

Vire o jogo com

Spring Framework



Casa do
Código

HENRIQUE LOBO WEISSMANN

Prefácio

Neste livro trataremos do Spring, um framework usado no desenvolvimento de aplicações corporativas baseado nos conceitos de *inversão de controle*, *injeção de dependências* e AOP. São palavras que muitas vezes soam alienígenas e muitas vezes não adquirem significado na mente do desenvolvedor. Este livro possui dois objetivos: clarificar o significado por trás destes conceitos e expor como sua aplicação acaba por gerar sistemas de qualidade superior, mais fáceis de manter e que, com isto, acabam por aumentar o valor agregado do nosso trabalho.

Mais que um framework para desenvolvimento de aplicações corporativas, vejo o Spring como uma ferramenta disciplinadora. Conforme o desenvolvedor vai se habituando ao seu modo de trabalho começa a valorizar ainda mais qualidades como uma melhor modularização do sistema, escrita de código mais simples, reaproveitamento de código legado e tecnologias já existentes, além da criação de interfaces mais significativas. No Spring podemos ver de forma nítida o núcleo da arquitetura de sistemas que é justamente o modo como componentes de software interagem entre si de uma maneira explícita. Para os que anseiam trilhar este caminho, é também um auxílio em sua formação.

O livro é dividido em duas partes. Na primeira tratamos dos conceitos em que se baseia o Spring. Veremos o que motivou sua criação, o que o framework trouxe de ganho para o desenvolvimento de aplicações corporativas e como são aplicados no núcleo do Spring que é seu container de inversão de controle/injeção de dependências. Com isto é fornecida ao leitor uma base sólida que pode ser aplicada tanto dentro quanto fora do contexto do Spring.

Na segunda parte temos uma abordagem mais *mão na massa*. Veremos aplicações do framework em situações reais do dia a dia, como por exemplo na camada de persistência, controle transacional, segurança e também criaremos uma aplicação web real baseada no Spring MVC. A grosso modo, pode-se dizer que quando escrevi a primeira parte do livro incluí o conteúdo que gostaria muito de ter aprendido na

faculdade, enquanto na segunda estão as informações que teriam me poupado muito tempo no meu cotidiano como desenvolvedor.

A quem se destina

Este livro se destina a todos aqueles que já possuem algum conhecimento da plataforma Java, que já terão uma bagagem mais rica para entender o que tento explicar neste texto. Caso não seja o seu caso, tudo bem: os três primeiros capítulos contêm o conceitual que você poderá aplicar em praticamente qualquer plataforma.

Agradecimentos

Agradeço à confiança e ao apoio (isto sem mencionar a paciência!) dos meus editores Paulo Silveira e Adriano Almeida sem os quais este livro não seria possível. Claro que eles não agiram sozinhos, Nanna - minha esposa - é fundamental por ter conseguido me aguentar neste processo de tradução do *Springolês* para o *Português*. Também preciso agradecer aos meus colegas, em especial ao Matheus Eduardo Moreira e o Felipe Zampa.

Os leitores tiveram participação extremamente importante: após o lançamento da versão beta fiquei maravilhado com a quantidade de pessoas interessadas neste material, o que me motivou a concluí-lo mais rápido e com a melhor qualidade possível. Alguns entraram em contato direto comigo com dicas e sugestões, e a estes ofereço meu agradecimento especial: Renan Reis, Rodrigo Monteiro, Adriano Faria Alves e Willardy Tyrone de Oliveira. E ei, também não posso deixar de agradecer ao apoio do Rodrigo Fernandes Moreira e Gabriela Corrêa da Eteg. Finalmente, para evitar que eu sofra consequências imprevisíveis, à minha mãe, Selma Weissmann. Infelizmente o espaço deste prefácio é limitado para conter todas as pessoas a quem devo agradecer, assim como minha memória. Caso seu nome me tenha escapado, por favor desculpe este mal agradecido: com certeza encontrarei alguma forma de retribuí-lo no futuro ok?

Muito obrigado a todos vocês.

Sumário

Os conceitos por trás do Spring	1
1 Lide com o alto acoplamento de forma elegante	3
1.1 Problema essencial: acoplamento	4
1.2 A famigerada inversão	10
2 Conheça o Spring Framework	15
2.1 Por que criaram o Spring?	15
2.2 O Mundo dos objetos em 2004	16
2.3 Os problemas do EJB em 2004	18
2.4 Alternativas começam a surgir: IoC/DI e AOP	19
2.5 Container: o sujeito que torna tudo isto possível	22
2.6 Do início ao fim de um objeto, entenda o ciclo de vida	22
2.7 Spring em partes	27
2.8 O Container	28
2.9 Trabalhando com AOP e Aspects	29
2.10 Instrumentação de código	29
2.11 Acesso a dados e integração	30
2.12 Aplicações na Web com o Spring	30
2.13 E ainda mais componentes!	31
2.14 Resumindo	31
3 Conhecendo o Container	33
3.1 Preparando seu ambiente de desenvolvimento	34
3.2 Nosso sistema de exemplo	37
3.3 Declarando beans	37

3.4	Instanciação por factory method	43
3.5	Mapeando atributos complexos	46
3.6	Usando o container	49
3.7	O ciclo de vida do container	52
3.8	Escopos	53
3.9	Instanciação tardia	57
3.10	Aproveitando o ciclo de vida dos beans	58
3.11	Quando o bean conhece seu container	61
3.12	Modularizando a configuração	63
3.13	Aplicando herança na definição dos beans	64
3.14	Spring Expression Language (SpEL)	65
3.15	Resumindo	70
4	Minimizando o XML com autowiring, anotações e Java	71
4.1	Autowiring: automatizando a injeção de dependências	71
4.2	Vantagens e limitações da injeção automática	76
4.3	Facilitando ainda mais com Anotações	77
4.4	Configuração programática com Java	86
4.5	XML, anotações ou configuração programática?	93
5	AOP: adicionando novos comportamentos aos beans	95
5.1	Como identificar os interesses transversais	96
5.2	Entendendo AOP	98
5.3	Preparando o ambiente de desenvolvimento	100
5.4	A implementação do primeiro aspecto	102
5.5	Entenda os advices	103
5.6	Use os around advices	104
5.7	Use o advice before	106
5.8	E onde entra o Proxy nessa história?	108
5.9	Entenda a execução	110
5.10	Um uso interessante para AOP: Log de erros	111
5.11	Declare aspectos com anotações AspectJ e esquemas XML	112
5.12	A sintaxe AspectJ de declaração de point cuts	116
5.13	Concluindo com um resumo conceitual	121

Spring Framework na prática	123
6 Colocando a mão na massa	125
6.1 Obtendo o código fonte	126
6.2 Configure seu ambiente de desenvolvimento	127
6.3 O que vêm por aí	128
7 Desenvolva aplicações web com Spring MVC	131
7.1 A base para tudo: MVC	132
7.2 Dispatcher Servlet: o maestro por trás do Spring MVC	133
7.3 A preparação do projeto	135
7.4 Como trabalhar com conteúdo estático	140
7.5 Nosso primeiro controlador	142
7.6 A definição da camada de visualização	144
7.7 Trabalhe com redirecionamentos e sessão do usuário	152
7.8 A definição do método de acesso	153
7.9 Receba parâmetros de entrada	154
7.10 Faça redirecionamentos	154
7.11 Como lidar com a sessão do usuário e a classe ModelAndView	155
7.12 O chato e repetitivo trabalho de criar formulários	157
7.13 Ainda há mais de SpringMVC por vir	161
8 Ações recorrentes com o SpringMVC	163
8.1 Faça a validação de formulários	164
8.2 Envie seus avatares para o servidor com upload de arquivos	168
8.3 Defina o corpo da resposta	171
8.4 Faça requisições assíncronas com AJAX	173
8.5 Concluindo com uma surpresa	176
9 Acesso a dados	179
9.1 DAO: no centro da mentalidade Spring	180
9.2 Conecte-se ao banco de dados	181
9.3 De volta ao DAO: o problema com as exceções	186
9.4 Templates: acabe com o código repetido	188
9.5 O trabalho com JDBC	190

9.6	O trabalho com o Hibernate	200
9.7	Integre com a JPA	207
9.8	Concluindo	211
10	Gerenciando transações	213
10.1	Entendendo as transações	213
10.2	As políticas transacionais	215
10.3	Preparando o ambiente de desenvolvimento	219
10.4	Como o Spring implementa o suporte a transações	219
10.5	Transações programáticas	221
10.6	Declarando transações	223
10.7	Concluindo	225
11	Protegendo nossa aplicação com Spring Security	227
11.1	Entendendo autenticação e autorização	228
11.2	Os módulos do Spring Security	229
11.3	Configurando os filtros de acesso	231
11.4	Configurando o contexto do Spring	233
11.5	O formulário de login	237
11.6	Escrevendo um provedor de autenticação	239
11.7	Usando SpEL	242
11.8	Protegendo invocação de métodos em beans	243
11.9	Tags	244
11.10	Conclusão	245
12	E ai, gostou?	247
	Bibliografia	252

Parte I

Os conceitos por trás do Spring

CAPÍTULO 1

Lide com o alto acoplamento de forma elegante

Todo objeto é carente: e aí o problema começa.

Spring Framework é um framework voltado para o desenvolvimento de aplicações corporativas para a plataforma Java, baseado nos conceitos de inversão de controle e injeção de dependências. Esta é a descrição padrão que encontramos na grande maioria dos materiais sobre o assunto.

Vamos fazer diferente? Que tal não simplesmente descrever o funcionamento da ferramenta, mas sim apresentar os problemas que motivaram a sua criação? São problemas comuns no desenvolvimento de sistemas que ocorrem independentemente da plataforma e o Spring no final das contas é apenas uma ferramenta interessante para tratá-los.

Estas dificuldades podem ser agrupadas em duas categorias: *essenciais* e *específicos da plataforma*. Neste capítulo o objetivo é apresentá-las brevemente para, no transcorrer deste livro, entender como o Spring nos ajuda a minimizá-las.

1.1 PROBLEMA ESSENCIAL: ACOPLAMENTO

Problemas essenciais são aqueles que independem da plataforma. Fred Brooks no seu texto clássico “No Silver Bullet” [2] aponta quatro **dificuldades essenciais** inerentes ao desenvolvimento de qualquer software:

- **Conformidade:** todo software precisa ser compatível com o ambiente no qual será executado. Isto inclui conformidade com sistema operacional, acesso a interfaces de sistemas legados e as próprias restrições físicas do ambiente.
- **Invisibilidade:** como podemos visualizar um software? Apesar dos avanços em notações como a UML ainda não chegamos a uma solução definitiva para este problema. Boa parte do nosso raciocínio se baseia em metáforas visuais que dificilmente se enquadram no desenvolvimento de projetos de software.
- **Mutabilidade:** os requisitos mudam constantemente e, com isto, faz-se necessário que a arquitetura de nossos sistemas seja projetada de tal forma que possa se adequar com o mínimo de impacto possível a estas transformações.
- **Complexidade:** provavelmente a pior das dificuldades, pois está diretamente relacionada aos limites do nosso intelecto. Conforme o software se desenvolve a complexidade das regras de negócio e das integrações que este precisa fazer evoluem até chegar a um ponto no qual os custos de manutenção se tornam proibitivos (o livro clássico de Fred Brooks, “The Mythical Man-Month” [1] pode ser resumido a esta questão).

A orientação a objetos quando bem aplicada consegue minimizar estes problemas essenciais através da metáfora do objeto. É uma ideia simples: se vivemos em um mundo rodeado de objetos interagindo entre si, por que não metaforizá-los em uma linguagem de programação que nos permita **simular** estas interações a fim de resolver problemas computacionais? Um plano perfeito no qual nada pode dar errado, não é? Claro que não.

Minimizamos o problema da conformidade a partir de interfaces bem definidas, com relação à invisibilidade, temos a UML, e a própria metáfora do objeto já é *quase* visual. Mutabilidade? Sempre podemos adicionar mais uma classe pra modificar algum comportamento, e com relação à complexidade, bem, agora temos classes e as coisas ficaram mais simples, certo? Não.

Se ignoramos o fato de que este relacionamento entre objetos não é simples, terminamos reféns de uma *armadilha complexa* da qual possivelmente a única saída

normalmente é a reescrita total do nosso código. Se já há complexidade no desenvolvimento procedural, com objetos em cena acabamos por adicionar também a complexidade inerente do gerenciamento de suas interações. Nosso *plano perfeito* falhou, e Robert C. Martin (aka “Uncle Bob”) em seu texto “The Dependency Inversion Principle” [10] aponta alguns sintomas de que nosso projeto pode estar apodrecendo. Estes são:

- **Fragilidade:** altere um trecho, veja diversos outros se partirem.
- **Rigidez:** consequência da fragilidade, o software é difícil de ser modificado. Neste caso, tanto devido à sua fragilidade quanto por razões não técnicas. A própria gerência da empresa começa a barrar mudanças no sistema por estar traumatizada por um passado doloroso de bugs que surgem da solução de outros.
- **Imobilidade:** há componentes maravilhosos no software, que poderiam ser aproveitados em outros projetos. Infelizmente, sua remoção é tão custosa que fica mais barato simplesmente reimplementar a funcionalidade (isto sem mencionar que agora você tem de dar manutenção em mais de um lugar)

O que torna um projeto frágil, rígido e imóvel é a maneira como se encontra definida a interdependência entre seus componentes [10]. A solução para evitar estes sintomas é desenharmos nosso software de tal forma que consigamos gerenciar de maneira efetiva o modo como seus objetos se relacionam. Dizemos que um sistema possui **alto acoplamento** quando suas classes dependem muito umas das outras. Quanto mais uma classe sabe a respeito de outra, maior é o acoplamento entre as duas e, conseqüentemente, alterações em uma normalmente acarretam em mudanças de comportamento em locais inesperados do sistema.

Para expor o problema, vou iniciar com um exemplo bastante simples: uma integração. Nosso integrador é composto por basicamente três classes: um DAO que busca informações a partir de documentos no formato XML, o Integrador em si, que possui algoritmos fantásticos de otimização (o que o torna um candidato para reutilização) e um banco de dados relacional, que é o destino das informações que extraímos do nosso DAO (neste exemplo iremos ignorar classes de domínio a fim de manter a simplicidade).

O QUE É UM DAO?

DAO significa *Data Access Object*. É uma estratégia de implementação que tem como objetivo separar a lógica de acesso a dados das regras de negócio de sua aplicação. Eis a ideia: escreva o código de acesso a dados em uma classe separada que implemente uma interface. Faça com que o resto do seu sistema tenha acesso direto apenas a esta interface. Precizou trocar a fonte de dados? Crie uma nova implementação desta interface e pronto: não precisa mais alterar sua classe cliente.

Como seu sistema saberá qual implementação usar? É o assunto deste livro!

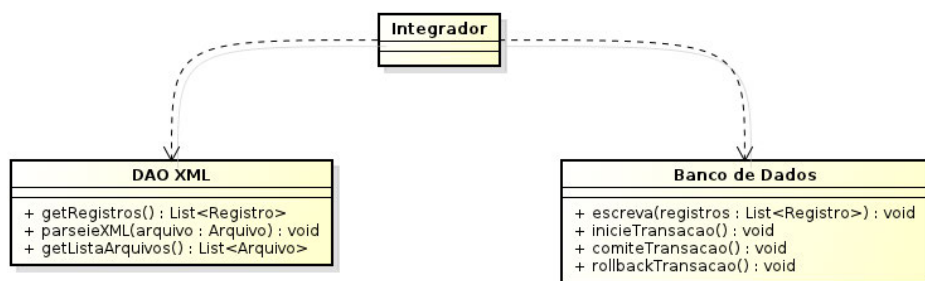


Figura 1.1: A primeira versão de nosso integrador

Um termo muito comum quando falamos sobre o Spring Framework é *dependência*. Nosso Integrador possui duas, que são as classes DAO XML e Banco de Dados, ou seja, para que nosso integrador exista, estas duas classes **obrigatoriamente** precisam estar presentes.

Em um primeiro momento nosso sistema parece perfeito. O que poderia dar errado? Nada, a não ser três das quatro dificuldades expostas por Fred Brooks:

- **Conformidade:** tanto a classe de acesso a banco de dados quanto nosso DAO que acessa documentos XML precisam estar de acordo com as limitações impostas respectivamente pelo sistema gerenciador de banco de dados e pelas

restrições de esquema XML e do sistema de arquivos. Isto já era esperado, o problema é que a classe *Integrador* possui acesso direto aos detalhes de implementação de suas dependências.

- **Mutabilidade:** e se no futuro houver a necessidade de buscarmos dados não de arquivos XML, mas de outro banco de dados ou se o destino não for mais um banco de dados, mas um web service? Ainda pior: e se o cliente quiser decidir qual fonte e destino usar em tempo de execução?
- **Complexidade:** o *Integrador* além de lidar com as especificidades do seu maravilhoso algoritmo de processamento de dados também precisa lidar com as idiossincrasias das fontes de origem e destino dos dados.

A classe *Integrador* possui alto nível de acoplamento em relação aos detalhes de implementação tanto da origem quanto do destino dos dados. Qualquer mudança em uma das dependências de *Integrador* e seu funcionamento pode ser comprometido. A qualidade de um software é inversamente proporcional ao grau de acoplamento de seus componentes.

Ainda há tempo de salvar este sistema. O que precisamos fazer é incluir *abstrações*. O que vêm a ser uma abstração? É o resultado do processo no qual, a partir da análise dos diversos usos de uma categoria de objetos, começamos a observar comportamentos que sempre estão presentes. Podemos pensar em interfaces Java e classes abstratas como abstrações, pois estas entidades contém comportamentos (leia-se métodos) que serão comuns a todas as classes que as implementem/realizem.

Vamos clarear o conceito de abstração, observando a segunda versão do nosso sistema de integração facilite:

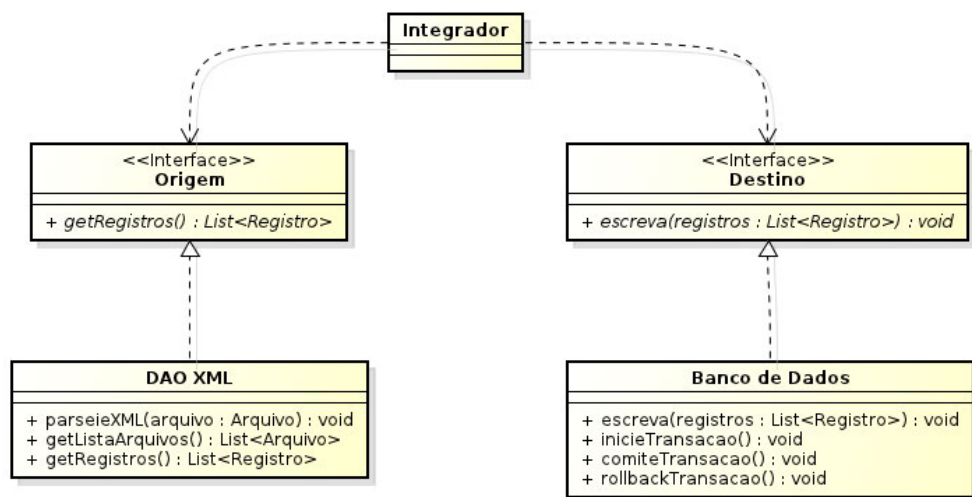


Figura 1.2: A segunda versão de nosso integrador

As interfaces *Origem* e *Destino* são abstrações. A classe *Integrador* não precisa conhecer detalhes sobre parseamento de arquivos XML ou a respeito de transações em bancos de dados. Só precisa saber que existe uma função que lhe retorna uma lista de registros a partir de uma origem e de um método que possa ser usado para persistir o resultado do seu processamento. Repare que fizemos uma abstração aqui ao identificarmos apenas o comportamento estritamente necessário para que uma origem seja uma *origem* e um destino um *destino*. Como resultado, reduzimos significativamente o acoplamento em nosso sistema, visto que agora o *Integrador* só tem acesso ao que **de fato** lhe interessa.

Você pode se perguntar neste momento: *"como a complexidade do meu sistema foi reduzida se na prática acabaram de ser introduzidos dois elementos a mais no design original?"*. Bem, primeiro o encapsulamento está maior, pois a classe *Integrador* não possui mais conhecimento a respeito de detalhes de implementação tanto da origem quanto do destino dos dados. Segundo, como o *Integrador* não sabe com qual fonte/destino de dados está lidando, **qualquer** classe que implemente as interfaces *Origem* ou *Destino* pode ser usada e por último, a mudança de comportamento em tempo de execução, onde na primeira versão do sistema, caso fosse necessário incluir uma nova fonte de dados, era necessário recompilar nosso código. Agora não mais, basta que nosso *Integrador* decida qual implementação

deseja usar.

Na realidade, nosso sistema adquiriu uma flexibilidade até então inexistente. Qualquer um pode implementar tanto uma fonte quanto uma origem de dados para o sistema. Na imagem abaixo podemos ter uma noção desta flexibilidade.

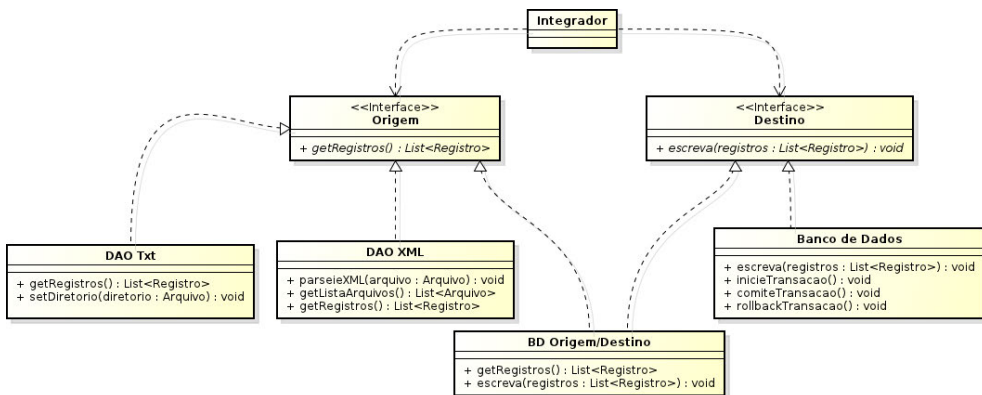


Figura 1.3: Possibilidades infinitas para nosso sistema

O que fizemos foi na realidade aplicar o “Princípio da Inversão de Dependências” [10], onde **Módulos de alto nível não devem depender de módulos de baixo nível, apenas devem depender de abstrações e Abstrações não devem depender de detalhes e sim detalhes devem depender de abstrações.**

Módulos de alto nível são aqueles que implementam o que realmente interessa em nosso projeto, ou seja, a lógica de negócio. Em nosso exemplo, este corresponde à classe Integrador. As classes que implementam as interfaces Origem e Destino são de nível mais baixo, pois não estão diretamente relacionadas ao objetivo da nossa aplicação, operando na prática a função de suporte à classe principal.

Ao mesmo tempo, vemos nas figuras 1.2 e 1.3 que as duas partes da regra se aplicam: tanto o Integrador como nossas fontes de dados dependem apenas de nossas abstrações. E nossas abstrações, dependem de quem? Apenas de si mesmas.

Está resolvido o problema do alto acoplamento do ponto de vista estático. A questão que sobra neste momento é: e do ponto de vista dinâmico, isto é, como é feita a instanciação de nossas classes e como estas serão repassadas à classe cliente?

1.2 A FAMIGERADA INVERSÃO

“Inversão de Controle”, “Princípio da Inversão de Dependências”, que *inversão* é esta? É a aplicação do **princípio de Hollywood**, cujo nome é baseado em um clichê comum naquela cidade. Quando atores concorrem por um papel, os produtores responsáveis pela peça/filme costumam dizer aos participantes do processo seletivo: “não nos chame, deixe que nós chamamos você”. Assim os produtores evitam ter suas linhas telefônicas inutilizadas por atores ansiosos por uma resposta.

Todo framework é na realidade a realização da *inversão de controle*. Pense em um servlet: você não precisa se preocupar em implementar um loop que fique verificando a todo momento a chegada de um sinal de uma porta. Tudo o que você precisa fazer é implementar um método de *callback* que será executado sempre que a porta for ativada. O mesmo princípio é aplicado na criação de uma interface gráfica em Swing: o desenvolvedor não precisa ficar verificando a todo momento se o usuário clicou em um botão: basta que você implemente as ações a serem executadas quando alguém o pressionar.

Quando aplicamos o conceito de inversão de controle, estamos na realidade transferindo a responsabilidade de verificar a ocorrência de eventos no sistema para um sistema maior, implementado por alguém que muito provavelmente entende este domínio estrutural muito melhor que eu ou você. E com isto escrevemos menos código, pois nos livramos deste fardo, passando a, em teoria, nos preocuparmos apenas com o que realmente nos interessa que é a lógica de negócio.

Porém o termo *inversão de controle*, quando aplicado ao nosso caso que é o Spring não é uma boa alternativa pois, como foi observado nos exemplos acima, qualquer framework implementa este padrão. Sendo assim, a descrição que expus no primeiro parágrafo se mostra completamente vazia: “(...) framework baseado nos conceitos de inversão de controle/injeção de dependências (...)”. Quem aponta este erro conceitual foi Martin Fowler no seu texto *Inversion of Control Containers and the Dependency Injection Pattern* [5].

Se todo framework é a aplicação da inversão de controle, dizer que o Spring se baseia neste princípio é um pleonasmo. Uma descrição melhor seria portanto a de que é um framework baseado no conceito de *injeção de dependências*, que é uma especialização da inversão de controle. Pronto: agora temos uma descrição mais interessante. Na injeção de dependências, não é a classe cliente a responsável por definir quais serão suas dependências. Esta responsabilidade é delegada a um *container de injeção de dependências*.

