Sumário

Versão: undefined

Capítulo 1

O MEDO DE SE ADAPTAR

Ao desenvolver projetos é comum que precisemos nos adaptar a situações novas e, assim, modificar o sistema para comportar melhor algo já existente ou implementar uma funcionalidade nova.

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

Outra palavra que causa desconforto é *mudança*. É corriqueiro pensarmos em mudança como perda, seja perda 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, e resistir a elas é fonte de diversos problemas que hoje vemos acontecendo em grande parte das empresas de desenvolvimento de software.

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, (a menos que tenhamos um backup) não conseguimos voltar ao estado anterior.

Não é de hoje que esse problema existe, por isso já há diversas formas de contornar ele. 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 que 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 precisamos de um lugar único onde vai ficar o nosso servidor que irá controlar as versões. E cada cliente se conecta ao servidor.

Além dessa categoria temos também ferramentas de *controle de versão distribuido*. Dentre elas temos as também algumas que se destacaram bastantes:

- **Git** - criado em 2005 por Linus Torvalds (criador do *Linux*) para controlar a versão do kernel do linux. Devido a ferramenta que estava sendo utilizada não estar atendendo os requisitos de performance entre outros fatores. *Git* é largametne adotado por muitas comunidades de desenvolvedores. Ele será a ferramenta que iremos utilizar durante o curso.

- **Mercurial** - também criado em 2005, 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** - também de 2005, criado pela empresa mantedora do Ubuntu, também compartilha dos mesmos comandos dos sistemas mais tradicionais. Era uma migração natural para os projetos open source hospedados no SourceForge, mas vem perdendo força, hoje em dia.

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. Só sincronizamos com o repositório central no momento que considerarmos adequado.

O primeiro passo para trabalhar com o Git é fazer uma cópia do repositório na nossa máquina. Chamamos esse processo de **clone**.

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

Ou criar um repositório local.

```
$ git init
```

Agora podemos começar a 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
  #
  # Changed but not updated:
  #   (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.esta.gerenciando.ainda
```

Essa resposta diz qual é o status do seu repositório local e o que você pode fazer com as alterações. Enquanto não adicionar um arquivo ao repositório, ele aparece na seção *Untracked files*. Para comitar esse arquivo, você precisa adicioná-lo ao próximo commit (como o Git está sugerindo):

```
$ git add arquivo.que.o.git.nao.esta.gerenciando.ainda
```

Por padrão o Git só comitará os arquivos que estiverem na seção *Changes to be commited*. Devemos então invocar o `git add` nos arquivos que estão na seção *Changed but not updated* para começar a registrar as mudanças.

```
$ 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 *Changed but not updated*, 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. Para fazer isso usamos o comando `git reset --hard` passando um identificador de versão.

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

Para descobrir o identificador da versão desejada, podemos usar o comando `git log` , que produz 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 fbbe794
Author: Claudineide <clau@clau.com>
Date:   Wed Nov 24 13:51:32 2010 -0200

    Adicionando outro tipo de combustível
```

Ás vezes percebemos que cometemos algum engano antes mesmo de comitar um arquivo, nesse caso desejamos reverter o arquivo para o estado anterior. Com o Git, basta usar o comando:

```
git checkout arquivo.errado
```

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 nele:

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

2. Crie um arquivo de texto com o nome **bio.txt** no diretório do seu repositório e no conteúdo dele coloque 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 não está sendo rastreado (*untracked*) pelo *git*.

3. Faça com que o arquivo seja rastreado (*tracked*) pelo *git*, e efetue o commit desse arquivo.

```
$ git add bio.txt
$ git commit -m "Adicionando arquivo bio"
```

4. Verifique qual o identificador (*hash*) do *commit* que você acabou de fazer.

```
$ git log
```

5. Inclua uma novas linhas no arquivo com algumas informações sobre o que você gosta de fazer , verifique novamente o estado do seu repositório, adicione o arquivo para ser comitado e faça um novo commit.

```
$ 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* agora como o git já está rastreando esse arquivo, ele está no estado *not staged*. Precisamos colocar ele em *staged* para que possamos efetuar o commit do mesmo, para isso já faremos o commit adicionando os arquivos nele :

Agora o arquivo está apto para ser comitado.

```
$ 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 repositórios de código que disponibilizam espaço gratuito para os projetos. Por vezes, a condição é que o código fique aberto, enquanto outros gratuitos permitem até mesmo repositórios fechados.

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, por exemplo para o Git, pode ser instalado um gitorious (http://gitorious.org/) em uma máquina da empresa.

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.

1.5 EXERCÍCIO - TRABALHANDO COM REPOSITÓRIOS REMOTOS.

1. Crie uma conta em https://github.com (caso já tenha conta pule esse exercício)
    -
        - Preencha os campos:
            - *Pick a username*
            - *Your email address*
            - *Create a password*
        - Clique em *Sing up for Github*

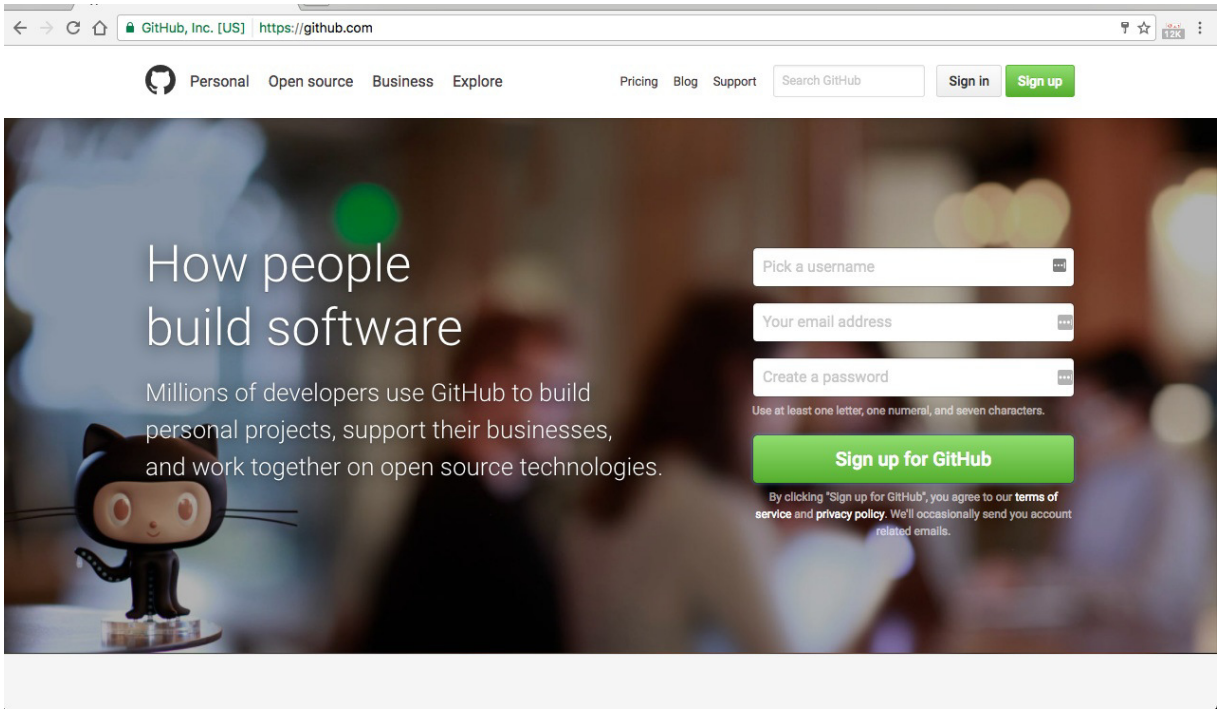


Figura 1.1: Homepage Github

- Selecione o tipo de conta que você quer usar:
  - Gratuita repositórios publicos ilimitados
  - Pagar para repositórios privados ilimitados

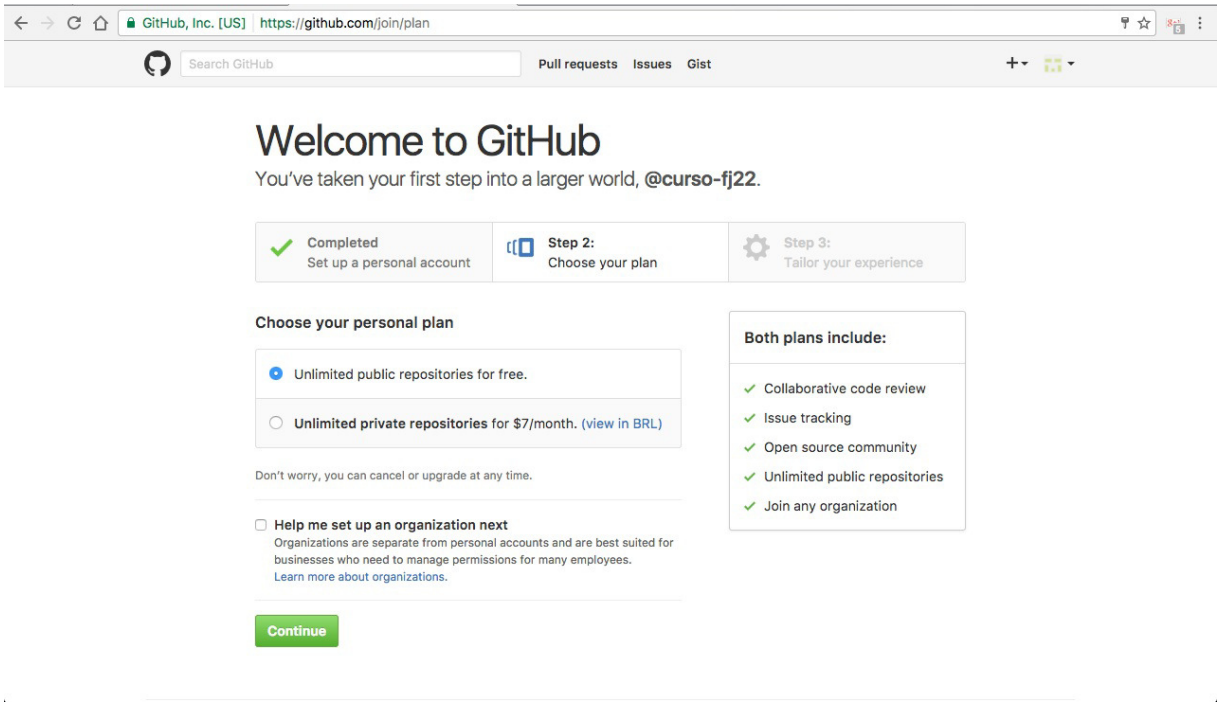


Figura 1.2: Welcome page step 2

- Selecione o nível de experiência com programação
- Selecione com o que você planeja usar o *Github*
- Qual o mais próximo que você se descrive
- O que você tem interesse

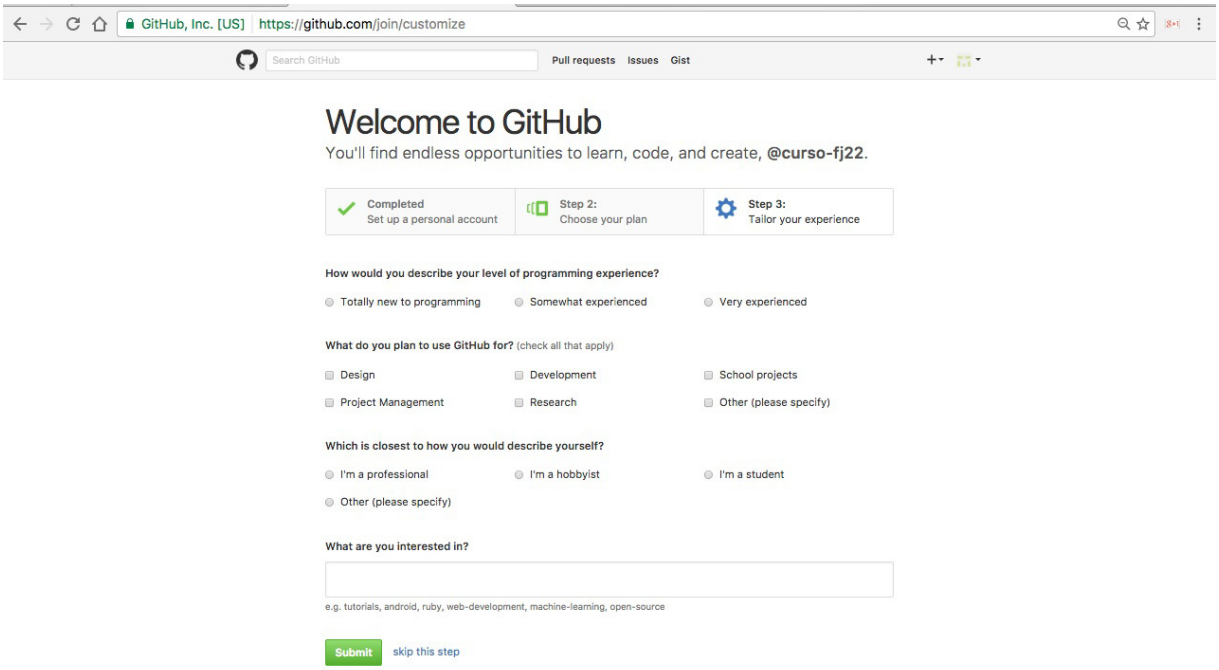


Figura 1.3: Welcome page step 3

- Selecione *Start a project* (pois não temos nenhum projeto)

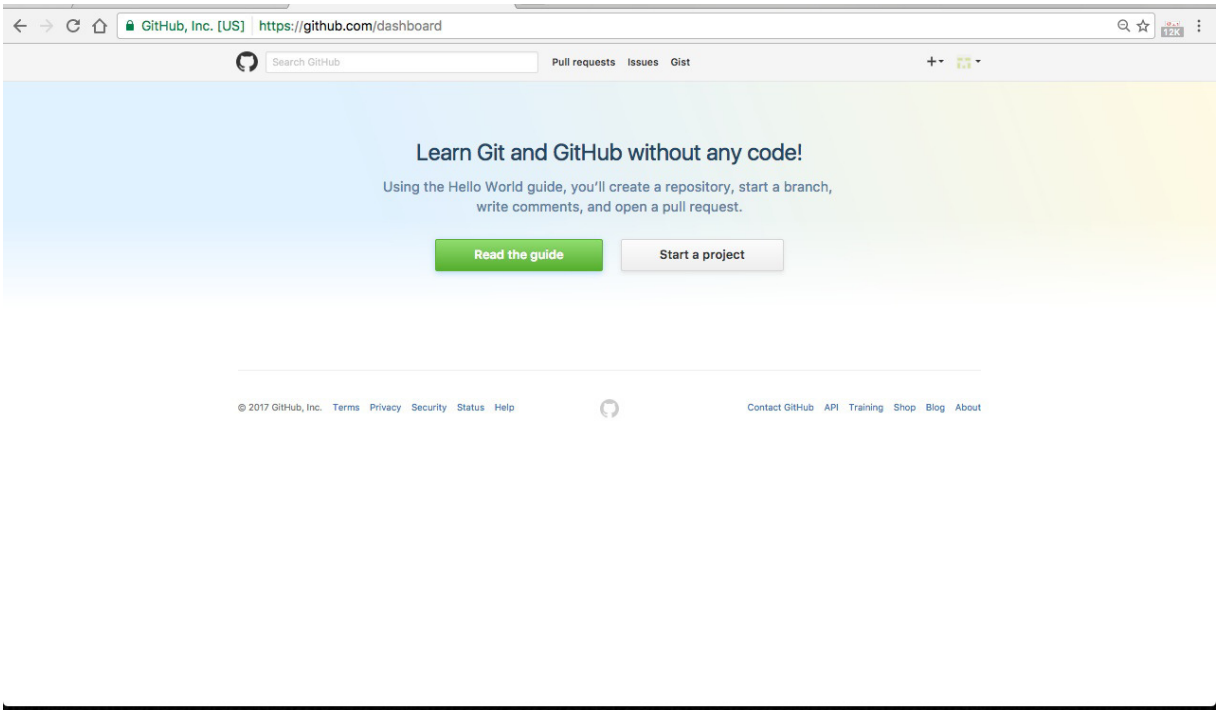


Figura 1.4: Dashboard

- Foi enviado um email de confirmação.
- Acesse seu e-mail e confirme a criação da sua conta.

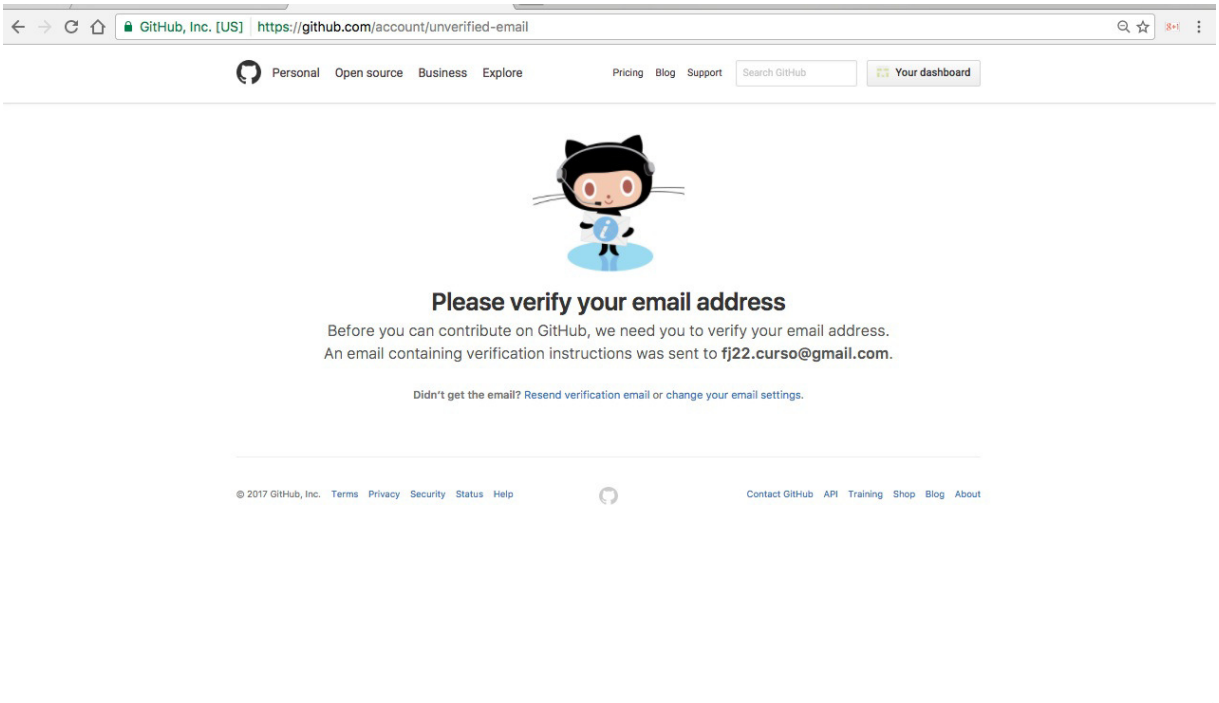


Figura 1.5: Verificação do e-mail

- Clique no link *Verify email address*

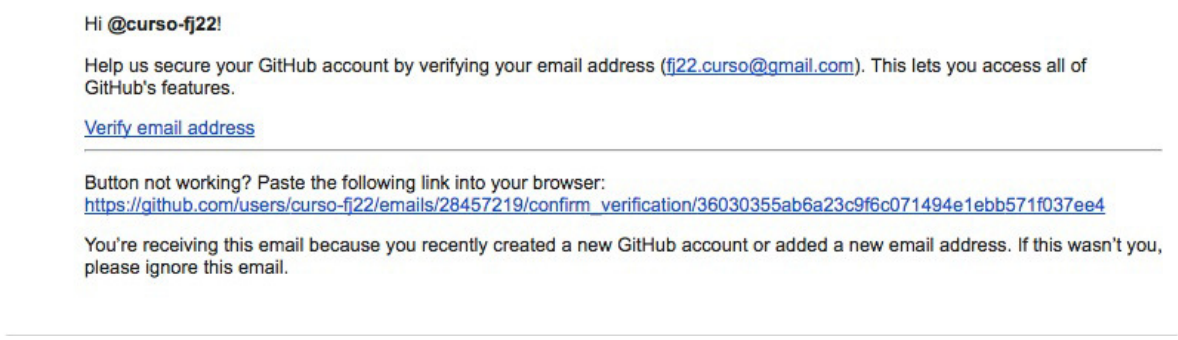


Figura 1.6: Email de verificação

- Sua conta está ativa.

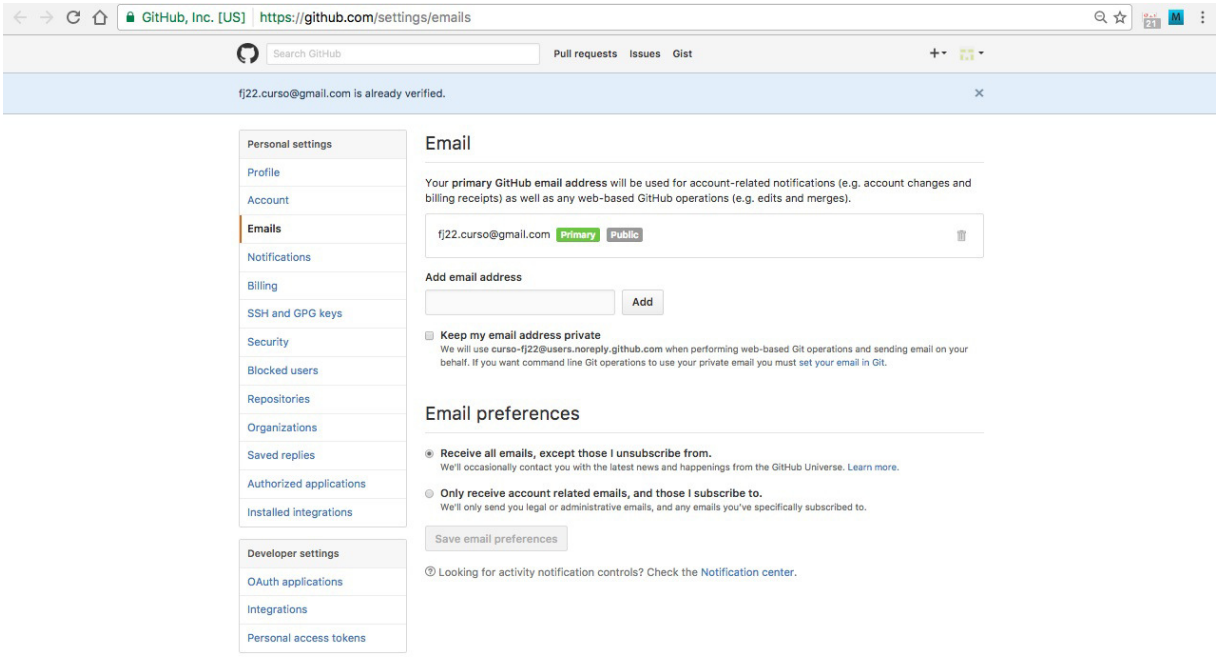


Figura 1.7: Verificado

2. Vamos fazer um *fork* do repositório fj22-ingressos da conta da caelum para nossa conta:
   - Acesse https://github.com/caelum/fj22-ingressos.git
   - Clique em *Fork*
3. Clone o projeto https://github.com/caelum/fj22-ingressos.git

