



Desenvolvimento de Software em Grails

Antonio Francisco do Prado

Bruno Lorenzo Lopes

Delano Medeiros Beder

Sumário

1 .Minhas Tarefas - A primeira aplicação em Grails.....	8
1.1 .O framework Grails.....	8
1.2 .Minhas Tarefas - Detalhes da aplicação.....	9
1.3 . Criando uma aplicação e definindo as classes de domínio.....	10
1.4 .Personalizando os controllers.....	21
1.5 .Personalizando as views.....	23
1.6 .Configurando um banco Postgres.....	24
2 . Spring Security - Como tornar sua aplicação segura.....	25
2.1 . Configurando o Spring Security Core.....	25
2.2 . Configurando o Spring Security UI.....	32
2.3 . Utilizando as classes do Spring Security nas nossas classes de domínio.....	33

Lista de Figuras

Figura 1.1: Diagrama de classes da aplicação MinhasTarefas.....	10
Figura 1.2: Criação de uma aplicação Grails no IntelliJ Idea.....	11
Figura 1.3: Criação de classe de domínio no IntelliJ.....	13
Figura 1.4: Tela da inicial da aplicação MinhasTarefas.....	16
Figura 1.5: Tela de listagem das listas de tarefas, inicialmente ela se encontra vazia.....	16
Figura 1.6: Tela de cadastro de uma nova lista de tarefas.....	17
Figura 1.7: Listagem de tarefas agora apresenta uma única lista, que acabmos de criar.....	18
Figura 1.8: Listagem com diversas listas de tarefas criadas no BootStrap.groovy.....	19
Figura 2.1: Tela de login do Spring Security Core.....	28

Lista de Tabelas

Tabela 1.1: Relação de arquivos de views e suas respectivas telas criados pelo Grails.....	14
Tabela 2.1: Permissões padrão do Spring Security e seus significados.....	30

1 .Minhas Tarefas - A primeira aplicação em Grails

Este capítulo visa mostrar como desenvolver uma aplicação *web* simples denominada **Minhas Tarefas** utilizando o *framework* Grails, o qual é apresentado sucintamente na seção 1.1. Nessa aplicação, o usuário pode cadastrar listas de tarefas para organizar suas atividades do dia a dia, bem como marcar quais já foram realizadas.

Os detalhes dessa aplicação são apresentados na seção 1.2. Na seção 1.3, é mostrado como criar uma aplicação Grails, definir classes de domínio, que são utilizadas para representar e armazenar as informações presentes no sistema, e permitir o seu cadastro utilizando o Scaffolding do Grails.

A estrutura básica dos *controllers*, os quais tratam as chamadas as *urls* da aplicação realizadas pelo usuário é abordada na seção 1.4. A seção 1.5 mostra como modificar as *views*, que são as páginas visualizadas pelos usuários. Na seção 1.6, veremos como implementar as funcionalidades da aplicação usando o Grails.

1.1 .O framework Grails

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

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

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

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.

1.2 .Minhas Tarefas - Detalhes da aplicação

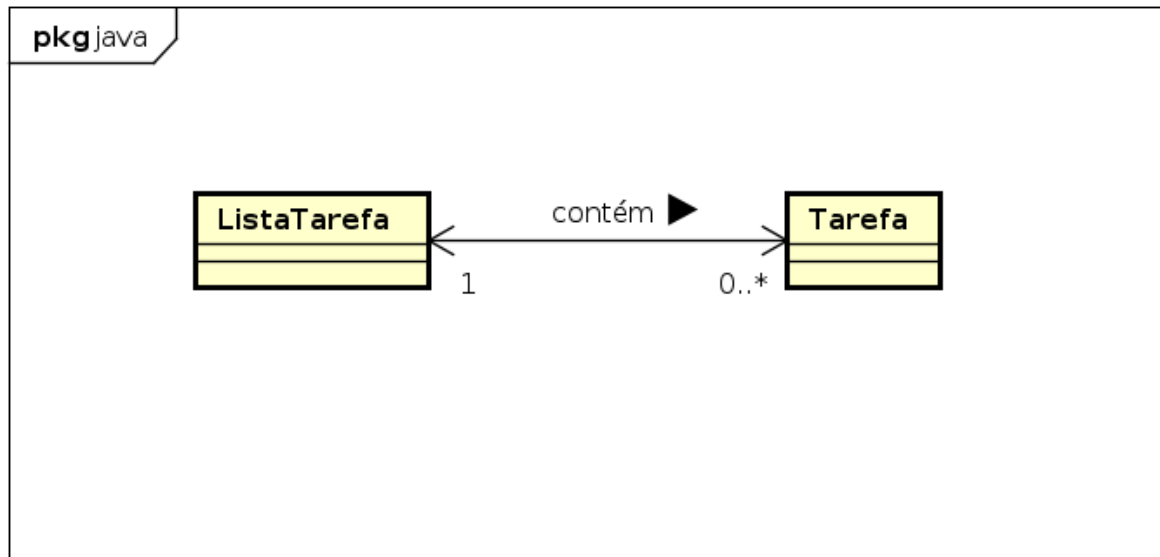
A aplicação Minhas Tarefas permite ao usuário cadastrar e gerenciar listas de tarefas, facilitando o gerenciamento das mesmas. Cada lista pode ter um significado diferente, como por exemplo, uma lista de compras de supermercado, filmes que o usuário quer assistir, tarefas domésticas, entre outras.

Cada uma das listas de tarefa pode ter uma ou mais tarefas, de acordo com a necessidade do usuário. Uma tarefa pode estar em dois estados distintos. O primeiro estado é "aberta", ou seja, ainda não foi realizada. O segundo estado é "concluída", caso o usuário já tenha terminado de realizar a tarefa.

O usuário tem a capacidade de filtrar as tarefas pela lista na qual elas estão inseridas e/ou pelo seu estado. Com isso consegue facilmente encontrar as tarefas desejadas. Apesar de simples, essa aplicação permitirá apresentar conceitos importantes no desenvolvimento *web*, utilizando o *framework* Grails.

Para simplificar o entendimento da aplicação devemos analisar o diagrama de classes da aplicação, reproduzido na Figura 1.1. Nele vemos que uma ListaTarefa pode ter zero ou mais tarefas. Já uma determinada tarefa, por sua vez, deve estar associada a somente uma lista.

Outro detalhe interessante é a navegabilidade das classes. A partir da lista, podemos identificar as tarefas presentes nela. De modo semelhante, a partir de uma tarefa também é possível identificar a qual lista ela está associada.



powered by Astah

Figura 1.1: Diagrama de classes da aplicação MinhasTarefas

1.3 . Criando uma aplicação e definindo as classes de domínio

A criação de uma aplicação Grails é bastante simples e pode ser realizada pelo terminal, com o seguinte comando:

grails create-app MinhasTarefas

Outro modo é criar um projeto utilizando alguma IDE com suporte ao Grails. No caso do IntelliJ IDE, a criação de um projeto segue os seguintes passos:

- No menu principal, selecione: Create New Project ==> Grails.
- Em nome do projeto, digite MinhasTarefas e clique em Finish (Figura 1.2). O IntelliJ IDE executa o comando grails create-app.

