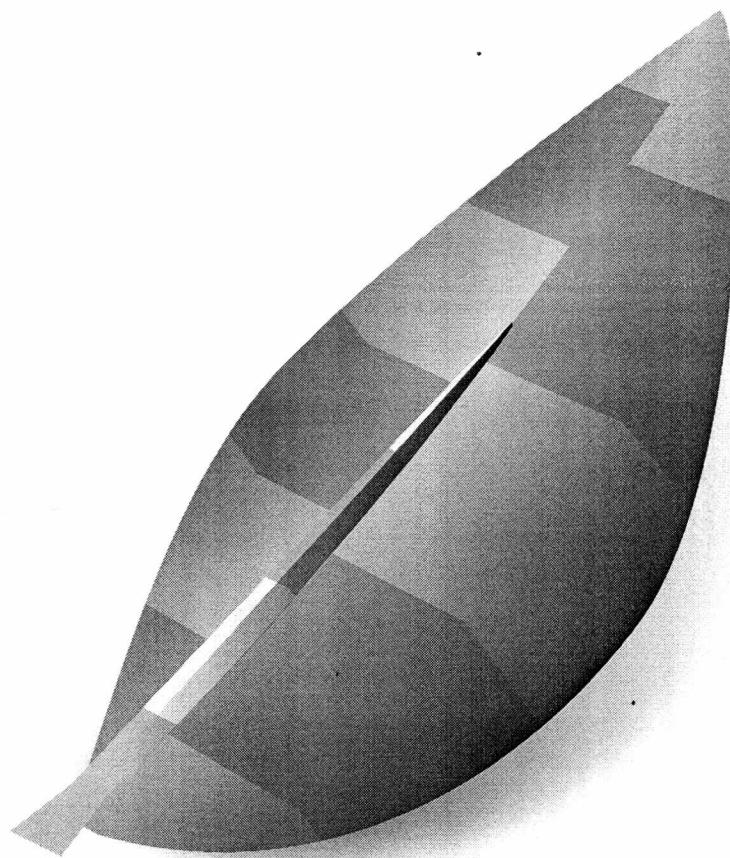


FJ-27

Spring Framework



Caelum
Ensino e Inovação

www.caelum.com.br



A Caelum atua no mercado com consultoria, desenvolvimento e ensino em computação. Sua equipe participou do desenvolvimento de projetos em vários clientes e, após apresentar os cursos de verão de Java na Universidade de São Paulo, passou a oferecer treinamentos para o mercado. Toda a equipe tem uma forte presença na comunidade através de eventos, artigos em diversas revistas, participação em muitos projetos *open source* como o VRaptor e o Stella e atuação nos fóruns e listas de discussão como o GUJ.

Com uma equipe de mais de 80 profissionais altamente qualificados e de destaque do mercado, oferece treinamentos em Java, Ruby on Rails e Scrum em suas três unidades - São Paulo, Rio de Janeiro e Brasília. Mais de 8 mil alunos já buscaram qualificação nos treinamentos da Caelum tanto em suas unidades como nas próprias empresas com os cursos *incompany*.

O compromisso da Caelum é oferecer um treinamento de qualidade, com material constantemente atualizado, uma metodologia de ensino cuidadosamente desenvolvida e instrutores capacitados tecnicamente e didaticamente. E oferecer ainda serviços de consultoria ágil, mentoring e desenvolvimento de projetos sob medida para empresas.

Comunidade



Nossa equipe escreve constantemente artigos no **Blog da Caelum** que já conta com 150 artigos sobre vários assuntos de Java, Rails e computação em geral. Visite-nos e assine nosso RSS:

- blog.caelum.com.br



Acompanhe a Caelum no **Twitter**:

- twitter.com/caelum



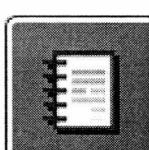
O **GUJ** é maior fórum de Java em língua portuguesa, com 700 mil posts e 70 mil usuários. As pessoas da Caelum participam ativamente, participe também:

- www.guj.com.br



Assine nossa **Newsletter** para receber notícias técnicas de Java, Rails, Agile e Web, além de novidades, eventos e artigos:

- www.caelum.com.br/newsletter



No site da Caelum há algumas de nossas **Apostilas** disponíveis gratuitamente para download e alguns dos **artigos** de destaque que escrevemos:

- www.caelum.com.br/apostilas
- www.caelum.com.br/artigos

Índice

1 Simplificando o desenvolvimento Enterprise	1
1.1 Por que Spring?	1
1.2 Sobre o curso FJ-27	1
1.3 Tirando dúvidas	2
1.4 Bibliografia	2
2 Injeção de dependências e Inversão de Controle	3
2.1 Nossa projeto	3
2.2 Dependências e o alto acoplamento	4
2.3 Injeção de dependências	5
2.4 Inversão de controle e o princípio de hollywood	7
2.5 Para saber mais: Registry como alternativa a DI	7
2.6 Para saber mais...	8
3 Spring Framework	9
3.1 O que é?	9
3.2 Arquitetura do Spring	10
3.3 Introdução aos beans do Spring	12
3.4 Configurando o Spring	12
3.5 Iniciando o container	13
3.6 Injetando as Dependências: por construtor e por setter	13
3.7 Exercícios: Projeto com Spring IoC	14
3.8 Exercícios opcionais: Injecão com Setter e Tipos Básicos	17
3.9 Exercícios Opcionais: Valores Literais	18

4 Spring MVC	21
4.1 Spring MVC	21
4.2 Criando seu projeto web no Eclipse	21
4.3 Configurando o Spring MVC	22
4.4 View Resolvers	22
4.5 Seu primeiro Controller	23
4.6 Configurando	23
4.7 Exercícios: Nosso primeiro controller	24
4.8 Enviando dados para View	24
4.9 RequestMapping mais a fundo	25
4.10 O Controller de Produto	26
4.11 Gerenciamento e Injeção	26
4.12 Exercícios - Listando os Produtos	27
4.13 URLs amigáveis	28
4.14 Melhor uso do HTTP	28
4.15 Exercícios: Melhor uso do http	29
4.16 Recebendo paramêtros de Formulario	30
4.17 Forward X Redirect	31
4.18 Exercícios - Cadastro de Produto	32
4.19 Escopos	32
4.20 Web Escopos: Request, Session, Global Session	33
5 Integrando Spring, Hibernate e tecnologias de Persistência	35
5.1 Introdução - Onde o Spring pode ajudar ao usar o Hibernate	35
5.2 Configurando o Hibernate no Spring	35
5.3 ProdutoHibernateDAO	36
5.4 @Repository	37
5.5 Exercícios	37
5.6 @Qualifier	39
5.7 Exercícios	39
5.8 Não quero escolher um objeto, quero um padrão!	39
5.9 Exercícios	40
5.10 Transações com Spring - Spring Transaction	40
5.11 Configurando o TransactionManager	41

5.12 Controle de Session com o Spring	41
5.13 Exercícios	42
5.14 Exercícios - Alterar Produtos	43
5.15 Exercício Opcional: Remoção de produtos	44
5.16 Aumentando nosso domínio	44
5.17 Entidade	44
5.18 Exercícios	46
5.19 Mostrando as movimentações na tela	49
5.20 Exercícios	49
5.21 Lidando com LazyInitializationException	50
5.22 Solução Open Session in View	51
5.23 Exercícios	51
5.24 Transações Avançadas	52
5.25 Para saber mais: HibernateTemplate e outros Templates	53
6 Validação com Bean Validation	55
6.1 Introdução	55
6.2 Bean Validation	55
6.3 Hibernate Validator	55
6.4 Exercícios	56
6.5 Mostrando erros em tela	57
6.6 Exercícios	58
6.7 Customizando mensagens	59
6.8 Exercício	60
7 Spring Security	61
7.1 Introdução ao Spring Security	61
7.2 Exercícios - Configurando o Spring Security	62
7.3 - Access Control List	62
7.4 Exercícios - Criando as classes de Domínio	63
7.5 Interfaces de Acesso	64
7.6 Exercícios - Suporte a Acesso de Dados	67
7.7 Login	70
7.8 Exercícios - Logando no Sistema	71

8 Apêndice - AOP - Programação Orientada a Aspectos	75
8.1 Introdução a Programação Orientada a Aspectos	75
8.2 AOP na prática	76
8.3 Elementos de um Aspecto	77
8.4 AOP com Spring	77
8.5 AOP no Spring com @AspectJ-style	77
8.6 Exercícios - AOP	78
8.7 Definindo Pointcuts e Advices	79
8.8 Exercícios	81
8.9 Outros Advices	82
8.10 Para saber mais: Combinando Pointcuts	83
8.11 Para saber mais: @Around	84
9 Apêndice - Integração entre Sistemas	85
9.1 Spring Integration	85
9.2 Exercício: Spring Integration	86
10 Apêndice - Enviando Email	89
10.1 Enviando E-mail com Spring	89
10.2 Exercício: Spring Mail	90

Versão: 14.2.30

Simplificando o desenvolvimento Enterprise

"Há duas formas de se construir o design de um software: Uma forma é fazê-lo tão simples que obviamente não terá deficiências e a outra forma é fazê-lo tão complicado que não terá deficiências óbvias."
— C.A.R. Hoare, The 1980 ACM Turing Award Lecture

1.1 - Por que Spring?

Com suas ideias de um container leve, o Spring vem sendo apresentado há anos como uma alternativa mais simples à arquitetura oficial do Java EE baseada em EJBs. Já em 2008, a própria SpringSource já mostrava que a oferta de empregos com Spring se equiparava a de EJB:

<http://bit.ly/springsource-ejb>

E uma busca atualizada mostra que, de 2008 até hoje, a tendência se manteve: a procura por Spring cresceu bastante e a de EJB continua estável:

<http://bit.ly/jobtrends-spring-ejb>

Como veremos, o princípio fundamental do Spring é o uso de *Injeção de Dependências* nos componentes da sua aplicação. Além disso, provê um container com diversos serviços já prontos para sua aplicação. Desde o surgimento do Spring, os princípios de *Inversão de Controle* e *Injeção de Dependências* se tornaram fundamentais em qualquer tipo de aplicação. Outros frameworks de injeção surgiram, como o Pico Container e o Google Guice, e até o próprio Java EE incorporou essas técnicas em sua última versão com o *CDI - Context and Dependency Injection*.

A importância da injeção de dependências hoje é tão grande que, em uma recente entrevista com os autores do livro *"Design Patterns"* em comemoração aos 15 anos do livro, Erich Gamma, um dos autores, disse que, se houvesse uma segunda edição do livro, um padrão a ser incluído seria *Dependency Injection*:

<http://bit.ly/entrevista-design-patterns>

1.2 - Sobre o curso FJ-27

O curso FJ-27 foi reformulado com as novidades do Spring 3.x, a versão mais recente do framework. É um curso focado nos principais conceitos do Spring, desde Inversão de Controle e Injeção de Dependências até tópicos mais avançados como Orientação a Aspectos (AOP) e tecnologias como SpringMVC, REST e integração com Hibernate e outros frameworks.

Este é um curso prático, com muitos exercícios que abordam os tópicos discutidos em aula. Recomendamos fortemente que você faça todos os exercícios e esclareça suas dúvidas em aula. Foque seus estudos principalmente nos conceitos fundamentais.

1.3 - Tirando dúvidas

Para tirar dúvidas dos exercícios, ou de Java em geral, recomendamos o fórum do site do GUJ (<http://www.guj.com.br/>), onde sua dúvida será respondida prontamente.

Fora isso, sinta-se à vontade para entrar em contato com seu instrutor e tirar todas as dúvidas que tiver durante o curso.

1.4 - Bibliografia

É possível aprender muitos dos detalhes e pontos não cobertos no curso em tutoriais na Internet em portais como o GUJ, em blogs (como o da Caelum: <http://blog.caelum.com.br>) e em muitos Sites especializados.

Mas se você deseja algum livro para expandir seus conhecimentos ou ter como guia de referência, temos algumas indicações dentre várias possíveis:

Sobre Java e melhores práticas

- Refactoring, Martin Fowler
- Effective Java, Joshua Bloch
- Design Patterns, Erich Gamma et al

Sobre o Spring Framework

- Spring in Action, Craig Walls e Ryan Breidenbach
- Referência oficial: <http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/>

Injeção de dependências e Inversão de Controle

“Se queres conversar comigo, define primeiro os termos que usas comigo”
– Voltaire

Nesse capítulo, você vai entender:

- o que é Injeção de Dependências e que problemas ela resolve;
- o que é Inversão de Controle e tipos de inversões;
- como usar frameworks para satisfazer as dependências do seu projeto.

2.1 - Nosso projeto

Ao longo do curso FJ-27, vamos desenvolver um sistema de controle de *estoque de produtos*. Vamos desenvolver desde seu modelo, sua persistência, a camada Web, o controle de usuários e segurança. Tudo usando o **Spring Framework** e diversas de suas características.

Vamos começar pensando na modelagem do domínio de negócio. Primeiro, temos a classe **Produto**:

```
package br.com.caelum.estoque.model;

public class Produto {

    private Long id;
    private String descricao;
    private Long quantidade;

    //getters e setters
}
```

Para persistirmos os produtos, usaremos o padrão **DAO**. Nossa DAO será modelado como uma interface Java. Desta forma, poderemos ter várias implementações do DAO e trocá-las conforme necessário.

As operações que teremos no **ProdutoDAO**:

```
package br.com.caelum.estoque.dao;

public interface ProdutoDAO {
    void salvar(Produto produto);
    void alterar(Produto produto);
    List<Produto> listar();
    Produto buscarPorId(Long id);
}
```

No curso, será particularmente interessante ter o DAO como interface, já que vamos ver Hibernate apenas mais pra frente. Até lá, para podermos criar nosso sistema usando persistência, vamos criar uma implementação simples do DAO em memória. Mais pra frente, vamos trocá-la por uma implementação completa com Hibernate.

Nosso ProdutoMemoriaDAO, por enquanto apenas com os métodos de listagem e adição implementados:

```
package br.com.caelum.estoque.dao;

public class ProdutoMemoriaDAO implements ProdutoDAO {

    private List<Produto> produtos = new ArrayList<Produto>();

    public ProdutoMemoriaDAO() {
        Produto produto = new Produto();
        produto.setDescricao("Mac Book");
        produto.setQuantidade(40);

        produtos.add(produto);
    }

    public List<Produto> listar(){
        return produtos;
    }

    public void salvar(Produto produto) {
        produtos.add(produto);
    }

    public Produto buscarPorId(Long id) {
        // ainda não implementado
        return null;
    }

    public void alterar(Produto produto) {
        // ainda não implementado
    }
}
```

2.2 - Dependências e o alto acoplamento

Teremos também uma classe GerenciadorDeProduto responsável por pedir dados de um Produto ao usuário, depois persistir esse objeto e por fim mostrar alguma confirmação.

Se estivéssemos na Web, seria equivalente a pegar os dados do request enviado por um formulário, persistir e depois mostrar uma página de confirmação. Vamos ver a parte Web em um capítulo mais a frente. Por enquanto, nossa classe GerenciadorDeProduto será:

```
public class GerenciadorDeProduto {  
    private ProdutoDAO produtoDao;  
  
    public GerenciadorDeProduto() {  
        this.produtoDao = new ProdutoMemoriaDAO();  
    }  
  
    public void novoProduto() {  
        Produto produto = pedeDadosDoProdutoAoUsuario();  
        this.produtoDao.salvar(produto);  
        mostraConfirmacao();  
    }  
}
```

É uma classe bastante simples. Ela apenas controla a execução das coisas entre o modelo, o DAO e a interação com o usuário.

Mas ela tem um problema. Nossa classe `GerenciadorDeProduto` está acoplada não só à interface `ProdutoDAO` como à implementação concreta `ProdutoMemoriaDAO`. Repare que o construtor conhece a implementação específica e dá `new` na classe.

Dizemos que a classe tem uma **dependência** tanto com `ProdutoDAO` quanto com `ProdutoMemoriaDAO`.

Mas, quando olhamos a lógica de negócios propriamente dita, no método `novoProduto`, percebemos que o que é realmente necessário para executá-la é *algum DAO*. A lógica em si não depende de qual instância especificamente da implementação que estamos usando. (imagine, como será nosso caso daqui a pouco, que tenhamos mais de uma implementação, como a `ProdutoHibernateDAO`)

No fundo, precisamos então apenas de *algum ProdutoDAO* mas acabamos tendo que dar `new` diretamente na implementação para usá-la. **Nossa aplicação resolve a implementação e a dependência, chamando diretamente a classe específica.**

O problema desse tipo de chamada é que, no dia que precisarmos mudar a implementação do DAO, precisaremos mudar nossa classe. Agora imagine que tenhamos 10 classes no sistema que usem nosso `ProdutoDAO`. Se todas dão `new` na implementação específica a ser usada (agora, `ProdutoMemoriaDAO`), quando formos mudar a implementação para, digamos, `ProdutoHibernateDAO`, precisaremos mudar em vários lugares.

E ainda ainda um segundo problema em gerenciar a dependência diretamente na aplicação. Se a outra classe precisar de um processo de construção mais complicado, precisaremos nos preocupar com isso. Imagine a futura classe `ProdutoHibernateDAO` recebera uma `Session` do Hibernate. Ou então uma `ProdutoJDBCDAO` que precise da `Connection`. Se formos dar `new` nessas classes, precisaríamos resolver essas exigências que, se formos pensar bem, nem deveriam ser responsabilidades nossas.

2.3 - Injeção de dependências

O pattern **Dependency Injection (DI)** - Injeção de dependências - procura resolver esses problemas. A ideia é a classe não mais resolver suas dependências por conta própria mas apenas declarar que depende de alguma outra classe. E de alguma outra forma que não sabemos, alguém resolverá essa dependência para nós.

O código da `GerenciadorDeProduto` fica com uma mudança util:

```
public class GerenciadorDeProduto {  
    private ProdutoDAO produtoDao;  
  
    public GerenciadorDeProduto(ProdutoDAO produtoDao) {  
        this.produtoDao = produtoDao;  
    }  
  
    public void novoProduto() {  
        Produto produto = pedeDadosDoProdutoAoUsuario();  
        this.produtoDao.salvar(produto);  
        mostraConfirmacao();  
    }  
}
```

Repare que, agora, nossa classe não está mais acoplada à implementação alguma do ProdutoDAO - nem import do ProdutoMemoriaDAO existe mais. O que nossa classe diz é que ela **depende de algum ProdutoDAO** sem acoplar-se a nenhuma implementação específica e sem se preocupar em como resolver essa dependência.

Mas quem for usar essa minha classe, agora, vai precisar satisfazer as dependências. Pode parecer que estamos apenas empurrando o problema, mas temos que olhar as classes isoladamente. Perceba que, do ponto de vista *apenas da GerenciadorDeProduto*, estamos recebendo nossas dependências no construtor. Elas estão sendo **injetadas** para nós. Como isso vai acontecer de verdade não é problema nosso.

E, o melhor, imagine que o DAO em uso é aquele do Hibernate. E que a classe ProdutoHibernateDAO também vai usar injeção de dependências para receber a Session que ela precisa. Veja um esboço dessa classe:

```
public class ProdutoHibernateDAO implements ProdutoDAO {  
  
    private Session session;  
  
    public ProdutoHibernateDAO(Session session) {  
        this.session = session;  
    }  
  
    // implementacoes dos metodos com Hibernate, usando a Session  
}
```

Aqui, temos dois pontos importantes. O primeiro é que a classe ProdutoHibernateDAO está usando DI para não precisar se preocupar em abrir uma Session (um processo complicado). Mas, mais importante, é lembrar da nossa GerenciadorDeProduto: imagine ter que gerenciar nossa dependência do DAO diretamente e ter que se preocupar não só com o ProdutoHibernateDAO como com a dependência de Session que ele tem. Precisaríamos resolver as duas coisas, dando ainda mais trabalho.