Para entender melhor o que vêm a ser um container, vamos partir para a quarta versão do nosso sistema, que pode ser vista na figura 1.4:

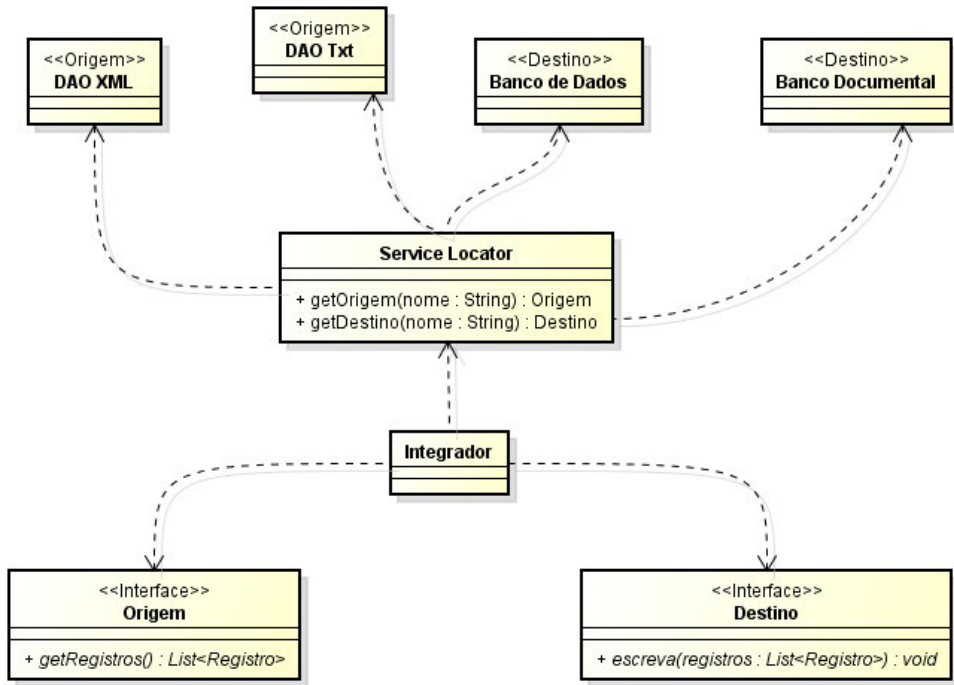


Figura 1.4: A quarta versão de nosso integrador

Nesta nossa nova evolução, foi incluído um novo elemento em nosso sistema: o **Service locator**. Esta classe implementa um padrão de projeto do mesmo nome, cuja função é localizar instâncias de classes que implementem determinados serviços (leia interfaces ou classes abstratas). Nesta nova roupagem do nosso sistema, a classe **Integrador** delegaria ao **Service Locator** a função de encontrar a implementação correta dos serviços de Entrada e Saída de dados. Podemos imaginar que nosso projeto possua um arquivo de configuração que é lido em sua inicialização, e seria o responsável por efetuar as buscas necessárias, tal como exemplificado abaixo:

```
public class Integrador {

    public Integrador() throws Exception {
```

```
Properties configuracao = new Properties();
InputStream stream =
    new FileInputStream("configuracao.properties");
configuracao.load(stream);

String nomeOrigem = configuracao.getProperty("nome_origem");
String nomeDestino = configuracao.getProperty("nome_destino");

ServiceLocator locator = new ServiceLocator()
setOrigem(locator.getOrigem(nomeOrigem);
setDestino(locator.getDestino(nomeDestino);
}

// demais métodos
}
```

Não precisamos nos preocupar com como o Service locator encontra estas instâncias. Ele simplesmente as encontra. Há a inversão de controle parcial aqui. Nossa classe cliente delega a tarefa de instanciação para outro componente, mas ainda participa ativamente na injeção. No entanto, ainda temos alguns problemas: nossa classe Integrador está acoplada a uma nova entidade: o Service Locator que apesar de nos fornecer as instâncias corretas de nossas dependências, a injeção **ainda é** feita pela classe cliente. Temos portanto uma inversão de controle bem meia boca.

O ideal seria que nossa classe Integrador não possuísse nenhuma dependência além de Origem e Destino. Sai o Service Locator, entra o container de injeção de dependências.

Todo container de injeção de dependências apresenta basicamente o mesmo comportamento: a partir de configurações que podem estar na forma de documentos XML, anotações Java ou mesmo código executável, este se encarrega de instanciar corretamente as classes que definimos nestas configurações e a partir destas, fazer todo o controle de ciclo de vida e injeção de dependências.

Simplificando ao máximo, o Spring é um competente container de injeção de dependências em cima do qual foram construídos diversos módulos com o objetivo de facilitar o desenvolvimento de aplicações corporativas. No transcorrer deste livro vamos aprender com detalhes o seu funcionamento.

E no meio deste, quando achar que a injeção de dependências resolveu todos os seus problemas, irei lhe mostrar que isto é apenas parcialmente verdade. A tal da

programação orientada a aspectos resolve boa parte dos problemas restantes decorrentes do alto acoplamento. Espero que goste.

CAPÍTULO 2

Conheça o Spring Framework

Aonde os objetos se encontram.

Agora que sabemos quais os nossos problemas essenciais (acredite, os mais difíceis) chegou o hora de conhecermos o ferramental que irá minimizá-los e neste processo, ser apresentados a uma série de dificuldades específicas da plataforma Java (SE e EE) que o Spring Framework ajuda a resolver.

Vamos falar mais agora sobre o Spring. Este é um capítulo fundamental para quem nunca trabalhou com o framework. Veremos uma série de conceitos importantes para o bom uso da ferramenta como por exemplo beans, containers, ciclo de vida e muitos outros. Caso já possua o conhecimento básico a respeito do Spring e dos problemas que tratarei aqui, sugiro que pule para o próximo capítulo.

2.1 POR QUE CRIARAM O SPRING?

Os germes por trás do Spring aparecem pela primeira vez em 2002 com a publicação do livro *Expert One-To-One J2EE Design and Development*[7] de Rod Johnson. É

um marco na história do desenvolvimento de aplicações corporativas baseadas na plataforma Java EE por apresentar uma crítica bastante convincente ao padrão de desenvolvimento empurrado pela Sun Microsystems para implementação de lógica de negócios: os EJBs (*Enterprise JavaBeans*).

Tal como apresentado por Rod Johnson, EJB é uma excelente opção para desenvolvimento de aplicações distribuídas, o que não necessariamente implica na melhor solução a ser adotada para todos os casos, principalmente quando objetos distribuídos não são uma necessidade. Este é o argumento básico por trás do livro.

Em 2003 é iniciado o desenvolvimento do Spring Framework, porém ele só foi apresentado oficialmente como versão final ao público em 2004 com o livro *Expert One-To-One J2EE Development Without EJB* [8]. Para melhor entender as razões que motivaram a criação do Spring é interessante voltarmos para 2004 quando este foi introduzido.

2.2 O MUNDO DOS OBJETOS EM 2004

A plataforma Java EE era bastante diferente da que encontramos em 2012. Para começar, esta era conhecida como J2EE (Java 2 Enterprise Edition) e o modelo de desenvolvimento para objetos de negócio incentivado pela Sun era o EJB, cuja versão atual era a 2.1, lançada em novembro do ano anterior.

O segundo livro de Rod Johnson é lançado em junho de 2004. Dado a novidade da especificação 2.1 do EJB, a versão dominante da plataforma ainda era a 2.0, publicada em agosto de 2001.

A versão 5 da linguagem Java só seria lançada oficialmente em setembro de 2004. Não podíamos contar com recursos tidos como básicos hoje como por exemplo anotações, generics, autoboxing e muitos outros. Curiosamente, estas funcionalidades já existiam na linguagem C# da plataforma .net, a qual Rod Johnson conhecia a algum tempo e que menciona como ferramentas poderosas para desenvolvimento em seu segundo livro [8].

O EJB se encontrava (e ainda se encontra) no centro da especificação Java EE. A ideia é realmente linda: desenvolver componentes que encapsulem as regras de negócio em uma plataforma que facilite a implementação de aplicações distribuídas de alto desempenho e escalabilidade.

Bastaria ao desenvolvedor implementar o que realmente interessava naquele EJB e este seria gerenciado pelo servidor de aplicações que ofereceria toda a infra estrutura necessária para a execução deste componente, como por exemplo controle

transacional declarativo, uma plataforma de persistência de dados, por exemplo, o JDO, a primeira versão dos *Entity Beans*), pooling de threads, sistema de mensageria e muitos outros recursos poderosos.

Outra promessa do EJB é que surgiria um mercado só de componentes terceirizados visto que em teoria, um EJB que fosse implementado seguindo as especificações da Sun poderia ser executado sem problemas em qualquer servidor de aplicações. Afinal de contas, é um padrão, certo? Um mundo teórico perfeito: o desenvolvedor só precisava se preocupar em implementar a sua lógica de negócio e se conseguisse criar algo que fosse reaproveitável em outros sistemas, poderia inclusive ficar rico vendendo seus EJBs! Nada poderia dar errado!

Só havia um pequeno problema: **o tempo gasto com requisitos não funcionais muitas vezes era maior que com os funcionais**. Como não haviam ainda anotações, o desenvolvedor ao implementar seus EJBs precisava satisfazer uma série de interfaces e ainda gastava um bom tempo escrevendo arquivos de configuração no formato XML, como o famigerado `ejb-jar.xml`. O mais interessante é que das interfaces a serem implementadas, na maior parte das vezes pouquíssimos métodos eram de fato úteis: a maior parte se encontrava ali apenas ocupando espaço.

Para piorar ainda mais a situação, a portabilidade entre servidores era na prática um mito e o tal mercado maravilhoso que surgiria por causa daquela plataforma nunca se concretizou. O surgimento de soluções como o XDoclet, EJB-Gen e diversas ferramentas desenvolvidas especificamente para um ou outro servidor de aplicações (e que na prática eram apenas geradores de código) eram um sintoma claro de que havia algo muito errado com a plataforma J2EE.

Sim, o EJB era maravilhoso para a construção de aplicações corporativas *de grande porte*, que necessitavam realmente de objetos distribuídos, clusterização massiva e toda a parafernália que só é de fato útil para a menor parte dos projetos de TI. Neste contexto a maior parte dos desenvolvedores se via de mãos atadas, tendo de sacrificar sua produtividade em prol de uma série de tecnologias que, na prática, raramente usava.

Pior: a maioria dos desenvolvedores sequer sabia pra que serviam os EJBs. O mais interessante é que no horizonte víamos a emergência da plataforma .net da Microsoft, que se mostrava muito mais pragmática apesar de não apresentar a alta portabilidade do Java.

O próprio Rod Johnson em seu segundo livro [8] acreditava que os EJBs seriam em um futuro próximo considerados tecnologia legada. Como bem sabemos, não foi isto o que aconteceu.

2.3 OS PROBLEMAS DO EJB EM 2004

Um dos principais problemas com os EJBs nesta época é o fato de não poderem ser executados *fora* de um servidor de aplicações. Com a emergência do desenvolvimento orientado a testes e a confirmação de que esta prática ampliava tanto a produtividade quanto a qualidade dos sistemas, a implementação de testes unitários e integrados em EJBs sempre requeria malabarismos imensos por parte dos desenvolvedores.

Pior que isto: os servidores de aplicação sempre foram pesados (o que só começou a ser revertido recentemente em 2011, tendo como marco o lançamento do JBoss AS 7), o que tornava o desenvolvimento e depuração destes componentes uma tarefa bastante árdua mesmo para as estações de trabalho mais poderosas de 2004. Na prática, como a maior parte dos desenvolvedores não precisava de aplicações distribuídas, pouquíssimos eram os que realmente entendiam o propósito de um EJB real.

O leitor pode se perguntar neste momento a razão de tantos problemas com o EJB nesta época. Basicamente foi o modo como a especificação foi criada. Ao invés de ser o resultado de uma evolução baseada na observação dos usos e desusos em produção o EJB foi ditado por um comitê.

Comitês são ótimos quando o objetivo é garantir a longevidade de um padrão, visto serem normalmente compostos por um grande número de participantes. O problema é que o produto desenvolvido normalmente é criado antes que qualquer um o ponha em prática no mercado, ao contrário do que ocorre em projetos desenvolvidos por um grupo menor de desenvolvedores ou comercial. Pior: como há uma gama imensa de interesses envolvidos, normalmente este processo é lento, impedindo que adaptações sejam feitas em tempo hábil para que uma tecnologia se mantenha competitiva.

O que todos desejavam era claro: as vantagens oferecidas pela plataforma Java EE porém com a produtividade e clareza que o mercado exigia naquele momento. Uma plataforma para o desenvolvimento de aplicações corporativas que fosse pragmática.

A plataforma J2EE tinha como objetivos resolver diversos requisitos que na prática eram apenas potenciais. Alguns exemplos são a necessidade de atender à maior gama possível de tipos de clientes existentes: então o desenvolvedor implementava os EJBs para serem externalizados via RMI, IIOP, etc. No entanto, no dia a dia, o mais comum era haver apenas um tipo de cliente dentro da corporação, muitas vezes baseado em padrões bem mais interoperantes como o SOAP.

Outro exemplo era o medo reinante na época de que, do dia pra noite, se fizesse necessária a troca do sistema gerenciador de banco de dados. Convenhamos: este é o tipo de alteração que raríssimas vezes um desenvolvedor se vê obrigado a tratar dentro de uma corporação ou mesmo dentro de uma software house responsável pelo desenvolvimento de algum produto específico. Dada esta constante necessidade em atender todas as situações possíveis, o desenvolvedor se via na maior parte das vezes guiando-se não pelos requisitos que deveriam atender seu cliente, mas sim aos impostos pela tecnologia. Era nítido que havia algo de muito errado no reino da Javolândia.

2.4 ALTERNATIVAS COMEÇAM A SURGIR: IoC/DI E AOP

Nesta mesma época começam a se popularizar entre os desenvolvedores Java os conceitos de inversão de controle (IoC - *Inversion of Control*)/Injeção de Dependências (DI - *Dependency Injection*) e Programação Orientada a Aspectos (AOP - *Aspect Oriented Programming*).

As ideias por trás da IoC/DI foram tratadas no capítulo 1. Como já sabemos, estas técnicas tem como objetivo atacar o problema do alto acoplamento entre os componentes do nosso sistema. A AOP tem o mesmo objetivo, só que vai um passo além, aonde a orientação a objetos não consegue entrar sem criar monstros.

Com a injeção de dependências conseguimos identificar os módulos mais facilmente visíveis de nosso sistema. Seguindo o paradigma de orientação a objetos, separamos os comportamentos de nossa aplicação em classes que são referenciadas em todos os pontos de nosso sistema que precisem usar a funcionalidade encapsulada. Este tipo de comportamento é perfeitamente aceitável quando as dependências entre os objetos está diretamente relacionada à lógica de negócio implementada. Por exemplo, o nosso integrador exposto na introdução, que precisava de um DAO para receber os dados a serem processados e outro para enviar o resultado do processamento.

No entanto há alguns comportamentos que aparecem por toda a aplicação que não estão necessariamente relacionados à lógica de negócio que estamos implementando. Observamos que este tipo de comportamento costuma gerar uma infinidade de código repetitivo espalhado por todo o sistema. Chamamos este comportamento secundário de *aspecto*.

Um bom exemplo de aspecto seria um controle de segurança. Vamos imaginar que seja de nosso interesse permitir a execução de alguns métodos em nosso sistema

apenas se o usuário corrente tiver permissão para tal. Poderíamos implementar a funcionalidade em uma classe que encapsule a lógica de acesso e em seguida injetá-la em diversas outras classes do nosso sistema, tal como no exemplo abaixo:

```
public class ClasseMuitoImportante {

    private ControleAcesso controleAcesso;
    public ControleAcesso getControleAcesso() {
        return controleAcesso;
    }
    public void setControleAcesso(ControleAcesso ca) {
        controleAcesso = ca;
    }

    public void metodoImportante() {
        if (getControleAcesso().possuiPermissaoParaTal()) {
            // executa código incrivelmente importante
        }
    }
}

public class OutraClasseMuitoImportante {
    private ControleAcesso controleAcesso;
    public ControleAcesso getControleAcesso() {
        return controleAcesso;
    }
    public void setControleAcesso(ControleAcesso ca) {
        controleAcesso = ca;
    }

    public void apagarTudo() {
        if (getControleAcesso().possuiPermissao()) {
            // apaga TUDO
        }
    }
}
```

Repare como nas duas classes vemos o mesmo código sendo repetido:

```
private ControleAcesso controleAcesso;
public ControleAcesso getControleAcesso() {
    return controleAcesso;
}
```

```
}  
public void setControleAcesso(ControleAcesso ca) {  
    controleAcesso = ca;  
}  
public void algumMetodo() {  
    if (getControleAcesso().possuiPermissao()) {  
        // faça algo  
    }  
}
```

Este disco arranhado espalhado por todo nosso código é o que chamamos *aspecto*, ou seja, uma característica que é comum a vários pontos.

Enquanto a injeção de dependências modulariza nosso sistema usando o que a Orientação a Objetos tem a nos oferecer, a AOP diminui ainda mais o acoplamento isolando os *aspectos* que não são facilmente identificáveis e isolados pela orientação a objetos.

Identificado e isolados os aspectos, tudo o que a AOP precisa fazer é interceptar em tempo de execução as chamadas aos métodos onde detectamos a necessidade de aplicação do comportamento identificado e, com base nisso, alterar o comportamento de nossas classes sem que estas sequer saibam disto.

Não se preocupe neste momento se não conseguiu compreender exatamente como a AOP funciona: a explicação que dei é a mais superficial possível, apenas para trazer o assunto a tona. Teremos um capítulo inteiro só para tratar este assunto.

O importante agora é saber que aplicando IoC + DI + AOP, o Spring consegue evitar que precisemos de um servidor de aplicações Java completo. Usando um servidor de Servlets “simples”, como Jetty ou Tomcat, conseguiremos praticamente os mesmos resultados que só poderiam ser obtidos anteriormente com um servidor de aplicações como JBoss, WebLogic ou WebSphere.

RESUMINDO INVERSÃO DE CONTROLE, INJEÇÃO DE DEPENDÊNCIAS E PROGRAMAÇÃO ORIENTADA A ASPECTOS

A grosso modo, você pode pensar o seguinte: a IoC controla o ciclo de vida de nossas aplicações, a DI define quais classes iremos instanciar e em quais lugares iremos injetá-las. Já a AOP vai além na modularização que a DI nos proporciona, adicionando novos comportamentos às nossas classes.

2.5 CONTAINER: O SUJEITO QUE TORNA TUDO ISTO POSSÍVEL

O servidor de aplicações Java também é chamado de *Container EJB*. No final das contas o Spring irá apresentar um container alternativo no qual possa ser gerenciado o ciclo de vida de nossos objetos de negócio. Chegou portanto o momento de apresentar uma série de conceitos fundamentais para a compreensão do restante deste livro.

O átomo de toda a nossa história é o *objeto de negócio*. Este é a instância de uma classe aonde implementamos nossos requisitos funcionais. O EJB é um objeto de negócios e, no caso do Spring, nós o chamamos de *bean*.

Um *bean* é um objeto que possui seu *ciclo de vida* gerenciado pelo *container* de IoC/DI do Spring. Uma boa maneira de entender como a inversão de controle se aplica é descrever o que vêm a ser o tal do *ciclo de vida*.

2.6 DO INÍCIO AO FIM DE UM OBJETO, ENTENDA O CICLO DE VIDA

O *Ciclo de vida* diz respeito às etapas de execução de um objeto. As fases pelas quais um objeto de negócio passa durante a execução de um programa, são ilustrados na figura 5.1.

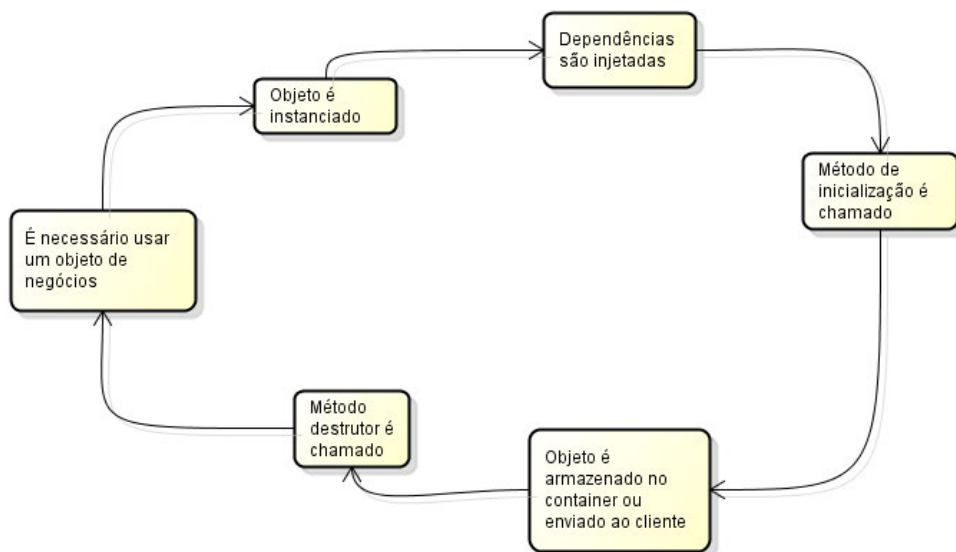


Figura 2.1: Ciclo de vida de um objeto

- 1) O objeto é instanciado;
- 2) Se houverem dependências a serem injetadas no objeto, estas devem ser injetadas;
- 3) Se após as dependências tiverem sido injetadas for necessário executar um método de inicialização, este deverá ser invocado;
- 4) O objeto já inicializado e com todas as suas dependências injetadas é enviado ao cliente que dele necessita;
- 5) Uma vez não mais necessário, existindo um método destrutor, este deverá ser invocado e o objeto descartado do *container*.

Não é difícil imaginar a execução de todo este procedimento programaticamente. Com relação à última ação, bastaríamos escrever um método destrutor, algo como um `finalize`, em nossa classe e rezar para que este seja executado no momento certo pela JVM. Mas isso não é lá muito elegante, certo?

Voltando ao *bean*, este é mais que um mero *JavaBean*. Deve ser visto como um *componente*. O que diferencia um componente dos demais objetos, como um *JavaBean*, por exemplo, são as seguintes características:

- **Possui baixa granularidade**, ou seja, seus clientes não precisam saber quais as suas dependências internas;
- O que realmente importa para seus clientes é **a interface disponibilizada pelo objeto**, que define quais os serviços oferecidos pelo mesmo (é o seu contrato);
- São **facilmente substituíveis por outras implementações** que mantenham o mesmo contrato (plugabilidade).

O QUE É UM JAVA BEAN?

Foi um padrão adotado pela Sun para a escrita de componentes reutilizáveis. Todo JavaBean deve satisfazer três condições:

- 1) Deve possuir um construtor público que não receba parâmetros
- 2) Todos os seus atributos visíveis devem ser declarados como privados (ou protegidos) e acessados apenas por métodos get e set.
- 3) Deve implementar a interface `java.io.Serializable`

Exemplo de JavaBean:

```
class Pessoa implements java.io.Serializable {  
    // construtor publico sem argumentos  
    public Pessoa() {}  
  
    private String nome;  
  
    // getter  
    public String getNome() {return this.nome;}  
  
    // setter  
    public void setNome(String valor) {this.nome = valor;}  
}
```

Podemos dizer portanto que o *container* é o elemento arquitetural de nossa aplicação, responsável por gerenciar o ciclo de vida dos *componentes* do nosso sistema. Todo *container* (EJB ou não) funciona basicamente da mesma maneira: são alimentados por alguma forma de configuração e com base nelas, criam internamente as regras de gerenciamento dos componentes ali descritos.

Um *container* pode ser *intrusivo* ou não. É intrusivo quando requer que seus objetos gerenciados dependam de si de alguma maneira, como por exemplo, obrigando-os a implementar alguma interface ou possuírem alguma anotação específica.

O *container* EJB é um exemplo de intrusivo. Nas versões anteriores à 3.0 era necessário que todo EJB implementasse uma série de interfaces para que pudessem ser reconhecidos (isto além do arquivo de configuração). Mesmo após a versão 3.0, ainda podemos dizer que há um certo nível de intrusão pelo fato de serem necessárias anotações Java para indicar pontos de injeção de dependências e métodos de *callback* a serem chamados durante o ciclo de vida do objeto.

No caso do Spring temos um *container* com intrusividade mínima pois, ao contrário do EJB, trabalhamos com POJOs (Plain Old Java Object). Um POJO é uma classe Java normal que não precisa implementar nenhuma interface ou possuir determinada anotação para que possa ser gerenciada por um *framework*. De certo modo é a classe que implementa regras de negócio de uma forma ideal, ou seja, que se preocupa única e exclusivamente com a execução de nossos requisitos funcionais.

Com intrusividade mínima, temos como ganho um maior reaproveitamento de código, visto este não mais depender de uma plataforma específica para que seja executado. Outra vantagem importante é facilitar a escrita de testes, dado a redução das dependências externas ao objeto que até então precisaríamos simular, para conseguir realizar testes.

Outra diferença importante entre *containers* diz respeito ao seu peso: podem ser leves (“lightweight”, como o Spring) ou pesados (“heavy weight”, como o container EJB). De acordo com Rod Johnson em seu livro *Expert One-To-One J2EE Development Without EJB*[8], todo *container* leve possui as seguintes características:

- Não são intrusivos ou são minimamente intrusivos;
- Possuem inicialização rápida e consomem poucos recursos computacionais;
- São independentes do ambiente de execução. O Spring por exemplo pode ser executado tanto dentro quanto fora de um servidor de aplicações ou até mesmo em applets;

- Definem requisitos mínimos (praticamente inexistentes) para que um componente possa ser gerenciado.

Independente de ser intrusivo ou não, leve ou pesado, todo *container* deve satisfazer três requisitos: ser configurável, controlar o ciclo de vida dos seus objetos e prover métodos de obtenção, também conhecidos como *lookup*, de componentes de negócio.

Dado que a principal falha do EJB anterior ao 3.0 foi o excessivo tempo que gastávamos com requisitos não funcionais, devemos fazer uma pergunta final sobre *containers* : **realmente precisamos de um já que nosso objetivo é simplificar a arquitetura?**

A resposta é sim. Um *container* nos oferece algumas vantagens que justificam seu uso. A primeira delas diz respeito à plugabilidade. Dado que as implementações de nossos componentes são definidas pela configuração, fica muito fácil substituir uma implementação por outra sem quebrar o sistema.

O QUE É UM LOOKUP?

Usamos esta expressão para designar a obtenção de objetos a partir de um container. Todo container deve prover alguma interface que nos forneça métodos que possibilitem obter objetos a partir do seu nome ou tipo.