```
$ git clone https://github.com/<Seu usuario aqui>/fj22-ingressos.git
```

```
$ cd fj22-ingressos
```

4. Crie um arquivo chamado "README.md" na pasta *fj22-ingressos*, com um conteúdo de descrição sobre você. Agora verifique qual o status de seu repositório:

```
$ git status
```

5. Vamos adicionar o arquivo para ser rastreado pelo *git*:

```
$ git add README.md
```

6. Antes de efetuar os commits, devemos configurar o nome e o email do nosso usuário no Git. Digite:

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

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

7. Execute um commit com uma mensagem de descrição condizente. Por exemplo:

```
$ git commit -m "adicionando descrição sobre um contribuinte novo ao projeto"
```

8. Agora precisamos mandar nossas alterações para nosso repositório remoto:

```
$ git push
```

1.6 LIDANDO COM BUILD E DEPENDENCIAS

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

O *IDE* se encarrega de colocar nossas dependencias no *classpath*(bastando apenas dizermos quais são as dependencias), criar o diretório que será colocado nosssa já compiladas, compilar o projeto, empacotar a aplicação e etc. Podemos até interferir nesse processo adicionando plugins para coletar métricas, extrair relatórios, fazer o deploy entre outras coisas.

Porém alguns desses processos que nossa *IDE* facilita, ainda tem muita intervenção manual. Por exemplo imagine que temos no nosso projeto precisamos usar o *Hibernate*, para colocar a dependencia do *Hibernate* no *classpath*, precisamos baixar o *.jar* do *Hibernate* e todas os arquivos *.jar* que o *Hibernate* depende. Imagine agora que no nosso projeto além do *Hibernate* vamos usar *Spring*, precisamos baixar o *.jar* do *Spring* e todos os *.jar* que o *Spring* depende.

Até agora nenhum problema, exceto o fato de termos que ficar baixando um monte de arquivo*.jar*. Mas e se o *Spring* e o *Hibernate* dependem do mesmo arquivo *.jar*? Nesse casos teremos que lidar com esse conflito manualmente, sempre antes de baixar olhando se essa dependencia já está no nosso projeto.

Para não ter nenhum duplicidade ou pior ter a mesma dependencia em versões diferentes.

Esse problema é tão comum que foram criadas várias ferramentas para automatizar esse processo de build e gerenciamento de depdencias.

Em java temos uma ferramenta bem popular para automação do build chamada *Ant*. Com ele, escrevemos um arquivo xml, onde temos nosso script de build, declarando os passos para construir o seu projeto (criar diretório de output, compilar nossas classe, compilar nossos teste, rodar nossos testes, empacotar a aplicação e etc).

Porém com ela não tinhamos ainda a gestão das nossas dependencias, e por isso usavamos outra ferramenta chamada *Ivy*.

Com o *Ivy* também tinhamos um arquivo xml onde declaramos nossas dependencias, e através de um repositório central o *Ivy* conseguia baixa-las.

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

Posteriormente pegaram o melhor dos dois mundos entre *Ant* e *Ivy* e criaram uma ferramenta hibrida chamada *Maven*.

Com o *Maven* temos um unico arquivo xml chamado **pom.xml**, nesse arquivo declaramos todas nossas dependencias e plugins para nosso processo de build, run ou deploy.

Além disso o *Maven* tem uma estrutura de pasta que deve ser seguida, e um ciclo de build bem definido.

Por padrão a estrutura de pasta que o *Maven* pede para seguirmos é:

```
+-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, xml, properties e arquivos estáticos do projeto
- src/main/webapp/ - _JSP_s, WEB-INF e etc
- src/test/java - classes de testes

- src/test/resources - arquivos de configuração, xml, properties e arquivos estáticos para teste

O ciclo de build do *Maven* tem as seguintes fases:

- validate - essa fase será verificado se a estrutura do projeto está de acordo
- compile - compila as classes de *src/main/java*
- test - compila as classes de *src/main/test* e roda todos os testes
- package - empacota nossa aplicação
- verify - informações de qualidade, verifica resultado de testes de integração (entre outras coisas)
- install - instala a dependencia localmente
- deploy - faz o deploy da sua aplicação baseado nas configurações do *pom.xml*

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

```
<project lns="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 basicas do seu projeto-->
  <groupId>br.com.caleum</groupId>
  <artifactId>fj22-ingresso</artifactId>
  <version>1.0</version>

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

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

          <!--Declarando a dependencia 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>


</project>
```

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

A sintaxe desse utilitário é `mvn` seguido da ação (*goal*) 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 temos também o *goal* `clean` , que apaga o diretório de output gerado a partir de compilação.

Como o *maven* tem um ciclo de build bem definido, ele sempre irá garantir que todas as fases antes do *goal* foi executado. Por exemplo, ao usar o *goal* `test` o *maven* irá executar `validate` , `compile` e depois o `test` . Sempre que executarmos um *goal* que tenha alguma antecessor, será executado seus antecessores e depois o *goal* solicitado.

Podemos inclusive combinar *goals*, por exepmlo: `mvn clean package` isso fára a limpeza dos outputs, compilará nosso projeto, rodará os testes e empacotará nossa aplicação.

Além dos *goals* padrões podemos utilizar plugins, um plugin por sua vez pode ter vários *goals*. E cada *goal* estará associado à uma fase do ciclo de build.

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

Ex.:

```
$ mvn dependency:tree
```

Esse plugin lista a arvore de dependencias da nossa aplicação.