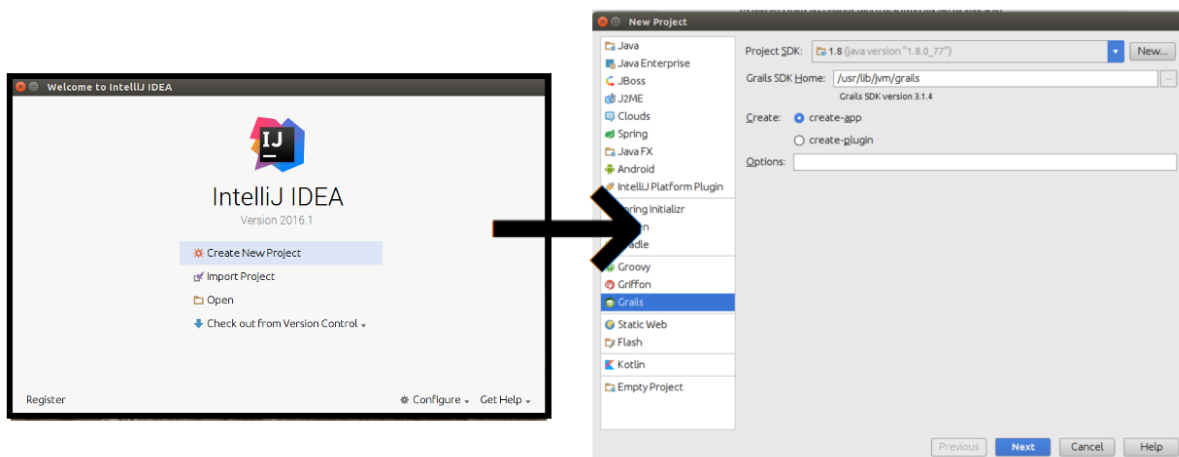


Figura 1.2: Criação de uma aplicação Grails no IntelliJ Idea.

Após a execução desse comando, o projeto será criado, ou seja, alguns arquivos e uma estrutura pré-definida de diretórios. A lista dos diretórios criados é apresentada a seguir:

Diretório	Descrição
grails-app/domain	Onde se encontra o M do MVC. Ou seja, onde se encontram as classes de Domínio, ou modelos.
grails-app/controllers	Onde se encontra o C do MVC. Ou seja, onde se encontram os controladores.
grails-app/views	Onde se encontra o V do MVC. Ou seja, onde se encontram as visões (arquivos.gsp – Groovy Server Pages).
grails-app/taglib	Onde se encontram as bibliotecas de marcas (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.

Para criar a primeira classe de domínio `ListaTarefa`, a qual será utilizada para representar cada uma das listas criadas pelo usuário, podemos digitar no terminal o seguinte comando:

grails create-domain-class br.ufscar.minhasTarefas.ListaTarefa

Ao especificar o nome completo da classe de domínio, ou seja, o nome do pacote (`br.ufscar.minhasTarefas`) e o nome da classe (`ListaTarefa`), o Grails criará a estrutura de diretórios para representar o pacote.

De modo análogo, podemos utilizar o IntelliJ para criar a classe de domínio (Figura 1.3), basta seguir esses passos:

- Selecione `New ==> Grails Domain Class`.
- Digite `br.ufscar.minhasTarefas.ListaTarefa` como o nome da classe de domínio e clique em `Finish`.

Nos dois casos, o arquivo da classes de domínio (`ListaTarefa.groovy`) e a estrutura de seu pacote será criada no diretório `grails-app/domain`.

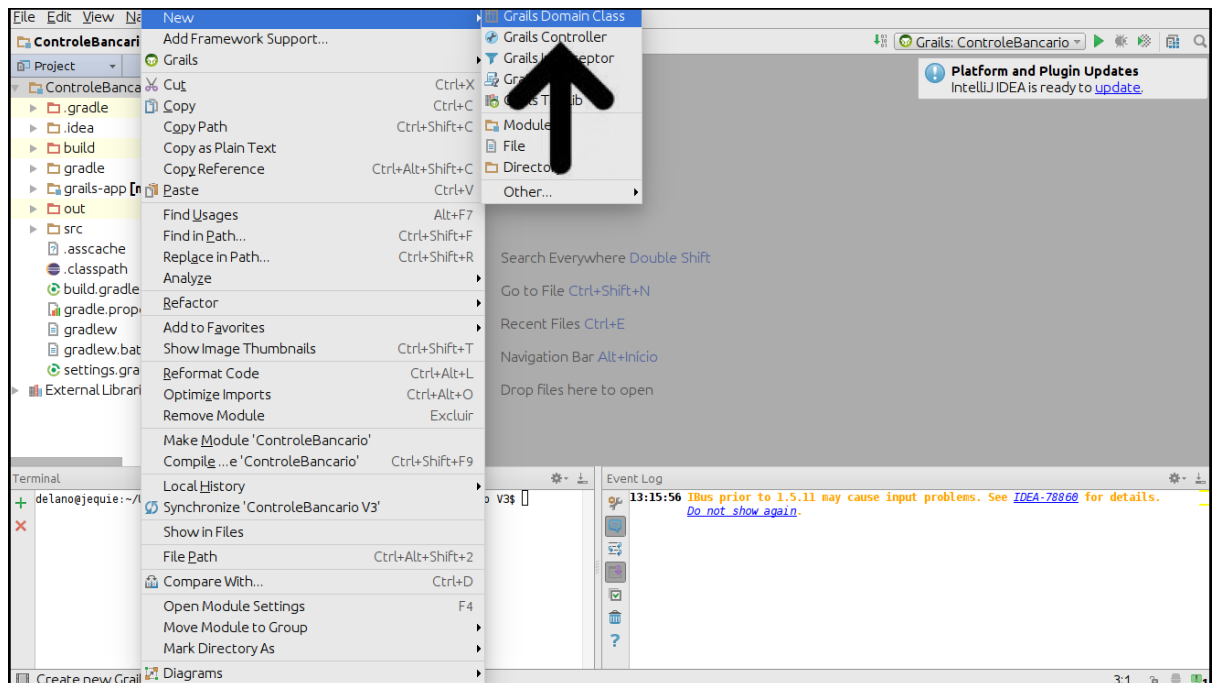


Figura 1.3: Criação de classe de domínio no IntelliJ.

Inicialmente a classe estará vazia:

```
package br.ufscar.minhasTarefas
class ListaTarefa {
    static constraints = {
    }
}
```

É importante ressaltar, que por padrão, o Grails define automaticamente uma variável *id* para as classes de domínio. Esse atributo é utilizado como chave primária das tabelas que o Grails cria no banco. Por ser uma chave primária artificial, essa estratégia é denominada *surrogate*. Dada essa explicação, vamos alterar a classe de domínio para adicionar algumas informações úteis:

```
package br.ufscar.minhasTarefas
class ListaTarefa {
    String nome
    Boolean preferida = false
    Boolean ativa = true
    static constraints = {
    }
}
```

```
}
```

Desse modo, nossa classe armazenará três informações importantes sobre nossa lista de Tarefas. Primeiro, seu nome, para permitir a sua identificação. Também serão armazenadas outras duas informações. Se a lista é uma das listas preferidas do usuário, e se ela está ativa.