Contudo, como usamos DI na classe GerenciadorDeProjeto, nem sabemos qual é a implementação do DAO em uso e, muito menos, suas eventuais dependências. **Se gerenciar as próprias dependências já é ruim, o que dizer de gerenciar as dependências das dependências.**

Mas voltamos ao ponto de que, quem for usar a GerenciadorDeProjeto terá um imenso trabalho! Ele terá que conhecer a GerenciadorDeProjeto, resolver a dependência dela em ProdutoDAO, o que, por sua vez, fará ele conhecer a implementação ProdutoHibernateDAO e ainda suas dependências em Session, além de saber como se cria uma Session. Parece muita coisa para uma classe só.

É aí que entram em cena os **Frameworks de Injeção de dependência**. Em projetos maiores, teremos muitas classes com muitas dependências umas nas outras, tanto direta quanto indiretamente. Resolver esse complexo grafo de dependências na mão é possível mas muito trabalhoso. Existem frameworks cujo objetivo é resolver as dependências.

Registrarmos nesses frameworks todas as nossas classes e deixamos a cargo dele resolver as interdependências entre elas. Nos preocupamos apenas em criar as classes usando o pattern de injeção de dependências e em registrar as classes nos frameworks.

E um dos frameworks mais importantes nessa parte de DI é justo o **Spring**.

2.4 - Inversão de controle e o princípio de hollywood

Injeção de dependência é, na verdade, um modo de aplicar uma técnica muito mais ampla chamada de **Inversão de Controle (IoC - Inversion of Control)**. A ideia principal de IoC é inverter quem está no controle das invocações no nosso código.

Lembre do nosso sistema: antes, nós invocávamos diretamente o construtor da `ProdutoMemoriaDAO` resolvendo as dependências manualmente. Nós controlávamos como e quando a criação do DAO aconteceria.

Quando passamos a usar DI, invertemos o controle da criação desse DAO. Nossa classe não sabe mais nem como e nem quando a dependência será criada. Alguém criará essa instância e passará para nós. Usando Spring, é o framework que passa a dar `new` nas nossas classes e não nossa aplicação diretamente.

Isso é inverter o controle. O framework passa a chamar as coisas para nós ao invés da nossa aplicação ser a responsável por isso.

Há quem chame IoC do **Princípio de Hollywood**. Dizem que há uma frase bastante famosa no mundo dos artistas de Hollywood que diz: "**Don't call us; we call you**" - ou "*Não nos chame; nós te chamamos*". É a ideia do IoC onde paramos de invocar os métodos daquilo que precisamos e passamos a receber coisas prontas e a ser invocados pelo nosso framework.

DI é uma forma de se fazer IoC mas não a única. A ideia geral é plugar o código da aplicação em algum framework que fará as invocações certas e tudo funcionará. Alguns outros exemplos:

- Quando usamos uma `HttpServlet`, implementamos um método `doGet` para tratar requisições Web. Mas não nos preocupamos em receber essas requisições e invocar o método no momento certo. Apenas criamos nossa servlet, jogamos ela no container e ele invoca o método para nós.
- Quando programamos para Desktop com Swing, frequentemente precisamos tratar eventos do usuário como, por exemplo, um clique. Ao invés de chamarmos um método para saber se o clique foi feito, no Swing, registramos um *Listener* e somos notificados do evento. Ou seja, o Swing invoca o método do nosso listener no momento do clique sem precisarmos nos preocupar.
- Ao usar EJBs, às vezes precisamos saber, por exemplo, em que momento nosso objeto foi passivado. Podemos saber isso implementando um método de *callback* que será invocado pelo container no momento da passivação. No EJB 2.x, usávamos o famoso `ejbPassivate`; no 3.x, podemos anotar qualquer método com `@PrePassivate`.

Há ainda outros exemplos onde invertemos o controle de invocação do nosso código e deixamos a cargo de alguém as chamadas no momento certo.

2.5 - Para saber mais: Registry como alternativa a DI

Outra forma de obtermos a instância de um `ProdutoDAO` sem precisar dar `new` diretamente e se acoplar com a implementação da interface é utilizar o design pattern **Registry**. Muito utilizado até a versão 1.4 da J2EE, o Registry é um padrão de projeto que utiliza a solução de registrar os recursos em um diretório de componentes central - no EJB, era o famoso **JNDI (Java Naming and Directory Interface)**. Assim, temos um lugar comum onde as classes podem obter referências a outras classes.

No exemplo, poderíamos fazer a classe `GerenciadorDeProduto` obter uma referência de `ProdutoDAO` por meio do Registry JNDI. Primeiro, precisamos registrar a instância do DAO que queremos (processo que é chamado de **bind**). Geralmente isto é feito pelo container J2EE utilizando arquivos de configuração (*deployment descriptors*) em um código parecido com este:

```
//Registra do DAO no Registry
Context ctx = new InitialContext();
ctx.bind("produtoDao", new ProdutoMemoriaDAO());
```

Nossa classe `GerenciadorDeProduto` pode, então, obter uma referência do tipo `ProdutoDAO` buscando no Registry JNDI pelo nome registrado. Este processo tem o nome de **lookup**:

```
package br.com.caelum.fj27;

public class GerenciadorDeProduto {

    private ProdutoDAO produtoDao;

    public GerenciadorDeProduto() {
        try{
            Context ctx = new InitialContext();
            produtoDao = (ProdutoDAO) ctx.lookup("produtoDao");
        } catch (NamingException ex) {
            // tratamento de exceção
        }
    }

    // outros métodos
}
```

Desta maneira, a inicialização do componente fica fora da classe `GerenciadorDeProduto` e, normalmente, pode ser feita através de arquivos de configuração. Mas, mesmo assim, o lookup da instância de `ProdutoDAO` continuará na classe `GerenciadorDeProduto`. Ou seja, continuamos invocando algo e, portanto, isso não é considerado Inversão de Controle. É apenas mais uma forma de se evitar o acoplamento com as implementações específicas das dependências.

2.6 - Para saber mais...

Para saber mais sobre IoC e DI, recomendamos a leitura dos artigos do Martin Fowler:

<http://martinfowler.com/articles/injection.html> <http://martinfowler.com/bliki/InversionOfControl.html>

Spring Framework

“Homens sábios fazem provérbios, tolos os repetem”

– Samuel Palmer

Neste capítulo, vamos ver o que é o Spring Framework, qual sua arquitetura e qual seu princípio básico de funcionamento.

3.1 - O que é?

O Spring na verdade é um **framework de inversão de controle** e não apenas de um framework de injecção de dependências. Ele faz mais coisas que apenas resolver suas dependências, é um framework completo de serviços para seus componentes.

O **Spring Framework** é um container leve (*lightweight container*), ou container de POJO's segundo o pessoal do próprio Spring, com recursos de inversão de controle e orientação à aspectos. O termo *container* é usado pelo Spring, pois ele gerencia o ciclo de vida dos objetos configurados nele, assim como um container Java EE faz com os EJB's. Porém, é considerado um container leve, pois ao contrário do container Java EE em suas primeiras versões, ele não é intrusivo e as classes da aplicação não possuem dependências com o Spring.

Além disso, podemos dizer que o Spring é também um framework de infraestrutura, pois provê serviços como gerenciamento de transações e de segurança, além de se integrar com frameworks ORM. Com ele você pode ter praticamente todos os benefícios de Java Enterprise Edition (JTA, JPA, JCA, Remoting, Servlets, JMS, JavaMail, JDBC, etc) mas sem utilizar EJB's.

Container

Podemos dizer que containers são programas utilizados para gerenciar componentes de uma aplicação. Um exemplo de um container é um container de servlets, onde sua função é gerenciar os componentes de interface web que conhecemos como Servlets e JSPs.

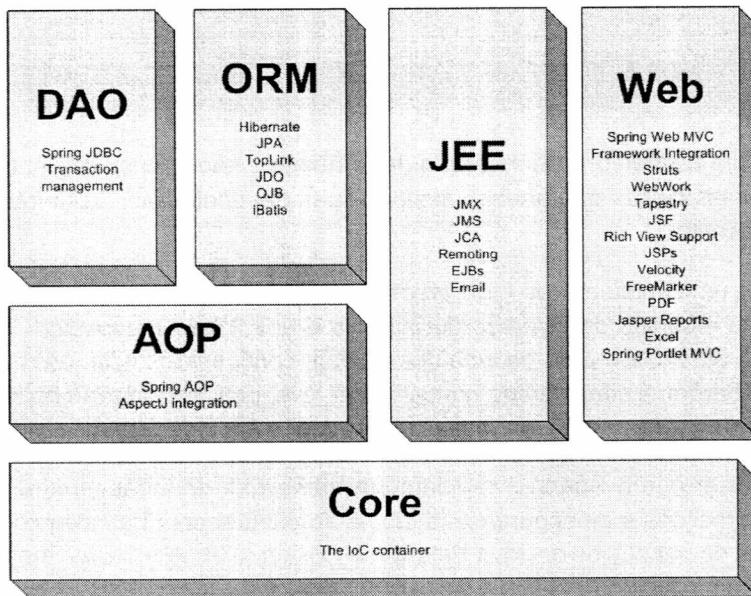
Princípios do Spring

- O código da aplicação não deve depender da API Spring;
- Spring não deve competir com boas soluções já existentes no mercado, mas deve ser capaz de se integrar a elas;
- Teste de unidade é uma tarefa crítica e o container não deve interferir neste objetivo.

Benefícios do Spring

- O container leve elimina a necessidade de códigos repetitivos como lookups;
- Framework com inversão de controle da injeção de dependência que resolve automaticamente as dependências entre beans;
- Código comum à várias seções, como gerenciamento de transações, são implementados com AOP;
- Disponibiliza camadas de abstração para outras tecnologias (como Hibernate, JPA, JDBC, JavaMail, Quartz, etc).

3.2 - Arquitetura do Spring



Atualmente, o Spring está dividido em sete módulos:

Módulo Core

O módulo principal é o Core e representa as principais funcionalidades do Spring, como o container para inversão de controle e injeção de dependências. Neste módulo, o principal elemento é a interface BeanFactory, que provê uma implementação do factory pattern, removendo do sistema a necessidade de singletons e desacoplando as dependências de configurações e especificações da lógica de negócios do sistema.

Os outros são módulos de integração com diversas outras tecnologias que o Spring suporta, tais como:

Módulo AOP

- Spring AOP
- Integração com AspectJ

Módulo DAO

- Spring JDBC
- Gerenciamento de transações

Módulo ORM

- Hibernate
- JPA
- TopLink
- JDO
- OJB
- iBatis

Módulo Java EE

- JMX
- JMS
- JCA
- Remoting
- EJBs
- Email

Módulo Web

- Spring Web MVC
- Integração com Struts
- Integração com WebWork
- Integração com Tapestry
- Integração com JSF
- Integração com Rich View Support
- Integração com JSPs
- Integração com Velocity
- Integração com FreeMarker
- Integração com PDF

- Integração com JasperReports
- Integração com Excel
- Integração com Spring Portlet MVC

3.3 - Introdução aos beans do Spring

Para entendermos o funcionamento do Spring, temos que entender o conceito de **Bean** que ele usa. Para o Spring, qualquer objeto da sua aplicação e que está sob o gerenciamento do Spring é considerado um *bean*.

Um *bean* nada mais é do que uma classe Java que estará sob o gerenciamento do Spring, o container de IoC/DI é o responsável pelo gerenciamento deste bean, bem como suas dependências.

3.4 - Configurando o Spring

Configurando os beans

As configurações do Spring acontecem principalmente em um arquivo XML chamado **applicationContext.xml**. No Spring 2.5, foi adicionado o suporte a anotações, mas muitas tarefas ainda precisam do XML e ele ainda é usado em muitos projetos. Por isso, vamos primeiro ver como fazer as configurações dos nossos beans via XML antes de ver a abordagem com anotações.

Veja a configuração de um bean no XML do Spring:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="produtoDao" class="br.com.caelum.estoque.dao.ProdutoMemoriaDAO" />

</beans>
```

Tag beans

A tag `<beans>` é o elemento root do nosso arquivo XML. Dentro dela teremos as tags `<bean>` que representam a configuração dos objetos do nosso sistema na forma de bean para o Spring.

Na tag `<bean>`, temos duas propriedades principais:

- **id** - representa o identificador do nosso bean, o Spring não aceita id's repetidos;
- **class** - representa o FQN da nossa classe.

FQN

FQN significa **Fully-Qualified Name**, ou nome completo de uma classe. O FQN de uma classe é o nome da classe com o seu pacote completo. Por exemplo, o FQN da classe `Object` é `java.lang.Object`, o FQN da interface `List` é `java.util.List`.

3.5 - Iniciando o container

Após feita a configuração dos nossos **beans** no Spring, precisamos iniciar o container de IoC. Esta inicialização é muito simples e pode ser feita de diversas formas. Quando trabalhamos dentro de um container Web com algum framework MVC, essa inicialização costuma ser automática com apenas algumas configurações.

Mas quando estamos trabalhando direto no `main`, sem container, precisamos iniciar o Spring explicitamente. Podemos fazer isso usando a classe `ApplicationContext`:

```
ApplicationContext context =  
    new ClassPathXmlApplicationContext("classpath:applicationContext.xml");
```

Criado o contexto do Spring, podemos pedir uma instância de qualquer bean registrado lá. Invocando o framework diretamente, basta passar o id do componente:

```
ProdutoDAO dao = (ProdutoDAO) context.getBean("produtoDao");
```

A partir da versão 3 do Spring podemos utilizar um segundo parâmetro do método `getBean` para definir o tipo do bean. Assim nem é preciso fazer o cast:

```
ProdutoDAO dao = context.getBean("produtoDao", ProdutoDao.class);
```

Também podemos descobrir o bean apenas pelo tipo (e não pela id):

```
ProdutoDAO dao = context.getBean(ProdutoDao.class);
```

No exemplo acima deve existir apenas um bean compatível com a interface `ProdutoDao`. Caso contrário o Spring cria um exceção.

3.6 - Injetando as Dependências: por construtor e por setter

Como vimos anteriormente, um dos principais recursos do Spring é o gerenciamento automático das dependências dos beans. Mas, por padrão, a amarração (**wiring**) das dependências não é feita de forma automática. Por isso, somos obrigados a configurá-las uma por uma no arquivo XML.

Para fazermos essa a amarração das dependências com o Spring, temos duas opções: por construtor ou por métodos `setter`.

Injetando dependência por construtor

Na classe `GerenciadorDeProduto` que fizemos antes, declaramos a dependência com `ProdutoDAO` no construtor. É a forma comumente preferida hoje em dia e o Spring a suporta sem problemas.

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       xsi:schemaLocation="  
           http://www.springframework.org/schema/beans  
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
```

```
<bean id="gerenciadorProduto" class="br.com.caelum.estoque.GerenciadorDeProduto" >
    <constructor-arg ref="produtoDao" />
</bean>

<bean id="produtoDao" class="br.com.caelum.estoque.dao.ProdutoMemoriaDAO" />

</beans>
```

Injetando dependência por setter

Poderíamos ter escrito nossa classe `GerenciadorDeProduto` para receber a instância do DAO que ela precisa através de um método `setProdutoDao` ao invés de receber no construtor.

Essa foi a forma preferida e incentivada no Spring durante muito tempo. No XML, a configuração é um pouco diferente:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="gerenciadorProduto" class="br.com.caelum.estoque.GerenciadorDeProduto" >
        <property name="produtoDao" ref="produtoDao" />
    </bean>

    <bean id="produtoDao" class="br.com.caelum.estoque.dao.ProdutoMemoriaDAO" />

</beans>
```

3.7 - Exercícios: Projeto com Spring IoC

- 1)
 - Vá em **File -> Import**
 - Selecione **General, Existing Projects into workspace**. Clique Next
 - Selecione **Archive File** e indique o arquivo **/caelum/zips/27/fj27-projeto-spring.zip**
- 2) Crie a classe `Produto` no pacote `br.com.caelum.estoque`:

```
package br.com.caelum.estoque;

public class Produto {

    private Long id;
    private String descricao;
    private Integer quantidade;

    //getters e setters
}
```

- 3) Crie a interface `ProdutoDAO` no pacote `br.com.caelum.estoque.dao`

```
package br.com.caelum.estoque.dao;

public interface ProdutoDAO {
    void salvar(Produto produto);
    void alterar(Produto produto);
    List<Produto> listar();
    Produto buscarPorId(Long id);
}
```

- 4) Como ainda não vimos a integração do Spring com Hibernate e nosso objetivo é focar no Spring por enquanto, vamos criar uma implementação de DAO bem simples que apenas adiciona os produtos em uma lista em memória.

Crie a classe `ProdutoMemoriaDAO` no pacote `br.com.caelum.estoque.dao` com a implementação simples do DAO em memória. Ainda não implementaremos o método de alteração:

```
package br.com.caelum.estoque.dao;

public class ProdutoMemoriaDAO implements ProdutoDAO {

    private List<Produto> produtos = new ArrayList<Produto>();

    public List<Produto> listar(){
        return produtos;
    }

    public void salvar(Produto produto) {
        produtos.add(produto);
    }

    public Produto buscarPorId(Long id) {
        return produtos.get(id.intValue() - 1);
    }

    public void alterar(Produto produto) {
    }
}
```

- 5) Vamos criar agora nossa classe `GerenciadorDeProduto`. Mais tarde, vamos transformá-la em um componente de um sistema Web usando Spring MVC. Por isso, vamos fazê-la um pouco diferente da que vimos antes.

Ao invés de mostrar uma mensagem de confirmação para o usuário, vamos apenas retornar uma String de confirmação para ele. E o objeto `Produto` que pediríamos para o usuário preencher com os dados, vamos receber como argumento no método.

Crie a classe `GerenciadorDeProduto` no pacote `br.com.caelum.estoque`:

```
package br.com.caelum.estoque;

public class GerenciadorDeProduto {
    private ProdutoDAO produtoDao;

    public GerenciadorDeProduto(ProdutoDAO produtoDao) {
        this.produtoDao = produtoDao;
    }

    public void novoProduto(Produto produto) {
        System.out.println("Salvando Produto.");
        this.produtoDao.salvar(produto);
    }

    public List<Produto> getProdutos(){
        return produtoDao.listar();
    }
}
```

Repare que, como estamos usando DI, declaramos a dependência com o DAO no construtor.

6) Fazemos agora criar toda a configuração do Spring com os beans que acabamos de fazer.

a) Abra o arquivo chamado **applicationContext.xml** que se encontra no diretório **src** do seu projeto. Este é o arquivo de configuração do Spring.

b) Coloque a configuração entre a tag `<beans></beans>`:

```
<bean id="gerenciadorProduto" class="br.com.caelum.estoque.GerenciadorDeProduto" >
    <constructor-arg ref="produtoDAO"/>
</bean>

<bean id="produtoDAO" class="br.com.caelum.estoque.dao.ProdutoMemoriaDAO" />
```

7) Para testarmos a injeção dos componentes que temos até agora, vamos utilizar uma classe simples com método `main` chamada `TesteInjecao`, que já existe em seu projeto.

Nela, criamos o `ApplicationContext` do Spring e, logo em seguida, pediremos uma instância do nosso gerenciador de produtos. Queremos obter essa instância sem precisar satisfazer as dependências dela.

Abra a classe `TesteInjecao` que está no pacote `br.com.caelum.estoque`:

```
package br.com.caelum.estoque;

public class TestaInjecao {
    public static void main (String[] args) {
        // cria o contexto do Spring
        ApplicationContext context =
            new ClassPathXmlApplicationContext("applicationContext.xml");

        // obtém o gerenciador
        GerenciadorDeProduto gerenciador =
            context.getBean("gerenciadorProduto", GerenciadorDeProduto.class);

        // cria um Produto como se o usuário tivesse preenchido um formulário
        Produto produto = new Produto();
        produto.setDescricao("Livro Spring in Action");
        produto.setQuantidade(10);

        // chama a lógica e vê o resultado
        gerenciador.novoProduto(produto);

        // verifica que o produto foi adicionado ao nosso dao em memória
        for (Produto p : gerenciador.getProdutos()) {
            System.out.println("Descrição: " + p.getDescricao());
            System.out.println("Quantidade: " + p.getQuantidade());
        }
    }
}
```

Execute essa classe como um programa Java e veja o resultado.

3.8 - Exercícios opcionais: Injeção com Setter e Tipos Básicos

- 1) Vamos agora na nossa classe GerenciadorDeProduto injetar o ProdutoDAO através do *setter*.

Na classe GerenciadorDeProduto **comente** o construtor e gere um setter para receber o ProdutoDAO:

```
public class GerenciadorDeProduto {
    private ProdutoDAO produtoDao;

    //comente o construtor e gere o setter
    public void setProdutoDAO(ProdutoDAO produtoDao) {
        this.produtoDao = produtoDao;
    }

    public String novoProduto(Produto produto) {
        this.produtoDao.salvar(produto);
        return "salvo";
    }

    public List<Produto> getProdutos() {
        return produtoDao.listar();
    }
}
```

Repare que ainda estamos usando DI, mas declaramos agora a dependência no setter.

- 2) Falta alterar a configuração do Spring para definir a injeção pelo setter.

- a) Procure a declaração do bean **gerenciadorProduto** no arquivo **applicationContext.xml**.
- b) Tire a injeção pelo construtor, coloque a pelo setter:

```
<bean id="gerenciadorProduto" class="br.com.caelum.estoque.GerenciadorDeProduto">
    <property name="produtoDAO" ref="produtoDAO"/>
</bean>

<bean id="produtoDAO" class="br.com.caelum.estoque.dao.ProdutoMemoriaDAO" />
```

- 3) Rode o TestaInjecao.

3.9 - Exercícios Opcionais: Valores Literais

- 1) Vamos criar uma classe para testar a injeção de valores literais e referências para outros beans.

Cria uma classe SpringBean dentro do pacote `br.com.caelum.estoque`:

```
package br.com.caelum.estoque;

public class SpringBean { }
```

- 2) Na classe SpringBean inclua os seguintes atributos (Properties e List vem do pacote `java.util`):

```
private String nome;
private Integer quantidade;
private Properties propriedades;
private List<String> nomes;
```

Gere os getters e setters para todos os atributos.

- 3) Agora, vamos injetar os 5 tipos acima no bean do Spring:

Declare o bean XML.

- Abra o arquivo **applicationContext.xml** e adicione os seguintes *:properties* :

```
<bean id="springBean" class="br.com.caelum.estoque.SpringBean" >

    <property name="nome" value="Caelum" />

    <property name="quantidade" value="10" />

    <property name="propriedades">
        <props>
            <prop key="cidade">São Paulo</prop>
            <prop key="pais">Brasil</prop>
        </props>
    </property>

    <property name="nomes">
        <list>
```

```
        <value>Jose</value>
        <value>Joao</value>
    </list>
</property>

</bean>
```

- 4) Vamos criar uma classe para testar as injecões: Crie a classe `TesteInjecao2` no pacote `br.com.caelum.estoque`.

