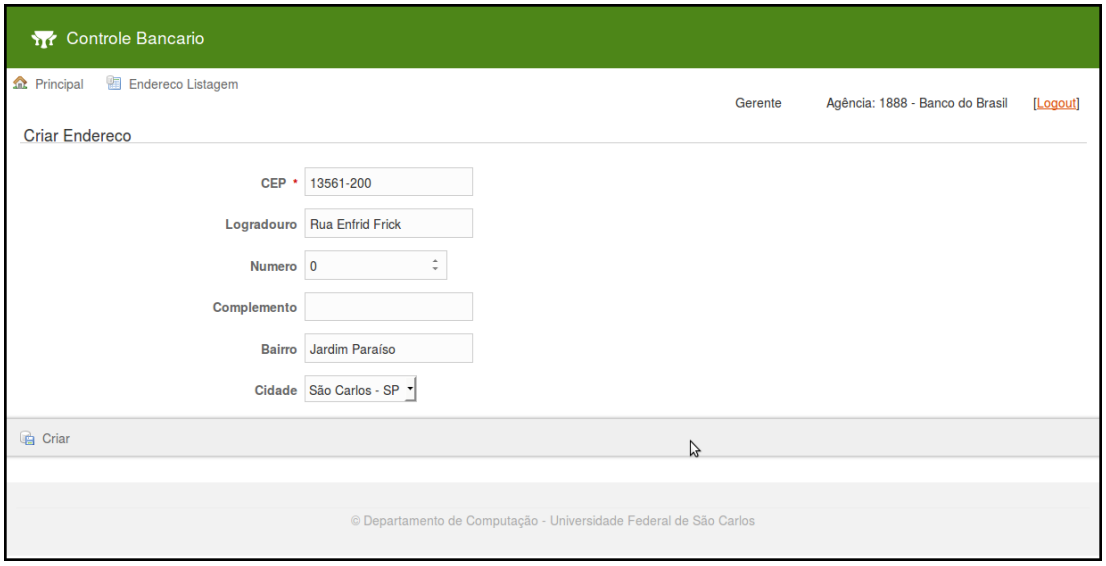


Figura 4.6: Visão **endereco/create.gsp**: Preenchimento automático de atributos

4.6 Considerações finais

Esse capítulo apresentou a terceira versão da implementação da aplicação **ControleBancario**. O código-fonte dessa aplicação (`ControleBancarioV3`) encontra-se disponível em um repositório *GitHub*[24].

Dando continuidade, o próximo capítulo apresenta o *overview* de mais algumas funcionalidades presentes em Grails que não foram abordadas nos capítulos anteriores.

[24]URL: `https://github.com/delanobeder/FG`

5 — 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.

5.1 Serviços Web

Serviços web consistem em fornecer uma API web para a sua aplicação web e são geralmente implementados utilizando REST.

5.1.1 REST

REST não é uma tecnologia em si, pode ser considerada mais como um 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 (*Create*, *Read*, *Update* e *Delete*).

A abordagem mais fácil para criar uma API REST em Grails é expor uma classe de domínio como um recurso REST. Isto pode ser feito através da adição da transformação **grails.rest.Resource** a qualquer classe de domínio (Código 5.1).

Ao adicionar a transformação **grails.rest.Resource** e especificar uma URI, a classe de domínio estará automaticamente disponível como um recurso REST nos formatos XML ou JSON. Além disso, a transformação registrará o mapeamento necessário e criará um controlador (no caso do exemplo, **LivroController**). Por fim, é possível alterar o padrão de retorno (formato JSON ao invés do formato XML) ao definir os atributos de formatos na transformação **grails.rest.Resource**.

```
import grails.rest.*

@Resource(uri='/livros', formats=['json', 'xml'])
class Livro {

    String titulo

    static constraints = {
        titulo blank:false
    }
}
```

Código 5.1: Classe de domínio – recurso REST

Dessa forma, quando acessando a URL `http://localhost:8080/livros`, definida no atributo **uri** da transformação **grails.rest.Resource**, iria renderizar uma resposta para o acesso ao recurso REST – a lista de livros no formato JSON:

```
[{"id":1,"titulo":"The Definitive Guide to Grails"},{"id":2,"titulo":"Grails in Action"}]
```

Caso a URL fosse `http://localhost:8080/livros.xml`, a resposta seria – a lista de livros no formato XML:

```
<list>
  <livro id="1">
    <titulo>The Definitive Guide to Grails</titulo>
  </livro>
  <livro id="2">
    <titulo>Grails in Action</titulo>
  </livro>
</list>
```

Para maiores informações sobre serviços web REST, consulte o endereço: `https://grails.github.io/grails-doc/latest/guide/webServices.html`.