Apesar da classe não estar completa, uma vez que ainda não estão presentes os itens que compõem a lista, já é possível utilizar o Scaffolding do Grails para gerar o cadastro de listas. Para isso, podemos utilizar o seguinte comando no terminal:

```
grails generate-all br.ufscar.minhasTarefas.ListaTarefa
```

O comando apresenta a seguinte saída no terminal:

```
| Rendered template Controller.groovy to destination grails-app/controllers/br/ufscar/
minhasTarefas/ListaTarefaController.groovy
| Rendered template Spec.groovy to destination src/test/groovy/br/ufscar/
minhasTarefas/ListaTarefaControllerSpec.groovy
| Scaffolding completed for grails-app/domain/br/ufscar/minhasTarefas/
ListaTarefa.groovy
| Rendered template edit.gsp to destination grails-app/views/listaTarefa/edit.gsp
| Rendered template create.gsp to destination grails-app/views/listaTarefa/create.gsp
| Rendered template index.gsp to destination grails-app/views/listaTarefa/index.gsp
| Rendered template show.gsp to destination grails-app/views/listaTarefa/show.gsp
| Views generated for grails-app/domain/br/ufscar/minhasTarefas/ListaTarefa.groovy
```

Podemos perceber que vários arquivos foram gerados. Os arquivos com extensão gsp são aqueles relacionados as views. As views são as páginas que o usuário consegue interagir pelo navegador. A Tabela 1.1 relaciona os arquivos gerados com a tela.

Arquivo	Tela
create.gsp	Cadastro de uma nova lista
edit.gsp	Edição das informações de uma lista existente
index.gsp	Tabela com a lista das listas de tarefas
show.gsp	Detalhes de uma lista

Tabela 1.1: Relação de arquivos de views e suas respectivas telas criados pelo Grails.

Outro arquivo gerado, é o `ListaTarefaController.groovy`. Nesse arquivo está definido um *controller* (controlador). Controllers tem como objetivo atender as urls que o usuário acessa. A seguir apresentamos de modo compacto os métodos do controller `ListaTarefaController`.

```
def index(Integer max) {...}
def show(ListaTarefa listaTarefa) {...}
def create() {...}
@Transactional
def save(ListaTarefa listaTarefa) {...}
def edit(ListaTarefa listaTarefa) {...}
@Transactional
def update(ListaTarefa listaTarefa) {...}
@Transactional
def delete(ListaTarefa listaTarefa) {...}
protected void notFound() {...}
```

Cada um desses métodos, com exceção do método `notFound`, responde a uma URL específica. Os métodos `index`, `show`, `create` e `edit` apresentam respectivamente as telas de listagem, detalhes do objeto, criação de um novo objeto e edição de um objeto existente. Já os métodos `save`, `update` e `delete` respectivamente servem para criar, alterar e excluir uma nova lista de tarefas no banco de dados.

Para vermos o cadastro em ação, podemos rodar nossa aplicação. Pelo terminal basta executarmos o seguinte comando:

grails run-app

Esse comando faz com que a aplicação seja executada. O log nos informa o endereço no qual a aplicação fica disponível para ser acessada:

| Running application...

Grails application running at http://localhost:8080 in environment: development

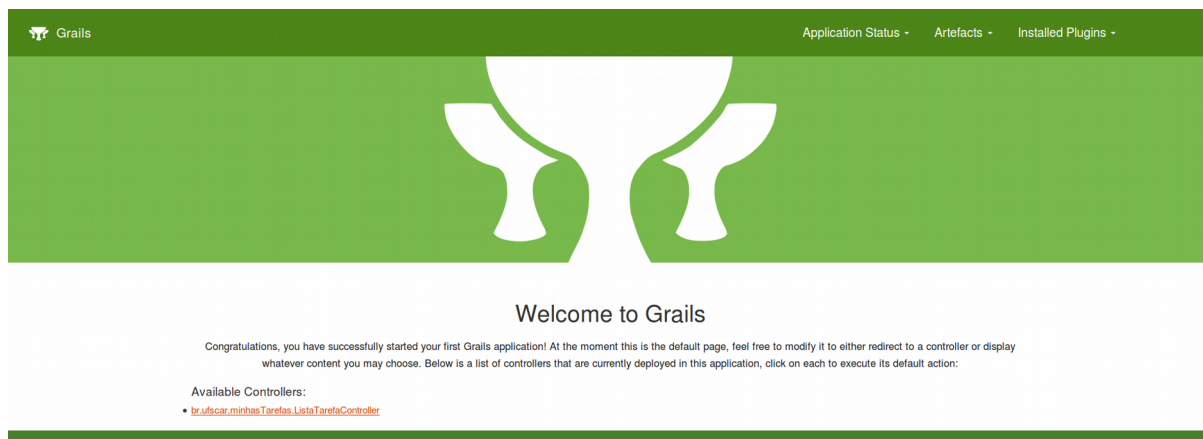


Figura 1.4: Tela da inicial da aplicação MinhasTarefas.

Ao entrar no endereço informado (<http://localhost:8080>) a tela apresentada na figura será exibida. Veremos que em “Available Controllers”, o nosso controller ListaTarefaController será apresentado. Ao clicarmos nele, veremos que teremos acesso à uma listagem, inicialmente vazia, de listas de tarefas (Figura 1.5):

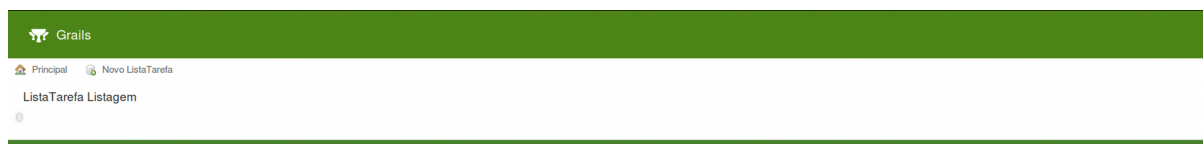
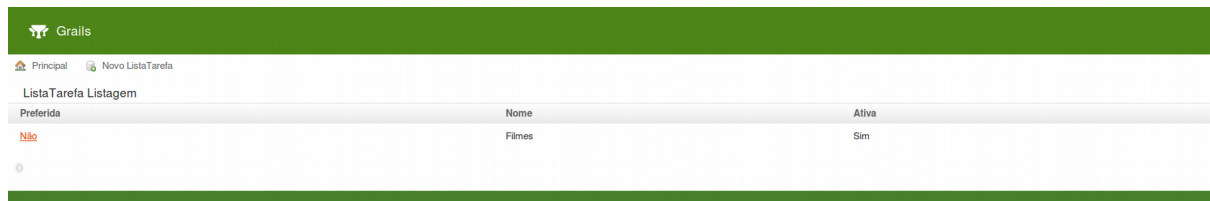


Figura 1.5: Tela de listagem das listas de tarefas, inicialmente ela se encontra vazia.

Ao clicar em “Nova ListaTarefa”, é aberta uma página (Figura 1.6) na qual é possível cadastrar uma nova lista de tarefas. Pode-se digitar um nome para a lista e clicar em “Criar”. A nova lista criada passa a ser exibida na listagem de listas de tarefas (Figura 1.7):

The screenshot shows a web application interface for creating a new task list. At the top, there is a green header bar with the Grails logo and the text 'Grails'. Below the header, there is a navigation bar with two links: 'Principal' and 'ListaTarefa Listagem'. The main content area is titled 'Criar ListaTarefa'. It contains a form with the following fields: 'Preferida' with a checkbox, 'Nome' with a text input field containing the value 'Filmes', and 'Ativa' with a checkbox. At the bottom of the form, there is a 'Criar' button. The form is set against a light gray background with a white border.

Figura 1.6: Tela de cadastro de uma nova lista de tarefas.



