

Desenvolvendo na prática com Spring e testes

Curso FJ-22



Sumário

1 O medo de se adaptar	1
1.1 Caindo de paraquedas em um projeto	1
1.2 Controle de versão com Git	3
1.3 Exercício - Iniciando com Git	5
1.4 Serviços de repositório de código	6
1.5 Exercício Opcional - Criando uma conta no Github	12
1.6 Utilizando GitHub	16
1.7 Exercício - Fork e Clone do projeto	19
1.8 Lidando com build e dependências	20
1.9 Exercício - Usando gerenciador de dependência e build	23
1.10 Exercício - Rodando projeto no Eclipse com maven	24
2 Trabalhando com Branches	28
2.1 Juntando commits de outra branch	29
2.2 Exercício - Criando nossas branches	33
2.3 Começando a trabalhar no Sistema	35
2.4 Exercício - Adicionando a sessão ao sistema	39
2.5 Analisando a Regra de Negócio	43
2.6 Testes de Unidade	46
2.7 JUnit	47
2.8 Fazendo nosso primeiro teste	48
2.9 Exercício - Garantindo que a validação de horários para cadastrar uma sessão está correta	51
3 Adicionando Preço	54
3.1 Melhorando a modelagem do Sistema	54
3.2 Exercício - Colocando preço na Sala e Filme	55
3.3 Aplicando Strategy	58
3.4 Exercício - Criando descontos e ingresso	59
3.5 Exercício - Testando um Desconto	61
3.6 Exercício Opcional - Testando os demais Descontos	61

4 Melhorando a Usabilidade da Aplicação	63
4.1 Definindo o catálogo de filmes e a tela de detalhes	63
4.2 Exercício - Criando o catálogo de filmes e tela de detalhes do filme com sessões para compra	65
4.3 Trazendo dados reais para nossa aplicação	66
4.4 Exercício - Consumindo serviço para detalhes do filme	69
5 Iniciando o processo de Venda	72
5.1 Criando tela para escolha de lugar	72
5.2 Exercício - Criando tela para seleção de lugares	74
5.3 Selecionando Ingresso	76
5.4 Exercício - Implementando a seleção de lugares, ingressos e tipo de ingressos.	77
5.5 Exercícios Opcionais - Testando a verificação de lugares disponíveis	81
6 Criando o Carrinho	82
6.1 O Carrinho de compras	82
6.2 Exercício - Implementando a tela de compras	83
6.3 Melhorando a usabilidade do carrinho	85
6.4 Exercício - Desabilitando a seleção do lugar que já está no carrinho	86
6.5 Exibindo os detalhes da compra	86
6.6 Exercício - Implementando a tela de checkout	87
6.7 Realizando a compra	88
6.8 Exercícios - Implementando a compra	90
7 Apêndice: Implementando Segurança	93
7.1 Protegendo nossas URIs	93
7.2 Configurando Spring Security	93
7.3 Exercício - Implementando segurança em nossa aplicação	98
7.4 Usuário, Senha e Permissão	100
7.5 Exercício - Implementando UserDetails, UserDetailsService e GrantedAuthority	103
8 Apêndice: Criando uma nova Conta	107
8.1 Implementando Segurança em nosso Sistema	107
8.2 Exercício - Criando formulário de solicitação de acesso	109
8.3 Configurações para envio de e-mails	111
8.4 Exercício - Configurando o envio de e-mails	114
8.5 Salvando token e enviando e-mail	115
8.6 Exercício - Enviando e-mail	116
8.7 Implementando validação	118
8.8 Exercício - Validando token	120
8.9 Persistindo o usuário	122

O MEDO DE SE ADAPTAR

Ao desenvolver projetos é comum que precisemos nos adaptar a situações novas e, assim, modificar o sistema para melhorar as lógicas já existentes ou implementar uma funcionalidade nova.

Mas a própria palavra *adaptação* já nos dá um sentimento ruim sobre o que vai acontecer, pois muitas vezes a ligamos com "fazer gambiarra" em vez de pensar em ações limpas e evolutivas.

Outra palavra que causa desconforto é *mudança*. É corriqueiro pensarmos em mudança como perda, seja de trabalho, de privacidade, ou mesmo de alguns trechos de código que penamos para construir no passado e dos quais nos orgulhamos.

Contudo, mudanças e adaptações são necessárias e naturais durante a vida de qualquer projeto. Resistir a elas é fonte de diversos problemas que hoje vemos acontecendo em grande parte das empresas de desenvolvimento de software, como funcionalidades com falhas, adiamento de atualizações e criação de novas lógicas ao invés de alterar as já existentes.

1.1 CAINDO DE PARAQUEDAS EM UM PROJETO

Quando entramos em um projeto, é comum que tenhamos medo de mexer em alguma coisa e quebrar dezenas de outras. Mas por que temos esse medo de quebrar código já existente? Porque, dependendo da mudança que efetuamos no código, fica bem difícil voltar ao estado anterior.

Não é de hoje que esse problema existe, por isso, já há diversas formas de contorná-lo. Por exemplo:

- Manter um backup do arquivo (localmente ou remotamente);
- Manter o código antigo comentado antes de uma alteração.

O problema com essas abordagens é que elas são todas muito manuais, além do que na segunda abordagem ainda corremos o risco de alguém sem querer tirar o comentário do código e termos comportamentos inesperados. Código comentado é um exemplo do que chamamos de "Code Smells" ou, na tradução mais frequente, "Maus cheiros de código". Os *Code Smells* são sintomas de más práticas e devem ser tratados assim que possível.

BIBLIOGRAFIA SOBRE CODE SMELLS

Se você gostaria de estudar mais sobre os chamados *Code Smells*, há três livros bastante conhecidos onde há explicações ou catálogos sobre tais sintomas. São eles:

- Refactoring - Martin Fowler
- xUnit Testing Patterns - Gerard Meszaros
- Clean Code - Robert Martin

Todas essas soluções são para resolver o mesmo problema: versionar um arquivo. Pensando nisso, foram criadas ferramentas para controle de versões. Dentre elas, algumas ganharam bastante destaque, como, por exemplo:

- **CVS** - um dos primeiros a popularizar o uso de controle de versões em projetos open source, foi criado em 1990 e controla cada arquivo individualmente. Cada arquivo tem um número de versão de acordo com o número de vezes que foi modificado. Seus commits são lentos e trabalhar com branches é algo custoso.
- **SVN** - dez anos mais tarde, a ideia de manter uma versão para cada arquivo já não parecia tão boa. A principal diferença entre o CVS e o SVN é que no último as versões são por atualização do repositório como um todo em vez de por arquivos. Trabalhar com branches se torna mais simples, mas elas ainda são meras cópias do repositório.
- **SourceSafe** - a solução adotada pela Microsoft para controle de versão surgiu em 1994 e usualmente é utilizada com "lock de arquivos", isto é, enquanto uma pessoa altera um arquivo, as outras não têm acesso a ele. Essa foi uma estratégia para minimizar conflitos.

Todas as ferramentas citadas acima estão na categoria de *controle de versão centralizado*, ou seja, é necessário um lugar único para o servidor que irá controlar as versões e cada cliente terá que se conectar a esse servidor.

Há também ferramentas de *controle de versão distribuído*, em que cada cliente tem seu próprio repositório. Abaixo estão algumas que se destacaram bastante, todas criadas em 2005:

- **Git** - criado por Linus Torvalds, criador do *Linux*, para controlar a versão do kernel do seu sistema operacional, pois a ferramenta que estava sendo utilizada não estava atendendo a requisitos como, por exemplo, o de performance. O *Git* é largamente adotado por muitas comunidades de desenvolvedores e, por isso, será a ferramenta que utilizaremos durante o curso.
- **Mercurial** - o Mercurial, ou Hg, é bastante utilizado pela comunidade de Python e provê uma

transição mais suave do que o Git para os ex-usuários de SVN e CVS por utilizar os mesmos nomes de comandos.

- **Bazaar** - criado pela empresa mantenedora do Ubuntu, também utiliza os mesmos comandos dos sistemas mais tradicionais. Foi uma migração natural para os projetos open source hospedados no SourceForge, mas atualmente está perdendo força.

1.2 CONTROLE DE VERSÃO COM GIT

O Git é um sistema de controle de versão com diversas características que o tornam interessante para o trabalho no mercado atual. Veremos elas durante esse curso.

Uma das características importantes é a de poder comitar nossas alterações na máquina local, mesmo que estejamos sem conexão com a internet ou com a rede interna onde está o repositório Git. Podemos só sincronizar com o repositório central no momento que considerarmos adequado.

Há duas formas para começarmos a trabalhar com o Git: fazer uma cópia de um repositório já existente na nossa máquina, processo conhecido por **clone**,

```
$ git clone git://url.do.repositorio
```

ou criar um repositório local.

```
$ git init
```

Agora podemos alterar ou adicionar novos arquivos ao repositório.

Após fazer isso, podemos usar o comando `git status` para ver quais arquivos foram modificados:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   arquivo.que.sera.comitado
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   arquivo.modificado.mas.que.nao.sera.comitado
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       arquivo.que.o.git.nao.está.gerenciando.ainda
```

Essa resposta diz qual é o status do seu repositório local e o que você pode fazer com as alterações. Os arquivos novos, que ainda não existem no repositório, aparecem na seção *Untracked files* e os já existentes, mas com alteração, aparecem em *Changes not staged for commit*. O processo de enviar mudanças ou criações de arquivos ao repositório é conhecido por *commit*, mas por padrão o Git só

comitará os arquivos que estiverem na seção *Changes to be committed*. Devemos então usar o comando `git add` passando o caminho de todos os arquivos que queremos adicionar ao próximo *commit*, tanto os já existentes quanto os novos, como o Git está sugerindo:

```
$ git add arquivo.que.o.git.nao.está.gerenciando.ainda arquivo.modificado.mas.que.nao.sera.comita  
do
```

Podemos rodar o `git status` novamente e verificaremos que todos os arquivos estarão na seção *Changes to be committed*. Podemos então efetuar o *commit* já informando uma mensagem que descreva o que foi feito.

```
$ git commit -m "Incluindo testes novos para o sistema de pagamento via cartão."
```

Quando a intenção for a de adicionar todos os arquivos que estão em *Changes not staged for commit*, existe um atalho:

```
$ git commit -a -m "Efetuando um commit local com todos os arquivos de uma única vez."
```

Ao executar o `git status` novamente, o resultado é que não existe mais nenhuma pendência:

```
$ git status  
# On branch master  
nothing to commit (working directory clean)
```

A partir desse instante, podemos voltar para qualquer versão comitada anteriormente. Fazemos isso usando o comando `git reset --hard` e passando um identificador de versão.

```
$ git reset --hard identificadorDeVersao
```

Para obter um histórico dos commits do seu repositório e descobrir o identificador da versão desejada, podemos usar o comando:

```
git log
```

Será produzida uma saída como essa:

```
commit 1c57c4f27f3f18d46093f5fbc5cf2a8b330f13d6  
Author: Josenildo <jose@nildo.com.br>  
Date:   Wed Nov 24 15:00:24 2010 -0200
```

Otimizando a hidrodinâmica

```
commit 6abf51de170ae09e20aede3b0a01f5aa27f39299  
Author: Marivalda <marivalda@zip.com.br>  
Date:   Wed Nov 24 14:41:59 2010 -0200
```

Corrigindo avarias na asa

```
commit b28152b62c1d2fca891784423773c0abef0b03c2  
Merge: cdbbcc9 fbbbe794  
Author: Claudineide <clau@clau.com>  
Date:   Wed Nov 24 13:51:32 2010 -0200
```

Adicionando outro tipo de combustível

É possível mostrar um resumo, em apenas uma linha, do histórico de commits de um repositório

utilizando a opção `--oneline` :

```
git log --oneline
```

A resposta será algo como:

```
1c57c4f Otimizando a hidrodinâmica  
6abf51d Corrigindo avarias na asa  
b28152b Adicionando outro tipo de combustível
```

Para limitar o número de commits, há a opção `-n` :

```
git log --oneline -n 2
```

Teremos:

```
1c57c4f Otimizando a hidrodinâmica  
6abf51d Corrigindo avarias na asa
```

Às vezes percebemos que escrevemos código errado antes mesmo de comitar um arquivo e desejamos reverter esse arquivo para o estado anterior. Com o Git, basta usar o comando:

```
git checkout arquivo.errado
```

Para que o `git` consiga fazer o `commit` no nosso repositório, temos que informar para ele qual a conta que será usada para enviar as alterações e qual o nome que deve ser utilizado nas mensagens de `commit`. Para isso, vamos executar os comandos:

```
$ git config --global user.name "<Digite seu nome aqui>"  
$ git config --global user.email "<Digite o mesmo e-mail da sua conta>"
```

O comando `git config` é utilizado para adicionar/substituir alguma configuração. A opção `--global` indica que essa configuração é global para o usuário logado atualmente no computador, ou seja, o `git` utilizará as mesmas configurações para a conta logada atualmente na máquina.

MAIS SOBRE GIT

Com o Git é possível fazer muito mais coisas do que vamos falar no curso. É possível fazer commits parciais, voltar para versões antigas, criar branches, etc. Além disso existem interfaces gráficas que ajudam a fazer algumas das operações mais comuns, como o `gitk`, `gitg` e `gitx`.

Aprenda mais sobre o Git em: <http://help.github.com/>

1.3 EXERCÍCIO - INICIANDO COM GIT

1. Através do terminal, crie um novo diretório com o nome **perfil** e inicie um repositório em branco dentro dele:

```
$ mkdir perfil  
$ cd perfil  
$ git init
```

2. No diretório **perfil**, crie um arquivo de texto com o nome **bio.txt**, altere o conteúdo dele inserindo seu nome e verifique o estado do repositório:

```
$ touch bio.txt  
$ echo "Seu nome e sobrenome" >> bio.txt  
$ git status
```

Como você pode observar, o arquivo ainda não está sendo rastreado pelo *git*, isso é, está como *untracked*.

3. Faça com que o arquivo seja rastreado (*tracked*) pelo *git* para podermos enviá-lo ao repositório. Em seguida, verifique novamente o estado do repositório:

```
$ git add bio.txt  
$ git status
```

4. Antes de efetuar qualquer commit, é necessário configurar o nome e o email do seu usuário no Git. Digite:

```
$ git config --global user.name "<Digite seu nome aqui>"  
$ git config --global user.email "<Digite seu e-mail aqui>"
```

5. Envie o arquivo ao repositório e depois confira o estado do repositório:

```
$ git commit -m "Adicionando arquivo bio"  
$ git status
```

6. Verifique qual o identificador, *hash*, do *commit* que você acabou de fazer:

```
$ git log
```

Caso o histórico dos commits ocupe a tela inteira do terminal, aperte a tecla `q` para voltar à linha de comando.

7. Inclua novas linhas no arquivo com algumas informações sobre o que você gosta de fazer e verifique novamente o estado do seu repositório:

```
$ echo "Gosto de programar, tocar violão e praticar esportes!" >> bio.txt  
$ git status
```

Perceba que o arquivo está em um estado diferente. Antes de fazer o primeiro commit, o arquivo estava *untracked*, mas, como o *git* já está rastreando esse arquivo, ele está no estado *not staged*.

8. Para enviar as mudanças ao repositório, precisamos primeiro passar o arquivo alterado para *staged* e então efetuar o commit do mesmo. Faremos esses dois passos de uma vez:

```
$ git commit -am "Adicionando atividades preferidas."
```

1.4 SERVIÇOS DE REPOSITÓRIO DE CÓDIGO

Seja qual for a opção por sistema de controle de versões, há diversos serviços que disponibilizam espaço gratuito para os repositórios dos projetos. Em alguns, a condição para a gratuitade do serviço é que o código fique aberto, em outros é possível ter repositórios fechados, mas com algumas restrições.

Para casos onde os clientes possuem um acordo de que o código não pode ficar retido em ambientes de terceiros, tais serviços não são suficientes. Nesse caso, pode ser instalada em uma máquina da empresa alguma ferramenta que gerencie o git no servidor como o Bonobo (<https://bonobogitserver.com/>), se o sistema for Windows, ou seguir o tutorial (<https://git-scm.com/book/en/v2/Git-on-the-Server-The-Protocols>), se for Linux.

Contudo, com a ênfase recente em elasticidade e contratação de software como serviço, a importância de terceirizar essa infraestrutura tem crescido dentro das empresas. Dos serviços online, destacamos alguns deles:

- **Github** - o serviço mais utilizado para repositórios Git, até mesmo como ferramenta social para código, por suas funcionalidades que facilitam a colaboração.
- **BitBucket** - a escolha mais comum dos usuários de Mercurial, hoje também trabalha com o Git. O BitBucket é gratuito para repositórios de até 5 usuários.
- **Assembla** - com suporte a SVN e Git, o Assembla oferece 1 repositório privado de até 3 colaboradores gratuitamente. Além disso, ele também vem com um sistema de chamados integrado.
- **SourceForge** - antes o repositório mais utilizado pelos desenvolvedores de software *open source*, o SourceForge hoje oferece suporte para Git, Mercurial e SVN.

Durante o curso, utilizaremos o **GitHub** como repositório de código remoto. Ao acessar <https://github.com>, veremos a seguinte página:

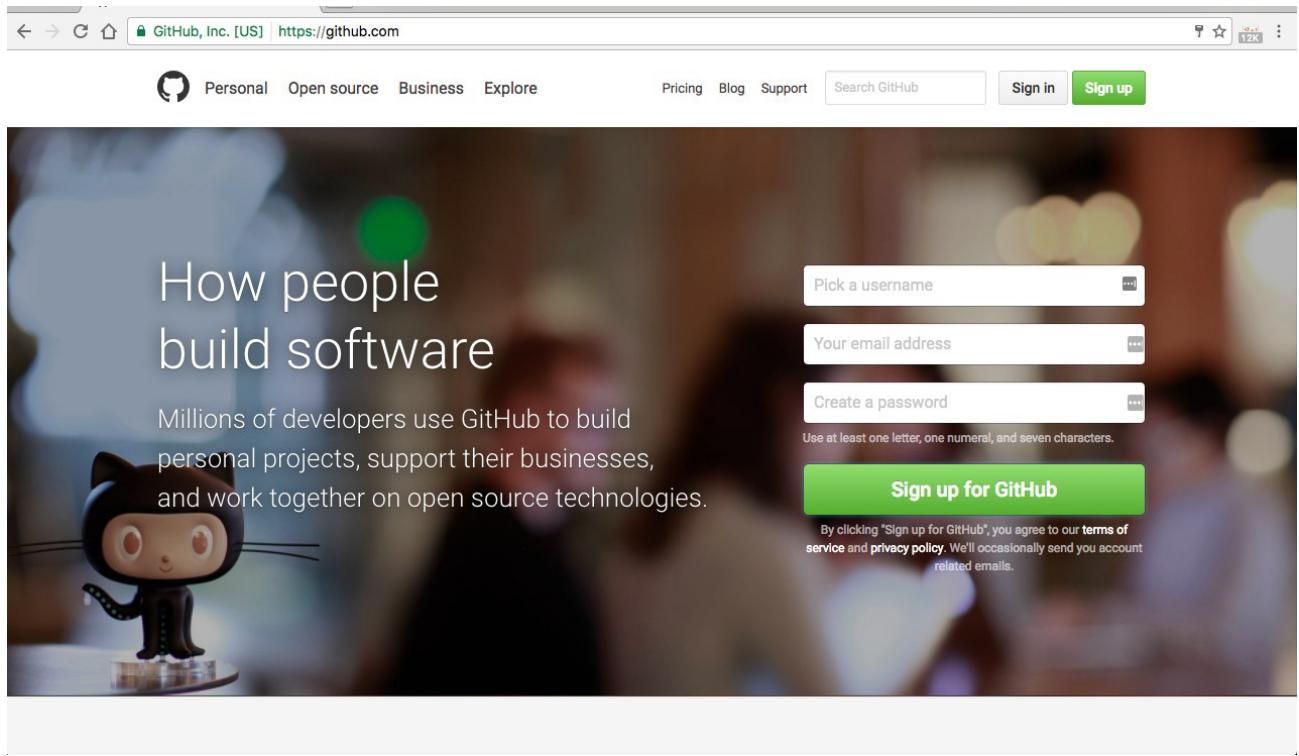


Figura 1.1: Homepage Github

No caso de ainda não termos uma conta, podemos criá-la preenchendo os campos: *Pick a username*, *Your email address* e *Create a password*.

Após preencher esses campos, vamos clicar no botão *Sign up for GitHub*. Assim, acabamos de fazer a primeira das 3 etapas do processo de criação de conta.

A segunda etapa é selecionar o tipo de conta que queremos usar. O **GitHub** tem uma modalidade gratuita e uma paga. Na modalidade gratuita, podemos criar repositórios **públicos** ilimitados. Já na modalidade paga, você tem a possibilidade de criar repositórios **públicos** e **privados** com um custo mensal ou anual.

Welcome to GitHub

You've taken your first step into a larger world, @curso-fj22.

✓ Completed Set up a personal account	💡 Step 2: Choose your plan	⚙️ Step 3: Tailor your experience
---	---	--

Choose your personal plan

- Unlimited public repositories for free.
- Unlimited private repositories for \$7/month. ([view in BRL](#))

Don't worry, you can cancel or upgrade at any time.

Help me set up an organization next
Organizations are separate from personal accounts and are best suited for businesses who need to manage permissions for many employees.
[Learn more about organizations.](#)

Both plans include:

- ✓ Collaborative code review
- ✓ Issue tracking
- ✓ Open source community
- ✓ Unlimited public repositories
- ✓ Join any organization

Continue

Figura 1.2: Welcome page step 2

Na terceira etapa, podemos selecionar um nível de experiência, o(s) uso(s) que pretendemos dar à nossa conta do **GitHub** e o que melhor nos descreve.

Welcome to GitHub

You'll find endless opportunities to learn, code, and create, @curso-fj22.

✓ Completed Set up a personal account	💡 Step 2: Choose your plan	⚙️ Step 3: Tailor your experience
---	---	--

How would you describe your level of programming experience?

Totally new to programming Somewhat experienced Very experienced

What do you plan to use GitHub for? (check all that apply)

<input type="checkbox"/> Design	<input type="checkbox"/> Development	<input type="checkbox"/> School projects
<input type="checkbox"/> Project Management	<input type="checkbox"/> Research	<input type="checkbox"/> Other (please specify)

Which is closest to how you would describe yourself?

I'm a professional I'm a hobbyist I'm a student
 Other (please specify)

What are you interested in?

(e.g. tutorials, android, ruby, web-development, machine-learning, open-source)

Submit [skip this step](#)

Figura 1.3: Welcome page step 3

Após preenchermos tudo, será enviado um e-mail de confirmação e será exibida uma tela onde poderemos escolher entre ler um *guia* e *iniciar um projeto*.

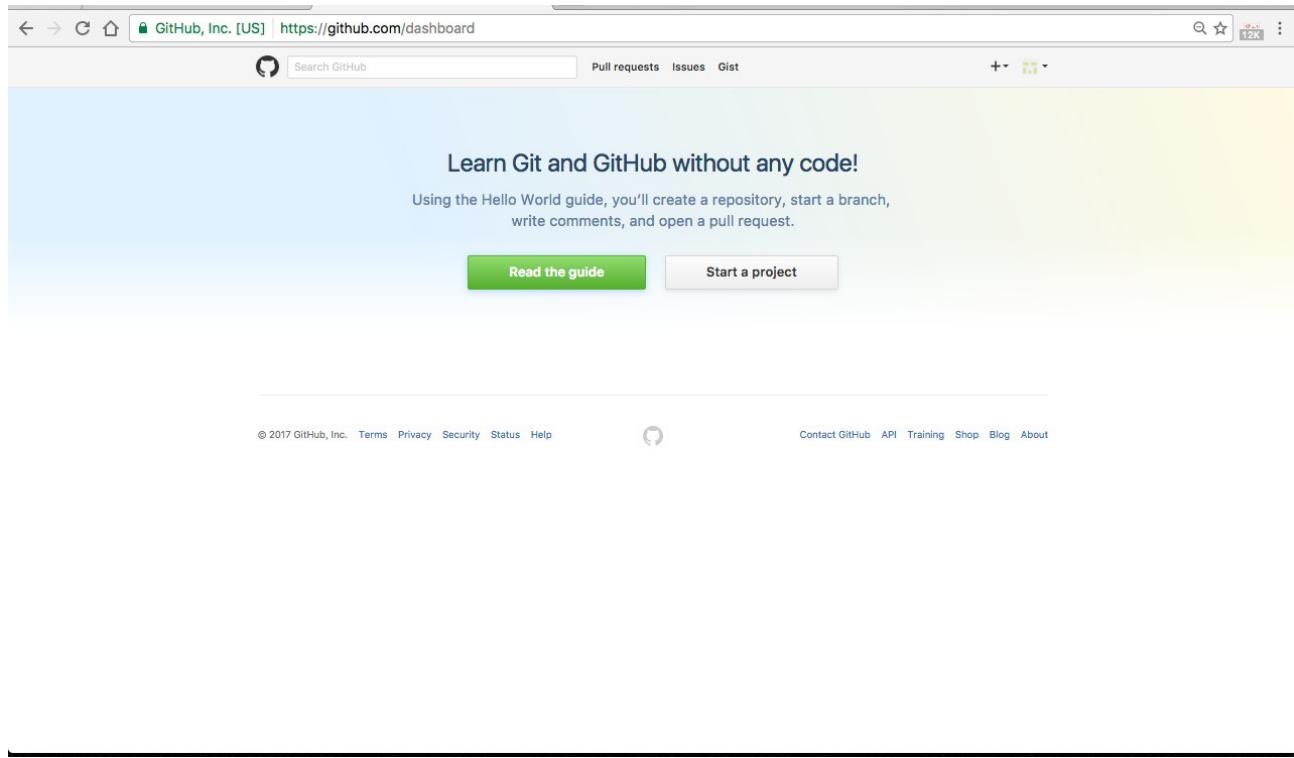


Figura 1.4: Dashboard

Quando clicarmos em *Start a Project*, será exibida uma tela informando que precisamos confirmar a criação da nossa conta através do nosso e-mail:

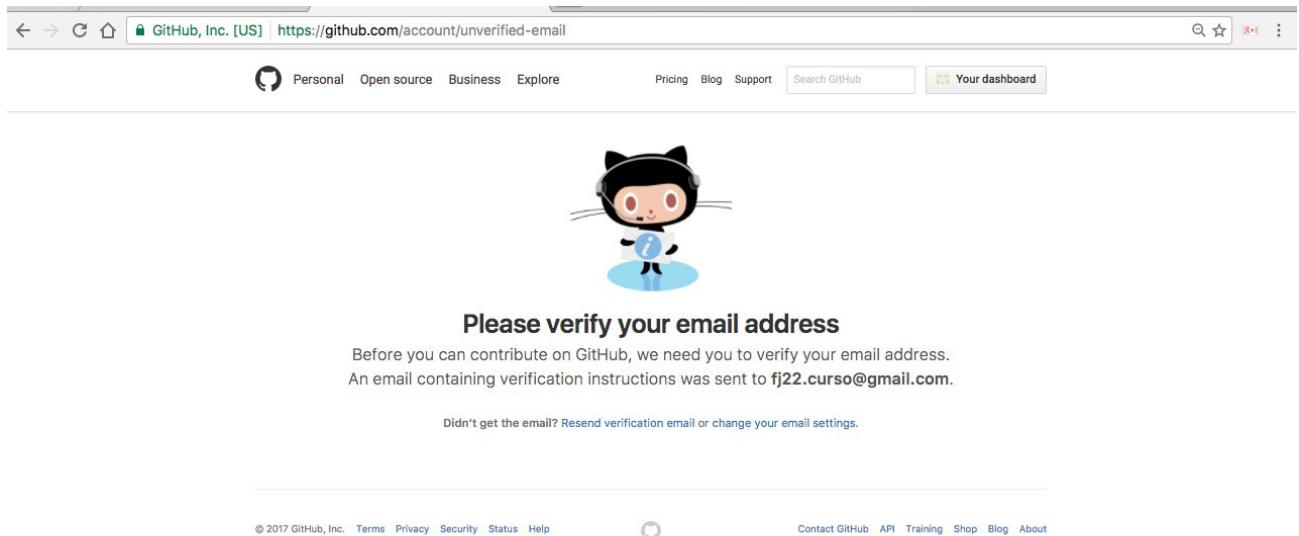


Figura 1.5: Verificação do e-mail

Ao analisarmos o e-mail recebido, vemos o link para a verificação *Verify email address*:

Hi @curso-fj22!

Help us secure your GitHub account by verifying your email address (fj22.curso@gmail.com). This lets you access all of GitHub's features.

[Verify email address](#)

Button not working? Paste the following link into your browser:
https://github.com/users/curso-fj22/emails/28457219/confirm_verification/36030355ab6a23c9f6c071494e1ebb571f037ee4

You're receiving this email because you recently created a new GitHub account or added a new email address. If this wasn't you, please ignore this email.

Figura 1.6: Email de verificação

Ao clicarmos nesse link, seremos redirecionados para a tela de configuração da nossa conta:

The screenshot shows the GitHub 'Email' settings page. On the left, a sidebar lists various personal settings categories: Profile, Account, **Emails** (which is selected), Notifications, Billing, SSH and GPG keys, Security, Blocked users, Repositories, Organizations, Saved replies, Authorized applications, and Installed integrations. Below these are Developer settings: OAuth applications, Integrations, and Personal access tokens. The main content area is titled 'Email' and contains a message: 'Your primary GitHub email address will be used for account-related notifications (e.g. account changes and billing receipts) as well as any web-based GitHub operations (e.g. edits and merges.)'. It shows the email 'fj22.curso@gmail.com' is already verified and is marked as 'Primary'. There is a 'Public' button next to it. Below this is a section to 'Add email address' with a 'Save email preferences' button at the bottom. A note about keeping the email private is also present.

Figura 1.7: Verificado

Com isso, nossa conta será criada e estará pronta para ser usada.

1.5 EXERCÍCIO OPCIONAL - CRIANDO UMA CONTA NO GITHUB

(pule este exercício caso já tenha uma conta)

1. Acesse <https://github.com>.

2. Preencha os campos:

- *Pick a username*
- *Your email address*
- *Create a password*
- Clique em *Sign up for Github*

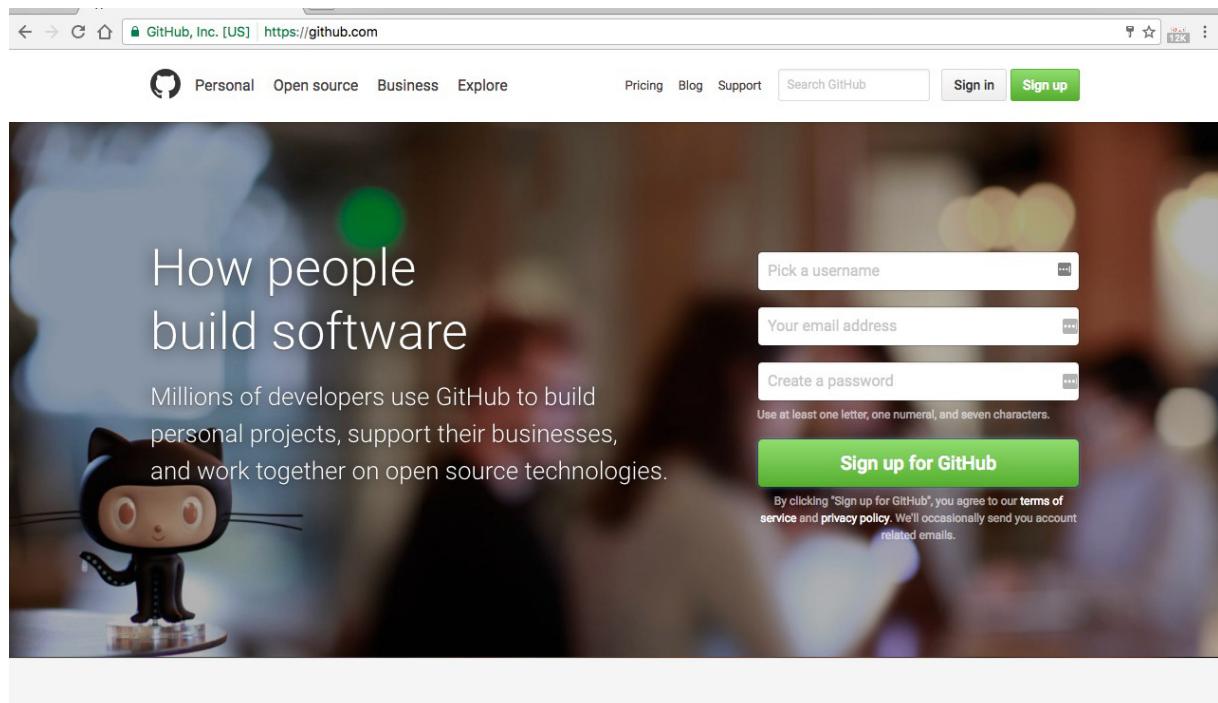


Figura 1.8: Homepage GitHub

3. Selecione o tipo de conta que você quer usar:

- Gratuita: repositórios públicos ilimitados
- Paga: repositórios privados e públicos ilimitados

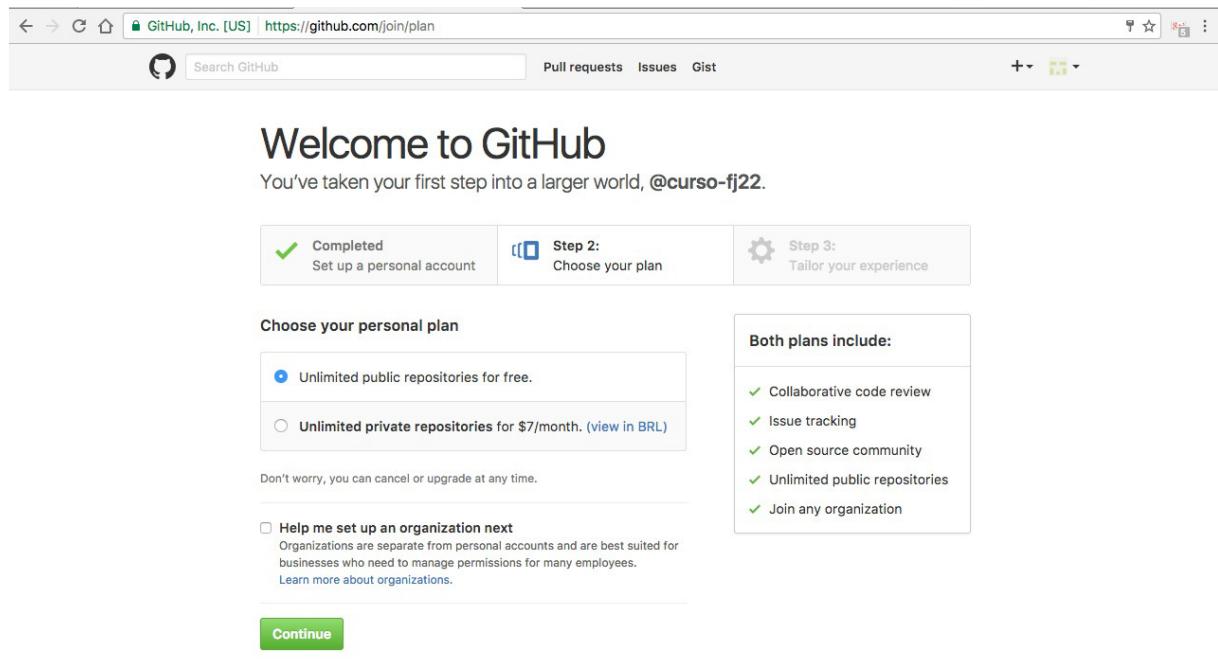


Figura 1.9: Welcome page step 2

4. Selecione:

- O seu nível de experiência com programação,
- Para que você planeja usar o Github,
- A opção que lhe descreve melhor.

Por fim, descreva seus interesses:

The screenshot shows the 'Welcome to GitHub' page from GitHub's join customization process. At the top, there's a navigation bar with links for 'Pull requests', 'Issues', and 'Gist'. Below the header, the title 'Welcome to GitHub' is displayed, followed by the subtext 'You'll find endless opportunities to learn, code, and create, @curso-fj22.' Three progress steps are shown: 'Completed' (Set up a personal account), 'Step 2: Choose your plan', and 'Step 3: Tailor your experience' (which is currently selected). The main content area contains several questions and input fields:

- How would you describe your level of programming experience?** Radio buttons for 'Totally new to programming', 'Somewhat experienced', and 'Very experienced' are shown, with 'Somewhat experienced' being selected.
- What do you plan to use GitHub for? (check all that apply)** Checkboxes for 'Design', 'Development', 'School projects', 'Project Management', 'Research', and 'Other (please specify)' are listed.
- Which is closest to how you would describe yourself?** Radio buttons for 'I'm a professional', 'I'm a hobbyist', and 'I'm a student' are shown, with 'I'm a hobbyist' being selected.
- What are you interested in?** A text input field with placeholder text 'e.g. tutorials, android, ruby, web-development, machine-learning, open-source'.

At the bottom, there are two buttons: a green 'Submit' button and a 'skip this step' link.

Figura 1.10: Welcome page step 3

5. Após clicar em *Submit*, será exibido uma tela de dashboard:

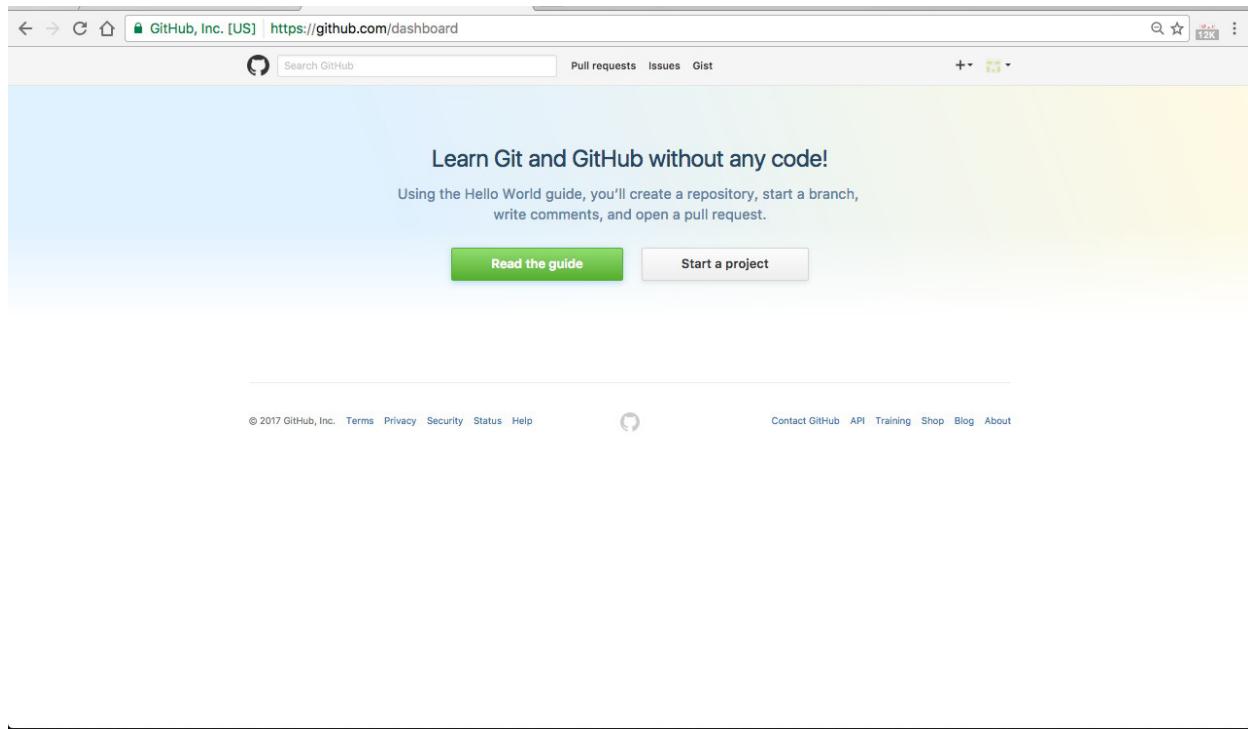


Figura 1.11: Dashboard

6. Se clicarmos em *Start a project*, será avisado que precisamos verificar nosso e-mail para ativar nossa conta:

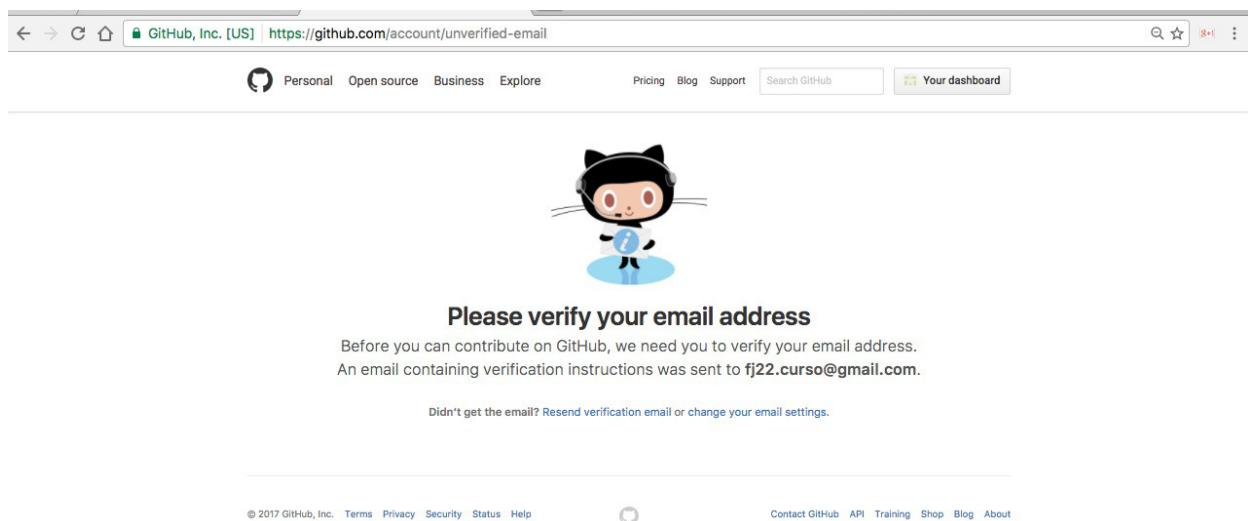


Figura 1.12: Verificação do e-mail

7. No e-mail que foi enviado pelo *GitHub*, vamos clicar no link *Verify email address*:

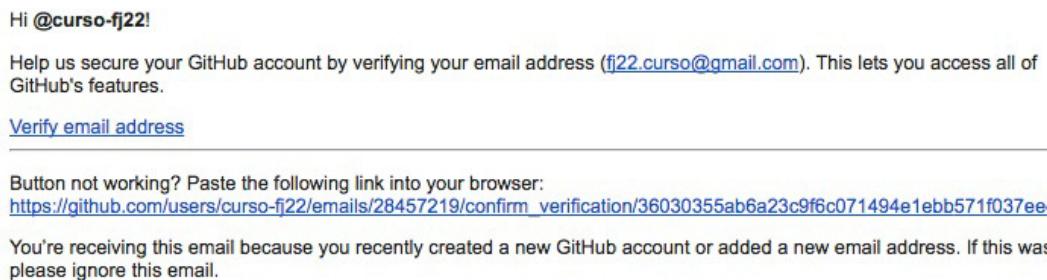


Figura 1.13: Email de verificação

8. Feito isso, seremos redirecionados para a tela de configurações da nossa conta indicando que nossa conta foi ativada:

The screenshot shows the "Email" section of the GitHub account settings. On the left, there's a sidebar with "Personal settings" and various options like Profile, Account, Emails (which is selected), Notifications, Billing, SSH and GPG keys, Security, Blocked users, Repositories, Organizations, Saved replies, Authorized applications, Installed integrations, Developer settings, OAuth applications, Integrations, and Personal access tokens. The main area is titled "Email" and contains the message "fj22.curso@gmail.com is already verified." Below this, there's a "Add email address" input field and a "Keep my email address private" checkbox. At the bottom, there are "Save email preferences" and "Looking for activity notification controls? Check the Notification center." buttons.

Figura 1.14: Verificado

1.6 UTILIZANDO GITHUB

Como mencionado anteriormente, utilizaremos o *GitHub* como nosso repositório remoto de código.

Inclusive, o projeto base para o curso encontra-se em <https://github.com/caelum/fj22-ingressos.git>:

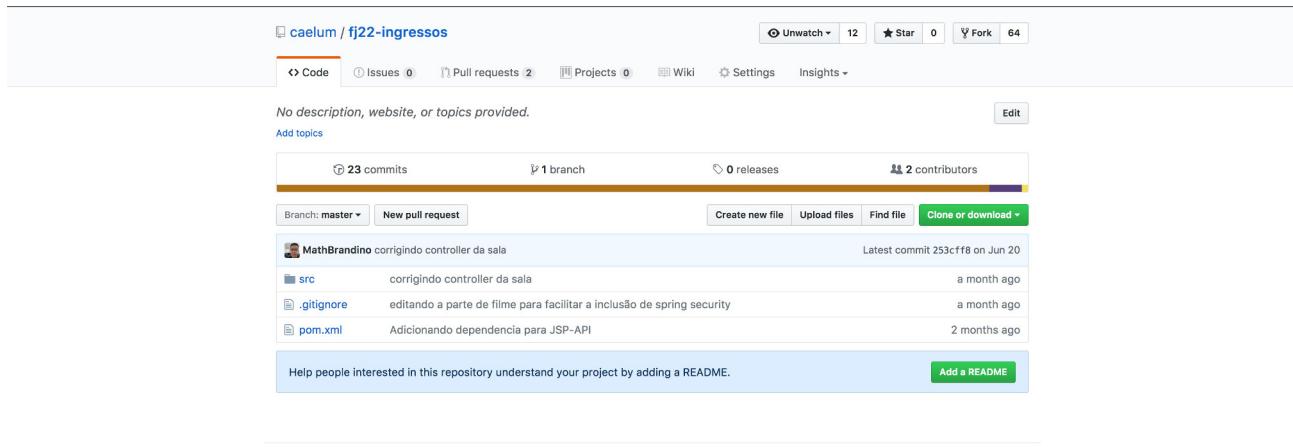


Figura 1.15: Projeto Base

O objetivo desse projeto é simular a venda de ingressos em um cinema e decidiram utilizar Spring MVC e Hibernate na aplicação. O código já está iniciado e já possui algumas funcionalidades, como adicionar salas e filmes, mas ainda não está finalizado. Nós vamos continuar esse projeto para permitir a compra de ingressos.

Porém, ele está na conta da caelum e, a menos que algum administrador nos libere o acesso a esse repositório, não será possível enviar nosso código local diretamente para esse projeto (push).

Felizmente, como esse repositório está público, podemos fazer uma cópia do mesmo para nossa conta. Esse processo de copiarmos um repositório de outra conta para a nossa é chamado de **Fork**.

Se reparamos bem, na página há um botão chamado *Fork*:

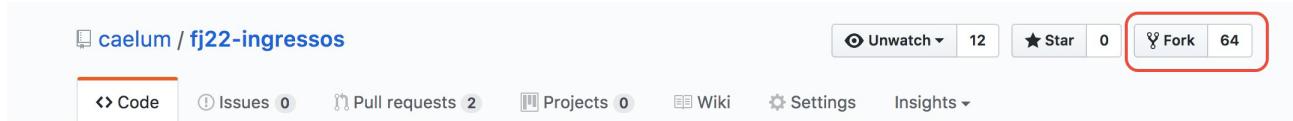


Figura 1.16: Botão Fork

Ao clicarmos nesse botão, será exibido uma animação indicando que o repositório está sendo copiado para nossa conta.

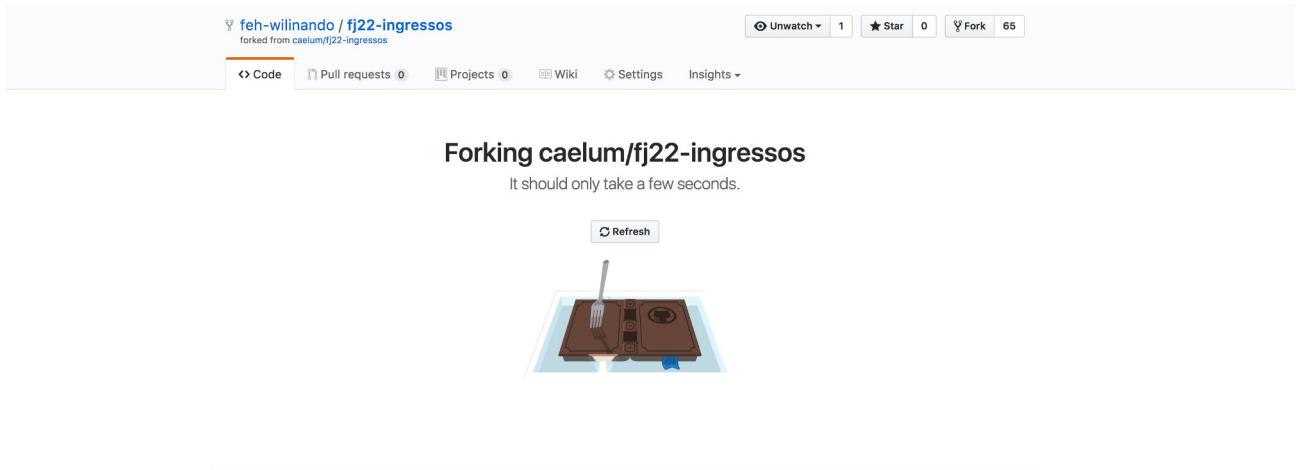


Figura 1.17: Fork em andamento

Ao término será exibido o mesmo repositório, só que em nossa conta:

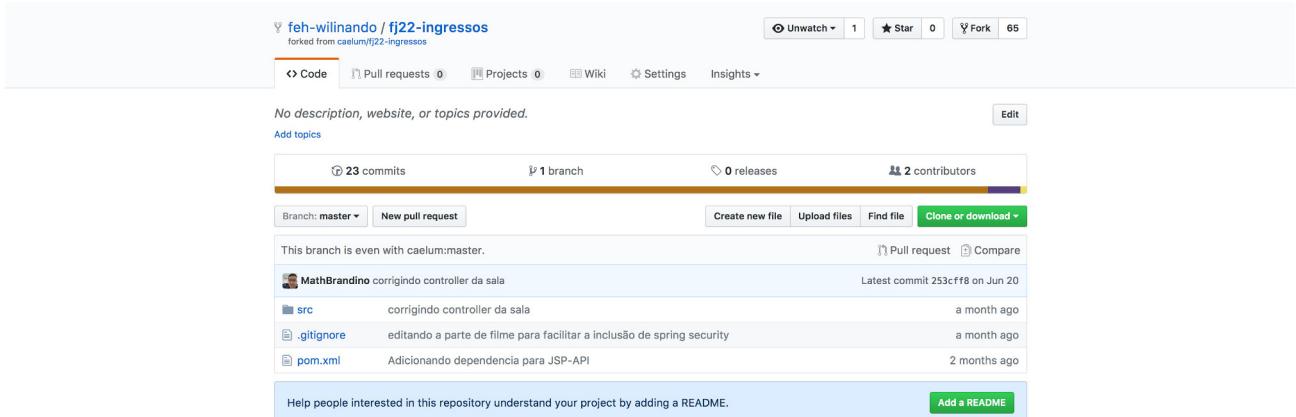


Figura 1.18: Fork encerrado

Agora sim podemos fazer um `git clone` do repositório em nossa conta e começar a desenvolver.

README.md

É muito comum ter em repositórios um texto contendo a finalidade do projeto, informações sobre os desenvolvedores ou sobre a utilização do projeto em si. Para exibir essas informações, podemos criar um arquivo na raiz do repositório chamado *README.md*. Nesse arquivo, podemos escrever utilizando a linguagem de marcação **Markdown**, que tem uma syntax simples para formatar o texto. Para saber mais sobre markdown, consulte:

<https://daringfireball.net/projects/markdown/syntax>

etc	Info about Eclipse dist files	3 years ago
vraptor-blank-project	updating vraptor version	2 years ago
vraptor-core	implements serializable on MessageList.class	4 months ago
vraptor-musicjungle	remove variable before the return	a year ago
vraptor-site	Merge pull request #1081 from angeliski/master	8 months ago
.gitignore	using getResource to find beans.xml	3 years ago
.travis.yml	trying java8 to fix UnsupportedClassVersionError	2 years ago
LICENSE	Applying properly our license	3 years ago
README.md	Update README.md	3 months ago
pom.xml	[maven-release-plugin] prepare for next development iteration	10 months ago
update-site.sh	installing grunt & deps in the script	3 years ago
README.md		



build passing maven central 4.2.0-RC5 release v4.2.0-RC4 License Apache 2

A web MVC action-based framework, on top of CDI, for fast and maintainable Java development.

Downloading directly or using it through Maven

For a quick start, you can use this snippet in your maven POM:

```
<dependency>
    <groupId>br.com.caelum</groupId>
    <artifactId>vraptor</artifactId>
    <version>4.2.0-RC3</version> <!-- or the latest version-->
```

Figura 1.19: README

1.7 EXERCÍCIO - FORK E CLONE DO PROJETO

1. Para poder enviar *commits* no projeto `fj22-ingressos` , faça um *fork* do repositório `fj22-ingressos` da conta da Caelum para a sua conta:

- Acesse <https://github.com/caelum/fj22-ingressos.git>
 - Clique em *Fork*
2. Fora do repositório anterior, **perfil**, clone o seu projeto, que está em
`https://github.com/<Seu usuario aqui>/fj22-ingressos.git` :
- ```
$ git clone https://github.com/<Seu usuario aqui>/fj22-ingressos.git
$ cd fj22-ingressos
```
3. Crie um arquivo chamado `README.md` na pasta `fj22-ingressos` com um conteúdo de descrição sobre você. Depois, verifique qual o status de seu repositório:
- ```
$ touch README.md  
$ echo "Descrição sobre você" >> README.md  
$ git status
```
4. Adicione o arquivo para ser rastreado pelo *git*:
- ```
$ git add README.md
```
5. Execute um commit com uma mensagem de descrição condizente, por exemplo:
- ```
$ git commit -m "adicionando descrição sobre um contribuinte novo ao projeto"
```
6. Por fim, envie as alterações comitadas para o repositório remoto:
- ```
$ git push
```

## 1.8 LIDANDO COM BUILD E DEPENDÊNCIAS

Hoje em dia é muito fácil fazer o build de um projeto *Java* devido às facilidades que nossos IDEs nos proporcionam.

O *IDE* se encarrega de colocar nossas dependências no *classpath*, bastando apenas dizer quais são as dependências, se encarrega também de criar o diretório em que ficarão nossas classes já compiladas, de compilar o projeto, de empacotar a aplicação entre outras funções. Podemos até interferir nesse processo adicionando plugins para coletar métricas, extrair relatórios, fazer o deploy, entre outras coisas.

Porém, em alguns desses processos que nosso *IDE* facilita, ainda há muita intervenção manual. Por exemplo: imagine que no nosso projeto precisamos usar o *Hibernate*. Para colocar a dependência do *Hibernate* no *classpath*, precisamos baixar o *.jar* do *Hibernate* e todos os arquivos *.jar* que o *Hibernate* precisa para funcionar. Imagine agora que, nesse mesmo projeto, também vamos usar *Spring*. Precisamos então baixar o *.jar* do *Spring* e todos os *.jar* de que o *Spring* depende.

Até agora nenhum problema, exceto o fato de termos que ficar baixando um monte de arquivos *.jar*. Mas e se o *Spring* e o *Hibernate* dependerem do mesmo arquivo *.jar*? Em casos como esse, para não ter nenhuma duplicidade, ou pior, ter a mesma dependência em versões diferentes, teremos que lidar com o conflito de dependências manualmente, sempre olhando se a dependência em questão já está no nosso

projeto antes de baixar.

Esse problema é tão comum que foram criadas várias ferramentas para automatizar processos como o de build e o de gerenciamento de dependências.

Em Java, temos uma ferramenta bem popular para automação do build chamada *Ant*. Com ela, escrevemos em um arquivo xml nosso script de build, declarando os passos necessários para construir o projeto, como criar o diretório de output, compilar nossas classes, compilar nossos testes, rodar nossos testes, empacotar a aplicação, etc.

Porém, apenas com ela, ainda ficávamos sem a gestão das nossas dependências e, por isso, usávamos outra ferramenta, chamada *Ivy*.

Para usar o *Ivy*, também usamos um arquivo xml onde declaramos nossas dependências. Ele então consegue baixá-las através de um repositório central.

Usando a soma do *Ant* e do *Ivy*, conseguíamos uma forma versátil e automatizada para nosso processo de build completo.

Posteriormente, uniram o que tem de melhor no *Ant* e no *Ivy* e criaram uma ferramenta híbrida chamada *Maven*.

Com o *Maven*, temos um único arquivo xml chamado **pom.xml** e, nesse arquivo, declaramos todas as dependências e plugins necessários para nosso processo de build, run ou deploy.

O arquivo **pom.xml** tem a seguinte estrutura básica:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/maven-4.0.0.xsd">

 <!-- Informações básicas do seu projeto -->
 <groupId>br.com.caleum</groupId>
 <artifactId>fj22-ingresso</artifactId>
 <version>1.0</version>

 <!-- Configurações do build * Opcional-->
 <build>
 ...
 </build>

 <!-- Declaração de dependências * Opcional-->
 <dependencies>

 <!-- Declarando a dependência do hibernate-core na versão 5.2.6.Final -->
 <dependency>
 <groupId>org.hibernate</groupId>
 <artifactId>hibernate-core</artifactId>
 <version>5.2.6.Final</version>
 </dependency>
 </dependencies>

```

```
</dependencies>
```

```
</project>
```

Além disso, o *Maven* tem uma estrutura padrão de pastas que deve ser seguida, como visto a seguir:

```
+src/
| |
| +- main/
| | |
| | +- java/
| | +- resources/
| | +- webapp/ (em caso de projetos web com empacotamento _.war_)
|
| +- test/
| | |
| | +- java/
| | +- resources/
```

- src/main/java/ - classes do nosso projeto.
- src/main/resources/ - arquivos de configuração, como, xml, properties e arquivos estáticos do projeto.
- src/main/webapp/ - pasta WEB-INF com seus arquivos internos (*web.xml*, por exemplo) e arquivos *JSP*.
- src/test/java - classes de testes.
- src/test/resources - arquivos de configuração, como, xml e properties, e arquivos estáticos para teste.

O *Maven* também tem um ciclo de build bem definido que conta com as seguintes fases:

- **validate** - valida se a estrutura do projeto está correta e se todas as informações necessárias estão disponíveis;
- **compile** - compila as classes de *src/main/java*;
- **test** - compila as classes de *src/main/test* e roda todos os testes;
- **package** - empacota a aplicação;
- **verify** - mostra informações de qualidade de código, como o resultado de testes de integração;
- **install** - instala a dependência localmente;
- **deploy** - faz o deploy da aplicação baseado nas configurações do *pom.xml*.

Ao instalar o *Maven*, é disponibilizado um utilitário em linha de comando chamado `mvn`.

A sintaxe desse utilitário é `mvn` seguido do *goal*(ação) que queremos executar. Os *goals* padrões são exatamente as fases do ciclo de build:

```
$ mvn validate
$ mvn compile
```

```
$ mvn test
$ mvn package
$ mvn verify
$ mvn deploy
```

Além desses, há muitos outros *goals*, como, por exemplo, o `clean`, que apaga o diretório de output gerado no processo de compilação.

Como o *Maven* tem um ciclo de build bem definido e cada fase depende das anteriores, ele sempre irá garantir que todas as fases antes do *goal* escolhido sejam executadas. Por exemplo, ao usar o *goal* `test`, o *Maven* irá executar as fases `validate`, `compile` e só depois a `test`, nessa ordem.

Podemos inclusive combinar *goals*. Por exemplo, se executarmos `mvn clean package`, o *Maven* chamará o *goal* `clean`, que faz a limpeza dos outputs, e o *goal* `package`, que validarão nosso projeto verificando se já possui todas as informações necessárias, compilará nosso projeto, rodará os testes e empacotará nossa aplicação.

Além dos *goals* padrões do *Maven*, podemos utilizar outros *goals* através de *plugins*.

A sintaxe de utilização de um *plugin* é `mvn plugin:goal`.

Ex.:

```
$ mvn dependency:tree
```

O *plugin* `dependency` seguido do *goal* `tree`, lista a árvore das dependências já baixadas na nossa aplicação.

Também existe um *plugin* do *Maven* para o *Jetty*. Para rodar o servidor, basta utilizar o comando `mvn jetty:run`.

#### INSTALANDO O MAVEN EM CASA

Para utilizar o *Maven* em casa, é necessário baixar os binários no site <https://maven.apache.org/download.cgi> e instalar seguindo os passos referentes ao seu sistema operacional que estão em <https://maven.apache.org/install.html>.