```
1 public class TesteInjecao2 {
2
3     public static void main(String[] args) {
4
5         ApplicationContext applicationContext =
6             new ClassPathXmlApplicationContext("applicationContext.xml");
7         SpringBean springBean =
8             (SpringBean) applicationContext.getBean("springBean");
9
10        if (springBean != null) {
11
12            System.out.println("instanciado!");
13            System.out.println(springBean.getNome());
14            System.out.println(springBean.getQuantidade());
15            System.out.println(springBean.getPropriedades().get("cidade"));
16
17            for (String string : springBean.getNomes()) {
18                System.out.println(string);
19            }
20        }
21    }
22 }
```

Rode a classe e verifique na saída do console que todas os atributos foram injetados pelo Spring.

Spring MVC

"O verdadeiro discípulo é aquele que supera o mestre."

— Aristóteles

4.1 - Spring MVC

O padrão de projeto MVC é a forma mais utilizada para o desenvolvimento de aplicações Web. A divisão entre as camadas *Model* e *View* através do intermediário *Controller* simplifica a evolução da aplicação e também a manutenção de código existente.

Existem várias implementações de frameworks MVC no mercado. Entre os mais famosos temos o *JavaServer Faces* e o *Struts*. Nos módulos do Spring, encontramos também um framework MVC que traz aos desenvolvedores as vantagens citadas acima, mas também, integração simples com os outros serviços de infraestrutura do Spring, como DI, AOP e Transações.

4.2 - Criando seu projeto web no Eclipse

Vamos importar um novo projeto para nosso Eclipse com toda a estrutura de um projeto Web já pronto. Além disso, os jars do Spring e as classes que fizemos até agora também serão incluídas.

Vamos usar como container Web o Apache Tomcat 6, que pode ser baixado em <http://tomcat.apache.org>

Instalando e configurando o Tomcat

- Entre na pasta **caelum** no seu Desktop e vá na pasta **21**
- Dê dois cliques para abrir o arquivo **apache-tomcat-6.x.x.zip**
- Clique em **Extract** e selecione o **Desktop**
- Volte ao Eclipse e clique com o botão direito na aba **Server**
- Selecione **New -> Server**
- Adicione um Servidor da pasta **Apache Tomcat** versão 6.X
- Clique em **Finish**
- Inicie o Tomcat pelo ícone da barra de ferramentas

Importando o projeto

- Vá em **File -> Import**
- Selecione **General, Existing Projects into workspace**. Clique Next
- Selecione **Archive File** e indique o arquivo **/caelum/zips/27/fj27-projeto-mvc.zip**
- Dê start no Tomcat e teste a url <http://localhost:8080/estoque/>

4.3 - Configurando o Spring MVC

O Spring MVC foi desenvolvido com base na classe `org.springframework.web.servlet.DispatcherServlet` que é um *Servlet* que trata as requisições, redirecionamentos e outras funções que facilitam o uso do framework. Como todo *Servlet*, precisamos configurá-lo no **web.xml** da aplicação. Na configuração do `DispatcherServlet` é importante lembrar de informar o diretório onde está o XML com as configurações dos beans que o Spring vai gerenciar, esse diretório é indicado através do parâmetro `contextConfigLocation`.

Veja como fica o trecho do `web.xml` com a configuração:

```
<servlet>
    <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/spring/app-config.xml
        </param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>
    <url-pattern>*.html</url-pattern>
</servlet-mapping>
```

4.4 - View Resolvers

No XML de configuração do Spring é necessário registrar o `ViewResolver`, que tem como objetivo integrar `Controller` e `View`, deixando-os desacoplados. Com o `View Resolver`, podemos utilizar várias tecnologias diferentes com nossos controllers. Por exemplo, JSP, Velocity ou XSLT.

Na configuração do bean, podemos dizer o `prefix` e `suffix` de onde estarão nossas views. Quando um Controller é executado, ele devolve o nome da View que deve ser renderizada para à `DispatcherServlet`. Ela por sua vez passa o nome para o `ViewResolver` que descobre a view concatenando o nome retornado ao `prefix` e `suffix` configurados.

Vamos usar a seguinte configuração:

```
<bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/" />
```

```
<property name="suffix" value=".jsp" />
</bean>
```

Nesta configuração, se o Controller devolvesse, por exemplo, a string "index", o ViewResolver procuraria a seguinte view para renderizar "WEB-INF/views/index.jsp".

4.5 - Seu primeiro Controller

A partir do Spring 2.5, é possível utilizar anotações para a configuração dos Controllers. Essas anotações trouxeram muitos benefícios para a utilização do Spring MVC como a simplicidade de configuração de objetos para representar a camada Controller.

Entre as anotações que o Spring MVC disponibiliza, temos a `@Controller` que indica que o Bean deve ser gerenciado. E a anotação `@RequestMapping` que informa a URL de acesso ao devido método no controller.

Usando essas duas anotações para criar um controller teríamos:

```
package br.com.caelum.fj27.exemplo;

@Controller
public class MundoController {

    @RequestMapping("ola")
    public String executar {
        System.out.println("Executando o Controller");
        return "ok"
    }
}
```

Para executar o exemplo acima, basta chamar a URL da aplicação terminando em `/ola`. Ex: <http://localhost:8080/spring/ola.html>

O método anotado com `@RequestMapping`, não tem restrição quanto a sua assinatura. A opção de retornar uma String é utilizada para informar qual view deve ser renderizada após a execução do método.

No exemplo anterior, a view renderizada se chama `ok` e terá seu diretório e extensão informadas pelo ViewResolver. Por exemplo: **WEB-INF/views/ok.jsp**.

4.6 - Configurando

A anotação `@Controller` não é lida por default. O Spring sempre usa o XML como padrão e para usarmos classes anotadas, temos que configurar o XML do Spring com a seguinte instrução:

```
<mvc:annotation-driven/>
```

Com isso, o Spring vai reconhecer as classes anotadas com `@Controller`.

Além do elemento para reconhecer as anotações `@Controller`, precisamos informar qual o pacote o Spring deve rastrear os componentes anotados:

```
<context:component-scan base-package="br.com.caelum.estoque" />
```

Lembrando que é necessário configurar o namespace corretamente.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xmlns:context="http://www.springframework.org/schema/context"
xmlns:mvc="http://www.springframework.org/schema/mvc"
xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd
    http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">
<beans>
```

4.7 - Exercícios: Nosso primeiro controller

- 1) Vamos agora, criar a classe `ProdutosController` no pacote `br.com.caelum.estoque.controller`. Iremos também anotá-la com `@Controller` para que o Spring MVC possa reconhecê-la.

```
package br.com.caelum.estoque.controller;

//imports

@Controller
public class ProdutosController {

    @RequestMapping("index")
    public String index() {
        return "produtos/index";
    }

}
```

- 2) Configure o XML do Spring para habilitar o suporte a anotação `@Controller` e informar onde o spring deve procurar por beans anotados. Adicione as seguintes linhas ao seu **app-config.xml** no diretório **WEB-INF/spring/app-config.xml**:

```
<context:component-scan base-package="br.com.caelum.estoque" />

<mvc:annotation-driven />
```

- 3) Falta criar nossa *View*. Crie uma pasta chamada **produtos** em **WEB-INF/views/** e, dentro, uma JSP chamada **index.jsp** com o seguinte conteúdo de teste:

```
<html>
    Spring MVC!
</html>
```

- 4) Inicie o servidor e acesse seu controller chamando a URL `http://localhost:8080/estoque/index.html`

4.8 - Enviando dados para View

Os controllers do Spring podem enviar dados para as view renderizadas. Os métodos do *Controller* que irão devolver dados a tela, devem declarar `ModelAndView` como seu tipo de retorno.

O objeto `ModelAndView` carrega a informação da view a ser renderizada e também dos dados que devem ser disponibilizados na view. Exemplo:

```
@Controller  
public class ProdutosController {  
  
    @RequestMapping("listarProdutos")  
    public ModelAndView listar(){  
        List<Produto> produtos = new ArrayList<Produto>();  
  
        Produto macBook = new Produto();  
        macBook.setDescricao("Mac Book");  
        macBook.setQuantidade(100);  
  
        Produto ipad = new Produto();  
        ipad.setDescricao("IPad");  
        ipad.setQuantidade(200);  
  
        produtos.add(macBook);  
        produtos.add(ipad)  
  
        ModelAndView modelAndView = new ModelAndView("produto/lista");  
        modelAndView.addObject(produtos);  
  
        return modelAndView;  
    }  
}
```

Acessamos o ProdutosController pela URL <http://localhost:8080/estoque/produtos/listarProdutos.html>. Na execução do método `listar()`, criamos uma lista contendo alguns produtos. Depois, criamos um objeto `ModelAndView` e passamos no construtor a página de retorno **lista** que fica dentro do diretório **produto** - “WEB-INF/views/produto/lista.jsp”.

Nossa lista de produtos é então adicionada no objeto `ModelAndView` e o método termina retornando o objeto `modelAndView`.

Agora, podemos acessar a lista de produtos na JSP usando *Expression Language*. É importante saber qual o nome que usamos para o acesso de nossa lista de produtos. O Spring utiliza uma convenção para isso. No nosso exemplo, acessaríamos da seguinte maneira na JSP:

```
<c:forEach items="${produtoList}" var="produto">  
    ${produto.descricao} -  
    ${produto.quantidade} <br>  
</c:forEach>
```

O nome `produtoList` é gerado pelo Spring devido ao nosso `List` ser declarado com um tipo genérico `Produto`: `List<Produto>`.

Se tivermos um `List` de `String` - `List<String>` -, a forma de acesso seria - `${stringList}` -.

4.9 - RequestMapping mais a fundo

`@RequestMapping` permite o mapeamento de urls por *Controller*. Por exemplo, podemos mapear um controller para **/produtos** e então para o acesso a qualquer método mapeado dentro do controller seria necessário a chamada a URL da classe anteriormente. Ex:

```
@Controller  
@RequestMapping(value="/produtos")  
public class ProdutosController {  
  
    @RequestMapping(value="/listar")  
    public ModelAndView listar(){  
        //..  
    }  
  
    @RequestMapping(value="/salvar")  
    public String salvar(){  
  
    }  
}
```

Para acessar o método `listar()` de `ProdutosController` precisamos chamar a seguinte a url que nosso controller foi mapeado `/produtos` e então a url do método `/listar`. Ex: `http://localhost:8080/estoque/produtos/listar.html`

4.10 - O Controller de Produto

O *Controller* de Produto vai acessar a camada DAO que, no momento, é representada com dados em memória. O método `listar()` de `ProdutoMemoriaDAO` retornará uma lista com todos os produtos cadastrados. O *Controller* deve obter esses dados e disponibilizá-los para o view que então vai listá-los em tela.

4.11 - Gerenciamento e Injeção

No último capítulo, mapeamos a classe `ProdutoMemoriaDAO` e informamos que ela era uma dependência de `GerenciadorDeProduto` no XML. O Spring também provê a configuração de componentes e suas dependências fazendo o uso de anotações. Então vamos configurar `ProdutoMemoriaDAO` como um componente para poder injeta-la como dependência de `ProdutosController`.

A anotação `@Component` deve ser utilizada para configurar objetos que o Spring deve gerenciar em seu container.

```
package br.com.caelum.dao;  
  
@Component  
public class ProdutoMemoriaDAO implements ProdutoDAO {  
    //..  
}
```

Essa anotação é equivalente a seguinte linha de XML:

```
<bean name="produtoMemoriaDAO" class="br.com.caelum.dao.ProdutoMemoriaDAO"/>
```

A anotação `@Autowired` deve ser utilizada para indicar uma dependência e que a mesma deve ter seu valor injetado.

```
@Controller  
public class ProdutosController {  
  
    @Autowired  
    private ProdutoDAO produtoDAO;  
}
```

Só podemos injetar beans gerenciados em outros beans também gerenciados. No exemplo acima injetamos o ProdutoDAO no ProdutosController porque ambos são gerenciados pelo Spring as anotações @Component e @Controller tem como objetivo o mesmo, sendo informar o Spring que ele deve gerenciar aqueles objetos.

4.12 - Exercícios - Listando os Produtos

- 1) Adicione a anotação @Component na sua classe ProdutoMemoriaDAO.

```
@Component  
public class ProdutoMemoriaDAO implements ProdutoDAO {  
}
```

- 2) Vamos criar o método que vai listar os produtos em nossa classe ProdutosController. O método listar() vai fazer uma chamada ao ProdutoDAO que vai ser injetado pelo Spring e adicionar seu resultado no objeto ModelAndView.

```
@Controller  
@RequestMapping(value="/produtos")  
public class ProdutosController {  
  
    @Autowired  
    private ProdutoDAO produtoDAO;  
  
    // metodo index()  
  
    @RequestMapping(value="/listar")  
    public ModelAndView listar() {  
        ModelAndView modelAndView = new ModelAndView("produtos/lista");  
        modelAndView.addObject(produtoDAO.listar());  
        return modelAndView;  
    }  
}
```

- 3) Crie um JSP chamado **lista.jsp** dentro do diretório “WEB-INF/views/produtos/lista.jsp” para mostrar os produtos que são disponibilizados pela chamada do método listar() de ProdutosController.

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>  
<html>  
    <head>  
        <title>Lista Produtos</title>  
    </head>  
    <body>  
        <c:forEach items="${produtoList}" var="produto">  
            ${produto.descricao} - ${produto.quantidade}<br/>  
        </c:forEach>  
    </body>  
</html>
```

- 4) Inicie o servidor e teste a listagem <http://localhost:8080/estoque/produtos/listar.html>

4.13 - URLs amigáveis

@RequestMapping pode ser usada também para criação de urls bonitinhos. Supondo que temos o ProdutosController e ele tem um método que deve listar um produto específico de determinado *id*. O método poderia ter a seguinte estrutura:

```
@Controller  
@RequestMapping(value="/produtos")  
public class ProdutosController {  
  
    @RequestMapping(value="/listar")  
    public ModelAndView listar(Long id){  
        //..  
    }  
  
}
```

E poderíamos executar o método através da url “/produtos/listar?idade=22” o parâmetro idade seria passado ao listar() que por sua vez filtraria o produto por seu *id* e então devolveria para a View.

Ultimamente um recurso muito utilizado na web são as urls amigáveis, que são urls mais legíveis ao usuários. Poderíamos implementar nosso método dessa forma:

```
@Controller  
@RequestMapping(value="/produtos")  
public class ProdutosController {  
  
    @RequestMapping(value="/listar/{id}")  
    public ModelAndView listar(@PathVariable Long id){  
        //..  
    }  
  
}
```

Agora podemos executar nosso método através da seguinte url “produtos/listar/22” e assim o método listar() seria executado, o Spring por sua vez iria verificar que informamos que um valor vindo na url seria uma variável {id} e que informamos que o mesmo iria ser passado ao método @PathVariable

4.14 - Melhor uso do HTTP

O uso dos verbos do http corretamente é muito importante e trazem benefícios para nossa aplicação. O GET por exemplo, tem como objetivo buscar informações e por isso seus parâmetros são passados na url. Se utilizarmos isso de maneira correta os usuários da aplicação podem se beneficiar para fazer o bookmark de url de relatórios. Como as informações vão por parâmetros de url, eles podem chama-las todas as vezes sem problemas e também passar para que outras pessoas vejam o mesmo relatório. Podemos informar ao Spring que devido método só pode ser acessado por um determinado verbo. Ex:

```
@Controller  
@RequestMapping(value="/produtos")  
public class ProdutosController {  
  
    @RequestMapping(value="/listar/{id}", method=RequestMethod.GET)  
    public ModelAndView listar(@PathVariable Long id){  
        //..  
    }  
  
}
```

Se tentarmos acessar a url “/produtos/listar/22” utilizando um formulário com o verbo *POST* o spring não permitira devido a url ser ligada a utilização do verbo *GET*.

4.15 - Exercícios: Melhor uso do http

- Vamos criar o método `mostrar()` em `ProdutosController`. Nele vamos usar o recurso de Url Friendly e receber o id do Produto detalhado na url. Iremos também colocar ambos métodos `mostrar()` e `listar()` para apenas responderem ao verbo *GET* do http porque ambos são utilizados para pesquisa.

```
package br.com.caelum.estoque.dao;

// imports

@Controller
@RequestMapping(value="/produtos")
public class ProdutosController {

    @Autowired
    private ProdutoDAO produtoDAO;

    //metodo index()

    @RequestMapping(value="/listar", method=RequestMethod.GET)
    public ModelAndView listar() {
        ModelAndView modelAndView = new ModelAndView("produtos/lista");
        modelAndView.addObject(produtoDAO.listar());
        return modelAndView;
    }

    @RequestMapping(value="/mostrar/{id}" , method=RequestMethod.GET)
    public ModelAndView mostrar(@PathVariable Long id){
        ModelAndView modelAndView = new ModelAndView("produtos/mostrar");
        modelAndView.addObject(produtoDAO.buscarPorId(id));
        return modelAndView;
    }
}
```

- Precisamos também, criar a jsp que vai mostrar o Produto detalho. O nome da jsp é `mostrar` e deve ser criada no diretório “WEB-INF/views/produtos”.