The screenshot shows a web application with a green header bar containing the Grails logo and the text 'Grails'. Below the header, there are two navigation links: 'Principal' and 'Novo ListaTarefa'. The main content area is titled 'ListaTarefa Listagem'. It contains a table with three columns: 'Preferida', 'Nome', and 'Ativa'. The 'Preferida' column has a single entry 'Não' in red text. The 'Nome' column has a single entry 'Filmes'. The 'Ativa' column has a single entry 'Sim'. Below the table, there is a small circular icon.

Preferida	Nome	Ativa
Não	Filmes	Sim

Figura 1.7: Listagem de tarefas agora apresenta uma única lista, que acabmos de criar.

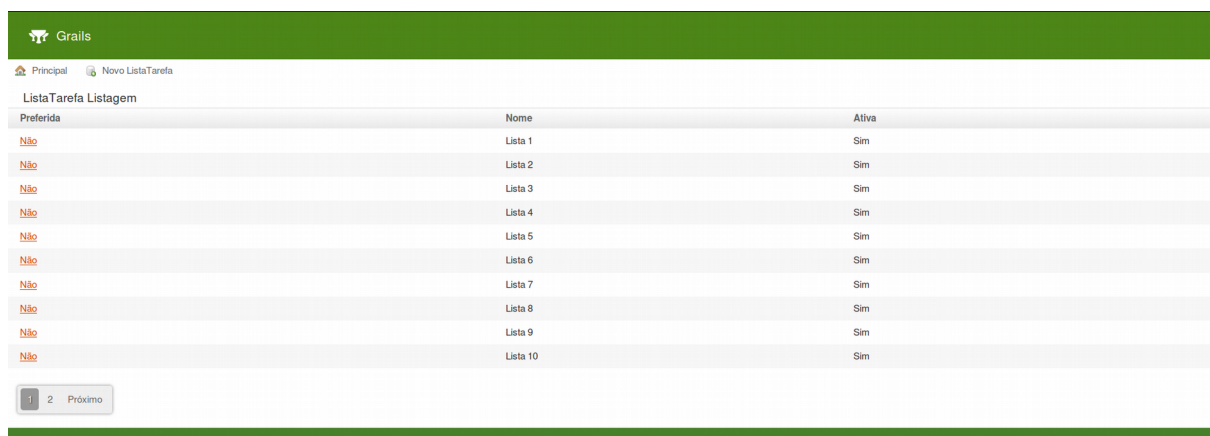
Se nós fecharmos nossa aplicação no terminal, apertando as teclas Ctrl e C simultaneamente, e abrirmos novamente a aplicação, veremos que a listagem estará novamente vazia. Isso ocorre porque por padrão o Grails apaga o banco de dados e o recria toda vez que iniciamos a aplicação. Um modo de garantir que o programa já inicialize com algumas listas pré-cadastradas é utilizar o arquivo `grails-app/init/Bootstrap.groovy`. Para isso, devemos alterar esse arquivo do seguinte modo (as adições de código encontram-se em negrito):

```
import br.ufscar.minhasTarefas.ListaTarefa

class Bootstrap {
    def init = { servletContext ->
        15.times { i ->
            ListaTarefa listaTarefa = new ListaTarefa(nome: "Lista ${i +
1}").save(failOnError: true)
        }
    }
    def destroy = {
    }
}
```

A clousure `init` da classe `Bootstrap` é executada toda vez que a aplicação é iniciada. Com isso, podemos utilizá-la para cadastrar elementos que queremos deixar já disponíveis no banco de dados. No código exibido, definimos que 15 vezes (`15.times`) vamos criar uma nova `ListaTarefa` (`new ListaTarefa`) com um nome que vai variar de acordo com a variável `i`, a qual é incrementada automaticamente. Além disso, estamos utilizando o método `save`. O método `save` é injetado automaticamente pelo Grails (GORM) em todos os objetos das classes de domínio, e serve para gravar instâncias no banco de dados. Como parâmetro dele, estamos dizendo que deve ser lançada uma exceção caso algum erro ocorra (`failOnError: true`).

Agora, a listagem vai apresentar 15 listas de tarefas, sendo 10 por página, conforme a Figura 1.8:



Preferida	Nome	Ativa
Não	Lista 1	Sim
Não	Lista 2	Sim
Não	Lista 3	Sim
Não	Lista 4	Sim
Não	Lista 5	Sim
Não	Lista 6	Sim
Não	Lista 7	Sim
Não	Lista 8	Sim
Não	Lista 9	Sim
Não	Lista 10	Sim

Figura 1.8: Listagem com diversas listas de tarefas criadas no `Bootstrap.groovy`.

Agora podemos criar nossa segunda classe de domínio (`Tarefa`), com o mesmo procedimento utilizado para criar a classe `ListaTarefa`. Após a criação da classe, vamos alterar o arquivo para que ele fique como a seguir:

```
package br.ufscar.minhasTarefas
class Tarefa {
    String nome
    Boolean concluida = false
    static belongsTo = ['lista': ListaTarefa]
```

```

        static constraints = {
        }
    }
}

```

A classe tarefa possui o campo nome, que será utilizado para identificar a tarefa, e o campo concluida, para determinar se a tarefa já foi realizada. Além disso, vemos uma estrutura nova: *static belongsTo*.

A variável belongsTo a qual é estática, ou seja, da classe e não do objeto, é o modo de informar ao Grails que nosso objeto pertence a um objeto de outra classe. No caso, ao objeto lista, da classe ListaTarefa. Desse modo, se uma lista à qual uma ou mais determinadas tarefas estiverem associadas for apagada, automaticamente o Grails vai remover todas as tarefas associadas a essa lista. Isso faz bastante sentido no nosso caso, pois não faz sentido termos uma tarefa se ela não estiver associada a uma lista.

Com o belongsTo, também garantimos a visibilidade necessária para atender o diagrama de classes (Figura 1.1), pois a partir de uma tarefa, conseguiremos chegar a nossa lista. Podemos observar esse comportamento ao gerar as views e controllers para a tarefa:

grails generate-all br.ufscar.minhasTarefas.Tarefa

Ao abrir na aplicação a página de create de uma tarefa (<http://localhost:8080/tarefa/create>) veremos que existe um campo Lista, que conterá como opções as listas que criamos no Bootstrap.groovy.

No entanto, para o belongsTo funcionar, é necessário que o objeto ListaTarefa saiba a quais tarefas ele está associado. Portanto, devemos alterar também a classe ListaTarefa, adicionando uma estrutura *hasMany*.

```

package br.ufscar.minhasTarefas
class ListaTarefa {
    String nome
    Boolean preferida = false
    Boolean ativa = true
    static hasMany = ['tarefas': Tarefa]
    static constraints = {
    }
}

```

A estrutura *hasMany* também é um campo estático e tem uma sintaxe parecida com a do *belongsTo*. No entanto, com ela, o Grails sabe que uma ListaTarefa “tem” zero ou mais tarefas. Com ela, obtemos a navegabilidade desejada, já que a partir da lista, conseguiremos chegar em todas as tarefas que ela possui. E como dito anteriormente,

agora é possível o *belongsTo* funcionar adequadamente. Portanto, podemos fazer o teste adicionando uma tarefa a alguma lista, e depois remover a lista. A tarefa também será removida.