## 1.9 EXERCÍCIO - USANDO GERENCIADOR DE DEPENDÊNCIA E BUILD

1. Acesse o diretório do projeto, baixe as dependências e depois liste-as:

```
$ cd fj22-ingressos
$ mvn dependency:resolve
$ mvn dependency:tree
```

2. Empacote a aplicação e rode o *jetty*:

```
$ mvn package
$ mvn jetty:run
```

Alternativamente, você pode rodar os dois comandos de uma só vez:

```
$ mvn package jetty:run
```

3. No navegador, acesse <http://localhost:8080> e navegue pelo sistema.

4. Pare o *jetty* e limpe o diretório target do seu projeto:

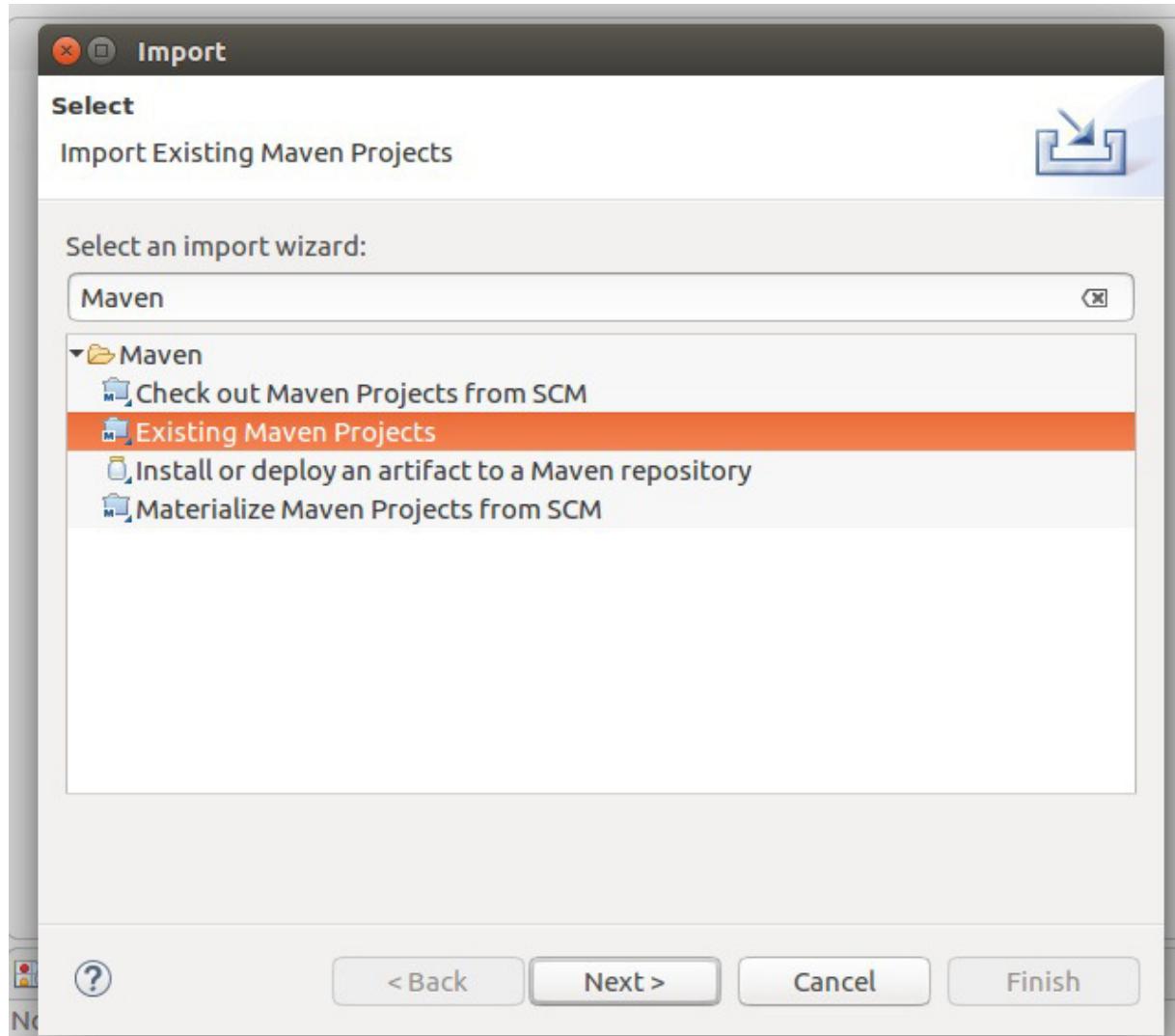
- No terminal, pressione *CTRL+C* para interromper a execução do *jetty*.
- Em seguida, execute:

```
$ mvn clean
```

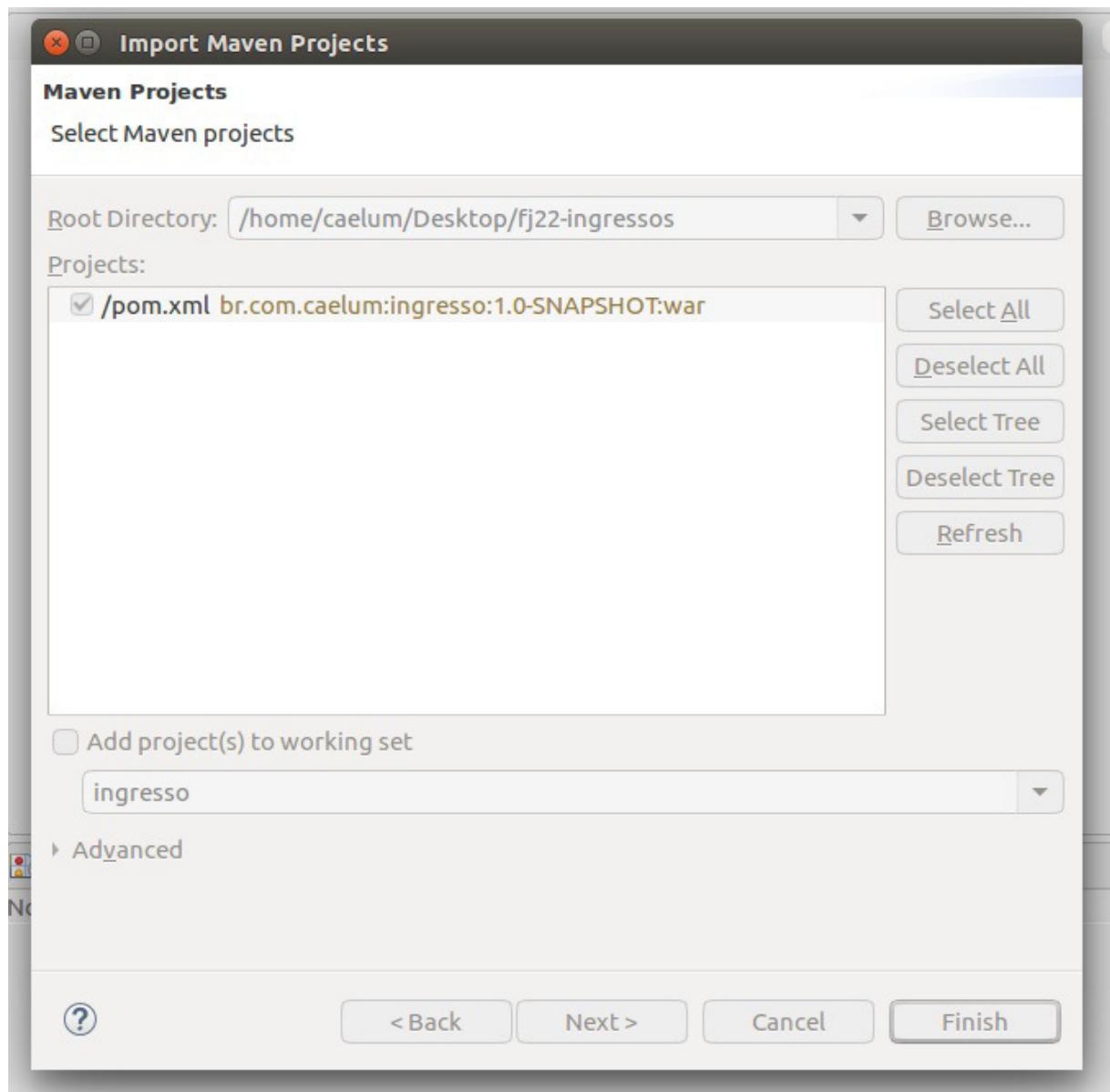
## 1.10 EXERCÍCIO - RODANDO PROJETO NO ECLIPSE COM MAVEN

1. Vamos importar o projeto no eclipse:

- File >> import
- Selecione *Existing maven project*



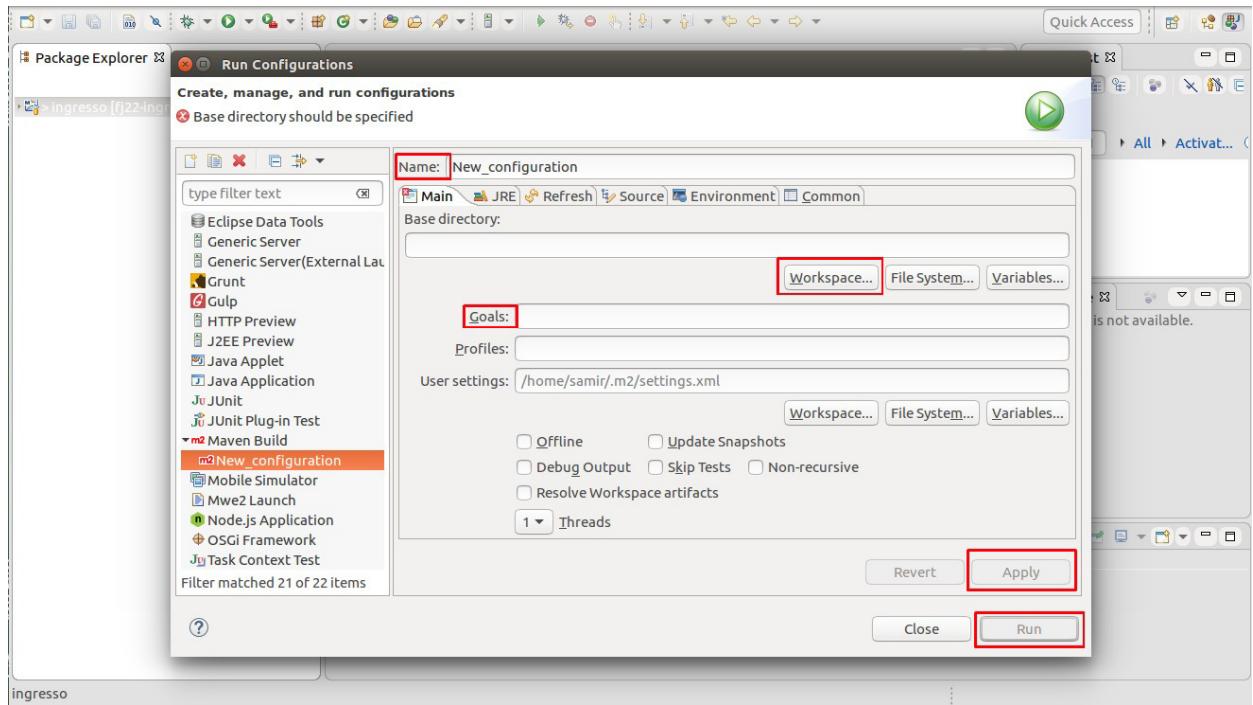
- Escolha a pasta do projeto *fj22-ingressos*



- Selecione o arquivo *pom.xml* que apareceu.

2. Vamos configurar a execução através do *Maven* no *Eclipse*:

- Clique com o botão direito sobre o projeto e selecione: Run As >> Run configurations...
- Clique com o botão direito sobre *Maven Build* >> New
- Se o campo *Base Directory* estiver vazio, clique em *Workspace* e selecione o projeto
- No campo *Name* preencha com "Rodar com maven"
- No campo *Goals* preencha com `clean package jetty:run`
- Clique no botão *Apply* e depois em *Run*



3. No navegador, acesse <http://localhost:8080> e navegue pelo sistema.

# TRABALHANDO COM BRANCHES

Depois de liberarmos o código de uma aplicação para o cliente, continuamos melhorando funcionalidades existentes e criando novas funcionalidades, sempre comitando essas alterações. Mas, e se houver um bug que precisa ser corrigido imediatamente? O código com a correção do bug seria liberado junto com funcionalidades pela metade, que ainda estavam em desenvolvimento.

Para resolver esse problema, a maioria dos sistemas de controle de versão permitem utilizar **branches**: linhas independentes de desenvolvimento nas quais podemos trabalhar livremente, versionando quando quisermos, sem atrapalhar outras mudanças no código.

Em projetos que usam Git podemos ter tanto branches locais, presentes apenas na máquina do programador, quanto branches remotas, que apontam para outras máquinas. Por padrão, a branch principal é chamada **master**, tanto no repositório local quanto no remoto. Idealmente, a **master** será uma branch estável, isto é, o código nessa branch estará testado e pronto para ser entregue.

Para listar as branches existentes em seu repositório Git, basta executar:

```
git branch
```

## PARA SABER MAIS: BRANCHES E O HEAD

No Git, uma branch é bem leve: trata-se apenas de um ponteiro para um determinado commit. Podemos mostrar os commits para os quais as branches estão apontando com o comando

```
git branch -v .
```

Há também um ponteiro especial, chamado **HEAD**, que aponta para a branch atual.

## Criando uma branch

Uma prática comum é ter no repositório branches novas para o desenvolvimento de funcionalidades que ainda estão em andamento, contendo os commits do que já foi feito até então. Para criar uma branch nova de nome **work** a partir do último commit da **master**, faça:

```
git branch work
```

Ao criar uma nova branch, ainda não estamos automaticamente nela. Para selecioná-la:

```
git checkout work
```

#### CRIANDO E SELECIONANDO UMA BRANCH

É possível criar e selecionar uma branch com apenas um comando:

```
git checkout -b work
```

Para visualizar o histórico de commits de todas as branches, podemos fazer:

```
git log --all
```

Para uma representação gráfica baseada em texto do histórico, há a opção:

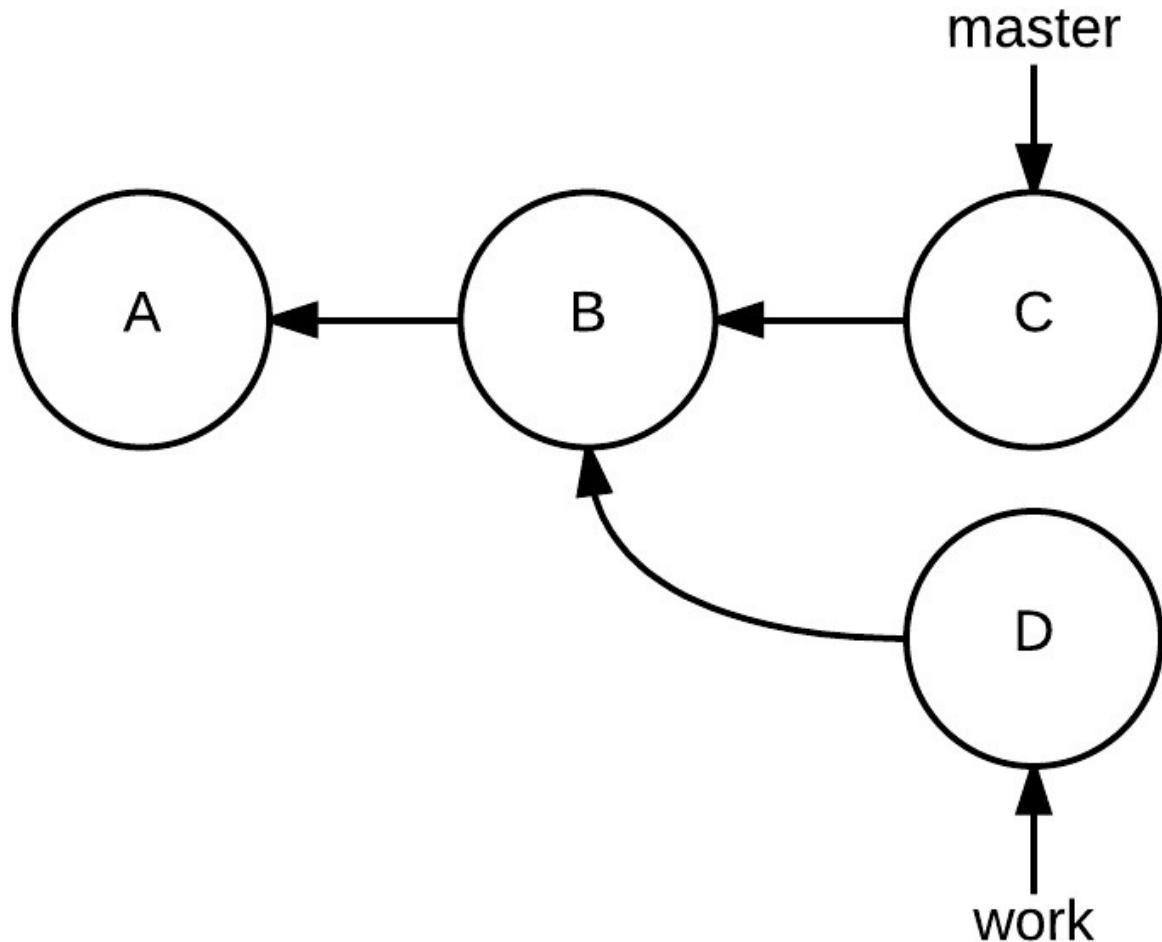
```
git log --graph
```

## 2.1 JUNTANDO COMMITS DE OUTRA BRANCH

E como fazemos para liberar as melhorias e novas funcionalidades? É preciso mesclar o código de uma branch com a branch `master`.

Em geral, os sistemas de controle de versão tem um comando chamado **merge**, que permite fazer a junção de uma branch em outra de maneira automática.

Vamos dizer que temos o seguinte cenário: nossa `master` tem os commits `A` e `B`. Então, criamos uma branch `work`, implementamos uma nova funcionalidade e realizamos o commit `D`. Depois, voltamos à `master` e, ao obter do repositório remoto as mudanças feitas por um outro membro do time, recebemos o commit `C`.



#### PARA SABER MAIS: OS COMMITS E SEUS PAIS

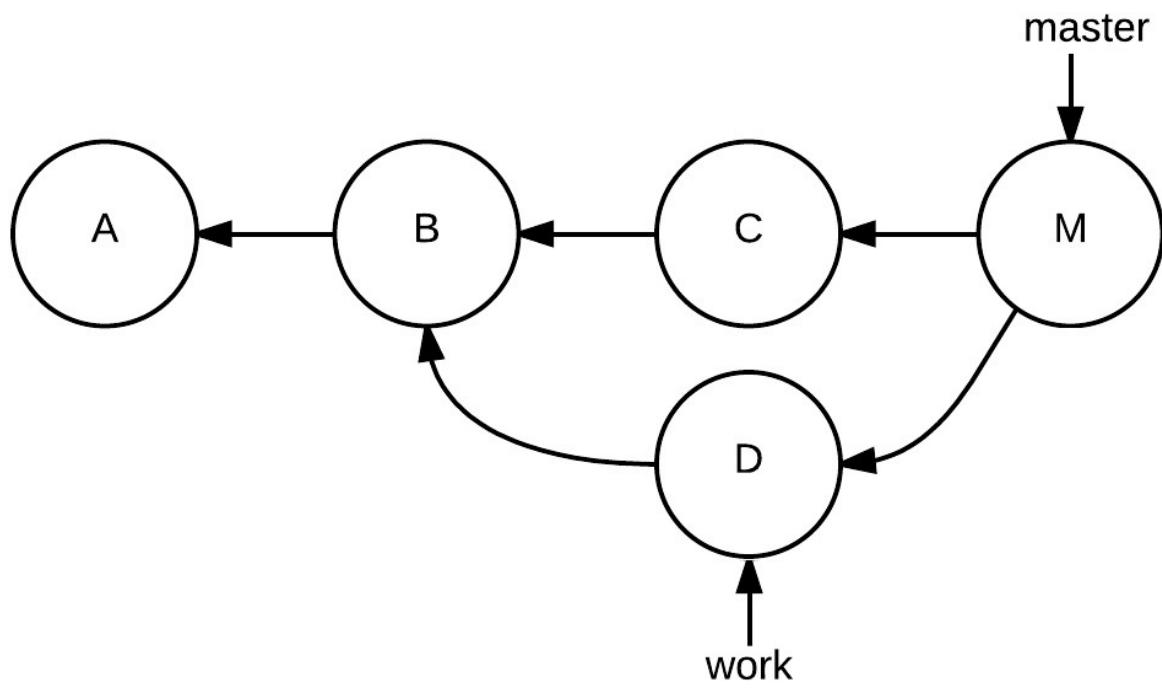
No Git, todo commit tem um ou mais pais, com exceção do primeiro. Para mostrar os commits com seus respectivos pais, podemos utilizar o comando `git log --oneline --parents`.

Se estivermos na branch `master`, podemos fazer o merge das alterações que estão na branch `work` da seguinte maneira:

```
git merge work
```

Quando fizermos o merge, as alterações da branch `work` são colocadas na branch `master` e é criado um commit `M` só para o merge. O git até mesmo abre um editor de texto para que possamos definir a mensagem desse commit de merge.

Se visualizarmos o gráfico de commits da `master` com os comandos `git log --graph` ou `gitk`, teríamos algo como:



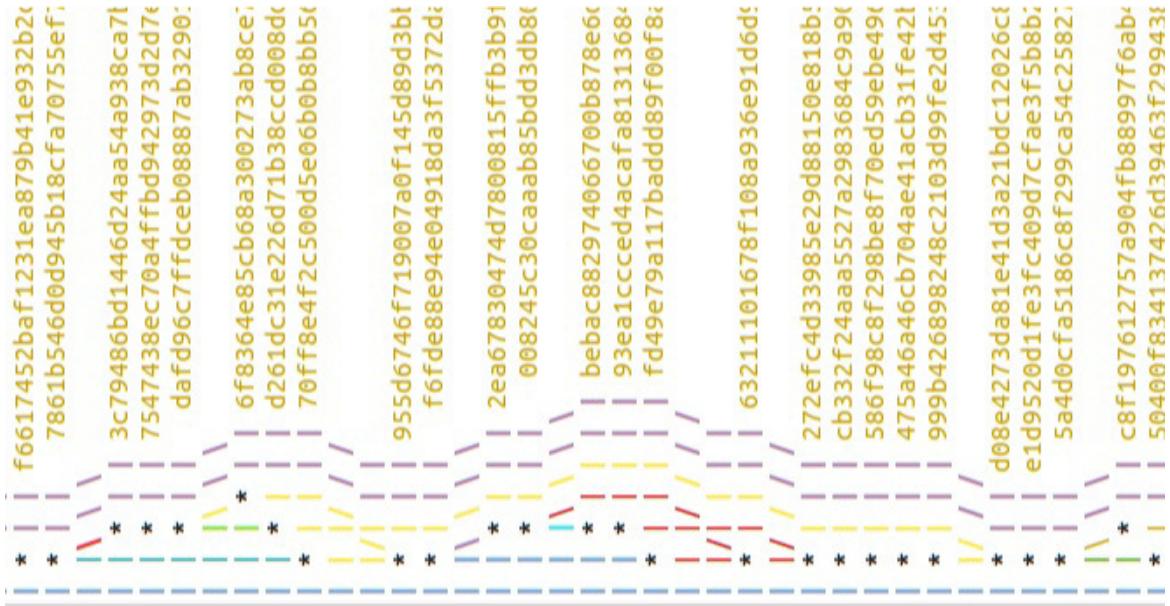
#### O MERGE DO TIPO FAST-FORWARD

Há um caso em que um `git merge` não vai gerar um commit de merge: quando não há novos commits na branch de destino. No nosso caso, se não tivéssemos o commit `C`, o merge seria feito simplesmente apontando a branch `master` para o commit `D`. Esse tipo de merge é chamado de *fast-forward*.

## Simplificando o histórico com rebase

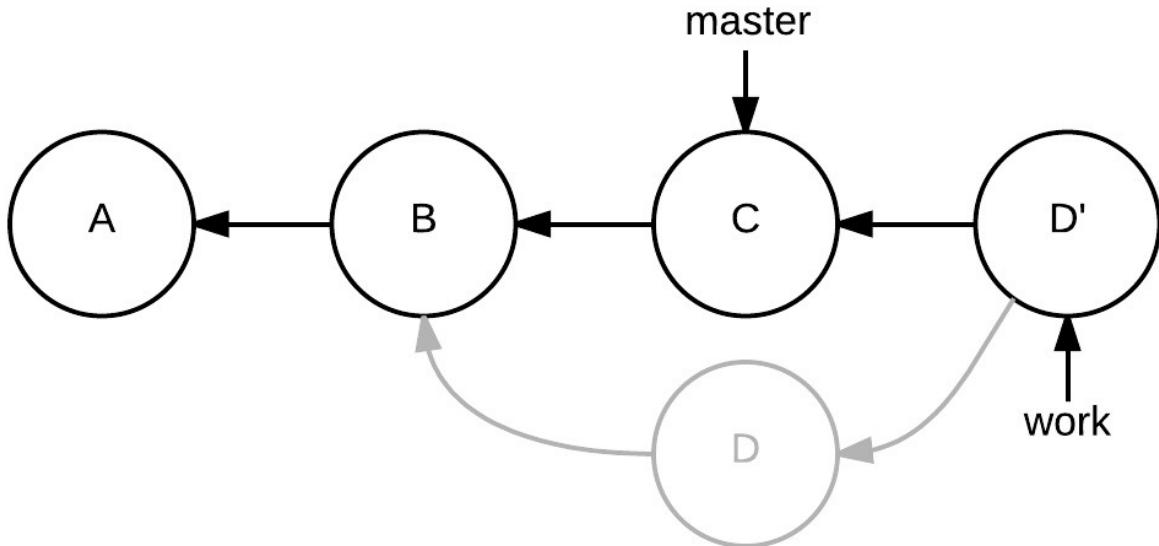
À medida que a aplicação vai sendo desenvolvida, é natural que vários merges sejam feitos, mas, se olharmos o histórico de commits da nossa branch `master`, veremos que haverá um commit específico para cada merge feito.

Como esses commits de merge criam um desvio na árvore de commits, o histórico do projeto acaba ficando bem poluído.



Para resolver isso, o Git possui o comando **rebase**, que permite mesclar o código de uma branch em outra, mas deixando o histórico de maneira mais linear. Para efetuar o rebase da `master` na `work`:

```
git rebase master
```



Após o rebase, a branch `work` teria sua *base* alterada do commit `B` para o commit `C`, linearizando o histórico de commits. Porém, ao efetuar o rebase, o histórico é modificado: o commit `D` é descartado e um novo commit quase idêntico, o `D'`, é criado.

Com a base da branch `work` refeita, é possível fazer um merge do tipo fast-forward na `master`, sem gerar um commit de merge.

## Boa prática: não fazer rebase na master

Se fizermos um `git rebase work` enquanto estivermos na branch principal, os commits da `master` seriam descartados, "clonados" e aplicados na branch `work`.

Mudar commits da `master` é perigoso pois é a branch que compartilhamos com a nossa equipe. O Git se perderia facilmente.

Uma boa prática no uso do Git é modificar o mínimo possível a branch principal. No caso de um rebase, nunca o faça de um jeito que possa alterar o histórico da `master`.

### O bom rebase

Mas então fazer um rebase de uma outra branch na `master` sem correr o risco de mudar a branch principal?

- Vá para a outra branch: `git checkout work`
- Faça o rebase da `master` na outra branch: `git rebase master`
- O commit *base* da outra branch teria sido refeito, obtendo os commits da `master` e clonando os novos commits da branch.
- Volte à branch principal: `git checkout master`
- Faça um merge da outra branch na principal: `git merge work`
- Como a outra branch já teria os commits da `master`, o merge acaba sendo um *fast forward*, deixando o histórico linear.

### Para saber mais: a controvérsia entre merge e rebase

Há uma grande controvérsia na maneira considerada ideal de trabalhar com Git: usar rebase, deixando o histórico limpo, ou usar merge, ganhando total rastreabilidade do que aconteceu?

Recomendamos uma abordagem que utiliza rebase porque, com o histórico limpo, tarefas como navegar pelo código antigo, revisar código novo e reverter mudanças pontuais ficam mais fáceis.

Mais detalhes em: <http://www.vidageek.net/2009/07/06/git-workflow/>

## 2.2 EXERCÍCIO - CRIANDO NOSSAS BRANCHES

1. Crie uma nova branch para podermos trabalhar fora da branch de produção:

```
git branch work
```

2. Verifique se a branch foi criada:

```
git branch
```

3. Agora vá para a branch criada. Para isso, execute o comando:

```
git checkout work
```

4. Altere o arquivo `README.md`, colocando informações sobre o projeto e avisando que é você quem está desenvolvendo.

5. Comite a alteração, mantendo-se na branch `work` :

```
git commit -am "colocando mais informações no README.md"
```

6. Veja que a alteração foi gravada no repositório, lembrando que para sair do `log` precisamos apertar a tecla `q` :

```
git log
```

7. Volte à branch `master`:

```
git checkout master
```

8. Veja que o arquivo `README.md` ainda está com o conteúdo anterior!

9. Indique no `pom.xml` que você é o desenvolvedor do projeto:

```
<project ...>
 <!-- restante omitido... -->
 <developers>
 <developer>
 <name>John Doe</name>
 <email>jdoe@example.com</email>
 </developer>
 </developers>
</project>
```

10. Ainda na branch `master`, comite as alterações do `pom.xml` :

```
git commit -am "adicionando developer no pom.xml"
```

11. Observe no gráfico dos quatro últimos commits que houve uma divergência nas branches `work` e `master` :

```
git log --oneline --all --graph -n 4
```

12. Vamos considerar nossa tarefa como *concluída* e que podemos fazer com que nosso trabalho entre em produção. Para isso, primeiramente, faça o rebase da branch `master` na `work` :

```
git checkout work
git rebase master
```

13. Veja no gráfico de commits das branches que o histórico foi linearizado e o commit da branch `work` foi aplicado depois do commit da `master` :

```
git log --oneline --all --graph -n 4
```

Note que o commit da branch `work` mudou de identificador. Já o da `master` permaneceu intocado.

14. Repare que as alterações no `pom.xml` estão disponíveis na branch `work` !

15. Agora que a branch `work` já tem as novidades da `master`, podemos fazer um merge da `work` na `master`. O merge será do tipo fast-forward.

```
git checkout master
git merge work
```

16. Observe, no histórico de commits, que o commit foi incorporado à branch `master`:

```
git log
```

17. Delete a branch `work`, já que você não a usará mais nesse instante:

```
git branch -d work
```

## 2.3 COMEÇANDO A TRABALHAR NO SISTEMA

Para nosso sistema funcionar da maneira correta, precisamos permitir a venda de ingressos. Porém, ao irmos no cinema, nós não compramos um ingresso para o filme, compramos para uma **Sessão**, que representa o filme sendo exibido em um determinado horário e em uma sala específica. Nosso sistema deve tratar os objetos da forma mais próxima possível do mundo real, então vamos modelar a classe `Sessão` que deve conter as informações do horário, da sala e do filme.

Como queremos que as `Sessões` sejam armazenadas no banco de dados, precisamos falar que nosso modelo também representa uma *entidade*:

```
@Entity
public class Sessao {

 @Id
 @GeneratedValue
 private Integer id;

 private LocalTime horario;

 @ManyToOne
 private Sala sala;

 @ManyToOne
 private Filme filme;

 /**
 * @deprecated hibernate only
 */
 public Sessao() {
 }

 public Sessao(LocalTime horario, Filme filme, Sala sala) {
 this.horario = horario;
 this.filme = filme;
 this.sala = sala;
 }

 //getters e setters
```

```
}
```

Todas as configurações utilizadas acima são para o Hibernate. A maioria delas já vimos no curso de *Java para Web*, mas há duas novas: a `@ManyToOne`, que determina que a relação entre os modelos é de *muitas* sessões para *uma* sala e também para *um* filme, e o comentário em *Javadoc*, avisando que o construtor sem argumentos só está lá porque o Hibernate precisa dele para fazer o mapeamento.