Veremos no capítulo 3 como fazer um lookup no Spring. Mas para matar sua curiosidade, a especificação JNDI da plataforma Java EE, muito usado pelos EJBs na realidade é uma interface de lookup. No código abaixo, quando executamos o método `lookup` no objeto `Context`, estamos executando a obtenção de um objeto já preparado pelo container para execução.

```
Context ctx = InitialContext();

// Lookup em ação
Object objeto = ctx.lookup("meuObjeto");
```

Uniformidade de configuração é outra vantagem importante: a partir do momento em que padronizamos como é o ciclo de vida e dependências de nossos com-

ponentes em um único ponto, aumenta-se significativamente a *manutenibilidade* de nossos projetos.

Costumo mencionar também como vantagem o argumento da autoridade. Em grande parte dos casos, quem implementou o gerenciamento de ciclo de vida, injeção de dependências e demais serviços oferecidos pelo *framework* possui uma experiência no meio muito maior que a minha ou sua.

Quando não estamos trabalhando com uma versão de testes, normalmente podemos ter certeza de que estamos lidando com um software devidamente testado não só pelos desenvolvedores que o implementaram como que também pela comunidade que o usa. Quanto maior a comunidade, maior, em tese, é a quantidade de testes em ambiente de produção pelos quais passou aquele código.

Finalmente, outra vantagem interessante oferecida pelos *containers* é o fato de nos propiciarem um ponto único de acesso aos serviços que compõem nossa aplicação, garantindo assim uma maior padronização do projeto e consequentemente, maior facilidade de adaptação de novos membros da equipe a este.

2.7 SPRING EM PARTES

Não seria exagero dizer que o Spring é a prova de que os conceitos de inversão de controle e injeção de dependências realmente funcionam, dado que praticamente todas as funcionalidades do framework são baseadas em seu *container*. Observando os módulos básicos do framework é difícil não se maravilhar com o poder destas técnicas.

É importante termos esta visão panorâmica dos módulos que compõem o framework para que fique claro o quão abrangente ele é: basicamente o Spring abrange todas as necessidades de uma aplicação corporativa.

Em sua versão básica, o Spring é formado por seis componentes: o *container* de inversão de controle, suporte a AOP, instrumentação, acesso a dados/integração, suíte de testes e web, como pode ser visto na figura 2.2.

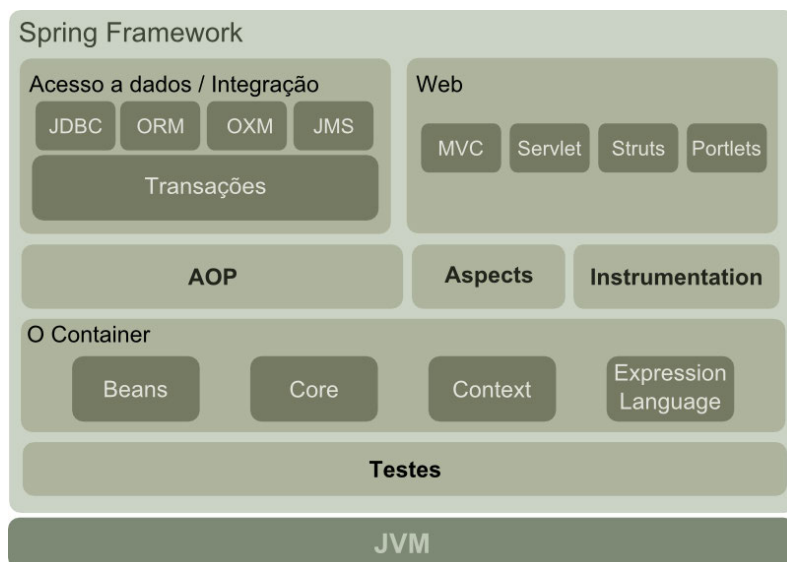


Figura 2.2: Os componentes básicos do Spring

É interessante observar que como requisito básico de funcionamento não temos um servidor de aplicações ou Servlets, mas apenas a JVM. Este foi um dos principais diferenciais do Spring em seu lançamento: trazer para o desenvolvedor recursos de computação corporativa que até então só eram oferecidos por servidores de aplicação pesados.

Como exemplo destes recursos podemos citar controle transacional declarativo e gerenciamento de ciclo de vida de componentes.

Um ponto importante que deve ser mencionado é o fato do Spring, ao contrário de um container EJB, não ser uma solução *tudo ou nada*. Você usa apenas os componentes que interessam ao seu projeto, podendo ignorar tranquilamente aquilo que não precisa. Consequentemente, temos como resultado uma arquitetura bem mais enxuta e fácil de trabalhar.

Mas voltemos aos nossos módulos.

2.8 O CONTAINER

Se quiser trabalhar com o Spring, este provavelmente é o único módulo obrigatório. Os módulos *core* e *beans* são o núcleo do framework aonde são implementados o suporte à inversão de controle e injeção de dependências.

Já no módulo `context` encontra-se a implementação do *ApplicationContext*. No Spring há dois tipos de *containers*: *BeanFactory* e *ApplicationContext*.

O *BeanFactory* existe desde a primeira versão do *framework*. É basicamente uma implementação sofisticada do padrão *Factory*, baseada em configuração. Fornece apenas o suporte básico a IoC e DI e encontra-se implementado no módulo *core*. A segunda versão é baseada no *BeanFactory* e oferece recursos corporativos mais avançados, como por exemplo gerenciamento de recursos, internacionalização.

Na prática, raríssimas vezes o *BeanFactory* é usado, e seu uso é inclusive desencorajado pelo próprio pessoal da SpringSource [14], que é a equipe responsável pelo desenvolvimento do framework. Na prática seu uso atualmente só faria sentido em ambientes computacionais extremamente restritos, como por exemplo applets ou dispositivos móveis.

Finalmente, temos também o módulo *Expression Language* (SpEL). Como veremos no próximo capítulo, a forma mais popular de configuração do Spring é através de documentos XML. A SpEL nos fornece uma linguagem muito parecida com a EL que estamos acostumados a usar com JSPs, porém voltada para a configuração do *container*. Ela torna nossos arquivos de configuração “vivos”, na medida em que a partir do seu uso podemos definir valores de configuração em tempo de execução e não em tempo de configuração.

2.9 TRABALHANDO COM AOP E ASPECTS

O Spring vêm acompanhado do seu próprio framework que implementa a programação orientada a aspectos. Este pode vir em dois “sabores”. Tradicional, que é a primeira implementação de AOP do Spring ou *Aspects*, que é o uso do *AspectJ* como motor de AOP do Spring, uma alternativa bem mais poderosa que a original.

Não se preocupe, como disse anteriormente, teremos um capítulo só para entendermos como a AOP realmente funciona no Spring.

2.10 INSTRUMENTAÇÃO DE CÓDIGO

O projeto não acaba quando fazemos o deploy. É importante lembrar que a maior parte do custo de um software é a sua manutenção e a equipe do Spring tinha plena consciência disto. O módulo de instrumentação facilita a vida do pessoal de suporte oferecendo facilidades na implementação de JMX. Esta tecnologia nos permite acompanhar em tempo de execução tudo o que acontece com nossos sistemas, gerando diversas estatísticas interessantes.

2.11 ACESSO A DADOS E INTEGRAÇÃO

Toda aplicação corporativa que se preze, precisa acessar e escrever dados em alguma fonte de dados. De cara, o Spring nos oferece suporte às principais tecnologias de persistência adotadas por desenvolvedores Java. Há suporte para JDBC, ORMs como Hibernate, iBatis, JPA, JDO e OXM. O suporte a estas tecnologias se dá através de *templates*, que reduzem significativamente a quantidade de código *boilerplate*, ou seja, de infra estrutura, que precisamos escrever nestas situações, como por exemplo abrir e fechar conexões, iniciar e finalizar transações etc.

Além do suporte a estas tecnologias, este módulo nos oferece também uma nova hierarquia de exceções que é bem mais interessante do que as oferecidas originalmente pelas tecnologias que são abstraídas.

Um dos pontos de venda do *framework* encontra-se neste módulo: o controle transacional. Spring nos oferece uma implementação de transações declarativas e programáticas que vai além do que é oferecido hoje pelos servidores de aplicação. Mais que isto: torna a necessidade destes coisa do passado.

Finalmente e não menos importante, há também suporte nativo a JMS. Tal como ocorre com as diferentes fontes de dados abrangidas pelo *framework*, temos aqui a aplicação de *templates* que tornam a vida do desenvolvedor muito mais tranquila no momento de lidar com suas filas JMS.

2.12 APLICAÇÕES NA WEB COM O SPRING

Quando você pensa que o Spring já faz coisas demais, não é raro os que estão começando com essa tecnologia, se surpreenderem ao descobrir que, “de brinde”, há um *framework* que implementa o MVC de uma forma extremamente produtiva para você. Trata-se do Spring MVC.

O interessante deste módulo é que não foi planejado, mas surgiu conforme a equipe de desenvolvimento do *framework* foi aceitando o fato de que o Struts, *framework* que dominava o mercado, apresentava uma série de limitações que dificultavam bastante a separação de interesses entre as camadas de controle, negócio e visualização.

Uma outra curiosidade sobre o Spring MVC é o fato dele ser a base de outro projeto muito importante para a SpringSource: o *framework* Grails, que é baseado em Groovy e apresenta o mesmo nível de produtividade oferecido pelo Ruby on Rails.

Além deste *framework* MVC, há também suporte básico para que o *container* do Spring seja aplicado em uma aplicação Java web convencional, através de pequenas

alterações no arquivo `web.xml` e suporte a alguns *frameworks* web de mercado como JSF, Struts 1.x e 2.x, WebWork e Tapestry.

2.13 E AINDA MAIS COMPONENTES!

O *container* do Spring é tão versátil que, pouco tempo após o seu lançamento, a Spring Source, entidade responsável pelo desenvolvimento do *framework*, desenvolveu para ele uma série de componentes opcionais.

Dentre os mais famosos podemos citar o Spring Security, que cuida da segurança da aplicação de uma maneira extremamente robusta e pode ser aplicado em qualquer projeto Spring, além de ser uma maravilhosa aplicação do conceito de AOP.

Outro componente interessante é o Spring Data, usado para lidarmos com bases de dados relacionais ou não, tal como é feito no módulo de acesso a dados/integração do Spring, abstrai para o desenvolvedor aqueles detalhes tediosos e repetitivos de implementação, através de *templates* muito fáceis de serem usados.

Se quiser integrar sua aplicação com redes sociais é possível usar o módulo Spring Social, que torna muito fácil a comunicação com serviços consagrados como Twitter, Facebook, LinkedIn e outros.

Para aqueles que trabalham com processamento em lote mais pesado há o Spring Batch: uma biblioteca extremamente leve, que torna a vida de quem precisa trabalhar com Spring e processamento pesado em lotes uma tarefa bem mais agradável.

E estes por mais incrível que pareça não são todos os módulos adicionais. Veremos esses e vários outros no decorrer desse livro!

2.14 RESUMINDO

Neste capítulo pudemos conhecer os conceitos básicos que nos guiarão no transcorrer deste livro a partir dos problemas que justificaram a criação do Spring. Feito isto, tivemos também uma visão panorâmica dos módulos básicos que compõem o *framework* e inclusive conhecemos também alguns módulos opcionais que poderão tornar nossa vida mais fácil daqui para frente.

Não se assuste se estes conceitos ainda não estiverem fazendo sentido para você. A partir do próximo capítulo, quando começamos a por a mão na massa mesmo com o Spring, veremos como tudo se encaixa em um quebra cabeças que, na prática, possui apenas um tipo e peça: o *container* de inversão de controle e injeção de dependências.

CAPÍTULO 3

Conhecendo o Container

Chegou o momento de lidarmos diretamente com o Spring Framework. O ponto de partida será o container de inversão de controle/injeção de dependências, chamado na documentação oficial como “Core Container” [14]. Serão vistos quatro de seus componentes fundamentais: Core, Beans, Expression Language (SpEL) e Context.

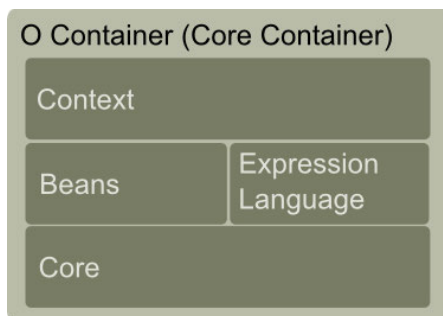


Figura 3.1: O que compõe o Core Container

Na base do *Core Container* se encontra o módulo *Core*. Muitos acreditam erroneamente ser este o local aonde se encontra implementado o container do Spring, o que é *quase* verdade. Na realidade aí se encontram as classes básicas que permitirão a sua implementação no módulo *Beans*. Em *Core* são encontradas facilitadores no gerenciamento de classloaders, reflexão, tratamento de recursos, strings, etc. Resumindo, é neste módulo que se encontra toda a infraestrutura necessária que permite a implementação do container. Na prática, entretanto, sempre trabalhamos com estes dois módulos juntos como se fossem um só.

Usando como base o *Core* temos o módulo *Beans*, onde encontramos a primeira versão do container que é o *BeanFactory*. Este é uma implementação sofisticada do padrão de projetos *factory* que torna desnecessária a codificação manual de *singletons* e nos permite o desacoplamento total do gerenciamento de dependências e ciclo de vida de nossos beans de nosso código fonte através do mecanismo de configuração implementado neste módulo.

Como veremos o mecanismo mais popular de configuração do Spring é o XML. Com o objetivo de adicionar maior dinamismo à configuração do framework a equipe de desenvolvimento do Spring criou a Spring Expression Language (mais conhecida como SpEL), que é uma linguagem baseada em expressões que lembra bastante à *Unified Expression Language* (EL) presente no JSP. Esta linguagem nos permite definir valores e comportamentos aos nossos beans definidos usando XML ou anotações em tempo de execução, o que é impossível (ou ao menos bastante difícil) sem este recurso.

Finalmente, temos o módulo *Context*, aonde se encontra a implementação mais avançada e comumente usada do container que é o *ApplicationContext*. Esta, baseada na base fornecida pelos módulos *Core* e *Beans* oferece recursos poderosos como por exemplo internacionalização, propagação de eventos, AOP, gerenciamento de recursos e acesso a funcionalidades básicas da plataforma Java EE.

3.1 PREPARANDO SEU AMBIENTE DE DESENVOLVIMENTO

Para iniciar nosso trabalho é importante que seja descrito como deve ser feita a configuração do seu projeto. Há basicamente duas maneiras de configurar seu projeto: usando ou não um gerenciador de dependências baseado no Maven.

Configurando manualmente

Para adicionar manualmente as dependências do Spring baixe o framework em

<http://www.springsource.org/download/community>. Sempre há duas versões disponíveis para download: com ou sem a documentação. Dada a alta qualidade da documentação do projeto e o fato de ainda hoje em 2012 nem sempre termos a Internet disponível, é recomendável escolher a versão que contenha a documentação básica do framework.

Feito isto, basta adicionar ao seu classpath (ou usando sua IDE de sua preferência) os seguintes arquivos jar: `spring-core-3.1.1.RELEASE.jar`, `spring-context-3.1.1.RELEASE.jar`, `spring-beans-3.1.1.RELEASE.jar`, `spring-expression-3.1.1.RELEASE.jar`. Também é necessário adicionar a única dependência externa que o Spring usará neste capítulo que é o Commons Logging da Fundação Apache que deve estar na versão 1.1 ou posterior.

Configurando com Maven

Sua vida será bem mais fácil caso esteja trabalhando com algum gerenciador de dependências baseado em Maven. Como o módulo *Context* é baseado nos outros três módulos que trataremos neste capítulo, apenas esta dependência deve ser incluída em seu projeto e consequentemente todas as demais virão por consequência. O código abaixo expõe um exemplo desta configuração usando Maven.

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>3.1.1.RELEASE</version>
</dependency>
```

Escolha da IDE

Atualmente as IDEs mais populares como por exemplo Netbeans, Eclipse e JetBrains IDEA oferecem suporte ao Spring através de plugins dada a popularidade do framework. No entanto o melhor suporte atualmente é oferecido pelo SpringSource Tool Suite.

O SpringSource Tool Suite (STS) é a distribuição customizada do Eclipse desenvolvida pela equipe da SpringSource e que pode ser baixada em <http://www.springsource.org/downloads/sts>. É o ambiente ideal para o desenvolvedor habituado a trabalhar com o Eclipse pois já vêm com todos os plugins necessários para que o desenvolvedor possa começar a trabalhar imediatamente sem que precise se preocupar com a instalação de complementos em sua IDE. Dentre os plugins destaca-se

o suporte aos demais projetos da Spring Source como por exemplo Spring Batch, Integration e MVC. O excelente suporte a outros projetos da empresa como Grails e Roo também justificam o seu uso.

Para os iniciantes há alguns recursos que auxiliam bastante o aprendizado, como por exemplo o auto completar quando estamos a editar arquivos de configuração e a possibilidade de visualizar graficamente os relacionamentos entre nossos beans tal como pode ser visto na imagem abaixo.

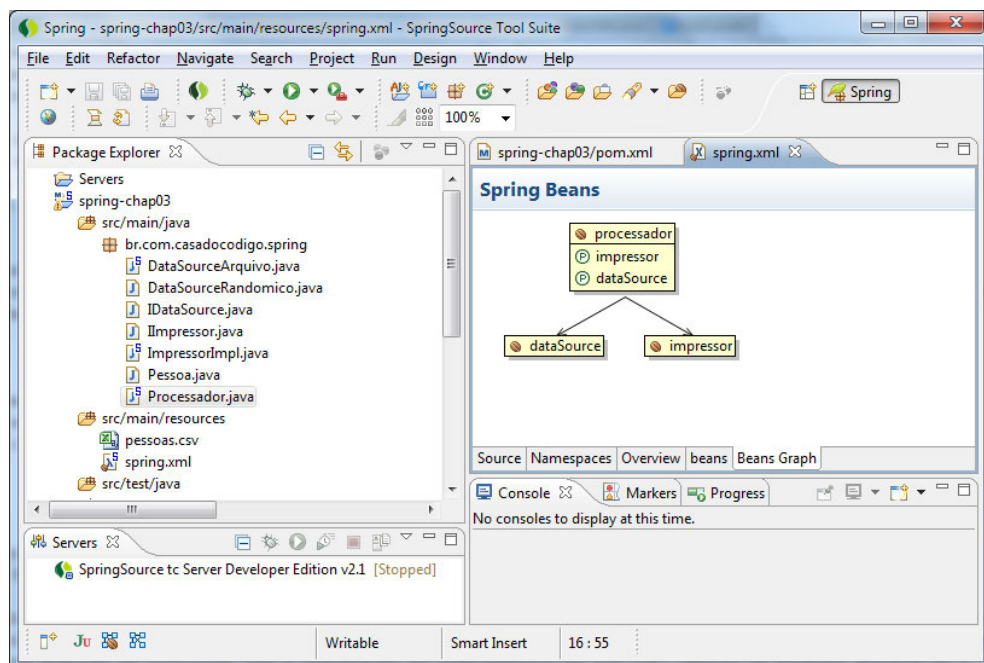


Figura 3.2: SpringSource Tool Suite

A instalação do STS é bastante simples: basta baixar o instalador adequado ao seu sistema operacional em <http://www.springsource.org/downloads/sts> e executá-lo. Caso já possua o Eclipse instalado, é também possível instalar manualmente os plugins presentes no STS. Para isto, basta seguir as instruções presentes no manual de instalação da IDE [9].

É importante lembrar que a escolha da IDE é completamente opcional dado que praticamente todas as mais adotadas pelo mercado oferecem suporte ao framework. Sendo assim, caso não se sinta à vontade com o STS é importante que saiba que a

escolha da IDE em nada influenciará seu aprendizado do Spring usando este livro.

3.2 NOSSO SISTEMA DE EXEMPLO

A melhor maneira de aprendermos como funciona um framework é usando-o. Nosso ponto de partida é um sistema bastante simples tal como fizemos no primeiro capítulo. Este exemplo é ideal para começarmos pois é simples o suficiente para que possamos ver o básico mas ao mesmo tempo nos oferece espaço para que possamos transformá-lo em um monstro complexo e com isto evidenciar o poder que o Spring nos fornece.

Trata-se de um sistema de BI inicialmente composto por três elementos: uma fonte de dados, um processador responsável pela execução dos cálculos e, finalmente, um impressor que irá formatar os resultados de tal forma que se tornem legíveis aos usuários finais do sistema.

Inicialmente nosso *processador* será a parte imutável do nosso sistema e, como pode ser observado no diagrama de classes abaixo possui duas dependências. Como queremos máxima flexibilidade, já implementamos nosso código de acordo com o princípio de *Inversão de Dependências*, segundo o qual nosso componente de nível mais alto (Processador) possui como dependências diretas apenas abstrações que, neste caso, são as interfaces *FonteDados* e *Impressor*.

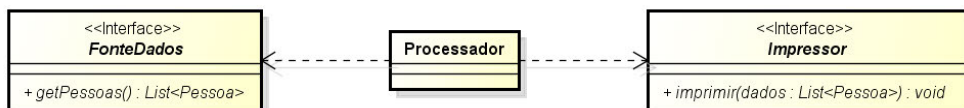


Figura 3.3: Situação Inicial do nosso sistema

3.3 DECLARANDO BEANS

Para que um bean exista são necessários três elementos: as configurações, o container e a classe que o implementa. No caso do Spring as configurações podem vir de três fontes: XML, anotações ou código Java. Nosso foco inicial será o primeiro formato, pois este torna explícito de uma maneira bastante didática o funcionamento do container. Como veremos no transcorrer deste capítulo, as fontes restantes tem como objetivo principal a redução do XML ou mesmo sua eliminação, o que aumenta bastante a produtividade do desenvolvedor.

A configuração deve nos fornecer os dados mínimos para que o container consiga instanciar e gerenciar o ciclo de vida de nossos beans. A única informação que obrigatoriamente devemos fornecer é qual a classe que implementa o bean. Usamos a tag `<bean>` para declarar um bean e nesta o único atributo obrigatório é `class`. Sendo assim, para declararmos uma implementação de nossa interface *FonteDados* que obtenha nossos usuários usando um arquivo texto configuramos o bean tal como no exemplo abaixo:

```
<bean id="fonteDados" class="br.com.casadocodigo.FonteDadosArquivo"/>
```

O atributo *id*, como o próprio nome já diz, identifica unicamente um bean no container. Caso o atributo não esteja presente, o bean seria instanciado e gerenciado pelo Spring da mesma maneira. A diferença no caso seria quem nomearia o bean: o próprio container. É interessante observar que um bean pode ter mais de um nome, mas não se preocupe com isto por enquanto, este tema será abordado mais à frente e só é de fato útil em projetos maiores que envolvam equipes muito grandes.

Se nosso bean já tem um nome, é fundamental definirmos também qual a sua origem, ou seja, qual classe o implementa. Para tal usamos o atributo `class`, o único obrigatório, aonde devemos fornecer o nome completo da classe.

Quando nosso container for instanciado e alimentado com estas configurações, o bean será instanciado e em seguida estará disponível para uso pelo restante do sistema.

No entanto nossa configuração ainda está incompleta. Definir quais objetos instanciar é ótimo, porém o padrão de projeto *factory* já resolveria nosso problema. Neste momento nosso container ainda não passa de um *factory* configurável por XML, o que ainda não justifica a adoção do Spring. O leitor atento já deve ter observado que a classe que estamos instanciando se chama *FonteDadosArquivo*. Que tal se também definíssemos qual o arquivo em que se encontram os dados que queremos instanciar? Entra em ação a tag `<property>`. Podemos definir qual objeto carregar modificando nossa configuração para que fique similar ao exemplo abaixo:

```
<bean id="fonteDados" class="br.com.casadocodigo.FonteDadosArquivo">
  <property name="arquivo" value="/arquivos/pessoas.csv"/>
</bean>
```

Há dois atributos na tag `<property>`. Primeiro é necessário definir qual atributo de nosso objeto deverá ser modificado, o que é feito com o atributo `name`, que faz referência ao padrão *JavaBean* que deve, por convenção, ser adotado por todos os

nossos beans. De acordo com esta propriedade, deve haver implementado na classe *FonteDadosArquivo* um método chamado *setArquivo* que receba como parâmetro um objeto do tipo *String*. O valor a ser *injetado* encontra-se definido no atributo *value*. É interessante observar que não é necessário se preocupar com o tipo a ser recebido pelo setter, pois o tipo é descoberto em tempo de execução pelo container.

Com base nesta configuração mais rica, o container irá executar a sequência de operações descrita abaixo:

- 1) Bean *fonteDados* do tipo *br.com.casadocodigo.FonteDadosArquivo* é instanciado
- 2) O valor da propriedade *arquivo* é definido como sendo igual a *"/arquivos/pessoas.csv"*
- 3) Bean instanciado e configurado disponível para ser usado pelo resto do sistema

É interessante observar que o bean só é disponibilizado para uso após seu estado ter sido inicializado. Como “brinde” acabamos de ganhar a solução para toda uma gama de problemas decorrentes da obtenção concorrente de objetos cuja inicialização do estado não foi totalmente concluída. E este sequer era nosso objetivo inicial!

No entanto isoladamente um bean não é de grande utilidade. O poder do container começa a surgir a partir do momento em que a injeção de dependências começa a entrar em ação. Podemos ver no código abaixo a configuração completa do nosso sistema tal como se encontra no estado atual.