1.7 EXERCÍCIO - USANDO GERENCIADOR DE DEPENDÊNCIA E BUILD

1. Acesse o diretório do projeto e liste e baixe as dependencias:

   ```
   $ cd fj22-ingressos

   $ mvn dependency:tree

   $ mvn dependency:resolve
   ```

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

   ```
   $ mvn package

   $ mvn 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:

   - Precione *CTRL+C* para interromper a execução do *jetty*.

     ```
     $ mvn clean
     ```

1.8 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 >> Run configurations...
   - Clique com o botão direito sobre *Maven Build* >> New
   - No campo *Name* preencha com "Rodar com maven"
   - No campo *Goals* preencha com `clean jetty:run`
   - Clique no botão *Apply* e depois em *Run*

3. No navegador acesse *http://localhost:8080* e navegue pelo sistema.

CAPÍTULO 2

TRABALHANDO COM BRANCHES

Depois de liberarmos o código de uma aplicação para o cliente, continuamos melhorando funcionalidades existentes e criando novas funcionalidades. E, claro, comitando frequentemente 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, ainda em desenvolvimento.

Para resolver esse problema, a maioria dos sistemas de controle de versão tem **branches**: linhas independentes de desenvolvimento nas quais podemos trabalhar livremente, versionando quando quisermos e sem atrapalhar outros membros do nosso time.

Em projetos que usam Git, podemos ter tanto branches locais, presentes apenas na sua máquina, quanto branches remotas, que apontam para outras máquinas. Por padrão, a branch principal é chamada **master**, tanto local quanto remotamente. Idealmente, a `master` será uma branch estável, isto é, gostaríamos o máximo possível que essa branch tenha código 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 uma branch chamada **work**, que contém os commits das funcionalidades que ainda estão em desenvolvimento. Para criar a branch `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:

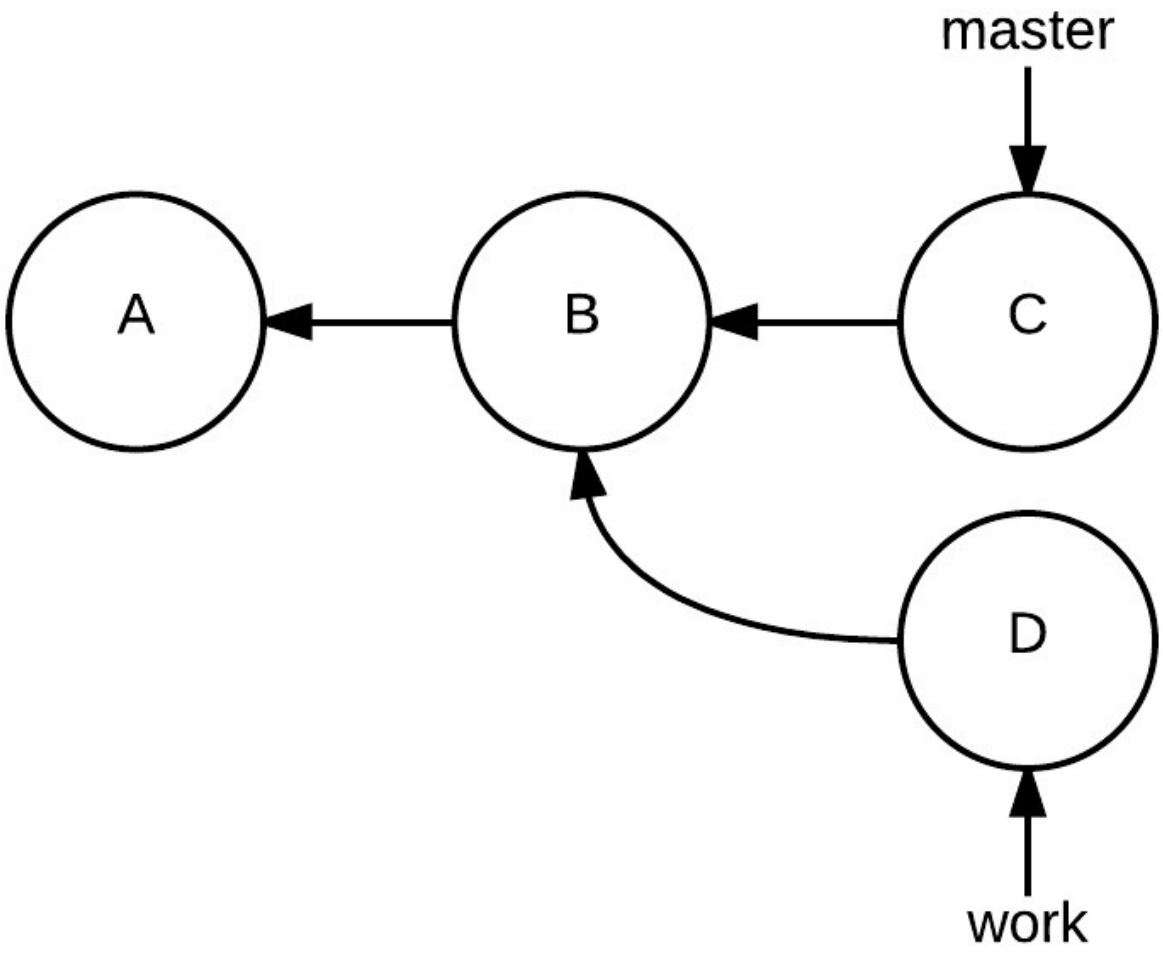
```
git checkout -b work
```

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 uma 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

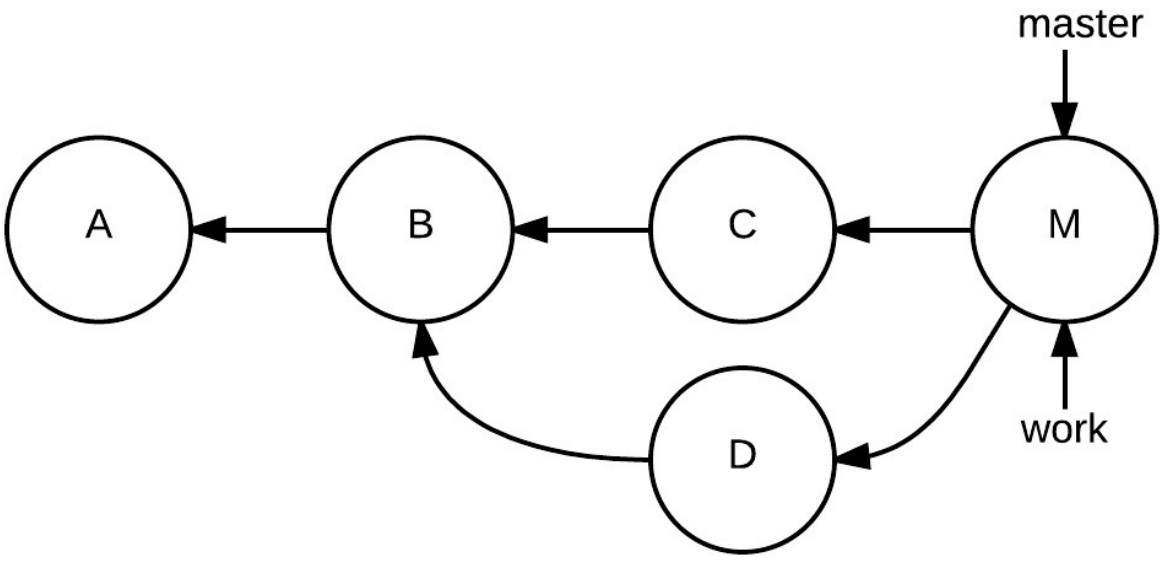
No Git, todo commit tem um ou mais pais, com exceção do primeiro. Para mostrar os commits com seus respectivos pais, podemos utilizar o comando `git log --oneline --parents` .

Se estivermos na branch `master` , podemos fazer o merge das alterações da 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:



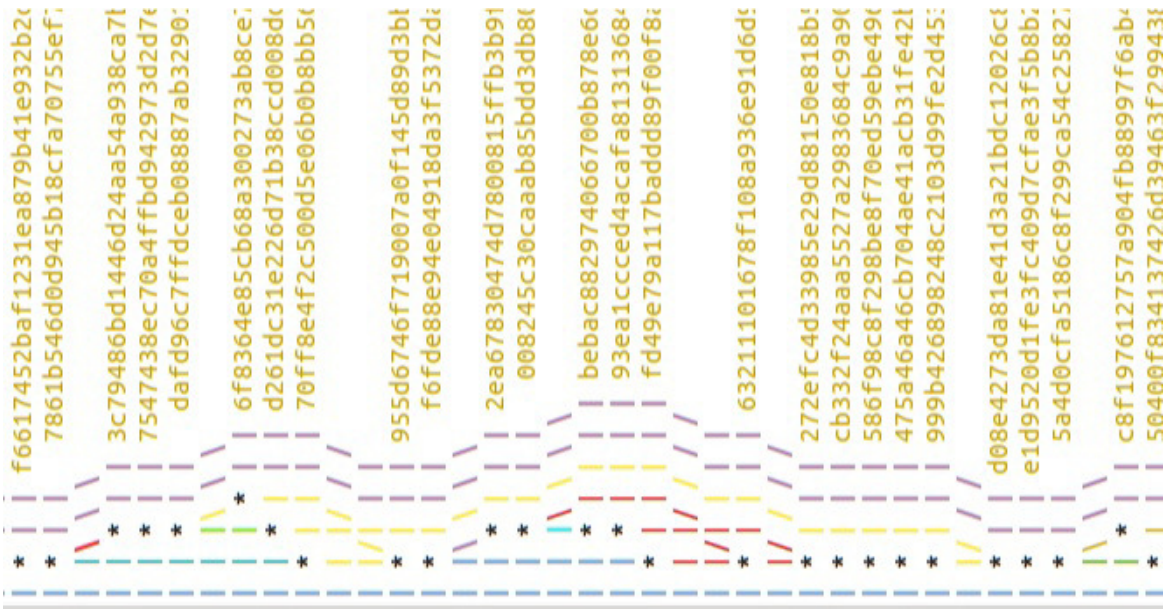
O MERGE DO TIPO FAST-FORWARD

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

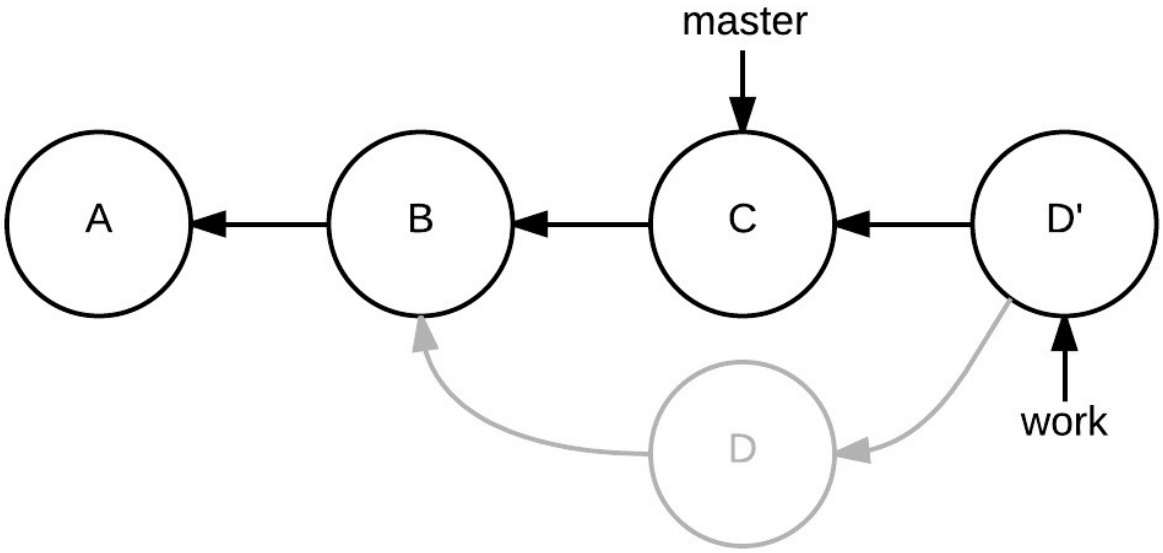
À medida que a aplicação vai sendo desenvolvida, é natural que vários merges sejam feitos, mas se olharmos o histórico de commits da nossa branch `master`, veremos que haverá um commit específico para cada merge feito.

Além de criar um desvio na árvore de commits, esses commits de merges e desvios podem poluir o histórico do projeto.



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.

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 BRANCHS

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 utilize o comando :

```
git checkout work
```

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

5. Commit seu trabalho :

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

6. Como nós já terminamos essa tarefa, podemos fazer com que nosso trabalho entre em produção, para isso faça o rebase.

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

7. 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 começarmos a fazer nosso sistema funcionar da maneira correta, precisamos vender os ingressos, contudo ao irmos no cinema geralmente nós não compramos o ingresso para para o filme, mas sim para uma **Sessão** que possui um filme. E a idéia é que nosso sistema trate os objetos da forma mais próxima do mundo real, então para isso vamos modelar a `Sessão` , que deve possuir o horário, qual é a sua respectiva sala e também qual será o filme :

```
public class Sessao {
```

```
    private LocalTime horario;

    private Sala sala;

    private Filme filme;

    //getters e setters

}
```

Gostaríamos que as `Sessões` sejam armazenadas no banco de dados, para isso precisaremos falar que nosso modelo 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.setFilme(filme);
        this.sala = sala;
    }

    //getters e setters

}
```

Agora que já temos a modelagem pronta, podemos criar algum ponto em nossa aplicação para podermos manipular as sessões, como já vimos no curso FJ-21, a responsabilidade de ligar a *view* ao *model* é feita na camada *controller*, para isso é necessário criarmos nosso controller de sessão :

```
@Controller
public class SessaoController {

}
```

Bom agora precisamos fazer com que nosso controller possa responder algumas chamadas, inicialmente precisamos levar para a tela de cadastro de sessão, para isso criaremos um método que retorne a tela, mas será necessário enviar algumas coisas para a tela, por exemplo a sessão precisa de um

filme, no momento de criação da sessão é interessante mostrarmos todos os filmes em cartaz, além disso também será necessário saber qual é a sala que a sessão está alocada, isso deverá vir para nós atráves da chamada:

```
@Controller
public class SessaoController {

    @GetMapping("/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 quando alguém fizer um requisição do tipo `get` para aquele endereço,este método deverá ser chamado e ainda recebemos um inteiro que chegará via parâmetro, para isso usamos a anotação `@RequestParam("nomeDoParametro)` . Repare que nosso método devolve um objeto do tipo `ModelAndView` , que vai nos ajudar a fazer as tarefas já descritas, além disso na construção deste objeto nós já passamos qual deve ser a página que ele deve retornar.

Além disso vamos precisar enviar algumas informações que temos no código para nossa View, como qual é a sala que estamos manipulando e quais são os filmes disponíveis, essas informações estão todas em nosso banco de dados e para os manipular em nosso sistema estamos usando o padrão de projeto `DAO` , que são gerenciados pelo *Spring*, por isso para conseguirmos pegar esses objetos, sem termos que gerar um alto acoplamento em nossa classe, usaremos o conceito visto no curso FJ-21, **injeção de dependência**, para fazermos a injeção usaremos a anotação `@Autowired` :