Vamos adicionar no Bootstrap a criação de algumas tarefas para nossas listas:

```
import br.ufscar.minhasTarefas.ListaTarefa
import br.ufscar.minhasTarefas.Tarefa
class Bootstrap {
    def init = { servletContext ->
        criarListasComTarefas()
    }
    private criarListasComTarefas() {
        15.times { i ->
            ListaTarefa listaTarefa = new ListaTarefa(nome: "Lista ${i +
1}").save(failOnError: true)
            3.times { j ->
                new Tarefa(nome: "Tarefa ${j + 1}", lista:
listaTarefa).save(failOnError: true)
            }
        }
    }

    def destroy = {
    }
}
```

Como parâmetro na criação da tarefa passamos a lista a qual ela está associada. Muito simples. Agora que já vimos como criar classes de domínio e relacioná-las, podemos olhar como adaptar nossos controllers.

1.4 .Personalizando os controllers

Como dito anteriormente, os controllers tratam as chamadas das urls feitas pelo usuário através do navegador. Por exemplo, ao entrar pela aplicação na lista de tarefas, ou digitar o endereço <http://localhost:8080/tarefa/> o método index do controller TarefaController será chamado. Podemos personalizar a vontade os controllers criados pelo Grails, seja alterando métodos já existentes, quanto criando novos métodos.

Vamos inicialmente alterar o método index adicionando dois filtros nele, um para filtrar tarefas concluídas e outro para filtrar tarefas de uma determinada lista. O método index ficará como a seguir:

```
def index(Integer max) {
```

```

params.max = Math.min(max ?: 10, 100)
def idLista = params.long('idLista')
def concluidas = params.concluidas?params.boolean('concluidas'):null
def listaTarefas = Tarefa.withCriteria(params) {
    if (idLista != null) {
        eq('lista.id', idLista)
    }
    if (concluidas != null) {
        eq('concluida', concluidas)
    }
}

respond listaTarefas, model:[tarefaCount: listaTarefas.size,
listasDisponiveis: ListaTarefa.all,
filtroConcluidas: concluidas,
filtroLista: idLista]
}

```

As alterações estão em negrito. Primeiro passo é receber os parâmetros que o usuário vai informar, ou seja, o id da lista e se deve-se buscar tarefas concluídas ou não. Na url no browser, o usuário pode passar esses parâmetros do seguinte modo:

<http://localhost:8080/tarefa/index?concluida=true&idLista=7>

O parametro concluida pode ser true ou false, e o idLista deve ser o id de uma lista existente, caso contrário, nenhuma tarefa será retornada.

A obtenção desses parâmetros respectivamente nas variáveis concluidas e idLista é feito utilizando o params, que é uma variável disponibilizada pelo Grails com todos os parâmetros vindos da URL. Além disso, utilizamos params.long para transformar o parametro em Long, no caso do idLista, e params.boolean para transformar em Boolean no caso do concluidas.

Após ter esses dois parâmetros, fazemos a consulta no banco de dados de tarefas que atendam os dois critérios utilizando o Tarefa.withCriteria. Esse é um modo de fazer buscas que o Grails permite. Dentro da closure, verificamos se a variável idLista e concluidas possuem valores não nulos, e se esse for o caso, utilizamos o eq (de equal) para informar que queremos buscar no campo concluida, objetos com o valor da variável concluidas. Do mesmo modo, procuramos uma tarefa que tenha uma lista com o id igual a variável idLista.

Testando no browser, podemos verificar que o método retorna corretamente somente as tarefas que atendem aos parâmetros da consulta. É importante observar que vamos passar para a view alguns modelos:

filtroConcluidas: concluidas, filtroLista: idLista

Fazemos isso colocando essas variáveis no model, que é um mapa passado como parametro no método respond. O método respond serve para chamar uma view e passar um modelo para ela. Agora implementaremos na view esse filtro.

1.5 .Personalizando as views

Vamos mudar a nossa view index.gsp para apresentar os filtros que criamos. É bem simples, basta adicionar o código abaixo em negrito logo após a única tag H1 presente no arquivo:

```
<g:form>
  <g:select id="filtro-lista" name="idLista" from="${listasDisponiveis}"
    optionKey="id" optionValue="nome"
    noSelection="['': 'Selecione uma lista']"
    value="${filtroLista}"/>
  <g:select id="filtro-concluidas" name="concluidas" from="${
    [
      ['value': 'Abertas', key: false],
      ['value': 'Concluídas', key: true]
    ]
  }"
    optionKey="key" optionValue="value" value="${
{filtroConcluidas}"
    noSelection="['': 'Todas']"/>
  <g:submitButton id="filtrar" name="filtrar" value="Filtrar"/>
</g:form>
```

A tag do rails g:form define um formulário HTML. As tags g:select servem para criar selects html. O Atributo name dessas tags é o nome do parâmetro que chegará ao controller, por isso eles devem ser iguais fizemos no controller (idLista e concluidas). O atributo from define de qual variável virão os itens da lista exibidos no select. No caso do concluidas, nós criamos um mapa na própria view, com as opções. Já no caso das listas, os valores possíveis vieram do controller.

Com os atributos optionValue e optionKey, definimos respectivamente o atributo da lista que possui o texto que será apresentado no select e o valor que será enviado para o controller. Por fim, com o noSelection definimos o texto que será apresentado caso o usuário não queira selecionar nenhuma das opções disponíveis

A última tag, `g:submitButton` cria um botão com o texto definido em `Value`, e serve para enviar o form HTML. Podemos rodar nossa aplicação e ver que tudo está funcionando adequadamente.

1.6 .Configurando um banco Postgres

O próximo passo para nossa aplicação é fazer ela utilizar um banco de dados específico. Vamos utilizar como exemplo um banco de dados Postgres. No Grails é muito simples a configuração de um novo banco de dados. Basta incluir a dependência do driver do banco em questão (em **negrito**) na seção `dependencies` no arquivo `build.gradle`:

```
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"  
    console "org.grails:grails-console"  
    profile "org.grails.profiles:web:3.1.6"  
    runtime "com.bertramlabs.plugins:asset-pipeline-grails:2.8.2"  
    runtime "com.h2database:h2"  
    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"  
    compile "org.postgresql:postgresql:9.4-1201-jdbc41"  
}
```

E depois configurar o nome da classe do driver do banco utilizado, o usuário, a senha, e o endereço do banco no arquivo `application.yml`, na seção `datasource`, que fica localizada em `environments, development`:

```
development:  
  dataSource:  
    dbCreate: create-drop
```

```
url: jdbc:postgresql://localhost:5432/minhasTarefas
driverClassName: org.postgresql.Driver
username: postgres
password: postgres
```

Basta agora parar a aplicação e rodá-la novamente para que o banco usado seja o do Postgres. No próximo capítulo veremos como adicionar uma camada de segurança para nossa aplicação, com criação de usuários e papéis. Além disso veremos como melhorar nossas *views* para adaptá-las ao Spring Security Core.

2 . Spring Security - Como tornar sua aplicação segura

O Spring Security¹ é um framework cujo foco é a prover a autorização e autenticação de usuários para aplicações Java. Ele é muito utilizado no mundo Java, devido sua flexibilidade para ser customizado de acordo com a necessidade do projeto. Nesse capítulo, iremos ver como incorporá-lo no nosso projeto Grails (Seção 2.1), usando o Grails Spring Security Core Plugin².