Agora que já temos a modelagem pronta, precisamos manipular as sessões em algum ponto da nossa aplicação. A responsabilidade de direcionar uma requisição que vem da tela a uma lógica específica, seguindo o padrão MVC, é da camada *controller*.

Nosso controller deverá direcionar uma chamada para uma tela de cadastro de sessão e, portanto, criaremos um método que retorne a tela que está no arquivo *sessao/sessao.jsp*. Essa tela já está criada no projeto, assim como todas as outras que veremos.

Toda sessão será exibida em uma sala. Portanto, precisamos saber qual é a sala em que a sessão será alocada no momento de sua criação. Receberemos essa informação no momento da chamada para a tela:

```
@Controller
public class SessaoController {

 @GetMapping("/admin/sessao")
 public ModelAndView form(@RequestParam("salaId") Integer salaId) {

 ModelAndView modelAndView = new ModelAndView("sessao/sessao");

 return modelAndView;
 }
}
```

Usamos a anotação `@GetMapping("url")` para deixar claro que este método será chamado apenas quando alguém fizer um requisição do tipo `get` para o endereço informado. Para não haver dúvida de qual o parâmetro que esperamos receber em uma determinada variável, usamos a anotação `@RequestParam("nomeDoParametro")` logo antes da variável. Por fim, nosso método devolve um objeto do tipo `ModelAndView`, diferente do retorno em texto com que já estávamos acostumados. Essa classe une o trabalho da `Model`, que disponibiliza objetos na `view`, com o retorno em texto do método, que informa qual página será processada. No exemplo acima, informamos a `view` já na construção do objeto do tipo `ModelAndView`.

A tela de cadastro deve mostrar a sala em que a sessão está sendo criada e uma lista com os filmes disponíveis. Desse modo, precisamos enviar essas informações, que temos no banco de dados, para a `view`. Acessaremos a sala e a lista de filmes utilizando o padrão de projeto `DAO`, ou seja, através de objetos do tipo `SalaDao` e `FilmeDao`. Como estes objetos são gerenciados pelo *Spring* e queremos reduzir ao máximo o acoplamento entre nossas classes, usaremos o conceito de **injeção de dependência** para conseguirmos pegar estes objetos. Para fazermos a injeção, usaremos a anotação `@Autowired`:

```
@Controller
```

```

public class SessaoController {

 @Autowired
 private SalaDao salaDao;

 @Autowired
 private FilmeDao filmeDao;

 //Método form omitido
}

```

Com os objetos já injetados, para mandar a sala e os filmes para a *view*, usaremos o método `addObject()` da classe `ModelAndView`, que recebe uma `String` para identificar o objeto na página e o objeto em si:

```

@Controller
public class SessaoController {

 @Autowired
 private SalaDao salaDao;

 @Autowired
 private FilmeDao filmeDao;

 @GetMapping("/admin/sessao")
 public ModelAndView form(@RequestParam("salaId") Integer salaId) {

 ModelAndView modelAndView = new ModelAndView("sessao/sessao");

 modelAndView.addObject("sala", salaDao.findOne(salaId));
 modelAndView.addObject("filmes", filmeDao.findAll());

 return modelAndView;
 }
}

```

Agora, precisamos recuperar os dados que foram escolhidos na tela, mas eles são representados por diversos objetos e só teremos o `id` de cada. Para deixar claro que esses dados não representam uma sessão completa, pois são as informações que vieram do formulário, vamos criar uma classe pra representar especificamente o formulário de sessão. Essa classe terá a responsabilidade de nos devolver uma sessão completa:

```

public class SessaoForm {

 private Integer id;

 @NotNull
 private Integer salaId;

 @DateTimeFormat(pattern="HH:mm")
 @NotNull
 private LocalTime horario;

 @NotNull
 private Integer filmeId;
}

```

```

//getters e setters

public Sessao toSessao(SalaDao salaDao, FilmeDao filmeDao){
 Filme filme = filmeDao.findOne(filmeId);
 Sala sala = salaDao.findOne(salaId);

 Sessao sessao = new Sessao(horario, filme, sala);
 sessao.setId(id);

 return sessao;
}
}

```

Depois que for preenchido o formulário, precisamos realizar a persistência. Como o estado do servidor será alterado e não queremos mostrar os dados na url, eles serão enviados pelo método http POST , método diferente de quando foi feita a chamada para mostrar o formlário. Por isso, foi possível manter a mesma url, /admin/sessao.

Para a inserção no banco de dados, precisamos do objeto do tipo Sessao completo, então vamos chamar o método do SessaoForm que fará isso pra gente. Além disso, temos que deixar o Spring ciente de que essa operação deverá ser feita dentro de uma transação. Faremos isso com a anotação @Transactional . Depois de inserir, vamos redirecionar para a listagem de sessões através da url /sala/id/sessoes, onde id é o valor do id da sala:

```

@Controller
public class SessaoController {

 @Autowired
 private SalaDao salaDao;

 @Autowired
 private FilmeDao filmeDao;

 @Autowired
 private SessaoDao sessaoDao;

 //demais métodos

 @PostMapping(value = "/admin/sessao")
 @Transactional
 public ModelAndView salva(@Valid SessaoForm form) {

 ModelAndView modelAndView = new ModelAndView("redirect:/sala/"+form.getSalaId()+"/sessoes");
 Sessao sessao = form.toSessao(salaDao, filmeDao);
 sessaoDao.save(sessao);
 return modelAndView;
 }
}

```

O método que recebe a chamada para /sala/id/sessoes já existe, o listaSessoes , assim como a página de listagem, mas falta disponibilizarmos a lista de filmes para a view. Para isso, vamos criar um

método no `DAO` para pegar a lista de filmes no banco de dados e alterar o método `listaSessoes` para inserir a lista no objeto do tipo `ModelAndView`.

Voltando para o `SessaoForm`, vemos que colocamos nele diversas validações, então, caso alguma dessas validações não seja respeitada, o código de persistência não deveria sequer ser processado. Para isso, precisaremos validar se não ocorreu nenhum erro e, caso seja identificado algum, nós retornaremos o usuário para o próprio form. Assim como já fizemos no curso FJ-21, vamos pedir para o *Spring* nos fornecer um objeto do tipo `BindingResult` e usá-lo para a validação.

Porém, se o usuário tiver preenchido o formulário com erros, ele vai querer ver as informações que havia preenchido antes. Vamos então enviar para o método `form` o objeto do tipo `SessaoForm`:

```
@PostMapping(value = "/admin/sessao")
@Transactional
public ModelAndView salva(@Valid SessaoForm form, BindingResult result) {

 if (result.hasErrors()) return form(form.getSalaid(), form);

 ModelAndView modelAndView = new ModelAndView("redirect:/sala/"+form.getSalaid()+"//sessoes");

 Sessao sessao = form.toSessao(salaDao, filmeDao);

 sessaoDao.save(sessao);

 return modelAndView;
}
```

Por fim, vamos alterar o método `form` para receber esse objeto como parâmetro e disponibilizá-lo na *view*:

```
@Controller
public class SessaoController {

 .

 .

 @GetMapping("/admin/sessao")
 public ModelAndView form(@RequestParam("salaid") Integer salaid, SessaoForm form) {

 form.setSalaid(salaid)

 ModelAndView modelAndView = new ModelAndView("sessao/sessao");

 modelAndView.addObject("sala", salaDao.findOne(salaid));
 modelAndView.addObject("filmes", filmeDao.findAll());
 modelAndView.addObject("form", form);

 return modelAndView;
 }

 //demais métodos
}
```

## 2.4 EXERCÍCIO - ADICIONANDO A SESSÃO AO SISTEMA

---

1. Crie uma nova branch para começar a trabalhar no projeto:

```
git checkout -b work
```

2. No pacote `br.com.caelum.ingresso.controller`, crie uma classe com o nome `SessaoController` e anote-a com `@Controller` para que ela seja um controller para o Spring. Nessa classe, crie um método chamado `form` para atender a requisições na URI `/admin/sessao` para o verbo http `GET`. Esse método deve receber o id da sala como parâmetro na URI e o nome desse parâmetro será `salaId`. No método, crie um novo objeto do tipo `ModelAndView` passando para o seu construtor o caminho da página que retornaremos para o usuário, que será `sessao/sessao`. O método também deve disponibilizar para a tela a sala, baseado no parâmetro `salaId`, e a lista com os filmes. Não se esqueça de injetar os atributos `SalaDao` e `FilmeDao` para conseguir recuperar as informações do banco de dados:

```
@Controller
public class SessaoController {

 @Autowired
 private SalaDao salaDao;
 @Autowired
 private FilmeDao filmeDao;

 @GetMapping("/admin/sessao")
 public ModelAndView form(@RequestParam("salaId") Integer salaId) {

 ModelAndView modelAndView = new ModelAndView("sessao/sessao");

 modelAndView.addObject("sala", salaDao.findOne(salaId));
 modelAndView.addObject("filmes", filmeDao.findAll());

 return modelAndView;
 }
}
```

3. Crie uma classe de modelo no pacote `br.com.caelum.ingresso.model` para representar a *Sessão* e anote-a com `@Entity` do pacote `javax.persistence`. Ela deve ter `id`, `horario` e deve estar vinculada a uma `Sala` e a um `Filme`. Para o horário, utilizaremos o tipo `LocalTime` do Java 8:

```
@Entity
public class Sessao {

 @Id
 @GeneratedValue
 private Integer id;
 private LocalTime horario;

 @ManyToOne
 private Sala sala;

 @ManyToOne
 private Filme filme;

 /**
 * @deprecated hibernate only
 */
 public Sessao() {
```

```

 }

 public Sessao(LocalTime horario, Filme filme, Sala sala) {
 this.horario = horario;
 this.filme = filme;
 this.sala = sala;
 }

 //demais getters e setters
}

```

4. Como o formulário da sessão usa mais de um modelo, criaremos uma classe chamada `SessaoForm` no pacote `br.com.caelum.ingresso.model.form` para representar especificamente o formulário. Nessa classe, adicionaremos as anotações para que as validações do formulário sejam feitas corretamente. Criaremos também um método na classe que retorne um objeto `Sessao` baseado nos dados que foram preenchidos pelo usuário no formulário, para futuramente persistirmos no banco de dados:

```

public class SessaoForm {

 private Integer id;

 @NotNull
 private Integer salaId;

 @DateTimeFormat(pattern="HH:mm")
 @NotNull
 private LocalTime horario;

 @NotNull
 private Integer filmeId;

 public Sessao toSessao(SalaDao salaDao, FilmeDao filmeDao){
 Filme filme = filmeDao.findOne(filmeId);
 Sala sala = salaDao.findOne(salaId);

 Sessao sessao = new Sessao(this.horario, filme, sala);

 return sessao;
 }

 //getters e setters
}

```

5. Precisamos agora disponibilizar nosso `SessaoForm` na request para o formulário. Vamos aproveitar e já setar o id da sala para esse form:

```

@Controller
public class SessaoController {

 //atributos injetados com @Autowired

 @GetMapping("/admin/sessao")
 public ModelAndView form(@RequestParam("salaId") Integer salaId, SessaoForm form) {
 form.setSalaId(salaId);
 }
}

```

```

 ModelAndView modelAndView = new ModelAndView("sessao/sessao");

 modelAndView.addObject("sala", salaDao.findOne(salaId));
 modelAndView.addObject("filmes", filmeDao.findAll());
 modelAndView.addObject("form", form);

 return modelAndView;
 }
}

```

6. Como precisamos salvar sessões no banco de dados e precisamos listar as sessões de uma sala, vamos criar o DAO de sessão no pacote `br.com.caelum.ingresso.dao`:

```

@Repository
public class SessaoDao {

 @PersistenceContext
 private EntityManager manager;

 public void save(Sessao sessao) {
 manager.persist(sessao);
 }

 public List<Sessao> buscaSessoesDaSala(Sala sala) {
 return manager.createQuery("select s from Sessao s where s.sala = :sala",
 Sessao.class)
 .setParameter("sala", sala)
 .getResultList();
 }
}

```

7. Na classe `SessaoController`, vamos injetar um atributo do tipo `SessaoDao` e criar um método que receberá o verbo http *POST* do nosso form e persistirá uma sessão no banco de dados. Anote esse método com `@Transactional` para que as operações sejam feitas dentro de uma transação.

```

@Controller
public class SessaoController {

 //demais atributos injetados com @Autowired

 @Autowired
 private SessaoDao sessaoDao;

 //demais métodos

 @PostMapping(value = "/admin/sessao")
 @Transactional
 public ModelAndView salva(@Valid SessaoForm form, BindingResult result) {

 if (result.hasErrors()) return form(form.getSalaid(), form);

 ModelAndView modelAndView = new ModelAndView("redirect:/admin/sala/" +
 form.getSalaid() + "/sessoes");

 Sessao sessao = form.toSessao(salaDao, filmeDao);

 sessaoDao.save(sessao);

 return modelAndView;
 }
}

```

```
 }
}
```

8. Para mostrar as sessões de uma sala, vamos para a classe `SalaController`. Nela vamos injetar o `SessaoDao` para acessar o banco de dados e vamos alterar o método `listaSessoes`, que já existe na classe, para disponibilizar as sessões na página de listagem:

```
@Controller
public class SalaController {
 @Autowired
 private SessaoDao sessaoDao;

 //demais atributos e métodos

 @GetMapping("/admin/sala/{id}/sessoes")
 public ModelAndView listaSessoes(@PathVariable("id") Integer id) {
 //comandos já existentes
 view.addObject("sessoes", sessaoDao.buscaSessoesDaSala(sala));
 //comandos já existentes
 }
}
```

9. Agora que incluímos a sessão no projeto, vamos commitar o trabalho que fizemos e testar todas as funcionalidades que acabamos de implementar.
10. Com nossa tarefa de adicionar a sessão no sistema finalizada e testada, podemos juntar o que foi feito na branch work para a branch master através do rebase e, depois de conferir que o commit está na branch master, deletar a branch work.

#### PARA SABER MAIS: EM CASA

Se for utilizar o projeto em casa, lembre que o usuário e senha informados ao datasource que o Spring está gerenciando no xml do Spring devem ser os mesmos que o usuário e senha do seu banco de dados

## 2.5 ANALISANDO A REGRA DE NEGÓCIO

Implementamos a funcionalidade de adicionar a sessão na sala, mas temos que pensar no mundo real, onde a sessão que o usuário quer adicionar pode estar em conflito com as sessões já existentes. Portanto, antes de simplesmente adicionar, a sessão precisa passar por uma validação, que será feita por uma classe que criaremos com essa única responsabilidade. Vamos chamá-la de `GerenciadorDeSessao` e ela terá a função de verificar se a sessão que estamos tentando adicionar cabe entre as sessões daquela sala:

```
public class GerenciadorDeSessao {
```

```

 public boolean cabe(Sessao sessaoAtual) {
 return true;
 }
}

```

Para podermos fazer a verificação, temos que ter acesso a todas as sessões que acontecerão naquela sala, para isso pediremos a lista de sessões como parâmetro no construtor:

```

public class GerenciadorDeSessao {
 private List<Sessao> sessoesDaSala;

 public GerenciadorDeSessao(List<Sessao> sessoesDaSala) {
 this.sessoesDaSala = sessoesDaSala;
 }

 public boolean cabe(Sessao sessaoAtual) {
 return true;
 }
}

```

Agora, precisamos percorrer a lista de sessões já existentes e, para cada sessão já alocada na sala, vamos checar:

- Se a sessão nova A começa antes de uma sessão já existente B, a duração do filme da sessão A deve acabar antes da sessão B começar;
- Se a sessão nova A começa depois de uma sessão já existente B, a duração do filme da sessão B deve acabar antes da sessão A começar.

Estamos usando a API de data do Java 8, que foi baseada na biblioteca **Joda Time** e possui vários métodos que vão nos ajudar a fazer essas validações:

```

public boolean cabe(Sessao sessaoAtual) {
 for (Sessao sessaoDoCinema : sessoesDaSala) {
 if (!horarioisValido(sessaoDoCinema, sessaoAtual)) {
 return false;
 }
 }
 return true;
}

private boolean horarioisValido(Sessao sessaoExistente, Sessao sessaoAtual) {
 LocalDate hoje = LocalDate.now();

 LocalDateTime horarioSessao = sessaoExistente.getHorario().atDate(hoje);
 LocalDateTime horarioAtual = sessaoAtual.getHorario().atDate(hoje);

 boolean ehAntes = horarioAtual.isBefore(horarioSessao);
}

```

```

if (ehAntes) {

 return horarioAtual
 .plus(sessaoAtual.getFilme().getDuracao())
 .isBefore(horarioSessao);
} else {

 return horarioSessao
 .plus(sessaoExistente.getFilme().getDuracao())
 .isBefore(horarioAtual);
}

```

Nosso código está validando todas as possibilidades que listamos ainda há pouco. Contudo, como já estamos usando alguns recursos do Java 8, seria interessante usarmos mais alguns para nos acostumarmos com esses novos comandos. Por exemplo, ao invés de fazermos esse `for` e `if`, podemos usar outro recurso bem interessante do Java 8, o 'Stream', que transforma nossa lista em um fluxo de sessões:

```

public boolean cabe(Sessao sessaoAtual) {
 Stream<Sessao> stream = sessoesDaSala
 .stream();

 // restante do código
}

```

Agora, para cada objeto do fluxo, queremos verificar se o horário da sessão nova é válido e, como resposta de cada comparação, teremos um `boolean`. Para conseguirmos pegar cada objeto do fluxo e retornar um `boolean`, usaremos um especialista nisso, o método `map()` do `Stream`, que devolverá um novo fluxo, convertido no tipo que desejamos. No nosso caso, teremos um `Stream<Boolean>` :

```

public boolean cabe(Sessao sessaoAtual) {
 Stream<Sessao> stream = sessoesDaSala
 .stream();

 Stream<Boolean> booleanStream = stream
 .map(sessaoExistente -> horarioisValido(sessaoExistente, sessaoAtual));

 // restante do código
}

```

Como já possuímos um fluxo com todos os `Boolean`s, vamos checar se em algum momento o horário da sessão nova estava inválido. Se isso acontecer, saberemos que a sessão não cabe naquela sala. Temos então que fazer alguma comparação entre todos eles, para ter essa resposta de uma vez. Utilizando o comparador `&&`, ficaria algo dessa forma :

```

if (boolean1 && boolean2 && ... && booleanX){
 return true;
}

return false;

```

Repare que, em ambos os casos, nós acabamos reduzindo tudo apenas para um `boolean`. Mas gerar esse código pode ser um pouco trabalhoso: teremos que percorrer novamente o `Stream`, passando elemento por elemento.

Felizmente, esse objeto é inteligente e já sabe fazer isso para nós. Existe outro método que nos ajudará com esse serviço, o `reduce()`, que consegue reduzir vários elementos em um único, a partir de uma lógica que passarmos para ele. No nosso caso, precisamos apenas falar qual é a estratégia que deve ser executada. Desse modo, usaremos o método `Boolean.logicalAnd(b1, b2)`, onde `b1` e `b2` são dois booleanos e que nos retorna o resultado da operação lógica `AND` entre os parâmetros:

```
public boolean cabe(Sessao sessaoAtual) {
 Stream<Sessao> stream = sessoesDaSala
 .stream();

 Stream<Boolean> booleanStream = stream
 .map(sessaoExistente -> horarioIsValidado(sessaoExistente, sessaoAtual));

 booleanStream.reduce(Boolean::logicalAnd)

 // restante do código
}
```

Queremos que seja devolvido um `Boolean` e que seja feita uma redução a cada dois booleanos utilizando o operador lógico `&&`, mas o que vai acontecer se nossa lista estiver vazia? É uma situação fora do esperado e em algum momento receberemos uma exception. Para evitar isso, uma solução bem conhecida por nós desenvolvedores é fazer uma validação na lista antes de usá-la no método `cabe`. No entanto, no Java 8 há uma forma mais elegante de resolvemos esse impasse. O próprio método `reduce` nos devolve um objeto do tipo `Optional`, que por trás dos panos já realiza esta verificação implicitamente:

```
public boolean cabe(Sessao sessaoAtual) {
 Stream<Sessao> stream = sessoesDaSala
 .stream();

 Stream<Boolean> booleanStream = stream
 .map(sessaoExistente -> horarioIsValidado(sessaoExistente, sessaoAtual));

 Optional<Boolean> optionalCabe = booleanStream.reduce(Boolean::logicalAnd)

 return optionalCabe.orElse(true);
}
```

Ao utilizarmos o método `orElse` ocorre uma verificação para checar se existe qualquer valor, no nosso caso, booleano, atribuído ao `Optional` e, caso não exista, estamos definindo que o valor padrão `true` deverá ser retornado.

## 2.6 TESTES DE UNIDADE

**Testes de unidade** são testes que testam apenas uma classe ou método, verificando se seu

comportamento está de acordo com o desejado. Em testes de unidade, verificamos a funcionalidade da classe e/ou método em questão passando o mínimo possível por outras classes ou dependências do nosso sistema.

#### UNIDADE

Unidade é a menor parte testável de uma aplicação. Em uma linguagem de programação orientada a objetos como o Java, a menor unidade é um método.

O termo correto para esses testes é **testes de unidade**, porém o termo *teste unitário* propagou-se e é o mais encontrado nas traduções.

Em testes de unidade, não estamos interessados no comportamento real das dependências da classe, mas em como a classe em questão se comporta diante das possíveis respostas das dependências, ou então, se a classe modificou as dependências da maneira esperada.

Para isso, quando criamos um teste de unidade, simulamos a execução de métodos da classe a ser testada. Fazemos isso passando parâmetros, caso necessário, ao método testado e definimos o resultado que esperamos. Se o resultado for igual ao que definimos como esperado, o teste passa. Caso contrário, falha.

#### ATENÇÃO

Muitas vezes, principalmente quando estamos iniciando no mundo dos testes, é comum criarmos alguns testes que testam muito mais do que o necessário, mais do que apenas a unidade. Tais testes se transformam em verdadeiros **testes de integração** (esses sim são responsáveis por testar o sistemas como um todo).

Portanto, lembre-se sempre: testes de unidade testam **apenas** unidades!

## 2.7 JUNIT

O **JUnit** ([junit.org](http://junit.org)) é um framework muito simples que facilita a criação destes testes de unidade e em especial sua execução. Ele possui alguns métodos que tornam seu código de teste bem legível e fácil de fazer as **asserções**.

Uma **asserção** é uma afirmação: alguma invariante que em determinado ponto de execução você quer garantir que é verdadeira. Se aquilo não for verdade, o teste deve indicar uma falha, a ser reportada

para o programador, indicando um possível bug.

À medida que você mexe no seu código, você roda novamente toda aquela bateria de testes com um comando apenas. Com isso você ganha a confiança de que novos bugs não estão sendo introduzidos (ou reintroduzidos) conforme você cria novas funcionalidades e conserta antigos bugs. Mais fácil do que ocorre quando fazemos os testes dentro do `main`, executando um por vez.

O JUnit possui integração com todas as grandes IDEs, além das ferramentas de build, que vamos conhecer mais a frente. Vamos agora entender um pouco mais sobre anotações e o `import` estático, que vão facilitar muito o nosso trabalho com o JUnit.

## Usando o JUnit - configurando Classpath e seu JAR no Eclipse

O JUnit é uma biblioteca escrita por terceiros que vamos usar no nosso projeto. Precisamos das classes do JUnit para escrever nossos testes. E, como sabemos, o formato de distribuição de bibliotecas Java é o JAR, muito similar a um ZIP com as classes daquela biblioteca.

Precisamos então do JAR do JUnit no nosso projeto. Mas quando rodarmos nossa aplicação, como o Java vai saber que deve incluir as classes daquele determinado JAR (dependência) junto com nosso programa?

É aqui que o **Classpath** entra na história: é nele que definimos qual o "*caminho para buscar as classes que vamos usar*". Temos que indicar onde a JVM deve buscar as classes para compilar e rodar nossa aplicação.

Há algumas formas de configurarmos o *classpath*:

- Configurando uma variável de ambiente (**desaconselhado**);
- Passando como argumento em linha de comando (**trabalhoso**);
- Utilizando ferramentas como Ant e Maven (como já está sendo feito);
- Deixando o eclipse configurar por você (como fizemos nos cursos anteriores).

No nosso caso, o próprio *Maven* já está gerenciando as dependências através do arquivo *pom.xml*. Portanto basta adicionarmos essa uma nova dependência indicando ao *Maven* a nova dependência:

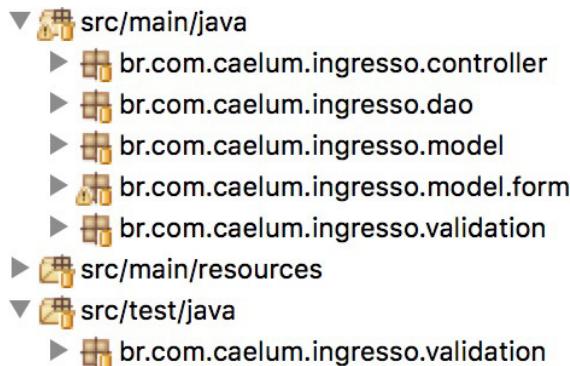
```
<dependency>
 <groupId>junit</groupId>
 <artifactId>junit</artifactId>
 <version>4.12</version>
</dependency>
```

## 2.8 FAZENDO NOSSO PRIMEIRO TESTE

Nosso `GerenciadorDeSessao` tem uma lógica bem crucial para nossa aplicação, pois é através dele que poderemos armazenar nossas sessões e não seria legal descobrirmos um bug em produção, pois

poderia acarretar à empresa um grande prejuízo. Para evitar essa situação, vamos entregar um software com mais qualidade, onde tudo esteja funcionando corretamente. E, para podermos garantir que tudo estará conforme esperamos é necessário gerarmos testes.

Para isso, geraremos nossa primeira classe de teste. Por convenção, ela deverá ficar na mesma pasta que a classe que estamos testando, contudo, no *source folder* de teste do nosso projeto (*src/test/java*):



Outra convenção que teremos é no nome da classe, ela terá o mesmo nome da classe que estamos testando acrescido do sufixo **Test**:

```
public class GerenciadorDeSessaoTest {
}
```

Com nossa classe de teste já definida, basta criarmos nossos testes. Sendo assim, precisaremos definir quais serão os cenários que vamos cobrir. Por exemplo :

- Não sermos capazes de adicionar uma sessão no mesmo horário de começo de outra.
- Não podermos adicionar uma sessão cuja duração conflite com o horário da próxima sessão.
- Não podermos adicionar uma sessão com inicio que conflite com a duração da sessão que está passando.
- Podermos adicionar se não tiver nenhuma sessão.
- Podermos adicionar se o horário não conflitar com o horário da sessão anterior.
- Podermos adicionar se a duração da sessão não conflitar com o horário da próxima sessão.

Vamos iniciar pelo teste mais simples, que é validar se podemos adicionar uma Sessão se a lista estiver vazia. Para isso, precisaremos criar este cenário, através de um método anotado com `@Test` em nossa classe. A annotation `@Test` diz para o JUnit que o método deve ser interpretado como um teste.

```
public class GerenciadorDeSessaoTest {
 @Test
 public void deveAdicionarSeListaDaSessaoEstiverVazia(){
 }
```

```
}
```

Agora, será necessário criar o **cenário** para que o nosso teste seja executado dentro do método, desse modo, vamos criar uma lista vazia e passar para o nosso `GerenciadorDeSessao` :

```
public class GerenciadorDeSessaoTest {

 @Test
 public void deveAdicionarSeListaDaSessaoEstiverVazia(){
 List<Sessao> sessoes = Collections.emptyList();
 GerenciadorDeSessao gerenciador = new GerenciadorDeSessao(sessoes);

 }

}
```

Precisamos ainda criar nossa `Sessao` e pedir para nosso `GerenciadorDeSessao` a validar:

```
public class GerenciadorDeSessaoTest {

 @Test
 public void deveAdicionarSeListaDaSessaoEstiverVazia(){
 List<Sessao> sessoes = Collections.emptyList();
 GerenciadorDeSessao gerenciador = new GerenciadorDeSessao(sessoes);

 Filme filme = new Filme("Rogue One", Duration.ofMinutes(120), "SCI-FI", BigDecimal.ONE);
 filme.setDuracao(120);
 LocalTime horario = LocalTime.parse("10:00:00");
 Sala sala = new Sala("");

 Sessao sessao = new Sessao(horario, filme, sala);

 boolean cabe = gerenciador.cabe(sessao);

 }

}
```

O cenário do nosso teste está pronto, mas repare que em nenhum momento fizemos alguma validação para checar se tudo deu certo como imaginávamos que deveria. Portanto, faremos a parte mais importante do teste : **Verificações**. Usaremos a classe `Assert` que é uma especialista nisto. Como sabemos que o resultado do nosso teste deve ser `true` , então, vamos pedir para ela verificar isso:

```
public class GerenciadorDeSessaoTest {

 @Test
 public void deveAdicionarSeListaDaSessaoEstiverVazia(){
 List<Sessao> sessoes = Collections.emptyList();
 GerenciadorDeSessao gerenciador = new GerenciadorDeSessao(sessoes);

 Filme filme = new Filme("Rogue One", Duration.ofMinutes(120), "SCI-FI", BigDecimal.ONE);
 filme.setDuracao(120);
 LocalTime horario = LocalTime.parse("10:00:00");
 Sala sala = new Sala("");

 Sessao sessao = new Sessao(horario, filme, sala);

 boolean cabe = gerenciador.cabe(sessao);

 }

}
```

```

 Assert.assertTrue(cabe);
 }

}

```

Como iremos realizar os testes de alguns cenários e em todos eles usaremos sessões, a criação desses objetos se repetiria em cada um dos testes. Para evitarmos isso, podemos indicar para o *JUnit* que algum método deve ser executado antes mesmo dos testes, e assim podemos preparar os atributos e reutilizá-los em todos os casos de teste.

Para isso, basta criarmos um método *void* e anotá-lo com `@Before`:

```

public class GerenciadorDeSessaoTest {

 private Filme rogueOne;
 private Sala sala3D;
 private Sessao sessaoDasDez;
 private Sessao sessaoDasTreze;
 private Sessao sessaoDasDezoito;

 @Before
 public void preparaSessoes(){

 this.rogueOne = new Filme("Rogue One", Duration.ofMinutes(120), "SCI-FI");
 this.sala3D = new Sala("Sala 3D");

 this.sessaoDasDez = new Sessao(LocalTime.parse("10:00:00"), rogueOne, sala3D);
 this.sessaoDasTreze = new Sessao(LocalTime.parse("13:00:00"), rogueOne, sala3D);
 this.sessaoDasDezoito = new Sessao(LocalTime.parse("18:00:00"), rogueOne, sala3D);
 }

 // Métodos de teste omitidos
}

```

## 2.9 EXERCÍCIO - GARANTINDO QUE A VALIDAÇÃO DE HORÁRIOS PARA CADASTRAR UMA SESSÃO ESTÁ CORRETA

- Crie uma nova branch para começar a trabalhar nessa funcionalidade:

```
git checkout -b work
```

- Precisamos verificar se o horário da sessão que estamos gravando não irá encavalgar com o horário de uma sessão já existente. Sendo assim, vamos criar uma classe que faça essa validação.