```
<html>
    <head>
        <title>Detalhes do Produto</title>
    </head>
    <body>
        Id : <input value="${produto.id}" disabled="disabled"/><br>
        Descrição : <input value="${produto.descrição}" disabled="disabled"/><br>
        Quantidade : <input value="${produto.quantidade}" disabled="disabled"/><br>
    </body>
</html>
```

- A listagem de produtos deve ser alterada para ter um link que vai chamar o método `mostrar`, para o Produto ser detalhado.

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
    <head>
        <title>Lista Produtos</title>
    </head>
    <body>
        <c:forEach items="${produtoList}" var="produto">
            ${produto.descricao} - ${produto.quantidade} -
            <a href="/estoque/produtos/mostrar/${produto.id}.html">detalhes</a><br/>
        </c:forEach>
    </body>
</html>
```

4) Reinicie o servidor de aplicação e teste a página de detalhes.

4.16 - Recebendo parâmetros de Formulario

Um grande poder dos frameworks MVC atuais é a capacidade de converter dados em texto passados como parâmetros via HTTP para objetos do nosso domínio. Se temos o seguinte bean:

```
public class Produto {

    private Long id;
    private String descricao;
    private Integer quantidade;

    // get e set
}
```

O Controller que tem que receber o objeto do tipo `Produto` como parâmetro para o método, deve declarar no parâmetro do método que vai ser chamado.

```
@Controller
@RequestMapping(value="/produtos")
public class ProdutosController{

    @RequestMapping(value="/salvar")
    public String salvar(Produto produto){
        System.out.println(produto.getDescricao());
        System.out.println(produto.getQuantidade());
        return "ok";
    }
}
```

O formulário HTML, que vai fazer a requisição ao método `salvar()` deve passar os parâmetros que devem ser setados no parâmetro do método. O nome dos parâmetros sendo enviado devem seguir o nome do conjunto de getters e setters do parâmetro do método sendo chamado. No nosso exemplo, o parâmetro seria o `Produto` e os getters e setters seriam para as propriedades `descricao` e `quantidade`.

```
<html>
<form action="/estoque/produtos/salvar.html">
    <input type="text" name="descricao"/>
    <input type="text" name="quantidade"/>
```

```
<input type="submit" value="Salvar"/>
</form>
</html>
```

Quando a requisição é feita pela URL `http://localhost:8080/estoque/produtos/salvar.html` o método `salvar()` de `ProdutosController` é executado. O Spring MVC verifica que esse método recebe um parâmetro que é do tipo `Produto`. Então o Spring MVC procura na requisição parâmetros que tenham o nome de alguma das propriedades do bean.

No exemplo acima, são enviados os parâmetros **descricao** e **quantidade**. O Spring então verifica que nosso bean `Produto` tem duas propriedades com o mesmo nome. Lembrando que propriedades em Java é o conjunto `get` e `set` para um atributo, no exemplo `getDescricao()`, `setDescricao()` e `getQuantidade()`, `setQuantidade()`.

4.17 - Forward X Redirect

Toda execução de método em algum *Controller* pode devolver *String* ou *ModelAndView* que informam qual view deve ser renderizada. Podemos também não dizer o nome de uma view mas a url de outro método em um *Controller*.

Por exemplo:

```
@Controller
@RequestMapping(value="/produtos")
public class ProdutosController {

    @RequestMapping(value="/index")
    public String index(){
        return "produtos/listar.html"
    }

    @RequestMapping(value="/listar")
    public ModelAndView listar(){
        //....
    }
}
```

No exemplo acima ao chamar **/index** o Spring faz o forward para **/produtos/listar** executando então o método `listar()`. O forward é o comportamento padrão do spring mas podemos também fazer o *redirect* da ação.

```
@Controller
@RequestMapping(value="/produtos")
public class ProdutosController {

    @RequestMapping(value="/index")
    public String index(){
        return "redirect:produtos/listar.html"
    }

    @RequestMapping(value="/listar")
    public ModelAndView listar(){
        //....
    }
}
```

4.18 - Exercícios - Cadastro de Produto

- 1) Crie a jsp **form.jsp** no diretório “WEB-INF/views/produtos” para o cadastro de produtos.

```
<html>
    <head>
        <title>Cadastro de Produtos</title>
    </head>
    <body>
        <form action="/estoque/produtos/salvar.html" method="post">
            Descricao: <input type="text" name="descricao"/><br/>
            Quantidade: <input type="text" name="quantidade"/><br/>
            <input type="submit" value="Cadastrar">
        </form>
    </body>
</html>
```

- 2) Em **ProdutosController** vamos criar o método **form()**, que quando chamado direciona o usuário para a página de cadastro. E o método **salvar()** que recebe o **Produto** que está sendo cadastrado.

```
@Controller
@RequestMapping(value="/produtos")
public class ProdutosController {

    @Autowired
    private ProdutoDAO produtoDAO;

    // index()

    //listar()

    //mostrar()

    @RequestMapping(value="/form" , method=RequestMethod.GET)
    public String form() {
        return "produtos/form";
    }

    @RequestMapping(value="/salvar" , method=RequestMethod.POST)
    public String salvar(Produto produto) {
        produtoDAO.salvar(produto);
        return "redirect:/produtos/listar.html";
    }
}
```

- 3) Inicie o servidor e teste o seu cadastro <http://localhost:8080/estoque/produtos/form.html>

4.19 - Escopos

Perceba que ao injetar **ProdutoMemoriaDAO** na classe **ProdutosController** sempre temos o mesmo objeto em memória. Só perdemos os dados cadastrados quando reiniciamos nosso servidor de aplicação. Isso acontece devido aos escopos do Spring.

Escopo Singleton

O escopo do tipo *singleton* define que será criado uma única instância do bean para todas requisições recebidas pelo Spring para aquele bean. O Spring garante que esta instância será única para cada instância do IoC container do Spring. Se tivermos um único BeanFactory ou ApplicationContext, teremos uma única instância de um bean do tipo *singleton*. *singleton* é o padrão.

Escopo Prototype

O escopo do tipo *prototype* define que será criada uma nova instância do bean para cada requisição recebida pelo Spring para aquele bean.

4.20 - Web Escopos: Request, Session, Global Session

Além dos escopos *singleton* e *prototype*, o Spring disponibiliza mais três escopos de uso específico em aplicações web: *request*, *session* e *global_session*.

Escopo Request

O escopo do tipo *request* define que o ciclo de vida do bean será atrelado a um Request HTTP. O Spring garante que cada request possui sua própria instância do bean. Ao final do request, o objeto é descartado.

Escopo Session

O escopo do tipo *session* gerencia o ciclo de vida do bean baseado na HttpSession do usuário.

Escopo Global Session

O escopo do tipo *global session* gerencia o ciclo de vida do bean baseado na HttpSession do usuário. Este escopo é semelhante ao escopo de *session* visto anteriormente, com a diferença que ele pode ser lido em outros contexto para aplicação que usam a tecnologia de Portlets.

Integrando Spring, Hibernate e tecnologias de Persistência

5.1 - Introdução - Onde o Spring pode ajudar ao usar o Hibernate

Dentre as diversas características do Spring uma das que mais chama a atenção é a integração nativa com diversas tecnologias importantes do mercado, dentre eles está o **hibernate**, a principal ferramenta ORM para Java.

Mas como o Spring ajuda a trabalhar com o Hibernate?

De diversas maneiras! A primeira delas é através da principal funcionalidade do Spring, o controle de objetos. O Spring pode controlar a abertura e fechamento da SessionFactory do Hibernate, com isso **todos** os componentes do Spring podem receber como dependência a SessionFactory, agora, controlada pelo Spring.

Outra parte importante do Spring é o controle transacional que ele oferece. Ao delegar o controle da SessionFactory para o Spring, ele consegue abrir e fechar transações de maneira automática, de maneira muito parecida com o JTA.

Além de se integrar ao Hibernate o Spring também pode se integrar à JPA, JDBC e outras ferramentas de persistência.

5.2 - Configurando o Hibernate no Spring

Para funcionar a integração com o Hibernate o Spring disponibiliza uma *Factory Bean* para a SessionFactory do Hibernate. Utilizando esta *factory* fazemos com que a SessionFactory do hibernate seja gerenciada pelo Spring.

AnnotationSessionFactoryBean

A *Factory Bean* que o Spring possui é a `org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean`. Esta classe deve ser configurada como um bean do Spring em seu xml de configuração.

A classe `AnnotationSessionFactoryBean` nos permite configurar todas as propriedades do Hibernate diretamente no arquivo do Spring, não necessitando o uso do arquivo de configuração do próprio do Hibernate `hibernate.cfg.xml`.

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean"
    destroy-method="destroy">

    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.connection.url">jdbc:mysql://localhost/fj27</prop>
            <prop key="hibernate.connection.driver_class">com.mysql.jdbc.Driver</prop>
            <prop key="hibernate.connection.username">root</prop>
        </props>
    </property>

```

```
<prop key="hibernate.connection.password"></prop>
<prop key="hibernate.connection.autocommit">true</prop>
<prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
<prop key="hibernate.show_sql">true</prop>
<prop key="hibernate.format_sql">true</prop>
<prop key="hibernate.hbm2ddl.auto">update</prop>
</props>
</property>
<property name="annotatedClasses">
    <list>
        <value>br.com.caelum.estoque.model.Produto</value>
    </list>
</property>
</bean>
```

Ao integrarmos o Hibernate ao Spring, a configuração do Hibernate, que antes era feita separadamente, passa a ser feita diretamente no Spring. Isso é necessário para deixarmos o Spring gerenciar a criação da SessionFactory do Hibernate permitindo assim que o Spring faça a injecão automática da SessionFactory do Hibernate.

destroy-method

A propriedade *destroy-method* da tag **bean** é o nome do método que será invocado sempre que o Spring finalizar o ciclo de vida da nossa **SessionFactory**.

hibernate.cfg.xml

Ao invés de configurar o hibernate direto no Spring, podemos manter o arquivo hibernate.cfg.xml separado e apenas referenciá-lo no Spring:

```
<property name="configLocation">
    <value>
        classpath:location_of_config_file/hibernate.cfg.xml
    </value>
</property>
```

5.3 - ProdutoHibernateDAO

Agora podemos criar um novo DAO que realmente acesse o banco de dados. Para isso basta pedir para o Spring injetar uma *SessionFactory* em alguma classe. Lembrando que o Spring injeta apenas beans gerenciados em outros também gerenciados, para isso devemos declarar que nosso DAO deve ser gerenciado pelo Spring anotando-o com `@Component`.

```
@Component
public class ProdutoHibernateDAO implements ProdutoDAO {

    private Session session;

    @Autowired
    public ProdutoHibernateDAO(SessionFactory factory) {
        session = factory.openSession();
    }

    public List<Produto> listar(){
        List<Produto> produtos = session.createQuery("from Produto").list();
        return produtos;
    }

    public void salvar(Produto produto) {
        session.save(produto);
    }

    public Produto buscar(Long id) {
        return session.get(Produto.class, id);
    }

    public void alterar(Produto produto) {
        session.update(produto);
    }

}
```

5.4 - @Repository

Anteriormente comentamos que as anotações `@Controller` e `@Component` tinham o mesmo objetivo, indicar ao Spring que os beans anotados devem ser gerenciados e que a diferença entre as mesmas era apenas semântica. Para o caso dos DAOs temos a mesma situação, podemos anotar nossos DAOs com `@Repository`, ela terá o mesmo efeito da `@Component`.

5.5 - Exercícios

- 1) Faça o mapeamento da classe Produto.

```
package br.com.caelum.estoque;

@Entity
public class Produto {

    @Id
    @GeneratedValue
    private Long id;
    private String descricao;
    private Integer quantidade;

    //getters e setters
}
```

- 2) Abra o arquivo `app-config.xml` no diretório “WEB-INF/spring”, e adicione as seguintes linhas

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.connection.url">jdbc:mysql://localhost/fj27</prop>
            <prop key="hibernate.connection.driver_class">com.mysql.jdbc.Driver</prop>
            <prop key="hibernate.connection.username">root</prop>
            <prop key="hibernate.connection.password"></prop>
            <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.format_sql">true</prop>
            <prop key="hibernate.hbm2ddl.auto">update</prop>
            <prop key="hibernate.connection.autocommit">true</prop>
        </props>
    </property>
    <property name="annotatedClasses">
        <list>
            <value>br.com.caelum.estoque.Produto</value>
        </list>
    </property>
</bean>
```

- 3) Crie a classe ProdutoHibernateDAO no pacote br.com.caelum.estoque.dao:

```
@Repository
public class ProdutoHibernateDAO implements ProdutoDAO {

    private Session session;

    @Autowired
    public ProdutoHibernateDAO(SessionFactory factory) {
        session = factory.openSession();
    }

    public List<Produto> listar(){
        List<Produto> produtos = session.createQuery("from Produto").list();
        return produtos;
    }

    public void salvar(Produto produto) {
        session.save(produto);
    }

    public Produto buscarPorId(Long id) {
        return (Produto) session.get(Produto.class, id);
    }

    public void alterar(Produto produto) {
        session.update(produto);
    }
}
```

- 4) Reinicie o servidor e tente fazer o cadastro de algum produto. **Não funcionará**. Leia a exceção e sua causa raiz (*root cause*). O que aconteceu?

5.6 - @Qualifier

Ao testar o sistema o Spring lançou uma exceção na nosso console. Isso aconteceu porque o Spring descobre o que injetar nos *beans* através do **tipo** do objeto a ser injetado. No nosso caso o Spring tenta injetar algum *bean* que implemente a interface ProdutoDAO, porém existem duas implementações, a ProdutoMemoriaDAO e a ProdutoHibernateDAO, com isso o Spring se perde na hora de fazer a injeção.

Neste nosso caso poderíamos simplesmente apagar a ProdutoMemoriaDAO que resolveria o problema, porém em cenários que realmente devam existir duas implementações devemos especificar qual delas queremos receber como injeção. É exatamente esse o objetivo da anotação **@Qualifier**.

Com o **@Qualifier** podemos especificar através do id do *bean* qual implementação queremos receber.

```
public class ProdutoController {  
  
    @Autowired  
    @Qualifier("produtoHibernateDAO")  
    private ProdutoDAO produtoDAO;  
  
    //código antigo  
}
```

Indicamos o nome **produtoHibernateDAO** porque o Spring configura o id dos *beans* anotados através do nome da classe. A sua política é se o nome não foi passado diretamente via uma das anotações **@Controller**, **@Component** e **@Repository** ele simplesmente coloca o nome da classe anotada transformando a primeira letra do nome em minúscula. Neste caso ProdutoHibernateDAO fica **produtoHibernateDAO**.

Fora a facilidade proporcionada pelo Spring, também tiramos proveito do uso do polimorfismo, que permitiu uma troca tranquila da implementação.

5.7 - Exercícios

- 1) Altere sua classe controller para receber o ProdutoHibernateDAO ao invés do ProdutoMemoriaDAO

```
@Controller  
@RequestMapping(value="/produtos")  
public class ProdutosController {  
  
    @Autowired  
    @Qualifier("produtoHibernateDAO")  
    private ProdutoDAO produtoDAO;  
  
    //código antigo  
}
```

- 2) Cadastre alguns produtos.

5.8 - Não quero escolher um objeto, quero um padrão!

Embora possamos resolver a ambiguidade da injeção através do **@Qualifier**, muitas vezes um *bean* é muito mais injetado que o outro. Nesse cenário o Spring oferece uma anotação que marca um *bean* como sendo o padrão a ser injetado a **@Primary**. Caso não seja usado nenhum **@Qualifier** o *bean* anotado com **@Primary** será o injetado.

5.9 - Exercícios

1) Anote a classe ProdutoHibernateDAO com @Primary:

```
@Repository  
@Primary  
public class ProdutoHibernateDAO implements ProdutoDAO{  
    //codigo ja existente  
}
```

2) Retire o @Qualifier do atributo produtoDAO da classe ProdutoController, deixando apenas a anotação @Autowired:

```
public class ProdutoController{  
  
    @Autowired  
    ProdutoDAO produtoDAO;  
}
```

5.10 - Transações com Spring - Spring Transaction

Além do gerenciamento da SessionFactory o Spring também pode controlar a abertura e fechamento de transações.

O controle de transação pode ser feito via xml ou via anotação.

O mais simples é via anotação, porém, por motivos de referência segue abaixo um exemplo de como o controle de transação pode ser feito via xml:

```
<bean id="produtosController" class="br.com.caelum.estoque.controller.ProdutosController"/>  
  
<tx:advice id="txAdvice" transaction-manager="txManager">  
    <tx:attributes>  
        <tx:method name="*"/>  
    </tx:attributes>  
</tx:advice>  
  
<aop:config>  
    <aop:pointcut id="metodosProdutosController"  
        expression="execution(* br.com.caelum.estoque.controller.ProdutosController.*(..))"/>  
    <aop:advisor advice-ref="txAdvice" pointcut-ref="metodosProdutosController"/>  
</aop:config>
```

A configuração parece complicada pois é dividida em duas partes e usa orientação a aspectos. No trecho `<tx:advice>` é configurado quais métodos serão *"interceptados"* pelo controle de transação. No trecho `<aop:config>` habilitamos o AOP, para que ele possa cortar nossos métodos e inserir outros comportamentos.

A configuração por anotação é mais simples. Apenas precisamos indicar para o Spring quais métodos ele deve abrir e fechar uma transação. Isso é feito usando a anotação `@Transaction` em um método, ou mesmo em uma classe, que tem o mesmo efeito de anotar todos os métodos da classe. Podemos então anotar nossos DAO's com o `@Transaction` e assim ter suporte a transação.

```
@Repository  
@Transactional  
public class ProdutoHibernateDAO implements ProdutoDAO {  
    //codigo  
}
```

Porém até o momento nosso sistema não fez o uso de transações. O hibernate está configurado com o autocommit ativado, dando commit em todas nossas operações automaticamente. Também temos que lembrar que o tipo de tabela utilizada no MySQL não da suporte a transações e por isso apenas o flush do autocommit basta para os dados serem persistidos.

O MySQL tem dois tipos de tabela, MyISAM e InnoDB. MyISAM que é a *default* no Ubuntu, não oferece suporte a transações e por isso não tivemos problemas até o momento. Para testarmos o controle de transações vamos alterar o tipo de tabela do MySQL, mudando o dialeto do Hibernate.

Alterar:

```
<prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
```

Para :

```
<prop key="hibernate.dialect">org.hibernate.dialect.MySQL5InnoDBDialect</prop>
```

E também remover a linha do autocommit:

```
<prop key="hibernate.connection.autocommit">true</prop>
```

5.11 - Configurando o TransactionManager

Para o suporte a transação ser ativado precisamos fazer duas configurações no xml do Spring.

A primeira é habilitar o gerenciador de transação. Porém como o controle do Spring pode ser para JPA, Hibernate, JDBC e outros, devemos configurar o gerenciador exatamente para uma dessas tecnologias. No caso do hibernate a única dependência que o TransactionManager precisa é uma SessionFactory.

```
<bean id="transactionManager"  
      class="org.springframework.orm.hibernate3.HibernateTransactionManager">  
    <property name="sessionFactory" ref="sessionFactory"></property>  
</bean>
```

A segunda é parte é avisar que o controle de transações será feito via anotação, parecido com a forma de habilitar o uso de anotações para o Spring MVC.

```
<tx:annotation-driven/>
```

5.12 - Controle de Session com o Spring

Para o Spring poder abrir transações com o Hibernate é preciso que ele abra uma Session, para depois fazer o begin da transação, com isso, para que nossos dados sejam de fato committedos precisamos usar esta mesma Session que o Spring abre para fazer o controle da transação. Como consegui-la?

Uma vez que o Spring já abriu a sessão basta pedir para a SessionFactory pegar a session já aberta através do método `getCurrentSession()`.

```
@Repository  
@Transactional  
public class ProdutoHibernateDAO implements ProdutoDAO {  
  
    private SessionFactory sessionFactory;  
  
    @Autowired  
    public ProdutoHibernateDAO(SessionFactory factory) {  
        this.sessionFactory = factory;  
    }  
  
    public void salvar(Produto produto) {  
        this.sessionFactory.getCurrentSession().save(produto);  
    }  
  
    //outros métodos  
}
```

Obter a session através do getCurrentSession é o recomendado pela documentação do Spring.