```
<?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.xsd">

  <bean id="fonteDados" class="br.com.casadocodigo.FonteDadosArquivo">
    <property name="arquivo" value="/arquivos/pessoas.csv"/>
  </bean>

  <bean id="impressor" class="br.com.casadocodigo.ImpressorImpl"/>

  <bean id="processador" class="br.com.casadocodigo.Processador">
    <property name="impressor" ref="impressor"/>
    <property name="fonteDados" ref="fonteDados"/>
  </bean>
</beans>
```

```
</bean>  
</beans>
```

A primeira novidade que vemos no arquivo completo é a inclusão da tag `<beans>` que é o elemento raiz do documento. Não é necessário se intimidar com as declarações de namespaces presentes neste documento visto que atualmente a maior parte das IDEs, com seu suporte ao Spring já oferecem a possibilidade de gerar automaticamente este documento para os novatos no framework. Como veremos mais adiante, o Spring oferece uma série de namespaces que tornam a vida do desenvolvedor muito mais fácil ampliando a semântica do XML padrão.

Na versão completa de nossa configuração, devemos focar nossa atenção sobre o bean *processador*. Um novo atributo aparece na tag `<property>`: *ref* (abreviação de *reference*), cujo valor é o nome de um dos beans definidos na configuração. Esta é a tão falada injeção de dependências em ação. Novamente, é importante observar que não precisamos nos preocupar com o tipo dos objetos que estamos injetando pois o próprio container se encarrega desta tarefa. Caso haja algum problema com a tipagem dos objetos, o container irá disparar uma exceção nos informando do erro para que sejam feitas as alterações necessárias nas configurações e os beans se tornem disponíveis.

Novamente, no caso do bean *processador*, este só estará disponível após ter sido instanciado e todas as suas dependências terem sido instanciadas, preparadas e injetadas. É importante observar que nosso container já pode ser visto como uma classe *factory* melhor evoluída que não só decide quais classes instanciar e quais propriedades definir em tempo de execução, como também já inicia o processo de injeção de dependências. No transcorrer deste capítulo a distância entre o padrão *factory* e nosso container tenderá a crescer até chegarmos ao ponto que serão vistos como primos distantes.

Tipos de injeção

O Spring trabalha com dois tipos de injeção de dependências. A que expomos acima é chamada *injeção por setter* (*setter injection*). Como o próprio nome diz, as dependências são injetadas em nosso bean a partir de métodos *setter* que implementamos em nossas classes. Nossa outra opção é a injeção por construtor *constructor injection*.

Para entender melhor a injeção por construtor, imagine que nossa classe *Processador* possua um construtor tal como o abaixo:

```
public Processador(FonteDados fonteDados, Impressor impressor) {  
    setFonteDados(fonteDados);  
    setImpressor(impressor);  
}
```

Para executar a injeção por construtor, entra em ação a tag `<constructor-arg>` que podemos ver em ação no exemplo abaixo:

```
<bean id="processador" class="br.com.casadocodigo.Processador">  
    <constructor-arg ref="fonteDados"/>  
    <constructor-arg ref="impressor"/>  
</bean>
```

Assim como a tag `<property>`, `<constructor-arg>` pode receber como atributo tanto `value` quanto `ref`. No exemplo acima, a ordem de digitação das tags importa. Repare que coincide com a ordem em que os parâmetros são passados ao construtor da classe. É importante mencionar que também é possível digitá-los fora da ordem, mas neste caso faz-se obrigatória a presença do atributo `index` que deve receber um valor numérico que represente a ordem do parâmetro a partir do zero no construtor, tal como pode ser visto no exemplo abaixo:

```
<bean id="processador" class="br.com.casadocodigo.Processador">  
    <constructor-arg ref="impressor" index="1"/>  
    <constructor-arg ref="fonteDados" index="0"/>  
</bean>
```

Uma pergunta pertinente neste momento é a respeito da razão pela qual existem estas duas opções de injeção. Uma só não tornaria o aprendizado do framework mais simples? Talvez, mas com certeza também o tornaria bem mais limitado. Deve ser levado em consideração que um dos objetivos principais por trás do Spring é o reaproveitamento de código legado de modo que sua integração seja a mais leve possível. Por mais leve, como já mencionado, deve ser entendido *com o mínimo possível de dependências com relação ao código fonte do próprio Spring*.

Existe uma longa discussão a respeito de qual seria a melhor forma de injeção. Os mais puristas defendem a injeção por construtor como a melhor. O argumento é que ao definirmos em um construtor todas as dependências de uma classe tornamos evidente quais atributos não devem em hipótese alguma ser alterados após a instanciamento da classe. O problema desta abordagem é que conforme nosso código

se desenvolve e novos atributos são incluídos em nossas classes, o número de parâmetros a ser passado para nossos construtores tende a crescer e, consequentemente, sua manutenção vai se tornando mais complicada.

Em prol da injeção por setter estão dois argumentos: a de que permite a modificação das dependências injetadas após a instanciação da classe e também o fato de evitar problemas decorrentes de construtores com número excessivo de parâmetros.

No final das contas, nem uma nem outra alternativa é a mais recomendada. Qual opção usar deve sempre ser uma decisão tomada com base no bom senso, e não apenas no que um guru ou outro afirma a respeito do assunto.

Nada impede também que usemos também uma solução mista envolvendo os dois tipos de injeção de dependências em um mesmo bean. A configuração abaixo é um exemplo disto:

```
<bean id="processador" class="br.com.casadocodigo.Processador">
  <constructor-arg ref="fonteDados" index="0"/>
  <constructor-arg index="1">
    <null/>
  </constructor-arg>
  <property name="impressor" ref="impressor"/>
</bean>
```

Neste exemplo reaproveitamos o mesmo construtor passando o valor *null* para o segundo argumento e injetando a dependência *impressor* via *setter*.

Dando nome aos beans

Assim como pessoas possuem apelidos, o mesmo podemos fazer com beans. Há dois atributos da tag `<bean>` que podemos usar para nomearmos nossos componentes. O primeiro já conhecemos: trata-se de `id`. Usamos este atributo quando precisamos definir um nome único para nossos beans.

O novo atributo se chama `name`, que é usado quando definimos mais de um nome para o mesmo bean. Neste atributo podemos inclusive incluir uma lista de nomes, que devem estar separados pelo caractere de espaço, ponto e vírgula ou vírgula. O código abaixo expõe um exemplo de uso destes atributos:

```
<!-- Um bean com dois nomes: um definido por id, outro por name -->
<bean id="impressor" name="impressorArquivo"
      class="br.com.casadocodigo.ImpressorImpl"/>
<!-- Um bean com dois nomes, todos definidos na tag name -->
```

```
<bean name="impressor, impressorArquivo"  
      class="br.com.casadocodigo.ImpressorImpl"/>
```

A possibilidade de criar mais de um nome para o mesmo bean é um recurso pouco usado na maior parte dos projetos. Normalmente esta possibilidade é explorada em projetos maiores nos quais times distribuídos desenvolvem partes do sistema que serão integrados posteriormente. É uma maneira interessante de manter os padrões de nomenclatura adotados por times distribuídos. Se seu projeto não for tão grande assim simplesmente não faz mais sentido, além de ser bem mais produtivo adotar um padrão de nomes unificado.

Caso seu projeto não possua um padrão de nomenclatura, uma boa prática é usar o adotado e sugerido pela equipe da SpringSource. Esta sugere que beans sigam o padrão camel cased, no qual o nome sempre é inicialmente digitado em letras minúsculas. Caso o nome seja composto, cada palavra é separada digitando seu primeiro caractere em letra maiúscula, tal como nestes exemplos: *importador*, *importadorArquivo*, *importadorBancoDeDados*.

3.4 INSTANCIAÇÃO POR FACTORY METHOD

Nem sempre é interessante que um bean seja instanciado por um construtor como fizemos até o momento. Muitas vezes é necessário que estes sejam instanciados usando um método presente em uma classe *factory*.

Factory é um padrão de projeto cuja descrição mais famosa se encontra no livro “Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos” da GoF [4]. Esta prática de desenvolvimento consiste na implementação de uma classe, chamada *factory* (*fabrica*), cujo objetivo é a instanciação de novos objetos. Toda classe *factory* possui pelo menos um método instanciador (*factory method*) responsável pela instanciação (e talvez preparo) de novos objetos.

No software de exemplo que estamos usando neste capítulo poderíamos ter implementado uma classe *factory* tal como a exposta no diagrama abaixo:

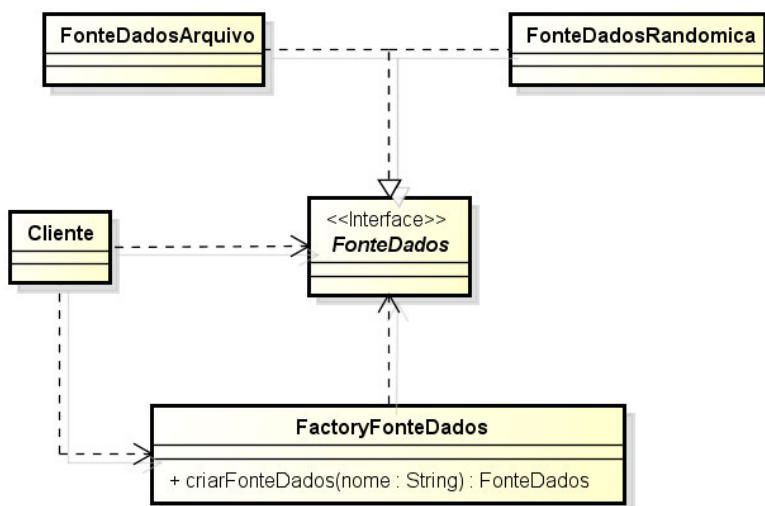


Figura 3.4: Exemplo de aplicação do padrão Factory

Neste exemplo, a classe *FactoryFonteDados* possui um *factory method* chamado *criarFonteDados* que recebe como parâmetro uma *String* representando o tipo de *FonteDados* que o cliente deseja. A classe cliente obtém como resultado uma instância de *FonteDados* com base no parâmetro fornecido e, com isto, fica completamente desacoplado de todas as implementações desta interface, além de também não precisar se preocupar com qualquer lógica que possa ser necessária na implementação destes objetos.

Na API JDBC do Java também podemos ver uma aplicação deste padrão, tal como no exemplo abaixo:

```
import java.sql.*
// Ignorando início do código
Class.forName("com.mysql.jdbc.Driver");
Connection con =
    DriverManager.getConnection("jdbc:mysql://localhost/database");
```

O método *getConnection* da classe *DriverManager* é uma implementação do padrão *Factory*. É importante observar que *factory methods* não precisam receber parâmetros. Interessante observar a aplicação aqui do conceito de inversão de controle: delegamos à classe *factory* a responsabilidade de escolher qual a implementação correta de *FonteDados* a ser obtida, o que, de outra maneira, caberia à classe cliente, cujo

acoplamento às demais partes do sistema seria bem maior.

A instanciação de beans a partir de um *factory method* pode se dar de duas maneiras: usando um método estático ou não. Tendo como base o exemplo exposto no box *Factory* podemos ver no código abaixo como declarar a instanciação de um bean a partir de um método estático.

```
<bean id="fonteDadosEstatico"
      class="br.com.casadocodigo.FactoryFonteDados"
      factory-method="criarEstatico">
    <constructor-arg index="0" value="arquivo"/>
</bean>
```

Em nossa classe *FactoryFonteDados* implementamos o método estático *criarEstatico* de tal modo que receba como parâmetro um nome que identifique qual tipo de *FonteDados* deve ser instanciada. Se o valor for igual a *arquivo*, será retornada uma instância de *FonteDadosArquivo*, e se for randômico, *FonteDadosRandomico*.

É importante observar que, apesar de nosso objetivo ser a obtenção de uma instância de *FactoryMethod*, o atributo *class* da tag `<bean>` recebe como valor não o nome de uma classe que implemente esta interface, mas sim o nome da nossa classe *factory*. Dado que o método estático recebe parâmetros, reaproveitamos a tag `<constructor-arg>`, passando como valor o tipo de bean que desejamos como retorno.

Usar um método não estático é igualmente simples. Basta que o *factory method* a ser invocado esteja implementado em algum outro bean declarado em nosso arquivo de configuração. No exemplo abaixo podemos ver um exemplo deste tipo de configuração.

```
<!-- O bean que contém nosso factory method -->
<bean id="factory" class="br.com.casadocodigo.FactoryFonteDados"/>

<bean id="fonteDadosInstancia" factory-bean="factory"
      factory-method="criar">
    <constructor-arg index="0" value="randomico"/>
</bean>
```

Observe que neste caso não é necessário incluir o atributo *class* na tag `<bean>` quando declaramos o bean *fonteDadosInstancia*. A razão é simples: neste caso não faz sentido, dado que o tipo será decidido em tempo de execução e o Spring não tem como, de antemão, saber qual seria este.

Finalmente, para declarar um bean a ser instanciado por um *factory method* que não receba parâmetros como valor, basta não incluir mais a tag `<constructor-arg>`, como pode ser visto no exemplo abaixo:

```
<bean id="fonteDadosArquivo" factory-bean="factory"
      factory-method="criar">
  <property name="arquivo" value="/caminho/para/arquivo"/>
</bean>
```

É importante observar que a instanciação por métodos instanciadores não nos impede de injetar valores nos atributos de nossos beans, como pode ser visto no nosso último exemplo.

3.5 MAPEANDO ATRIBUTOS COMPLEXOS

O formato XML nos permite definir integralmente o estado inicial de nossos beans. Isto implica que a tag `<property>` é bem mais que o que mostramos até este momento. Esta, além de possibilitar definirmos o valor de atributos primitivos e strings também possibilita o mapeamento de atributos complexos como listas, mapas, instâncias de `java.util.Properties` e arrays. E estas são apenas algumas das possibilidades que iremos expor nesta seção.

Listas

Imagine uma implementação da interface *FonteDados* tal como a abaixo:

```
class FonteDadosArquivos implements FonteDados {
  private List<String> arquivos;
  public List<String> getArquivos() {return arquivos;}
  public void setArquivos(List<String> lista) {arquivos = lista;}
  // restante da classe ignorado para aumentar a clareza
}
```

Para mapear uma lista de strings, a tag `<property>` aceita em seu interior a tag `<list>`, que podemos ser usada no código abaixo:

```
<bean class="br.com.casadocodigo.FonteDadosArquivos">
  <property name="arquivos">
    <list>
      <value>/caminho/para/arquivo1.csv</value>
```

```
        <value>/caminho/para/arquivo2.csv</value>
        <value>/caminho/para/arquivo3.csv</value>
    </list>
</property>
</bean>
```

A tag `<value>` é usada para representar um valor primitivo ou *String*. Poderíamos também ter usado a tag `<array>` ao invés de `<list>` e obteríamos o mesmo resultado. Ambas injetariam um objeto do tipo *java.util.ArrayList* no atributo *arquivos* do bean.

Imagine agora que tenhamos alterado nossa implementação da classe *FonteDadosArquivos* para que seu atributo *arquivos* não receba uma lista de strings, mas sim uma lista de instâncias do tipo *java.io.File* tal como no snippet abaixo:

```
class FonteDadosArquivos implements FonteDados {
    private List<java.io.File> arquivos;
    //restante da classe ocultado para facilitar a leitura
}
```

A tag `<list>` também aceita em seu interior a tag `<bean>`. Sendo assim, é possível declarar beans anônimos tal como no exemplo abaixo sem problema algum.

```
<bean class="br.com.casadocodigo.FonteDadosArquivos">
    <property name="arquivos">
        <list>
            <bean class="java.io.File">
                <constructor-arg value="/caminho/para/arquivo1"/>
            </bean>
            <bean class="java.io.File">
                <constructor-arg value="/caminho/para/arquivo2"/>
            </bean>
            <bean class="java.io.File">
                <constructor-arg value="/caminho/para/arquivo3"/>
            </bean>
        </list>
    </property>
</bean>
```

Caso os beans não fossem anônimos, outra tag útil é a tag `<ref>`, que aponta para a referência de um bean externo, tal como no exemplo abaixo:

```
<list>
  <ref bean="beanArquivo1"/>
  <ref bean="beanArquivo2"/>
</list>
```

Mapas

Uma outra implementação da interface *FonteDados* poderia ter suas fontes de dados mapeadas por tipo em uma estrutura do tipo *java.util Map* tal como no exemplo abaixo:

```
public class FonteDadosMapaArquivos implements FonteDados{
    private Map<String, Object> mapa;
    public Map<String, Object> getMapa() {return mapa;}
    public void setMapa(Map<String, Object> valor) {mapa = valor;}
    // restante oculto para maximizar a clareza do exemplo
}
```

Entra em ação a tag `<map>`, que usaríamos para mapear o atributo *mapa* da classe acima da seguinte maneira:

```
<bean name="fonteDadosMapa"
      class="br.com.casadocodigo.FonteDadosMapaArquivos">
  <property name="mapa">
    <map>
      <entry key="arquivo">
        <ref bean="arquivo"/>
      </entry>
      <entry key="arquivoString"
              value="/caminho/para/arquivo.csv"/>
    </map>
  </property>
</bean>
```

No interior da tag `<map>` incluímos uma lista de elementos do tipo `<entry>` que, como o próprio nome indica, representam uma entrada no mapa. O atributo `key` de `<entry>` identifica sua chave. Em seu interior são aceitos elementos dos tipos `<ref>`, `<bean>`, `<value>`, `<list>`, `<array>` e `<map>`.

Properties

Um dos objetos mais usados da API Java padrão é a classe *java.util.Properties*. Como já era de se esperar, o formato XML do Spring oferece suporte especial a esta classe. Um exemplo prático é a classe *SessionFactory* do Hibernate para a qual o framework, como veremos mais adiante no livro, integra-se muito bem.

O snippet XML abaixo é quase auto explicativo:

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
  <!--
    Restante oculto para que possamos
    focar apenas no mapeamento de propriedades
  -->
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">
        org.hibernate.dialect.MySQL5InnoDBDialect
      </prop>
      <prop key="hibernate.hbm2ddl.auto">update</prop>
    </props>
  </property>
</bean>
```

A tag `<props>` representa um objeto do tipo *Properties*. Em seu interior é aceita a tag `<prop>` que, como o próprio nome já diz, é usada para mapear as propriedades aceitas pelo objeto. No interior da tag é incluído o valor da propriedade que queremos mapear.

3.6 USANDO O CONTAINER

Com a definição de nossos beans pronta chegou o momento de usá-los, ou seja, finalmente veremos como instanciar e usar nosso container. Mencionei anteriormente que o container do Spring possui vêm em duas versões: *BeanFactory* e *ApplicationContext*. Qual usar?

A equipe da SpringSource já nos responde isto na própria documentação do projeto [14]: o uso do *BeanFactory* só se justifica em situações nas quais as restrições computacionais sejam extremas como, por exemplo na implementação de applets Java. Na prática, entretanto, a diferença no consumo de recursos entre as duas versões é mínima, o que torna seu uso desnecessário. Entra a pergunta: se posso ter

tudo, por que me restringir ao mínimo? Sendo assim, no decorrer deste livro usaremos apenas o *ApplicationContext*.

Atualmente *BeanFactory* ainda existe apenas para fins de retrocompatibilidade. Dado que a interface *ApplicationContext* a estende e a própria *SpringSource* a desaconselha, usaremos apenas *ApplicationContext* como nosso container daqui para frente.

Instanciando o container

Para usarmos o container nosso primeiro passo é instanciá-lo. Das diversas variações de *ApplicationContext*, nosso foco inicial será em apenas duas que são *ClassPathXmlApplicationContext* e *FilePathXmlApplicationContext*. A diferença entre os dois está apenas na fonte que será usada para obter os documentos XML responsáveis pela configuração de nossos beans que são respectivamente o *classpath* da aplicação e o sistema de arquivos local aonde nosso projeto será executado.

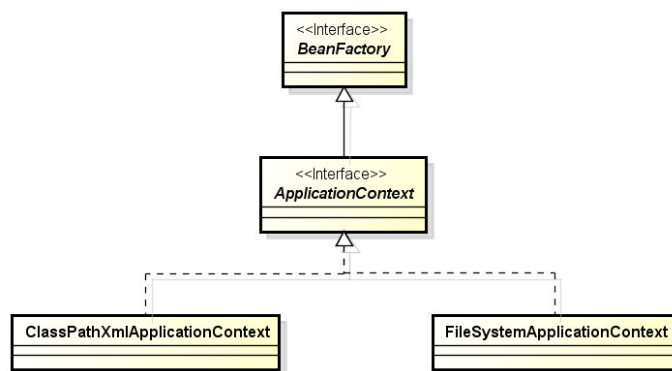


Figura 3.5: Hierarquia dos ApplicationContexts

O construtor mais usado recebe como parâmetro uma *String* representando a localização do arquivo de configuração. O exemplo abaixo instancia um container que busca o arquivo *spring.xml* a partir do *classpath* da aplicação.

```
import org.springframework.context.ApplicationContext;
import
    org.springframework.context.support.ClassPathXmlApplicationContext;
(...)
```

```
ApplicationContext context =  
    new ClassPathXmlApplicationContext("spring.xml");
```

Obtendo beans (lookup)

A interface *ApplicationContext* não poderia ser mais simples: dada nossa configuração anterior, se desejássemos obter o bean *processador* a partir do seu identificador basta usar a função *getBean* como no exemplo abaixo:

```
ApplicationContext context =  
    new ClassPathXmlApplicationContext("spring.xml");  
Processador bean = (Processador) context.getBean("processador");  
bean.processe();
```

Existem basicamente duas versões do método *getBean* que podem ser usadas quando nosso objetivo é obter um bean a partir do seu nome.

```
// a versão mais primitiva que nos obriga a fazer um type casting manual  
Processador bean = (Processador) context.getBean("processador");  
// Usando generics  
Processador bean = context.getBean("processador", Processador.class);
```

Como mencionado acima, nem sempre damos nomes aos nossos beans. Nestes casos, obtemos o bean sobre o qual iremos trabalhar passando como parâmetro a classe ou interface do objeto desejado.

```
// buscando pela classe  
Processador bean = context.getBean(Processador.class);
```

Há também situações em que por razões diversas precisamos de mais de um bean que realizem uma mesma classe ou interface e precisamos processar algumas ou todas estas instâncias. Entra em ação a variante *getBeansOfType*, que retorna um mapa no qual a chave representa o identificador único de cada um dos beans obtidos.

```
Map<String, Processador> beans =  
    context.getBeansOfType(Processador.class);
```

O leitor atento neste ponto pode se perguntar: o que ocorre quando pedimos um bean que não exista? Nesta situação o container dispara uma exceção do tipo *org.springframework.beans.BeansException* da qual podemos tirar algum proveito como no seguinte trecho:

```
Processador processador = null;
try {
    processador = context.getBean("processador");
} catch (org.springframework.beans.BeansException ex) {
    // instancie um processador de emergência
    // caso nenhum esteja disponível no container
    processador = new Processador();
    // prepare-o a gosto :)
}
```

É importante mencionar que *BeansException* é uma exceção de tempo de execução (não checada) e, portanto, você não precisa incluir toda obtenção de um bean dentro de um bloco *try..catch*. Caso ocorra algum erro durante a instanciação, injeção de dependências ou gerência de ciclo de vida dos beans normalmente é fatal.

E por mais incrível que pareça, conhecendo apenas estas variantes dos métodos de *lookup* o desenvolvedor iniciante já consegue trabalhar (e bem) com o container do Spring.

3.7 O CICLO DE VIDA DO CONTAINER

Acabamos de ver de uma maneira bastante rústica como trabalhar com o Spring. Basicamente declaramos nossos beans em nosso arquivo de configuração, instanciamos nosso container com esta e em seguida executamos chamadas a funções de lookup para que possamos trabalhar com nossos beans. Obviamente o funcionamento real do Spring é bem mais rico e oferece uma gama bem maior de possibilidades ao desenvolvedor.

O funcionamento real do container pode ser melhor compreendido observando o diagrama abaixo:

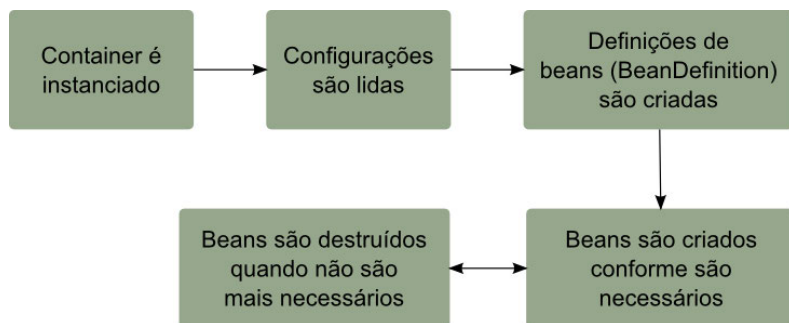


Figura 3.6: O ciclo de vida de um container

O container assim como o bean possui seu próprio ciclo de vida que inicia-se no momento em que este é instanciado e alimentado com nossas configurações. Cada declaração de bean que escrevemos dará origem a um objeto do tipo *BeanDefinition*. Esta classe encontra-se implementada no módulo *Beans* e é a responsável por representar internamente o modo como definimos a configuração de cada um de nossos beans.

Para facilitar a sua compreensão é interessante pensar na classe *BeanDefinition* como a *receita* seguida pelo Spring no gerenciamento dos beans manipulados pelo container. *BeanDefinition* diz ao container qual classe instanciar, quais atributos deverão ser preenchidos, as dependências a serem injetadas e como deve ser tratado o ciclo de vida de cada bean.

Com todas estas informações em memórias e prontas para uso, inicia-se o ciclo de vida dos próprios beans, que serão instanciados e destruídos conforme seja necessário. O leitor atento pode-se perguntar neste momento: *quando um bean deve ser criado ou destruído e como isto é definido?*, o que é o assunto da próxima seção.

3.8 ESCOPOS

Quando um bean deve ser instanciado ou destruído e quantas instâncias podem ser criadas é definido pelo seu escopo (*scope*). Um dos objetivos do container é tirar das mãos do desenvolvedor a responsabilidade pela instanciação dos objetos.

O escopo padrão do Spring é o *singleton* e, portanto, é aquele adotado em todos os beans quando não definimos nossos escopos explicitamente. Um bean declarado com este escopo é instanciado uma única vez pelo container e mantido em memória até que o container seja destruído. Antes de expor as opções oferecidas pelo Spring

é interessante pensarmos em algumas questões que nos ajudam a escolher a opção correta de escopo para nossos beans.

Quão cara é a construção de um bean? - Imagine um bean que represente um objeto caro de ser instanciado e configurado como por exemplo um pool de conexões ou um *EntityManager* JPA. Estes são os tipos de objeto que queremos ver construídos uma única vez durante a execução de nossos sistemas e não toda vez que se tornam necessários por razões de performance.

O estado interno de um objeto pode ser compartilhado por mais de um usuário do sistema? - No desenvolvimento de aplicações corporativas *acesso concorrente* é um dos maiores problemas a ser resolvido. Em sua raiz está o estado compartilhado que é a principal razão por trás de soluções como mecanismos de sincronização. Imagine que no exemplo de nosso capítulo cada instância de *Processador* para executar seu trabalho para um usuário precise possuir um estado único para aquele usuário. Agora imagine que a instância de um usuário seja compartilhada por outro. Uma série de consequências inesperadas poderiam ocorrer. A escolha do escopo correto para nossos beans evita este tipo de situação de uma forma bastante simples.