5.2 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 (ou integrados). Para melhor explicar esses dois conceitos, utilizaremos a definição presente no livro de Weissmann [9]:

> *Grails nos fornece por padrão dois tipos de testes: unitário e integrado. O objetivo do teste unitário é nos fornecer o ferramental necessário para que possamos verificar o funcionamento de uma funcionalidade de modo isolado, ou seja, sem que precisemos iniciar todo o sistema para isto.*
>
> *Isso nos permite focar apenas na lógica que estamos implementando naquele trecho do sistema, mas por outro lado, dado que sempre precisamos interagir com outros componentes arquiteturais, por exemplo, a camada de persistência, um novo problema surge. Como fazê-lo? A resposta é simples: criando versões falsas dos mesmos (os mocks). (...)*

> *Testes unitários nos fornecem, portanto, dependências do nosso código que funcionam em "condições ideais de temperatura e pressão", ou seja, elas funcionam exatamente como gostaríamos, o que pode ser um problema, mas nos ajuda a focar melhor naquilo que desejamos verificar.*
>
> *Já testes integrados resolvem o problema da "condição ideal de temperatura e pressão" ao nos disponibilizar o sistema completo. Você não precisa mais de mocks, pois estará usando o próprio sistema para isso. São fundamentais para que vejamos qual será o comportamento do sistema quando for para produção, fornecendo-nos um ambiente que é o mais similar possível a este.*
>
> (WEISSMANN, H. L. *Falando de Grails – Altíssima produtividade no desenvolvimento web*. São Paulo: Casa do Código, 2015.)

Todos os testes se encontram no diretório **src/test/groovy (ou src/integration-test/groovy)** 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 em desses diretórios. Da mesma forma que o GORM é fortemente baseado no framework *hibernate*, o arcabouço de testes presente em Grails é fortemente baseado no *Spock*[25].

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**.

Assim como diversos aspectos do Grails, aqui é necessário ater-se à algumas convenções. Toda classe de teste possui o sufixo **Spec** em seu nome. Sendo assim, os testes unitários para a classe de domínio **Usuario**, por exemplo, ficariam em **src/test/groovy/UsuarioSpec.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 **Spec** no final do arquivo, Grails o inclui automaticamente.

5.2.1 Grails: TDD e BDD

Grails prega a automação dos testes unitários e de integração, conscientizando os desenvolvedores de sua importância e dá suporte pleno à adoção das técnicas de teste de software *Test-Driven Development (TDD)* e *Behaviour-Driven Development (BDD)*.

Test-Driven Development faz parte dos princípios ágeis de desenvolvimento. Tal técnica foi criada por Kent Beck que impulsionou a ideia de escrever testes automatizados antes da implementação do código [10].

A prática do *Test Driven Development (TDD)* pode ser resumida de uma forma bastante rudimentar em uma frase: escreva seus testes antes do código. É na realidade a aplicação de um modelo de desenvolvimento baseado em rápidos e pequenos ciclos [9].

Test Driven Development (TDD) é um processo de desenvolvimento iterativo, no qual cada iteração é feita com a escrita de um teste que faz parte da especificação do que será implementado. Todas as iterações são caracterizadas por não serem longas e propiciarem um rápido feedback sobre o código que esta sendo desenvolvido. O início do desenvolvimento com *TDD* é feito

[25] `https://code.google.com/archive/p/spock`

definindo-se uma meta, ao contrário de definir o código que será implementado para resolver determinado problema [11].