Crie a classe `GerenciadorDeSessao` no pacote `br.com.caelum.ingresso.validacao` e, baseado em uma lista de sessões de uma sala, verifique se o horário da sessão que queremos gravar cabe nessa sala.

```

public class GerenciadorDeSessao {

 private List<Sessao> sessoesDaSala;

 public GerenciadorDeSessao(List<Sessao> sessoesDaSala) {
 this.sessoesDaSala = sessoesDaSala;
 }
}

```

```

private boolean horarioisValido(Sessao sessaoExistente, Sessao sessaoAtual) {

 LocalDate hoje = LocalDate.now();

 LocalDateTime horarioSessao = sessaoExistente.getHorario().atDate(hoje);
 LocalDateTime horarioAtual = sessaoAtual.getHorario().atDate(hoje);

 boolean ehAntes = horarioAtual.isBefore(horarioSessao);

 if (ehAntes) {

 return horarioAtual
 .plus(sessaoAtual.getFilme().getDuracao())
 .isBefore(horarioSessao);
 } else {

 return horarioSessao
 .plus(sessaoExistente.getFilme().getDuracao())
 .isBefore(horarioAtual);
 }
}

public boolean cabe(Sessao sessaoAtual) {

 Optional<Boolean> optionalCabe = sessoesDaSala
 .stream()
 .map(sessaoExistente ->
 horarioisValido(sessaoExistente, sessaoAtual)
)
 .reduce(Boolean::logicalAnd);

 return optionalCabe.orElse(true);
}
}

```

3. Precisamos garantir que a validação que acabamos de implementar está funcionando. Para isso, criaremos alguns testes unitários que irão validar os cenários que previmos utilizando *Junit*. Para isso, adicione uma nova dependência no arquivo *pom.xml* para que o *Maven* adicione as bibliotecas do framework ao nosso projeto:

```

<dependency>
 <groupId>junit</groupId>
 <artifactId>junit</artifactId>
 <version>4.12</version>
</dependency>

```

4. Crie a classe `GerenciadorDeSessaoTest` no pacote `br.com.caelum.ingresso.validacao`, porém, no *source folder* de teste (*src/test/java*). Caso as pastas *test* e *java* não existam, crie elas primeiro. A anotação `@Test` e a classe `Assert` devem ser importadas do pacote `org.junit`. Para não repetirmos a criação dos objetos usados para definir os cenários, utilizaremos a anotação `@Before`:

```

public class GerenciadorDeSessaoTest {

 private Filme rogueOne;
 private Sala sala3D;

```

```

private Sessao sessaoDasDez;
private Sessao sessaoDasTreze;
private Sessao sessaoDasDezoito;

@Before
public void preparaSessoes(){

 this.rogueOne = new Filme("Rogue One", Duration.ofMinutes(120), "SCI-FI");
 this.sala3D = new Sala("Sala 3D");

 this.sessaoDasDez = new Sessao(LocalTime.parse("10:00:00"), rogueOne, sala3D);
 this.sessaoDasTreze = new Sessao(LocalTime.parse("13:00:00"), rogueOne, sala3D);
 this.sessaoDasDezoito = new Sessao(LocalTime.parse("18:00:00"), rogueOne, sala3D);
}

@Test
public void garanteQueNaoDevePermitirSessaoNoMesmoHorario() {

 List<Sessao> sessoes = Arrays.asList(sessaoDasDez);
 GerenciadorDeSessao gerenciador = new GerenciadorDeSessao(sessoes);
 Assert.assertFalse(gerenciador.cabe(sessaoDasDez));
}

@Test
public void garanteQueNaoDevePermitirSessoesTerminandoDentroDoHorarioDeUmaSessaoJaExistente()
{
 List<Sessao> sessoes = Arrays.asList(sessaoDasDez);
 Sessao sessao = new Sessao(sessaoDasDez.getHorario().minusHours(1), rogueOne, sala3D);
 GerenciadorDeSessao gerenciador = new GerenciadorDeSessao(sessoes);
 Assert.assertFalse(gerenciador.cabe(sessao));

}

@Test
public void garanteQueNaoDevePermitirSessoesIniciandoDentroDoHorarioDeUmaSessaoJaExistente()
{
 List<Sessao> sessoesDaSala = Arrays.asList(sessaoDasDez);
 GerenciadorDeSessao gerenciador = new GerenciadorDeSessao(sessoesDaSala);
 Sessao sessao = new Sessao(sessaoDasDez.getHorario().plusHours(1), rogueOne, sala3D);
 Assert.assertFalse(gerenciador.cabe(sessao));

}

@Test
public void garanteQueDevePermitirUmaInsercaoEntreDoisFilmes(){
 List<Sessao> sessoes = Arrays.asList(sessaoDasDez, sessaoDasDezoito);
 GerenciadorDeSessao gerenciador = new GerenciadorDeSessao(sessoes);
 Assert.assertTrue(gerenciador.cabe(sessaoDasTreze));
}
}

```

5. Faça o commit do que foi feito até agora na branch work. Em seguida, execute a aplicação e tente adicionar algumas sessões com horários conflitantes para garantir que a validação está ocorrendo.
6. Com a validação de inserção de sessões finalizada e testada, podemos juntar o que foi feito na branch work para a branch master através do rebase.

# ADICIONANDO PREÇO

## 3.1 MELHORANDO A MODELAGEM DO SISTEMA

Agora que temos bastante coisa funcionando, precisamos deixar nosso sistema fazer algo bem importante para a regra de negócio, que é poder ganhar por sessão comprada.

Nossa regra de negócio diz que cada filme deve ter seu preço, para poder ser algo dinâmico e rentável para o cinema, onde filmes que tendam a ter mais audiência devem ter um preço superior. Logo, precisaremos ter um atributo preço em nosso modelo `Filme`. Mas, qual será o seu tipo?

```
@Entity
public class Filme {

 @Id
 @GeneratedValue
 private Integer id;
 private String nome;
 private Duration duracao;
 private String genero;
 private ??? preco;
}
```

Poderíamos facilmente colocar como `double`, contudo, existem alguns problemas nisso! O principal deles é o fato do `double` ser **inexato**. Dado este problema corriqueiro, foi criada uma classe que cuida da exatidão do valor, a classe `BigDecimal`.

```
@Entity
public class Filme {

 @Id
 @GeneratedValue
 private Integer id;
 private String nome;
 private Duration duracao;
 private String genero;
 private BigDecimal preco;
}
```

Algo bem importante que deve ser observado é que não podemos deixar um filme estar sem preço em nenhum lugar que estivermos o manipulando. Portanto, teremos que forçar o `Filme` a ter preço e, para isso, atribuiremos essa informação no construtor da classe:

```

public Filme(String nome, Duration duracao, String genero, BigDecimal preco) {
 this.nome = nome;
 this.duracao = duracao;
 this.genero = genero;
 this.preco = preco;
}

```

Agora, precisamos adicionar o campo preço tanto na listagem de filmes quanto no formulário de cadastro de filme para que possamos obter e disponibilizar essas informações.

```

<!-- restante do form -->
<div class="form-group">
 <label for="preco">Preço:</label>
 <input id="preco" type="text" name="preco" class="form-control"
 value="${filme.preco}">
 <c:forEach items="${bindingResult.getFieldErrors('preco')}" var="error">
 ${error.defaultMessage}
 </c:forEach>
</div>

<!-- botão de gravar -->

```

Além disso, temos outra regra de negócio muito importante que define que cada sala deve possuir seu preço, já que podemos ter salas onde há um diferencial como ser 3D, iMax, fora a parte de manutenção. Portanto, a sala também deverá ter seu próprio preço.

Já sabemos o que é necessário para isso : adicionar o atributo, o receber através do construtor e atualizar as telas.

Algo que ainda não falamos mas acaba ficando implícito é : ao irmos ao cinema não pagamos pela sala ou pelo filme, mas sim pela sessão, então, precisamos fazer com que a sessão tenha seu preço correto, que deve ser a soma dos preços da sala e do filme:

```

@Entity
public class Sessao {

 // demais métodos

 private BigDecimal preco;

 public Sessao(LocalTime horario, Filme filme, Sala sala) {
 this.horario = horario;
 this.setFilme(filme);
 this.sala = sala;
 this.preco = sala.getPreco().add(filme.getPreco());
 }

 public BigDecimal getPreco(){
 return preco;
 }
}

```

## 3.2 EXERCÍCIO - COLOCANDO PREÇO NA SALA E FILME

- Crie uma nova branch para a nova funcionalidade.

2. Adicione um atributo preço do tipo `BigDecimal` na classes `Sala` . Crie os métodos `get` e `set` para o mesmo em ambas as classes e adicione um novo parâmetro nos construtores para receber um valor para o novo atributo:

```
@Entity
public class Sala {

 // Demais atributos

 private BigDecimal preco;

 /**
 * @deprecated hibernate only
 */
 public Sala() {
 }

 public Sala(String nome, BigDecimal preco) {
 this.nome = nome;
 this.preco = preco;
 }

 //getters e setters e demais métodos
}
```

3. Faça o mesmo para a classe `Filme` :

```
@Entity
public class Filme {

 // Demais atributos

 private BigDecimal preco;

 /**
 * @deprecated hibernate only
 */
 public Filme() {
 }

 public Filme(String nome, Duration duracao, String genero, BigDecimal preco) {
 this.nome = nome;
 this.duracao = duracao;
 this.genero = genero;
 this.preco = preco;
 }

 //getters e setters e demais métodos
}
```

4. Ao adicionar o novo parâmetro ao construtor (ou o próprio construtor com todos os parâmetros), quebramos nossa classe `GerenciadorDeSessaoTest` . No método de preparação para os testes, vamos passar o novo parâmetro no construtor da `Sala` e do `Filme`:

```
public class GerenciadorDeSessaoTest {

 @Before
 public void preparaSessoes(){
```

```

 this.rogueOne = new Filme("Rogue One", Duration.ofMinutes(120), "SCI-FI", BigDecimal.ONE);
 this.sala3D = new Sala("Sala 3D", BigDecimal.TEN);

 this.sessaoDasDez = new Sessao(LocalTime.parse("10:00:00"), rogueOne, sala3D);
 this.sessaoDasTreze = new Sessao(LocalTime.parse("13:00:00"), rogueOne, sala3D);
 this.sessaoDasDezoito = new Sessao(LocalTime.parse("18:00:00"), rogueOne, sala3D);
 }

}

```

5. Vamos alterar as páginas de formulário e listagem da Sala e do Filme para adicionar o campo de preço. Use o atalho do Eclipse Ctrl + Shift + C para remover ou adicionar comentários:

- No arquivo `src/main/webapp/WEB-INF/views/sala/sala.jsp`, remova os comentários ao redor do input para o preço.
- No arquivo `src/main/webapp/WEB-INF/views/sala/lista.jsp`, remova os comentários para as TAGs `<th>` e `<td>` para a coluna preço:

```

<th class="text-center">Preço</th>

<td class="text-center">${sala.preco}</td>

```

- No arquivo `src/main/webapp/WEB-INF/views/filme/filme.jsp`, remova os comentários ao redor do input para o preço.
- No arquivo `src/main/webapp/WEB-INF/views/filme/lista.jsp`, remova os comentários para as TAGs `<th>` e `<td>` para a coluna preço:

```

<th>Preço</th>

<td>${filme.preco}</td>

```

6. Adicione um novo atributo preço do tipo `BigDecimal` na classe `Sessao` e, no construtor da classe, atribua a ele o resultado da soma dos preços da Sala e do Filme. Não se esqueça de criar os métodos get e set do novo atributo, para que possamos recuperar seu valor futuramente:

```

@Entity
public class Sessao {

 // Demais atributos

 private BigDecimal preco;

 /**
 * @deprecated hibernate only
 */
 public Sessao() {
 }

 public Sessao(LocalTime horario, Filme filme, Sala sala) {
 this.horario = horario;
 this.filme = filme;
 this.sala = sala;
 this.preco = sala.getPreco().add(filme.getPreco());
 }
}

```

```
// demais getters e setters
}
```

7. Altere o arquivo de listagem da sessão `src/main/webapp/WEB-INF/views/sessao/lista.jsp`, removendo os comentários para as TAGs `<th>` e `<td>` para a coluna preço.
8. Rode a aplicação e verifique se o preço da sessão é corretamente exibido na tela do sistema.
9. Crie um teste na pasta `src/test/java` no pacote `br.com.caelum.ingresso.model` para garantir que a sessão retorna a soma dos preços da Sala e Filme :

```
public class SessaoTest {

 @Test
 public void oPrecoDaSessaoDeveSerIgualASomaDoPrecoDaSalaMaisOPrecoDoFilme() {

 Sala sala = new Sala("Eldorado - IMax", new BigDecimal("22.5"));
 Filme filme = new Filme("Rogue One", Duration.ofMinutes(120),
 "SCI-FI", new BigDecimal("12.0"));

 BigDecimal somaDosPrecosDaSalaEFilme = sala.getPreco().add(filme.getPreco());

 Sessao sessao = new Sessao(LocalTime.parse("10:00:00"), filme, sala);

 Assert.assertEquals(somaDosPrecosDaSalaEFilme, sessao.getPreco());
 }
}
```

10. Execute o teste, commit seu trabalho e em seguida junte o que foi feito na branch work para a branch master através do rebase.

### 3.3 APPLICANDO STRATEGY

No último passo nós adicionamos o preço à sessão, mas quando vamos ao cinema não compramos uma sessão e sim um ingresso para poder ingressar numa sessão. Como o ingresso tem um peso bem grande em nosso sistema, é interessante que tenhamos uma classe para representá-lo. Também é necessário definir quais serão os atributos que nosso `Ingresso` vai possuir. Precisaremos ao menos da própria `Sessão` e de um preço:

```
public class Ingresso {

 private Sessao sessao;
 private BigDecimal preco;
}
```

Quando é feita a compra de um ingresso, o preço pode sofrer um desconto, por exemplo, para estudante, bancos ou outras promoções. Mas nosso sistema ainda não está preparado para tratar

descontos.

Uma maneira bem elegante de definirmos que cada desconto possua sua própria regra de negócio e ainda assim faça tudo que um desconto precisa fazer é criarmos uma interface `Desconto`:

```
public interface Desconto {
 public BigDecimal aplicarDescontoSobre(BigDecimal precoOriginal);
}
```

Agora, podemos criar nossos descontos, por exemplo, para estudante:

```
public class DescontoParaEstudantes implements Desconto {

 private BigDecimal metade = new BigDecimal(2.0);

 @Override
 public BigDecimal aplicarDescontoSobre(BigDecimal precoOriginal) {
 return precoOriginal.divide(metade);
 }
}
```

Desta maneira, nosso ingresso passa a poder usar o `Desconto` para calcular o preço! E, assim que construirmos nosso objeto `Ingresso`, devemos informar qual é o `Desconto` utilizado:

```
public class Ingresso {
 this.sessao = sessao;

 private Sessao sessao;
 private BigDecimal preco;

 public Ingresso(Sessao sessao, Desconto desconto) {
 this.preco = desconto.aplicarDescontoSobre(sessao.getPreco());
 }

 //getters
}
```

Com essa nova implementação, não importa qual é o desconto que será passado, mas que ele execute a regra de negócio, ainda que tenhamos um desconto que realmente não aplique desconto.

A forma como resolvemos esse impasse, usando **polimorfismo** para permitir passar qualquer classe que implemente uma interface, é uma solução bem conhecida pela comunidade, tanto que é um *Design Pattern* conhecido como **Strategy**.

### 3.4 EXERCÍCIO - CRIANDO DESCONTOS E INGRESSO

1. Crie uma nova branch para a nova funcionalidade.
2. Crie a interface `Desconto` no pacote `br.com.caelum.ingresso.model.descontos`:

```

public interface Desconto {
 BigDecimal aplicarDescontoSobre(BigDecimal precoOriginal);
}

```

3. Crie a classe com implementação de desconto para Estudantes no pacote br.com.caelum.ingresso.model.descontos :

```

public class DescontoParaEstudantes implements Desconto {

 @Override
 public BigDecimal aplicarDescontoSobre(BigDecimal precoOriginal) {
 return precoOriginal.divide(new BigDecimal("2.0"));
 }
}

```

4. Crie a classe com implementação de desconto para Bancos:

```

public class DescontoParaBancos implements Desconto {

 @Override
 public BigDecimal aplicarDescontoSobre(BigDecimal precoOriginal) {
 return precoOriginal.subtract(trintaPorCentoSobre(precoOriginal));
 }

 private BigDecimal trintaPorCentoSobre(BigDecimal precoOriginal) {
 return precoOriginal.multiply(new BigDecimal("0.3"));
 }
}

```

5. Ainda é necessário criarmos outra implementação, para casos onde não haverá desconto:

```

public class SemDesconto implements Desconto {

 @Override
 public BigDecimal aplicarDescontoSobre(BigDecimal precoOriginal) {
 return precoOriginal;
 }
}

```

6. Crie a classe Ingresso no pacote br.com.caelum.ingresso.model com atributos Sessao e Preco . Crie um construtor que receba uma Sessao e um Desconto como parâmetros. Faça com que o preço do ingresso seja o resultado da aplicação do desconto sobre o preço da sessão. Crie, também, um construtor sem parâmetros, para que nossa classe possa ser utilizada futuramente pelo Hibernate:

```

public class Ingresso {

 private Sessao sessao;
 private BigDecimal preco;

 /**
 * @deprecated hibernate only
 */
 public Ingresso(){
 }
}

```

```

public Ingresso(Sessao sessao, Desconto tipoDeDesconto) {
 this.sessao = sessao;
 this.preco = tipoDeDesconto.aplicarDescontoSobre(sessao.getPreco());
}

public Sessao getSessao() {
 return sessao;
}

public BigDecimal getPreco() {
 return preco;
}
}

```

## 3.5 EXERCÍCIO - TESTANDO UM DESCONTO

- Crie uma classe de teste `DescontoTest` no pacote `br.com.caelum.ingresso.model.desconto` na pasta `src/test/java`, garantindo que a lógica de ingressos sem desconto está sendo aplicada corretamente :

```

public class DescontoTest {

 @Test
 public void naoDeveConcederDescontoParaIngressoNormal() {

 Sala sala = new Sala("Eldorado - IMAX", new BigDecimal("20.5"));
 Filme filme = new Filme("Rogue One", Duration.ofMinutes(120),
 "SCI-FI", new BigDecimal("12"));
 Sessao sessao = new Sessao(LocalTime.parse("10:00:00"), filme, sala);
 Ingresso ingresso = new Ingresso(sessao, new SemDesconto());

 BigDecimal precoEsperado = new BigDecimal("32.5");

 Assert.assertEquals(precoEsperado, ingresso.getPreco());
 }
}

```

- Reinicie a aplicação e verifique os resultados dos testes para garantir que os descontos estão sendo aplicados corretamente. Em seguida, junte o que foi feito na branch nova para a branch master através do rebase.

## 3.6 EXERCÍCIO OPCIONAL - TESTANDO OS DEMAIS DESCONTOS

- Na classe `DescontoTest` crie mais dois testes que garantam que os demais descontos estão sendo aplicados corretamente.

```

public class DescontoTest {

 @Test
 public void deveConcederDescontoDe30PorcentoParaIngressosDeClientesDeBancos() {

 Sala sala = new Sala("Eldorado - IMAX", new BigDecimal("20.5"));
 Filme filme = new Filme("Rogue One", Duration.ofMinutes(120),
 "SCI-FI", new BigDecimal("12"));

```

```

 "SCI-FI", new BigDecimal("12"));
Sessao sessao = new Sessao(LocalTime.parse("10:00:00"), filme, sala);
Ingresso ingresso = new Ingresso(sessao, new DescontoParaBancos());

BigDecimal precoEsperado = new BigDecimal("22.75");

Assert.assertEquals(precoEsperado, ingresso.getPreco());

}

@Test
public void deveConcederDescontoDe50PorcentoParaIngressoDeEstudante() {

 Sala sala = new Sala("Eldorado - IMAX", new BigDecimal("20.5"));
 Filme filme = new Filme("Rogue One", Duration.ofMinutes(120),
 "SCI-FI", new BigDecimal("12"));
 Sessao sessao = new Sessao(LocalTime.parse("10:00:00"), filme, sala);
 Ingresso ingresso = new Ingresso(sessao, new DescontoParaEstudantes());

 BigDecimal precoEsperado = new BigDecimal("16.25");

 Assert.assertEquals(precoEsperado, ingresso.getPreco());
}
}

```

2. **Desafio:** Utilize o `@Before` para reduzir a repetição de código nos testes.

## CAPÍTULO 4

# MELHORANDO A USABILIDADE DA APLICAÇÃO

## 4.1 DEFININDO O CATÁLOGO DE FILMES E A TELA DE DETALHES

Para que os usuários possam escolher o filme para o qual desejam comprar ingressos, podemos disponibilizar uma tela que exibirá um catálogo de todos os filmes disponíveis.



Portanto, visando possibilitar que o usuário accesse essa tela, teremos que disponibilizar um novo mapeamento em nossa classe `FilmeController` e adicionar um novo link ao menu superior do sistema, o qual fará a requisição para o novo endereço mapeado:

```
public class FilmeController{
 //Atributos omitidos

 @GetMapping("/filme/em-cartaz")
 public ModelAndView emCartaz(){
 ModelAndView modelAndView = new ModelAndView("filme/em-cartaz");

 modelAndView.addObject("filmes", filmeDao.findAll());

 return modelAndView;
 }
}
```

```
//Demais métodos omitidos
}
```

Ao ver nosso catálogo de filmes, o usuário naturalmente irá clicar em uma das opções e vai esperar que seja exibido mais detalhes sobre aquele filme, como, por exemplo, a duração, elenco, sinopse, banner e, até mesmo, uma breve avaliação do filme.

Para tanto, vamos criar essa nova tela que vai apresentar os detalhes do filme. A tela será composta basicamente por labels para descrever os dados que iremos inserir futuramente:

```
<div class=" col-md-6 col-md-offset-3">
 <h1>Título</h1>
 <image src="" />

 <div>
 <label for="ano">Ano</label>

 </div>

 <div>
 <label for="diretores">Diretores</label>

 </div>

 <div>
 <label for="escritores">Escritores</label>

 </div>

 <div>
 <label for="atores">Atores</label>

 </div>

 <div>
 <label for="descricao">Descrição</label>

 </div>

 <div>
 <label for="avaliacao">Avaliação</label>

 </div>

 <table class="table table-hover">
 <thead>
 <th>Sala</th>
 <th>Horário</th>
 <th>Ações</th>
 </thead>
 <tbody>
 <c:forEach items="${sessoes}" var="sessao">
 <tr>
 <td>${sessao.sala.nome}</td>
 <td>${sessao.horario}</td>
 <td>

 Comprar

 </td>
 </tr>
 </c:forEach>
 </tbody>
 </table>
</div>
```

```


</td>
</tr>
</c:forEach>
</tbody>
</table>
</div>

```

Por fim, precisamos fazer com que nosso controller disponibilize a lista de sessões do filme para a tela:

```

@GetMapping("/filme/{id}/detalhe")
public ModelAndView detalhes(@PathVariable("id") Integer id){
 ModelAndView modelAndView = new ModelAndView("/filme/detalhe");

 Filme filme = filmeDao.findOne(id);
 List<Sessao> sessoes = sessaoDao.buscaSessoesDoFilme(filme);

 modelAndView.addObject("sessoes", sessoes);

 return modelAndView;
}

```

## 4.2 EXERCÍCIO - CRIANDO O CATÁLOGO DE FILMES E TELA DE DETALHES DO FILME COM SESSÕES PARA COMPRA

1. Altere a classe `FilmeController` e adicione o método `emCartaz` :

```

@GetMapping("/filme/em-cartaz")
public ModelAndView emCartaz(){
 ModelAndView modelAndView = new ModelAndView("filme/em-cartaz");

 modelAndView.addObject("filmes", filmeDao.findAll());

 return modelAndView;
}

```

2. Altere o arquivo `src/main/webapp/WEB-INF/tags/template.tag` e remova o comentário para que no menu tenha um link para acesso a `/filme/em-cartaz` :

```


- Filmes
- Salas
- Comprar


```

3. Vamos criar o método `buscaSessoesDoFilme` em `SessaoDao` :

```

public List<Sessao> buscaSessoesDoFilme(Filme filme) {
 return manager.createQuery("select s from Sessao s where s.filme = :filme", Sessao.class)
 .setParameter("filme", filme)

```

```
 .getResultSet();
 }
```

4. Injete SessaoDao no FilmeController para poder buscar as sessões:

```
@Controller
public class FilmeController {

 @Autowired
 private FilmeDao filmeDao;
 @Autowired
 private SessaoDao sessaoDao;

 .
 .
}
```

5. Vamos criar uma action em FilmeController para atender as requisições *GET* para /filme/{id}/detalhe :

```
@GetMapping("/filme/{id}/detalhe")
public ModelAndView detalhes(@PathVariable("id") Integer id){
 ModelAndView modelAndView = new ModelAndView("/filme/detalhe");

 Filme filme = filmeDao.findOne(id);
 List<Sessao> sessoes = sessaoDao.buscaSessoesDoFilme(filme);

 modelAndView.addObject("sessoes", sessoes);

 return modelAndView;
}
```

6. Reinicie a aplicação e acesse a página de filmes em cartaz. Clique em algum filme disponível e visualize a tela de detalhes do mesmo.

## 4.3 TRAZENDO DADOS REAIS PARA NOSSA APLICAÇÃO

Nossa tela de detalhes já está criada ( `src/main/webapp/WEB-INF/views/filme/detalhe.jsp` ), no entanto, nós ainda não possuímos as informações necessárias relativas ao filme selecionado para passar para ela. Para nos ajudar a fazer isso, existe uma API bem conhecida que pode ser consumida, a OMDB.

A primeira coisa que é necessário fazermos é a requisição. O próprio Spring já tem uma classe específica que pode nos ajudar, a classe `RestTemplate`. Para podermos utilizá-la, basta a instanciarmos:

```
RestTemplate client = new RestTemplate();
```

Agora, é necessário apenas fazermos a requisição:

```
DetalhesDoFilme detalhesDoFilme = client.getForObject(url, DetalhesDoFilme.class);
```

Vamos entender melhor o que aconteceu nessa última linha! Deixamos claro para o `RestTemplate` que ele precisa fazer uma requisição do tipo *Get* e que essa requisição irá devolver um objeto que nós

temos, nesse caso o `DetalhesDoFilme`.

A classe `DetalhesDoFilme` vai representar a resposta e, por isso, precisamos informar o que ela deve conhecer. Em outras palavras, precisamos informar como fazer o *binding* das chaves da resposta para cada atributo dela. Para facilitar esse trabalho, usaremos uma anotação que o `Spring` já utiliza por trás dos panos e é provida pelo `Jackson`, um especialista em interpretar e criar jsons.

```
public class DetalhesDoFilme {

 @JsonProperty("Title")
 private String titulo;

 @JsonProperty("Year")
 private Integer ano;

 @JsonProperty("Poster")
 private String imagem;

 @JsonProperty("Director")
 private String diretores;

 @JsonProperty("Writer")
 private String escritores;

 @JsonProperty("Actors")
 private String atores;

 @JsonProperty("Plot")
 private String descricao;

 @JsonProperty("imdbRating")
 private Double avaliacao;

 // getters e setters
}
```

Voltando para nossa requisição, precisamos saber se realmente houve resposta, porque pode ser que aquela API não tenha nenhum dado sobre o filme que estamos fazendo a busca. Portanto, vamos usar a mesma ideia de `Optional` para seguir com a estrutura do projeto.

```
DetalhesDoFilme detalhesDoFilme = client.getForObject(url, DetalhesDoFilme.class);
Optional.of(detalhesDoFilme);
```

Existe também a possibilidade de fazer a requisição e ter algum problema, seja indisponibilidade da API, estarmos sem internet ou qualquer outro problema nessa requisição. Por conta disso, é necessário precaver esse comportamento que gerará uma `Exception` e, nesse caso, também precisaremos definir um retorno para o usuário. Como estamos trabalhando com `Optional`, devolveremos um `Optional` vazio:

```
try {

 DetalhesDoFilme detalhesDoFilme = client.getForObject(url, DetalhesDoFilme.class);
 Optional.of(detalhesDoFilme);
}
```

```

} catch(RestClientException e){
 Optional.empty();
}

```

Por fim, precisamos deixar esse comportamento isolado dentro um método:

```

public Optional<DetalhesDoFilme> request(Filme filme) {

 RestTemplate client = new RestTemplate();

 String url = // endereço de onde estamos obtendo as informações

 try {
 DetalhesDoFilme detalhesDoFilme = client.getForObject(url, DetalhesDoFilme.class);
 return Optional.of(detalhesDoFilme);
 } catch (RestClientException e) {
 return Optional.empty();
 }
}

```

Ainda é necessário deixarmos esse nosso método em alguma classe que precisa ser gerenciada pelo Spring , para isso usaremos a anotação `@Component` que faz com que a classe se torne um Bean .

```

@Component
public class ImdbClient {

 public Optional<DetalhesDoFilme> request(Filme filme) {

 RestTemplate client = new RestTemplate();

 String url = // endereço de onde estamos obtendo as informações

 try {
 DetalhesDoFilme detalhesDoFilme = client.getForObject(url, DetalhesDoFilme.class);
 return Optional.of(detalhesDoFilme);
 } catch (RestClientException e) {

 return Optional.empty();
 }
 }
}

```

Só temos um problema! Até agora, se tivermos qualquer problema nessa requisição, não saberemos o que nos levou a esse ponto. Poderíamos fazer um simples `println` da `Exception` , mas seria bem chato de conseguir o identificar no console, principalmente em projetos grandes, onde existam vários usuários e, eventualmente, algumas falhas. Nesses casos, é comum o uso de alguma biblioteca específica para *log* de sistemas. A boa notícia é que no mundo Java temos uma bastante conhecida e utilizada, chamada *Log4J*.

Vamos começar a utiliza-la em nosso sistema e, para isso, precisamos fazer a definição do objeto que ficará *logando* as coisas para nós:

```

import org.apache.log4j.Logger;

@Component
public class ImdbClient {

 private Logger logger = Logger.getLogger(ImdbClient.class);

 //restante

}

```

Nessa linha, declaramos que o `Logger` será responsável pela classe `ImdbClient`. Sendo assim, basta deixarmos nosso `Logger` fazer o trabalho dele:

```

} catch (RestClientException e){
 logger.error(e.getMessage(), e);
 return Optional.empty();
}

```

Nesse caso, ele irá gerar um log sempre que ocorrer uma `RestClientException` e, além disso, podemos o utilizar para fazer debug, passar informação e mais algumas opções.