Para facilitar a adição de novos usuários, e o gerenciamento do acesso aos mesmos, vamos utilizar o Spring Security UI Plugin³. Veremos como utilizá-lo na Seção 2.2

2.1 . Configurando o Spring Security Core

O primeiro passo para utilizarmos o Spring Security Core é adicionar a dependência do plugin no arquivo `build.gradle`. Para isso, é só repetirmos o que fizemos no caso do plugin do Postgres, adicionando a dependência na seção `dependencies`. Essa seção agora ficará assim (com o destaque para o Spring Security Core):

```
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"
```

1 <http://projects.spring.io/spring-security/>

2 <https://github.com/grails-plugins/grails-spring-security-core>

3 <https://github.com/grails-plugins/grails-spring-security-ui/>

```

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"
console "org.grails:grails-console"
profile "org.grails.profiles:web:3.1.6"
runtime "com.bertramlabs.plugins:asset-pipeline-grails:2.8.2"
runtime "com.h2database:h2"
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"
compile 'org.postgresql:postgresql:9.4-1201-jdbc41'
compile 'org.grails.plugins:spring-security-core:3.0.4'
}

```

No terminal, devemos digitar:

grails compile

Com isso nossa aplicação vai baixar a dependência do Spring Security Core, e vai permitir que utilizemos um *script* disponível no plugin. Esse script deve ser chamado assim como o *create-app* e *create-domain-class* (que também são scripts Grails). Ou seja, vamos digitar no seguinte formato:

grails nome_do_script <parametros>

No caso do plugin do Spring Security, vamos usar o *script* s2-quickstart. Esse script tem como função criar as classes de domínio que serão usadas para persistir os usuários do nosso sistema, as permissões (ou *roles*) existentes, e o relacionamento entre os usuários e as permissões que eles possuem. Dado que vamos querer ter as classes *Usuario*, e *Permissao*, respectivamente para armazenar os usuários e as permissões, a chamada ao script ficará assim:

grails s2-quickstart br.ufscar.minhasTarefas.seguranca Usuario Permissao

O primeiro parâmetro (br.ufscar.minhasTarefas.seguranca) é o nome do pacote onde as classes serão colocadas. O segundo parâmetro, é o nome da classe que vai representar

os usuários do sistema, e o último parâmetro é o nome da classe que representará as permissões.

Após a execução do script, serão criadas três classes no pacote `br.ufscar.minhasTarefas.seguranca`: `Usuario`, `Permissao`, e `UsuarioPermissao`. Como mencionado anteriormente, a classe `Usuario` representa os usuários do sistema, a classe `Permissao` representa as permissões disponíveis, e a classe `UsuarioPermissao` é um *hasMany* que relaciona usuários às permissões que eles têm.

Não precisaremos por enquanto alterar essas classes, sendo assim vamos nos ater a mais um arquivo criado pelo *script* do Spring Security. O arquivo `application.groovy`, reproduzido abaixo, é equivalente ao `application.yml`, mas utiliza uma sintaxe *groovy* para fazer as configurações do projeto.

```
// Added by the Spring Security Core plugin:
grails.plugin.springsecurity.userLookup.userDomainClassName =
'br.ufscar.minhasTarefas.seguranca.Usuario'
grails.plugin.springsecurity.userLookup.authorityJoinClassName =
'br.ufscar.minhasTarefas.seguranca.UsuarioPermissao'
grails.plugin.springsecurity.authority.className =
'br.ufscar.minhasTarefas.seguranca.Permissao'
grails.plugin.springsecurity.controllerAnnotations.staticRules = [
    [pattern: '/',          access: ['permitAll']],
    [pattern: '/error',     access: ['permitAll']],
    [pattern: '/index',     access: ['permitAll']],
    [pattern: '/index.gsp', access: ['permitAll']],
    [pattern: '/shutdown',  access: ['permitAll']],
    [pattern: '/assets/**', access: ['permitAll']],
    [pattern: '/*/js/**',   access: ['permitAll']],
    [pattern: '/*/css/**',  access: ['permitAll']],
    [pattern: '/*/images/**', access: ['permitAll']],
    [pattern: '/*/favicon.ico', access: ['permitAll']]
]
grails.plugin.springsecurity.filterChain.chainMap = [
    [pattern: '/assets/**',   filters: 'none'],
    [pattern: '/*/js/**',     filters: 'none'],
    [pattern: '/*/css/**',    filters: 'none'],
    [pattern: '/*/images/**', filters: 'none'],
    [pattern: '/*/favicon.ico', filters: 'none'],
    [pattern: '/*',           filters: 'JOINED_FILTERS']
]
```

Nesse arquivo vemos que o plugin é configurado para utilizar as três classes criadas. Além disso, temos a propriedade `staticRules`, que está sendo utilizada para permitir o acesso (*permitAll*) a algumas páginas, como o `index.gsp` por exemplo. A outra propriedade *chainMap* é utilizada para liberar o acesso aos recursos web como javascripts, css e imagens.

Ao rodar a aplicação vemos que existem dois novos *controllers* (`LoginController` e `LogoutController`). Eles são utilizados respectivamente para logar e deslogar o usuário. Se tentarmos acessar nossas listas, veremos que o nosso acesso está bloqueado e seremos redirecionados para uma tela de login, exibida na Figura 2.1.

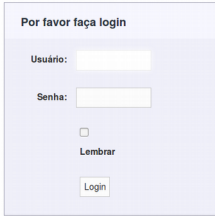


Figura 2.1: Tela de login do Spring Security Core.

Nós não temos nenhum usuário cadastrado, portanto não conseguiremos acessar as páginas de listas. Isso ocorre porque por padrão o Spring Security Core bloqueia o acesso a qualquer método que não tenha uma permissão definida. Para criarmos nossos usuários, vamos utilizar o `Bootstrap.groovy` novamente. Ele ficará assim (com alterações em negrito):

```
import br.ufscar.minhasTarefas.ListaTarefa
import br.ufscar.minhasTarefas.Tarefa
import br.ufscar.minhasTarefas.seguranca.Permissao
import br.ufscar.minhasTarefas.seguranca.Usuario
import br.ufscar.minhasTarefas.seguranca.UsuarioPermissao
class Bootstrap {
```

```

def init = { servletContext ->
    criarUsuariosAutorizados()
    criarListasComTarefas()
}

private void criarUsuariosAutorizados() {
    Permissao permissaoAdministrador = new Permissao(authority:
'ROLE_ADMIN').save(failOnError: true)
    Usuario usuarioAdministrador = new Usuario(username: 'admin',
password: 'root').save(failOnError: true)
    UsuarioPermissao autorizacaoAdministrador = new
UsuarioPermissao(usuario: usuarioAdministrador,
                    permissao: permissaoAdministrador).save(failOnError: true)
    Permissao permissaoGerenciarListas = new Permissao(authority:
'ROLE_GERENCIAR_LISTAS').save(failOnError: true)
    Usuario usuarioNormal = new Usuario(username: 'usuario', password:
'normal').save(failOnError: true)
    UsuarioPermissao autorizacaoGerenciaListas = new
UsuarioPermissao(usuario: usuarioNormal,
                    permissao: permissaoGerenciarListas).save(failOnError:
true)
}

private criarListasComTarefas() {
    15.times { i ->
        ListaTarefa listaTarefa = new ListaTarefa(nome: "Lista ${i +
1}").save(failOnError: true)
        3.times { j ->
            new Tarefa(nome: "Tarefa ${j + 1}", lista:
listaTarefa).save(failOnError: true)
        }
    }
}

def destroy = {
}
}

```

Adicionamos duas permissões, uma de administrador (*ROLE_ADMIN*) e uma de usuários normais para gerenciar as listas (*ROLE_GERENCIAR_LISTAS*).