Quanto tempo uma instância realmente deve existir? - O que nos volta à primeira questão: objetos cuja instanciação seja cara normalmente duram o tempo de execução do sistema. No entanto não é interessante que alguns objetos durem muito mais que uma requisição. Por exemplo: um objeto que contenha os dados digitados em um formulário web não é feito para que dure mais do que um conjunto finito (e preferencialmente pequeno) de requisições.

Estes três questionamentos abrangem a maior parte dos problemas com os quais precisamos nos preocupar no momento em que definimos o escopo de nossos beans. Mantendo-as em mente, é hora de apresentar os escopos que o Spring nos oferece por padrão e como instruir nosso container a aplicá-los em nossos objetos.

Singleton

Trata-se do escopo padrão aplicado pelo Spring. Garante que apenas uma instância do bean seja instanciada pelo container e compartilhada por todos os outros beans que a usem como dependência. É recomendada nos casos abaixo:

- A construção do objeto é cara, como por exemplo um pool de conexões, o objeto *SessionFactory* do *Hibernate* ou o *EntityManager* do JPA. Nestes casos, é interessante manter a mesma instância do objeto compartilhada por todos os

objetos clientes do sistema, garantindo assim que sua instanciação não prejudique a performance do sistema como um todo.

- O estado do objeto pode ser compartilhado por mais de um objeto cliente sem problema ou então é interessante que o estado seja compartilhado por mais de um objeto.

No nosso exemplo o bean *fonteDados* é um caso em que o escopo se aplica bem. Como é uma fonte de dados, internamente esta classe poderia ter algum mecanismo de cacheamento que favorecesse chamadas subsequentes aos métodos de pesquisa implementados. É interessante observar que o escopo no entanto não foi aplicado ao bean *impressor*, visto que não desejamos que seu estado interno seja compartilhado por mais de um objeto cliente da mesma.

Para declarar um bean como um singleton há duas opções: você pode não fazer nada visto ser o escopo padrão usado pelo Spring ou incluir na definição deste o atributo *scope* com o valor *singleton* tal como no exemplo abaixo:

```
<bean id="fonteDados" class="br.com.casadocodigo.FonteDadosArquivo"
      scope="singleton">
  <property name="arquivo" value="/arquivos/pessoas.csv"/>
</bean>
```

O código abaixo ilustra bem o comportamento do escopo *singleton*.

```
FonteDadosArquivo fonteDados1 =
    context.getBean("fonteDados", FonteDadosArquivo.class);
FonteDadosArquivo fonteDados2 =
    context.getBean(FonteDadosArquivo.class);
// O teste abaixo falharia caso fonteDados1
// e fonteDados2 fossem instâncias diferentes
assert fonteDados1 == fonteDados2;
```

Prototype

O escopo *prototype* garante que toda vez que o container receba uma requisição por um bean uma nova instância seja criada. É normalmente adotado em situações nas quais o bean tenha curta duração. Seu uso, de acordo com as questões que apresentamos acima se faz recomendado quando:

- A construção do objeto é barata.

- O seu estado não deve ser compartilhado por mais de uma classe cliente
- O objeto seja útil por um curto espaço de tempo

Definimos um bean como *prototype* passando o valor de mesmo nome do contexto para o atributo *scope* tal como no exemplo abaixo:

```
<bean id="processador" class="br.com.casadocodigo.Processador"
      scope="prototype">
  <property name="impressor" ref="impressor"/>
  <property name="fonteDados" ref="fonteDados"/>
</bean>
```

Assim como no caso do escopo *singleton*, podemos escrever um teste bastante simples para ilustrar seu funcionamento:

```
Processador processador1 =
    context.getBean("processador", Processador.class);
Processador processador2 =
    context.getBean(Processador.class);
// O teste abaixo falharia caso estivéssemos lidando
// com a mesma instância
assert processador1 != processador2
```

Request e Session

Para o ambiente web Spring nos oferece mais dois escopos bastante interessantes que são *request* e *session* que mantêm uma instância do bean durante a existência respectivamente de uma requisição e de uma sessão de usuário. Como trataremos do desenvolvimento para este ambiente apenas posteriormente neste livro, iremos expor aqui apenas um exemplo de como declaramos beans neste escopo.

```
<!-- Usando o escopo de sessão -->
<bean id="processador" class="br.com.casadocodigo.Processador"
      scope="session"/>

<!-- Usando o escopo de requisição -->
<bean id="processador" class="br.com.casadocodigo.Processador"
      scope="request"/>
```

É um excelente exemplo de como o escopo evita problemas de acesso concorrente em nosso código. Usando o escopo de sessão, cada usuário do sistema possui

sua própria instância do bean, cujo estado diz respeito apenas ao que aquele usuário está fazendo naquele momento.

Escopos customizados

Os quatro escopos acima são os mais comumente usados pelo Spring Framework e conseguem atender à maior parte dos casos. Uma das características mais interessantes do Spring é sua flexibilidade. Sendo assim, caso ocorra uma situação na qual os escopos oferecidos não lhe atendam, é possível criar seu próprio escopo, bastando para tal escrever uma implementação da interface `org.springframework.beans.factory.config.Scope`.

A escrita de escopos customizados normalmente é feita por desenvolvedores responsáveis pela implementação de novos frameworks ou bibliotecas que desejem adicionar suporte especial ao Spring Framework. É muito útil em contextos nos quais é necessário manter os beans somente enquanto determinado tipo de conversação esteja ocorrendo como, por exemplo, em situações nas quais seja necessário tratar transações distribuídas ou integração com sistemas remotos.

Trata-se de um uso bastante específico que foge do objetivo deste livro e, portanto, fica aqui apenas a menção a esta possibilidade oferecida pelo framework.

3.9 INSTANCIAÇÃO TARDIA

Por padrão as implementações de *ApplicationContext* inicializam todos os beans de escopo *singleton* quando sua configuração é parseada. Esta é uma prática bastante sadia, visto que caso ocorra algum problema neste processo o desenvolvedor é imediatamente avisado do problema, o que torna este comportamento uma importante ferramenta na etapa de desenvolvimento de nossos projetos. No ambiente de produção a história é diferente: há beans cuja inicialização no *startup* do sistema não é necessária ou mesmo desejável. É o caso de beans cujo uso ocorre somente um bom tempo após a inicialização do sistema *se* de fato chegarem a ser usados.

Nestas situações faz sentido postergar a inicialização do componente somente para quando esta de fato se fizer necessária. Com isto diminuimos o tempo de *boot* do sistema e ainda economizamos memória. Beans com inicialização tardia são chamados pela equipe da SpringSource de *lazy-initialized beans*. No formato XML há duas maneiras de definir este comportamento. A primeira é adicionando o atributo `lazy-init` na tag `<bean>` com o valor `true` como no código abaixo:


```
<bean id="fonteDados" class="br.com.casadocodigo.FonteDadosArquivo"
      lazy-init="true"/>
```

A segunda maneira consiste em definir o comportamento padrão para todos os beans definidos no arquivo de configuração adicionando o atributo `default-lazy-init` na tag `<beans>` tal como no código abaixo:

```
<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.xsd"
default-lazy-init="true"
>
<!-- Conteúdo omitido para aumentar a clareza -->
</beans>
```

Deste modo, os beans só serão inicializados quando forem usados pela primeira vez.

3.10 APROVEITANDO O CICLO DE VIDA DOS BEANS

O fato de delegarmos a responsabilidade pela instanciação e destruição de nossos beans ao container não implica que não possamos ou devamos tirar proveito do ciclo de vida dos mesmos. Todo bean tem “direito” a dois métodos de callback que são executados pelo container logo após sua inicialização e antes de sua destruição tal como pode ser visto no ciclo de vida ampliado que pode ser visto na imagem abaixo.

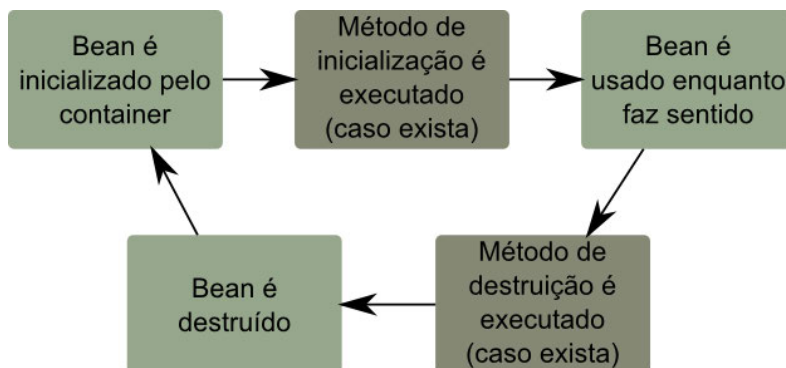


Figura 3.7: O ciclo de vida do bean

Iremos ver duas maneiras de tirar proveito destes métodos: uma é a pesada através da implementação de interfaces, enquanto a outra é a leve usando apenas os mecanismos de configuração do container.

Inicialização do bean

Talvez seja interessante que, uma vez tenham sido injetadas todas as dependências em seu objeto algum código de inicialização seja executado. Por exemplo: para garantir um tempo de resposta menor, nosso bean *fonteDados* poderia, logo após ser iniciado já carregar todos os registros presentes em um arquivo de nosso servidor. Como mencionado, neste caso há duas alternativas.

A primeira alternativa é a que chamamos de *pesada*, visto que envolve a implementação da interface *org.springframework.beans.factory.InitializingBean* e, portanto, adiciona ao nosso projeto a dependência física a uma classe do Spring Framework. Podemos modificar nossa classe *FonteDadosRandomico* portanto para que fique similar ao código abaixo:

```
import org.springframework.beans.factory.InitializingBean;
public class FonteDadosArquivo implements FonteDados, InitializingBean {
    // O método da interface que precisa ser implementado
    public void afterPropertiesSet() throws Exception {
        // Código responsável pelo pré-carregamento dos dados
    }
    // Restante da classe ignorado para facilitar a leitura
}
```

O nome do único método a ser implementado por esta interface é bastante instrutivo: *afterPropertiesSet*. Uma vez que o processo de preparo e injeção de dependências do bean esteja completo este método será invocado pelo container. Como mencionado anteriormente, uma das vantagens obtidas a partir deste método de callback é que evitamos uma série de problemas que poderíamos enfrentar em situações nas quais um objeto é usado antes de ser completamente inicializado. Nosso bean só estará disponível para uso a partir do momento em que este método for executado.

A segunda alternativa, e a mais recomendável por manter nosso código completamente desacoplado do Spring envolve alterarmos apenas os nossos arquivos de configuração. Basta adicionarmos o atributo *init-method* à tag *<bean>* como é feito no exemplo abaixo:

```
<bean id="fonteDados" class="br.com.casadocodigo.FonteDadosArquivo"
    init-method="init"/>
```

Há uma única regra neste caso: o método deve ser do tipo *void* e não receber nenhum argumento como valor. Caso o método não exista uma exceção do tipo *org.springframework.beans.factory.BeanCreationException* será lançada informando a ausência do método declarado.

É possível minimizar a quantidade de configurações a ser digitada e ainda padronizar o comportamento do seu projeto inteiro adicionando o atributo `default-init-method` à tag `<beans>` como no exemplo abaixo:

```
<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.xsd"
default-init-method="init">
<!-- Ocultado o conteúdo para facilitar a leitura -->
</beans>
```

A mesma condição do atributo `init-method` é mantida. Caso o método exista em qualquer um dos beans declarados, este será executado. Caso o método não exista, nenhuma exceção será disparada. É importante mencionar que a presença do atributo `init-method` em uma tag `<bean>` sobrescreve o comportamento padrão que definimos na tag `<beans>`.

Finalizando um bean

Há momentos em que é interessante executar algum código no momento em que nossos beans são finalizados. Este normalmente é o ponto em que são executadas as operações de limpeza como por exemplo liberação de recursos ou mesmo o envio de mensagens para sistemas remotos informando o fato de determinado objeto estar sendo deposto. Basicamente o mesmo procedimento que aplicamos no callback de inicialização será repetido.

A primeira opção e menos recomendada é a implementação da interface *org.springframework.beans.factory.DisposableBean*. Um exemplo de sua implementação pode ser visto no código abaixo:

```
import org.springframework.beans.factory.DisposableBean;
public class FonteDadosArquivo implements FonteDados, InitializingBean {
    // O método da interface que precisa ser implementado
    public void destroy() throws Exception {
        // recursos são liberados e operação de limpeza executada
    }
}
```

```
    }  
    // Restante da classe ignorado para facilitar a leitura  
}
```

Há um único método a ser implementado que é o *destroy*. É interessante observar que neste caso temos uma vantagem em cima do próprio comportamento padrão da linguagem Java, na qual podemos até implementar métodos destrutores em nossas classes, mas não temos certeza de quando e mesmo se este será de fato executado. Temos aqui uma previsibilidade bem maior a respeito da execução do nosso método destrutor.

A opção mais recomendada para obter é quase idêntica à que adotamos no caso do callback de inicialização. Na tag `<bean>` adicionamos o atributo `destroy-method` e, se quisermos padronizar a execução do método de finalização em todos os beans, é incluído o atributo `default-destroy-method` à tag `<beans>`. As mesmas regras mencionadas para o método de callback de inicialização se aplicam neste caso. O código abaixo expõe um exemplo desta configuração:

```
<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.xsd"  
default-destroy-method="destroy">  
  
    <bean id="fonteDados" class="br.com.casadocodigo.FonteDadosArquivo"  
        destroy-method="cleanup"/>  
  
</beans>
```

É importante deixar aqui uma nota a respeito do comportamento dos métodos de callback de destruição quando o bean possui escopo *prototype*. Estes beans não possuem a sua finalização gerenciada pelo container: são apenas inicializados e em seguida excluídos do mesmo, não sendo cacheados como ocorre com os demais escopos que foram apresentados anteriormente.

3.11 QUANDO O BEAN CONHECE SEU CONTAINER

Apesar de um dos objetivos principais do Spring é ser o mais desacoplado possível de nossas classes de negócio, há momentos em que é interessante a existência de algum acoplamento nesta direção. É muito fácil tornar um

bean consciente de sua condição. Basta que sua classe implemente a interface `org.springframework.context.ApplicationContextAware` caso estejamos trabalhando com um container do tipo `ApplicationContext` ou implemente a interface `org.springframework.beans.factory.BeanFactoryAware` caso estejamos lidando com um container do tipo `BeanFactory`.

Ambas as interfaces possuem apenas um método que deve ser implementado tal como exposto no código abaixo:

```
public interface ApplicationContextAware extends Aware {
    void setApplicationContext(ApplicationContext applicationContext)
        throws BeansException;
}

public interface BeanFactoryAware extends Aware {
    void setBeanFactory(BeanFactory beanFactory) throws BeansException;
}
```

Um bom exemplo de uso deste comportamento ocorre quando um bean do tipo *prototype* acidentalmente se torna um *singleton*, tal como pode ser observado na configuração abaixo:

```
<bean id="impressor" class="br.com.casadocodigo.ImpressorImpl"
      scope="prototype"/>

<bean id="processador" class="br.com.casadocodigo.Processador"
      scope="singleton">
    <property name="impressor" ref="impressor"/>
</bean>
```

Apesar do bean *impressor* ser declarado como *prototype*, uma de suas instâncias, aquela a ser injetada no bean *impressor* será um “singleton acidental”, dado que todo bean pertencente ao escopo *singleton* só é instanciado uma vez. E se nosso bean *processador* sempre precisar de uma nova instância de *impressor*? Uma solução para este problema é torná-lo consciente do seu container implementando a interface `ApplicationContextAware`.

Dado que todo `ApplicationContext` na realidade estende a interface `BeanFactory`, ao implementarmos a interface `BeanFactoryAware` estaremos tornando nossa classe independente do tipo de container. Podemos alterar seu código fonte para que fique similar ao código abaixo o que, ao menos em teoria, resolveria o problema do “singleton acidental”.

```
public class Processador implements BeanFactoryAware {

    private BeanFactory beanFactory;

    public void setBeanFactory(BeanFactory bf) throws BeansException {
        this.beanFactory = bf;
    }

    /* Resolvido o problema. Agora sempre será
    retornada uma nova instância de Impressor,
    e a declaração desta dependência em nosso arquivo de configuração
    se torna desnecessária.*/
    public Impressor getImpressor() {
        return beanFactory.getBean(Impressor.class);
    }
}
```

3.12 MODULARIZANDO A CONFIGURAÇÃO

Conforme nosso sistema vai se desenvolvendo novos beans vão aparecendo e, consequentemente, o tamanho dos nossos arquivos de configuração aumentam. É importante lembrar que a complexidade não se encontra apenas em nosso código fonte, mas também nos nossos artefatos de configuração. Se não houver uma solução adequada para que possamos melhor organizar o modo como declaramos nossos beans com certeza a manutenção do nosso software será altamente comprometida.

O Spring nos permite modularizar nossos arquivos de configuração. Assim podemos organizar nossos beans de maneira lógica facilitando a manutenção. É uma prática comum, por exemplo, ter um arquivo apenas para a declaração de beans de acesso a dados, outro para geração de relatórios e por assim em diante. Caso sua configuração seja modular, todas as implementações de *ApplicationContext* oferecem um construtor que recebe como parâmetro um array de strings representando a localização dos seus arquivos de configuração. No exemplo abaixo podemos ver um exemplo deste construtor na classe *ClassPathXmlApplicationContext*.

```
ApplicationContext context =
    new ClassPathXmlApplicationContext({"spring-daos.xml",
    "spring-relatorios.xml", "spring-processamento.xml"});
```

Uma vez instanciado, o container irá mesclar internamente os três arquivos, de tal forma que, na prática, todos se tornam na realidade um só. É também possível

que um arquivo de configuração importe o conteúdo de outro. Para tal, usamos a tag `<import>`. Esta preferencialmente deve ser incluída no início do arquivo que está a importar um novo recurso, e possui um único atributo: `resource`, cujo valor deve ser o caminho relativo ao arquivo a ser importado. Podemos ver um exemplo do seu uso no código abaixo:

```
<!-- A definição do bean fonteDadosInstancia fica no
arquivo spring-factory.xml -->

<import resource="spring-factory.xml"/>

<bean id="impressor" class="br.com.casadocodigo.ImpressorImpl" />

<bean id="processador" class="br.com.casadocodigo.Processador">
    <property name="fonteDados" ref="fonteDadosInstancia"/>
    <property name="impressor" ref="impressor"/>
</bean>
```

3.13 APLICANDO HERANÇA NA DEFINIÇÃO DOS BEANS

Para finalizar, podemos reduzir a quantidade de texto a ser digitado em nossos arquivos de configuração aplicando o conceito de herança. A definição de um bean pode ser baseada em outra. Sendo assim, em situações em que dois beans são bastante parecidos, o grosso da configuração é definida em um destes e o segundo apenas *herda* estas, sobrescrevendo as diferenças. Para obter este resultado usamos o atributo *parent* da tag `<bean>`, que deve ser preenchido com o nome do bean pai. No código abaixo podemos ver um exemplo da aplicação deste recurso.

```
<bean id="impressor" class="br.com.casadocodigo.Impressor"/>

<bean id="fonteDadosArquivo"
      class="br.com.casadocodigo.FonteDadosArquivo">
    <property name="arquivo" value="/caminho/para/Arquivo"/>
</bean>

<bean id="fonteDadosRandomico"
      class="br.com.casadocodigo.FonteDadosRandomico"/>

<bean id="processador" class="br.com.casadocodigo.Processador">
    <property name="fonteDados" ref="fonteDadosArquivo"/>
```

```
<property name="impressor" ref="impressor"/>
</bean>
<!-- Aplicando herança -->
<bean id="processadorFilho" parent="processador">
    <property name="fonteDados" ref="fonteDadosRandomico"/>
</bean>
```

O bean *processadorFilho* possui a mesma classe e a mesma dependência ao bean *impressor*. A única diferença está na versão da fonte de dados a ser usada. Enquanto *processador* usa uma baseada em arquivos, *processadorFilho* adota uma baseada em dados randômica.

3.14 SPRING EXPRESSION LANGUAGE (SpEL)

Até este momento nossa configuração é estática. Todos os valores de nossos atributos e dependências são conhecidos apenas em tempo de desenvolvimento. Sabemos que o mundo não é tão simples assim, e em não raras situações precisamos adequar o funcionamento do nosso software às condições do ambiente em que este é executado. Esta limitação foi a motivação por trás da criação da Spring Expression Language (SpEL) que, como veremos nesta seção, adiciona dinamismo às nossas configurações.

Introduzida na versão 3.0 do framework, SpEL nos permite injetar valores tanto por setter quanto construtores em nossos beans com base expressões que são executadas em tempo de execução. Trata-se de uma linguagem baseada em expressões muito parecida com a Expression Language que vemos no JSP (foi inspirada nesta aliás), de tal maneira que, assim como esta, fornece ao desenvolvedor uma curva de aprendizado muito pouco acentuada.

Sintaxe básica

Como já era de se esperar, a sintaxe da SpEL é bastante similar à que encontramos na EL, sempre seguindo a forma abaixo:

```
<property name="atributo" value="#{expressão a ser calculada}"/>
```

O conteúdo que estiver entre `#{` e `}` sempre deve retornar um valor que pode ser na sua forma um literal. No exemplo abaixo definimos o atributo *maxNumArquivos* do bean *fonteDados* com um literal inteiro simples.


```
<bean id="fonteDadosArquivos"
      class="br.com.casadocodigo.FonteDadosListaArquivos">
  <property name="maxNumArquivos" value="#{5}"/>
</bean>
```

Obteríamos o mesmo resultado sem uma expressão, simplesmente digitando "5" no atributo *value*. Este exemplo surgiu apenas para expor como declarar um literal simples. Abaixo há alguns exemplos de como declarar outros literais com a SpEL:

```
<!-- Incluindo um valor inteiro -->
#{34}
<!-- Incluindo um valor de ponto flutuante -->
#{3.14}
<!-- Usando uma string como literal -->
#{'texto'}
<!-- Incluindo uma string com aspas duplas
quando usamos aspas simples no xml -->
value='#{"texto"}'
<!-- Incluindo um valor booleano -->
#{true}
```

Operações básicas e comparação

Usar literais isolados não trás vantagem alguma ao uso do Spring sem o SpEL. O recurso começa a ficar mais interessante quando começamos a aplicar operações e comparações em nossas expressões.

Todas as operações matemáticas que usamos em Java estão disponíveis na SpEL: soma (+), subtração (-), multiplicação (*), divisão (/), módulo (%) e, como brinde, um novo operador para potenciação (^). Abaixo podemos ver alguns exemplos de aplicação de aritmética no nosso bean *fonteDadosArquivos*.

```
<bean id="fonteDadosArquivos"
      class="br.com.casadocodigo.FonteDadosListaArquivos">
  <!-- Aplicando multiplicação -->
  <property name="maxNumArquivos" value="#{5 * 3}"/>
</bean>
```

Há também operadores relacionais que retornam valores booleanos: menor que (lt), maior que (gt), igual (eq), diferente (ne), maior ou igual (ge) e menor ou igual (le). Abaixo podemos ver alguns exemplos de sua aplicação:

```
<!-- Retorna true -->
#{5 gt 3}
<!-- Retorna false -->
#{5 eq 3}
```

Como toda linguagem de expressão, operadores lógicos também existem: and, or e not, cuja aplicação podemos ver abaixo:

```
<!-- Retorna true -->
#{(5 gt 3) and (5 lt 8)}
<!-- Retorna false -->
#{not true}
#{not ((5 gt 3) and (5 lt 8))}
```

Outro operador importante é o ternário condicional, que funciona exatamente como o que estamos acostumados a usar em Java. Para os novatos, sua sintaxe é bastante simples:

(expressão) ? (valor caso verdadeiro) : (valor caso falso)

Podemos ver um exemplo da aplicação deste operador no código abaixo:

```
<bean id="fonteDadosArquivos"
      class="br.com.casadocodigo.FonteDadosListaArquivos">
  <!-- O valor de maxNumArquivos será igual a 5 -->
  <property name="maxNumArquivos" value="#{5 > 4 ? 5 : 3}"/>
</bean>
```

Referenciando outros beans

Os exemplos do SpEL expostos até agora são inúteis. O poder da linguagem surge a partir do momento em que passamos a acessar atributos presentes em outros beans. Imagine que tenhamos um segundo bean chamado *fonteDadosArquivosCopia* que use as propriedades definidas no bean *fonteDadosArquivo* tal como no código abaixo:

```
<bean id="fonteDadosListaArquivos"
      class="br.com.casadocodigo.FonteDadosListaArquivos">
  <property name="arquivos">
    <list>
      <bean class="java.io.File">
        <constructor-arg value="/opt/arquivos/arquivo1.csv"/>
      </bean>
```

```

        <bean class="java.io.File">
            <constructor-arg value="/opt/arquivos/arquivo2.csv"/>
        </bean>
        <bean class="java.io.File">
            <constructor-arg value="/opt/arquivos/arquivo3.csv"/>
        </bean>
        <bean class="java.io.File">
            <constructor-arg value="/opt/arquivos/arquivo4.csv"/>
        </bean>
    </list>
</property>
</bean>

<bean id="fonteDadosArquivosCopia"
      class="br.com.casadocodigo.FonteDadosListaArquivos">
    <property name="arquivos"
      value="#{fonteDadosListaArquivos.arquivos}"/>
</bean>

```

A lista de arquivos de *fonteDadosArquivosCopia* será a mesma que definimos no bean *fonteDadosArquivo*. A sintaxe usada para acessar outros beans e seus respectivos atributos é igualmente simples:

(identificador do bean).(nome do atributo/propriedade)

Na realidade, o nome do atributo ou propriedade é opcional. Podemos referir apenas o bean se quisermos, substituindo assim o atributo *ref* se quisermos, como no seguinte exemplo:

```
<property name="impressor" value="#{impressor}"/>
```

Também é possível usar como valor o resultado da invocação de um método. Poderíamos definir o atributo *numMaximoArquivos* do bean *fonteDadosArquivosCopia* da seguinte maneira:

```

<bean id="fonteDadosArquivosCopia"
      class="br.com.casadocodigo.FonteDadosListaArquivos">
    <property name="arquivos"
      value="#{fonteDadosListaArquivos.arquivos}"/>
    <property name="numMaximoArquivos"
      value="#{fonteDadosListaArquivos.arquivos?.size()}/>
</bean>

```

O valor do atributo será igual ao número de itens presentes no atributo *arquivos* do bean *fonteDadosListaArquivos*. Há uma novidade neste ponto: o operador de proteção de nulidade `?`. Caso o atributo *arquivos* de *fonteDadosListaArquivos* fosse nulo, não teríamos como resultado uma exceção do tipo *NullPointerException*.