Behaviour-Driven Development (BDD) é uma técnica ágil [12], e uma evolução do *Test-Driven Development (TDD)*, como uma junção de outras técnicas ágeis existentes que são úteis para o desenvolvimento de software, cujo destaque para sua utilidade se deve à redução dos custos com modificações no software e funções do comportamento. O *BDD* como uma técnica ágil incentiva a participação e a colaboração de todos os membros de um projeto [13]. Ao longo dos anos, *BDD* progrediu para um processo que abrange análise de requisitos e desenvolvimento do código.

Tal prática de teste faz uso da linguagem narrativa em junção com a linguagem ubíqua, para a escrita de casos de testes e favorece a definição do comportamento do sistema, a linguagem que se aplica ao BDD é retirada dos cenários criados durante a fase de análise ou levantamento de requisitos e permite uma comunicação entre todos os membros da equipe.

5.2.2 BDD: um exemplo prático

Nessa seção apresentaremos um exemplo simples da utilização do *BDD*, através da ferramenta de testes *Spock*, na automação de testes.

Ao adotarmos o *BDD*, a nomenclatura dos termos devem ser enfatizados – serão adotados os termos "comportamento" (*behaviour*) e "especificação" ao invés de testes. Para maiores detalhes sobre o porquê dessa nomenclatura, consulte o artigo original sobre *Behaviour-Driven Development (BDD)* de Dan North [12].

Dessa forma, no jargão da ferramenta de testes Spock, não temos testes, mas sim comportamentos – sendo um conjunto de comportamentos, considerado como uma especificação (arquivo **XSpec.groovy**).

O *BDD* incentiva os desenvolvedores a escreverem testes cujo nome não seja uma função "usual" tal como **testCriarEndereco**, mas sim frases que possam ser compreendidas, além de pelos desenvolvedores, também pela equipe de negócio.

Figura 5.1 apresenta um dos comportamentos presente no arquivo **EnderecoControllerSpec.groovy**. Esse comportamento testa a funcionalidade de criação de instâncias da classe de domínio **Endereco** – ação **create()** do controlador **EnderecoController**.

```
@TestFor(EnderecoController)
@Mock(Endereco)
class AgenciaControllerSpec extends Specification {

// Demais comportamentos dessa especificação

  void "Test the create action returns the correct model"() {
      when:"The create action is executed"
           controller.create()

      then:"The model is correctly created"
           model.endereco!= null
  }
}
```

Figura 5.1: Especificação de testes para o controlador EnderecoController

As palavras reservadas **when:** e **then:** demarcam blocos de iteração usados pela ferramentas de testes Spock. O primeiro, **when:** (quando), é usado para definir as condições em cima das quais deve-se verificar um comportamento. É normalmente onde estão definidas as variáveis que serão usadas na verificação. Conforme descrito na Figura 5.1, é o bloco em que será checado como o código se comportará ao criar uma instância da classe de domínio **Endereco**.

O próximo bloco, **then:** , demarca aquilo que deve-se verificar. Neste caso, deve ser escrita uma expressão booleana por linha. Voltando ao caso da Figura 5.1, deseja-se se certificar de que a instância da classe de domínio **Endereco** foi criada e não é nula.

Por fim, é importante mencionar que a especificação **EnderecoController**, através da anotação **@Mock**, obtém um *mock* da classe de domínio **Endereco** – uma versão do GORM feita especificamente para testes unitários que funciona inteiramente em memória.

Escrevendo testes unitários para controladores

Nessa seção, vamos abordar um simples controlador hipotético denominado **TestController** no qual será simulado as operações mais usuais na codificação desse tipo de controlador.

Renderização de texto: Vamos começar pela situação mais simples possível: uma ação que renderiza um texto simples. Para tal, imagine que nosso controlador possua uma ação chamada **index** cuja implementação vemos a seguir:

```
def index() {
   render "Olá Mundo !"
}
```

Essa funcionalidade poderia ser testada pelo comportamento abaixo:

```
void "testando a ação index"() {
    when:
      controller.index()
    then:
      response.text == "Olá Mundo !"
}
```

Renderização de GSP: Outra ação comum de um controlador é a renderização de uma página GSP. Novamente, algo bastante simples de ser verificado. Imagine a ação a seguir:

```
def renderGSP() {
   render view: 'renderGSP', model: [titulo: "Grails in Action"]
}
```