Além disso, vamos utilizar *annotations* do Spring Security para permitir o acesso aos controllers *ListaTarefa* e *Tarefa*. *Annotations* permitem que os metadados de uma classe sejam colocados dentro da própria classe, ao invés de utilizar um arquivo de configuração como XML.

Vamos definir que somente usuários com a permissão `ROLE_GERENCIAR_LISTAS` possam cadastrar listas e tarefas. Sendo assim, vamos alterar os controllers `ListaTarefa` e `Tarefa`. O controller `ListaTarefaController` ficará assim(omitidos os códigos dos controllers):

```
package br.ufscar.minhasTarefas

import static org.springframework.http.HttpStatus.*
import grails.transaction.Transactional
import grails.plugin.springsecurity.annotation.Secured
@Transactional(readOnly = true)
@Secured(['ROLE_GERENCIAR_LISTAS'])
class ListaTarefaController {...}
```

Já o `TarefaController` ficará assim:

```
package br.ufscar.minhasTarefas

import static org.springframework.http.HttpStatus.*
import grails.transaction.Transactional
import grails.plugin.springsecurity.annotation.Secured
@Transactional(readOnly = true)
@Secured(['ROLE_GERENCIAR_LISTAS'])
class TarefaController {...}
```

Agora ao tentar acessar esses controllers com o usuário *admin*, seremos informados que não temos permissão, já com o usuário *usuario* teremos acesso as listas. É importante dizer que poderíamos utilizar annotations diretamente nos métodos. Ou seja, dentro de um *controller*, um método poderia exigir uma permissão, enquanto outro método poderia exigir outra.

Também é importante ressaltar que o Spring Security tem algumas permissões pré-definidas, listadas na tabela Tabela 1.1 com seus respectivos significados:

Permissão	Significado
IS_AUTHENTICATED_ANONYMOUSLY	Qualquer um pode acessar a página, mesmo que não tenha logado no sistema
IS_AUTHENTICATED_REMEMBERED	Qualquer usuário logado, seja explicitamente (digitando login e senha) ou por meio de um <i>cookie</i> de “manter logado”.
IS_AUTHENTICATED_FULLY	Somente usuário logado explicitamente

Tabela 2.1: Permissões padrão do Spring Security e seus significados.

É necessário agora permitir que o usuário consiga deslogar da aplicação. Para isso, vamos alterar nossas views para que o usuário consiga fazer o *logout*. O Grails utiliza o mesmo layout para todas as nossas views. Esse *layout* está definido no arquivo `main.gsp` localizado em `views/layout/main.gsp`. Vamos usar algumas *tags* do Spring Security Core para personalizar as views da aplicação.

A intenção é apresentar o *link* para o usuário conseguir fazer o login, caso ainda não esteja logado. Se o usuário estiver logado, vamos apresentar o nome dele e dar a opção dele deslogar. Para isso vamos acrescentar o seguinte código em negrito no nosso arquivo `main.gsp`:

```
<ul class="nav navbar-nav navbar-right">
  <g:pageProperty name="page.nav" />
  <li class="dropdown">
    <sec:ifNotLoggedIn>
      <g:link controller="login">Login</g:link>
    </sec:ifNotLoggedIn>
    <sec:ifLoggedIn>
      <a href="#" class="dropdown-toggle" data-toggle="dropdown"
role="button" aria-haspopup="true" aria-expanded="false">
        Usuário: <sec:username />
        <span class="caret"></span> </a>
      <ul class="dropdown-menu">
        <li>
          <g:link controller="logout">Logout</g:link>
        </li>
      </ul>
    </sec:ifLoggedIn>
  </li>
</ul>
```

A tag `sec:ifNotLoggedIn` faz com que o código existente dentro dela somente seja processado caso não exista usuário logado na aplicação. Já a tag `sec:ifLoggedIn` faz exatamente o contrário. O código dentro dela somente será processado caso exista um usuário logado.

Além dessas tags do Spring Security, usamos a tag `g:link` do Grails, para criar um *link*. Definimos o *controller* Logout como parâmetro. Com isso o Grails vai criar um link relativo ao nome da aplicação, do seguinte modo:

localhost:8080/<nome_controller>/<nome_ação_padrão>

No nosso exemplo, o link fica:

http://localhost:8080/logout/index

Dado que a ação padrão dos controllers no Grails é a *index*. No entanto, devemos configurar o *plugin* Spring Security Core para permitir *logout* com o método **HTTP GET**, dado que por padrão ele só aceita o método **POST**. Para alterar essa configuração basta adicionar o seguinte trecho de código no arquivo `application.groovy`:

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

Na próxima seção veremos como instalar o Spring Security UI.

2.2 . Configurando o Spring Security UI

O Spring Security UI é um plugin que adiciona algumas views e controllers para a aplicação, para permitir o gerenciamento de usuários e permissões. Com ele fica muito fácil criar e apagar usuários e permissões, bem como atribuir ou revogar permissões para os usuários.

Assim como nos casos anteriores, é muito simples adicionar o plugin na nossa aplicação. Basta acrescentar a seguinte linha na seção dependencies do `build.gradle`:

```
compile 'org.grails.plugins:spring-security-ui:3.0.0.M2'
```

Com o *plugin* adicionado, veremos que alguns *controllers* serão adicionados, no entanto não teremos acesso a eles. Vamos liberá-lo para acesso a usuários com a permissão "ROLE_ADMIN". Para isso, vamos alterar a propriedade `staticRules` do arquivo `application.groovy` para que fique assim:

```
grails.plugin.springsecurity.controllerAnnotations.staticRules = [  
    [pattern: '/',          access: ['permitAll']],  
    [pattern: '/error',     access: ['permitAll']],  
    [pattern: '/index',     access: ['permitAll']],  
    [pattern: '/index.gsp', access: ['permitAll']],  
    [pattern: '/shutdown',  access: ['permitAll']],  
    [pattern: '/assets/**', access: ['permitAll']],  
    [pattern: '/*/*/js/**', access: ['permitAll']],  
    [pattern: '/*/*/css/**', access: ['permitAll']],  
    [pattern: '/*/*/images/**', access: ['permitAll']],  
    [pattern: '/*/*/favicon.ico', access: ['permitAll']],  
    [pattern: '/role/**',   access: ['ROLE_ADMIN']],  
]
```

```

    [pattern: '/user/**',          access: ['ROLE_ADMIN']],
    [pattern: '/securityInfo/**',access: ['ROLE_ADMIN']]
]

```

Com essas alterações, nosso usuário admin já terá acesso as funcionalidades de cadastro de usuários e permissões.