Pesquisando listas

Um recurso avançado da SpEL é o processamento de listas. É possível com esta linguagem retornar listas baseando-se em consultas feitas em outras listas. A melhor maneira de entender esta possibilidade é com exemplos, sendo assim, segue abaixo um bom exemplo deste tipo de aplicação:

```
<bean id="fonteDadosArquivosGrandes"
      class="br.com.casadocodigo.FonteDadosListaArquivos">
  <property name="arquivos"
    value=
      "#{fonteDadosListaArquivos.arquivos.[exists() and size() ge 1024]}" />
</bean>
```

O bean *fonteDadosArquivosGrandes* receberá uma nova lista, cujo conteúdo será todos os arquivos definidos no atributo *arquivos* de *fonteDadosListaArquivos* que existam e, ao mesmo tempo, possuam tamanho maior ou igual a 1024 bytes.

Propriedades do ambiente de execução

Outro uso poderoso da SpEL é a possibilidade de obtermos informações a respeito do nosso ambiente de execução. Para tal usamos o objeto *systemProperties* que sempre está implicitamente disponível a toda expressão, que corresponde a uma chamada ao método *getProperties()* da classe *java.lang.System*. Sendo assim, se fosse nosso objetivo definir uma fonte de dados que sempre buscasse um arquivo dentro do diretório *home* do usuário corrente poderíamos definir nosso bean tal como no exemplo abaixo:

```
<bean id="fonteDadosArquivoUsuario"
      class="br.com.casadocodigo.FonteDadosArquivo">
  <property name="arquivo"
    value="#{systemProperties['user.home']}/arquivo.csv" />
</bean>
```

Como pode ser observado neste exemplo, a SpEL nos permite inclusive concatenar strings. Neste caso, o valor da variável de ambiente *user.home* será con-

catenado ao restante do texto `"/arquivo.csv"`. *O interessante é que a mesma sintaxe que aplicamos ao objeto `systemProperties` pode ser aplicado a qualquer objeto do tipo `java.util.Properties`:*

3.15 RESUMINDO

Este foi um capítulo bastante denso em que pudemos ver a maior parte do que está por trás do mecanismo de inversão de controle e injeção de dependências do Spring. Para tal apresentamos o formato de configuração mais usado do Spring que é o XML e, com ele, vimos o básico desde como declarar um bean, os modos que temos de instanciá-los, como injetar dependências e valores de atributos até maneiras de tirarmos proveito tanto do ciclo de vida do container como do próprio bean. Para finalizar, demos dinamismo às nossas configurações apresentando a SpEL, uma linguagem baseada em expressões incluída no Spring 3.0 que, desde então, permeia todo o restante do framework, incluindo configurações baseadas em anotações.

Por falar em anotações, o leitor deve estar preocupado com a quantidade de XML que digitamos neste capítulo. Espero ter provado o quão poderoso é este formato (que ainda não foi completamente explorado neste capítulo), porém devemos nos atentar ao fato de que todo este digitar de configurações não é a maneira mais produtiva que temos de trabalhar. Sendo assim, preparo-o para o próximo capítulo, aonde veremos como minimizar ao máximo a necessidade de digitarmos configurações ao nosso container aumentando significativamente a produtividade oferecida pelo framework.

CAPÍTULO 4

Minimizando o XML com autowiring, anotações e Java

Neste ponto já conhecemos bastante a respeito do modo como a inversão de controle/injeção de dependências é implementada no Spring Framework. Até então só usamos o formato mais popular de se configurar o container, que é o XML. Chegou a hora de aumentar a nossa produtividade minimizando o trabalho manual. Vamos ver outras formas de configuração oferecidos pelo framework, mas vale lembrar que você encontrará a configuração via XML em diversos projetos e até mesmo como principal referência na documentação.

4.1 AUTOWIRING: AUTOMATIZANDO A INJEÇÃO DE DEPENDÊNCIAS

Já nas primeiras versões do Spring ficou claro que o formato XML pode facilmente se tornar um gargalo na produtividade do desenvolvedor. A primeira solução encon-

trada para o problema foi a inclusão da injeção automática de dependências, mais conhecida como *autowiring*. Este recurso possibilita ao container descobrir em tempo de execução quais as dependências que devem ser injetadas em cada bean SEM que o desenvolvedor precise instruí-lo para tal.

Para melhor entender seu funcionamento vamos apresentar um exemplo mais elaborado que os anteriores. Trata-se de um sistema de gestão comercial composto por três *DAOs*, todos derivados da classe *AbstractDAO* que, por sua vez, possui como dependência um objeto do tipo *DataSource* que representa uma fonte de dados qualquer.

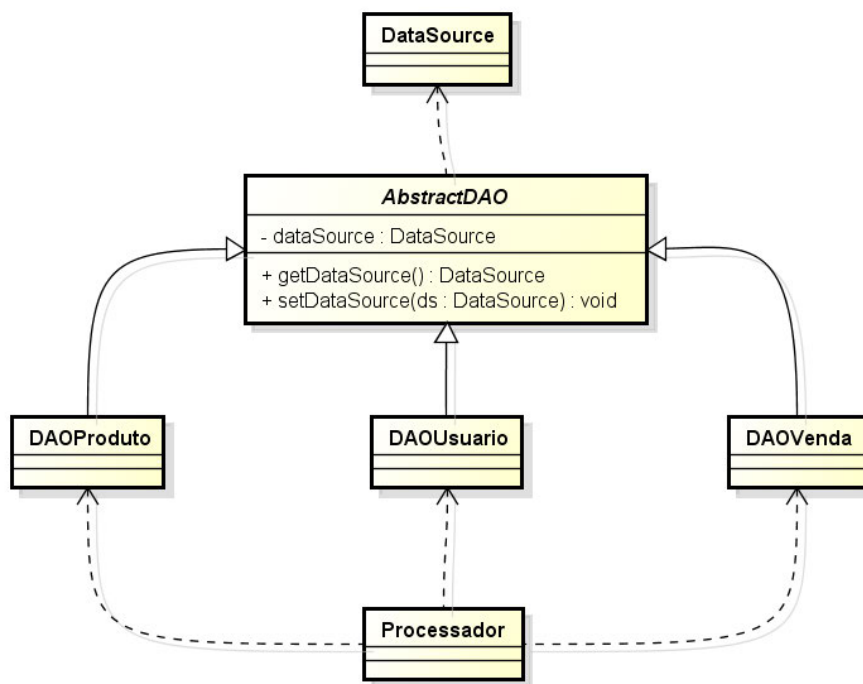


Figura 4.1: Visão geral do nosso sistema

Com base no que já sabemos, a primeira versão do nosso arquivo de configuração poderia ser similar à descrita abaixo:

```
<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.xsd">

<bean id="dataSource" class="br.com.casadocodigo.DataSource">
    <property name="connectionString"
        value="jdbc:mysql://localhost/db"/>
</bean>

<bean id="daoProduto" class="br.com.casadocodigo.DAOProduto">
    <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="daoUsuario" class="br.com.casadocodigo.DAOUsuario">
    <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="daoVenda" class="br.com.casadocodigo.DAOVenda">
    <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="processador" class="br.com.casadocodigo.Processador">
    <property name="daoProduto" ref="daoProduto"/>
    <property name="daoUsuario" ref="daoUsuario"/>
    <property name="daoVenda" ref="daoVenda"/>
</bean>
</beans>
```

Observando esta configuração fica mais claro como o formato XML pode prejudicar nossa produtividade. A quantidade de código repetido é considerável. Note que o bean *dataSource*, compartilhado por todos os DAOs do sistema foi injetada manualmente todas as vezes que foi necessária. Dado que a plataforma Java possui um mecanismo de reflexão bastante poderoso e há apenas um bean de cada tipo necessário para o funcionamento do sistema, não é difícil imaginar um algoritmo que descubra automaticamente quais dependências precisam ser satisfeitas com base apenas nos tipos de beans declarados na configuração.

Este recurso se chama *autowiring*. Aplicando-o na configuração acima, obtemos um arquivo de configuração significativamente menor, como pode ser observado no código abaixo:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans default-autowire="byType"
```



```
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.xsd">

<bean class="br.com.casadocodigo.DataSource">
    <property name="connectionString"
        value="jdbc:mysql://localhost/db"/>
</bean>

<bean class="br.com.casadocodigo.DAOProduto"/>
<bean class="br.com.casadocodigo.DAOUsuario"/>
<bean class="br.com.casadocodigo.DAOVenda"/>
<bean class="br.com.casadocodigo.Processador"/>
</beans>
```

Na nova configuração foi necessário digitar apenas 7 linhas descrevendo nossos beans ao invés das 17 escritas anteriormente. Outro ponto interessante: dado que o *container* descobrirá quais dependências precisam ser satisfeitas, não precisamos sequer definir nomes de nossos beans.

Para que o container aplique o *autowiring* podemos adotar dois caminhos. O primeiro e menos trabalhoso é instruí-lo a aplicar esta técnica a todos os beans que se encontrem declarados no mesmo arquivo. Como fizemos na última versão de nossa configuração, basta adicionar o atributo `default-autowire` à tag `<beans>`. É também possível aplicar o autowiring apenas a alguns beans definidos no mesmo arquivo. Como já era de se esperar, basta acrescentar o mesmo atributo à tag `<bean>` que representa a definição de bean em cima da qual a injeção automática deva ser aplicada. Independente do local aonde o atributo seja incluído, este sempre pode receber um dos valores descritos abaixo, que descrevem qual a estratégia a ser adotada pelo container:

- `no`: valor default adotado pelo container. Autowiring não será adotado.
- `byType`: uma dependência será injetada automaticamente caso seja encontrada apenas uma definição do bean que possua tipo compatível.
- `byName`: o Spring irá injetar a dependência com base no nome. Serão buscados beans que possuam o mesmo nome da propriedade a ser preenchida.
- `constructor`: similar ao algoritmo `byType`, com a diferença de que buscará por construtores não padrão para executar a injeção ao invés de propriedades.

- **autodetect:** o container irá inicialmente tentar a injeção do tipo constructor. Caso não seja encontrado um construtor no bean a ser injetado, procederá para a estratégia `byType`.

Evitando ambiguidades

Para que a injeção automática funcione é fundamental que o container não tenha dúvidas a respeito de qual dependência injetar. Expondo melhor o problema, vamos complicar nosso exemplo. Suponha que o bean `DAOUsuario` precise buscar seus dados de um banco de dados legado. Ingenuamente alteramos nossa configuração para que fique tal como no exemplo abaixo:

```
<beans default-autowire="byType"
  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.xsd">

  <bean class="br.com.casadocodigo.DataSource">
    <property name="connectionString"
      value="jdbc:mysql://localhost/db"/>
  </bean>

  <bean id="dataSourceUsuarios"
    class="br.com.casadocodigo.DataSource">
    <property name="connectionString"
      value="jdbc:mysql://localhost/users"/>
  </bean>

  <bean id="daoUsuario" class="br.com.casadocodigo.DAOUsuario">
    <property name="dataSource" ref="dataSourceUsuarios"/>
  </bean>

  <bean class="br.com.casadocodigo.DAOProduto"/>
  <bean class="br.com.casadocodigo.DAOVenda"/>
  <bean class="br.com.casadocodigo.Processador"/>
</beans>
```

Observe que a dependência foi definida explicitamente para o bean `daoUsuario` mas não para os demais DAOs. Como consequência, quando o container for iniciado não saberá qual fonte de dados injetar nos beans do tipo `DAOVenda`

e DAOProduto. O resultado será o levantamento de uma exceção do tipo `org.springframework.beans.factory.UnsatisfiedDependencyException`.

Há duas soluções para o problema: a primeira é definir manualmente a fonte de dados de cada DAO em nosso sistema. A solução mais inteligente é instruir o container a respeito de quais beans **não são** candidatos a serem injetados automaticamente. Resolvemos este problema adicionando o atributo `autowire-candidate` com o valor `false` no bean a ser excluído. Em nosso exemplo, bastaria alterar a definição do bean `dataSourceUsuarios` tal como no exemplo abaixo:

```
<bean id="dataSourceUsuarios" class="br.com.casadocodigo.DataSource"
      autowire-candidate="false">
  <property name="connectionString"
    value="jdbc:mysql://localhost/users"/>
</bean>
<bean id="daoUsuario" class="br.com.casadocodigo.DAOUsuario">
  <property name="dataSource" ref="dataSourceUsuarios"/>
</bean>
```

Há um detalhe interessante a ser observado na configuração acima: o bean `daoUsuario` define explicitamente qual bean deve ser injetado em sua propriedade `dataSource`. A injeção de dependências automática é anulada em situações como esta, em que a configuração seja feita explicitamente.

4.2 VANTAGENS E LIMITAÇÕES DA INJEÇÃO AUTOMÁTICA

A vantagem óbvia por trás do *autowiring* é a redução da configuração. O ganho oculto é a evolução natural e automática de nossa configuração em tempo de desenvolvimento, visto que o programador não precisa atualizar constantemente seus arquivos XML conforme progride no processo de criação do seu código.

A principal limitação da técnica é a substituição do *explícito* pelo *implícito*. Até então sempre tínhamos certeza de como seria feita a injeção de dependências em nossos beans pois o fazíamos manualmente. A partir do momento em que adotamos o *autowiring* delegamos esta responsabilidade ao container. Em projetos nos quais o número de beans seja pequeno e o número de candidatos a serem injetados seja o aceitável (1) o ganho de produtividade é garantido, mas conforme o projeto evolui e o número de candidatos aumenta o desenvolvedor se verá constantemente precisando excluir um ou outro candidato e, ainda pior, acabará voltando para o modo manual de configuração. Na esmagadora maioria dos casos os erros serão de-

tectados em tempo de desenvolvimento, pois o container lançará uma exceção do tipo `org.springframework.beans.factory.UnsatisfiedDependencyException`. É importante salientar que há a possibilidade de erros sutis, em tempo de execução, decorrentes da injeção equivocada de um bean ou outro pelo container.

Outra limitação do *autowiring* é o fato deste não injetar automaticamente propriedades do tipo primitivo, listas ou strings. Nestes casos o desenvolvedor deve definir seus valores explicitamente sempre, mesmo porque não há como o container adivinhar seus valores.

A regra de ouro na adoção do *autowiring* é usá-lo apenas em projetos pequenos ou, no caso de projetos maiores, naqueles em que sejam definidas convenções rígidas bem divulgadas e adotadas por toda a equipe de desenvolvimento.

4.3 FACILITANDO AINDA MAIS COM ANOTAÇÕES

A versão 2.0 do Spring trouxe um reforço importante para a tarefa de redução do XML sob a forma de configurações baseadas em anotações Java. Com a adição deste recurso os desenvolvedores ganharam outro impulso em sua produtividade reduzindo a quase nada a necessidade de precisarmos escrever configuração no formato XML. Trata-se de uma ferramenta poderosa que serviu inclusive como uma das inspirações para a JSR-330 que define o padrão de injeção de dependências adotado como padrão pela plataforma Java EE a partir de sua versão 5.

Ao invés de simplesmente competir com esta JSR, a SpringSource optou por abraçá-la, tornando seu container compatível com a mesma ao mesmo tempo que possibilitando ao desenvolvedor tirar proveito das capacidades únicas do Spring. Como veremos, há também suporte à JSR-250 (*Common Annotations for the Java Platform*), que forma uma das bases do EJB 3.0.

A grosso modo temos aqui uma evolução da injeção automática por tipo com a transferência de nossa configuração dos arquivos XML para um contexto mais próximo de seu uso, que é a classe onde devem ser injetadas suas dependências.

Definindo pontos de injeção

Ironicamente, o primeiro passo na redução do XML nesta seção será incrementando-o com o namespace context. Para tal, modificaremos a última versão do nosso arquivo de configuração para que fique tal como no exemplo abaixo:

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

<context:annotation-config/>

<bean id="dataSource" class="br.com.casadocodigo.DataSource">
    <property name="connectionString"
        value="jdbc:mysql://localhost/db"/>
</bean>

<bean id="daoProduto" class="br.com.casadocodigo.DAOProduto"/>
<bean id="daoUsuario" class="br.com.casadocodigo.DAOUsuario" />
<bean id="daoVenda" class="br.com.casadocodigo.DAOVenda"/>
<bean id="processador" class="br.com.casadocodigo.Processador"/>
</beans>

```

Repare que o atributo `default-autowire` foi removido da tag `<beans>`, criando a falsa impressão de que estamos desabilitando a injeção automática. Na realidade estamos apenas modificando sua forma. A tag `<context:annotation-config/>` informa o container de que nossa configuração será definida por anotações.

Neste momento a novidade é a forma como definiremos os pontos de injeção de dependências. Ao invés do XML convencional, usaremos agora duas anotações fornecidas pelo Spring: `@Required` e `@Autowired`.

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Required;

public abstract class AbstractDAO {

    private DataSource dataSource;
    public DataSource getDataSource() {return dataSource;}

    @Required @Autowired
    public void setDataSource(DataSource ds) {dataSource = ds;}
}

```

A anotação `@Required` (opcional) só pode ser aplicada em métodos e setters. Possui a função de instruir o container a respeito da obrigatoriedade.

idade de uma dependência, forçando-o a disparar uma exceção do tipo `org.springframework.beans.factory.UnsatisfiedDependencyException` caso não seja encontrado um candidato que a satisfaça.

Já a anotação `@Autowired`, como o próprio nome diz, indica um ponto aonde a injeção automática deve ser aplicada. Esta pode ser usada em métodos, atributos e construtores (importante mencionar que apenas um construtor por classe pode recebê-la). Caso opte por usá-la, é possível descartar o uso da anotação `@Required`, tal como no exemplo abaixo:

```
@Autowired(required=true)
public void setDataSource(DataSource ds) {dataSource = ds;}
```

Como mencionado no início desta seção, o Spring também oferece suporte às anotações da JSR-330. A principal vantagem em seu uso é o fato do código fonte não ter dependência direta ao Spring e sim classes que sempre serão providas pela plataforma Java EE.

A anotação mais popular neste caso é `@Inject`. Como por padrão seu código fonte não vem com o Spring, é necessário acrescentar a biblioteca necessária ao classpath do seu projeto. Caso este seja baseado em Maven, basta adicionar a dependência exposta abaixo ou então fazer o download manual do arquivo jar em <http://code.google.com/p/atinject/>.

```
<dependency>
  <groupId>javax.inject</groupId>
  <artifactId>javax.inject</artifactId>
  <version>1</version>
</dependency>
```

Assim como `@Autowired`, `@Inject` pode ser usada em métodos, atributos ou construtores (mantendo a limitação de só poder anotar um construtor por classe). Comparada à anotação `@Autowired`, possui a desvantagem de não possuir o atributo `required`.

```
@Inject
public void setDataSource(DataSource ds) {dataSource = ds;}
}
```

Outra opção é a anotação `@Resource` introduzida na JSR-250. Como era de se esperar, seu uso é praticamente idêntico ao que encontramos nas anotações expostas anteriormente. Um outro ponto importante é a possibilidade de usar as três anotações dentro de uma mesma classe, como pode ser visto no exemplo abaixo:

```
public class Processador {

    @Resource
    public void setDaoProduto(DAOProduto daoProduto) {
        this.daoProduto = daoProduto;    }

    @Autowired
    public void setDaoUsuario(DAOUsuario daoUsuario) {
        this.daoUsuario = daoUsuario;    }

    @Inject
    private DAOVenda daoVenda;

    // Outros métodos do processador
}
```

Encontrando os beans

Até este momento a declaração dos beans ainda é feita diretamente em nosso documento XML. A única diferença está no fato de termos movido as definições de injeção para nosso código fonte. A real redução do XML só ocorre quando instruímos o container a buscar as definições de nossos beans varrendo o classpath do nosso sistema em busca de classes devidamente anotadas, que possam ser reconhecidas como beans.

Modificaremos mais uma vez nosso arquivo de configuração de tal modo que este tenha no final apenas duas linhas significativas. Instruiremos o container a buscar nossas definições de beans varrendo o classpath inserindo a tag `<context:component-scan/>` em nosso arquivo XML como pode ser visto no exemplo abaixo:

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

    <context:annotation-config/>
    <context:component-scan base-package="br.com.casadocodigo"/>
```

```
</beans>
```

A tag `<component-scan>` possui o parâmetro `base-package` obrigatório que recebe como valor o nome do pacote que contém todas as nossas classes que implementam os beans de nosso sistema. Entra em cena o conceito de estereótipo. Uma classe é reconhecida pelo Spring como uma configuração de bean se possui uma anotação que defina o tipo de bean (estereótipo) que esta representa.

O tipo mais básico de componente é identificado no Spring pela anotação `@Component`. Para identificar uma classe como um componente, basta adicionar esta anotação como pode ser visto no código abaixo:

```
@Component("processador")
public class Processador {
    ...
}
```

Como pôde ser visto, ainda com anotações é possível nomear nossos beans passando como parâmetro sua identificação. Caso este valor esteja ausente, o nome do bean será definido pelo container automaticamente. Além do estereótipo `@Component` o Spring também oferece uma série de outras opções como por exemplo `@Repository` para identificar *DAOs*, `@Controller` para controladores do Spring MVC, `@Service` para controle transacional mais fino e muitos outros. A existência de múltiplos estereótipos permite à equipe de desenvolvimento implementar comportamentos distintos para cada tipo de bean, enriquecendo assim ainda mais o poder da ferramenta.

Além das anotações do Spring usadas para identificação de estereótipos, também podemos usar a anotação `@Named` definida na JSR-330, que possuirá exatamente o mesmo comportamento da anotação `@Component`. Abaixo podemos ver alguns exemplos de sua utilização.

```
import javax.inject.Named;

@Named // Anotando um componente sem nome explícito
public class Processador {}

@Named("processador") // Definindo explicitamente o nome do bean
public class Processador {}
```


Filtrando componentes

Há situações nas quais não é interessante o carregamento de todas as definições de beans presentes em um pacote. Em nosso documento XML quando incluímos apenas a tag `<context:component-scan/>`, estamos instruindo o container a carregar todas as definições de beans contidas no pacote definido pelo atributo `base-package`. Caso seja necessário incluir apenas parte destas definições, entram em ação as tags `<context:include-filter>` e `<context:exclude-filter>`, ambas aninhadas dentro de `<context:component-scan>` e, como o próprio nome já diz, definem respectivamente quais definições de beans devem ser incluídas e ignoradas em nossa configuração.

Tanto `<context:include-filter>` quanto `<context:exclude-filter>` possuem dois parâmetros obrigatórios que são `type` e `expression`. O primeiro identifica o tipo de expressão usada para encontrar os componentes no classpath do sistema enquanto o segundo define a expressão que aplicaremos na filtragem.

Por padrão o Spring oferece os seguintes tipos de expressão:

- `annotation`: no parâmetro `expression` deverá estar presente o nome completo da anotação de identificação de beans (estereótipo) que será incluída no filtro.
- `regex`: passamos uma expressão regular que identifique o tipo de classe que entrará na filtragem.
- `assignable`: define qual a classe base ou interface implementada pelas classes anotadas que deverá ser incluída no filtro.
- `aspectj`: uma expressão no formato AspectJ para identificar os componentes.
- `custom`: é possível implementar filtros customizados na varredura do classpath, que são basicamente classes que implementem a interface `org.springframework.core.type.TypeFilter`.

No código abaixo podemos ver a aplicação de filtros na varredura customizada do nosso classpath:

```
<context:component-scan base-package="br.com.casadocodigo.dao">
  <!--
    Inclui todos os componentes cujo
    nome da classe comece com DAO
  -->
```

```
-->
<context:include-filter type="regex" expression="DAO*"/>
<!-- Exclui todos os componentes do estereótipo Controller -->
<context:exclude-filter type="annotation"
    expression="org.springframework.stereotype.Controller"/>
</context:component-scan>

<context:component-scan base-package="br.com.casadocodigo.dao">
    <!-- Inclui apenas classes com classe mãe DAOBase -->
    <context:include-filter type="assignable"
        expression="br.com.casadocodigo.DAOBase"/>
</context:component-scan>
```

Definindo escopos

Definimos o escopo de um bean usando a anotação `@Scope` provida pelo Spring. Esta recebe como parâmetro o nome do escopo que são os mesmos apresentados no capítulo anterior: `singleton`, `prototype`, `request` ou o nome do seu escopo customizado. No código abaixo podemos ver algumas aplicações desta anotação.

```
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;
...
@Scope("prototype")
@Component
public class DAOUsuario {...}

@Scope("request")
@Component
public class DAOProduto {...}
```

Assim como na configuração XML convencional, caso seja omitido o escopo de um bean será adotado o padrão *singleton*.

Ciclo de vida

É possível definir métodos de callback a serem executados após a injeção e antes da destruição dos beans, tal como aprendemos a fazer no capítulo anterior. Para tal usamos as anotações `@PostConstruct` e `@PreDestroy` definidas na JSR-250. No código abaixo podemos ver um exemplo de aplicação das mesmas:

```
import javax.annotation.PostConstruct; //JSR-250
import javax.annotation.PreDestroy; //JSR-250
import javax.inject.Named; //JSR-330

@Named
public class DAOProduto extends AbstractDAO {
    @PostConstruct
    public void init() {
        System.out.println("\n\nDAO Produto iniciado\n\n\n");
    }

    @PreDestroy
    public void destroy() {
        System.out.println("\n\nDAO Produto Finalizado\n\n\n");
    }
}
```

É importante mencionar que o desenvolvedor continua com a opção de usar as interfaces `org.springframework.beans.factory.InitializingBean` ou `org.springframework.beans.factory.DisposableBean` pra obter o mesmo resultado.

Limitando o número de candidatos

Dado que as anotações são uma evolução do conceito de *autowiring* apresentado na seção anterior, o problema da ambiguidade de configuração precisa ser tratado. A solução no nosso caso se dá através da adoção de qualificadores, que servem para fornecer uma melhor identificação aos nossos beans.