```
@Controller
public class SessaoController {

    @Autowired
    private SalaDao salaDao;

    @Autowired
    private FilmeDao filmeDao;

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

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

        return modelAndView;
    }
}
```

Com os objetos já injetados, podemos mandar os dados para a *View* para isso usaremos o método `addObject()` , que recebe uma `String` que representa o id do elemento na página e em seguida passamos o objeto :

```
@Controller
public class SessaoController {

    @Autowired
    private SalaDao salaDao;

    @Autowired
    private FilmeDao filmeDao;

    @GetMapping("/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 temos que ter uma forma de gerenciar nossa tela e poder recuperar os dados que nela estão, que são representados por diversos objetos, para isso vamos criar uma classe para abstrair isso :

```
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;
    }
}
```

Podemos passar esse objeto para a tela também para podermos o recuperar populado no momento de persistirmos os dados do formulario, para isso faremos que o próprio *Spring* passe esse objeto para gente :

```
@Controller
public class SessaoController {
```

```
        @Autowired
        private SalaDao salaDao;

        @Autowired
        private FilmeDao filmeDao;

        @GetMapping("/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;
        }
    }
```

É necessário realizar a persistência, para isso inicialmente teremos que fazer nosso controller escutar que já pode armazenar os dados, para isso criaremos outro método, só que desta vez para podermos deixar tudo feito "por trás dos panos", usaremos outro método http, o `POST` , como usaremos o banco de dados, precisamos deixar o *Spring* ciente sobre a conexão para isso também usaremos uma anotação, `@Transactional` :

```
    @Controller
    public class SessaoController {

        @Autowired
        private SalaDao salaDao;

        @Autowired
        private FilmeDao filmeDao;

        @Autowired
        private SessaoDao sessaoDao;

        //demais métodos

        @PostMapping(value = "/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;
        }

    }
```

Se olharmos bem, colocamos diversas validações em nosso objeto que representa o formulário, então caso não a respeitarmos o sistema não deveria funcionar da maneira que esperamos, não devia sequer ser processado nosso código de persistência. Então precisaremos validar se não ocorreu nenhum erro, caso seja identificado algum, nós retornamos o usuário para o próprio form, para fazermos essa validação, vamos pedir para o *Spring* nos fornecer um objeto do tipo `BindingResult`, através dele será feita a validação :

```
@PostMapping(value = "/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;
}
```

2.4 EXERCÍCIO - CRIANDO NOSSO PRIMEIRO CONTROLLER

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`, faça com que a mesma seja um controller para o *Spring*.

```
@Controller
public class SessaoController {

}
```

3. Crie um método na `SessaoController` para atender a requisições na URI */sessao* para o verbo http *GET*. Esse método deve receber como parâmetro na URI o id da sala, o nome do parâmetro deve ser *salaId*.

```
@Controller
public class SessaoController {

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

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

        return modelAndView;
    }
}
```

4. Crie uma classe de modelo que represente nossa sessão, nela deve ter `id`, `horario` e deve estar

vinculado à uma `Sala` e um `Filme`. Para o horário iremos utilizar o tipo `LocalTime`. Ela deve ser criada no pacote `br.com.caelum.ingresso.model`:

```java
@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.setFilme(filme);
        this.sala = sala;
    }

    public Sala getSala() {
        return sala;
    }

    public void setSala(Sala sala) {
        this.sala = sala;
    }

    public Integer getId() {
        return id;
    }

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

    public LocalTime getHorario() {
        return horario;
    }

    public void setHorario(LocalTime horario) {
        this.horario = horario;
    }

    public Filme getFilme() {
        return filme;
    }

    public void setFilme(Filme filme) {
        this.filme = filme;
```

```
        }

        public LocalTime getHorarioTermino() {
            return this.horario.plus(filme.getDuracao().toMinutes(), ChronoUnit.MINUTES);
        }
    }
```

1. Esse método deve disponibilizar os seguintes objetos na request: A sala baseado no parâmetro *salaId* e a lista com os filmes.

```
@Controller
public class SessaoController {

    @Autowired
    private SalaDao salaDao;

    @Autowired
    private FilmeDao filmeDao;

    @GetMapping("/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;
    }
}
```

2. Como o formulário da sessão usa mais de um modelo, criaremos uma classe que represente o formulário chamada `SessaoForm` no pacote `br.com.caelum.ingresso.model.form` .

```java
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

}
```

1. Precisamos agora disponibilizar nosso `SessaoForm` na request para o formulário e já vamos setar o id da sala para esse form.

```
    @Controller
    public class SessaoController {

        @Autowired
        private SalaDao salaDao;

        @Autowired
        private FilmeDao filmeDao;

        @GetMapping("/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;
        }
    }
```

2. Vamos criar o método que irá receber o *POST* do nosso form e que deverá salvar uma sessão. Para isso precisaremos criar um método no `SessaoForm` que retorne um objeto `Sessao`.

```java
  @Controller
  public class SessaoController {

      @Autowired
      private SalaDao salaDao;

      @Autowired
      private FilmeDao filmeDao;

      @Autowired
      private SessaoDao sessaoDao;

      //demais métodos

      @PostMapping(value = "/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()+"/sessoe
s");

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

          sessaoDao.save(sessao);

          return modelAndView;
      }
  }

  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;
            }
        }
```

1. Vamos criar o DAO de sessão:

```java
@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();
    }
}
```

1. Junte o que foi feito na sua branch work para a branch master, através do rebase.

2.5 ANALISANDO A REGRA DE NEGÓCIO

Implementamos a funcionalidade de adicionar a sessão na sala, contudo pensando no mundo real nós não podemos simplesmente adicionar, é necessário verificar se podemos de fato fazer a inserção da sessão, para que não aja conflito com as demais sessões existentes.

Para isso podemos criar uma classe especialista para verificar isso para gente, vamos chama-la de

`GerenciadorDeSessao` , ela terá que 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 todas as sessões que acontecerão naquela sala, para isso pediremos a lista como parâmetro em nosso 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 comparando o horário da sessão que estamos tentando adicionar não conflita nos seguintes pontos :

- Começa antes de uma sessão, contudo a duração do filme da sessão que estamos tentando adicionar deve acabar antes da próxima sessão
- Começa depois de uma sessão, contudo a duração do filme da sessão que já está passando deve acabar antes da sessão que estamos tentando adicionar

Estamos usando a API de data do Java 8, que foi baseada na biblioteca **Joda Time** que 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) {
```

```
        LocalTime horarioSessao = sessaoExistente.getHorario();
        LocalTime horarioAtual = sessaoAtual.getHorario();

        boolean ehAntes = horarioAtual.isBefore(horarioSessao);

        if (ehAntes) {

            return horarioAtual.plusMinutes(sessaoAtual.getFilme().getDuracao().toMinutes())
                    .isBefore(sessaoExistente.getHorario());
        } else {

            return sessaoExistente.getHorarioTermino().isBefore(horarioAtual);

        }
    }
```

Nosso código, está validando todas as possiblidades que listamos ainda a pouco, contudo como vimos estamos usando recursos do Java 8 e se vamos usar alguns, seria interessante usarmos mais alguns por exemplo, invés de fazermos esse for e if, podemos usar outro recurso bem interesante, o 'Stream', que será o fluxo dos nossos dados :

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

        // restante do código
    }
```

Agora queremos verificar se o horário é válido e a resposta disso será um boolean, então se tiver em algum momento um horário inválido ele não deve gravar, para conseguirmos pegar cada objeto e retornar um boolean, usaremos um especialista nisso, o método `map()` do `Stream` , que devolverá um `Stream` convertido no tipo que desejamos :

```
    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 `Boolean` s, temos que fazer algo uma comparação entre todos eles, utilizando o comparador `&` , ficaria algo dessa forma :

```
if (boolean1 && boolean2 && ... && boleanX){
    return true;

}

return false;
```

Repare que em ambas os casos, nós acabamos reduzindo tudo apenas para um `boolean` , contudo

gerar esse código pode ser um pouco trabalhoso, teremos que percorrer novamente `Stream` , contudo esse objeto é um pouco mais inteligente e já sabe fazer isso para nós, existe outro método que nós ajudará com esse serviço, o `reduce()` que fará basicamente isto por nós, precisamos apenas falar qual é o tipo do objeto a ser retornado e qual é a estrategia que deve ser executada para isso :

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

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

    booleanStream.reduce(Boolean::logicalAnd)

    // restante do código
}
```

Falamos que será devolvido um `Boolean` e faremos verificação entre os dois utilizando o operador lógico &&, contudo o que vai acontecer se nossa lista estiver vazia ? O nada disso acontecerá e pior, ainda levaríamos uma exception, uma solução bem conhecida por nós desenvolvedores é fazer uma validação na lista, para nos precavermos, entretanto no Java 8 há uma forma mais elegante de resolvermos 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 -> horarioIsValido(sessaoExistente, sessaoAtual));

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

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

Ao utilizarmos o método `orElse` , o que está acontecendo por trás dos panos é a validação se existe qualquer valor, caso não exista vamos devolver um valor padrão, que no nosso caso é um 'true'.

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 (no caso de ser 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) é um framework muito simples para facilitar 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 junto com nosso programa? (dependência)

É aqui que o **Classpath** entra 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 (veremos mais a frente);
- Deixando o eclipse configurar por você.

No Eclipse, é muito simples:

- Clique com o botão direito em cima do nome do seu projeto.
- Escolha a opção *Properties*.
- Na parte esquerda da tela, selecione a opção "*Java Build Path*".

E, nessa tela:

- "*Java Build Path*" é onde você configura o *classpath* do seu projeto: lista de locais definidos que, por padrão, só vêm com a máquina virtual configurada;
- Opções para adicionar mais caminhos, "Add JARs..." adiciona Jar's que estejam no seu projeto; "Add External JARs" adiciona Jar's que estejam em qualquer outro lugar da máquina, porém guardará uma referência para aquele caminho (então seu projeto poderá não funcionar corretamente quando colocado em outro micro, mas existe como utilizar variáveis para isso);

No caso do JUnit, por existir integração direta com Eclipse, o processo é ainda mais fácil, como veremos no exercício. Mas para todas as outras bibliotecas que formos usar, basta copiar o JAR e adicioná-lo ao *Build Path*.

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, poderia acarretar a empresa um grande prejuízo. Para evitar essa situação, vamos entregar um software com mais qualidade, onde tudo esteja funcionando corretamente, 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 pasta que a classe que estamos testando, contudo ficará no *source folder* de teste, nosso projeto :

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

```
public class GerenciadorDeSessaoTest {


}
```

Com nossa classe de teste já definida, basta criarmos nossos testes, para isso precisaremos definir quais serão os cenários que vamos cubrir, por exemplo :

- Não sermos capazes de adicionar uma sessão no mesmo horário de começo de outra

- Não podermos adicionar uma sessão que a duração conflite com o horário da próxima sessão
- Não podermos adicionar uma sessão com inicio 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 poder adicionar se a lista estiver vazia. Para isso precisaremos criar este cenário, para isso vamos criar um método em nossa classe :

```
public class GerenciadorDeSessaoTest {

    public void deveAdicionarSeListaDaSessaoEstiverVazia(){


    }

}
```

A primeira coisa que temos que falar para o JUnit é que esse nosso método `deveAdicionarSeListaDaSessaoEstiverVazia` , deve ser interpretado como um teste, para isso usaremos a annotation `@Test` :

```
public class GerenciadorDeSessaoTest {

    @Test
    public void deveAdicionarSeListaDaSessaoEstiverVazia(){


    }

}
```

Agora será necessário criar o **cenário** para que o nosso teste seja executado, para isso precisaremos 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);



    }

}
```

Agora precisamos 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();
        filme.setDuracao(120);
        LocalTime horario = LocalTime.now();
        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, para isso faremos a parte mais importante do teste : **Verificações**. Para isso usaremos a classe `Assert` que é uma especialista nisto, sabemos que o resultado do nosso teste deve ser `true` , vamos pedir para o `Assert` verificar isso para gente :

```
public class GerenciadorDeSessaoTest {

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

        Filme filme = new Filme();
        filme.setDuracao(120);
        LocalTime horario = LocalTime.now();
        Sala sala = new Sala("");

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

        boolean cabe = gerenciador.cabe(sessao);

        Assert.assertTrue(cabe);
    }

}
```

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

1. Precisamos verificar se o horário da sessão que estamos gravando não irá encavalar com o horário de uma sessão já existente. Para isso, vamos criar uma classe que faça essa validação.

   - Crie a classe `GerenciadorDeSessoes` 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.

     ```java



```

```
public class GerenciadorDeSessao {

    private List<Sessao> sessoesDaSala;

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

      public boolean cabe(final Sessao sessaoAtual) {

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

          return optionalCabe.orElse(true);
      }


      private boolean horarioIsValido(Sessao sessaoExistente, Sessao sessaoAtual) {

          LocalTime horarioSessao = sessaoExistente.getHorario();
          LocalTime horarioAtual = sessaoAtual.getHorario();

          boolean ehAntes = horarioAtual.isBefore(horarioSessao);

          if (ehAntes) {

              return horarioAtual.plusMinutes(sessaoAtual.getFilme().getDuracao().toMinutes())
                      .isBefore(sessaoExistente.getHorario());
          } else {

              return sessaoExistente.getHorarioTermino().isBefore(horarioAtual);

          }
      }

  }
```

1. Precisamos garantir que a validação que acabamos de implementar está funcionando. Para isso, criaremos um teste unitário que vai validar os cenários que previmos.

   - Crie a classe `GerenciadorDeSessaoTest` no pacote `br.com.caelum.ingresso.validacao` porém no *source folder* de teste (*src/test/java*).