2.3 . Utilizando as classes do Spring Security nas nossas classes de domínio

Agora que já temos todo o controle de permissões de usuários implementado, vamos refinar nossas classes de domínio, para que nossos usuários possam ver somente suas listas e tarefas. Para isso ser possível, vamos incluir uma relação com a classe Usuario tanto na lista como na tarefa. Nossas classes de domínio ficarão assim:

```

package br.ufscar.minhasTarefas
import br.ufscar.minhasTarefas.seguranca.Usuario
class ListaTarefa {
    String nome
    Boolean preferida = false
    Boolean ativa = true
    Usuario usuario
    static hasMany = ['tarefas': Tarefa]
    static constraints = {
    }
}

package br.ufscar.minhasTarefas
import br.ufscar.minhasTarefas.seguranca.Usuario
class Tarefa {
    String nome
    Boolean concluida = false
    Usuario usuario
    static belongsTo = ['lista': ListaTarefa]
    static constraints = {
        nome()
    }
}

```

Agora vamos alterar nosso BootStrap para definir o usuário dono da lista.

```

import br.ufscar.minhasTarefas.ListaTarefa
import br.ufscar.minhasTarefas.Tarefa
import br.ufscar.minhasTarefas.seguranca.Permissao
import br.ufscar.minhasTarefas.seguranca.Usuario
import br.ufscar.minhasTarefas.seguranca.UsuarioPermissao
class BootStrap {
  def init = { servletContext ->
    def usuarios = criarUsuariosAutorizados()
    criarListasComTarefas(usuarios.usuarioNormal)
  }
  private def criarUsuariosAutorizados() {
    Permissao permissaoAdministrador = new Permissao(authority:
'ROLE_ADMIN').save(failOnError: true)
    Usuario usuarioAdministrador = new Usuario(username: 'admin',
password: 'root').save(failOnError: true)
    UsuarioPermissao autorizacaoAdministrador = new
UsuarioPermissao(usuario: usuarioAdministrador,
    permissao: permissaoAdministrador).save(failOnError: true)
    Permissao permissaoGerenciarListas = new Permissao(authority:
'ROLE_GERENCIAR_LISTAS').save(failOnError: true)
    Usuario usuarioNormal = new Usuario(username: 'usuario', password:
'normal').save(failOnError: true)
    UsuarioPermissao autorizacaoGerenciaListas = new
UsuarioPermissao(usuario: usuarioNormal,
    permissao: permissaoGerenciarListas).save(failOnError:
true)
    return [usuarioNormal: usuarioNormal, usuarioAdmin:
usuarioAdministrador]
  }
  private criarListasComTarefas(Usuario usuario) {
    15.times { i ->
      ListaTarefa listaTarefa = new ListaTarefa(nome: "Lista ${i +
1}", usuario: usuario).save(failOnError: true)
      3.times { j ->
        new Tarefa(nome: "Tarefa ${j + 1}", lista: listaTarefa,
usuario: usuario).save(failOnError: true)
      }
    }
  }
  def destroy = {
  }
}

```

É importante observar que agora o método `criarUsuariosAutorizados` retorna um mapa, com duas chaves: `usuarioAdmin` e `usuarioNormal`. Cada uma delas possui respectivamente o usuário esperado. Usamos o `usuarioNormal` como parâmetro do método `criarListasComTarefas`, e no construtor da classe `ListaTarefa` e da `Tarefa` passamos ele como parâmetro, associando assim as listas e as tarefas a esse usuário.

Agora para garantir que um usuário veja somente as listas e tarefas, vamos alterar os controllers dessas duas classes para que seja considerado o usuário na hora de filtrar. Como já vimos, o método `index` é o responsável por apresentar as listas. Sendo assim, teremos os *controllers* `ListaTarefaController` e `TarefaController` alterados conforme o código abaixo:

```
package br.ufscar.minhasTarefas
import br.ufscar.minhasTarefas.seguranca.Usuario
import static org.springframework.http.HttpStatus.*
import grails.transaction.Transactional
import grails.plugin.springsecurity.annotation.Secured
@Transactional(readOnly = true)
@Secured(['ROLE_GERENCIAR_LISTAS'])
class ListaTarefaController {
    static allowedMethods = [save: "POST", update: "PUT", delete: "DELETE"]
    def springSecurityService
    def index(Integer max) {
        params.max = Math.min(max ?: 10, 100)
        Usuario usuario = springSecurityService.currentUser
        respond ListaTarefa.findAllByUsuario(usuario, params), model:
[listaTarefaCount: ListaTarefa.countByUsuario(usuario, params)]
    }

    ...

}
```

```

package br.ufscar.minhasTarefas

import br.ufscar.minhasTarefas.seguranca.Usuario
import static org.springframework.http.HttpStatus.*
import grails.transaction.Transactional
import grails.plugin.springsecurity.annotation.Secured
@Transactional(readOnly = true)
@Secured(['ROLE_GERENCIAR_LISTAS'])
class TarefaController {
    static allowedMethods = [save: "POST", update: "PUT", delete: "DELETE"]
    def springSecurityService
    def index(Integer max) {
        params.max = Math.min(max ?: 10, 100)
        def idLista = params.long('idLista')
        def concluidas = params.concluidas?
        params.boolean('concluidas'):null
        Usuario usuario = springSecurityService.currentUser
        def listaTarefas = Tarefa.withCriteria(params) {
            if (idLista != null) {
                eq('lista.id', idLista)
            }
            if (concluidas != null) {
                eq('concluida', concluidas)
            }
            eq('usuario', usuario)
        }
        respond listaTarefas, model:[tarefaCount: listaTarefas.size,
listasDisponiveis: ListaTarefa.all,
filtroConcluidas: concluidas,
filtroLista: idLista]
    }
}

```

Podemos criar pela própria aplicação um novo usuário, adicionando a permissão de “ROLE_GERENCIAR_LISTAS”, e veremos que ele não conseguirá ver nenhuma lista, a menos que crie alguma. Para conseguir isso, injetamos o serviço *springSecurityService* provido pelo Spring Security Core. Depois pegamos o usuário logado com o método *currentUser* e por último utilizamos o usuário para fazer o filtro, com o método *findAllByUsuario* no caso das listas, e no *withCriteria* no caso das tarefas.

Agora vamos filtrar para que quando o usuário acesse a nossa aplicação, seja redirecionado para o login. Ao logar, caso seja administrador, o usuário será levado para a

página de gerenciamento de usuários. Se for usuário normal, será redirecionado para a listagem de listas de tarefas. Vamos alterar nosso `UrlMappings.groovy`, para que ao acessar a raiz do site (<http://localhost:8080>), ele seja redirecionado para um controller específico, o `MainController`. O `UrlMappings` é utilizado para configurar as rotas de nossa aplicação, ou seja, qual controller ou view vai atender um determinado endereço. Ele deverá ficar assim:

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

Agora o "/" (raiz da aplicação) redirecionará para o controller `MainController` (que ainda vamos criar). Esse controller deverá ser criado com o comando:

grails create-controller br.ufscar.minhasTarefas.MainController

O código deverá ficar como a seguir:

```
package br.ufscar.minhasTarefas
import grails.plugin.springsecurity.annotation.Secured
@Secured("IS_AUTHENTICATED_REMEMBERED")
class MainController {
    def springSecurityService
    def index() {
        def usuario = springSecurityService.currentUser
        def authority = usuario.getAuthorities()[0].getAuthority()
        if (authority.equals('ROLE_ADMIN')) {
            redirect(controller: 'user', action: 'search')
        } else if (authority.equals('ROLE_GERENCIAR_LISTAS')) {
            redirect(controller: 'listaTarefa')
        }
    }
}
```

Podemos observar que para acessar os métodos desse controller, o usuário deve estar ao menos logado na aplicação. Portanto, inicialmente ele será redirecionado pelo Spring Security Core para a página de login. Após fazer o login, vamos verificar as permissões que ele tem (authorities). Para isso usamos o método do Spring Security Core *currentUser*, e da classe de domínio Usuario *getAuthorities*. Depois com o *if*, verificamos qual tipo de permissão ele tem, e redirecionamos (método *redirect*) para o *controller* e *action* desejados. Com isso vimos o básico de como garantir a segurança de nossa aplicação