5.13 - Exercícios

- 1) Altere o arquivo app-config.xml para que o dialeto do MySQL seja *InnoDB* e com isso tenhamos suporte às transações com o banco de dados manualmente:

```
<prop key="hibernate.dialect">org.hibernate.dialect.MySQL5InnoDBDialect</prop>
```

- 2) Delete a linha que configura o autocommit:

```
<prop key="hibernate.connection.autocommit">true</prop>
```

- 3)
 - Tente cadastrar um produto novo

- Entre no MySQL para conferir a inserção do dado:
 - Abra o terminal do Linux
 - digite o comando

```
mysql -u root  
– agora dispare um select na tabela de Produtos
```

```
use fj27;  
select * from Produto;
```

- Repare que o produto não está lá!

Tabelas antigas

Se o produto foi inserido, provavelmente é porque a tabela ainda está no formato *MyISAM*, mude-a para *InnoDB* com o comando: alter table Produto type=innodb;

- 4) Adicione as seguintes linhas ao seu **app-config.xml**:

```
<bean id="transactionManager"
      class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>

<tx:annotation-driven/>
```

- 5) Altere sua classe ProdutoHibernateDAO para utilizar transações:

```
@Repository
@Transactional
public class ProdutoHibernateDAO implements ProdutoDAO {
    //codigo
}
```

- 6) **(Importante)** Altere a classe ProdutoHibernateDAO para obter a Session através da chamada sessionFactory.getCurrentSession().
- 7) Cadastre um novo Produto. E repare que agora o Produto tem o commit efetivado no Banco.

5.14 - Exercícios - Alterar Produtos

- 1) Vamos adicionar a ProdutosController os métodos editar() e alterar().

```
@Controller
@RequestMapping(value="/produtos")
public class ProdutosController {

    //código antigo

    @RequestMapping(value="/alterar" , method=RequestMethod.POST)
    public String alterar(Produto produto) {
        produtoDAO.alterar(produto);
        return "redirect:/produtos/listar.html";
    }

    @RequestMapping(value="/editar", method=RequestMethod.GET)
    public ModelAndView editar(Long id) {
        Produto produto = produtoDAO.buscarPorId(id);
        ModelAndView modelAndView = new ModelAndView("produtos/editar");
        modelAndView.addObject(produto);
        return modelAndView;
    }
}
```

- 2) Altere jsp **lista.jsp**, adicionando o link para a edição

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

```
</head>
<body>
    <c:forEach items="${produtoList}" var="produto">
        ${produto.descricao} - ${produto.quantidade} -
        <a href="/estoque/produtos/mostrar/${produto.id}.html">detalhes</a>
        - <a href="/estoque/produtos/editar.html?id=${produto.id}">editar</a> <br/>
    </c:forEach>
</body>
</html>
```

- 3) Crie a jsp **editar.jsp** no diretório “WEB-INF/views/produtos/”

```
<html>
    <head>
        <title>Alterar Produto</title>
    </head>
    <body>
        <form action="/estoque/produtos/alterar.html" method="post">
            <input type="hidden" name="id" value="${produto.id}"/><br/>
            Descricao: <input type="text" name="descricao"
                value="${produto.descricao}"/><br/>
            Quantidade: <input type="text" name="quantidade"
                value="${produto.quantidade}"/><br/>
            <input type="submit" value="Alterar">
        </form>
    </body>
</html>
```

- 4) Reinicie o servidor e altere algum produto.

5.15 - Exercício Opcional: Remoção de produtos

- 1) Para deixar o sistema mais completo, desenvolva a funcionalidade de remoção dos produtos.

5.16 - Aumentando nosso domínio

Nosso sistema de controle de estoque até o momento está bem simples. Apenas cadastrando, listando e alterando produtos. Vamos adicionar agora o conceito de Movimentação. Uma movimentação deve ser gerada para todas as operações realizadas no estoque, ou seja, se aumentarmos a quantidade de produtos em estoque. geramos uma movimentação de entrada, caso diminua, geramos uma movimentação de saída.

5.17 - Entidade

Devido ao nosso novo problema vamos adicionar ao nosso projeto a entidade de Movimentacao. Ele vai representar as movimentações realizadas. Entre seus atributos temos a data que foi efetuada, a quantidade do produto que foi alterada, o tipo da movimentacao que pode ser ENTRADA ou SAIDA e um relacionamento ao Produto em questão.

```
package br.com.caelum.estoque;
@Entity
public class Movimentacao {

    @Id
    @GeneratedValue
    private Long id;
    private Calendar data;
    @Enumerated(EnumType.STRING)
    private TipoMovimentacao tipo;
    private Integer quantidade;
    @ManyToOne
    private Produto produto;

    //getters e setters
}
```

Para facilitar o trabalho, o tipo de movimentação será uma **Enum**.

```
package br.com.caelum.estoque;

public enum TipoMovimentacao {
    ENTRADA, SAIDA
}
```

Porém não basta apenas termos esta nova entidade, precisamos alterar a lógica de alteração de produtos. Nesta nova lógica ao alterar a quantidade de um produto, calcularemos se a movimentação é de saída ou entrada e a quantidade movimentada. Para isso isolaremos esta lógica em um novo código:

```
@Service
public class GeradorDeMovimentacao {

    private final ProdutoDAO produtoDAO;

    @Autowired
    public GeradorDeMovimentacao(@Qualifier("produtoHibernateDAO") ProdutoDAO produtoDAO) {
        this.produtoDAO = produtoDAO;
    }

    public Movimentacao geraMovimentacao(Produto produto) {
        Movimentacao mov = new Movimentacao();
        mov.setData(Calendar.getInstance());
        mov.setProduto(produto);

        Integer quantidadeAtual = produtoDAO.estoqueAtual(produto);
        if(produto.getQuantidade() > quantidadeAtual){
            mov.setTipo(TipoMovimentacao.ENTRADA);
        }else{
            mov.setTipo(TipoMovimentacao.SAIDA);
        }
        mov.setQuantidade(Math.abs(produto.getQuantidade() - quantidadeAtual));
        return mov;
    }
}
```

Basicamente o que o método `geraMovimentacao` faz é receber um produto com a quantidade nova de itens e comparar com a atual, descobrindo se a movimentação é de entrada ou saída.

Também precisamos mexer no método alterar da ProdutosController para usar o geradorDeMovimentacao:

```
@RequestMapping(value="/alterar" , method=RequestMethod.POST)
public String alterar(Produto produto) {
    Movimentacao movimentacao = geradorDeMovimentacao.geraMovimentacao(produto);
    movimentacaoDao.salvar(movimentacao);
    produtoDAO.alterar(produto);
    return "redirect:/produtos/listar.html";
}
```

Agora toda as vezes que alterarmos a quantidade de um produto, será gerada uma movimentação.

5.18 - Exercícios

- 1) Crie a entidade Movimentacao e a classe TipoMovimentacao, não esqueça das anotações do Hibernate e de registrar a entidade no xml do Spring:

```
package br.com.caelum.estoque;
@Entity
public class Movimentacao {

    @Id
    @GeneratedValue
    private Long id;
    private Calendar data;
    @Enumerated(EnumType.STRING)
    private TipoMovimentacao tipo;
    private Integer quantidade;
    @ManyToOne
    private Produto produto;

    //getters e setters
}

package br.com.caelum.estoque;

public enum TipoMovimentacao {
    ENTRADA, SAIDA
}
```

- 2) Altere a interface ProdutoDAO adicionando o método estoqueAtual:

```
public interface ProdutoDAO {
    void salvar(Produto produuto);
    void alterar(Produto produto);
    List<Produto> listar();
    Produto buscarPorId(Long id);
    Integer estoqueAtual(Produto produto);
}
```

Ao alterar a interface precisamos implementar este novo método nas classes ProdutoHibernateDAO e ProdutoMemoriaDAO.

```
public class ProdutoHibernateDAO implements ProdutoDAO {  
    //outros metodos ja existentes  
  
    public Integer estoqueAtual(Produto produto) {  
        Query query = factory.getCurrentSession()  
            .createQuery("select quantidade from Produto where id = :pid");  
        query.setParameter("pid", produto.getId());  
        return (Integer) query.uniqueResult();  
    }  
}  
  
  
  
public class ProdutoMemoriaDAO implements ProdutoDAO {  
    //outros metodos ja existentes  
  
    public Integer estoqueAtual(Produto produto) {  
        for (Produto p : produtos) {  
            if(p.getId().equals(produto.getId())){  
                return p.getQuantidade();  
            }  
        }  
        throw new IllegalArgumentException("Produto nao encontrado");  
    }  
}
```

3) Crie a classe MovimentacaoDAO:

```
@Repository  
public class MovimentacaoDAO {  
  
    private final SessionFactory factory;  
  
    @Autowired  
    public MovimentacaoDAO(SessionFactory factory) {  
        this.factory = factory;  
    }  
  
    public void salvar(Movimentacao movimentacao) {  
        factory.getCurrentSession().save(movimentacao);  
    }  
}
```

4) Crie um novo *bean* do Spring para gerar as movimentações:

```
package br.com.caelum.estoque.controller;

@Service
public class GeradorDeMovimentacao {

    private final ProdutoDAO produtoDAO;

    @Autowired
    public GeradorDeMovimentacao(@Qualifier("produtoHibernateDAO") ProdutoDAO produtoDAO) {
        this.produtoDAO = produtoDAO;
    }

    public Movimentacao geraMovimentacao(Produto produto) {
        Movimentacao mov = new Movimentacao();
        mov.setData(Calendar.getInstance());
        mov.setProduto(produto);

        Integer quantidadeAtual = produtoDAO.estoqueAtual(produto);
        if(produto.getQuantidade()>quantidadeAtual){
            mov.setTipo(TipoMovimentacao.ENTRADA);
        }else{
            mov.setTipo(TipoMovimentacao.SAIDA);
        }
        mov.setQuantidade(Math.abs(produto.getQuantidade()-quantidadeAtual));
        return mov;
    }

}
```

- 5) Por fim, altere o método alterar do controller, ProdutoController, não esqueça de declarar as novas dependências para esta classe funcionar e tornar o método alterar transacional:

```
package br.com.caelum.estoque.controller;

@Controller
@RequestMapping(value = "/produtos")
public class ProdutosController {

    @Autowired
    @Qualifier("produtoHibernateDAO")
    private ProdutoDAO produtoDAO;
    @Autowired
    private GeradorDeMovimentacao geradorDeMovimentacao;
    @Autowired
    private MovimentacaoDAO movimentacaoDao;

    @RequestMapping(value="/alterar" , method=RequestMethod.POST)
    @Transactional
    public String alterar(Produto produto) {
        Movimentacao movimentacao = geradorDeMovimentacao.geraMovimentacao(produto);
        movimentacaoDao.salvar(movimentacao);
        produtoDAO.alterar(produto);
        return "redirect:/produtos/listar.html";
    }
}
```

- 6) Teste alterar um produto já existente e confira no banco se a movimentação foi gravada com sucesso.

Para entrar no mysql, vá ao terminal e digite:

```
mysql -u root
```

Já dentro do mysql busque as movimentações

```
use fj27
select * from Movimentacao;
```

5.19 - Mostrando as movimentações na tela

Ver as movimentações direto no banco não é uma maneira efetiva para mostrar as movimentações no banco de dados, por isso vamos mostrá-las na tela. Para isso usaremos a página já existente de visualização de um produto.

Para facilitar nosso trabalho, tornaremos o relacionamento entre as classes `Produto` e `Movimentacao` bidirecional:

```
@Entity
public class Produto {

    @Id
    @GeneratedValue
    private Long id;
    private String descricao;
    private Integer quantidade;

    @OneToMany(mappedBy="produto")
    private List<Movimentacao> movimentacoes;
}
```

Agora no JSP fica muito mais fácil de mostrar as movimentações de um produto:

```
<h2>Lista de Movimentações</h2>
<ul>
<c:forEach items="${produto.movimentacoes}" var="m">
    <li>
        ${m.tipo} - ${m.quantidade} -
        <fmt:formatDate value="${m.data.time}" pattern="dd/MM/yyyy - hh:mm"/>
    </li>
</c:forEach>
</ul>
```

5.20 - Exercícios

- 1) Modifique a classe `Produto` para ter um relacionamento bidirecional com a classe `Movimentacao`:

```
public class Produto {

    @Id
    @GeneratedValue
    private Long id;
    private String descricao;
    private Integer quantidade;

    @OneToMany(mappedBy="produto")
    private List<Movimentacao> movimentacoes;

    //gerar o getter e setter de movimentacoes
}
```

- 2) Adicione antes da tag </body> a listagem de movimentações no arquivo /WEB-INF/views/produtos/mostrar.jsp:

```
<h2>Lista de Movimentações</h2>
<ul>
<c:forEach items="${produto.movimentacoes}" var="m">
    <li>
        ${m.tipo} - ${m.quantidade} -
        <fmt:formatDate value="${m.data.time}" pattern="dd/MM/yyyy - hh:mm"/>
    </li>
</c:forEach>
</ul>
```

Não esqueça de adicionar no começo do arquivo as declarações de taglib:

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

- 3) Teste entrar na tela de detalhes de um produto que tenha movimentações, deu certo?

O esperado neste exercício é que dê a exception `LazyInitializationException`, mas qual o motivo para ela ter ocorrido?

5.21 - Lidando com LazyInitializationException

Ao carregar um produto o Hibernate, por padrão, **não** carrega as movimentações de um produto, pois, por padrão qualquer relacionamento que seja `@ManyToOne` é *LAZY*. Relacionamentos *lazy* são carregados no último momento possível, por exemplo ao tentar usar algum elemento da lista, ou tentar ver seu tamanho através do `size`.

Ao usar o gerenciamento de transação do Spring, cada operação que precisar de uma transação, acaba abrindo e fechando uma sessão também e aqui é que está o problema. A sessão que o produto estava associado será fechada ao final do método do controller (ou na DAO) que buscou o produto, com a sessão fechada o hibernate não consegue mais carregar (de maneira *lazy*) a lista de movimentação.

Nosso código propriamente dito não usa a lista de movimentações nenhuma vez, porém esta lista é lida pelo jsp, na *Expression Language* `${produto.movimentacoes}`. Na etapa de renderização do JSP a sessão do hibernate já foi encerrada, lançando uma `LazyInitializationException` no JSP.

O que fazer para corrigir este problema?

Na documentação do Hibernate, Gavin King cita a solução ***Open Session in View***.

5.22 - Solução Open Session in View

Na documentação hibernate é citado um padrão de desenvolvimento específico para resolver este problema, o **Open Session in View**. Esse padrão sugere que antes de executar uma lógica devemos abrir uma sessão e só fechá-la depois que o HTML (a View) for escrito, evitando erros de inicialização tardia.

Uma das implementações mais comuns de Open Session in View na Web é através de filtros web. Por ser uma solução tão comum e tão ligada ao controle de *Session* do Spring este filtro já está implementado através da classe `OpenSessionInViewFilter`. Basta registrá-lo como um filtro web comum que ele usará as configurações do próprio Spring.

5.23 - Exercícios

- 1) Declare o filtro do Spring no seu `web.xml`

```
<filter>
    <filter-name>hibernateFilter</filter-name>
    <filter-class>
        org.springframework.orm.hibernate3.support.OpenSessionInViewFilter
    </filter-class>
</filter>

<filter-mapping>
    <filter-name>hibernateFilter</filter-name>
    <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>
</filter-mapping>
```

- 2) O Filtro do Spring procura por um bean chamado **sessionFactory** para abrir uma sessão, porém ele não procura na servlet do Spring e sim em um contexto ativado pelo `ContextLoaderListener` do Spring, registre-o também no `web.xml`.

```
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

- 3) O `ContextLoaderListener` precisa saber onde está o arquivo de configuração do Spring (por padrão ele procura pelo `applicationContext.xml` dentro de `WEB-INF`).

Registre um `context-param` abaixo da tag `</listener>`

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/app-config.xml</param-value>
</context-param>
```

- 4) Para evitar o carregamento repetido das configurações do Spring vamos fazer a Servlet do Spring carregar não mais o `app-config.xml`, mas sim um arquivo de configurações vazio.

- Crie um arquivo chamado **empty.xml** dentro da pasta `/WEB-INF/spring` com o seguinte conteúdo:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
```

```
http://www.springframework.org/schema/beans  
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
```

```
</beans>
```

- Mude o <init-param> da servlet do Spring para ler o empty.xml:

```
<servlet>  
    <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>  
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
    <init-param>  
        <param-name>contextConfigLocation</param-name>  
        <param-value>  
            /WEB-INF/spring/empty.xml  
        </param-value>  
    </init-param>  
    <load-on-startup>1</load-on-startup>  
</servlet>
```

5.24 - Transações Avançadas

A anotação @Transactional que o Spring possui, tem diversas propriedades que permite a configuração de características variadas relacionadas a nossa transação:

- propagation - configura o nível de propagação da transação.
- isolation - configura o nível de isolamento da transação.
- readOnly - indica se a transação é somente leitura
- timeout - tempo de timeout da transação em milisegundos
- rollbackFor - array de exceptions que devem causar rollback na transação
- rollbackForClassName - array de nomes de exceptions que devem causar rollback na transação
- noRollbackFor - array de exceptions que **NÃO** devem causar rollback na transação
- noRollbackForClassName - array de nomes de exceptions que **NÃO** devem causar rollback na transação

Propagação

Ao trabalharmos com transações o Spring nos fornece seis tipos de propagação. A propagação ocorre quando uma lógica de negócios que utiliza uma transação chama outra que pode ou não utilizar transações.

A escolha do tipo de propagação permite criar novas transações, parar a transação atual, não aceitar a chamada se uma transação já está acontecendo, etc.

- REQUIRED
- REQUIRES_NEW
- MANDATORY
- NESTED
- NEVER

- NOT_SUPPORTED
- SUPPORTS

Isolamento

Uma transação pode ser isolada das outras que estão ocorrendo. As constantes da interface TransactionDefinition devem ser utilizadas:

- ISOLATION_READ_UNCOMMITTED - permite o que chamamos de leitura suja (dirty read), isto é, você pode visualizar alterações em outras linhas que estão participando de uma outra transação nesse instante e se algum **rollback** for executado na outra transação, sua leitura terá sido inválida.
- ISOLATION_READ_COMMITTED - proíbe o **dirty read** fazendo com que os dados lidos sejam os anteriores a alteração da outra transação em andamento.
- ISOLATION_REPEATABLE_READ - não permite que ocorra a situação onde você leia uma linha de dados, essa linha seja alterada por outra transação e então você leia novamente aquela linha. Se permitisse, você receberia dessa vez dados diferentes da primeira leitura (non-repeatable read).
- ISOLATION_SERIALIZABLE - implica nas mesmas limitações de ISOLATION_REPEATABLE_READ e não permite a situação onde é feita uma pesquisa com cláusula WHERE pela sua transação, outra transação insere uma linha que seria retornada na cláusula WHERE que você utilizou e uma nova pesquisa com a mesma cláusula WHERE retornaria o novo item (phantom read).

5.25 - Para saber mais: HibernateTemplate e outros Templates

Antes da versão 3 do Hibernate, as exceções lançadas por sua API eram checked, com isso o código ficava muito poluído com o tratamento obrigatório de exceções, já hoje a grande maioria das exceções lançadas pelo Hibernate são de `Runtime`, melhorando muito a legibilidade do código. Porém como no Hibernate 2 a história era outra uma das alternativas oferecidas pelo Spring na época, era o uso de `Templates`.

Os *templates* são classes que encapsulam e facilitam o trabalho com algumas API's. Para o Hibernate a classe `HibernateTemplate` ajuda traduzindo algumas *exception's* para alguma mais legível e significativa, porém além de traduzir essas exceções o Spring lançava apenas exceções *unchecked*, evitando alguns try-catch's.

Não só o `HibernateTemplate` ajudava nas exceções, ele também ajudava a controlar recursos como conexão e fazia o papel da JTA, controlando o ciclo de vida das transações.

Hoje o uso das classes `HibernateTemplate` e `JpaTemplate` são completamente opcionais, pois o controle de conexão, Session e transações é feito de outra maneira pelo Spring.

Uma classe que ainda vale a pena conferir é a `JdbcTemplate`. A API do `Jdbc` ainda é muito trabalhosa, ela impõe muitas vezes o modelo TCFTC (try-catch-finally-try-catch), porém usando a `JdbcTemplate`, não precisamos nos preocupar em fechar a conexão no finally com isso evitamos tratar a `SQLException` do método `close` de `Connection`.