     ```java public class GerenciadorDeSessaoTest {

     @Test
     public void garanteQueNaoDevePermitirSessaoNoMesmoHorario() {

         Filme filme = new Filme();
         filme.setDuracao(120);
         LocalTime horario = LocalTime.now();

         Sala sala = new Sala("");
         List<Sessao> sessoes = Arrays.asList(new Sessao(horario, filme, sala));
```

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

        GerenciadorDeSessao gerenciador = new GerenciadorDeSessao(sessoes);

        Assert.assertFalse(gerenciador.cabe(sessao));
    }

    @Test
    public void garanteQueNaoDevePermitirSessoesTerminandoDentroDoHorarioDeUmaSessaoJaExistente() {
        Filme filme = new Filme();
        filme.setDuracao(120);
        LocalTime horario = LocalTime.now();

        Sala sala = new Sala("");
        List<Sessao> sessoes = Arrays.asList(new Sessao(horario, filme, sala));

        Sessao sessao = new Sessao(horario.plusHours(1), filme, sala);
        GerenciadorDeSessao gerenciador = new GerenciadorDeSessao(sessoes);

        Assert.assertFalse(gerenciador.cabe(sessao));

    }

    @Test
    public void garanteQueNaoDevePermitirSessoesIniciandoDentroDoHorarioDeUmaSessaoJaExistente() {
        Filme filme = new Filme();
        filme.setDuracao(120);
        LocalTime horario = LocalTime.now();
        Sala sala = new Sala("");

          List<Sessao> sessoesDaSala = Arrays.asList(new Sessao(horario, filme, sala));

          GerenciadorDeSessao gerenciador = new GerenciadorDeSessao(sessoesDaSala);
          Assert.assertFalse(gerenciador.cabe(new Sessao(horario.plus(1,
                                        ChronoUnit.HOURS), filme, sala)));

      }

      @Test
      public void garanteQueDevePermitirUmaInsercaoEntreDoisFilmes() {
          Sala sala = new Sala("");

          Filme filme1 = new Filme();
          filme1.setDuracao(90);
          LocalTime dezHoras = LocalTime.parse("10:00:00");
          Sessao sessaoDasDez = new Sessao(dezHoras, filme1, sala);

          Filme filme2 = new Filme();
          filme2.setDuracao(120);
          LocalTime dezoitoHoras = LocalTime.parse("18:00:00");
          Sessao sessaoDasDezoito = new Sessao(dezoitoHoras, filme2, sala);

          List<Sessao> sessoes = Arrays.asList(sessaoDasDez, sessaoDasDezoito);

          GerenciadorDeSessao gerenciador = new GerenciadorDeSessao(sessoes);

          Assert.assertTrue(gerenciador.cabe(new Sessao(LocalTime.parse("13:00:00"), filme2, sala))
);

      }
  }
  ```
```
```

1. Depois de termos testado nosso `GerenciadorDeSessao` e nos certificarmos que ele está pronto para uso, podemos usá-lo em nosso controller:

```java @Controller public class SessaoController {

// restante do código

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

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

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

    List<Sessao> sessoesDaSala = sessaoDao.buscaSessoesDaSala(sessao.getSala());

    GerenciadorDeSessao gerenciador = new GerenciadorDeSessao(sessoesDaSala);

    if (gerenciador.cabe(sessao)) {
        sessaoDao.save(sessao);
        return new ModelAndView("redirect:/sala/" + form.getSalaId() + "/sessoes");
    }

    return form(form.getSalaId(), form);
}

}
```

Capítulo 3

ADICIONANDO PREÇO

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. Sendo que filmes que tendem a ter mais audiência devem ter um preço superior, logo precisaremos ter um atributo em nosso modelo `Filme`, mas qual será o tipo do preço ?

```
@Entity
public class Filme {

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

Podíamos facilmente colocar como double, contudo existem alguns problemas nisso, o principal dele é 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 é que não podemos mais deixar nenhum lugar que estamos manipulando o filme estar sem preço, para isso teremos que forçar o `Filme` a ter preço, portanto essa informação será atribuida no construtor :

```
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 na tanto na listagem quanto no formulário 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">
        <span class="text-danger">${error.defaultMessage}</span>
    </c:forEach>
</div>

<!-- botão de gravar -->
```

Além disso temos outra regra de negócio muito importante, cada sala deve possuir seu preço, já que podemos ter salas onde há um diferencial como ser 3D, iMax, fora a parte de manutenção. Para isso 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 que a sessão tenha seu preço correto, que deve ser a soma da sala com 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;
    }
```

3.1 EXERCÍCIO - COLOCANDO PREÇO NA SALA E FILME

1. Adicione um atributo preço do tipo `BigDecimal` nas classes `Sala` e `Filme` . Caso elas ainda não tenham construtores com todos os atributos, crie nas duas classes. (Lembre-se de manter um

construtor sem argumentos depreciado. Ele será usado somente por frameworks como *Hibernate*, *Spring* e etc...)

```java
@Entity
public class Sala {

    @Id
    @GeneratedValue
    private Integer id;

    @NotBlank
    private String nome;

    @OneToMany(fetch = FetchType.EAGER)
    private List<Lugar> lugares = new ArrayList<>();

    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
}

@Entity
public class Filme {

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

1. Ao adicionar o novo parâmetro ao construtor (ou o próprio construtor com todos os parâmetros), quebramos nossos testes da classe `GerenciadorDeSessaoTest`. Vamos passar todos os argumentos no momento de instanciar `Sala` e `Filme`:

```java
public class GerenciadorDeSessaoTest {

    @Test
    public void garanteQueNaoDevePermitirSessaoNoMesmoHorario() {

        Filme filme = new Filme("Rogue One", Duration.ofMinutes(120), "SCI-FI", BigDecimal.ONE);
        filme.setDuracao(120);
        LocalTime horario = LocalTime.now();

        Sala sala = new Sala("Eldorado - IMAX", BigDecimal.ONE);
        List<Sessao> sessoes = Arrays.asList(new Sessao(horario, filme, sala));

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

        GerenciadorDeSessao gerenciador = new GerenciadorDeSessao(sessoes);

        Assert.assertFalse(gerenciador.cabe(sessao));
    }

    @Test
    public void garanteQueNaoDevePermitirSessoesTerminandoDentroDoHorarioDeUmaSessaoJaExistente()
{
        Filme filme = new Filme("Rogue One", Duration.ofMinutes(120), "SCI-FI", BigDecimal.ONE);
        filme.setDuracao(120);
        LocalTime horario = LocalTime.now();

        Sala sala = new Sala("Eldorado - IMAX", BigDecimal.ONE);
        List<Sessao> sessoes = Arrays.asList(new Sessao(horario, filme, sala));

        Sessao sessao = new Sessao(horario.plusHours(1), filme, sala);
        GerenciadorDeSessao gerenciador = new GerenciadorDeSessao(sessoes);

        Assert.assertFalse(gerenciador.cabe(sessao));

    }

    @Test
    public void garanteQueNaoDevePermitirSessoesIniciandoDentroDoHorarioDeUmaSessaoJaExistente()
{
        Filme filme = new Filme("Rogue One", Duration.ofMinutes(120), "SCI-FI", BigDecimal.ONE);
        filme.setDuracao(120);
        LocalTime horario = LocalTime.now();
        Sala sala = new Sala("Eldorado - IMAX", BigDecimal.ONE);

        List<Sessao> sessoesDaSala = Arrays.asList(new Sessao(horario, filme, sala));

        GerenciadorDeSessao gerenciador = new GerenciadorDeSessao(sessoesDaSala);
        Assert.assertFalse(gerenciador.cabe(new Sessao(horario.plus(1, ChronoUnit.HOURS), filme,
sala)));

    }

    @Test
    public void garanteQueDevePermitirUmaInsercaoEntreDoisFilmes() {
        Sala sala = new Sala("Eldorado - IMAX", BigDecimal.ONE);
```

```
            Filme filme1 = new Filme("Rogue One", Duration.ofMinutes(120), "SCI-FI", BigDecimal.ONE);
            filme1.setDuracao(90);
            LocalTime dezHoras = LocalTime.parse("10:00:00");
            Sessao sessaoDasDez = new Sessao(dezHoras, filme1, sala);

            Filme filme2 = new Filme("Rogue One", Duration.ofMinutes(120), "SCI-FI", BigDecimal.ONE);
            filme2.setDuracao(120);
            LocalTime dezoitoHoras = LocalTime.parse("18:00:00");
            Sessao sessaoDasDezoito = new Sessao(dezoitoHoras, filme2, sala);

            List<Sessao> sessoes = Arrays.asList(sessaoDasDez, sessaoDasDezoito);

            GerenciadorDeSessao gerenciador = new GerenciadorDeSessao(sessoes);

            Assert.assertTrue(gerenciador.cabe(new Sessao(LocalTime.parse("13:00:00"), filme2, sala))
);
        }

    }
```

1. Vamos alterar as páginas de formulário e listagem da `Sala` e do `Filme` para adicionar o campo de preço:

   - No arquivo `src/main/webapp/WEB-INF/views/sala/sala.jsp` , adicione um novo input para o preço:

     ```html <%@ 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" %>

     ```
       <jsp:body>
           <div class=" col-md-6 col-md-offset-3">
           <c:set var="bindingResult" value="${requestScope['org.springframework.validation.BindingR
       esult.sala']}"/>

           <form action='/sala' method="post">
               <div class="form-group">
                   <input type="hidden" name="id" value="${sala.id}">

                   <div class="form-group">
                       <label for="nome">Nome:</label>
                       <input id="nome" type="text" name="nome" class="form-control" value="${sala.n
ome}">

                       <c:forEach items="${bindingResult.getFieldErrors('nome')}" var="error">
                           <span class="text-danger">${error.defaultMessage}</span>
                       </c:forEach>
                   </div>

                   <div class="form-group">
                       <label for="preco">Preço:</label>
                       <input id="preco" type="text" name="preco" class="form-control" value="${sala
.preco}">
     ```

```
                        <c:forEach items="${bindingResult.getFieldErrors('preco')}" var="error">
                            <span class="text-danger">${error.defaultMessage}</span>
                        </c:forEach>
                    </div>
                </div>

                <button type="submit" class="btn btn-primary">Gravar</button>
            </form>
            </div>
        </jsp:body>
    </ingresso:template>
```

* No arquivo `src/main/webapp/WEB-INF/views/sala/lista.jsp`, adicione uma coluna para o preço:

```html
    <%@ 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 uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
    <ingresso:template>
        <jsp:body>
            <div class=" col-md-6 col-md-offset-3">
            <table class="table table-hover ">
                <thead>
                <tr>
                    <th class="text-center">Nome</th>
                    <th class="text-center">Preço</th>
                    <th colspan="4" class="text-center">Ações</th>
                </thead>
                <tbody>
                <c:forEach var="sala" items="${salas}">
                    <tr>
                        <td class="text-center">${sala.nome}</td>
                        <td class="text-center">${sala.preco}</td>
                        <td class="col-md-1">
                            <a href="/sala/${sala.id}/sessoes/" class="btn btn-primary">
                                <span class="glyphicon glyphicon-blackboard" aria-hidden="true"></spa
n> Sessões
                            </a>
                        </td>
                        <td class="col-md-1">
                            <a href="/sala/${sala.id}/lugares/" class="btn btn-warning">
                                <span class="glyphicon glyphicon-th" aria-hidden="true"></span> Lugar
es
                            </a>
                        </td>
                        <td>
                            <a onclick="excluir(${sala.id})" class="btn btn-danger">Excluir</a>
                        </td>
                        <td>
                            <a href="/sala/${sala.id}" class="btn btn-info">Alterar</a>
                        </td>
                    </tr>
                </c:forEach>
                </tbody>
            </table>
            <div class="col-md-6 col-md-offset-3">
                <a href="/sala" class="btn btn-block btn-info">Novo</a>
            </div>
            </div>
```

```
            <script>
                function excluir(id) {
                    var url = window.location.href;
                    $.ajax({
                        url: "/sala/" + id,
                        type: 'DELETE',
                        success: function (result) {
                            console.log(result);

                            window.location.href = url;
                        }
                    });
                }
            </script>
        </jsp:body>
    </ingresso:template>
```

* No arquivo `src/main/webapp/WEB-INF/views/filme/filme.jsp`, adicione um novo input para o preço:

```html
    <%@ 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 uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

    <ingresso:template>
        <jsp:body>
            <div class=" col-md-6 col-md-offset-3">
            <c:set var="bindingResult" value="${requestScope['org.springframework.validation.BindingR
esult.filme']}"/>

            <form action='/filme' method="post">
                <input type="hidden" name="id" value="${filme.id}">

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

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

                <div class="form-group">
                    <label for="duracao">Duracao:</label>
                    <input id="duracao" type="text" name="duracao" class="form-control"
                        value="${filme.duracao.toMinutes()}">
                    <c:forEach items="${bindingResult.getFieldErrors('duracao')}" var="error">
                        <span class="text-danger">${error.defaultMessage}</span>
                    </c:forEach>
                </div>
```

```
                <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">
                        <span class="text-danger">${error.defaultMessage}</span>
                    </c:forEach>
                </div>

                <button type="submit" class="btn btn-primary">Gravar</button>
            </form>
            </div>
        </jsp:body>
    </ingresso:template>
```

* No arquivo `src/main/webapp/WEB-INF/views/filme/lista.jsp`, adicione uma coluna para o preço:

```html
    <%@ 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 uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>


    <ingresso:template>
        <jsp:body>
            <div class=" col-md-6 col-md-offset-3">
            <table class="table table-hover ">
                <thead>
                    <tr>
                        <th>Nome</th>
                        <th>Duração</th>
                        <th>Preço</th>
                        <th colspan="2" class="text-center">Ações</th>
                    </tr>
                </thead>
                <tbody>
                    <c:forEach var="filme" items="${filmes}">
                        <tr>
                            <td>${filme.nome}</td>
                            <td>${filme.duracao.toMinutes()}</td>
                            <td>${filme.preco}</td>

                            <td>
                                <a onclick="excluir(${filme.id})" class="btn btn-danger">Excluir</a>
                            </td>
                        </tr>
                    </c:forEach>
                </tbody>
            </table>
            <div class="col-md-6 col-md-offset-3">
                <a href="/filme" class="btn btn-block btn-info">Novo</a>
            </div>
            </div>
            <script>
                function excluir(id) {
                    var url = window.location.href;
                    $.ajax({
                        url:"/filme/" + id,
                        type: 'DELETE',
```

```
                    success: function (result) {
                        console.log(result);

                        window.location.href = url;
                    }
                });
            }
        </script>
    </jsp:body>
</ingresso:template>
```

1. Faça com que a classe `Sessao` atribua o resultado da soma dos preços da `Sala` e do `Filme` ao atributo `preco`:

```java
@Entity
public class Sessao {

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

    @ManyToOne
    private Sala sala;

    @ManyToOne
    private Filme filme;

    private BigDecimal preco;