## 4.4 EXERCÍCIO - CONSUMINDO SERVIÇO PARA DETALHES DO FILME

1. Adicione uma nova dependência no `pom.xml` para utilizarmos o `Log4J`:

```

<dependency>
 <groupId>log4j</groupId>
 <artifactId>log4j</artifactId>
 <version>1.2.17</version>
</dependency>

```

2. Crie a classe `DetalhesDoFilme` no pacote `br.com.caelum.ingresso.model` para representar os detalhes do filme retornado pela API <https://imdb-fj22.herokuapp.com/imdb?title=rogue+one>:

```

public class DetalhesDoFilme {

 @JsonProperty("Title")
 private String titulo;

 @JsonProperty("Year")
 private Integer ano;

 @JsonProperty("Poster")
 private String imagem;

 @JsonProperty("Director")
 private String diretores;

 @JsonProperty("Writer")
 private String escritores;

 @JsonProperty("Actors")
 private String atores;
}

```

```

 @JsonProperty("Plot")
 private String descricao;

 @JsonProperty("imdbRating")
 private Double avaliacao;

 // getters e setters
}

```

3. Vamos criar uma classe que irá consumir o serviço web. Crie a classe `ImdbClient` no pacote `br.com.caelum.ingresso.rest`:

```

import org.apache.log4j.Logger;

@Component
public class ImdbClient {

 private Logger logger = Logger.getLogger(ImdbClient.class);

 public Optional<DetalhesDoFilme> request(Filme filme){

 RestTemplate client = new RestTemplate();

 String titulo = filme.getNome().replace(" ", "+");

 String url = String.format("https://imdb-fj22.herokuapp.com/imdb?title=%s", titulo);

 try {
 DetalhesDoFilme detalhesDoFilme = client.getForObject(url, DetalhesDoFilme.class);
 return Optional.of(detalhesDoFilme);
 }catch (RestClientException e){
 logger.error(e.getMessage(), e);
 return Optional.empty();
 }
 }
}

```

4. Vamos alterar a action `detalhes`, na classe `FilmeController`, para consumir o serviço e disponibilizar na página as informações do filme. Para isso, precisamos de um objeto do tipo `ImdbClient` disponível nessa classe:

- Injete `ImdbClient` no `FilmeController`:

```

@Controller
public class FilmeController {

 @Autowired
 private FilmeDao filmeDao;

 @Autowired
 private SessaoDao sessaoDao;

 @Autowired
 private ImdbClient client;

 .
 .
}

```

}

- Altere a action de `detalhes` para obter as informações do filme através do atributo `client` e disponibilize-os para a página de detalhe:

```
@GetMapping("/filme/{id}/detalhe")
public ModelAndView detalhes(@PathVariable("id") Integer id){
 ModelAndView modelAndView = new ModelAndView("/filme/detalhe");

 Filme filme = filmeDao.findOne(id);
 List<Sessao> sessoes = sessaoDao.buscaSessoesDoFilme(filme);

 Optional<DetalhesDoFilme> detalhesDoFilme = client.request(filme);

 modelAndView.addObject("sessoes", sessoes);
 modelAndView.addObject("detalhes", detalhesDoFilme.orElse(new DetalhesDoFilme()));

 return modelAndView;
}
```

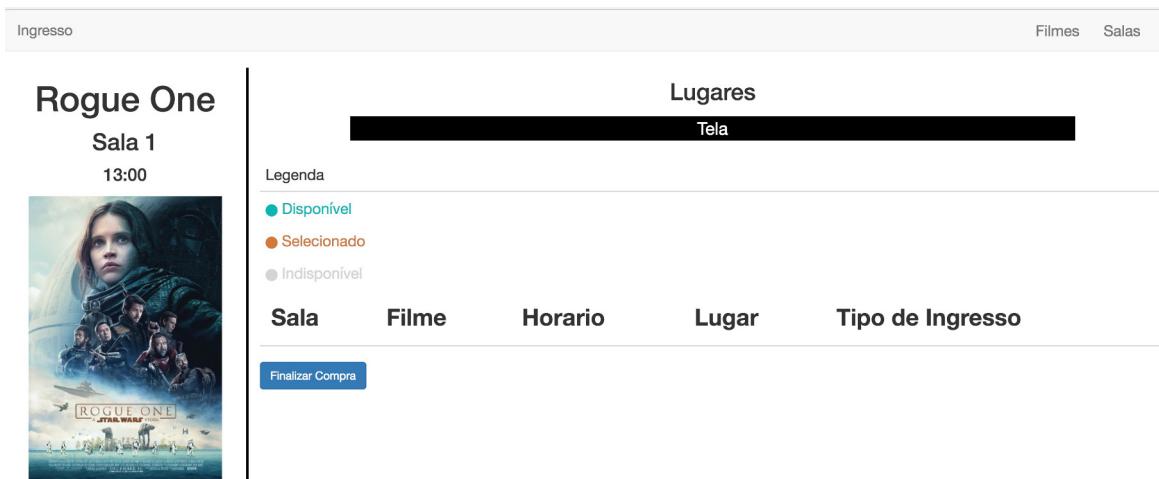
5. Reinicie a aplicação e verifique se os detalhes do filme foram obtidos corretamente através do consumo do serviço.

# INICIANDO O PROCESSO DE VENDA

## 5.1 CRIANDO TELA PARA ESCOLHA DE LUGAR

Naturalmente o usuário do nosso sistema vai desejar comprar um ingresso. Para isso, precisamos deixar disponível para ele quais são os lugares que ele pode comprar assim que ele decidir qual será a sessão que ele deseja ver.

Criaremos então uma nova tela que exibirá alguns detalhes do filme, desta forma:



Para chegarmos nesse resultado, teremos que fazer algumas *refatorações* no nosso sistema. A primeira é reaproveitar a nossa classe `ImdbClient` para continuar pegando as informações dos filmes, contudo, na segunda tela só queremos pegar o banner. Devemos então criar outra classe que vai mapear exatamente o que queremos:

```
public class ImagemCapa {
 @JsonProperty("Poster")
 String url;

 public String getUrl() {
 return url;
 }
}
```

```

 public void setUrl(String url) {
 this.url = url;
 }
}

```

Precisamos deixar a classe `ImdbClient` genérica, ou seja, fazer com que ela possa devolver tanto o `DetalhesDoFilme` quanto a nossa nova classe `ImagenCapa`, para isso temos que fazer essa sutil alteração :

```

public <T> Optional<T> request(Filme filme, Class<T> tClass){

 RestTemplate client = new RestTemplate();

 String titulo = filme.getNome().replace(" ", "+");

 String url = String.format("https://imdb-fj22.herokuapp.com/imdb?title=%s", titulo);

 try {
 return Optional.of(client.getForObject(url, tClass));
 }catch (RestClientException e){
 logger.error(e.getMessage(), e);
 return Optional.empty();
 }
}

```

Agora precisamos disponibilizar essas informações para a tela:

```

@GetMapping("/sessao/{id}/lugares")
public ModelAndView lugaresNaSessao(@PathVariable("id") Integer sessaoId){
 ModelAndView modelAndView = new ModelAndView("sessao/lugares");

 Sessao sessao = sessaoDao.findOne(sessaoId);
 Optional<ImagenCapa> imagemCapa = client.request(sessao.getFilme(), ImagenCapa.class);

 modelAndView.addObject("sessao", sessao);
 modelAndView.addObject("imagemCapa", imagemCapa.orElse(new ImagenCapa()));

 return modelAndView;
}

```

Vamos deixar os lugares disponíveis para a tela, para isso precisamos fazer uma pequena alteração na Sessão para que ela forneça para a tela os lugares:

```

public Map<String, List<Lugar>> getMapaDeLugares(){
 return sala.getMapaDeLugares();
}

```

E por fim vamos disponibilizar nossos lugares para a tela :

---

```

@GetMapping("/sessao/{id}/lugares")
public ModelAndView lugaresNaSessao(@PathVariable("id") Integer sessaoId){
 ModelAndView modelAndView = new ModelAndView("sessao/lugares");

 Sessao sessao = sessaoDao.findOne(sessaoId);

 Optional<ImagenCapa> imagemCapa = client.request(sessao.getFilme(), ImagenCapa.class);

```

```

modelAndView.addObject("sessao", sessao);
modelAndView.addObject("imagemCapa", imagemCapa.orElse(new ImagemCapa()));

return modelAndView;
}

```

## 5.2 EXERCÍCIO - CRIANDO TELA PARA SELEÇÃO DE LUGARES

1. Dentro da classe `SessaoController`, crie uma action para acessar `/sessao/{id}/lugares` :

```

@GetMapping("/sessao/{id}/lugares")
public ModelAndView lugaresNaSessao(@PathVariable("id") Integer sessaoId){
 ModelAndView modelAndView = new ModelAndView("sessao/lugares");

 return modelAndView;
}

```

2. Adicione o método para buscar a sessão por id:

```

@Repository
public class SessaoDao {

 //Demais métodos omitidos

 public Sessao findOne(Integer id) {
 return manager.find(Sessao.class, id);
 }
}

```

3. Disponibilize a sessão para a jsp :

```

@GetMapping("/sessao/{id}/lugares")
public ModelAndView lugaresNaSessao(@PathVariable("id") Integer sessaoId){
 ModelAndView modelAndView = new ModelAndView("sessao/lugares");

 Sessao sessao = sessaoDao.findOne(sessaoId);

 modelAndView.addObject("sessao", sessao);

 return modelAndView;
}

```

4. Crie a classe `ImagenCapa` no pacote de `br.com.caelum.ingresso.modelo`. Usaremos ela para pegar somente a imagem de capa do filme retornado pela api <https://imdb-fj22.herokuapp.com/imdb?title=NOME+DO+FILME> :

```

public class ImagemCapa {

 @JsonProperty("Poster")
 String url;

 public String getUrl() {
 return url;
 }

 public void setUrl(String url) {
 this.url = url;
 }
}

```

5. Altere a classe `ImdbClient` no pacote `br.com.caelum.ingresso.rest` para que ela seja genérica, ou seja indiferente da representação ( `DetalheDoFilme` ou `ImagenCapa` ), dessa forma será possível consumir a api e retornar de forma correta:

```
@Component
public class ImdbClient {

 private Logger logger = Logger.getLogger(ImdbClient.class);

 public <T> Optional<T> request(Filme filme, Class<T> tClass){

 RestTemplate client = new RestTemplate();

 String titulo = filme.getNome().replace(" ", "+");

 String url = String.format("https://imdb-fj22.herokuapp.com/imdb?title=%s", titulo);

 try {
 return Optional.of(client.getForObject(url, tClass));
 }catch (RestClientException e){
 logger.error(e.getMessage(), e);
 return Optional.empty();
 }
 }
}
```

6. Com essa alteração, quebramos o método `detalhes` na classe `FilmeController` . Vamos corrigi-lo informando qual a classe será utilizada no método `request` :

```
@GetMapping("/filme/{id}/detalhe")
public ModelAndView detalhes(@PathVariable("id") Integer id){
 ModelAndView modelAndView = new ModelAndView("/filme/detalhe");

 Filme filme = filmeDao.findOne(id);
 List<Sessao> sessoes = sessaoDao.buscaSessoesDoFilme(filme);

 Optional<DetalhesDoFilme> detalhesDoFilme = client.request(filme, DetalhesDoFilme.class);

 modelAndView.addObject("sessoes", sessoes);
 modelAndView.addObject("detalhes", detalhesDoFilme.orElse(new DetalhesDoFilme()));

 return modelAndView;
}
```

7. Vamos alterar o método `lugaresNaSessao` da classe `SessaoController` para disponibilizar a imagem de capa para nossa jsp . Não se esqueça de *injetar* o `ImdbClient` :

```
@GetMapping("/sessao/{id}/lugares")
public ModelAndView lugaresNaSessao(@PathVariable("id") Integer sessaoId){
 ModelAndView modelAndView = new ModelAndView("sessao/lugares");

 Sessao sessao = sessaoDao.findOne(sessaoId);
 Optional<ImagenCapa> imagemCapa = client.request(sessao.getFilme(), ImagemCapa.class);

 modelAndView.addObject("sessao", sessao);
 modelAndView.addObject("imagemCapa", imagemCapa.orElse(new ImagemCapa()));

 return modelAndView;
```

```
}
```

8. Crie o método `getMapaDeLugares` na classe `Sessao`, para podermos disponibilizar os lugares para tela :

```
public Map<String, List<Lugar>> getMapaDeLugares(){
 return sala.getMapaDeLugares();
}
```

9. Acesse a página de detalhes de algum filme e clique no botão para comprar alguma sessão disponível. A tela de seleção de lugares deverá ser exibida corretamente.

## 5.3 SELECIONANDO INGRESSO

Agora precisamos deixar a opção de nosso usuário poder de fato comprar o seu ingresso, contudo ainda precisamos definir qual é o tipo de `Ingresso` que ele está querendo comprar, com seu respectivo desconto.

Para sabermos disso vamos deixar mais evidente para o usuário, primeiro vamos criar um `Enum` que tenha todos esses tipo :

```
public enum TipoDeIngresso {

 INTEIRO(new SemDesconto()),
 ESTUDANTE(new DescontoEstudante()),
 BANCO(new DescontoDeTrintaPorCentoSobreBancos());

 private final Desconto desconto;

 TipoDeIngresso(Desconto desconto) {
 this.desconto = desconto;
 }

 public BigDecimal aplicaDesconto(BigDecimal valor){
 return desconto.aplicarDescontoSobre(valor);
 }

 public String getDescricao(){
 return desconto.getDescricao();
 }
}
```

Precisamos agora definir na nossa interface `Desconto` o método `getDescricao()` para que todas as classes que implementem sejam obrigadas a ter este método :

```
public interface Desconto {

 BigDecimal aplicarDescontoSobre(BigDecimal precoOriginal);
 String getDescricao();
}
```

Desta forma, precisamos fazer uma pequena alteração na nossa classe `Ingresso`, para que tenha o `TipoDeIngresso` invés do `Desconto` :

```

public class Ingresso {

 private Sessao sessao;
 private BigDecimal preco;

 public Ingresso(Sessao sessao, TipoDeIngresso tipoDeDesconto) {
 this.sessao = sessao;
 this.preco = tipoDeDesconto.aplicarDescontoSobre(sessao.getPreco());
 }

 // getters
}

```

Além disso nosso usuário vai precisar ter no seu ingresso, qual foi o Lugar que ele comprou, para isso vamos adicionar esse atributo ao no `Ingresso` :

```

public class Ingresso {

 private Sessao sessao;
 private BigDecimal preco;
 private Lugar lugar;

 public Ingresso(Sessao sessao, TipoDeIngresso tipoDeDesconto, Lugar lugar) {
 this.sessao = sessao;
 this.preco = tipoDeDesconto.aplicarDescontoSobre(sessao.getPreco());
 this.lugar = lugar;
 }

 // getters
}

```

## 5.4 EXERCÍCIO - IMPLEMENTANDO A SELEÇÃO DE LUGARES, INGRESSOS E TIPO DE INGRESSOS.

- Nosso ingresso além de ter um preço e uma sessão deve ter um lugar. Altere a classe `Ingresso`, adicione um atributo para o `Lugar` e receba-o no construtor:

```

public class Ingresso {

 private Sessao sessao;
 private Lugar lugar;
 private BigDecimal preco;

 public Ingresso(Sessao sessao, Desconto tipoDeDesconto, Lugar lugar) {
 this.sessao = sessao;
 this.preco = tipoDeDesconto.aplicarDescontoSobre(sessao.getPreco());
 this.lugar = lugar;
 }
}

```

```
// getters
```

2. Adicione na interface Desconto um método para pegar a descrição do desconto:

```
public interface Desconto {

 BigDecimal aplicarDescontoSobre(BigDecimal precoOriginal);
 String getDescricao();
}
```

3. Implemente o método getDescricao em todos os descontos:

```
public class SemDesconto implements Desconto {

 @Override
 public String getDescricao() {
 return "Normal";
 }
}

public class DescontoEstudante implements Desconto {

 @Override
 public String getDescricao() {
 return "Estudante";
 }
}

public class DescontoDeTrintaPorCentoSobreBancos implements Desconto {

 @Override
 public String getDescricao() {
 return "Desconto Banco";
 }
}
```

4. Ao invés de receber um desconto nosso ingresso deve receber um TipoDeIngresso , crie esse enum no pacote br.com.caelum.ingresso.model :

```
public enum TipoDeIngresso {

 INTEIRO(new SemDesconto()),
 ESTUDANTE(new DescontoEstudante()),
 BANCO(new DescontoDeTrintaPorCentoSobreBancos());

 private final Desconto desconto;

 TipoDeIngresso(Desconto desconto) {
 this.desconto = desconto;
 }

 public BigDecimal aplicaDesconto(BigDecimal valor){
 return desconto.aplicarDescontoSobre(valor);
 }

 public String getDescricao(){
 return desconto.getDescricao();
 }
}
```

5. Crie um atributo `TipoDeIngresso` na classe `Ingresso` e altere o ingresso para receber um `TipoDeIngresso` ao invés do desconto.

```
public class Ingresso {

 private Sessao sessao;

 private Lugar lugar;

 private BigDecimal preco;

 private TipoDeIngresso tipoDeIngresso;

 public Ingresso(Sessao sessao, TipoDeIngresso tipoDeIngresso, Lugar lugar) {
 this.sessao = sessao;
 this.tipoDeIngresso = tipoDeIngresso;
 this.preco = this.tipoDeIngresso.aplicaDesconto(sessao.getPreco());

 this.lugar = lugar;
 }

 //demais métodos
}
```

6. Como vamos persistir um ingresso no banco de dados para simular uma compra do usuário, adicione os mapeamentos do Hibernate nos atributos da classe. Inclua, também, um atributo do tipo `Integer` para representar o id do `Ingresso`:

```
@Entity
public class Ingresso {

 @Id
 @GeneratedValue
 private Integer id;

 @ManyToOne
 private Sessao sessao;

 @ManyToOne
 private Lugar lugar;

 private BigDecimal preco;

 @Enumerated(EnumType.STRING)
 private TipoDeIngresso tipoDeIngresso;

 public Ingresso(Sessao sessao, TipoDeIngresso tipoDeIngresso, Lugar lugar) {
 this.sessao = sessao;
 this.tipoDeIngresso = tipoDeIngresso;
 this.preco = this.tipoDeIngresso.aplicaDesconto(sessao.getPreco());
 this.lugar = lugar;
 }

 //demais métodos
}
```

7. Após a última alteração os testes da classe `DescontoTeste` quebraram por conta do construtor.

Corrija os testes fazendo-os receber um `TipoDeIngresso` e um `Lugar` no construtor do ingresso:

```
public class DescontoTest {

 @Test
 public void naoDeveConcederDescontoParaIngressoNormal(){
 Lugar lugar = new Lugar("A",1);
 Sala sala = new Sala("Eldorado - IMAX", new BigDecimal("20.5"));
 Filme filme = new Filme("Rogue One", Duration.ofMinutes(120),
 "SCI-FI", new BigDecimal("12"));
 Sessao sessao = new Sessao(LocalTime.parse("10:00:00"), filme, sala);
 Ingresso ingresso = new Ingresso(sessao, TipoDeIngresso.INTEIRO,lugar);

 BigDecimal precoEsperado = new BigDecimal("32.5");

 Assert.assertEquals(precoEsperado, ingresso.getPreco());
 }
}
```

8. Rode os testes e garanta que todos estão funcionando corretamente.
9. Precisamos saber quais lugares já foram ocupados em uma determinada sessão. Vamos adicionar um atributo do tipo `Set<Ingresso>` chamado `ingressos` na classe `Sessao`, para obtermos todos os ingressos que já foram vendidos para uma determinada sessão e seus respectivos lugares. Adicione no novo atributo a anotação `@OneToMany` para mapeá-lo para o Hibernate, bem como as propriedades da anotação `mappedBy="sessao"` e `fetch=FetchType.EAGER`, para dizer que a relação já foi mapeada pelo atributo `sessao` da classe `Ingresso` e para o Hibernate já trazer os ingressos do banco de dados quando buscarmos por uma `Sessao`, respectivamente.

```
@Entity
public class Sessao{
 //Demais atributos omitidos

 @OneToMany(mappedBy = "sessao", fetch = FetchType.EAGER)
 private Set<Ingresso> ingressos = new HashSet<>();

 //Demais métodos omitidos
}
```

10. Crie um método `isDisponivel` na classe `Sessao`, que deve verificar se o lugar está ou não disponível:

```
public boolean isDisponivel(Lugar lugarSelecionado) {
 return ingressos.stream().map(Ingresso::getLugar).noneMatch(lugar -> lugar.equals(lugarSelecionado));
}
```

11. Altere o método `lugaresNaSessao` na `SessaoController` e disponibilize para a `.jsp` a lista de tipos de ingressos, para seleção de tipos de ingressos:

```
@Controller
public class SessaoController {

 //demais métodos
```

```

@GetMapping("/sessao/{id}/lugares")
public ModelAndView lugaresNaSessao(@PathVariable("id") Integer sessaoId){
 ModelAndView modelAndView = new ModelAndView("sessao/lugares");

 Sessao sessao = sessaoDao.findOne(sessaoId);

 Optional<ImagemCapa> imagemCapa = client.request(sessao.getFilme(), ImagemCapa.class);

 modelAndView.addObject("sessao", sessao);
 modelAndView.addObject("imagemCapa", imagemCapa.orElse(new ImagemCapa()));
 modelAndView.addObject("tiposDeIngressos", TipoDeIngresso.values());

 return modelAndView;
}
}

```

## 5.5 EXERCÍCIOS OPCIONAIS - TESTANDO A VERIFICAÇÃO DE LUGARES DISPONÍVEIS

1. Vamos criar um teste na classe `SessaoTest` que garanta que nossa implementação do método `isDisponivel` está correto:

```

@Test
public void garanteQueOLugarA1EstaOcupadoE0sLugaresA2EA3Disponiveis(){
 Lugar a1 = new Lugar("A", 1);
 Lugar a2 = new Lugar("A", 2);
 Lugar a3 = new Lugar("A", 3);

 Filme rogueOne = new Filme("Rogue One", Duration.ofMinutes(120),
 "SCI_FI", new BigDecimal("12.0"));

 Sala eldorado7 = new Sala("Eldorado 7", new BigDecimal("8.5"));

 Sessao sessao = new Sessao(LocalTime.parse("10:00:00"), rogueOne, eldorado7);

 Ingresso ingresso = new Ingresso(sessao, TipoDeIngresso.INTEIRO, a1);

 Set<Ingresso> ingressos = Stream.of(ingresso).collect(Collectors.toSet());

 sessao.setIngressos(ingressos);

 Assert.assertFalse(sessao.isDisponivel(a1));
 Assert.assertTrue(sessao.isDisponivel(a2));
 Assert.assertTrue(sessao.isDisponivel(a3));
}

```

# CRIANDO O CARRINHO

## 6.1 O CARRINHO DE COMPRAS

Algo bem comum em sistemas de vendas é podermos ter um carrinho de compras para o usuário. Nossa sistema não será diferente! Por isso, vamos criar nossa classe `Carrinho` :

```
public class Carrinho {

}
```

Agora, precisamos fazer com que o `Spring` consiga ter acesso a ele e, portanto, usaremos a anotação `@Component`. Contudo, o carrinho de compras geralmente fica vivo durante toda a sessão do usuário. Desse modo, vamos deixar ele com o escopo de sessão:

```
@Component
@SessionScoped
public class Carrinho {

}
```

Para deixarmos nosso carrinho mais inteligente, vamos dar algumas propriedades a ele! Por exemplo, precisaremos ter a lista de ingressos que o usuário vai querer comprar e também uma forma de podermos adicionar os ingressos nessa lista:

```
public class Carrinho {

 private List<Ingresso> ingressos = new ArrayList<>();

 public void add(Ingresso ingresso){
 ingressos.add(ingresso);
 }

 //demais métodos e getters e setters
}
```

Agora, é necessário disponibilizar o carrinho para a tela, ou seja, quando pegarmos a sessão precisamos injetar nosso carrinho:

```
@Controller
public class SessaoController {

 @Autowired
```

```

 private Carrinho carrinho;
}

}

```

Como na nossa requisição estamos enviando somente os *id's* da sessão, precisamos pegar esses dados do banco e retornar um ingresso válido para adicionar ao carrinho. Para isso, vamos criar a classe `CarrinhoForm` que representará nosso formulário. Essa classe será responsável por pegar os dados selecionados pelo usuário na tela e, a partir deles, nos fornecer um objeto `Carrinho` para podermos finalizar a compra.

```

public class CarrinhoForm {

 private List<Ingresso> ingressos = new ArrayList<>();

 public List<Ingresso> getIngressos() {
 return ingressos;
 }

 public void setIngressos(List<Ingresso> ingressos) {
 this.ingressos = ingressos;
 }

 public List<Ingresso> toIngressos(SessaoDao sessaoDao, LugarDao lugarDao){

 return this.ingressos.stream().map(ingresso -> {
 Sessao sessao = sessaoDao.findOne(ingresso.getSessao().getId());
 Lugar lugar = lugarDao.findOne(ingresso.getLugar().getId());
 TipoDeIngresso tipoDeIngresso = ingresso.getTipoDeIngresso();
 return new Ingresso(sessao, tipoDeIngresso, lugar);
 }).collect(Collectors.toList());
 }
}

```

E, por fim, precisamos assim que o usuário decidir todos os ingressos que deseja, adicioná-los no nosso carrinho:

```

@PostMapping("/compra/ingressos")
public ModelAndView enviarParaPagamento(CarrinhoForm carrinhoForm){
 ModelAndView modelAndView = new ModelAndView("redirect:/compra");

 formulario.toIngressos(sessaoDao, lugarDao).forEach(carrinho::add);

 return modelAndView;
}

```

## 6.2 EXERCÍCIO - IMPLEMENTANDO A TELA DE COMPRAS

1. Ao selecionar o lugar e o tipo de ingresso, vamos adicionar os ingressos no carrinho. Portanto, vamos criar a classe `Carrinho` no pacote de modelos, `br.com.caelum.ingresso.model`.

---

```

public class Carrinho {

 private List<Ingresso> ingressos = new ArrayList<>();

 public void add(Ingresso ingresso){

```

```

 ingressos.add(ingresso);
 }

 //demais métodos e getters e setters
}

```

2. Altere o escopo do carrinho para `SessionScope`. Fique atento para pegar o import do Spring, do pacote `org.springframework.web.context.annotation`:

```

@Component
@SessionScope
public class Carrinho {

 //restante da implementação
}

```

3. Vamos disponibilizar o carrinho para a tela de seleção de lugares:

```

@Controller
public class SessaoController {
 //Demais atributos omitidos

 @Autowired
 private Carrinho carrinho;

 @GetMapping("/sessao/{id}/lugares")
 public ModelAndView lugaresNaSessao(@PathVariable("id") Integer sessaoId){
 ModelAndView modelAndView = new ModelAndView("sessao/lugares");

 Sessao sessao = sessaoDao.findOne(sessaoId);

 Optional<ImagemCapa> imagemCapa = client.request(sessao.getFilme(), ImagemCapa.class);

 modelAndView.addObject("sessao", sessao);
 modelAndView.addObject("carrinho", carrinho);
 modelAndView.addObject("imagemCapa", imagemCapa.orElse(new ImagemCapa()));
 modelAndView.addObject("tiposDeIngressos", TipoDeIngresso.values());

 return modelAndView;
 }

 //Demais métodos omitidos
}

```

4. Como na nossa requisição estamos enviando somente os *id's* da sessão, precisamos pegar esses dados do banco e retornar um ingresso válido para adicionar ao carrinho. Desse modo, no pacote `br.com.caelum.ingresso.model.form`, vamos criar a classe `CarrinhoForm` que representará nosso formulário. Nela, crie um método que irá retornar uma lista de ingressos válidos:

```

public class CarrinhoForm {

 private List<Ingresso> ingressos = new ArrayList<>();

 public List<Ingresso> getIngressos() {
 return ingressos;
 }

 public void setIngressos(List<Ingresso> ingressos) {
 this.ingressos = ingressos;
 }
}

```

```

 }

 public List<Ingresso> toIngressos(SessaoDao sessaoDao, LugarDao lugarDao){

 return this.ingressos.stream().map(ingresso -> {
 Sessao sessao = sessaoDao.findOne(ingresso.getSessao().getId());
 Lugar lugar = lugarDao.findOne(ingresso.getLugar().getId());
 TipoDeIngresso tipoDeIngresso = ingresso.getTipoDeIngresso();
 return new Ingresso(sessao, tipoDeIngresso, lugar);
 }).collect(Collectors.toList());
 }
}

```

5. Adicione o método `findOne` na classe `LugarDao`, localizada no pacote `br.com.caelum.ingresso.dao`:

```

public Lugar findOne(Integer id) {
 return manager.find(Lugar.class, id);
}

```

6. Vamos criar a classe `CompraController` no pacote `br.com.caelum.ingresso.controller`. Depois, na classe criada, adicione a *action* para a *URI* `/compra/ingressos`:

```

@Controller
public class CompraController {

 @Autowired
 private SessaoDao sessaoDao;
 @Autowired
 private LugarDao lugarDao;

 @Autowired
 private Carrinho carrinho;

 @PostMapping("/compra/ingressos")
 public ModelAndView enviarParaPagamento(CarrinhoForm carrinhoForm){
 ModelAndView modelAndView = new ModelAndView("redirect:/compra");

 carrinhoForm.toIngressos(sessaoDao, lugarDao).forEach(carrinho::add);

 return modelAndView;
 }
}

```

7. Para realizar a escolha de lugares na tela, vamos alterar o comportamento da tag `svg` adicionando a propriedade `onclick`. Quando o lugar for clicado, uma verificação deverá ser realizada para checar se o lugar já foi selecionado e alterar seu estado para *Ocupado* ou *Disponível*.

```

<svg class="assento ${sessao.isDisponivel(lugar) ? "disponivel" : "ocupado"}" onclick="${sessao.isDisponivel(lugar) ? 'changeCheckbox(this)' : '' }" ...
>

```

## 6.3 MELHORANDO A USABILIDADE DO CARRINHO

Ainda precisamos remover do carrinho os lugares que foram marcados e logo em seguida desmarcados, para isso vamos criar um método que faça a validação se já está dentro do carrinho.

Usaremos o método `anyMatch` da classe `Stream` para fazer a verificação de algum ingresso já presente no carrinho :

```
public boolean isSelecionado(Lugar lugar){
 return ingressos.stream().map(Ingresso::getLugar).anyMatch(l -> l.equals(lugar));
}
```

Agora, basta usar esse nosso método em nossa `.jsp`, no próprio item que representa o lugar, para que a verificação seja feita antes de adicionar um ingresso e não tenhamos a seleção de lugares duplicada :

```
<svg class="assento ${sessao.isDisponivel(lugar) && !carrinho.isSelecionado(lugar) ? "disponivel" : "ocupado" }"
 onclick="${sessao.isDisponivel(lugar) && !carrinho.isSelecionado(lugar) ? 'changeCheckbox(this)' : '' }"
 ...
 <!-- Restante -->
</svg>
```

## 6.4 EXERCÍCIO - DESABILITANDO A SELEÇÃO DO LUGAR QUE JÁ ESTÁ NO CARRINHO

1. Vamos adicionar um método no carrinho para verificar se um lugar está no carrinho ou não:

```
public boolean isSelecionado(Lugar lugar){
 return ingressos.stream().map(Ingresso::getLugar).anyMatch(l -> l.equals(lugar));
}
```

2. Vamos alterar a tela de seleção de lugares/lista, para que não seja possível selecionar os lugares que já foram adicionados no carrinho. Para isso, altere a tag `svg` já presente na nossa `.jsp` e adicione a nova verificação do carrinho nas propriedades `class` e `onclick` :

```
<svg class="assento ${sessao.isDisponivel(lugar) && !carrinho.isSelecionado(lugar) ? "disponivel" : "ocupado" }"
 onclick="${sessao.isDisponivel(lugar) && !carrinho.isSelecionado(lugar) ? 'changeCheckbox(this)' : '' }"
 ...
 <!-- Restante -->
</svg>
```

## 6.5 EXIBINDO OS DETALHES DA COMPRA

Antes de fechamos a nossa compra, nosso usuário quer checar se as informações estão corretas para poder fazer o pagamento.