Usamos a anotação `@Qualifier` para guiar o container na seleção de qual dependência deverá ser injetada em determinado atributo ou parâmetro de construtor ou método definido em um bean, tal como pode ser visto no código abaixo:

```
import org.springframework.beans.factory.annotation.Qualifier;
import javax.inject.Named("daoUsuario");

public class DAOUsuario {
    @Qualifier("dataSourceUsuario")
    private DataSource dataSource;
}
```

O valor passado à anotação é o identificador que ajuda na escolha de qual dependência deverá ser injetada naquele ponto. Por convenção, seu valor corres-

ponde ao nome do bean a ser injetado. É possível também mesclar qualificadores com configuração XML usando a tag `<qualifier>`. Sendo assim, o qualificador `dataSourceUsuario` exposto acima poderia ser identificado na configuração XML da seguinte forma:

```
<!-- Este bean será injetado em DAOUsuario -->
<bean id="dsUsuario" class="br.com.casadocodigo.DataSource">
    <qualifier value="dataSourceUsuario"/>
</bean>
<!-- Este bean não será injetado em DAOUsuario -->
<bean id="dsGeral" class="br.com.casadocodigo.DataSource">
    <property name="connectionString" value="jdbc:database//meuDB"/>
</bean>
```

A anotação `@Qualifier` é a mais genérica possível. Se for de interesse do desenvolvedor, é possível evoluir seu código semanticamente criando sua própria anotação qualificadora. Para tal, basta criar sua própria anotação aplicando-lhe a anotação `@Qualifier` tal como em:

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface TipoDataSource {
    String valor();
}
```

Em seguida, basta usar sua anotação customizada ao invés da padrão para limitar o número de candidatos a serem injetados em seus beans. A anotação qualificadora também pode ser aplicada a parâmetros de construtores ou métodos, tal como no exemplo abaixo:

```
public void setDataSource(@TipoDataSource("usuarios") DataSource ds) {
    this.dataSource = ds;
}
```

Vantagens e limitações do uso de anotações

Há um debate sem fim na comunidade a respeito da superioridade das anotações em relação ao XML. A principal vantagem das anotações está no fato destas serem incluídas diretamente nas classes, mais próximas portanto do contexto do desenvolvedor do que um arquivo separado. Outro benefício, como já dito, é o fato de reduzir

significativamente o tamanho do nosso arquivo de configuração que precisaríamos escrever sem o recurso.

Entretanto, é importante salientar algumas limitações. A principal vantagem do formato XML é o fato deste possibilitar alterar o funcionamento do nosso sistema sem a necessidade de recompilarmos o código fonte. Em situações em que a customização seja um aspecto crítico, como por exemplo no desenvolvimento de produtos de prateleira, a adoção das anotações podem acarretar mais perdas que ganhos. Apesar disto, é importante mencionar que qualquer configuração feita usando anotações pode ser sobrescrita em um arquivo XML. Por exemplo, é possível declarar um bean específico em XML e este será levado em consideração ao invés do mesmo definido usando anotações.

Outro ponto importante a ser levantado: a redução da configuração muitas vezes é ilusória quando usamos anotações, pois na prática na maior parte das vezes o que realmente estamos fazendo é transferir informações do formato XML para dentro de nossas classes.

4.4 CONFIGURAÇÃO PROGRAMÁTICA COM JAVA

A versão 3.0 do Spring trouxe como novidade a possibilidade de configurarmos nosso container com código Java, oferecendo possibilidades interessantes ao desenvolvedor como por exemplo configuração programática e, finalmente, a eliminação total da configuração XML.

Os modos de configuração até então apresentados são declarativos. O desenvolvedor descreve ao container o que instanciar e este se encarrega do trabalho de injeção de dependências e inicialização dos beans. Com a configuração baseada em código Java o programador passa a agir imperativamente sobre o ciclo de vida dos beans. Com base no que aprendemos até este momento, passo para o leitor um desafio: usando XML ou anotações, como definir qual tipo de classe instanciar baseado em algum fator de execução do sistema como por exemplo a hora corrente? Como veremos, este tipo de problema é trivialmente resolvido usando configuração baseada em código.

@Configuration e @Bean

Nos formatos de configuração apresentados anteriormente o ponto de partida sempre era um arquivo no formato XML. Com a configuração programática nosso ponto de partida passa a ser uma ou mais classes que possuam a anotação

@Configuration que, por baixo dos panos, consiste em mais um *estereótipo* (stereotype) oferecido pelo Spring.

Toda classe anotada com @Configuration é entendida pelo container como um bean responsável por armazenar uma ou mais definições de beans. Dado que estamos trabalhando com um estereótipo, seu uso é exatamente igual ao que já vimos na seção anterior como pode ser visto no código abaixo:

```
import org.springframework.context.annotation.Configuration;
// Sem definir um nome explicitamente
@Configuration
public class JavaConfig {...}

// Definindo um nome explicitamente
@Configuration("configuracaoJava")
public class JavaConfig {...}
```

Se @Configuration define um **repositório de beans**, em seu interior devemos incluir a anotação @Bean, que deve ser inserida apenas em funções equivale à tag <bean> que usamos na configuração no formato XML. No código abaixo podemos ver alguns exemplos de seu uso:

```
@Configuration
public class JavaConfig {
    @Bean
    public DAOUsuario daoUsuario() {
        return new DAOUsuario();
    }

    // Definindo mais de um nome para o mesmo bean
    @Bean(name={"daoProduto", "produtoDAO"})
    public DAOProduto criarDAOProduto() {
        return new DAOProduto();
    }

    // Definindo o nome do bean explicitamente
    @Bean(name="datasource")
    public DataSource getDataSource() {
        DataSource ds = null;
        GregorianCalendar data = new GregorianCalendar();
        if (data.get(GregorianCalendar.HOUR_OF_DAY) < 13) {
            ds = new DataSourceUsuarios();
        }
    }
}
```

```

    } else {
        ds = new DataSource();
    }
    return ds;
}
}

```

Vemos no exemplo anterior três usos da anotação `@Bean`. No primeiro caso, aplicamos a anotação sobre um método sem definir explicitamente o nome do bean. Nesta situação o nome do bean corresponderá ao nome do método que recebeu a anotação. Assim como na configuração XML, também é possível definir múltiplos nomes para um bean (*aliases*), que é o que fazemos no segundo caso da anotação, passando como parâmetro um array de strings contendo estes nomes.

Como pode ser observado, tudo o que os dois primeiros métodos do nosso código anterior simplesmente criam uma nova instância do bean. O uso realmente interessante pode ser observado no terceiro caso em que usamos a hora corrente do sistema para escolher qual implementação deve ser instanciada.

Finalmente, dado que `@Configuration` na realidade é um estereótipo, isto é, uma variação de `@Component`, poderíamos informar o container via XML a usar configurações baseadas em anotações com um arquivo similar ao exposto abaixo, sabendo que no pacote `br.com.casadocodigo.java` há apenas a classe `JavaConfig`:

```

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

    <context:component-scan base-package="br.com.casadocodigo.java"/>

</beans>

```

Ciclo de vida e escopo

A anotação `@Bean` possui dois atributos opcionais que nos permitem tirar proveito do ciclo de vida do bean por ela representado. Estes atributos são `initMethod` e `destroyMethod`, que recebem como valor o nome do método a ser executado após

a instanciação e injeção de dependências no bean e antes de sua destruição pelo container. No código abaixo podemos ver um exemplo de seu uso:

```
@Bean(initMethod="iniciar", destroyMethod="destruir")
public DAOUsuario getDaoUsuario() {
    return new DAOUsuario();
}
```

Também podemos definir o escopo de um bean. Neste caso, adicionamos a anotação `@Scope` provida pelo Spring. No caso, seu uso é exatamente o mesmo que vimos na configuração baseada por anotações, a diferença é que iremos anotar o método responsável pela instanciação do bean.

```
@Scope("prototype")
@Bean(initMethod="iniciar", destroyMethod="destruir")
public DAOUsuario getDaoUsuario() {
    return new DAOUsuario();
}
```

```
@Scope("request")
@Bean
public DAOProduto daoProduto() {
    return new DAOProduto();
}
```

Injetando dependências

Nos exemplos expostos até este momento todos os métodos construtores de beans expostos simplesmente instanciam objetos sem se preocupar com a injeção de dependências. Isto porque na realidade estamos novamente aqui usando a configuração baseada em anotações que, na prática, é o *autowiring* baseado em tipo.

Na configuração baseada em Java, no entanto, é importante observar que, uma vez que o método `@Bean` retorne seu valor, o container se encarrega de injetar todas as dependências deste objeto, o que pode gerar algumas situações para os iniciantes, como por exemplo a situação exposta no código abaixo:

```
@Configuration
public class JavaBuilder {
    @Bean
    public DataSource dataSource() {
        return new DataSource();
    }
}
```



```

    }
    @Bean
    public DAOProduto daoProduto() {
        return new DAOProduto();
    }
    @Bean
    public DAOProduto preConfigurado() {
        DAOProduto dao = new DAOProduto();
        dao.setDataSource(new DataSource());
        return dao;
    }
}

```

A primeira vista podemos ter a impressão de que o bean `preConfigurado` exposto acima recebe como dependência uma instância diferente da definida pelo bean `dataSource`, certo? Errado: uma vez obtida a instância do bean o container irá injetar como dependência o bean retornado pelo método `dataSource()`. A única maneira de se evitar este comportamento consiste em não implementar na classe `DAO-Produto` um método setter para o atributo `dataSource`, injetando esta dependência apenas pelo construtor.

Dado que uma classe anotada com `@Configuration` é em si um bean, esta também pode receber dependências. Sendo assim, poderíamos injetar uma dependência nesta e em seguida reaproveitá-la nas definições de beans que esta contenha, tal como no exemplo abaixo:

```

@Configuration
public class JavaBuilderDependencia {

    @Autowired
    private DataSource dataSource;

    @Bean
    public DAOProduto daoProduto() {
        DAOProduto daoProduto = new DAOProduto();
        daoProduto.setDataSource(dataSource);
        return daoProduto;
    }
}

```

Excluindo a configuração XML

Finalmente, como mencionado no início desta seção, veremos como tornar o Spring 100% livre da configuração XML. Para tal iremos usar uma implementação do `ApplicationContext` chamada `AnnotationConfigApplicationContext`, presente no pacote `org.springframework.context.annotation`. Ao contrário de `ClassPathXmlApplicationContext` e `FileSystemXmlApplicationContext`, esta classe recebe como parâmetro uma ou mais classes que contenham definições de beans, ou seja, aquelas que possuem uma anotação que as identifiquem como implementadoras de algum estereótipo reconhecido pelo Spring.

Há apenas duas novidades neste tipo de container, seus construtores e o método `register`. É possível instanciar este contexto de aplicação de qualquer uma das formas expostas no código abaixo:

```
import org.springframework.context.annotation.  
    AnnotationConfigApplicationContext;  
...  
  
AnnotationConfigApplicationContext context;  
  
// Pelo construtor padrão  
context = new AnnotationConfigApplicationContext();  
  
/*  
Passando apenas uma classe contendo definições de beans @Configuration  
ou alguma classe que possua um estereótipo anotado  
*/  
context = new AnnotationConfigApplicationContext(JavaBuilder.class);  
  
// Passando uma lista de classes anotadas  
context = new AnnotationConfigApplicationContext(  
    JavaBuilder.class, DataSource.class);  
  
// Passando o nome de um pacote base aonde buscar definições  
context = new AnnotationConfigApplicationContext("br.com.casadocodigo");  
  
// Passando o nome de mais de um pacote aonde buscar definições  
context = new AnnotationConfigApplicationContext(  
    "br.com.casadocodigo.java", "br.com.casadocodigo.daos");
```

É interessante observar que, ao contrário da configuração baseada em anotações

com XML em que podemos usar filtros para definir o que incluir e o que não incluir em nosso container, não possuímos este recurso usando apenas os construtores fornecidos por esta classe. Para que possamos possuir um controle mais fino sobre quais definições incluir usamos o método `register`, que recebe como parâmetro uma lista de classes anotadas a serem parseadas pelo container. É um método bastante útil quando queremos aplicar alguma lógica em cima de quais beans queremos em nosso container com base em condições externas à nossa aplicação.

Um bom uso deste método pode ser visto no seguinte exemplo:

```
AnnotationConfigApplicationContext context =
    new AnnotationConfigApplicationContext();
if (condição) {
    context.register(DataSource.class);
} else {
    context.register(DataSourceCustomizado.class);
}
context.register(DAOProduto.class, DAOUsuario.class);
// Atualiza a configuração do container após este ter sido alimentado
// com as classes acima
context.refresh();
```

Após termos invocado quantas vezes forem necessárias o método `register` é importante que seja executado o método `refresh` do container, que irá atualizar suas definições internas de beans.

Para finalizar, outro método interessante deste contexto de aplicação é `scan` que recebe como parâmetro uma lista de pacotes base aonde serão buscadas classes anotadas a serem incluídas em nosso container. Um exemplo do seu uso pode ser visto no código abaixo:

```
AnnotationConfigApplicationContext context =
    new AnnotationConfigApplicationContext();
if (condição) {
    context.scan("br.com.casadocodigo");
} else {
    context.scan("br.com.casadocodigo2", "br.com.dao");
}
// Atualizamos o estado interno do container
context.refresh();
```

Vantagens e limitações da configuração Java

A grande vantagem da configuração baseada em código fonte é a possibilidade do desenvolvedor possuir maior controle sobre o modo como seus beans são instanciados e seus estados internos definidos. Para os que não gostam do formato XML, pode-se dizer que outra grande vantagem é a possibilidade de nos livrarmos completamente deste formato.

É importante observar no entanto que as configurações baseadas em Java não foram feitas para substituir completamente o formato XML, mas sim para enriquecê-lo, e, como toda ferramenta, deve ser usado sem excessos pois possui a grande desvantagem de sempre exigir recompilação de código fonte quando alguma configuração precisa ser alterada, o que torna esta opção mais difícil de ser adotada em ambientes nos quais customização seja uma prioridade alta.

4.5 XML, ANOTAÇÕES OU CONFIGURAÇÃO PROGRAMÁTICA?

Uma das maiores críticas ao Spring Framework sempre foi o fato deste usar longos arquivos de configuração no formato XML que, como vimos neste capítulo, nem sempre é o mais produtivo para se trabalhar. Sendo assim neste capítulo vimos algumas respostas a esta crítica. Para começar, fomos apresentados ao *autowiring* que foi o nosso primeiro passo na redução da configuração. Em seguida, conhecemos a configuração baseada em anotações que reduziu ainda mais o tamanho dos nossos arquivos de configuração transferindo a configuração para classes ao invés de arquivos externos e, finalmente, para excluir completamente os arquivos de configuração, vimos a configuração baseada em código Java que permite ao desenvolvedor aplicar lógica na composição da definição de nossos beans.

Qual das três formas de configuração (XML, anotações e código Java) é a melhor não é uma pergunta factível de ser respondida dado que cada uma apresenta vantagens e desvantagens únicas. Anotações e Java nos obrigam a recompilar nosso código caso ocorram alterações, ao passo que documentos XML conforme evoluem vão se tornando cada vez mais difíceis de serem mantidos. No final das contas, o que deve ser levado em consideração não é o uso individual de cada um destes formatos, mas sim o uso combinado dos mesmos.

CAPÍTULO 5

AOP: adicionando novos comportamentos aos beans

A grosso modo, pode-se dizer que dois fatores levam um desenvolvedor a adotar o Spring Framework: a inversão de controle/injeção de dependências que vimos exaustivamente nos três capítulos anteriores e o suporte à programação orientada a aspectos - a tão falada AOP (*Aspect Oriented Programming*) - oferecido pelo framework.

É graças à AOP que o Spring fornece ao desenvolvedor acesso a recursos - *serviços corporativos* - até então disponíveis apenas em servidores de aplicações pesados como WebLogic, WebSphere, JBoss e outros. Enquanto a IoC minimiza o problema do alto acoplamento, ainda resta um tipo de funcionalidade que normalmente se encontra pulverizada em nossos sistemas, de tal modo que a orientação a objetos não nos permite isolar de uma forma tão limpa quanto gostaríamos. Este tipo de funcionalidade é chamada no jargão da AOP de *interesse transversal* (*cross-cutting concern*).

5.1 COMO IDENTIFICAR OS INTERESSES TRANSVERSAIS

A maior dificuldade enfrentada no primeiro contato com a AOP é a compreensão do que vêm a ser um *interesse transversal*. Vamos expor este conceito escrevendo um profilador (*profiler*) rudimentar. Um profilador é a ferramenta que usamos para descobrir gargalos de performance ou memória em nossos sistemas a partir da medição do tempo de execução e recursos computacionais consumidos em trechos do nosso software.

Não precisamos de um profilador completo para descobrir os gargalos de nossos sistemas. O código abaixo expõe uma primeira versão que nos possibilitará obter a mesma funcionalidade.

```
class DAO {
    void persistirObjeto(Object obj) {
        // obtenho o momento em que se iniciou a execução do código
        long tempoInicial = System.currentTimeMillis();

        // Executo o que realmente interessa neste método
        getSessionFactory().currentSession().save(obj);

        // Volto a obter o tempo total de execução
        long tempoFinal = System.currentTimeMillis() - tempoInicial;
        log.info("Método persistirObjeto levou " + tempoFinal + "ms");
    }
}
```

A mesma estratégia poderia ser adotada em outros pontos do sistema, como por exemplo em uma classe responsável pela emissão de relatórios:

```
class Relatório {
    void gerarRelatorio() {
        // obtenho o momento em que se inicia a execução do código
        long tempoInicial = System.currentTimeMillis();

        // executo o código que gera o relatório
        // (...)

        // obtenho o tempo final de execução
        long tempoFinal = System.currentTimeMillis() - tempoInicial;
        log.info("Método gerarRelatorio levou " + tempoFinal + "ms");
    }
}
```

```
}  
}
```

Observando os dois trechos acima, fica nítido que a única variação é o objetivo de cada método: pois o código que os envolve é basicamente o mesmo. Este código duplicado é o que chamamos de *interesse transversal*. Uma primeira tentativa de modularizá-lo poderia ser isolando-o em uma classe. Esta implementação é o que no jargão da AOP chamamos de aspecto (*aspect*), que está para a AOP assim como a classe está para a programação orientada a objetos.

Nossa primeira implementação deste aspecto rudimentar pode ser vista na classe a seguir:

```
class Profilador {  
    private long tempoInicial;  
    private long tempoFinal;  
    // Obtém o tempo inicial de execução  
    public void iniciarMedicao() {  
        tempoInicial = System.currentTimeMillis();  
    }  
    // Obtém o tempo final de execução  
    public void finalizarMedicao() {  
        tempoFinal = System.currentTimeMillis();  
    }  
    // Retorna o tempo total de execução  
    public long tempoExecucao() {  
        return tempoFinal - tempoInicial;  
    }  
}
```

Temos a impressão inicial de finalmente ter conseguido isolar a funcionalidade alvo. Agora basta injetar a classe *Profilador* nas classes que pretendemos medir o tempo de execução, isolando a chamada ao método e *voilà*!, modularidade obtida, certo? Errado: tudo o que fizemos foi adicionar mais uma dependência nestas classes e assim aumentar o problema do alto acoplamento. Além deste efeito colateral, analisando melhor o problema percebe-se que o mesmo código repetido irá voltar ao sistema, só que de outra forma, ou seja, temos uma solução ilusória para nosso problema.

Como veremos no restante deste capítulo, a AOP nos permite implementar aspectos sem que se torne necessária a inclusão de mais dependências ao código fonte.

5.2 ENTENDENDO AOP

Quando estivermos falando em AOP sempre tenha em mente a ideia de que com ela estamos na realidade *interceptando* a execução dos métodos presentes em nossos *beans*. A grosso modo, escreveremos interceptadores capazes de alterar o fluxo original de execução do nosso código. Os objetos cuja execução será interceptada também possuem um nome dentro do jargão da AOP: *objeto alvo* (*target objects*). Nos exemplos expostos anteriormente, nossos *objetos alvo* seriam as instâncias das classes *Relatório* ou *DAO*.

O mais interessante é que o *objeto alvo* não precisa implementar alguma interface ou possuir o código repetido apresentado anteriormente, preocupando-se portanto apenas com a razão pela qual foi implementado. O leitor atento deve estar se perguntando como isso é possível? Entra em cena outro conceito importante na implementação AOP do Spring: os *objetos proxidados* (*proxy objects*).

Quando falamos em classes *proxy*, estamos nos referindo ao padrão de projeto de mesmo nome que se tornou popular com o livro “Padrões de Projeto” [11] da GoF. É um padrão estrutural com o objetivo de controlar o acesso a um objeto.

A solução propõe a existência de pelo menos três elementos: a classe cliente, o *proxy* e a classe alvo. Aplicando-se a boa prática da programação orientada a objetos de codificarmos para interfaces, teremos as classes *proxy* e alvo implementando uma interface em comum: interface esta que será a dependência da classe cliente como pode ser visto no diagrama abaixo:

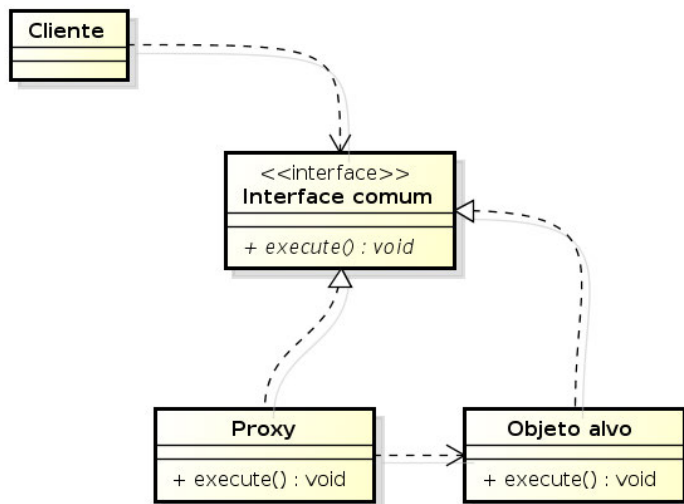


Figura 5.1: O padrão Proxy

Quando o objeto cliente invoca um método da interface, este na realidade estará acessando o objeto *proxy*, que se encarregará de executar ou não, de acordo com as regras de acesso que implementarmos em nosso *proxy*, o método original da classe alvo. Um bom exemplo de aplicação de *proxies* são frameworks de segurança como o Spring Security, que estudaremos mais a fundo ainda nesse livro, mas já podemos adiantar que lá, o *proxy* é implementado de tal forma que a identificação do usuário permita ou não a execução dos métodos que desejamos proteger.

Na plataforma Java não é necessário que as classes alvo e *proxy* implementem a mesma interface: para dizer a verdade, o *proxy* muitas vezes é gerado dinamicamente usando-se bibliotecas de manipulação de *bytecode* como CGLIB, ASM ou mesmo o suporte que o próprio Java SE oferece para este fim.

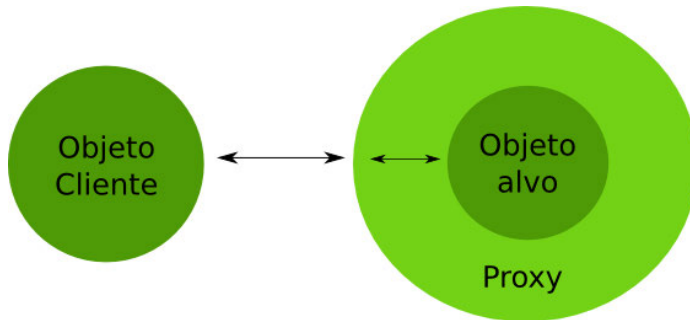


Figura 5.2: Proxies

Graças aos *proxies*, não precisamos modificar nossos objetos alvo. Neste momento outra dúvida surge: como se define quais métodos devem ser interceptados? O fazemos usando alguma regra ou linguagem de expressão que no jargão da AOP chamamos de *pontos de corte* (*point cuts*). A propósito, os *métodos interceptados* também possuem um nome especial, são os *pontos de junção* (*join points*).

Sendo assim, eis como a AOP funciona no Spring: o *objeto alvo* é encapsulado por um *proxy*, quando alguma classe cliente faz uma chamada a algum de seus *pontos de junção*, o *proxy* executa o aspecto antes ou após os *join points* e estes, finalmente, serão ou não executados de acordo com o aspecto implementado.

E sabe esta opção por executar o aspecto antes ou depois do *join point*? Também tem um nome: se chama *advice*. Nesta sessão temos portanto uma descrição intuitiva de como a AOP funciona. Nosso próximo passo será vê-la na prática.

5.3 PREPARANDO O AMBIENTE DE DESENVOLVIMENTO

Graças à natureza modular do Spring, é perfeitamente possível usar o framework sem seu suporte à AOP, que atualmente vêm em duas opções: a tradicional baseada na API do próprio Spring ou a baseada em esquemas XML e a sintaxe de definição de *point cuts* do AspectJ, que é um projeto que tem como objetivo trazer para a plataforma Java uma implementação completa deste paradigma de programação.

Quais componentes devem ser inseridos em seu projeto é portanto uma questão determinada pelo caminho a seguir: API do Spring ou AspectJ?

Em ambos os casos, a dependência obrigatória é o `spring-aop`. Lembre-se de se certificar de que este possui a mesma versão dos demais módulos do Spring que

seu projeto faz referência. Caso esteja usando o Maven, basta inserir a configuração abaixo no seu arquivo `pom.xml`.

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>3.1.1.RELEASE</version>
</dependency>
```

Em diversas situações os objetos alvo não implementarão uma interface específica, tornando-se necessária portanto a geração de *proxies* dinamicamente. Torna-se necessária a inclusão da biblioteca CGLIB 2 em seu projeto, cuja configuração do Maven segue a seguir:

```
<dependency>
  <groupId>cglib</groupId>
  <artifactId>cglib</artifactId>
  <version>2.2.2</version>
</dependency>
```

Finalmente, caso seja feita a escolha pelo suporte ao AspectJ, algumas bibliotecas deste *framework* também devem ser incluídas:

```
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjrt</artifactId>
  <version>1.6.12</version>
</dependency>
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>1.6.12</version>
</dependency>
```

Evite dores de cabeça com incompatibilidade de versões

Um problema comum aos que se iniciam no Spring Framework diz respeito a conflitos de versão dos módulos do Spring. Muitas vezes topamos com exceções em tempo de execução do tipo `LinkageError`, `NoClassDefFoundError`, `NoSuchMethodError`, `NoSuchFieldError` ou `ClassNotFoundException`. Na esmagadora maioria das vezes, este problema ocorre pois temos módulos do Spring com versões diferentes no classpath do sistema.