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

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

    // demais getters e setters

    public BigDecimal getPreco() {
        return preco;
    }
}
```

1. Crie um teste 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 BigDecima
l("12.0"));

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

            Sessao sessao = new Sessao(LocalTime.now(), filme, sala);

            assertEquals( somaDosPrecosDaSalaEFilme, sessao.getPreco() );

        }

    }
```

2. Altere o arquivo de listagem da sessão `src/main/webapp/WEB-INF/views/sessao/lista.jsp` para exibir o preço:

```
<%@ 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 uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<ingresso:template>
    <jsp:body>
        <div class=" col-md-6 col-md-offset-3">
        <h3>Sessões na sala: ${sala.nome}</h3>

        <table class="table table-hover ">
            <thead>
            <tr>
                <th>Nome</th>
                <th>Filme</th>
                <th>Duração</th>
                <th>Preço</th>
                <th colspan="2" class="text-center">Ações</th>
            </tr>
            </thead>
            <tbody>
            <c:forEach var="sessao" items="${sessoes}">
                <tr>
                    <td>${sessao.horario}</td>
                    <td>${sessao.filme.nome}</td>
                    <td>${sessao.filme.duracao.toMinutes()}</td>
                    <td>${sessao.preco}</td>
                    <td>
                        <a onclick="excluir(${sessao.id})" class="btn btn-danger">Excluir</a>
                    </td>
                </tr>
            </c:forEach>
            </tbody>
        </table>
        <div class="col-md-6 col-md-offset-3">
            <a href="/sessao?salaId=${sala.id}" class="btn btn-block btn-info">Nova</a>
        </div>
        </div>
        <script>
            function excluir(id) {
                var url = window.location.href;
                $.ajax({
                    url:"/sessao/" + id,
                    type: 'DELETE',
```

```
                    success: function (result) {
                        console.log(result);

                        window.location.href = url;
                    }
                });
            }
        </script>
    </jsp:body>
</ingresso:template>
```

3.2 APLICANDO 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:

```
public class Ingresso {


}
```

Agora é necessário definir quais serão os atributos que nosso `Ingresso` vai possuir. Nem precisamos pensar muito para responder que será preciso ao menos a própria `Sessão` e também do preço:

```
public class Ingresso {

    private Sessao sessao;
    private BigDecimal preco;

}
```

Agora precisamos pensar um pouco pois, 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 possue 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 DescontoEstudante implements Desconto {

    private BigDecimal metade = new BigDecimal(2.0);
```

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

Desta maneira, para nosso ingresso pode usar o `Desconto` para calcular o preço, assim que construirmos nosso objeto `Ingresso`, devemos informar qual é o `Desconto` utilizado:

```
public class Ingresso {

    private Sessao sessao;
    private BigDecimal preco;

    public Ingresso(Sessao sessao, Desconto desconto) {
        this.sessao = sessao;
        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 que resolvemos esse impasse, usar **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**.

3.3 EXERCÍCIO - CRIANDO DESCONTOS E INGRESSO

1. Crie a interface `Desconto` no pacote `br.com.caelum.ingresso.model.descontos`:

```
public interface Desconto {

    BigDecimal aplicarDescontoSobre(BigDecimal precoOriginal);

}
```

2. Crie três implementações de desconto: `DescontoEstudante`, `DescontoTrintaPorcentoParaBancos` e `SemDesconto`:

```
public class DescontoEstudante  implements Desconto {

    private BigDecimal metade = new BigDecimal("2.0");

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

```
public class DescontoDeTrintaPorCentoParaBancos implements Desconto {

    private BigDecimal percentualDeDesconto = new BigDecimal("0.3");

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

    private BigDecimal trintaPorCentoSobre(BigDecimal precoOriginal) {
        return precoOriginal.multiply(percentualDeDesconto);
    }
}

public class SemDesconto implements Desconto {

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

3. Crie a classe `Ingresso` no pacote `br.com.caelum.ingresso.model` com atributos `Sessao` e `Desconto`:

```
public class Ingresso {

    private Sessao sessao;
    private BigDecimal preco;

    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;
    }
}
```

4. Crie uma classe de teste para `Desconto`, garantindo que os três descontos são aplicados corretamente aos ingressos:

```java
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("1
2"));
        Sessao sessao = new Sessao(LocalTime.now(), filme, sala);
        Ingresso ingresso = new Ingresso(sessao, new DescontoDeTrintaPorCentoParaBancos());
```

```
            BigDecimal precoEsperado = new BigDecimal("22.75");

            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("1
2"));
            Sessao sessao = new Sessao(LocalTime.now(), filme, sala);
            Ingresso ingresso = new Ingresso(sessao, new DescontoEstudante());

            BigDecimal precoEsperado = new BigDecimal("16.25");


            assertEquals(precoEsperado, ingresso.getPreco());

        }

        @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("1
2"));
            Sessao sessao = new Sessao(LocalTime.now(), filme, sala);
            Ingresso ingresso = new Ingresso(sessao, new SemDesconto());


            BigDecimal precoEsperado = new BigDecimal("32.5");


            assertEquals(precoEsperado, ingresso.getPreco());


        }

    }
```

Capítulo 4

MELHORANDO A USABILIDADE DA APLICAÇÃO

Precisamos deixar nosso catálogos de filmes mais atrativo para o cliente final, algo como isso :

Para isso teremos que mexer um pouco mais com o bootstrap, agora cada item precisa ser pequeno card, para isso precisamos fazer essa definição :

```
<c:forEach var="filme" items="${filmes}">

    <div class="col-md-4 ">
        <a href="/filme/${filme.id}/detalhe">
            <div class="panel panel-default">
                <div class="panel-heading text-center"><strong>${filme.nome}</strong></div>
                <div class="panel-body">
                    <div>
                        <strong>Genero:</strong> ${filme.genero}
                    </div>
                    <div>
                        <strong>Duração:</strong> ${filme.duracao.toMinutes()} minutos
                    </div>
                </div>
            </div>
        </a>
    </div>
```

```
</c:forEach>
```

Agora precisamos definir o estilo que usaremos para o deixar desta forma :

```
<style>
    a:hover {
        text-decoration: none;
    }
    .panel{
        transition:         transform 0.7s;
    }
    .panel:hover{
        transform:          translateY(-0.5em);
    }
    .panelSize {
        min-height: 10.5em;
        min-width: 13em;
    }
</style>
```

Para usarmos esse estilo basta fazermos a importação nele dentro do card :

```
<div class="panel panel-default panelSize">
    .
    .
    .
</div>
```

4.1 EXERCÍCIO - CRIANDO TELA PARA LISTAGEM DOS FILMES

1. Crie uma tela com uma listagem de paineis do bootstrap. Cada *panel* deve ter o cabeçalho (*panel-heading*) com o nome do filme e no corpo (*panel-body*) deve ter o gênero e a duração do filme.

```
 <%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
https://cursos.alura.com.br/forum/topico-aula5-37997    <%@ 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">
             <c:forEach var="filme" items="${filmes}">

                     <div class="col-md-4 ">
                         <a href="/filme/${filme.id}/detalhe">
                             <div class="panel panel-default">
                                 <div class="panel-heading text-center"><strong>${filme.nome}</str
ong></div>
                                 <div class="panel-body">
                                     <div>
                                         <strong>Genero:</strong> ${filme.genero}
                                     </div>
                                     <div>
                                         <strong>Duração:</strong> ${filme.duracao.toMinutes()} mi
nutos
                                     </div>
                                 </div>
                             </div>
```

```
                    </a>
                </div>

            </c:forEach>
        </div>
    </jsp:body>
</ingresso:template>
```

2. Vamos melhorar o layout do nosso *panel*:

3. Crie a tag `<style>` antes de fechar a tag `</jsp:body>` :

```
<jsp:body>
    <div class=" col-md-6 col-md-offset-3">
        .
        .
        .
    </div>
    <style>
    </style>
</jsp:body>
```

4. Dentro da tag `<style>` remova a decoração dos links ao passar o mouse por cima deles:

```
<style>
    a:hover {
        text-decoration: none;
    }
</style>
```

5. Vamos adicionar um efeito, ao passar o mouse por cima vamos fazer com que seja deslocado o *panel* um pouco para cima:

```
<style>
    a:hover {
        text-decoration: none;
    }
    .panel{
        transition:         transform 0.7s;
    }

    .panel:hover{
        transform:             translateY(-0.5em);
    }
</style>
```

6. Como o `bootstrap` usa `float` para suas colunas. Precisamos fixar o tamanho dos nossos *panels*, do contrário ao ultrapassar o espaço horizontal disponivel, o `float` pode inserir o novo *panel* em um local não apropriado.

```
<style>
    a:hover {
        text-decoration: none;
    }
    .panel{
        transition:         transform 0.7s;
    }
    .panel:hover{
        transform:             translateY(-0.5em);
```

```
        }
        .panelSize {
            min-height: 10.5em;
            min-width: 13em;
        }
    </style>
```

7. Agora vamos adicionar essa classe ao nosso `panel` :

```
    <div class="panel panel-default panelSize">
        .
        .
        .
    </div>
```

4.2 EXERCÍCIO - CRIANDO CONTROLLER PARA LISTAGEM DE FILMES PARA COMPRA DE INGRESSO

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 `template.tag` para que no menu tenha um link para acesso à `/filme/em-cartaz` :

```
<div class="collapse navbar-collapse"
        id="bs-example-navbar-collapse-1">
    <ul class="nav navbar-nav navbar-right">
        <li><a href="/filmes">Filmes</a></li>
        <li><a href="/salas">Salas</a></li>
        <li><a href="/filme/em-cartaz">Comprar</a></li>
    </ul>
</div>
```

4.3 DEFININDO A TELA DE DETALHES

Naturalmente o usuário ao ver nosso catálogo de filmes irá clicar em uma das opções, ele espera 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 que isso seja possível vamos criar essa nova tela dentro do nosso sistema, que vai apresentar os detalhes da tela. Que basicamente serão vários labels :

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

    <div>
```

```
        <label for="ano">Ano</label>
        <span id="ano"></span>
    </div>

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

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

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

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

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

    <table class="table table-hover">
        <thead>
            <th>Sala</th>
            <th>Horario</th>
            <th>Ações</th>
        </thead>
        <tbody>
            <c:forEach items="${sessoes}" var="sessao">
                <tr>
                    <td>${sessao.sala.nome}</td>
                    <td>${sessao.horario}</td>
                    <td>
                        <a href="/sessao/${sessao.id}/lugares" class="btn">
                            Comprar
                            <span class="glyphicon glyphicon-blackboard" aria-hidden="true"></span>
                        </a>
                    </td>
                </tr>
            </c:forEach>
        </tbody>
    </table>
</div>
```

Agora precisamos fazer que nosso controller envie os dados 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;
}
```

4.4 EXERCÍCIO - CRIANDO TELA DE DETALHES DO FILME E SESSÕES DESSE FILME PARA COMPRA

1. vamos criar uma tela que irá exibir os detalhes do filme, escolhido e as sessões disponíveis:

```
<%@ 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" %>
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>

<ingresso:template>
    <jsp:body>
        <div class=" col-md-6 col-md-offset-3">
            <h1>Titulo</h1>
            <image src="" />

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

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

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

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

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

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

            <table class="table table-hover">
                <thead>
                    <th>Sala</th>
                    <th>Horario</th>
                    <th>Ações</th>
                </thead>
                <tbody>
```

```
                    <c:forEach items="${sessoes}" var="sessao">
                        <tr>
                            <td>${sessao.sala.nome}</td>
                            <td>${sessao.horario}</td>
                            <td>
                                <a href="/sessao/${sessao.id}/lugares" class="btn">
                                    Comprar
                                    <span class="glyphicon glyphicon-blackboard" aria-hidden="tru
e"></span>
                                </a>
                            </td>
                        </tr>
                    </c:forEach>
                </tbody>
            </table>
        </div>
    </jsp:body>
</ingresso:template>
```

2. Vamos criar uma action em `FilmeController` para atender as requisões *GET* para `/filme/{id}/detalhes`:

```
@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;
}
```

3. Para que nosso método funcione corretamente vamos injetar `SessaoDao` no `FilmeController`:

```
@Controller
public class FilmeController {

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

    .
    .
    .
}
```

4. 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)
                .getResultList();
    }
```

TRAZENDO DADOS REAIS PARA NOSSA APLICAÇÃO

Se repararmos nossa tela de detalhes ainda não possui todos os detalhes que falamos, para nos ajudar a fazer isso, existe uma `API` bem conhecida que pode ser consumida para isto, usaremos a *OMDB*.

Primeira coisa que é necessário fazermos é a requisição, o próprio `Spring` já tem uma classe especifica que pode nos ajudar, a classe : `RestTemplate` . Para podermos utiliza-la basta a instanciarmos :

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

Agora é necessário apenas que façamos nossa requisição :

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

Bom só para entender um pouco o que aconteceu agora nessa última linha, deixamos claro para o `RestTemplate` , que ele precisa fazer uma requisição do tipo *Get*, que irá devolver um objeto que nós temos, nesse caso o `DetalhesDoFilme` .

A classe `DetalhesDoFilme` vai representar a resposta, para isso é necessário que informemos o que ela deve conhecer, em outras palavras, em 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, que 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. Para isso 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 que 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 vazio :

```
try {

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

    Optional.of(detalhesDoFilme);

} catch(RestClientException e){

    Optional.empty();
}
```

Agora só faltou deixarmos esse comportamento isolado dentro um método :

```
public Optional<DetalhesDoFilme> request(Filme filme) {

    RestTemplate client = new RestTemplate();

    String url = // endereço de onde estamos batendo

    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 = //

        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 levarmos qualquer problema nessa requisição, não sabemos o que nos levou a esse ponto, podíamos fazer um simples `println` da `Exception`, mas seria bem chato de conseguir o identificar no console. Em projetos grandes, que existem vários usuários e eventualmente algumas falhas, para pegar o erro é um pouco mais chato, mas para solucionar esse problema eles usam uma biblioteca especifica de *log* de sistema, conhecida como : *Log4J*.

Vamos começar a utiliza-la em nosso sistema, 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, para isso :

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

Nesse caso ele vai dar um log para um erro, além disso podemos o utilizar para fazer debug, passar informação e mais algumas opções.