Sala	Lugar	Filme	Horário	Tipo do Ingresso	Preço
Sala 3	A1	Zootopia	18:00	Normal	45.00
<b>TOTAL 45.00</b>					

Nome:   
 SobreNome:   
 CPF:   
 Cartão de Crédito:  CVV:

**Comprar**

Para fazermos essa nossa nova tela, precisaremos ter um método em nosso `Controller` responsável por nos informar o estado da `Compra` :

```
@GetMapping("/compra")
public ModelAndView checkout(){

 ModelAndView modelAndView = new ModelAndView("compra/pagamento");

 modelAndView.addObject("carrinho", carrinho);
 return modelAndView;
}
```

Precisaremos ter no nosso carrinho um método para poder disponibilizar o valor total para que ele seja exibido na tela. Primeiramente obteremos todos os valores dos ingressos do carrinho usando um objeto `Stream` proveniente da lista de ingressos. Então, através do método `map` poderemos invocar o `getPreco` de cada um dos ingressos, obtendo assim um `Stream<BigDecimal>`. Usaremos, novamente, um método da classe `Stream` para obtermos a somatória de todos os valores contidos no mesmo :

```
public BigDecimal getTotal(){
 return ingressos.stream().map(Ingresso::getPreco).reduce(BigDecimal::add).orElse(BigDecimal.ZERO)
;
}
```

Estamos pegando todos os valores, somando e então devolvendo apenas o resultado, caso não tenha nenhum valor para ser somado, estamos devolvendo um `BigDecimal` de valor zero.

## 6.6 EXERCÍCIO - IMPLEMENTANDO A TELA DE CHECKOUT

- Na classe `CompraController` adicione uma action para `/compra` e disponibilize o carrinho para a jsp :

```
@GetMapping("/compra")
public ModelAndView checkout(){

 ModelAndView modelAndView = new ModelAndView("compra/pagamento");

 modelAndView.addObject("carrinho", carrinho);
```

```
 return modelAndView;
 }
```

2. Altere a classe carrinho e adicione um método que retorne o total a pagar:

```
public BigDecimal getTotal(){
 return ingressos.stream().map(Ingresso::getPreco).reduce(BigDecimal::add).orElse(BigDecimal.ZERO);
}
```

3. Reinicie o sistema, selecione um ingresso para alguma sessão, precione o botão para finalizar a compra e visualize a exibição da tela de detalhes da compra.

## 6.7 REALIZANDO A COMPRA

Ainda é necessário podermos finalizar a compra. para isso precisaremos adicionar o cartão como forma pagamento. Criaremos uma classe para representa-lo :

```
public class Cartao {

 private String numero;
 private Integer cvv;
 private YearMonth vencimento;

 //getters e setters
}
```

Nosso controller precisa saber que assim que ele entra na página de pagamento ele precisa ter um cartão, para isso faremos com que ele nos forneça um cartão :

```
@GetMapping("/compra")
public ModelAndView checkout(Cartao cartao){
 //restante do código
}
```

Como nosso data de vencimento do cartão é representada pelo objeto `YearMonth` e na nossa tela inserimos apenas com `String`, precisamos informar como deve ser feita a conversão. Para isso será necessário criarmos nosso próprio conversor, o que pode ser feito implementando a interface `Converter` :

```
public class YearMonthConverter implements Converter<String, YearMonth> {

 @Override
 public YearMonth convert(String text) {
 return YearMonth.parse(text, DateTimeFormatter.ofPattern("MM/yyyy"));
 }
}
```

Com nosso conversor criado, precisamos avisar ao Spring que ele existe, para isso teremos que fazer algumas mudanças no nosso `spring-context.xml` :

```
<mvc:annotation-driven conversion-service="conversionService"/>
```

```

<bean id="conversionService" class="org.springframework.context.support.ConversionServiceFactoryBean">
 <property name="converters">
 <set merge="true">
 <bean class="br.com.caelum.ingresso.converter.YearMonthConverter"/>
 </set>
 </property>
</bean>

```

Para saber se o cartão informado pelo usuário é válido, podemos criar um pequeno método de validação no mesmo, que checará se a data de expiração já está vencida ou não:

```

public boolean valido(){
 return vencimento.isAfter(YearMonth.now());
}

```

Criaremos um objeto que represente a compra dos ingressos em si, para que possamos persistí-la:

```

@Entity
public class Compra {

 @Id
 @GeneratedValue
 private Long id;

 @OneToMany(cascade = CascadeType.PERSIST)
 List<Ingresso> ingressos = new ArrayList<>();

 /**
 * @deprecated hibernate only
 */
 public Compra() {
 }

 public Compra(List<Ingresso> ingressos) {
 this.ingressos = ingressos;
 }

 // getters e setters
}

```

Como a compra é realizada através dos métodos do próprio `Carrinho`, podemos fazer com que o mesmo já nos forneça um objeto do tipo `Compra` baseado nos ingressos já inseridos no carrinho pelo cliente:

```

public class Carrinho{

 // Atributos e demais métodos omitidos

 public Compra toCompra(){
 return new Compra(ingressos);
 }
}

```

Por fim, basta criarmos um novo método no nosso `CompraController` que irá realizar a persistência da compra do usuário quando o mesmo finalizar a compra na tela:

```

@PostMapping("/compra/comprar")
@Transactional

```

```

public ModelAndView comprar(@Valid Cartao cartao, BindingResult result){
 ModelAndView modelAndView = new ModelAndView("redirect:/");

 if (cartao.isValido()){
 compraDao.save(carrinho.toCompra());
 }else{
 result.rejectValue("vencimento", "Vencimento inválido");
 return checkout(cartao);
 }

 return modelAndView;
}

```

## 6.8 EXERCÍCIOS - IMPLEMENTANDO A COMPRA

- Crie a classe `Cartao` no pacote `br.com.caelum.ingresso.model`:

```

public class Cartao {

 private String numero;
 private Integer cvv;
 private YearMonth vencimento;

 //getters e setters
}

```

- Faça com que o método `checkout` da classe `CompraController` receba um objeto `Cartao`:

```

@GetMapping("/compra")
public ModelAndView checkout(Cartao cartao){

 ModelAndView modelAndView = new ModelAndView("compra/pagamento");

 modelAndView.addObject("carrinho", carrinho);
 return modelAndView;
}

```

- Crie o conversor para `YearMonth` no pacote `br.com.caelum.ingresso.converter`:

```

public class YearMonthConverter implements Converter<String, YearMonth> {

 @Override
 public YearMonth convert(String text) {
 return YearMonth.parse(text, DateTimeFormatter.ofPattern("MM/yyyy"));
 }
}

```

- Altere a declaração da tag `annotation-driven` no arquivo `spring-context.xml`:

```
<mvc:annotation-driven conversion-service="conversionService"/>
```

- Registre o conversor no `spring-context.xml`:

```

<bean id="conversionService" class="org.springframework.context.support.ConversionServiceFactoryBean">
 <property name="converters">
 <set merge="true">
 <bean class="br.com.caelum.ingresso.converter.YearMonthConverter"/>
 </set>
 </property>

```

---

```
</bean>
```

6. Crie a classe `Compra` no pacote `br.com.caelum.ingresso.model`:

```
@Entity
public class Compra {

 @Id
 @GeneratedValue
 private Long id;

 @OneToMany(cascade = CascadeType.PERSIST)
 List<Ingresso> ingressos = new ArrayList<>();

 /**
 * @deprecated hibernate only
 */
 public Compra() {
 }

 public Compra(List<Ingresso> ingressos) {
 this.ingressos = ingressos;
 }

 public Long getId() {
 return id;
 }

 public void setId(Long id) {
 this.id = id;
 }

 public List<Ingresso> getIngressos() {
 return ingressos;
 }

 public void setIngressos(List<Ingresso> ingressos) {
 this.ingressos = ingressos;
 }
}
```

7. Adicione um método no carrinho para criar uma compra:

```
public Compra toCompra(){
 return new Compra(ingressos);
}
```

8. Adicione um método no cartão que retorne se o cartão é válido ou não:

```
public boolean isValido(){
 return vencimento.isAfter(YearMonth.now());
}
```

9. Adicione uma action em `CompraController` que deverá atender a `URI /compra/comprar` e nele implemente a persistencia da compra:

```
@PostMapping("/compra/comprar")
@Transactional
public ModelAndView comprar(@Valid Cartao cartao, BindingResult result){
 ModelAndView modelAndView = new ModelAndView("redirect:/");
```

```

 if (cartao.isValido()){
 compraDao.save(carrinho.toCompra());
 }else{
 result.rejectValue("vencimento", "Vencimento inválido");
 return checkout(cartao);
 }

 return modelAndView;
}

```

10. Injete um objeto CompraDao em CompraController :

```

@Controller
public class CompraController {

 @Autowired
 private SessaoDao sessaoDao;
 @Autowired
 private LugarDao lugarDao;

 @Autowired
 private CompraDao compraDao;

 @Autowired
 private Carrinho carrinho;

 //demais métodos
}

```

11. Como ainda não criamos a classe CompraDao , crie-a no pacote br.com.caelum.ingresso.dao para corrigir os erros existentes:

```

@Repository
public class CompraDao {

 @PersistenceContext
 private EntityManager manager;

 public void save(Compra compra) {
 manager.persist(compra);
 }
}

```

12. Adicione na página pagamento.jsp o seguinte div antes do botão de submit :

```

<div class="form-group">
 <div class="col-md-6">
 <label for="vencimento">Vencimento:</label>
 <input id="vencimento" type="text" name="vencimento" class="form-control">
 </div>
</div>

```

13. Rode a aplicação e verifique se a compra está sendo persistida.

# APÊNDICE: IMPLEMENTANDO SEGURANÇA

**APÊNDICE NÃO FAZ PARTE DO CURSO.** Os apêndices são conteúdos adicionais que não fazem parte da carga horária regular do curso. São conteúdos extras para direcionar seus estudos após o curso.

## 7.1 PROTEGENDO NOSSAS URIS

Agora que concluímos o desenvolvimento das funcionalidades principais da compra de ingressos, precisamos proteger para que somente pessoas autorizadas possam cadastrar *Filmes*, *Salas*, *Sessões*. Além de exigir que um comprador autentique-se para efetuar uma compra. Para isso precisamos identificar quais *URIs* devemos proteger e quais devemos deixar acessíveis.

Por exemplo, sabemos que a *URI* `/filme/em-cartaz` deve ser acessível por todos, já a *URI* `/filme` só deve ser acessível por pessoas autorizadas. Perceba que dessa maneira teremos que criar regras para cada *URI* individualmente. O que pode ser um problema de acordo com a quantidade de *URIs* que teremos no nosso sistema.

Para facilitar a criação dessas regras, podemos agrupar nossas *URIs* de uma forma que possamos aplicar as regras para cada grupo. Por exemplo, todas as *URIs* de cadastro que só devem ser efetuadas por administradores, podemos pré fixá-las com `/admin`. Nesse caso nossa *URI* `/filme` que deve ser protegida ficará como `/admin/filme`. Devemos repetir esse processo para as demais *URIs* de cadastro.

## 7.2 CONFIGURANDO SPRING SECURITY

Agora que temos nossas *URIs* agrupadas, podemos implementar a lógica para proteger nossa aplicação. Para isso teremos que a cada *request* verificar qual a *URI* que está sendo acessada, se for uma *URI* protegida devemos verificar se o usuário está logado e se o mesmo tem as permissões necessárias. Em caso negativo, devemos redirecioná-lo para uma tela de login.

Poderíamos implementar isso utilizando um *Filtro* da *spec* de *Servlet*. Porém lidar com todas as regras, navegações e segurança dos dados não é algo tão trivial.

Para facilitar essa tarefa, existem *Frameworks* responsáveis somente por segurança em uma aplicação WEB. Dentre eles dois que se destacam bastante são *KeyCloak* e *Spring Security*. Como estamos usando *Spring* em nossa aplicação, vamos implementar a segurança através do *Spring Security*.

Vamos começar configurando as regras de acesso baseado nas *URIs*. Para isso vamos criar uma classe que herde de `WebSecurityConfigurerAdapter`, e sobrescrever o método `protected void configure(HttpSecurity http) throws Exception`:

```
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
 @Override
 protected void configure(HttpSecurity http) throws Exception {
 .
 .
 .
 }
}
```

Dentro do método `configure(HttpSecurity http)` vamos usar o objeto `HttpSecurity` para declarar nossas regras. A classe `HttpSecurity` tem uma interface fluente, onde podemos encadear métodos.

Vamos começar protegendo as *URIs* que comecem com `/admin/`. Elas por sua vez só podem ser acessadas por usuário com o perfil de `ADMIN`.

```
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
 @Override
 protected void configure(HttpSecurity http) throws Exception {
 http
 .authorizeRequests()
 .antMatchers("/admin/**").hasRole("ADMIN");
 }
}
```

Agora queremos proteger *URIs* que comecem com `/compra/`, essas por sua vez só podem ser acessíveis por usuários com perfil de comprador.

```
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
 @Override
 protected void configure(HttpSecurity http) throws Exception {
 http
 .authorizeRequests()
 .antMatchers("/admin/**").hasRole("ADMIN")
 .antMatchers("/compra/**").hasRole("COMPRADOR");
 }
}
```

Já *URIs* começadas com `/filme` podem ser acessadas por qualquer um.

```
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
 @Override
 protected void configure(HttpSecurity http) throws Exception {
 http
 .authorizeRequests()
 .antMatchers("/admin/**").hasRole("ADMIN")
 .antMatchers("/compra/**").hasRole("COMPRADOR")
 .antMatchers("/filme/**").permitAll();
 }
}
```

```
 }
}
```

Da mesma forma a *URI* `/sessao/{id}/lugares` e `/` também pode ser acessada por qualquer um.

```
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
 @Override
 protected void configure(HttpSecurity http) throws Exception {
 http
 .authorizeRequests()
 .antMatchers("/admin/**").hasRole("ADMIN")
 .antMatchers("/compra/**").hasRole("COMPRADOR")
 .antMatchers("/filme/**").permitAll()
 .antMatchers("/sessao/**/lugares").permitAll()
 .antMatchers("//").permitAll();
 }
}
```

Agora precisamos definir qualquer *request* que tenha um perfil associado ou não mapeados devem ser autenticadas (no nosso caso são as *URIs* `/admin` e `/compra`).

```
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
 @Override
 protected void configure(HttpSecurity http) throws Exception {
 http
 .authorizeRequests()
 .antMatchers("/admin/**").hasRole("ADMIN")
 .antMatchers("/compra/**").hasRole("COMPRADOR")
 .antMatchers("/filme/**").permitAll()
 .antMatchers("/sessao/**/lugares").permitAll()
 .antMatchers("//").permitAll()
 .anyRequest()
 .authenticated();
 }
}
```

Precisamos também indicar qual a *URI* que deve ser usada para chegar na página de *Login* e qual *URI* deve ser usada para fazer *Logout*. E essas devem ser liberadas para qualquer um acessar.

```
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
 @Override
 protected void configure(HttpSecurity http) throws Exception {
 http
 .authorizeRequests()
 .antMatchers("/admin/**").hasRole("ADMIN")
 .antMatchers("/compra/**").hasRole("COMPRADOR")
 .antMatchers("/filme/**").permitAll()
 .antMatchers("/sessao/**/lugares").permitAll()
 .antMatchers("//").permitAll()
 .anyRequest()
 .authenticated()
 .and()
 .formLogin()
 .usernameParameter("email")
 .loginPage("/login")
 .permitAll()
 .and()
 .logout()
 .logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
 .permitAll();
 }
}
```

```
}
```

Por padrão o *Spring Security* vem habilitado a verificação de *CSRF*, que consiste em verificar se em todos nossos formulários tem um *INPUT HIDDEN* com um identificador (token) gerado aleatoriamente.

Como não estamos gerando esse *Token* e nem colocando esse *INPUT HIDDEN* em nossos formulários, vamos desabilitar uma prevenção de segurança contra ataques *CSRF*.

```
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
 @Override
 protected void configure(HttpSecurity http) throws Exception {
 http
 .csrf().disable().authorizeRequests()
 .antMatchers("/admin/**").hasRole("ADMIN")
 .antMatchers("/compra/**").hasRole("COMPRADOR")
 .antMatchers("/filme/**").permitAll()
 .antMatchers("/sessao/**/lugares").permitAll()
 .antMatchers("//").permitAll()
 .anyRequest()
 .authenticated()
 .and()
 .formLogin()
 .usernameParameter("email")
 .loginPage("/login")
 .permitAll()
 .and()
 .logout()
 .logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
 .permitAll();
 }
}
```

Além disso precisamos liberar *requests* para nossos arquivos estáticos dentro de `webapp/assets` .

Podemos adicionar um novo `antMatcher` para `/assets/**` e usar o `permitAll()` . Porém, para evitar colocar exceções as regras de acesso, vamos sobrescrever outro método da classe `WebSecurityConfigurerAdapter` .

Dessa vez vamos sobrescrever o método `public void configure(WebSecurity web) throws Exception` . Através do objeto `WebSecurity` podemos pedir para que seja ignorado uma ou mais *URIs*:

```
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
 @Override
 protected void configure(HttpSecurity http) throws Exception {
 http
 .csrf().disable().authorizeRequests()
 .antMatchers("/admin/**").hasRole("ADMIN")
 .antMatchers("/compra/**").hasRole("COMPRADOR")
 .antMatchers("/filme/**").permitAll()
 .antMatchers("/sessao/**/lugares").permitAll()
 .antMatchers("//").permitAll()
 .anyRequest()
 .authenticated()
 .and()
 .formLogin()
 .usernameParameter("email")
```

```

 .loginPage("/login")
 .permitAll()
 .and()
 .logout()
 .logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
 .permitAll();
}

@Override
public void configure(WebSecurity web) throws Exception {
 web.ignoring().antMatchers("/assets/**");
}
}

```

Por fim precisamos avisar ao *Spring* que essa classe deve ser utilizada para fazer as verificações de segurança. Para isso vamos anotá-la com `@EnableWebSecurity` :

```

@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
 @Override
 protected void configure(HttpSecurity http) throws Exception {
 http
 .csrf().disable().authorizeRequests()
 .antMatchers("/admin/**").hasRole("ADMIN")
 .antMatchers("/compra/**").hasRole("COMPRADOR")
 .antMatchers("/filme/**").permitAll()
 .antMatchers("/sessao/**/lugares").permitAll()
 .antMatchers("//").permitAll()
 .anyRequest()
 .authenticated()
 .and()
 .formLogin()
 .usernameParameter("email")
 .loginPage("/login")
 .permitAll()
 .and()
 .logout()
 .logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
 .permitAll();
 }

 @Override
 public void configure(WebSecurity web) throws Exception {
 web.ignoring().antMatchers("/assets/**");
 }
}

```

Agora precisamos registrar um filtro do *Spring Security* para que ele possa observar todas nossas *requests* e assim aplicar as regras de segurança que definimos.

O *Spring Security* disponibiliza uma classe chamada `AbstractSecurityWebApplicationInitializer` que já faz toda a parte de registrar o filtro e ficar observando as *requests*. Basta apenas que uma classe no nosso projeto herde de `AbstractSecurityWebApplicationInitializer`. E precisamos informar para o *Spring* que essa é uma classe de configuração, anotando-a com `@Configuration`.

---

```

@Configuration
public class SecurityInitializer extends AbstractSecurityWebApplicationInitializer {

```

```
}
```

Além disso precisamos que o ao inicializar o filtro seja carregado nossa classe de configuração de acesso.

Para isso vamos sobrescrever o construtor sem argumentos da nossa classe `SecurityInitializer` e chamar o construtor de `AbstractSecurityWebApplicationInitializer` passando para ele a classe de configuração.

```
@Configuration
public class SecurityInitializer extends AbstractSecurityWebApplicationInitializer {
 public SecurityInitializer() {
 super(SecurityConfiguration.class);
 }
}
```

## 7.3 EXERCÍCIO - IMPLEMENTANDO SEGURANÇA EM NOSSA APLICAÇÃO

1. Crie a classe `SecurityInitializer` no pacote `br.com.caelum.ingresso.configuracao` e faça com que ela herde de `AbstractSecurityWebApplicationInitializer` e anote-a com `@Configuration`:

```
@Configuration
public class SecurityInitializer extends AbstractSecurityWebApplicationInitializer {
}
```

2. Crie a classe `SecurityConfiguration` e faça com que ela herde de `WebSecurityConfigurerAdapter` anote-a com `@EnableWebSecurity`:

```
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
}
```

3. Sobrescreva o método `configure` que recebe um objeto `HttpSecurity`:

```
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

 @Override
 protected void configure(HttpSecurity http) throws Exception {
 }
}
```

4. Implemente nesse método as restrições para as `_URI_s`:

```
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

 @Override
```

```

protected void configure(HttpSecurity http) throws Exception {
 http
 .csrf().disable().authorizeRequests()
 .antMatchers("/admin/**").hasRole("ADMIN")
 .antMatchers("/compra/**").hasRole("COMPRADOR")
 .antMatchers("/filme/**").permitAll()
 .antMatchers("/sessao/**/lugares").permitAll()
 .antMatchers("/magic/**").permitAll()
 .antMatchers("//").permitAll()
 .anyRequest()
 .authenticated()
 .and()
 .formLogin()
 .usernameParameter("email")
 .loginPage("/login")
 .permitAll()
 .and()
 .logout()
 .logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
 .permitAll();
}

}

```

5. Sobrescreva o método `configure` que recebe um objeto `WebSecurity`, e libere os arquivos estáticos da nossa aplicação:

```

@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

 @Override
 protected void configure(HttpSecurity http) throws Exception {
 http
 .csrf().disable().authorizeRequests()
 .antMatchers("/admin/**").hasRole("ADMIN")
 .antMatchers("/compra/**").hasRole("COMPRADOR")
 .antMatchers("/filme/**").permitAll()
 .antMatchers("/sessao/**/lugares").permitAll()
 .antMatchers("/magic/**").permitAll()
 .antMatchers("//").permitAll()
 .anyRequest()
 .authenticated()
 .and()
 .formLogin()
 .usernameParameter("email")
 .loginPage("/login")
 .permitAll()
 .and()
 .logout()
 .logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
 .permitAll();
 }

 @Override
 public void configure(WebSecurity web) throws Exception {
 web.ignoring().antMatchers("/assets/**");
 }
}

```

6. Crie um construtor sem argumentos para a classe `SecurityInitializer` e nele chame o `super` e

passe a classe `SecurityConfiguration` para ele.

```
@Configuration
public class SecurityInitializer extends AbstractSecurityWebApplicationInitializer {

 public SecurityInitializer() {
 super(SecurityConfiguration.class);
 }

}
```

7. Crie um controller chamado `LoginController` e crie uma action para `/login` com método GET e retorne para a view login:

```
@Controller
public class LoginController {

 @GetMapping("/login")
 public String login(){
 return "login";
 }
}
```

## 7.4 USUÁRIO, SENHA E PERMISSÃO

Agora que já configuramos a segurança da nossa aplicação, precisamos criar algo que represente nosso usuário e quais permissões ele deve ter. Vamos começar modelando a classe `Permissao` e depois a classe `Usuario`:

```
@Entity
public class Permissao {
 @Id
 private String nome;

 public Permissao(String nome) {
 this.nome = nome;
 }

 /**
 * @deprecated hibernate only
 */
 public Permissao() {
 }

 //getters e setters
}

@Entity
public class Usuario {

 @Id
 @GeneratedValue
 private Integer id;

 private String email;
 private String password;

 @ManyToMany(fetch = FetchType.EAGER)
```

```

private Set<Permissao> permissoes = new HashSet<>();

/**
 * @deprecated hibernate only
 */
public Usuario() {
}

public Usuario(String email, String password, Set<Permissao> permissoes) {
 this.email = email;
 this.password = password;
 this.permissoes = permissoes;
}

//getters e setters
}

```

Precisamos que o *Spring Security* saiba pegar as informações de `login`, `senha` e `permissão`. Para isso precisamos que nossa classe `Permissao` implemente a interface `GrantedAuthority` e nossa classe `Usuario` implemente a interface `UserDetails`.

Dessa forma vamos ser obrigados a implementar os métodos que retornam exatamente as informações de `login`, `senha` e `permissão` (além de algumas outras se quisermos implementar).

```

@Entity
public class Permissao implements GrantedAuthority {
 // ... restante da implementação

 @Override
 public String getAuthority() {
 return nome;
 }
}

@Entity
public class Usuario implements UserDetails {
 // ... restante da implementação

 @Override
 public Collection<? extends GrantedAuthority> getAuthorities() {
 return permissoes;
 }

 @Override
 public String getPassword() {
 return password;
 }

 @Override
 public String getUsername() {
 return email;
 }

 @Override
 public boolean isAccountNonExpired() {
 return true;
 }
}

```

```

@Override
public boolean isAccountNonLocked() {
 return true;
}

@Override
public boolean isCredentialsNonExpired() {
 return true;
}

@Override
public boolean isEnabled() {
 return true;
}

```

Além disso precisamos de uma classe que busque no banco de dados um `UserDetails` a partir de um `username`. Para isso vamos criar um *DAO* e implementar a interface `UserDetailsService`.

```

@Repository
public class LoginDao implements UserDetailsService {

 @PersistenceContext
 private EntityManager manager;

 @Override
 public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
 try {
 return manager
 .createQuery("select u from Usuario u where u.email = :email", Usuario.class)
 .setParameter("email", email)
 .getSingleResult();
 } catch (NoResultException e){
 throw new UsernameNotFoundException("Email " + email + " Não encontrado!");
 }
 }
}

```

Por fim precisamos alterar nossas configurações de segurança para que seja utilizado `UserDetailsService` para validar se o usuário existe e/ou se ele tem as devidas permissões para acessar o recurso.

Para isso vamos alterar a classe `SecurityConfiguration` para que ela receba injetado um `UserDetailsService` e sobrescrever o método `protected void configure(AuthenticationManagerBuilder auth) throws Exception`. e fazer com que o objeto `AuthenticationManagerBuilder` use nosso `UserDetailsService` como meio de autenticação.

É uma má prática salvar a senha em texto puro no banco de dados. Por isso precisamos codificá-la/criptografá-la antes de salvar. Com isso, nossa autenticação deve saber como comparar a senha que foi digitada no formulário de login com a senha que está salva no banco.

Para isso precisamos informar qual o algoritmo para codificar/criptografar a senha foi utilizado na hora de salvar a senha no banco de dados. Existem diversos algoritmos para fazer essa tarefa por exemplo: *MD5*, *SHA1*, *SHA1-256*, *SHA1-512*, *BCrypt* entre outros.

No nosso caso iremos utilizar uma implementação do *BCrypt* que é recomendado na documentação do *Spring Security*.

Essa implementação é o `BCryptPasswordEncoder` :

```
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

 @Autowired
 private UserDetailsService userDetailsService;

 //demais métodos

 @Override
 protected void configure(AuthenticationManagerBuilder auth) throws Exception {
 auth.userDetailsService(userDetailsService).passwordEncoder(new BCryptPasswordEncoder());
 }

}
```

Como nossas configurações do *Spring* estão em *XML* precisamos que nossa classe `SecurityConfiguration` leia esse *XML*, do contrário não será possível injetar `UserDetailsService`. Para isso vamos anotar nossa classe com `@ImportResource("/WEB-INF/spring-context.xml")`

```
@EnableWebSecurity
@ImportResource("/WEB-INF/spring-context.xml")
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

 @Autowired
 private UserDetailsService userDetailsService;

 //demais métodos

 @Override
 protected void configure(AuthenticationManagerBuilder auth) throws Exception {
 auth.userDetailsService(userDetailsService).passwordEncoder(new BCryptPasswordEncoder());
 }

}
```

## 7.5 EXERCÍCIO - IMPLEMENTANDO USERDETAILS, USERDETAILSSERVICE E GRANTEDAUTHORITY

1. Crie a classe `Permissao` no pacote `br.com.caelum.ingresso.modelo` :

```
@Entity
public class Permissao {
 @Id
 private String nome;

 public Permissao(String nome) {
 this.nome = nome;
 }

 /**
 * @deprecated hibernate only
 */
}
```

```

public Permissao() {
}

public String getNome() {
 return nome;
}

public void setNome(String nome) {
 this.nome = nome;
}
}

```

2. Crie a classe `Usuario` no pacote `br.com.caelum.ingresso.modelo`:

```

@Entity
public class Usuario {

 @Id
 @GeneratedValue
 private Integer id;

 private String email;
 private String password;

 @ManyToMany(fetch = FetchType.EAGER)
 private Set<Permissao> permissoes = new HashSet<>();

 /**
 * @deprecated hibernate only
 */
 public Usuario() {
 }

 public Usuario(String email, String password, Set<Permissao> permissoes) {
 this.email = email;
 this.password = password;
 this.permissoes = permissoes;
 }

 //getters e setters
}

```

3. Faça com que a classe `Permissao` implemente a interface `GrantedAuthority` e faça com que o método `getAuthority` retorne o nome da permissão:

```

@Entity
public class Permissao implements GrantedAuthority {
 @Id
 private String nome;

 public Permissao(String nome) {
 this.nome = nome;
 }

 /**
 * @deprecated hibernate only
 */
 public Permissao() {
 }

```

```

public String getNome() {
 return nome;
}

public void setNome(String nome) {
 this.nome = nome;
}

@Override
public String getAuthority() {
 return nome;
}
}

```

4. Faça com que a classe `Usuario` implemente a interface `UserDetail` :

```

@Entity
public class Usuario implements UserDetails {
 ...

 @Override
 public Collection<? extends GrantedAuthority> getAuthorities() {
 return permissoes;
 }

 @Override
 public String getPassword() {
 return password;
 }

 @Override
 public String getUsername() {
 return email;
 }

 @Override
 public boolean isAccountNonExpired() {
 return true;
 }

 @Override
 public boolean isAccountNonLocked() {
 return true;
 }

 @Override
 public boolean isCredentialsNonExpired() {
 return true;
 }

 @Override
 public boolean isEnabled() {
 return true;
 }
}

```

5. Crie a classe `LoginDao` e implemente a interface `UserDetailsService` faça uma query para retornar um `Usuario` por email:

```

@Repository
public class LoginDao implements UserDetailsService {

 @PersistenceContext
 private EntityManager manager;

 @Override
 public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
 try {
 return manager
 .createQuery("select u from Usuario u where u.email = :email", Usuario.class)
 .setParameter("email", email)
 .getSingleResult();
 } catch (NoResultException e){
 throw new UsernameNotFoundException("Email " + email + " Não encontrado!");
 }
 }
}

```

6. Na classe SecurityConfiguration vamos adicionar a anotação @ImportResource para podermos injetar nosso UserDetailsService :

```

@EnableWebSecurity
@ImportResource("/WEB-INF/spring-context.xml")
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
 //restante da implementação
}

```

7. Injete UserDetailsService e sobrescreva o método configure que recebe um AuthenticationManagerBuilder :

```

@EnableWebSecurity
@ImportResource("/WEB-INF/spring-context.xml")
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

 @Autowired
 private UserDetailsService userDetailsService;

 //demais métodos

 @Override
 protected void configure(AuthenticationManagerBuilder auth) throws Exception {
 auth.userDetailsService(userDetailsService).passwordEncoder(new BCryptPasswordEncoder());
 }
}

```

# APÊNDICE: CRIANDO UMA NOVA CONTA

**APÊNDICE NÃO FAZ PARTE DO CURSO.** Os apêndices são conteúdos adicionais que não fazem parte da carga horária regular do curso. São conteúdos extras para direcionar seus estudos após o curso.