```
class Teste{  
    private JdbcTemplate jdbcTemplate;  
  
    public void setDataSource(DataSource dataSource) {  
        this.jdbcTemplate = new JdbcTemplate(dataSource);  
    }  
  
    public List<Produto> buscaPorNome(String nome) {  
        String sql = "select id, nome, preco from Produto where id = ?";  
  
        RowMapper mapper = new RowMapper() {  
            public Object mapRow(ResultSet rs, int rowNum) throws SQLException {  
                Produto p = new Produto();  
                p.setId(rs.getLong("id"));  
                p.setNome(rs.getString("nome"));  
                p.setPreco(rs.getDouble("preco"));  
                return p;  
            }  
        };  
  
        return jdbcTemplate.query(sql, new Object[] {Long.valueOf(id)}, mapper);  
    }  
}
```

Embora o código ainda seja procedural e braçal, este código mal se preocupa com exceptions, transações ou conexões todo o trabalho é feito pelo JdbcTemplate.

Validação com Bean Validation

6.1 - Introdução

Não existe tarefa mais comum hoje em dia do que validar dados em uma aplicação. Por exemplo, validamos se na camada de apresentação o usuário preencheu algum campo obrigatório, depois fazemos a mesma lógica de validação em nossa regra de negócio e por último validamos se os dados que serão salvos no banco também estão corretos. O que na maioria das vezes os desenvolvedores fazem é validar estas regras em todos os lugares, e muitas vezes resulta em validações complicadas e possíveis erros na aplicação. O Spring disponibiliza um design para validação próprio através da interface Validator. Mas com a versão 3 do framework os olhares se voltam para a integração que o mesmo disponibiliza com a especificação Bean Validation.

6.2 - Bean Validation

Na nova versão do Java EE lançada dezembro de 2009 existem grandes novidades: Servlets 3.0, JAX-RS, CDI, JSF 2.0 e a Bean Validation, definida pela JSR 303. Bean Validation especifica um padrão declarativo para baseado em anotações. Algumas anotações disponíveis são:

- @Min - especifica que o elemento anotado deve ter um número maior ou igual ao especificado.
- @Max - especifica que o elemento anotado deve ter um número menor ou igual ao especificado.
- @Size - valida o tamanho entre valores mínimo e máximo especificado.(Os tipos suportados incluem String, Collection, Map and Array)
- @NotNull - O elemento não pode ser null.
- @Null : O elemento deve ser null.
- @Pattern: Verifica se o elemento está dentro de uma Java regular expression.(Apenas String é suportado)

6.3 - Hibernate Validator

O Spring provê uma integração totalmente transparente com a Bean Validation. No entanto o mesmo não a implementa. É necessário utilizarmos uma implementação da API, como o Hibernate Validator que é a implementação de referência. Entre as formas disponibilizadas para utilizarmos um Validator da Bean Validation no Spring temos a classe LocalValidatorFactoryBean. A LocalValidatorFactoryBean detecta automaticamente o jar do Hibernate Validator, ou outro no classpath da aplicação sem nenhuma configuração extra. Sua declaração no xml de configuração é extremamente simples:

```
<bean id="validator"
      class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean" />
```

Assim podemos injetá-la em qualquer outro bean que é também gerenciado pelo Spring.

```
@Service  
public class PessoaService{  
  
    @Autowired  
    private Validator validator;  
  
}
```

Através do validator chamar o método validate() e o mesmo nos retorna uma lista de ConstraintViolation representando as validações violadas. O Spring 3 adiciona ao seu módulo MVC um forma declarativa de utilizarmos a integração com a Bean Validation. A anotação @Valid nos permite indicar que deve se validar um parâmetro de um controller de acordo com as anotações declaradas no objeto.

```
@Controller  
public class PessoaController {  
  
    public String salvar(@Valid Pessoa pessoa){  
        //....  
    }  
  
}
```

A anotação @Valid informa ao Spring MVC para chamar o método validate() do Validator para o parâmetro **pessoa**. @Valid é ativada através da configuração:

```
<mvc:annotation-driven/>
```

6.4 - Exercícios

- 1) Adicione validação a classe Produto.

```
@Entity  
public class Produto {  
  
    @Id  
    @GeneratedValue  
    private Long id;  
  
    @NotEmpty  
    private String descricao;  
  
    @NotNull  
    private Integer quantidade;  
}
```

- 2) Adicione validação a classe Movimentacao:

```
@Entity
public class Movimentacao {

    @Id
    @GeneratedValue
    private Long id;

    @NotNull
    private Calendar data;

    @Enumerated(EnumType.STRING)
    private TipoDeMovimentacao tipo;

    @NotNull
    private Integer quantidade;

    @ManyToOne
    @NotNull
    private Produto produto;
}
```

- 3) Altere os métodos salvar() e alterar() de ProdutoController, adicionando a anotação @Valid no parâmetro Produto:

```
@Controller
@RequestMapping(value="/produtos")
public class ProdutosController {

    // código antigo

    @RequestMapping(value="/salvar" , method = RequestMethod.POST)
    @Transactional
    public String salvar(@Valid Produto produto) {
        produtoDAO.salvar(produto);
        geraMovimentacao.entrada(produto);

        return "redirect:/produtos/listar.html";
    }

    @RequestMapping(value="/alterar" , method = RequestMethod.POST)
    @Transactional
    public String alterar(@Valid Produto produto) {
        produtoDAO.alterar(produto);
        geraMovimentacao.entrada(produto);

        return "redirect:/produtos/listar.html";
    }

    // código antigo
}
```

- 4) Tente cadastrar um produto com dados em branco!

6.5 - Mostrando erros em tela

Quando o Spring validation é executado e alguma validação é violada, as exceções de ConstraintViolation são lançadas para tela. Temos como receber um objeto que representa as violações da execução da validação.

BindingResult é o objeto que tem a análise do que aconteceu na validação, podemos pedir esse objeto para o Spring apenas declarando o mesmo como parâmetro de um método de controller.

```
@Controller  
@RequestMapping(value="/produtos")  
public class ProdutosController {  
  
    //...  
    @RequestMapping(value="/salvar", method=RequestMethod.POST)  
    public void salvar(@Valid Produto produto, BindingResult result){  
  
    }  
  
}
```

Com ele podemos verificar a existência de erros e redirecionar para uma página de erros antes de da tentativa de salvar o objeto e ter as exceções devolvidas ao usuário.

```
@Controller  
@RequestMapping(value="/produtos")  
public class ProdutosController {  
  
    //...  
    @RequestMapping(value="/salvar", method=RequestMethod.POST)  
    public String salvar(@Valid Produto produto, BindingResult result){  
  
        if(result.hasErrors()){  
            return "formulario";  
        }  
  
        //...  
    }  
}
```

Retornando para o formulario de cadastro podemos utilizar das taglibs do Spring para mostrarmos os erros ao usuário.

6.6 - Exercícios

- 1) Altere o método salvar() de ProdutosController para redirecionar no caso de erros.

```
@RequestMapping(value="/salvar" , method = RequestMethod.POST)  
@Transactional  
public String salvar(@Valid Produto produto, BindingResult result) {  
  
    if(result.hasErrors()) {  
        return "produtos/form";  
    }  
  
    produtoDAO.salvar(produto);  
    geraMovimentacao.entrada(produto);  
  
    return "redirect:/produtos/listar.html";  
}
```

2) Altere o método `form()` de `ProdutosController`.

```
@RequestMapping(value="/form" , method=RequestMethod.GET)
public String form(Model model) {
    model.addAttribute(new Produto());
    return "produtos/form";
}
```

3) Adiciona a taglib do Spring na `form.jsp`.

```
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<form:form method="post" action="salvar.html" commandName="produto">
    Descricao: <form:input path="descricao"/> <form:errors path="descricao"/><br/>
    Quantidade: <form:input path="quantidade"/> <form:errors path="quantidade"/><br/>
    <input type="submit" value="Salvar">
</form:form>
```

4) Altere o método `alterar()` de `ProdutosController` para redirecionar no caso de erros.

```
@RequestMapping(value="/alterar" , method = RequestMethod.POST)
@Transactional
public String alterar(@Valid Produto produto, BindingResult result) {

    if(result.hasErrors()) {
        return "produtos/editar";
    }

    produtoDAO.alterar(produto);
    geraMovimentacao.entrada(produto);

    return "redirect:/produtos/listar.html";
}
```

5) Adiciona a taglib do Spring na `editar.jsp`.

```
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<form:form action="/estoque/produtos/alterar.html" method="post" commandName="produto">
    <form:hidden path="id"/><br/>
    Descricao: <form:input path="descricao"/> <form:errors path="descricao"/><br/>
    Quantidade: <form:input path="quantidade"/> <form:errors path="quantidade"/><br/>
    <input type="submit" value="Alterar">
</form:form>
```

6) Tente cadastrar ou alterar um Produto com dados em branco!

6.7 - Customizando mensagens

É possível configurar as mensagens padrões de erros. Para isso é preciso configurar o Spring para ler um arquivo onde as chaves dos erros estejam com os valores redefinidos. O Spring permite a configuração de um loader para arquivos properties.

```
<bean id="messageSource"
      class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
```

```
<property name="basename" value="/WEB-INF/i18n/messages" />
</bean>
```

6.8 - Exercício

- 1) Crie a pasta **i18n** dentro de WEB-INF e nela crie o arquivo **messages.properties** adicionando o seguinte conteúdo.

```
NotEmpty.produto.descricao=Deve ser preenchido
NotNull.produto.quantidade=Deve ser preenchido
```

- 2) Configure o Spring para ler o arquivo de mensagens

```
<bean id="messageSource"
      class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <property name="basename" value="/WEB-INF/i18n/messages" />
</bean>
```

- 3) Tente cadastrar ou alterar um Produto com dados em branco!

Spring Security

7.1 - Introdução ao Spring Security

Spring também oferece uma alternativa ao padrão Java na segurança de aplicações de negócios. Usando o padrão Java, JAAS (Java Authentication and Authorization Service), as aplicações não são mais 100% portável entre servidores de aplicações. JAAS sempre depende de uma configuração específica do servidor. Além da desvantagem, JAAS tem a fama de ser complicado e burocrático demais.

Spring Security é uma alternativa flexível, simples e 100% portável entre servidores. Ele nem precisa de um servidor de aplicação, pode ser usado dentro de uma aplicação *standalone* ou dentro de um *servlet container*.

Spring Acegi

Spring Security também é conhecido como Acegi. Spring Security é o novo nome e foi introduzido na versão 2.0.

Para habilitarmos o Spring Secutiry precisamos fazer algumas configurações no web.xml do nosso *servlet container*. Entre as configurações o mais importante é o filtro DelegatingFilterProxy que vai interceptar as chamadas a aplicação e verificar as retrições que vamos configurar utilizando o Spring Secutiry. A configuração do Filtro é bem simples, mas é importante dizer exatamente onde o mesmo vai atuar com a *url-pattern* do mapeamento.

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>*</url-pattern>
</filter-mapping>
```

O Filtro é toda a base do Spring Security para a web. Mas também precisamos de uma outra configuração. O filtro requer um listener para disponibilizar acesso ao *WebApplicationContext* que é onde ficam os objetos gerenciados pelo o Spring. O listener deve ser configurado em conjunto com um parâmetro que informa a localidade do XML de configuração do Spring.

```
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<context-param>
    <param-name>contextConfigLocation</param-name>
```

```
<param-value>/WEB-INF/spring/app-config.xml</param-value>
</context-param>
```

7.2 - Exercícios - Configurando o Spring Security

- 1) Para utilizarmos o Spring Security precisamos definir um `listener` e um `filter` no `web.xml`. Ambos são os responsáveis por validar as estruturas de segurança que definirmos no XML do spring. O Filter é mapeado para interceptar todas as urls do projeto. E o listener precisa do parâmetro `contextConfigLocation` que informa onde o XML do Spring está.

```
<webapp>
/...

<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

</webapp>
```

7.3 - - Access Control List

O Spring assim como o JAAS, disponibiliza uma maneira de configuração em XML para os usuários e senhas da aplicação. Mas isso não é muito bem aceito pela comunidade, é inviável o controle de usuários via XML. Por exemplo, nenhum usuário pode se cadastrar no sistema dinamicamente, caso seja necessário um novo usuário acessar o sistema o tempo de desenvolvimento precisaria alterar o XML e subir a aplicação em produção para que o mesmo conseguisse se logar.

Todos os sistemas que requerem autenticação esperam que um usuário possa se cadastrar sem a intervenção do time de desenvolvimento e que o mesmo possa ter suas permissões atribuídas também dinamicamente. O Spring Security nos permite utilizar um modelo de dados próprio para o controle da segurança de nossa aplicação. E por isso vamos definir as classes de domínio `Usuario` e `Grupo` que vão representar o ACL do nosso sistema.

Ambas as classes são Entidades, e por isso faremos o mapeamento de objeto relacional utilizando as anotações do JPA.

A classe `Grupo` representa a ROLE de alguém no sistema, ou seja um usuário pode ter a ROLE de Administrador ou de Usuário que permite acesso a elementos diferentes dentro da aplicação. A Role é representada pelo nome da entidade `Grupo`.

```
package br.com.caelum.estoque;

@Entity
public class Grupo {

    @Id
    @GeneratedValue
    private Long id;
    private String nome;

    //getters e setters
}
```

A classe Usuário representa o login de alguem no sistema o mesmo possui login e senha de acesso. E uma lista das roles que o mesmo pode executar nos elementos do sistema.

```
package br.com.caelum.estoque;

@Entity
public class Usuario {

    @Id
    @GeneratedValue
    private Long id;
    private String login;
    private String senha;

    @OneToMany(fetch=FetchType.EAGER)
    private List<Grupo> grupos;

    // getters e setters
}
```

7.4 - Exercícios - Criando as classes de Domínio

- 1) Para implementarmos a segurança de nosso projeto, precisamos definir as Entidades que irão representar os Usuarios e suas Permissões no Banco. Aproveitamos para fazer o mapeamento ORM. Começamos implementando e mapeando a classe Usuario que vai representar os dados para login. Esta classe deve estar no pacote **br.com.caelum.estoque**.

```
package br.com.caelum.estoque;

@Entity
public class Usuario {

    @Id
    @GeneratedValue
    private Long id;
    private String login;
    private String senha;

    @OneToMany(fetch=FetchType.EAGER)
    private List<Grupo> grupos;

    // getters e setters
}
```

- 2) Temos que implementar e mapear a classe Grupo, também. Esta classe deve estar no pacote **br.com.caelum.estoque**.

```
package br.com.caelum.estoque;

@Entity
public class Grupo {

    @Id
    @GeneratedValue
    private Long id;
    private String nome;

    //getters e setters
}
```

- 3) É necessário configurar o XML do spring, informando o Hibernate sobre as novas entidades mapeadas.

```
<property name="annotatedClasses">
<list>
    ...
    <value>br.com.caelum.estoque.Grupo</value>
    <value>br.com.caelum.estoque.Usuario</value>
    ...
</list>
</property>
```

7.5 - Interfaces de Acesso

Como comentamos o Spring nos permite usar nosso próprio modelo de dados para o controle de segurança. O mesmo disponibiliza algumas interfaces para que nossas classes implementem e assim ele faça as chamadas requeridas pela API de segurança.

A interface `GrantedAuthority` deve ser implementada pela classe que representa as informações de ROLE. É necessário prover implementação ao método `getAuthority()` que retorna o nome da ROLE mantida pelo objeto. Nossa classe `Grupo` implementando a interface `GrantedAuthority` teria a seguinte estrutura:

```
@Entity
public class Grupo implements GrantedAuthority {

    @Id
    @GeneratedValue
    private Long id;
    private String nome;

    //getters e setters

    @Override
    public String getAuthority() {
        return getNome();
    }
}
```

O método `getAuthority()` delega para o `getNome()` para retornar o nome da ROLE. O nome das ROLES devem conhecer com as definidas no XML do Spring. Mais adiante veremos isso.

A interface `UserDetails` deve ser implementada pela classe que representa as informações do usuário. Para quem implementa está interface é requerido a sobreescrita de 7 métodos, todos informando algum aspecto de segurança.

O método `getUsername()` deve retornar o nome do usuário que está logado. Em nossa entidade `Usuario` mantemos o `username` na propriedade `login` da entidade. Por isso delegariamos a chamada desse método para `getLogin()`.

O método `getPassword()` deve retornar a senha do usuário que está logado. Em nossa entidade `Usuario` mantemos o `password` na propriedade `senha` da entidade. Por isso delegariamos a chamada desse método para `getSenha()`.

O método `isAccountNonExpired()` retorna `true` ou `false`. E pode ser utilizado para uma possível regra que determina um tempo de expiração para o usuário cadastrado.

O método `isAccountNonLocked()` retorna `true` ou `false`. E pode ser utilizado para uma possível regra que bloqueia a conta de usuário, por uma indiscrição ou algo similar.

O método `isCredentialsNonExpired()` retorna `true` ou `false`. E pode ser utilizada para uma regra de credenciais para usuários. Por exemplo uma credencial provendo o benefício de acessar algo a mais no sistema.

O método `isEnabled()` retorna `true` ou `false`. E pode ser utilizada para determinar se a conta do usuário está ativa ou não.