Essa funcionalidade poderia ser testada pelo comportamento abaixo. Não se verifica apenas se a página correta foi usada, mas também se o modelo é o esperado.

```
void "testando a ação renderGSP"() {
    when:
      controller.renderGSP()
    then: "o modelo deve ter a chave titulo"
      model.titulo == "Grails in Action"
      view == "/test/renderGSP"
}
```

Testando redirecionamento: Outra ação comum de um controlador é redirecionar para outra ação ou controlador. Analogamente, é simples de ser verificado. Imagine a ação a seguir:

```
def redirect(String papel){
   if (papel == "ROLE_ADMIN")
       redirect(controller:"admin")
   else
       redirect(controller: "cliente")
}
```

Essa funcionalidade poderia ser testada pelos 2 comportamentos abaixo que invocam a ação **redirect** do controlador passando diferentes parâmetros.

```
void "testando a ação redirect - papel ROLE_ADMIN"() {
    when:
      controller.redirect("ROLE_ADMIN")
    then:
      response.redirectedUrl == "/admin"
}

void "testando a ação redirect - papel ROLE_CLIENTE"() {
    when:
      controller.redirect("ROLE_CLIENTE")
    then:
      response.redirectedUrl == "/cliente"
}
```

5.3 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.

Exercício proposto

Um amigo seu que possui em casa um grande conjunto de mídias (CDs ou DVDs) onde estão armazenados músicas, filmes e jogos, cansado de nunca encontrar os seus CDs/DVDs, fica sabendo que você está estudando Grails e suplica a você que crie um sistema web para administrar as suas mídias. Para quebrar o galho deste seu amigo, neste exercício proposto, você implementará um catálogo de mídias em Grails.

Entidades do Catálogo

O catálogo de mídias será implementado através das seguintes entidades: **Midia**, **CD**, **Jogo**, **DVD**, **Faixa**, **Ator**, **Usuario**, **Papel** e **UsuarioPapel**.

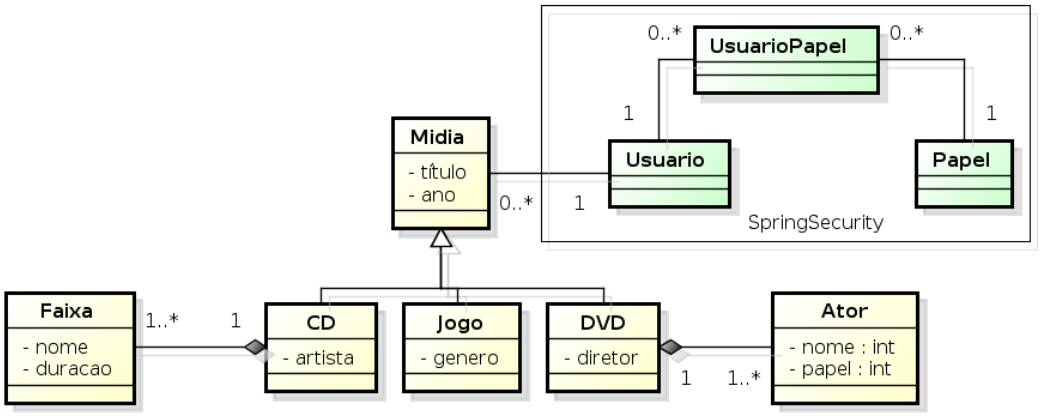


Figura 5.2: Catálogo de Mídias.

Entidade Midia

A entidade **Midia** é uma classe abstrata que deverá conter atributos para armazenar os seguintes dados sobre as mídias: **título** e **ano de criação**.

Entidade CD

A entidade **CD** (subclasse de **Midia**) é uma classe que representa um CD de música e deve conter atributos para armazenar os seguintes dados: **artista** (compositor/interpréte da obra) e **a lista de faixas** (entidade **Faixa**). Vale a pena salientar que é um relacionamento **1 para N** entre **CD** e **Faixa**.

Entidade Faixa

A entidade **Faixa** é uma classe que representa uma faixa de música e deve conter os seguintes atributos: **nome** da faixa e **duração** da faixa em segundos.

Classe Jogo

A classe **Jogo** (subclasse de **Midia**) representa um jogo eletrônico e deve conter o seguinte atributo: **gênero** do jogo eletrônico (Esportes, Corrida, RPG, Aventura, Tabuleiro, etc).