4.5 EXERCÍCIO - CONSUMINDO SERVIÇO PARA DETALHES DO FILME

1. Crie a classe `DetalhesDoFilme` no pacote `br.com.caelum.ingresso.modelo` representando os detalhes do filme retornado pela api `http://www.omdbapi.com/?t=Rogue+One&y=&plot=short&r=json` :

```
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
}
```

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

```
@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("http://www.omdbapi.com/?t=%s&y=&plot=short&r=json", titulo);

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

}
```

2. Vamos alterar a action `detalhe` no `FilmeController` para consumir o serviço e disponibilizar na página:

3. Injete `ImdbClient` no `FilmeController`:

```
@Controller
public class FilmeController {

    @Autowired
    private FilmeDao filmeDao;

    @Autowired
    private SessaoDao sessaoDao;
```

```
    @Autowired
    private ImdbClient client;

    .
    .
    .
}
```

- Altere a action 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;
}
```

1. Vamos alterar nossa página de detalhe para exibir os dados de detalhes do filme:

```
<%@ 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" %>
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>

<ingresso:template>
    <jsp:body>
        <div class=" col-md-6 col-md-offset-3">
            <h1>${detalhes.titulo}</h1>
            <image src="${detalhes.imagem}" />

            <div>
                <label for="ano">Ano</label>
                <span id="ano">${detalhes.ano}</span>
            </div>

            <div>
                <label for="diretores">Diretores</label>
                <span id="diretores">${detalhes.diretores}</span>
            </div>

            <div>
                <label for="escritores">Escritores</label>
                <span id="escritores">${detalhes.escritores}</span>
            </div>

            <div>
                <label for="atores">Atores</label>
                <span id="atores">${detalhes.atores}</span>
            </div>

            <div>
```

```
                <label for="descricao">Descrição</label>
                <span id="descricao">${detalhes.descricao}</span>
            </div>

            <div>
                <label for="avaliacao">Avaliação</label>
                <span id="avaliacao">${detalhes.avaliacao}</span>
            </div>

                <table class="table table-hover">
                    <thead>
                        <th>Sala</th>
                        <th>Horario</th>
                        <th>Ações</th>
                    </thead>

                    <tbody>
                        <c:forEach items="${sessoes}" var="sessao">
                            <tr>
                                <td>${sessao.sala.nome}</td>
                                <td>${sessao.horario}</td>
                                <td>
                                    <a href="/sessao/${sessao.id}/lugares" class="btn">
                                        Comprar
                                        <span class="glyphicon glyphicon-blackboard" aria-hidden=
"true"></span>
                                    </a>
                                </td>
                            </tr>
                        </c:forEach>
                    </tbody>
                </table>

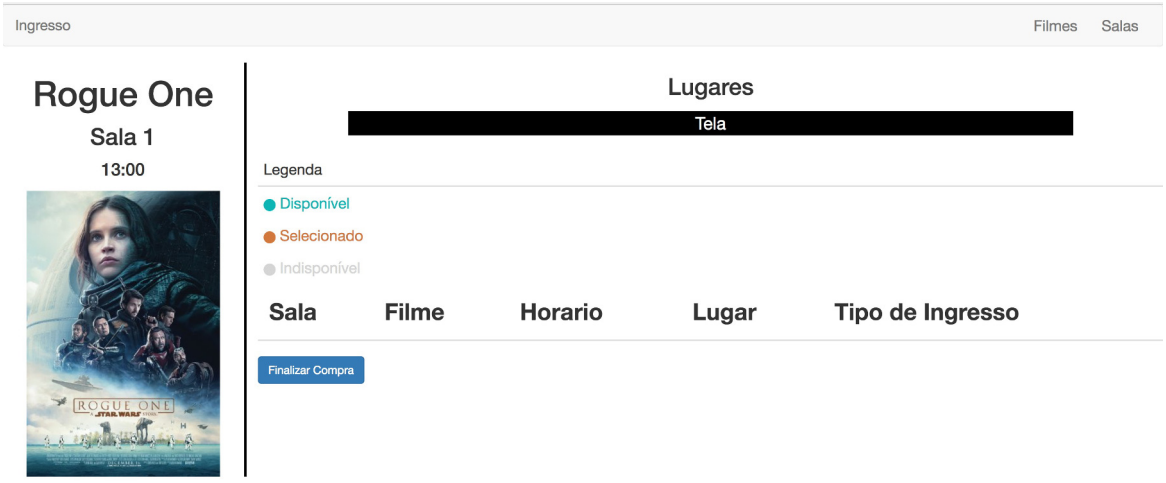
            </div>
        </jsp:body>
    </ingresso:template>
```

Capítulo 5

INICIANDO O PROCESSO DE VENDA

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 logo que decide qual será a sessão que ele deseja ver.

Para isso criaremos 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, para isso vamos 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;
```

```
    }
}
```

Agora precisamos deixar a classe `ImdbClient` generica, ou seja, que ela possa devolver tanto o `DetalhesDoFilme` quanto a nossa nova classe `ImagemCapa` , para isso temos que fazer essa sútil alteração :

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

    RestTemplate client = new RestTemplate();

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

    String url = String.format("http://www.omdbapi.com/?t=%s&y=&plot=short&r=json", titulo);

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

Precisamos apenas disponibilizar essas informações para a tela agora :

```
@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()));

    return modelAndView;
}
```

5.1 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 {

 .
 .
 .
```

```
    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 `ImagemCapa` no pacote de modelos usaremos ela para pegar somente a imagem de capa do filme retornado pela api `http://www.omdbapi.com/?t=NOME+DO+FILME&y=&plot=short&r=json` :

```
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` para que ela seja genérica, ou seja indiferente da representação (`DetalheDoFilme` ou `ImagemCapa`). Seja 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("http://www.omdbapi.com/?t=%s&y=&plot=short&r=json", 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:

```
@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` para consiga 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<ImagemCapa> imagemCapa = client.request(sessao.getFilme(), ImagemCapa.class);

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

    return modelAndView;
}
```

1. Acesse a página.

5.2 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 DescontoDeTrintaPorCentoParaBancos());

    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

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

```
    }

    // getters
}
```

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

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

E por fim temos que 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<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;
}
```

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

1. Nosso ingresso além de ter um preço e uma sessão deve ter um lugar. Altere a classe `Ingresso` e 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
```

1. Vamos adicionar na interface `Desconto` um método para pegar a descrição do desconto:

```
public interface Desconto {

 BigDecimal aplicarDescontoSobre(BigDecimal precoOriginal);
```

```
    String getDescricao();
}
```

1. Vamos implementar o método `getDescricao` em todos os descontos:

```
public class SemDesconto implements Desconto {

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

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

```
public class DescontoEstudante  implements Desconto {

    private BigDecimal metade = new BigDecimal("2.0");

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

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

```
public class DescontoDeTrintaPorCentoParaBancos implements Desconto {

    private BigDecimal percentualDeDesconto = new BigDecimal("0.3");

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

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

    private BigDecimal trintaPorCentoSobre(BigDecimal precoOriginal) {
        return precoOriginal.multiply(percentualDeDesconto);
    }
}
```

1. Ao invés de receber um desconto nosso ingresso deve receber um `TipoDeIngresso` , crie o enum `TipoDeIngresso` :

```
public enum TipoDeIngresso {

    INTEIRO(new SemDesconto()),
    ESTUDANTE(new DescontoEstudante()),
    BANCO(new DescontoDeTrintaPorCentoParaBancos());
```

```
    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();
    }

}
```

1. Altere o ingresso para receber um `TipoDeIngresso` ao invés do desconto:

```
@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 tipoDeDesconto, Lugar lugar) {
            this.sessao = sessao;
            this.tipoDeIngresso = tipoDeDesconto;
            this.preco = this.tipoDeIngresso.aplicaDesconto(sessao.getPreco();

            this.lugar = lugar;
        }

        //demais métodos

}
```

1. Após a última alteração os testes da classe `DescontoTeste` quebraram por conta do construtor. Corriga os testes fazendo-os receber o lugar no construtor do ingresso: ```java public class DescontoTest {

```
 @Test
 public void deveConcederDescontoDe30PorcentoParaIngressosDeClientesDeBancos(){

     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.now(), filme, sala);
     Ingresso ingresso = new Ingresso(sessao, TipoDeIngresso.BANCO, lugar);
```

```
        BigDecimal precoEsperado = new BigDecimal("22.75");

        assertEquals(precoEsperado, ingresso.getPrecoComDesconto());

    }


    @Test
    public void deveConcederDescontoDe50PorcentoParaIngressoDeEstudante(){

        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.now(), filme, sala);
        Ingresso ingresso = new Ingresso(sessao, TipoDeIngresso.ESTUDANTE, lugar);

        BigDecimal precoEsperado = new BigDecimal("16.25");


        assertEquals(precoEsperado, ingresso.getPrecoComDesconto());

    }

    @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.now(), filme, sala);
        Ingresso ingresso = new Ingresso(sessao, TipoDeIngresso.INTEIRO,lugar);


        BigDecimal precoEsperado = new BigDecimal("32.5");


        assertEquals(precoEsperado, ingresso.getPrecoComDesconto());


    }

}


1. Rode os testes e garanta que todos estão funcionando corretamente.

1. Altere a página `sessao/lugares.jsp` adicione a classe de css e `onclick` abaixo no SVG do lugar:
```html
    <svg class="assento ${sessao.isDisponivel(lugar) ? "disponivel" : "ocupado" }" onclick="${sessao.
isDisponivel(lugar) ? 'changeCheckbox(this)' : '' }" id="${lugar.id}" version="1.0" xmlns="http://www
.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" x="0px" y="0px"
                                            viewBox="0 0 318.224 305.246" enable-background="new
0 0 318.224 305.246" xml:space="preserve">
        .
        .
        .
    </svg>
```

1. Altere a tag `c:forEach` que fica logo em cima da tag `svg` :

```
<c:forEach var="lugar" items="${map.value}">
```
```

1. Altere a tag `c:forEach` que fica em baixo da tag `tbody` :

```
<c:forEach var="map" items="${sessao.mapaDeLugares}">
```

1. Para implementar o método `isDisponivel` na classe `Sessao` precisamos ter uma lista de ingressos dessa sessão, vamos alterar-la para ter uma lista de sessões:

```
@Entity
public class Sessao {

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

    @ManyToOne
    private Sala sala;

    @ManyToOne
    private Filme filme;

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


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

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

    // getters e setters
}
```

1. Vamos alterar a classe `Sessao` para adicionar o método `isDisponivel` , que deve verificar se o lugar está ou não disponivel:

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

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

```
@Test
public void garanteQueOLugarA1EstaOcupadoEOsLugaresA2EA3Disponiveis(){

    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.now(), rogueOne, eldorado7);

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

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

        sessao.setIngressos(ingressos);

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

3. Altere o método `lugaresNaSessao` 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;
    }
}
```

Capítulo 6

CRIANDO O CARRINHO

O CARRINHOS DE COMPRAS

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

```
public class Carrinho {



}
```

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

```
@Component
@SessionScoped
public class Carrinho {



}
```

Agora precisamos deixar nosso carrinho mais inteligente, dar algumas propriedades a ele, por exemplo, ele vai precisar 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 estamos pegando a sessão precisamos injetar nosso carrinho :

```
    @Controller
    public class SessaoController {

        @Autowired
```

```
    private Carrinho carrinho;

}
```

E por fim precisamos assim que o usuário decidir todos os ingressos que quer, adicionar 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;
    }
```

6.1 EXERCÍCIO - IMPLEMENTANDO A TELA DE COMPRAS

1. Ao selecionar o lugar e o tipo de ingresso, vamos adicionar os ingressos no carrinho. Para isso vamos criar a classe `Carrinho` no pacote de modelos.

```
public class Carrinho {

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

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

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

1. Altere o escopo do carrinho para `SessionScope` , fique atento para pegar o import do Spring:

```
@Component
@SessionScope
public class Carrinho {

    //restante da implementação
}
```

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

```
@Controller
public class SessaoController {
    .
    .
    .

    @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;
    }

}
```

1. 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. Nela crie um método que irá retornar uma lista de ingressos válida:

```
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());

    }
}
```

1. Adicione o método `findOne` na classe `LugarDao` :

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

1. Vamos criar a classe `CompraController` e 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;
        }

    }
```

6.2 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 :

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

E agora precisamos usar esse nosso método em nossa `jsp` no próprio item, que representa o lugar :

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

6.3 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));
    }
```

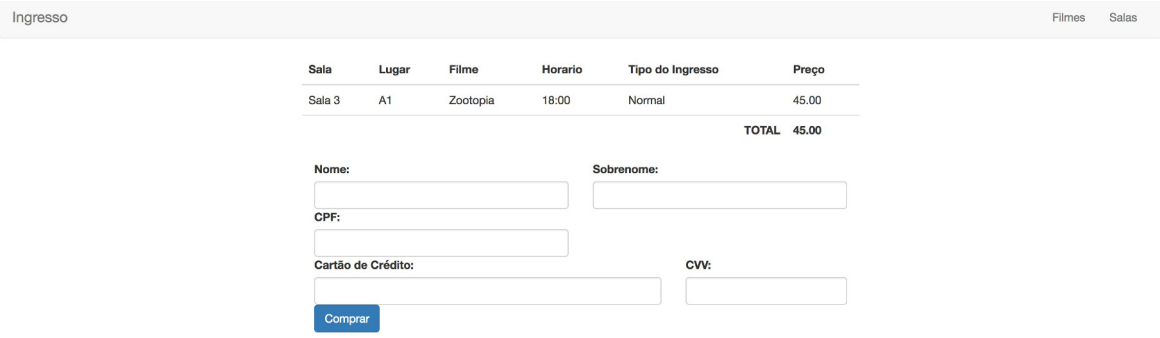
1. Vamos alterar a tela de seleção de lugares/ingresso, para que não seja possível selecionar os lugares que já foram adicionados no carrinho. Para isso altere o atributo `class` e `onclick` da tag `svg`

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

```
</svg>
```

6.4 EXIBINDO OS DETALHES DA COMPRA

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



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 :

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

Dessa forma, nosso código não vai compilar, pois ainda não existe o método `getPrecoComDesconto` na classe `Ingresso` :

```
 public BigDecimal getPreco() {
    return preco;
}

public BigDecimal getPrecoComDesconto(){
    return tipoDeIngresso.aplicaDesconto(preco);
}
```

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

6.5 EXERCÍCIO - IMPLEMENTANDO A TELA DE CHECKOUT

1. 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;
}
```

1. Precisamos listar o preço original do ingresso, o valor do desconto referente ao tipo do ingresso e o valor com desconto. Altere a classe ingresso e adicione esses métodos:

```
 public BigDecimal getPreco() {
    return preco;
}

public BigDecimal getPrecoComDesconto(){
    return tipoDeIngresso.aplicaDesconto(preco);
}
```

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

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

6.6 REALIZANDO A COMPRA

Ainda é necessário podermos finalizarmos 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 entramos apenas com `String` precisamos informar como deve ser a conversão, para isso será necessário criarmos nosso conversor :

```
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"/>
```