A implementação da nossa classe usuário vai retornar `true` para todos os métodos para simplificar o desenvolvimento.

```
@Entity
public class Usuario implements UserDetails {

    @Id
    @GeneratedValue
    private Long id;
    private String login;
    private String senha;

    @OneToMany(fetch=FetchType.EAGER)
    private List<Grupo> grupos;

    //getters e setters

    @Override
    public Collection<GrantedAuthority> getAuthorities() {
        return new ArrayList<GrantedAuthority>(grupos);
    }

    @Override
    public String getPassword() {
        return getSenha();
    }

    @Override
    public String getUsername() {
        return getLogin();
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }
}
```

Essas interfaces acima são utilizadas nas entidades que definimos para o nosso domínio. Mas o Spring Security não consegue automaticamente fazer uma consulta para verificar o login de um usuário. Ele precisa também que informemos a ele a consulta que deve ser chamada para verificar a autenticação de um usuário. Para isso usamos a interface `UserDetailsService` que requer que implementemos o método `loadUserByUsername()` que recebe como parâmetro o nome do usuário que está efetuando o login. Na assinatura desse método ele devolve um objeto do tipo `UserDetails`. Ou seja devemos fazer a consulta baseada no nome do usuário e retornar uma instância de usuário do nosso domínio que o mesmo implementa a interface `UserDetails`. Apesar do Spring apenas nos fornecer o nome do usuário para a consulta, ele faz o teste da senha com o objeto retor-

nado então se a consulta retorna um objeto com uma senha diferente da passada o Spring vai invalidar o login. Vamos fazer nossa classe DAO representar o tipo `UserDetailsService`.

```
package br.com.caelum.estoque.dao;

public interface UsuarioDAO extends UserDetailsService {

}

package br.com.caelum.estoque.dao;

@Repository
public class UsuarioHibernateDAO implements UsuarioDAO {

    private HibernateTemplate hibernateTemplate;

    @Autowired
    public UsuarioHibernateDAO (SessionFactory factory) {
        hibernateTemplate = new HibernateTemplate(factory);
    }

    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException, DataAccessException {
        List usuarios =
            hibernateTemplate.find("from Usuario where login = ?", username);

        if(!usuarios.isEmpty()) {
            return (Usuario) usuarios.get(0);
        }

        return null;
    }
}
```

O último passo na jornada de fazer o Spring Security utilizar nosso próprio modelo de objetos é indicar tudo que criamos a ele. Precisamos configurar nossa classe `UsuarioHibernateDAO` como o provider de segurança, e isso é possível porque o mesmo implementa a interface `UserDetailsService`.

```
<sec:authentication-manager>
    <sec:authentication-provider user-service-ref="usuarioHibernateDAO"/>
</sec:authentication-manager>
```

7.6 - Exercícios - Suporte a Acesso de Dados

- 1) Nossa entidade `Grupo` deve implementar a interface `GrantedAuthority`. Esta interface requer que implementemos 1 método informando o nome da *ROLE* que o spring deve validar.

```
@Entity
public class Grupo implements GrantedAuthority {

    @Id
    @GeneratedValue
    private Long id;
    private String nome;

    //getters e setters

    @Override
    public String getAuthority() {
        return getNome();
    }
}
```

- 2) Quando utilizamos o Spring Security acessando o banco de dados, o Spring requer que implementemos algumas interfaces pre-definidas nos objetos de domínio que representam a segurança. Para nossa Entidade Usuario vamos implementar a interface UserDetails do Spring. Esta interface requer que reimplementarmos 7 métodos utilizados para segurança.

```
@Entity
public class Usuario implements UserDetails {

    @Id
    @GeneratedValue
    private Long id;
    private String login;
    private String senha;

    @OneToMany(fetch=FetchType.EAGER)
    private List<Grupo> grupos;

    //getters e setters

    @Override
    public Collection<GrantedAuthority> getAuthorities() {
        return new ArrayList<GrantedAuthority>(grupos);
    }

    @Override
    public String getPassword() {
        return getSenha();
    }

    @Override
    public String getUsername() {
        return getLogin();
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }
}
```

- 3) Para o acesso a dados vamos definir um DAO para `Usuario`. Seguindo a mesma boa prática de programar orientado a interface vamos definir uma interface `UsuarioDAO`. Vamos definir também que nossa interface extenda a interface `UserDetailsService` do spring assim o DAO que implementar `UsuarioDAO` pode ser utilizado como um provider de segurança no Spring.

```
package br.com.caelum.estoque.dao;

public interface UsuarioDAO extends UserDetailsService {

}
```

- 4) Vamos definir a implementação do DAO para a interface UsuarioDAO. Precisamos prover uma implementação para o método `loadUserByUsername()` que herdamos da interface `UserDetailsService`, faça com que a interface `UsuarioDAO` herde da interface `UserDetailsService`. O método `loadUserByUsername` é chamado pelo Spring Security para a autenticação de um usuário no banco de dados. Nossa implementação recebe também a factory do Hibernate e faz o uso de `HibernateTemplate` por causa de transações e exceptions.

```
package br.com.caelum.estoque.dao;

@Repository
public class UsuarioHibernateDAO implements UsuarioDAO {

    private HibernateTemplate hibernateTemplate;

    @Autowired
    public UsuarioHibernateDAO (SessionFactory factory) {
        hibernateTemplate = new HibernateTemplate(factory);
    }

    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException, DataAccessException {
        List usuarios =
            hibernateTemplate.find("from Usuario where login = ?", username);

        if(!usuarios.isEmpty()) {
            return (Usuario) usuarios.get(0);
        }

        return null;
    }
}
```

- 5) Vamos configurar o XML do Spring `app-config.xml` e informar que o provider de autenticação é o nosso DAO `UsuarioHibernateDAO`. Com essa configuração o Spring Security vai fazer a chamada ao método `loadUserByUsername()` para validar o login de um usuário.

```
<security:authentication-manager>
    <sec:authentication-provider user-service-ref="usuarioHibernateDAO"/>
</security:authentication-manager>
```

7.7 - Login

Vamos criar a página de login para nosso sistema. A mesma deve seguir 3 convenções definidas pelo Spring Security. Os inputs que definem o nome e senha do usuário devem ter os respectivos nomes `j_username` e `j_password`. E a url definida no action do form deve bater com a configurada no XML do Spring. Nossa jsp ficaria assim:

```
<html>
    <head>
```

```

<title>Login</title>
</head>
<body>
    <form action="/estoque/login" method="post">
        Login : <input type="text" name="j_username"/><br/>
        Senha : <input type="text" name="j_password"/><br/>
        <input type="submit" value="Login">
    </form>
</body>
</html>

```

Bem simples. A configuração principal dessa parte é realizada no XML do Spring. Devemos definir as ROLES que nossos usuários podem ter e quais lugares as mesmas tem acesso. Devemos definir a url que a página de login deve chamar, e para onde o Spring tem que redirecionar caso alguém tente acessar algo que não possa. Também devemos informar para onde é realizado o redirecionamento em caso do login com sucesso. Todas essas configurações são realizadas no elemento *http* do Spring Security.

```

<sec:http auto-config="true">
</sec:http>

```

Em nosso sistema temos uma ROLE a de usuário, que define acesso para as funcionalidades do ProdutoController. Na configuração do Spring informamos um tipo de URL que deve ser interceptado pelo Security e o nome da ROLE, ou seja, do Grupo que pode acessar aquele tipo de url. Para definirmos que apenas Usuários cadastrados no grupo *ROLE_USER* podem ter acesso as funcionalidades de ProdutoController definimos:

```

<sec:http auto-config="true">
    <sec:intercept-url pattern="/produto/**" access="ROLE_USER" />
</sec:http>

```

Vamos definir no elemento *http* qual url deve ser chamada para a validação de segurança. Por exemplo, para termos a segurança do Spring executada ao chamar a url /login no sistema. Configuramos o atributo *login-processing-url* no elemento *form-login* de segurança. Configuramos também os atributos *login-page* que indica para onde deve ser redirecionado caso alguém tente acessar um recurso sem estar logado ou sem ter permissão, *default-target-url* para definir o alvo, ou seja a jsp que deve ser redirecionado no caso de login com sucesso e *always-use-default-target* para indicar que a todo login o redirecionamento é para o *default-target-url*.

```

<sec:http auto-config="true">
    <sec:intercept-url pattern="/produto/**" access="ROLE_USER" />

    <sec:form-login login-page="/login.html" always-use-default-target="true"
        default-target-url="/produto/listar.html" login-processing-url="/login"/>

    <sec:logout logout-url="/logout" logout-success-url="/login.html" />
</sec:http>

```

7.8 - Exercícios - Logando no Sistema

- Precisamos de uma jsp para o Usuário efetuar o login. Vamos criar uma jsp dentro do diretório "view/usuarios". Precisamos primeiro criar uma pasta chamada *usuarios* em "WEB-INF/views". E depois criar uma jsp chamada *login.jsp* dentro de "WEB-INF/views/usuarios". O conteúdo da nossa jsp contém dois input, Nome e Senha para o usuário se logar.

```
<html>
```

```

<head>
    <title>Login</title>
</head>
<body>
    <form action="/estoque/login" method="post">
        Login : <input type="text" name="j_username"/><br/>
        Senha : <input type="password" name="j_password"/><br/>
        <input type="submit" value="Login">
    </form>
</body>
</html>

```

- 2) Como nós seguimos a boa prática de colocar as jsps abaixo do diretório “WEB-INF” o usuário não tem acesso a nenhuma jsp sem a passagem por um **Controller** do Spring. Por isso vamos criar a classe `UsuariosController` e configurar uma url de acesso para nossa página de login.

```

package br.com.caelum.estoque.controller;

@Controller
public class UsuariosController {

    @RequestMapping(value="/login")
    public String index() {
        return "usuarios/login";
    }
}

```

- 3) Todas as classes de infraestrutura para o controle da segurança de nossa aplicação já foram definidas. Agora precisamos configurar o Spring as permissões, url de login, página de login, url de logout, o que fazer se um usuário inválido tentar se logar. Essas definições são feitas no XML do Spring `app-config.xml`.

```

<security:http auto-config="true">
    <security:intercept-url pattern="/produtos/**" access="ROLE_USER" />

    <security:form-login login-page="/login.html" always-use-default-target="true"
        default-target-url="/produtos/listar.html" login-processing-url="/login"/>

    <security:logout logout-url="/logout" logout-success-url="/login.html" />
</security:http>

```

- 4) Para testarmos precisamos de um usuário no banco de dados. Executem o Script abaixo para no mysql.

```

insert into Usuario (login,senha) values ('eu','eu');
insert into Grupo (nome) values ('ROLE_USER');
insert into Usuario_Grupo values (1,1);

```

- 5) Inicie o Servidor. E chame acesse a url da página de login `http://localhost:8080/estoque/login.html`. Tente se logar com um login e senha que não existem no Banco de Dados, e verifique que no console do eclipse que o Spring executa o método `loadUserByUsername()` e o hibernate imprime a consulta. Devido ao fato do usuário não existir o Spring redireciona o fluxo para a mesma página de login.

- 6) Vamos fazer um segundo teste agora. Ainda sem se logar tente acessar uma das funcionalidades definidas em `ProdutosController`. Por exemplo a listagem `http://localhost:8080/estoque/produtos/listar.html` ou o ca-

dastro de produto `http://localhost:8080/estoque/produtos/form.html`. Verifique que o Spring redireciona também para a página de login devido a nenhum usuário com permissão estar logado.

- 7) Agora vamos logar no sistema. Acessa a página de login e entre com o usuário cadastrado no banco. Repare no console a execução da consulta e que automaticamente você é redirecionado para a página de listagem de `ProdutosController`. Cadastre alguns produtos, teste a edição de outros.
- 8) Agora vamos executar a url de logout. Chame no navegador `http://localhost:8080/estoque/logout`. Verifique que você foi automaticamente redirecionado para a página de login. Agora tente acessar alguma a listagem de produtos `http://localhost:8080/estoque/produtos/listar.html` e mais uma vez você vai ser redirecionado para a página de login devido a nenhum usuário com permissão estar logado.

Apêndice - AOP - Programação Orientada a Aspectos

"Dificuldade em um relacionamento indica uma longa história de problemas."

– Provérbio Chinês

8.1 - Introdução a Programação Orientada a Aspectos

É comum, em um sistema orientado a objetos, um mesmo comportamento aparecer em diversos objetos sem muita relação entre eles. Dizemos que eles aparecem de maneira **ortogonal** à responsabilidade de cada classe: eles pertencem a um grande grupo de classes, porém, ao mesmo tempo, não parece fazer sentido aquele código estar presente nas mesmas.

Os exemplos mais clássicos de comportamentos que costumam estar espalhados por toda a aplicação são: o controle de acesso, autenticação e autorização, o sistema de log, transação e de auditoria.

É muito comum encontrar exemplos na Internet e até mesmo projetos em produção onde o código de log está espalhado por toda sua aplicação. Criar um componente de Log, utilizando uma biblioteca como o Log4J, não é o suficiente, uma vez que instâncias desse controlador de log continuam sendo utilizadas por todas as suas lógicas de negócio: código de uma mesma responsabilidade está espalhado por inúmeras classes.

Esse sistema de log não é uma preocupação principal do código de suas classes, e sim uma preocupação que atravessa (**crosscut**) diversas outras preocupações, formando um **aspecto** da mesma.

A **programação orientada a aspectos** (AOP) tem ganho popularidade, possibilitando o tratamento dessas preocupações ortogonais de uma maneira que você não tenha um mesmo código espalhado por inúmeras classes que não tem exatamente essa responsabilidade.

Diversos frameworks e extensões da linguagem trabalham com isso, como o AspectJ, AspectWerkz e o JBossAOP. Através deles você determina pontos de corte da sua aplicação (**pointcuts**), locais estes que deverão ter um comportamento a mais, seja antes e/ou depois daqueles pontos. Outra forma de fazer isto sem utilizar um framework, é alterar o bytecode das classes carregadas em tempo de execução.

Mitos sobre o AOP

Hoje em dia há muita discussão a respeito de como usar AOP. Muitas pessoas dizem que AOP pode ser substituído pelo bom uso de patterns, ou as vezes uso direto de reflexão e manipulação de bytecode. Outras pessoas dizem que esse tipo de abordagem geraria código muito confuso e poderia quebrar o encapsulamento.

O artigo a seguir discute diversos fatos a respeito do AOP que ele considera mitos:
<http://www.ibm.com/developerworks/library/j-aopwork15/>

8.2 - AOP na prática

Imagine que precisamos colocar um sistema de *logging* nas nossas classes como a GerenciadorProjeto, por exemplo. Podemos fazer da forma convencional, adicionando o código de *logging* dentro da própria classe GerenciadorProjeto:

```
package br.com.caelum.estoque.service;

@Service
public class GeraMovimentacao {

    private final MovimentacaoRepository movimentacaoRepository;
    private Logger logger = Logger.getLogger(GeraMovimentacao.class);

    @Autowired
    public GeraMovimentacao(MovimentacaoRepository movimentacaoRepository) {
        this.movimentacaoRepository = movimentacaoRepository;
    }

    public void entrada(Produto produto) {
        if(logger.isInfoEnabled()){
            logger.info("criando a movimentacao");
        }

        Movimentacao movimentacao = new Movimentacao();
        movimentacao.setTipo(TipoMovimentacao.ENTRADA);
        movimentacao.setData(Calendar.getInstance());
        movimentacao.setProduto(produto);
        movimentacao.setQuantidade(getQuantidade(produto));

        movimentacaoRepository.salvar(movimentacao);
    }

    private Integer getQuantidade(Produto produto) {

        if(logger.isInfoEnabled()){
            logger.info("obtendo a quantidade movimentada");
        }

        Integer quantidade = produto.getQuantidade();

        List<Movimentacao> movimentacoes
            = movimentacaoRepository.getMovimentacoesDo(produto);

        for(Movimentacao movimentacao : movimentacoes) {
            if(movimentacao.isEntrada()) {
                quantidade -= movimentacao.getQuantidade();
            }
        }

        return quantidade;
    }
}
```

Este código de log é um clássico código que estará presente em diversos pontos do sistema. Código espalhado pelo sistema todo caracteriza um **aspecto**, desta forma, ao invés de ter esse código direto na classe GerenciadorProjeto e diversas outras, poderíamos isolá-lo em um aspecto que realiza esta tarefa.

Assim, deixamos o código relacionado a *logging* (*requisito ortogonal, infra estrutura*) separado da nossa classe de negócio.

Alguns outros exemplos comuns de aspectos incluem o controle transacional, auditoria, autorização e cache automático de resultados.

8.3 - Elementos de um Aspecto

A definição de um aspecto comumente é dividida em:

- **Join Point**: pontos de junção são os locais da aplicação onde os aspectos deverão ser aplicados. O exemplo mais clássico é a execução de um método, porém blocos de tratamento de excessões (try-catch) também poderiam ser considerados como *join points*.
- **Advice** é quem define o comportamento tomado pelo aspecto em um determinado *join point*. Sempre escrevemos o código dos aspectos dentro de *Advices*.
- **Pointcut** é um conjunto de *join points* a ser alvo de um particular *Advice*. Cada *Advice* deve ser associado a um *Pointcut*. Geralmente, os pointcuts são implementados como expressões regulares, mas poderiam ser qualquer tipo de coleção de *join points*.

Os *Advices* ainda podem ser ainda divididos em diversos tipos. Os mais comuns são o *Before Advice*, *After Returning Advice*, *After Throwing Advice*, *After (finally) Advice* e *Around Advice*.

8.4 - AOP com Spring

O Spring possibilita a criação de aspectos de **duas** formas: *Spring AOP* puro e usando *AspectJ*.

AspectJ é o framework para criação de aspectagem mais conhecido do mercado e fornece duas possibilidades para a criação de aspectos: linguagem própria, ou classes Java comuns com anotações especiais. A segunda alternativa também é conhecida como *@AspectJ-style*, já que a principal anotação é a *@Aspect*.

Usando *Spring AOP* puro, a definição dos aspectos pode ser feita nos arquivos xml de configuração do Spring, ou com classes Java comuns através de anotações do *AspectJ* (*@AspectJ-style*). A diferença do uso das anotações do *AspectJ* neste caso é que todo o processo de aplicação dos *advices* é feito pelo Spring. Apenas as anotações do *AspectJ* são aproveitadas.

8.5 - AOP no Spring com @AspectJ-style

Iremos inicialmente explorar a criação de aspectos usando anotações do *AspectJ*. O único requisito é configurar o Spring para ter este suporte. Para tal deveremos adicionar o namespace referente a parte de *aop* e dizer onde está o arquivo xsd associado:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-2.5.xsd"
```

```
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

<!-- ... -->

</beans>
```

Perceba que o namespace `xmlns:aop`, bem como seu xsd `spring-aop-2.5.xsd` foram adicionados. Basta agora habilitar o suporte a AOP. Para tal, existe a tag `<aop:aspectj-autoproxy/>`:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-2.5.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

    <!-- Resto do xml ... -->

    <aop:aspectj-autoproxy/>

</beans>
```

Os aspectos são simples componentes do Spring. Como já estamos usando anotações, basta criar um novo componente que será o nosso aspecto de logging:

```
import org.springframework.stereotype.Component;

@Component
public class LogAspect {
```

Para transformar este componente em um aspecto, basta agora anotá-lo com `@Aspect`. Esta anotação vem do AspectJ.

```
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LogAspect {
```

8.6 - Exercícios - AOP

- 1) Adicione a tag que habilita o suporte a AOP dentro da tag <beans>:

```
<aop:aspectj-autoproxy/>
```

- 2) Crie a classe de aspecto LoggingAspect no pacote br.com.caelum.estoque.aop:

```
package br.com.caelum.estoque.aop;

@Aspect
@Component
public class LoggingAspect {

}
```

8.7 - Definindo Pointcuts e Advices

Cada aspecto pode ter diversos Pointcuts, que podem ser definidos através da anotação @Pointcut do AspectJ. Anotações devem sempre estar associadas a um elemento, porém o pointcut não tem comportamento algum. Por este motivo, a anotação deve ser associada a um método void vazio, que serve **apenas para dar nome** ao Pointcut.

Além disso, o conjunto de join points que estão relacionados ao Pointcut é expresso através de uma sintaxe específica do AspectJ. Veja um exemplo:

```
@Aspect
@Component
public class LoggingAspect {

    @Pointcut("execution(public * *(..))")
    public void todosOsMetodosPublicos() {}

}
```

Adicionando mais de um pointcut no mesmo aspecto teríamos:

```
@Aspect
@Component
public class LoggingAspect {

    @Pointcut("execution(public * *(..))")
    public void todosOsMetodosPublicos() {}

    @Pointcut("execution(* *(..))")
    public void todosOsMetodos() {}

    @Pointcut("execution(void set*(..))")
    public void todosOsSetters() {}

}
```

A sintaxe para definição de pointcuts do AspectJ exige a definição de um tipo de pointcut. O tipo mais comum (e recomendado na maioria dos casos) é o execution, que diz respeito a join points que são execução de métodos. Os outros tipos possíveis são:

- execution(expr): inclui join points que sejam execução de métodos;

- `within(expr)`: inclui todos os join points dentro de uma classe ou pacote;
- `this(expr)`: inclui todos os join points de proxies que sejam instâncias de determinado tipo;
- `target(expr)`: inclui todos os join points de beans que sejam instâncias de determinado tipo;
- `args(expr)`: inclui todos os join points que recebam argumentos de determinado tipo;
- `@target(expr)`: inclui todos os join points de beans que sejam anotados com determinada anotação;
- `@args(expr)`: inclui todos os join points que recebam parâmetros anotados com determinada anotação;
- `@within(expr)`: inclui todos os join points que sejam de tipos que contenham determinada anotação (implementa um interface anotada, por exemplo);
- `@annotation(expr)`: inclui todos os join points anotados com determinada anotação;
- `bean(expr)`: inclui todos os join points de beans que tenham determinado nome. Este tipo de pointcut é específico do Spring e não funcionaria com AspectJ puro.