## 8.1 IMPLEMENTANDO SEGURANÇA EM NOSSO SISTEMA

Quando queremos criar uma nova conta em alguma aplicação, geralmente existe um formulário onde preenchemos nossa conta de e-mail.

E ao submetermos esse formulário é enviado um e-mail para a conta informada com um link para criação da nossa conta. Esse link nos leva a outro formulário onde precisamos preencher a senha e a confirmação da senha e as vezes um login.

Vamos implementar essa funcionalidade em nossa aplicação. Para isso precisamos ter um link na tela de login que irá levar um visitante para o formulário de solicitação de acesso.

Vamos adicionar na página de login um link para que os visitantes possam efetuar o cadastro.

```
...
<button class="btn btn-primary" type="submit">Entrar</button>
ou cadastrar-se
...
```

Precisamos agora implementar um *Controller* que ficará responsável por lidar com as *requests* para usuário. Vamos criar a classe *UsuarioController* e implementar um método com um mapeamento *\_GET* para a *URI* /usuario/request

```
@Controller
public class UsuarioController {
 @GetMapping("/usuario/request")
 public ModelAndView formSolicitacaoDeAcesso(){
 return new ModelAndView("usuario/form-email");
 }
}
```

Agora precisamos criar o formulário para o usuário informar o e-mail ao qual ele quer solicitar acesso. Vamos criar o arquivo `usuario/form-email.jsp`:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
 pageEncoding="UTF-8" %>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>
<%@ taglib tagdir="/WEB-INF/tags/" prefix="ingresso" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<ingresso:template>
 <jsp:body>
 <div class="col-md-6 col-md-offset-3">
 <form action="/usuario/request" method="post">
 ${param.error}

 <div class="form-group">
 <label for="login">E-mail:</label>
 <input id="login" type="text" name="email" class="form-control">
 </div>

 <button class="btn btn-primary" type="submit">Solicitar Acesso</button>
 </form>
 </div>
 </jsp:body>
</ingresso:template>
```

Ao submeter esse formulário vamos levar o usuário para uma página que informe a ele que, enviamos um e-mail para a criação do acesso. Para isso vamos criar o arquivo `usuario/adicionado.jsp`:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
 pageEncoding="UTF-8" %>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>
<%@ taglib tagdir="/WEB-INF/tags/" prefix="ingresso" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<ingresso:template>
 <jsp:body>
 <h3>Usuário criado com sucesso!</h3>

 <p>
 Enviamos um e-mail com um link de confirmação para ${usuario.email}.

 </p>

 Por favor confirme a criação do seu usuário, para liberar seu acesso.
 </p>

 </jsp:body>
</ingresso:template>
```

Como implementamos segurança em nossa aplicação precisamos liberar o acesso para *URIs* em `/usuario`.

```
@EnableWebSecurity
@ImportResource("/WEB-INF/spring-context.xml")
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

 @Autowired
 private UserDetailsService userDetailsService;
```

```

@Override
protected void configure(HttpSecurity http) throws Exception {
 http
 .csrf().disable().authorizeRequests()
 .antMatchers("/admin/**").hasRole("ADMIN")
 .antMatchers("/compra/**").hasRole("COMPRADOR")
 .antMatchers("/usuario/**").permitAll() // <== nova linha
 .antMatchers("/filme/**").permitAll()
 .antMatchers("/sessao/**/lugares").permitAll()
 .antMatchers("/magic/**").permitAll()
 .antMatchers("//").permitAll()
 .anyRequest()
 .authenticated()
 .and()
 .formLogin()
 .usernameParameter("email")
 .loginPage("/login")
 .permitAll()
 .and()
 .logout()
 .logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
 .permitAll();
}

@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
 auth.userDetailsService(userDetailsService).passwordEncoder(new BCryptPasswordEncoder());
}

@Override
public void configure(WebSecurity web) throws Exception {
 web.ignoring().antMatchers("/assets/**");
}

}

```

## 8.2 EXERCÍCIO - CRIANDO FORMULÁRIO DE SOLICITAÇÃO DE ACESSO

1. Na página `login.jsp`, adicione o link para cadastrar um novo usuário após o botão de *Entrar*:

```

...
<button class="btn btn-primary" type="submit">Entrar</button>
ou cadastrar-se
...

```

2. Crie a classe `UsuarioController` com um mapeamento *GET* para `/usuario/request/` e que retorne para página `usuario/form-email`:

```

@Controller
public class UsuarioController {
 @GetMapping("/usuario/request")
 public ModelAndView formSolicitacaoDeAcesso(){
 return new ModelAndView("usuario/form-email");
 }
}

```

```
}
```

3. Crie a página `usuario/form-email.jsp` :

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
 pageEncoding="UTF-8" %>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>
<%@ taglib tagdir="/WEB-INF/tags/" prefix="ingresso" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<ingresso:template>
 <jsp:body>
 <div class=" col-md-6 col-md-offset-3">
 <form action="/usuario/request" method="post">
 ${param.error}

 <div class="form-group">
 <label for="login">E-mail:</label>
 <input id="login" type="text" name="email" class="form-control">
 </div>

 <button class="btn btn-primary" type="submit">Solicitar Acesso</button>
 </form>
 </div>
 </jsp:body>
</ingresso:template>
```

4. Crie um mapeamento `POST` para `/usuario/request` que retorne para página `usuario/adicionado.jsp` :

```
@PostMapping("/usuario/request")
public ModelAndView solicitacaoDeAcesso(String email){

 ModelAndView view = new ModelAndView("usuario/adicionado");

 return view;
}
```

5. Crie a página `usuario/adicionado.jsp` :

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
 pageEncoding="UTF-8" %>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>
<%@ taglib tagdir="/WEB-INF/tags/" prefix="ingresso" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<ingresso:template>
 <jsp:body>
 <h3>Usuário criado com sucesso!</h3>

 <p>
 Enviamos um e-mail com um link de confirmação para ${usuario.email}.

 Por favor confirme a criação do seu usuário, para liberar seu acesso.
 </p>

 </jsp:body>
</ingresso:template>
```

6. Libere o acesso para `URI /usuario` na classe `SecurityConfiguration` :

---

```

@Override
protected void configure(HttpSecurity http) throws Exception {
 http
 .csrf().disable().authorizeRequests()
 .antMatchers("/admin/**").hasRole("ADMIN")
 .antMatchers("/compra/**").hasRole("COMPRADOR")
 .antMatchers("/usuario/**").permitAll() // <== nova linha
 .antMatchers("/filme/**").permitAll()
 .antMatchers("/sessao/**/lugares").permitAll()
 .antMatchers("/magic/**").permitAll()
 .antMatchers("//").permitAll()
 .anyRequest()
 .authenticated()
 .and()
 .formLogin()
 .usernameParameter("email")
 .loginPage("/login")
 .permitAll()
 .and()
 .logout()
 .logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
 .permitAll();
}

}

```

## 8.3 CONFIGURAÇÕES PARA ENVIO DE E-MAILS

Agora que temos a primeira parte do fluxo pronta, vamos implementar o envio do e-mail. O e-mail que queremos enviar deve ter um identificador único vinculado a conta que foi informada no formulário de solicitação de acesso.

Dessa forma não precisamos que o usuário preencha novamente a sua conta de e-mail. Esse identificador único pode ser qualquer coisa, um número, uma combinação de letras. No nosso caso iremos utilizar a classe `UUID` (Unique identifier).

Para dar mais semântica ao nosso código vamos modelar algo que represente, o vínculo entre o identificador único e a conta de e-mail informada. Para isso vamos criar a classe `Token`

```

@Entity
public class Token {

 @Id
 private String uuid;

 @Email
 private String email;

 public Token(){}
 public Token(String email) {
 this.email = email;
 }
 //getters e setters
}

```

Para não nos preocuparmos com a geração do `UUID` vamos adicionar um *Callback* da *JPA* na nossa entidade `Token`. Esses *callbacks* podem ser comparados com as *Triggers* em um banco dados. Ou seja antes ou depois de determinados eventos (*Insert*, *Update*, *Delete*) podemos executar algum código.

No nosso caso queremos que antes de salvar um `Token` no banco de dados, ele gere o identificador `UUID`. Para isso vamos criar um método e anotá-lo com `@PrePersist`, e nele preencher o atributo `uuid`:

```
@Entity
public class Token {
 //... restante da implementação

 @PrePersist
 public void prePersist(){
 uuid = UUID.randomUUID().toString();
 }
}
```

Os e-mails enviado a partir de uma aplicação geralmente seguem um template, e podemos ter mais de um tipo de e-mails em nossa aplicação. Mais uma vez podemos nos aproveitar do *Design Pattern Strategy* para criar tipos diferentes de templates de e-mails.

Para isso vamos criar uma interface `Email`:

```
public interface Email {
 String getTo();
 String getBody();
 String getSubject();
}
```

Vamos criar também um template de e-mail para solicitação de acesso.

```
public class EmailNovoUsuario implements Email {
 private final Token token;

 public EmailNovoUsuario(Token token) {
 this.token = token;
 }

 @Override
 public String getTo() {
 return token.getEmail();
 }

 @Override
 public String getBody() {
 StringBuilder body = new StringBuilder("<html>");
 body.append("<body>");
 body.append("<h2> Bem Vindo </h2>");
 body.append(String.format("Acesso o link para para criar seu login no sistema de ingressos.", makeURL()));
 body.append("</body>");
 body.append("</html>");

 return body.toString();
 }
}
```

```

@Override
public String getSubject() {
 return "Cadastro Sistema de Ingressos";
}

private String makeURL() {
 return String.format("http://localhost:8080/usuario/validate?uuid=%s", token.getUuid());
}
}

```

Agora precisamos criar uma classe que será responsável por enviar e-mails a partir da nossa aplicação. Essa classe deve receber um objeto `Email` e disparar o e-mail.

```

@Component
public class Mailer {

 @Autowired
 private JavaMailSender sender;

 private final String from = "Ingresso<cursofj22@gmail.com>";

 public void send(Email email) {
 MimeMessage message = sender.createMimeMessage();

 MimeMessageHelper messageHelper = new MimeMessageHelper(message);

 try {
 messageHelper.setFrom(from);
 messageHelper.setTo(email.getTo());
 messageHelper.setSubject(email.getSubject());
 messageHelper.setText(email.getBody(), true);

 sender.send(message);

 } catch (MessagingException e) {
 throw new IllegalArgumentException(e);
 }
 }
}

```

Para que o *Spring* consiga enviar e-mails, precisamos configurar o endereço do servidor de e-mail, porta, usuário, senha entre outras configurações.

Podemos observar essas configurações no arquivo `spring-context.xml`:

```

<bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">
 <property name="host" value="smtp.gmail.com"/>
 <property name="port" value="587"/>
 <property name="username" value="algum@gmail.com"/>
 <property name="password" value="UmaSenhaSuperSegura"/>

 <property name="javaMailProperties">
 <props>
 <prop key="mail.smtp.auth">true</prop>
 <prop key="mail.smtp.starttls.enable">true</prop>
 <prop key="mail.debug">true</prop>
 </props>

```

```
</property>
</bean>
```

## 8.4 EXERCÍCIO - CONFIGURANDO O ENVIO DE E-MAILS

1. Vamos aplicar mais uma vez o *Strategy* para que possamos ter vários tipos de e-mails na nossa aplicação. Para isso crie a interface `Email` no pacote `br.com.caelum.ingresso.mail`:

```
public interface Email {
 String getTo();
 String getBody();
 String getSubject();
}
```

2. Vamos criar o Token que será enviado no nosso e-mail de novo usuário:

```
@Entity
public class Token {

 @Id
 private String uuid;

 @Email
 private String email;

 public Token(){}
 public Token(String email) {
 this.email = email;
 }
 //getters e setters
 @PrePersist
 public void prePersist(){
 uuid = UUID.randomUUID().toString();
 }
}
```

3. Vamos criar uma implementação da interface `Email` que represente o e-mail de novo usuário. Para isso crie a classe `EmailNovoUsuario` no pacote `br.com.caelum.ingresso.mail` e faça com que ela implemente a interface `Email`:

```
public class EmailNovoUsuario implements Email {
 private final Token token;
 public EmailNovoUsuario(Token token) {
 this.token = token;
 }
 @Override
 public String getTo() {
 return token.getEmail();
 }
 @Override
 public String getBody() {
 StringBuilder body = new StringBuilder("<html>");

```

```

 body.append("<body>");
 body.append("<h2> Bem Vindo </h2>");
 body.append(String.format("Acesso o link para para criar seu login no sistema de ingressos.", makeURL()));
 body.append("</body>");
 body.append("</html>");

 return body.toString();
}

@Override
public String getSubject() {
 return "Cadastro Sistema de Ingressos";
}

private String makeURL() {
 return String.format("http://localhost:8080/usuario/validate?uuid=%s", token.getUuid());
}
}

```

4. Crie `Mailer` que será responsável por enviar e-mails a partir da nossa aplicação:

```

@Component
public class Mailer {

 @Autowired
 private JavaMailSender sender;

 private final String from = "Ingresso<cursofj22@gmail.com>";

 public void send(Email email) {
 MimeMessage message = sender.createMimeMessage();

 MimeMessageHelper messageHelper = new MimeMessageHelper(message);

 try {

 messageHelper.setFrom(from);
 messageHelper.setTo(email.getTo());
 messageHelper.setSubject(email.getSubject());
 messageHelper.setText(email.getBody(), true);

 sender.send(message);

 } catch (MessagingException e) {
 throw new IllegalArgumentException(e);
 }
 }
}

```

## 8.5 SALVANDO TOKEN E ENVIANDO E-MAIL

Agora que já temos a configuração para envio de e-mails, temos que persistir o *Token* no banco e disparar um e-mail para a conta informada no formulário.

Para isso vamos criar a classe `TokenDao` que irá persistir o *Token* no banco de dados:

```

@Repository
public class TokenDao {

```

```

@PersistenceContext
private EntityManager manager;

public void save(Token token) {
 manager.persist(token);
}
}

```

Além disso precisamos de alguém que use a classe `TokenDao` para persistir e ainda assim devolva o `Token` para enviarmos o e-mail. Vamos criar a classe que irá nos ajudar com a manipulação de `Tokens`.

```

@Component
public class TokenHelper {
 @Autowired
 private TokenDao dao;

 public Token generateFrom(String email) {

 Token token = new Token(email);

 dao.save(token);

 return token;
 }
}

```

Agora só precisamos alterar o método `solicitacaoDeAcesso` na classe `UsuarioController` para que ele use o `TokenHelper` e `Mailer` para salvar o `Token` e enviar o e-mail. Para isso vamos injetar `TokenHelper` e `Mailer` na classe `UsuarioController` e alterar o método `solicitacaoDeAcesso`

```

@Controller
public class UsuarioController {

 @Autowired
 private Mailer mailer;

 @Autowired
 private TokenHelper tokenHelper;

 @PostMapping("/usuario/request")
 @Transactional
 public ModelAndView solicitacaoDeAcesso(String email){

 ModelAndView view = new ModelAndView("usuario/adicionado")

 Token token = tokenHelper.generateFrom(email);

 mailer.send(new EmailNovoUsuario(token));

 return view;
 }
}

```

## 8.6 EXERCÍCIO - ENVIANDO E-MAIL

- Antes de enviar o e-mail precisamos gerar um token para o novo usuário. Para isso vamos criar a classe `TokenDao` :

```

@Repository
public class TokenDao {
 @PersistenceContext
 private EntityManager manager;

 public void save(Token token) {
 manager.persist(token);
 }
}

```

2. Para criar um novo e-mail de cadastro precisamos do Token salvo no banco. Para isso vamos criar a classe `TokenHelper` no pacote `br.com.caelum.ingresso.helper`:

```

@Component
public class TokenHelper {
 @Autowired
 private TokenDao dao;

 public Token generateFrom(String email) {

 Token token = new Token(email);

 dao.save(token);

 return token;
 }
}

```

3. Injete a classe `Mailer` e `TokenHelper` em `UsuarioController`:

```

@Controller
public class UsuarioController {

 @Autowired
 private Mailer mailer;

 @Autowired
 private TokenHelper tokenHelper;

 // ... demais implementações

}

```

4. Altere o método `solicitacaoDeAcesso` na classe `UsuarioController` para que ele envie o e-mail com um token para o usuário:

```

@PostMapping("/usuario/request")
@Transactional
public ModelAndView solicitacaoDeAcesso(String email){

 ModelAndView view = new ModelAndView("usuario/adicionado")

 Token token = tokenHelper.generateFrom(email);

 mailer.send(new EmailNovoUsuario(token));

 return view;
}

```

## 8.7 IMPLEMENTANDO VALIDAÇÃO

Agora precisamos nos preocupar com a segunda parte da liberação de acesso.

Quando o usuário clicar no link do e-mail ele deve ser redirecionado para um formulário solicitando a senha e a confirmação de senha mas somente se o *Token* que estiver no link for válido. Ou seja se o *Token* existir no banco de dados.

Vamos criar um método na classe `UsuarioController` chamado `validaLink` e este por sua vez deve pegar o parâmetro `uuid` que adicionamos no link.

```
@GetMapping("/usuario/validate")
public ModelAndView validaLink(@RequestParam("uuid") String uuid){

}
```

Vamos implementar um método na classe `TokenDao` que deve retornar um *Token* a partir de um `uuid` caso ele exista.

```
public Optional<Token> findByUuid(String uuid) {
 return manager.createQuery("select t from Token t where t.uuid = :uuid", Token.class)
 .setParameter("uuid", uuid)
 .getResultList()
 .stream()
 .findFirst();
}
```

Como não acessamos `TokenDao` diretamente no nosso controller vamos criar um método chamado `getTokenFrom` na classe `TokenHelper`.

```
public Optional<Token> getTokenFrom(String uuid) {
 return dao.findByUuid(uuid);
}
```

Agora vamos alterar o método `validaLink` para que ele pegue o *Token* a partir do `uuid`. Caso o *Token* não exista, vamos redirecionar o usuário para a tela de *Login* com uma mensagem.

```
@GetMapping("/usuario/validate")
public ModelAndView validaLink(@RequestParam("uuid") String uuid){

 Optional<Token> optionalToken = tokenHelper.getTokenFrom(uuid);

 if (!optionalToken.isPresent()){

 ModelAndView view = new ModelAndView("redirect:/login");

 view.addObject("msg", "O token do link utilizado não foi encontrado!");

 return view;
 }
}
```

Caso o *Token* exista precisamos redirecionar o usuário para o formulário que irá solicitar a senha e a

confirmação de senha.

Como não temos um modelo que represente *Token*, senha e confirmação de senha. Vamos começar criando um *DTO* chamado `ConfirmacaoLoginForm` e vamos aproveitar e já implementar um método que valida se a senha e a confirmação de senha são iguais.

```
public class ConfirmacaoLoginForm {
 private Token token;
 private String password;
 private String confirmPassword;

 public ConfirmacaoLoginForm(){}
 public ConfirmacaoLoginForm(Token token) {
 this.token = token;
 }
 //getters e setters
 public boolean isValid(){
 return password.equals(confirmPassword);
 }
}
```

Agora vamos redirecionar para o formulário que deve usar o *DTO* para receber os dados do formulário:

```
@GetMapping("/usuario/validate")
public ModelAndView validaLink(@RequestParam("uuid") String uuid){

 Optional<Token> optionalToken = tokenHelper.getTokenFrom(uuid);

 if (!optionalToken.isPresent()){

 ModelAndView view = new ModelAndView("redirect:/login");

 view.addObject("msg", "O token do link utilizado não foi encontrado!");

 return view;
 }

 Token token = optionalToken.get();
 ConfirmacaoLoginForm confirmacaoLoginForm = new ConfirmacaoLoginForm(token);

 ModelAndView view = new ModelAndView("usuario/confirmacao");
 view.addObject("confirmacaoLoginForm", confirmacaoLoginForm);

 return view;
}
```

Agora precisamos criar a página do formulário de senha e confirmação de senha:

```
<ingresso:template>
 <jsp:body>
 <form action="/usuario/cadastrar" method="post">
 ${param.error}
```

```

<input type="hidden" name="token.uuid" value="${confirmacaoLoginForm.token.uuid}">
<input type="hidden" name="token.email" value="${confirmacaoLoginForm.token.email}">

<div class="form-group">
 <label for="password">Senha:</label>
 <input id="password" type="password" name="password" class="form-control">
</div>

<div class="form-group">
 <label for="confirmPassword">Senha:</label>
 <input id="confirmPassword" type="password" name="confirmPassword" class="form-control">
</div>

<button class="btn btn-primary" type="submit">Cadastrar</button>

</form>
</jsp:body>
</ingresso:template>

```

## 8.8 EXERCÍCIO - VALIDANDO TOKEN

- Crie um método `validaLink` na classe `UsuarioController` com mapeamento `GET` para `/usuario/validate`. E faça com que ele receba o parâmetro `uuid` a partir da `request`:

```

@GetMapping("/usuario/validate")
public ModelAndView validaLink(@RequestParam("uuid") String uuid){

}

```

- Adicione o método `findByIdUuid` na classe `TokenDao`

```

public Optional<Token> findByIdUuid(String uuid) {
 return manager.createQuery("select t from Token t where t.uuid = :uuid", Token.class)
 .setParameter("uuid", uuid)
 .getResultList()
 .stream()
 .findFirst();
}

```

- Crie o método `getTokenFrom` na classe `TokenHelper` que receberá uma `String` com o `UUID`. Esse método deve buscar o token no banco de dados:

```

public Optional<Token> getTokenFrom(String uuid) {
 return dao.findByIdUuid(uuid);
}

```

- Altere o método `validaLink` para que ele use o método `getTokenFrom` da classe `TokenHelper`. Caso o token não esteja presente, deve retornar para página de login:

```

@.GetMapping("/usuario/validate")
public ModelAndView validaLink(@RequestParam("uuid") String uuid){

 Optional<Token> optionalToken = tokenHelper.getTokenFrom(uuid);

 if (!optionalToken.isPresent()){

```

```

 ModelAndView view = new ModelAndView("redirect:/login");

 view.addObject("msg", "O token do link utilizado não foi encontrado!");

 return view;

 }
}

```

5. Caso o *Token* esteja presente devemos redirecionar o usuário para uma página, onde ele irá digitar a senha e confirmar a senha. Vamos criar um *DTO* para esse formulário. Crie a classe *ConfirmacaoLoginForm* :

```

public class ConfirmacaoLoginForm {
 private Token token;
 private String password;
 private String confirmPassword;

 public ConfirmacaoLoginForm(){}
 public ConfirmacaoLoginForm(Token token) {
 this.token = token;
 }
 //getters e setters
 public boolean isValid(){
 return password.equals(confirmPassword);
 }
}

```

6. Altere o método *validaLink* para que seja redirecionado para página *usuario/confirmacao* caso o *Token* esteja presente:

```

@GetMapping("/usuario/validate")
public ModelAndView validaLink(@RequestParam("uuid") String uuid){

 Optional<Token> optionalToken = tokenHelper.getTokenFrom(uuid);

 if (!optionalToken.isPresent()){

 ModelAndView view = new ModelAndView("redirect:/login");

 view.addObject("msg", "O token do link utilizado não foi encontrado!");

 return view;
 }

 Token token = optionalToken.get();
 ConfirmacaoLoginForm confirmacaoLoginForm = new ConfirmacaoLoginForm(token);

 ModelAndView view = new ModelAndView("usuario/confirmacao");
 view.addObject("confirmacaoLoginForm", confirmacaoLoginForm);

 return view;
}

```

7. Crie a página *usuario/confirmacao.jsp* :

```

<ingresso:template>
 <jsp:body>
 <form action="/usuario/cadastrar" method="post">
 ${param.error}

 <input type="hidden" name="token.uuid" value="${confirmacaoLoginForm.token.uuid}">
 <input type="hidden" name="token.email" value="${confirmacaoLoginForm.token.email}">

 <div class="form-group">
 <label for="password">Senha:</label>
 <input id="password" type="password" name="password" class="form-control">
 </div>

 <div class="form-group">
 <label for="confirmPassword">Senha:</label>
 <input id="confirmPassword" type="password" name="confirmPassword" class="form-control">
 </div>

 <button class="btn btn-primary" type="submit">Cadastrar</button>
 </form>
 </jsp:body>
</ingresso:template>

```

## 8.9 PERSISTINDO O USUÁRIO

Quando o usuário submeter o formulário com a senha e confirmação de senha, precisamos verificar se o formulário está valido. E em caso positivo devemos salvar o usuário com a senha criptografada.

Vamos começar implementando o método que irá receber os dados do formulário de confirmação e fazer a validação do formulário.

```

@PostMapping("/usuario/cadastrar")
public ModelAndView cadastrar(ConfirmacaoLoginForm form){
 ModelAndView view = new ModelAndView("redirect:/login");

 if (form.isValid()) {
 // Salvar usuário no banco
 }

 view.addObject("msg", "O token do link utilizado não foi encontrado!");

 return view;
}

```

Precisamos pegar um usuário a partir do formulário. Para isso vamos criar o método `toUsuario` na classe `ConfirmacaoLoginForm`, esse método deve pegar um usuário no banco de dados caso exista ou criar um novo usuário.

Além disso ele deve criptografar a senha usando `BCrypt`.

```

public class ConfirmacaoLoginForm {
 //... restante da implementação

 public Usuario toUsuario(UsuarioDao dao, PasswordEncoder encoder) {

```

```

 String encryptedPassword = encoder.encode(this.password);

 String email = token.getEmail();

 Usuario usuario= dao.findByEmail(email).orElse(novoUsuario(email, encryptedPassword));

 usuario.setPassword(encryptedPassword);

 return usuario;
 }

 private Usuario novoUsuario(String email, String password){
 Set<Permissao> permissoes = new HashSet<>();
 permissoes.add(Permissao.COMPRADOR);

 return new Usuario(email, password, permissoes);
 }
}

```

Para que o método `toUsuario` funcione vamos implementar o método `findByEmail` na classe `UsuarioDao`

```

@Repository
public class UsuarioDao {

 @PersistenceContext
 private EntityManager manager;

 public Optional<Usuario> findByEmail(String email) {

 return manager
 .createQuery("select u from Usuario u where u.email = :email", Usuario.class)
 .setParameter("email", email)
 .getResultList()
 .stream()
 .findFirst();
 }
}

```

Vamos alterar o método `cadastrar` na classe `UsuarioController` para que ele pegue o usuário do formulário e posteriormente salve o mesmo.

```

@PostMapping("/usuario/cadastrar")
@Transactional
public ModelAndView cadastrar(ConfirmacaoLoginForm form){
 ModelAndView view = new ModelAndView("redirect:/login");

 if (form.isValid()) {
 Usuario usuario = form.toUsuario(usuarioDao, passwordEncoder);

 usuarioDao.save(usuario);

 view.addObject("msg", "Usuario cadastrado com sucesso!");

 return view;
 }

 view.addObject("msg", "O token do link utilizado não foi encontrado!");
}

```

```

 return view;
 }

```

Por fim vamos implementar o método `save` na classe `UsuarioDao`:

```

public void save(Usuario usuario) {

 if (usuario.getId() == null)
 manager.persist(usuario);
 else
 manager.merge(usuario);
}

```

## 8.10 EXERCÍCIO - CADASTRANDO USUÁRIO

- Crie o método `cadastrar` na classe `UsuarioController` com mapeamento `POST` para `/usuario/cadastrar` e verifique se o formulário é válido:

```

@PostMapping("/usuario/cadastrar")
public ModelAndView cadastrar(ConfirmacaoLoginForm form){
 ModelAndView view = new ModelAndView("redirect:/login");

 if (form.isValid()) {
 // Salvar usuário no banco
 }

 view.addObject("msg", "O token do link utilizado não foi encontrado!");

 return view;
}

```

- Adicione o método `toUsuario` na classe `ConfirmacaoLoginForm`:

```

public class ConfirmacaoLoginForm {

 //... restante da implementação

 public Usuario toUsuario(UsuarioDao dao, PasswordEncoder encoder) {

 String encryptedPassword = encoder.encode(this.password);

 String email = token.getEmail();

 Usuario usuario= dao.findByEmail(email).orElse(novoUsuario(email, encryptedPassword));

 usuario.setPassword(encryptedPassword);

 return usuario;
 }

 private Usuario novoUsuario(String email, String password){
 Set<Permissao> permissoes = new HashSet<>();
 permissoes.add(Permissao.COMPRADOR);

 return new Usuario(email, password, permissoes);
 }
}

```

- Crie a classe `UsuarioDao` e implemente o método `findByEmail`, que deve retornar um usuário a

partir do e-mail se existir:

```
@Repository
public class UsuarioDao {

 @PersistenceContext
 private EntityManager manager;

 public Optional<Usuario> findByEmail(String email) {

 return manager
 .createQuery("select u from Usuario u where u.email = :email", Usuario.class)
 .setParameter("email", email)
 .getResultList()
 .stream()
 .findFirst();
 }

}
```

4. Caso o formulário seja válido devemos persistir o usuário no banco de dados. Para isso altere o método `cadastrar` na classe `UsuarioController`:

```
@PostMapping("/usuario/cadastrar")
@Transactional
public ModelAndView cadastrar(ConfirmacaoLoginForm form){
 ModelAndView view = new ModelAndView("redirect:/login");

 if (form.isValid()) {
 Usuario usuario = form.toUsuario(usuarioDao, passwordEncoder);

 usuarioDao.save(usuario);

 view.addObject("msg", "Usuario cadastrado com sucesso!");

 return view;
 }

 view.addObject("msg", "O token do link utilizado não foi encontrado!");
 return view;
}
```

5. Adicione o método `save` na classe `UsuarioDao`:

```
public void save(Usuario usuario) {

 if (usuario.getId() == null)
 manager.persist(usuario);
 else
 manager.merge(usuario);
}
```