Entidade DVD

A entidade **DVD** (subclasse de **Midia**) é uma classe que representa um filme em DVD e deve conter atributos para armazenar os seguintes dados: **diretor** do filme e uma **lista dos principais atores/artistas** (entidade **Ator**) que atuaram no filme e o papel desempenhado no filme. Vale a pena salientar que é um relacionamento **1 para N** entre **DVD** e **Ator**.

Entidade Ator

A entidade **Ator** é uma classe que representa um ator/atriz e deve conter atributos para armazenar os seguintes dados: o **nome** do ator/atriz e o **papel** desempenhado no filme .

Entidades Usuario, Papel e UsuarioPapel

A entidade **Usuario** é uma classe que representa os usuário do sistema. Cada usuário terá uma coleção de mídias (entidade **Midia**) associadas a ele. Vale a pena salientar que é um relacionamento **1 para N** entre **Usuario** e **Midia** (As mídias são categorizados em: CDs de música, filmes em DVD e jogos).

A entidade **Papel** é uma classe que representa os papéis que os usuários podem desempenhar. Cada papel possui permissões a ele associadas.

A entidade **UsuarioPapel** é uma classe que representa o relacionamento muitos-para-muitos entre usuários e papéis. Ou seja, um usuário pode desempenhar vários papeis e um papel pode ser desempenhado por vários usuários.

As entidades **Usuario**, **Papel** e **UsuarioPapel** devem ser geradas através do comando **s2-quickstart** discutido no capítulo 3 do material.

Observações importantes

- É necessário a utilização do **mecanismo de herança** e associações **1 para N** para a implementação do catálogo.
- Internacionalização (I18n)
  - Português e uma língua estrangeira (Inglês, Francês, Alemão, Japonês, etc)
- Autenticação
  - Simples: login/senha (armazenado em banco de dados) da entidade **Usuario**.
- Versão *grails* (mínimo): 3.1.4

Funcionalidade: redirecionamento

Implemente uma página de *login* que autentica os usuários e realiza o direcionamento dependente do papel do usuário:

- Se papel do usuário é **Administrador**, direcione para a lista de usuários.
- Se papel do usuário é **Comum**, direcione para a lista de CDs do usuário logado.

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.

[9] WEISSMANN, H. *Falando de Grails – Altíssima produtividade no desenvolvimento web*. São Paulo: Casa do Código, 2015.

[10] BECK, K. *Test Driven Development: By Example*. Boston: Addison-Wesley, 2015.

[11] JOHANSEN, C. *Test-Driven JavaScript Development*. Boston: Addison-Wesley, 2011.

[12] NORTH, D. Introducing Behaviour-Driven Development. Better Software Magazine, 2006.

[13] LAZAR, I.; MOTOGNA, S.; PÂRV, B. Behavior Driven Development of Foundational UML Components. Eletronic Notes in Theoretical Computer Science, v. 264, n. 1, p. 91–105, 2010.

Índice Remissivo

A

Ambiente de Desenvolvimento 4

B

Banco de Dados 5, 10
Bootstrap 33

C

Comando
- grails create-integration-test 91
- grails create-unit-test 91

Comandos
- grails create-controller 27
- grails create-app 7
- grails compile 42
- grails create-controller 52, 54, 56
- grails create-domain-class 12
- grails generate-all 28, 52
- grails install-plugin 9
- grails install-templates 46
- grails run-app 37, 64
- grails s2-quickstart 43

Convenção
- *versus* Configuração 4, 8, 29
- Nomenclatura URL 30

E

Engenharia Web 1

G

GORM 5, 14
- Relacionamento de herança 24

I

Internacionalização - I18n 44

M

Modelo-Visão-Controlador (MVC) 2
- Controlador 27, 30
- Modelo 12
- Visão 27, 32

P

Plugins
- br-validation 9
- rest 68
- spring-security 41, 43, 47, 50

S

Scaffolding 27
    Dinâmico 27
    Estático 28
Serviços Web 89
    REST 89
Serviços *web* REST
    Cliente 87

U

URL
    Mapeamento 55

V

Validação de Dados 13