Todos os tipos de pointcuts recebem uma expressão que seleciona o quais serão os *join points* inclusos. Tais expressões podem fazer uso de *wildcards* (caracter *), dando uma flexibilidade enorme a definição de pointcuts.

Veremos com mais profundidade o tipo `execution`, que é o mais usado e adequado a grande maioria dos casos. Mais detalhes sobre os demais tipos podem ser encontrados na documentação do Spring AOP e do AspectJ:

<http://static.springframework.org/spring/docs/2.5.x/reference/aop.html>

<http://www.eclipse.org/aspectj/doc/released/progguide/index.html>

A sintaxe para definição de pointcuts do tipo `execution` segue sempre uma regra:

```
execution(modificador-acesso? retorno classe? metodo(parametros))
```

As opções que tem o ‘?’ são opcionais, além disso todas elas podem receber o caracter coringa ‘*’ em qualquer parte. O padrão para os parâmetros é diferente e funciona da seguinte forma:

- (): nenhum parâmetro
- (..): quaisquer parâmetros (zero ou mais)
- (*): apenas um parâmetro de qualquer tipo
- (*,Long,String): um de qualquer tipo, um Long e outro String, na ordem

Seguindo as regras acima, podemos criar pointcuts que representem pontos do sistema para aspectos serem aplicados, como por exemplo todos os métodos de gerenciadores:

```
@Aspect  
@Component  
public class LoggingAspect {  
  
    @Pointcut("execution(* br.com.caelum.estoque.service..*.*(..))")  
    public void metodosDeServico() {}  
  
}
```

Neste caso, todos os join points de métodos dentro de classes que estejam no pacote br.com.caelum.estoque.service e subpacotes (note o '..') entram para este pointcut definido com o nome de `metodosDeServiço()`.

Agora que temos o pointcut necessário, poderemos criar um advice dentro deste aspecto, relativo a este pointcut. Para cada tipo de advice, existe uma anotação equivalente (`@Before`, `@AfterReturning`, `@AfterThrowing`, `@After` - `finally` - e `@Around`).

```
@Aspect  
@Component  
public class LoggingAspect {  
  
    @Pointcut("execution(* br.com.caelum.estoque.service..*.*(..))")  
    public void metodosDeServiço() {}  
  
    @Before("metodosDeServiço()")  
    public void logaNoComeço() {}  
  
}
```

Devemos sempre dizer qual (ou quais) pointcut(s) está(ão) relacionado(s) ao advice que estamos criando. Repare que o argumento da anotação é o nome do pointcut associado.

Os métodos advice podem ainda **opcionalmente** receber um objeto do tipo `JoinPoint` que representa o join point onde o aspecto está sendo aplicado. Este join point guarda informações valiosas para o nosso advice de logging:

```
@Aspect  
@Component  
public class LoggingAspect {  
  
    @Before("metodosDeServiço()")  
    public void logaNoComeço(JoinPoint joinPoint) {  
        Logger logger = Logger.getLogger(joinPoint.getTarget().getClass());  
  
        String methodName = joinPoint.getSignature().getName();  
        String typeName = joinPoint.getSignature().getDeclaringTypeName();  
  
        logger.info("Executando método: " + methodName + " da classe: " + typeName);  
    }  
  
    @Pointcut("execution(* br.com.caelum.estoque.service..*.*(..))")  
    public void metodosDeServiço() {}  
  
}
```

8.8 - Exercícios

- 1) Crie o aspecto que adiciona capacidade de logging a todas as classes *Service*

```
@Aspect  
@Component  
public class LoggingAspect {  
  
    //Log4j  
    private Logger logger = Logger.getLogger(LoggingAspect.class);  
  
    @Before("metodosDeServiço()")  
    public void logaNoComeço(JoinPoint joinPoint) {  
  
        String methodName = joinPoint.getSignature().getName();  
        String typeName = joinPoint.getSignature().getDeclaringTypeName();  
  
        logger.info("Executando método: " + methodName + " da classe: " + typeName);  
    }  
  
    @Pointcut("execution(* br.com.caelum.estoque.service..*.*(..))")  
    public void métodosDeServiço() {}  
  
}
```

- 2) Para o Logger funcionar precisamos configurar o **log4j** para ele saber que deve logar nossa classe. Devemos adicionar configurar o arquivo log4j.properties que se encontra no nosso diretório “src” com a seguinte linha:

```
log4j.logger.br.com.caelum.estoque.aop=info
```

- 3) Start o tomcat e cadastre um ou dois produtos. Verifique então que no console vão ter as logadas as informações referente a execução do método `entrade()`.

8.9 - Outros Advices

Podemos também fazer o log automático do resultado de todos os métodos dos gerenciadores (mesmo pointcut de antes).

Para tal, basta definir um novo advice de outro tipo, que seja executado no fim do join point:

```
@Aspect  
@Component  
public class LoggingAspect {  
  
    @Before("metodosDeGerenciadores()")  
    public void logaNoComeço(JoinPoint joinPoint) {  
        Logger logger = Logger.getLogger(joinPoint.getTarget().getClass());  
  
        String methodName = joinPoint.getSignature().getName();  
        String typeName = joinPoint.getSignature().getDeclaringTypeName();  
  
        logger.info("Executando método: " + methodName + " da classe: " + typeName);  
    }  
  
    @AfterReturning(pointcut="metodosDeGerenciadores()")  
    public void logaOResultado(JoinPoint joinPoint) {  
  
    }  
  
    @Pointcut("execution(* br.com.caelum.fj27.gerenciador..*.*(..))")  
    public void métodosDeGerenciadores() {}  
  
}
```

A anotação @AfterReturning serve para definir um advice que é executado logo após o término dos join points. Para fazer o log do resultado, precisamos de alguma maneira conseguir capturar o resultado da execução do join point.

O interessante é que o advice pode receber um parâmetro que representa o retorno do join point. Dizemos qual dos parametros representa o retorno, através da própria anotação @AfterReturning.

```
@Aspect  
@Component  
public class LoggingAspect {  
  
    @Before("metodosDeGerenciadores()")  
    public void logaNoComeco(JoinPoint joinPoint) {  
        Logger logger = Logger.getLogger(joinPoint.getTarget().getClass());  
  
        String methodName = joinPoint.getSignature().getName();  
        String typeName = joinPoint.getSignature().getDeclaringTypeName();  
  
        logger.info("Executando metodo: " + methodName + " da classe: " + typeName);  
    }  
  
    @AfterReturning(pointcut="metodosDeGerenciadores()", returning = "retorno")  
    public void logaResultado(JoinPoint joinPoint, Object retorno) {  
        Logger logger = Logger.getLogger(joinPoint.getTarget().getClass());  
  
        String methodName = joinPoint.getSignature().getName();  
        String typeName = joinPoint.getSignature().getDeclaringTypeName();  
  
        logger.info("Retorno do método: " + methodName + " da classe: " + typeName +  
                   " foi: " + retorno);  
    }  
  
    @Pointcut("execution(* br.com.caelum.fj27.gerenciador..*.*(..))")  
    public void metodosDeGerenciadores() {}  
}
```

O parâmetro que representa o retorno pode ser de qualquer tipo. Porém, o Spring AOP só aplicará o aspecto nos métodos que tiverem retorno compatível, mesmo que o método esteja incluído no pointcut associado. Para fazer com que o advice seja aplicado a qualquer join point, usamos Object como tipo de retorno aceito.

Diferenças entre @AfterReturning e @After

Advices do tipo @AfterReturning só são executados caso o join point termine com sucesso. Se houver alguma exception, advices do tipo @AfterThrowing são executados. Advices do tipo @After são executados sempre, independente da ocorrência de exceptions.
Por este motivo, advices @After são conhecidos como do tipo *finally*.

8.10 - Para saber mais: Combinando Pointcuts

Pointcuts complexos podem ser construidos combinando outros mais simples através de operações && e || simples.

```
@Pointcut("execution(public * *(..))")  
public void metodosPublicos() {}  
  
@Pointcut("execution(void set*(*))")  
public void comecamComSetERecebemSoUmParametro() {}  
  
@Pointcut("metodosPublicos() && comecamComSetERecebemSoUmParametro()")  
public void setters() {}
```

Podemos ainda definir o conjunto de join points diretamente na definição do advice, sem precisar criar o pointcut explicitamente:

```
@Before("execution(public Session br.com.caelum.fj27..*.getSession(..))")  
public void fazAlgoAntesDeMetodosQueCriamSession() {  
    // ...  
}
```

8.11 - Para saber mais: @Around

Os advices do tipo @Around são os mais completos, já que permitem a execução de código antes e depois da execução do join point. Além disso, é o único que pode impedir (barrar) a execução do join point.

Advices deste tipo devem sempre ter como primeiro argumento um ProceedingJoinPoint. Através deste tipo específico de JoinPoint, é possível controlar a execução (ou não) do join point a qual o advice está sendo aplicado.

Advices @Around podem também modificar o retorno original e retornar algo diferente.

```
@Around  
public Object controleDeAcesso(ProceedingJoinPoint joinPoint) {  
    if(podeExecutar()) {  
        return joinPoint.proceed();  
    } else {  
        throw new IllegalAccessException();  
    }  
}
```

Apêndice - Integração entre Sistemas

9.1 - Spring Integration

Spring também fornece uma infraestrutura própria para o envio de mensagens sem a necessidade de usar JMS. Tudo configurável dentro do Spring com ou sem anotações. O framework se chama **Spring integration**.

Spring integration oferece um modelo completo para criar aplicações baseado em troca de mensagens. Ele encapsula a complexidade relacionada com o envio e recebimento de mensagens. As classes envolvidas são business components comuns e podem aproveitar da infraestrutura que Spring oferece, como IoC ou AOP.

Para habilitar as funcionalidades do Spring Integration é preciso registrar o namespace correto:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:integration="http://www.springframework.org/schema/integration"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-2.5.xsd
           http://www.springframework.org/schema/integration
           http://www.springframework.org/schema/integration/spring-integration-1.0.xsd">
```

A novidade aqui é o namespace `integration` junto com a declaração do schema. Registrando isso podemos habilitar as anotações do Spring Integration. Além disso, é necessário avisar que qualquer canal de comunicação pode ser criado automaticamente:

```
<integration:annotation-driven />
<integration:message-bus auto-create-channels="true"/>
```

O próximo passo é criar uma classe que funcione como um filtro da comunicação. Um Handler recebe uma mensagem de entrada e devolve uma mensagem para a saída. Entretanto o handler pode transformar a mensagem, rotear a outro lugar, executar qualquer regra de negócio ou parar o envio da mensagem.

Um Handler usa a anotação `@MessageEndpoint` informando qual é a entrada e saída. Por exemplo:

```
@MessageEndpoint(input="inputChannel", output = "outputChannel")
public class MessageHandler {
    //...
}
```

Aqui a entrada é representada pelo `inputChannel` e a saída pelo `outputChannel`. Se os canais de comunicação não existem ainda, o Spring Integration faz a criação automaticamente (`auto-create-channels="true"`).

O método que recebe a mensagem deve usar a anotação `@Handler`. A mensagem pode ser qualquer objeto e o nome do método pode ser escolhido livremente. Por exemplo:

```
@MessageEndpoint(input="inputChannel", output = "outputChannel")
public class MessageHandler {

    @Handler
    public Movimentacao handle(Movimentacao movimentacao) {
        System.out.println("Roteando movimentacao: " + movimentacao);
        //se devolver nulo o subscriber nao recebe a mensagem
        return movimentacao;
    }
}
```

Neste exemplo fica clara a simplicidade, estamos recebendo um projeto (a mensagem de ida) e devolvendo um projeto (a mensagem de retorno).

Falta criar uma classe que fica escutando a saída e recebendo, finalmente, a mensagem. Esta classe é um bean comum do Spring que tem um método para receber a mensagem. É preciso usar a anotação `@Subscriber`. Veja o exemplo:

```
@Component
public class MovimentacaoListener {

    @Subscriber(channel="outputChannel")
    public void onMessage(Movimentacao movimentacao) {
        System.out.println("Chegou movimentacao: " + movimentacao);
        //mais código aqui
    }
}
```

O bean define com a anotação `@Subscriber(channel="outputChannel")` qual é o canal para escutar. Isto é parecido com um `MessageListener` da especificação JMS. O nome do método e o tipo do parâmetro podem ser escolhidos livremente desde que o parâmetro seja compatível com o retorno do handler (`Movimentacao handle(Movimentacao movimentacao)`).

O último passo é criar e enviar a mensagem. Para o envio é preciso usar um objeto Spring do tipo `ChannelRegistry`, que dá acesso a qualquer canal de comunicação. Podemos injetar o bean:

```
@Autowired
private ChannelRegistry channelRegistry;
```

Usando o `ChannelRegistry` temos que acessar a entrada para enviar a mensagem:

```
MessageChannel inputChannel = channelRegistry.lookupChannel("inputChannel");
inputChannel.send(new GenericMessage<Movimentacao>(movimentacao));
```

A classe `GenericMessage` é um wrapper que serve como um envelope para o conteúdo original (produto).

9.2 - Exercício: Spring Integration

1) Adicione no buildpath do seu projeto os dois jars:

- org.springframework.integration.adapter-1.0.0.M4.jar
- org.springframework.integration-1.0.0.M4.jar

Os jar se encontram na pasta caelum/27/integration/

2) Adicione no app-config.xml:

```
<integration:annotation-driven />
<integration:message-bus auto-create-channels="true"/>
```

3) Crie a classe MessageHandler que representa o roteador no pacote br.com.caelum.estoque.integration:

```
@MessageEndpoint(input="inputChannel", output = "outputChannel")
public class MessageHandler {

    @Handler
    public Produto handle(Movimentacao movimentacao) {
        System.out.println("Roteando movimentacao: " + movimentacao);
        return movimentacao;
    }
}
```

4) Crie a classe MovimentacaoListener no mesmo pacote:

```
@Component
public class MovimentacaoListener {

    @Subscriber(channel="outputChannel")
    public void onMessage(Movimentacao movimentacao) {
        System.out.println("Chegou movimentacao: " + movimentacao);
    }
}
```

5) Altere a classe GeraMovimentacao. Injete o ChannelRegistry:

```
@Autowired
private ChannelRegistry channelRegistry;
```

6) Altere o método entrada(). Crie e envie uma mensagem:

```
///...
```

```
public void entrada(Produto produto) {
    Movimentacao movimentacao = new Movimentacao();
    movimentacao.setTipo(TipoMovimentacao.ENTRADA);
    movimentacao.setData(Calendar.getInstance());
    movimentacao.setProduto(produto);
    movimentacao.setQuantidade(getQuantidade(produto));

    MessageChannel inputChannel = channelRegistry.lookupChannel("inputChannel");
    inputChannel.send(new GenericMessage<Movimentacao>(movimentacao));

    movimentacaoRepository.salvar(movimentacao);
}
///...
```


Apêndice - Enviando Email

10.1 - Enviando E-mail com Spring

O Spring fornece uma camada de abstração para o envio de email. Através da interface MailSender podemos acessar as funcionalidades do Spring.

A implementação da interface é a classe JavaMailSenderImpl que pode ser definida como um bean no arquivo applicationContext.xml:

```
<bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="host" value="smtp.servidorDeEmail.com" />
    <property name="username" value="nomeDoUsuario" />
    <property name="password" value="senhaDoUsuario" />
    <property name="protocol" value="smtps" />
    <property name="port" value="465" />
</bean>
```

Assim é possível injetar o bean num componente do Spring, por exemplo através da anotação @Autowired:

```
@Service
public class GeraMovimentacao {

    private MailSender mailSender;

    //...
}
```

Repare que usamos a interface org.springframework.mail.MailSender, para ficar mais desacoplado.

Além disso, podemos preencher um email com valores padrões já no XML. A classe que representa um email se chama SimpleMailMessage:

Veja a declaração no XML:

```
<bean id="mailMessage" class="org.springframework.mail.SimpleMailMessage">
    <property name="subject" value="email gerado" />
    <property name="from" value="email@servidor.com" />
</bean>
```

Agora só falta injetar a mensagem de email preparada:

```
@Autowired
private MailMessage message;
```

Aqui também usamos a interface org.springframework.mail.MailMessage em vez da implementação declarado no XML.

10.2 - Exercício: Spring Mail

1) Adicione o `mail.jar` no buildpath do seu projeto. O jar se encontra na pasta `lib/j2ee` da distribuição do Spring.

2) Defina o bean para o envio de mensagens. Preencha as propriedades do `host`, `username` e `password` com os valores do seu provedor.

```
<bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="host" value="smtp.servidorDeEmail.com" />
    <property name="username" value="nomeDoUsuario" />
    <property name="password" value="senhaDoUsuario" />
    <property name="protocol" value="smtps" /> <!-- smtp -->
    <property name="port" value="465" />
    <property name="javaMailProperties">
        <props>
            <prop key="mail.smtps.auth">true</prop>
            <prop key="mail.smtps.starttls.enable">true</prop>
            <prop key="mail.smtps.debug">true</prop>
        </props>
    </property>
</bean>
```

3) Defina um bean que representa um email. Preencha um subject padrão:

```
<bean id="mailMessage" class="org.springframework.mail.SimpleMailMessage">
    <property name="subject" value="enviando email com Spring" />
</bean>
```

4) Injete os beans `mailMessage` e `mailSender` na classe `GerenciadorTarefa`. Adicione o código para enviar o email no método novo.

```
//...

public void entrada(Produto produto) {
    Movimentacao movimentacao = new Movimentacao();
    movimentacao.setTipo(TipoMovimentacao.ENTRADA);
    movimentacao.setData(Calendar.getInstance());
    movimentacao.setProduto(produto);
    movimentacao.setQuantidade(getQuantidade(produto));

    MessageChannel inputChannel = channelRegistry.lookupChannel("inputChannel");
    inputChannel.send(new GenericMessage<Movimentacao>(movimentacao));

    enviaEmail();

    movimentacaoRepository.salvar(movimentacao);
}

private void enviaEmail(){
    this.message.setTo("para@servidor.com");
    this.message.setSubject("nova tarefa criada");
    this.message.setText("descricao");

    this.mailSender.send(this.message);
}

//...
```

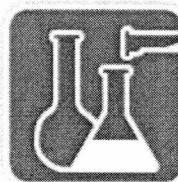
Conheça alguns de nossos cursos



FJ-11:
Java e Orientação a
objetos



FJ-25:
Persistência com JPA2 e
Hibernate



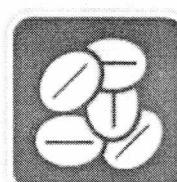
FJ-16:
Laboratório Java com Testes,
XML e Design Patterns



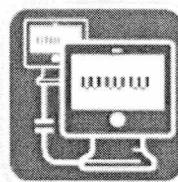
FJ-26:
Laboratório Web com JSF2 e
CDI



FJ-19:
Preparatório para Certificação
de Programador Java



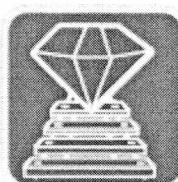
FJ-31:
Java EE avançado e
Web Services



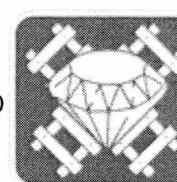
FJ-21:
Java para Desenvolvimento
Web



FJ-91:
Arquitetura e Design de
Projetos Java



RR-71:
Desenvolvimento Ágil para Web
2.0 com Ruby on Rails



RR-75:
Ruby e Rails avançados: lidando
com problemas do dia a dia

Para mais informações e outros cursos, visite: caelum.com.br/cursos

- ✓ Mais de 8000 alunos treinados;
- ✓ Reconhecida nacionalmente;
- ✓ Conteúdos atualizados para o mercado e para sua carreira;
- ✓ Aulas com metodologia e didática cuidadosamente preparadas;
- ✓ Ativa participação nas comunidades Java, Rails e Scrum;
- ✓ Salas de aula bem equipadas;
- ✓ Instrutores qualificados e experientes;
- ✓ Apostilas disponíveis no site.