Por fim falta termos um objeto que represente a compra para que possamos salva-lo :

```
@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
}
```

6.7 EXERCÍCIOS - IMPLEMENTANDO A COMPRA

1. 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
}
```

1. 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;
    }
```

1. Crie o conversor para `YearMonth` no pacote `converter`:

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

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

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

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

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

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

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

1. 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;
    }
}
```

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

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

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

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

1. 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;
}
```

1. 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
}
```

1. Crie a classe `CompraDao` no pacote `br.com.caelum.ingresso.dao` :

```
@Repository
public class CompraDao {

    @PersistenceContext
    private EntityManager manager;



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

1. 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>
```

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

CAPÍTULO 7

IMPLEMENTANDO SEGURANÇA

7.1 PROTEGENDO NOSSAS URIS

Agora que concluimos o desenvolvovimento 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 acessiveis.

Por exemplo, sabemos que a *URI* `/filme/em-cartaz` deve ser acessivel por todos, já a *URI* `/filme` só deve 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 EXERCÍCIO - ALTERANDO NOSSOS MAPEAMENTO DE URIS

1. Na classe `FilmeController` altere os seguintes mapeamentos
   - De `@GetMapping({"/filme", "/filme/{id}"})` para `@GetMapping({"/admin/filme", "/admin/filme/{id}"})`
   - De `@PostMapping("/filme")` para `@PostMapping("/admin/filme")`
   - De `@DeleteMapping("/filme/{id}")` para `@DeleteMapping("/admin/filme/{id}")`
2. No arquivo `WEB-INF/views/filme/filme.jsp` vamos alterar a action da tag `<form>` para a nova *URI*: De

   ```
   <form action='/filme' method="post">
   ```

   Para

   ```
   <form action='/admin/filme' method="post">
   ```

3. No arquivo `WEB-INF/views/filme/lista.jsp` vamos alterar o endereço do link para um novo filme para a nova *URI*: De:

```
<a href="/filme" class="btn btn-block btn-info">Novo</a>
```

Para:

```
<a href="/admin/filme" class="btn btn-block btn-info">Novo</a>
```

4. Ainda no arquivo `WEB-INF/views/filme/lista.jsp` vamos alterar o endereço da chamada para exclusão para a nova *URI*: De:

```
function excluir(id) {
    var url = window.location.href;
    $.ajax({
        url:"/filme/" + id,
        type: 'DELETE',
        success: function (result) {
            console.log(result);

            window.location.href = url;
        }
    });
}
```

Para:

```
function excluir(id) {
    var url = window.location.href;
    $.ajax({
        url:"/admin/filme/" + id,
        type: 'DELETE',
        success: function (result) {
            console.log(result);

            window.location.href = url;
        }
    });
}
```

5. Na Classe `LugarController` altere os seguintes mapeamentos:
   - De `@GetMapping("/lugar")` para `@PostMapping("/admin/lugar")`
   - De `@PostMapping("/lugar")` para `@PostMapping("/admin/lugar")`
6. Ainda na classe `LugarController` altere o retorno do método `salva` para: `return new ModelAndView("redirect:/admin/sala/"+salaId+"/lugares/");`
7. No arquivo `WEB-INF/views/lugar/lugar.jsp` vamos alterar a action da tag `<form>` para a nova *URI*: De:

```
<form action="/lugar" method="post">
```

Para:

```
<form action="/admin/lugar" method="post">
```

8. No arquivo `WEB-INF/views/lugar/lista.jsp` vamos alterar o endereço do link para um novo lugar para a nova *URI*: De:

```
<a href="/lugar?salaId=${sala.id}" class="btn btn-block btn-info">Novo</a>
```

Para:

```
<a href="/admin/lugar?salaId=${sala.id}" class="btn btn-block btn-info">Novo</a>
```

9. Na Classe `SalaController` altere os seguintes mapeamentos:
   - De `@GetMapping({"/sala", "/sala/{id}"})` para `@GetMapping({"/admin/sala", "/admin/sala/{id}"})`
   - De `@PostMapping("/sala")` para `@PostMapping("/admin/sala")`
   - De `@GetMapping("/salas")` para `@GetMapping("/admin/salas")`
   - De `@GetMapping("/sala/{id}/sessoes")` para `@GetMapping("/admin/sala/{id}/sessoes")`
   - De `@GetMapping("/sala/{id}/lugares/")` para `@GetMapping("/admin/sala/{id}/lugares/")`
   - De `@DeleteMapping("/sala/{id}")` para `@DeleteMapping("/admin/sala/{id}")`
10. No arquivo `WEB-INF/views/sala/sala.jsp` vamos alterar a action da tag `<form>` para a nova *URI*: De:

```
<form action='/sala' method="post">
```

Para:

```
<form action='/admin/sala' method="post">
```

11. No arquivo `WEB-INF/views/sala/lista.jsp` vamos alterar o endereço do link para uma nova sala para a nova *URI*: De:

```
<a href="/sala" class="btn btn-block btn-info">Novo</a>
```

Para:

```
<a href="/admin/sala" class="btn btn-block btn-info">Novo</a>
```

12. Ainda no arquivo `WEB-INF/views/sala/lista.jsp` vamos alterar o endereço dos links para sessão, lugares e alterar sala:
    - Sessão
      - De:

```
<a href="/sala/${sala.id}/sessoes/" class="btn btn-primary">
```

      - Para:

```
<a href="/admin/sala/${sala.id}/sessoes/" class="btn btn-primary">
```

    - Lugares:
      - De:

```
<a href="/sala/${sala.id}/lugares/" class="btn btn-warning">
```

- Para:

```
<a href="/admin/sala/${sala.id}/lugares/" class="btn btn-warning">
```

- Alteração da Sala
  - De:

```
`<a href="/sala/${sala.id}" class="btn btn-info">Alterar</a>
```

  - Para:

```
<a href="/admin/sala/${sala.id}" class="btn btn-info">Alterar</a>
```

13. Por ultimo no arquivo `WEB-INF/views/sala/lista.jsp` vamos alterar o trecho de javascript para exclusão da sala: De:

```
function excluir(id) {
    var url = window.location.href;
    $.ajax({
        url: "/sala/" + id,
        type: 'DELETE',
        success: function (result) {
            console.log(result);

            window.location.href = url;
        }
    });
}
```

Para:

```
function excluir(id) {
    var url = window.location.href;
    $.ajax({
        url: "/admin/sala/" + id,
        type: 'DELETE',
        success: function (result) {
            console.log(result);

            window.location.href = url;
        }
    });
}
```

14. Na Classe `SessaoController` altere os seguintes mapeamentos:

- De `@GetMapping("/sessao")` para `@GetMapping("/admin/sessao")`
- De `@PostMapping("/sessao")` para `@PostMapping("/admin/sessao")`

15. No arquivo `WEB-INF/views/sessao/sessao.jsp` vamos alterar a action da tag `<form>` para a nova *URI*: De:

```
<form action='/sessao' method="post">
```

Para:

```
<form action='/admin/sessao' method="post">
```

16. No arquivo `WEB-INF/views/sessao/lista.jsp` vamos alterar o endereço do link para uma nova sala para a nova *URI* e o trecho javascript para exclusão também:

    - Link para nova sessão
        - De:

```
<a href="/sessao?salaId=${sala.id}" class="btn btn-block btn-info">Nova</a>
```

        - Para:

```
<a href="/admin/sessao?salaId=${sala.id}" class="btn btn-block btn-info">Nova</a>
```

    - Trecho de javascript para exclusão:
        - De:

```
function excluir(id) {
    var url = window.location.href;
    $.ajax({
        url:"/sessao/" + id,
        type: 'DELETE',
        success: function (result) {
            console.log(result);

            window.location.href = url;
        }
    });
}
```

        - Para

```
function excluir(id) {
    var url = window.location.href;
    $.ajax({
        url:"/admin/sessao/" + id,
        type: 'DELETE',
        success: function (result) {
            console.log(result);

            window.location.href = url;
        }
    });
}
```

17. Por fim vamos alterar os links para Sala e Filme no arquivo `template.tag` : De:

```
<li><a href="/filmes">Filmes</a></li>
<li><a href="/salas">Salas</a></li>
```

    Para:

```
<li><a href="/admin/filmes">Filmes</a></li>
<li><a href="/admin/salas">Salas</a></li>
```

7.3 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.

Poderiamos 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 *KeyCloack* 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 ususário 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}/lugraes` 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 aleatóriamente.

Como não estamos gerando esse *Token* e nem colocando esse *INPUT HDDEN* em nossos formulários, vamos desabilitar uma prevenção de seguraçacontra ataques *CSRFs*.

```
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 exceçõe 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 anota-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);
    }
}
```

7.4 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 {

}
`
```

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

```
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

}
```

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

```
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
```

```
    protected void configure(HttpSecurity http) throws Exception {

    }

}
```

1. 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();
    }

}
```

1. 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/**");
    }

}
```

1. Crie um construtor sem argumentos para a classe `SecurityInitializer` e nele chame o `super` e passe a clase `SecurityConfiguration` para ele.

```
@Configuration
public class SecurityInitializer extends AbstractSecurityWebApplicationInitializer {

    public SecurityInitializer() {
        super(SecurityConfiguration.class);
    }

}
```

1. 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";
    }
}
```

7.5 USUARIO, SENHA E PERMISSAO

Agora que já configuramos a segurança da nossa aplicação, precisamos criar algo que represente nosso Usuario 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 `Ususário` 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());
    }

}
```

7.6 EXERCÍCIO - IMPLEMENTANDO USERDETAIL, USERDETAILSERVICES 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;
    }
}
```

1. 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
}
```

1. 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;
    }
}
```

1. 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;
    }

}
```

1. 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!");
        }

    }
}
```

2. 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 implementção
}
```

3. Injete `UserDetailsServices` 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());
    }

}
```

Capítulo 8

CRIANDO UMA NOVA CONTA

Quando queremos criar uma nova conta em alguma aplicação, gerealmente existe um formluário 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>

<a href="/usuario/request">ou cadastrar-se</a>
...
```

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 precisamo criar o formulário para o usuário informar o e-mail ao qual ele quer solicitar acesso. Vamos criar o arquivo `usario/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">
                <span class="text-danger">${param.error}</span>
```

```
                <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 <strong>${usuario.email}</strong>.<
br/>

            Por favor confirme a criação do seu usuário, para liberar seu acesso.
        </p>

    </jsp:body>
</ingresso:template>
```

Como implementamos seguraça 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/**");
    }

}
```

8.1 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>

<a href="/usuario/request">ou cadastrar-se</a>
...
```

1. 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");
    }
}
```

1. 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">
                <span class="text-danger">${param.error}</span>
```

```
                <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>
```

1. 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;
}
```

1. 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 <strong>${usuario.email}</strong>.<
br/>

            Por favor confirme a criação do seu usuário, para liberar seu acesso.
        </p>

    </jsp:body>
</ingresso:template>
```

1. 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();

}
```

Agora que temos a primeira parte do fluxo pronta, vamos implmentar 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 <a href=%s>link</a> 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 repsonsável 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>
```

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();
}
```

1. 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();
    }
}
```

1. 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 <a href=%s>link</a> 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());
    }
}
```

1. 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);
        }
    }
}
```

8.2 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;
    }
}
```

8.3 EXERCÍCIO - ENVIANDO E-MAIL

1. 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);
    }
}
```

1. 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;
    }
}
```

1. Injete a classe `Mailer` e `TokenHelper` em `UsuarioController`:

```
@Controller
public class UsuarioController {

    @Autowired
    private Mailer mailer;

    @Autowired
    private TokenHelper tokenHelper;

    // ... demais implementações

}
```

1. Altere o método `solicitacaoDeAcesso` na classe `UsuarioController` pare 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;
}
```

8.4 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">
                <span class="text-danger">${param.error}</span>


                <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-co
ntrol">
                </div>

                <button class="btn btn-primary" type="submit">Cadastrar</button>

            </form>
        </jsp:body>
```

```
</ingresso:template>
```

8.5 EXERCÍCIO - VALIDANDO TOKEN

1. 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){

}
```

1. Adicione o método `findByUuid` na classe `TokenDao`

```
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();
}
```

1. Crie o método `getTokenFrom` na classe `TokenHelper` que recebrá uma `String` com o `UUID`. Esse método deve buscar o token no banco de dados:

```
public Optional<Token> getTokenFrom(String uuid) {
    return dao.findByUuid(uuid);
}
```

1. 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;

    }
}
```

1. 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);
    }
}
```

1. 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;
}
```

1. Crie a página `usuario/confirmacao.jsp` :

```
<ingresso:template>
    <jsp:body>
        <form action="/usuario/cadastrar" method="post">
            <span class="text-danger">${param.error}</span>


            <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-co
ntrol">
                </div>

                <button class="btn btn-primary" type="submit">Cadastrar</button>

            </form>
        </jsp:body>
    </ingresso:template>
```

8.6 PERSISTINDO O USUÁRIO

Quando o usuáiro 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 ususário 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 encriptedPassword = encoder.encode(this.password);

        String email = token.getEmail();

        Usuario usuario= dao.findByEmail(email).orElse(novoUsuario(email, encriptedPassword));

        usuario.setPassword(encriptedPassword);

        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 métod `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);
}
```

8.7 EXERCÍCIO - CADASTRANDO USUÁRIO

1. 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;
}
```

1. Adicione o método `toUsuario` na classe `ConfirmacaoLoginForm` :

```
public class ConfirmacaoLoginForm {

    //... restante da implementação

    public Usuario toUsuario(UsuarioDao dao, PasswordEncoder encoder) {

        String encriptedPassword = encoder.encode(this.password);

        String email = token.getEmail();

        Usuario usuario= dao.findByEmail(email).orElse(novoUsuario(email, encriptedPassword));

        usuario.setPassword(encriptedPassword);

        return usuario;
    }

    private Usuario novoUsuario(String email, String password){
        Set<Permissao> permissoes = new HashSet<>();
        permissoes.add(Permissao.COMPRADOR);

        return new Usuario(email, password, permissoes);
    }
}
```

1. Crie a classe `UsuarioDao` e implemente o método `findByEmail` , que deve retornar um usuário a partir do e-mail se existir: ```java @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();
}
```

```
    }
```

```
1. Caso o formulário seja válido devemos persistir o usuário no banco de dados.
Para isso altere o método `cadastrar` na classe `UsuarioController`:

```java
    @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;
    }
```

1. Adicione o método `save` na classe `UsuarioDao`:

```
    public void save(Usuario usuario) {

        if (usuario.getId() == null)
            manager.persist(usuario);
        else
            manager.merge(usuario);
    }
```
```