Caso esteja usando Maven, uma boa solução para evitar este tipo de problema é a inclusão de propriedades no arquivo de configuração, cujo conteúdo corresponda à versão do Spring em uso no projeto. Um exemplo do uso de propriedades pode ser visto no arquivo `pom.xml` de exemplo a seguir:

```
<project>
  (...)
  <properties>
    <!--
      Repare: o nome da tag corresponde ao nome da propriedade.
    -->
    <spring.versao>3.1.1.RELEASE</spring.versao>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <!-- Usamos uma sintaxe similar à EL do
        JSP para inserir a propriedade no conteúdo do XML -->
      <version>${spring.versao}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-aop</artifactId>
      <version>${spring.versao}</version>
    </dependency>
  </dependencies>
  (...)
</project>
```

5.4 A IMPLEMENTAÇÃO DO PRIMEIRO ASPECTO

A melhor maneira que conheço para entender a mecânica por trás da AOP é usando a API do Spring. Apesar de ser uma opção bem menos popular atualmente, por não tirar proveito das anotações e dos esquemas XML, é uma opção que ainda se mostra bastante válida.

Iremos continuar o problema do nosso profilador que, desta vez, será aplicado à classe `DAO Pessoas`, que lê e persiste uma lista de objetos do tipo *Pessoa* e cuja implementação pode ser vista no código a seguir:

```
public class DAOPessoas {
    public void persistir(Pessoa[] pessoas, File arquivo)
        throws IOException {
        if (pessoas == null || pessoas.length == 0) {
            throw new IllegalArgumentException(
                "Lista de pessoas nula ou vazia");
        }
        if (arquivo == null) {
            throw new IllegalArgumentException(
                "Arquivo nulo passado como parâmetro");
        }
        ObjectOutputStream oos =
            new ObjectOutputStream(new FileOutputStream(arquivo));
        oos.writeObject(pessoas);
        oos.close();
    }

    public Pessoa[] ler(File arquivo) throws IOException {
        ObjectInputStream ois =
            new ObjectInputStream(new FileInputStream(arquivo));
        Pessoa[] resultado = null;
        try {
            resultado = (Pessoa[]) ois.readObject();
        } catch (ClassNotFoundException ex) {
            throw new IOException(
                "Erro ao ler arquivo. ClassNotFoundException", ex);
        }
        return resultado;
    }
}
```

5.5 ENTENDA OS ADVICES

Como mencionado acima neste capítulo, o *advice* é o componente da AOP responsável por definir o momento em que um aspecto deve ser executado. O Spring oferece alguns tipos de *advices* que podem ser implementados pelo programador. É importante mencionar que a implementação AOP do Spring não é tão completa quanto a do AspectJ, pois só intercepta execução de métodos. Não há interceptação de modificação de atributos ou construtores, como no AspectJ.

No entanto, esta limitação não é um problema para a esmagadora maioria dos

projetos, pois raras vezes topamos com estes casos. Quando há uma definição de atributos, poderíamos interceptar um método *:setter ao invés do atributo diretamente, e no caso do construtor, poderíamos interceptar um factory method::*

Abaixo seguem os tipos de *advice* suportados pelo Spring:

- **before**: é invocado antes da execução do método alvo (*point cut*);
- **after throws**: invocado caso seja disparada uma exceção após a execução do método alvo;
- **after returning**: invocado quando determinado tipo de retorno ou valor é retornado após a execução do método alvo;
- **introduction**: diz respeito à introdução de métodos em objetos *proxy* pelo Spring (veremos mais sobre este *advice* em um capítulo a parte).

5.6 USE OS AROUND ADVICES

Como nosso objetivo é medir o tempo de execução dos métodos da classe `DAO Pessoas`, nosso profilador deve executar três passos:

- Obter o momento em que a execução do método alvo é iniciada;
- Executar o método alvo;
- Obter o momento atual e subtrair-lo do momento inicial para descobrir quanto tempo levou a execução do método alvo.

Precisamos portanto de um *advice* do tipo *around*, que é também o mais genérico possível, pois ele nos permitirá executar algo antes e após o método que será interceptado.

Para que este seja implementado usando a API do Spring basta escrevermos uma classe que implemente a interface `org.aopalliance.intercept.MethodInterceptor`, tal como foi feito no código a seguir:

```
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class AdviceAroundProfilador implements MethodInterceptor {
    private List<Long> tempos = new ArrayList<Long>();
```

```
public List<Long> getTempos() {return tempos;}

public Object invoke(MethodInvocation invocation) throws Throwable {
    long momentoInicial = System.currentTimeMillis();
    /*
        O método proceed executa a invocação do método
        que estamos interceptando nos fornecendo seu
        valor de retorno.

        Assim é possível, por exemplo, substituir o valor retornado
        para a classe cliente do objeto proxiado ;)
    */

    Object resultado = invocation.proceed();
    long tempo = System.currentTimeMillis() - momentoInicial;
    getTempos().add(tempo);
    System.out.println("Tempo para executar " + tempo + "ms");

    /*
        Na linha abaixo retornamos o resultado da invocação do
        método interceptado
    */
    return resultado;
}
}
```

A interface `MethodInterceptor` requer a implementação de um único método, `invoke`, que recebe como parâmetro um objeto do tipo `MethodInvocation`, que representa, como o próprio nome já nos diz, a invocação de um método. Neste objeto, no entanto, o método que mais nos interessa é `proceed`, que retorna um objeto do tipo `java.lang.Object`, que representa o retorno do método alvo.

Nossa implementação é portanto bastante simples: a única diferença é que os tempos de execução são todos adicionados ao atributo `tempos`, do tipo `java.util.List` para consulta futura.

Um ponto importante em nosso profilador é o fato deste ser completamente independente de qualquer tipo de objeto alvo. Tanto faz se estou interceptando a execução de um objeto do tipo `DAOPessoas` ou `GeradorRelatorios`, tudo o que realmente interessa para nosso aspecto é que o método seja executado e seu tempo de resposta medido. Ainda mais importante, com esta nossa implementação do aspecto garantimos que `DAOPessoas` não precise saber da existência de um profilador.

5.7 USE O ADVICE BEFORE

Finalizada a implementação de nossa classe `DAOPessoas`, surge um novo requisito: precisamos evitar que esta classe sobrescreva arquivos já existentes. A solução imediata para este requisito poderia ser incluir mais um trecho de verificação no método `persistir`, como no exemplo a seguir:

```
public void persistir(Pessoa[] pessoas, File arquivo)
    throws IOException {

    if (pessoas == null || pessoas.length == 0) {
        throw new IllegalArgumentException(
            "Lista de pessoas nula ou vazia");
    }
    if (arquivo == null) {
        throw new IllegalArgumentException(
            "Arquivo nulo passado como parâmetro");
    }
    // Nossa nova validação
    if (arquivo.exists()) {
        throw new IllegalArgumentException(
            "O arquivo já existe e não pode ser sobrescrito!");
    }
    (...)
}
```

Em um primeiro momento, esta parece ser uma ótima solução. Mas e se quisermos incluir ainda mais restrições como, por exemplo, limitar quais os diretórios aonde nosso DAO possa gerar arquivos ou mesmo o número máximo de pessoas que possam ser inseridas? Iremos aumentar indefinidamente nosso método `persistir`? AOP salva nossa pele com o *advice* do tipo *before*, que é executado antes que o método alvo seja chamado.

Um objeto alvo pode ter quantos *advices* de determinado tipo quanto sejam necessários para um mesmo método. Sendo assim, poderíamos implementar um novo *advice* para cada nova restrição. Para mantermos a facilidade de compreensão deste texto, vamos implementar apenas um que verifique se o arquivo em cima do qual serão persistidos os objetos do tipo `Pessoa` exista ou não. Para tal, basta escrevermos uma classe que implemente a interface `org.springframework.aop.MethodBeforeAdvice`, como no código a seguir:

```
import java.io.File;
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

public class AdviceBeforeProfilador implements MethodBeforeAdvice {
    public void before(Method method, Object[] args, Object target)
        throws Throwable {

        if (method.getName().equals("persistir") && args[1] != null) {
            File arquivoAlvo = (File) args[1];
            if (arquivoAlvo.exists()) {
                System.out.println("Before Advice: arquivo já existe");
                args[1] = null;
            }
        }
    }
}
```

Como pode ser visto, esta interface requer que seja implementado um único método chamado `before`, que recebe como parâmetros uma instância do objeto `Method`, que representa o método interceptado, e um array de `Object`, representando os valores que serão passados ao método alvo e por último o objeto alvo que foi interceptado. Para que nossa regra seja aplicável, tiramos proveito do comportamento padrão do método `persistir` de `DAOPessoas`, que sempre dispara uma exceção do tipo `IllegalArgumentException`, caso uma instância do tipo `java.io.File` não seja passada como parâmetro.

Para tal, verificamos se o método `exists` do atributo `File` que seria passado ao método alvo retorna verdadeiro. Em caso positivo, nós substituímos seu valor na matriz por `null`. Sim, você leu certo: nós podemos alterar quais os parâmetros passados ao método alvo dentro do *advice* do tipo *before*, assim como também poderíamos fazer no *advice* do tipo *around*.

CUIDADO COM OS ASPECTOS QUE ESCREVE

Imagine um programador dando manutenção na classe `DAOPessoas` e se deparando com um *bug* do tipo: “O Sistema não permite persistir arquivos no diretório”. Uma de suas primeiras atitudes poderia ser a escrita de um teste unitário no qual a situação descrita fosse reproduzida e, para sua surpresa, o código naquele momento funcionaria perfeitamente.

Horas e horas de trabalho depois, nosso exausto programador se encontra com outro colega que, orgulhoso, descreve o seu trabalho na escrita de um *advice* que altera o valor dos parâmetros passados para esta classe usando AOP.

A possibilidade de modificarmos o valor de parâmetros a serem enviados a objetos alvo ou ainda lógicas mais complexas e intrusivas como a não execução de métodos, são ferramentas poderosas que devem ser usadas com cautela. Antes de executar este tipo de atividade, lembre-se de que o objeto alvo normalmente é implementado tendo como base um cenário original, diferente do que deu origem ao aspecto modificador.

Como boa prática, sempre que a estratégia de modificação de parâmetros for necessária, certifique-se de documentá-la e informar toda a equipe a seu respeito, pois isto pode salvar horas de trabalho dos desavisados e talvez até mesmo algumas amizades. É possível minimizar ainda mais o problema com a presença de testes integrados, que verifiquem o comportamento de nossos beans, ao invés de nossas classes isoladamente.

5.8 E ONDE ENTRA O PROXY NESSA HISTÓRIA?

Isoladamente, os *advices* que implementamos não irão habilitar a AOP. Para que isto ocorra faz-se necessário que declaremos nosso objeto *proxy*. Para tal, usaremos uma instância da classe `org.springframework.aop.framework.ProxyFactoryBean`. A maneira mais fácil de entender seu funcionamento é através da configuração XML:

```
<beans>
```

```
<!-- O nosso objeto alvo -->
```

```

<bean id="daoPessoasTarget" class="casadocodigo.DAOPessoas"/>

<!-- Nosso advice do tipo around -->
<bean id="aroundProfilador"
      class="casadocodigo.api.AdviceAroundProfilador"/>

<!-- Nosso advice do tipo before -->
<bean id="beforeProfilador"
      class="casadocodigo.api.AdviceBeforeProfilador"/>

<bean id="daoPessoas"
      class="org.springframework.aop.framework.ProxyFactoryBean">

    <property name="target" ref="daoPessoasTarget"/>
    <property name="interceptorNames">
        <list>
            <value>aroundProfilador</value>
            <value>beforeProfilador</value>
        </list>
    </property>

</bean>

</beans>

```

Como pode ser observado, declaramos nossos *advice*s e objeto alvo como se fossem *beans* convencionais a serem gerenciados pelo *container*. Logo em seguida, o que fazemos é definir duas importantes dependências do *bean* `daoPessoas`, que será o *proxy* acessado pelas classes clientes: `target`, que recebe como valor o *bean* `daoPessoasTarget` e `interceptorNames`, que possuirá como valor uma lista que aponta para os dois *advice*s que declaramos em nosso arquivo de configuração.

Para ilustrar o uso da classe *proxy* por um cliente, escrevemos um pequeno teste cuja implementação pode ser vista adiante:

```

DAO_Pessoas dao = (DAO_Pessoas) applicationContext.getBean("daoPessoas");
// Deve ser retornado o proxy, e não o objeto alvo
assert dao.getClass().equals(DAO_Pessoas.class) == false;
/* Suponha a existência dos objetos pessoas (do tipo Pessoa[])
   e arquivo (do tipo File) */
dao.persistir(pessoas, arquivo);
// Obtenho a instancia do advice around

```

```

AdviceAroundProfilador around =
    applicationContext.getBean(AdviceAroundProfilador.class);
/*
    Se executei o método, o tempo de execução
    obrigatoriamente precisa estar armazenado
    na lista de tempos do advice around
*/
assert ! around.getTempos().isEmpty();

/*
    Agora é testada a execução do advice before.
    Tentaremos executar o método persistir no mesmo
    arquivo usado anteriormente. Uma exceção do tipo
    IllegalArgumentException deverá portanto ser disparada,
    pois o parâmetro arquivo será substituído por null pelo
    advice.
*/
try {
    dao.persistir(pessoas, arquivo);
} catch (IllegalArgumentException ex) {
    assert "Arquivo nulo foi passado".equals(ex.getMessage());
}

```

5.9 ENTENDA A EXECUÇÃO

Com base no código produzido, até então possuímos a seguinte sequência de execução quando uma classe cliente acessa nosso *proxy*:

- 1) Cliente invoca método *persistir* no objeto *proxy*;
- 2) *Proxy* busca por algum advice do tipo *before* e, existindo, executa-o;
- 3) *Proxy* busca por algum advice do tipo *around* e, existindo, executa-o;
- 4) O *proxy* do tipo *around* executa o método do objeto alvo e o retorna para o *proxy*;
- 5) *Proxy* retorna valor retornado ao objeto cliente.

]

5.10 UM USO INTERESSANTE PARA AOP: LOG DE ERROS

Para finalizar nossa seção sobre o uso tradicional, que tal mais um exemplo prático? Precisamos registrar o momento em que algum parâmetro inválido é fornecido ao nosso DAO. Entra em ação o *advice* do tipo *after throws*. Escrever um *advice* deste tipo é bastante simples: basta que duas regras sejam seguidas:

- A classe deve implementar a interface `org.springframework.aop.ThrowsAdvice`;
- Devem ser implementados um ou mais métodos do tipo `void` cujo nome é `afterThrowing` e que receba como parâmetro uma exceção de qualquer tipo.

A interface `ThrowsAdvice` é na realidade uma interface de marcação, ou seja, esta não obriga o desenvolvedor a implementar nenhum método. Entra em ação a convenção definida pelo Spring, cuja aplicação podemos ver no exemplo:

```
import org.springframework.aop.ThrowsAdvice;

public class AdviceAfterThrows implements ThrowsAdvice {

    // Restante da classe oculto para facilitar a leitura

    // Interceptando exceções do tipo IllegalArgumentException
    public void afterThrowing(IllegalArgumentException ex) {
        enviarEmail("Parâmetros inválidos: " + ex.getMessage());
    }

    // Interceptando exceções do tipo IOException
    public void afterThrowing(IOException ex) {
        enviarEmail("IOException: " + ex.getMessage());
    }
}
```

No código anterior vemos que o mesmo *advice* pode interceptar mais de um tipo de exceção. Neste caso, as trataremos tanto do tipo `java.io.IOException` quanto `java.lang.IllegalArgumentException`. Assim, sempre que um erro dos tipos tratados forem disparados por nossos objetos alvo, um e-mail será enviado para o administrador do sistema. Novamente, não há muito o que ser alterado em nossa configuração XML, cujo resultado final pode ser visto a seguir:

```
<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.xsd">

<!-- O nosso target object -->
<bean id="daoPessoasTarget" class="casadocodigo.DAOPessoas"/>

<!-- Nosso advice do tipo around -->
<bean id="aroundProfilador"
      class="casadocodigo.api.AdviceAroundProfilador"/>

<!-- Nosso advice do tipo before -->
<bean id="beforeProfilador"
      class="casadocodigo.api.AdviceBeforeProfilador"/>

<!-- Advice do tipo after throws -->
<bean id="afterThrows" class="casadocodigo.api.AdviceAfterThrows"/>

<bean id="daoPessoas"
      class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target" ref="daoPessoasTarget"/>
  <property name="interceptorNames">
    <list>
      <value>aroundProfilador</value>
      <value>beforeProfilador</value>
      <!-- Inclusão do after throws -->
      <value>afterThrows</value>
    </list>
  </property>
</bean>

</beans>

```

5.11 DECLARE ASPECTOS COM ANOTAÇÕES ASPECTJ E ESQUEMAS XML

Finalmente, temos a forma mais moderna e atualmente mais popular de declararmos nossos aspectos com Spring usando esquemas XML e AspectJ.

É importante esclarecer que quando nos referimos ao AspectJ, na realidade apontamos para um modo de declaração de aspectos baseado em anotações. Curiosa-

mente, por padrão o Spring não vêm com suporte à todos os recursos oferecidos pelo AspectJ: usaremos apenas sua sintaxe para definição de *point cuts* e as anotações a que este *framework* oferece suporte. Por trás dos panos temos na realidade a mesma API do Spring para suporte à AOP que vimos na seção anterior, com a diferença de que agora esta se encontra “disfarçada de AspectJ”.

Como veremos, há também um novo esquema XML a ser aprendido, que adiciona semântica específica ao suporte a AOP. Com isto a leitura dos nossos arquivos de configuração fica mais fácil de ser compreendida e poderemos identificar mais claramente nesta quem são os *beans* convencionais e quais os nossos aspectos.

Incluindo o novo esquema em nosso arquivo de configuração

Esquemas XML nos possibilitam adicionar novos elementos a nossos arquivos de configuração XML do Spring. Para oferecer suporte ao AspectJ devemos inserir a definição do namespace `aop` no elemento `<beans>` de nosso arquivo de configuração, tal como no exemplo abaixo:

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.1.xsd">
  (...)
</beans>
```

Caso esteja trabalhando com o *SpringSource Tool Suite*, a inclusão deste esquema é ainda mais fácil. Basta abrir seu arquivo de configuração do Spring e escolher a aba *Namespaces*, tal como pode ser visto na imagem a seguir.

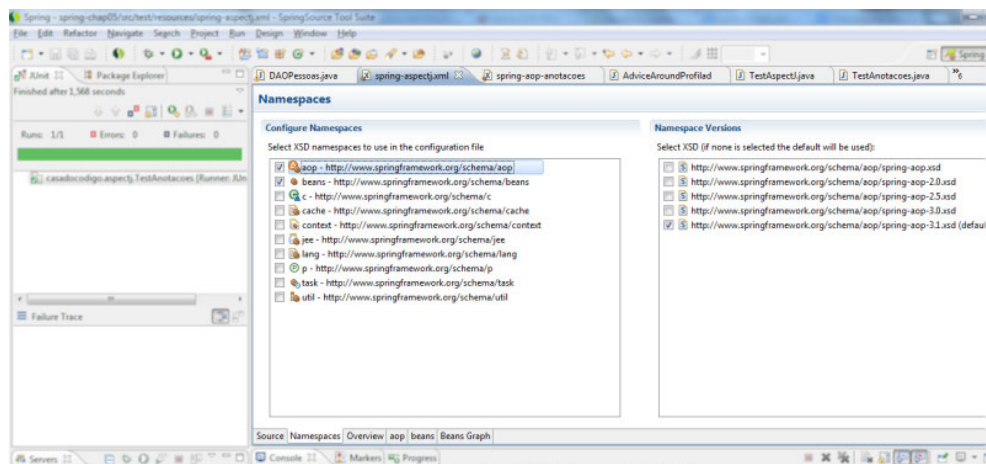


Figura 5.3: Usando o SpringSource Tool Suite para adicionar um novo esquema

Em seguida, basta selecionar o namespace *aop* no painel à esquerda que automaticamente a IDE modificará o arquivo para você.

Declarando aspectos com @Aspect

Com o namespace *aop* incluído em nosso arquivo de configuração, nosso próximo passo é instruí-lo a fazer duas coisas: a mais óbvia que é informá-lo de que usaremos anotações para definir nossos *beans* e habilitar o esquema de *auto proxy* do AspectJ. Voltando ao exemplo que vêm nos acompanhando neste capítulo, nosso arquivo de configuração, excluindo as definições de *namespaces* para facilitar a leitura, deve ficar tal como no código adiante:

<beans>

```
<!-- Informamos o container que iremos usar anotações -->
<context:annotation-config/>
```

```
<!-- Habilitando o auto proxiamiento do AspectJ -->
<aop:aspectj-autoproxy />
```

```
<!-- Em que pacote nossos aspectos se encontram -->
<context:component-scan
    base-package="casadocodigo.aspectj.anotacoes"/>
```

```

    <!-- O objeto alvo -->
    <bean id="daoPessoas" class="casadocodigo.DAOPessoas"/>

</beans>

```

A única alteração incluída em nosso arquivo de configuração foi a tag `<aop:aspectj-autoproxy />`. Podemos ver aqui aplicado o poder de simplificação que anotações e esquemas XML nos fornecem de maneira imediata. Enquanto usando a API padrão do Spring precisamos escrever 14 linhas significativas de configuração, agora precisamos escrever apenas 4: nossa configuração ficou “apenas” 72% menor.

Configuração pronta, agora precisamos apenas escrever nossos aspectos. Para começar, vamos ver como implementar o mais genérico dos advices que é o *around*. Nossa nova implementação pode ser vista abaixo:

```

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Aspect // Identifica o bean como um aspecto
@Component // Identifica o aspecto como um bean
public class AdviceAroundProfiladorAnotacoes {

    private List<Long> tempos = new ArrayList<Long>();

    public List<Long> getTempos() {return tempos;}

    @Around("execution(* casadocodigo.DAOPessoas.* (..))")
    public Object profilar(ProceedingJoinPoint joinPoint)
        throws Throwable {

        long momentoInicial = System.currentTimeMillis();
        Object resultado = joinPoint.proceed();
        long tempo = System.currentTimeMillis() - momentoInicial;
        tempos.add(tempo);
        System.out.println("Tempo para executar = " + tempo + "ms");
        return resultado;
    }
}

```

```
}
```

Fica claro que é basicamente a mesma implementação do nosso profilador. A diferença está no fato de incluirmos duas novas anotações. `@Aspect`, como o próprio nome diz, identifica um *bean* como um aspecto. Já a anotação `@Around` identifica que o método `profilar` - que obrigatoriamente deve receber um parâmetro do tipo `ProceedingJoinPoint` - é um *advice* do tipo *around*.

O mais interessante do uso das anotações é que podemos ter mais de um *advice* do mesmo tipo - *around* - implementado em uma mesma classe: basta incluir mais anotações `@Around` em seu corpo. O leitor deve estar curioso a respeito do significado do parâmetro que passei para esta anotação, certo?

5.12 A SINTAXE ASPECTJ DE DECLARAÇÃO DE POINT CUTS

Como mencionado, uma das maiores vantagens do AspectJ é a sua sintaxe para definição de *point cuts*. Como visto em nosso último aspecto, a anotação `@Around` recebe como parâmetro uma expressão. O assunto desta seção será justamente entender o que esta significa.

Caso você já tenha trabalhado com AspectJ, é importante mencionar que, como o Spring não oferece suporte a 100% deste *framework*, e também pelo fato de nossa AOP ser completamente baseada em *proxies*, não há suporte nativo para todas as possibilidades que esta linguagem para definição de *point cuts* nos oferece, o que não é em si uma grande desvantagem, visto que o suportado dará conta da esmagadora maioria das situações encontradas no desenvolvimento de aplicações corporativas.

A sintaxe aplicada é razoavelmente simples e a imagem abaixo nos ajudará a entendê-la melhor:

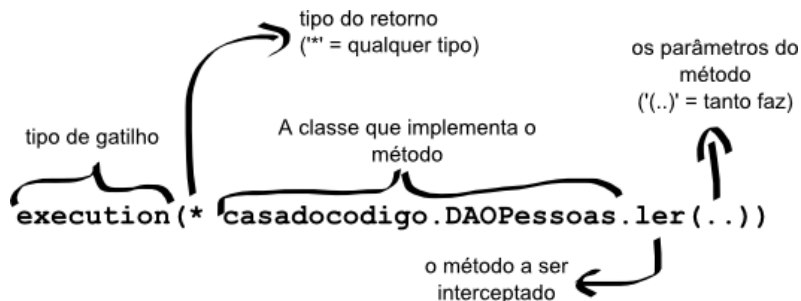


Figura 5.4: Definindo um point cut

O primeiro elemento sempre é o gatilho responsável por disparar o aspecto. Na esmagadora maioria das vezes este será do tipo `execution` que, como o próprio nome já diz, identifica a execução do método alvo. Por esta razão neste capítulo daremos muito mais atenção a este tipo.

TIPOS DE GATILHO MAIS COMUNS

- `execution()`: identifica a execução do método alvo.
- `within()`: define um join point dentro de um conjunto de tipos.
- `this()`: o aspecto será aplicado a objetos do tipo do bean declarado.
- `annotation()`: identifica join points de acordo com as anotações passadas como parâmetro.
- `args`: identifica join points que recebam determinada lista de parâmetros.

O gatilho `execution` recebe como parâmetro a especificação do método alvo, que é composta por quatro componentes. O primeiro diz respeito ao tipo de retorno

esperado, seguido do caminho completo até a classe, nome do método e, finalmente, quais os parâmetros que este receberá.

É importante mencionar que podemos usar caracteres coringa na declaração do nome da classe e método alvo, tal como podemos ver nos exemplos:

```
/*
O método pode retornar qualquer valor. A classe pode estar
em qualquer pacote, porém deve se chamar 'DAOPessoas'.
O método pode possuir qualquer nome, o que indica que pegaremos
todos os métodos, ignorando os parâmetros que estes possam receber.
*/

execution(* *.DAOPessoas.*(..))

/*
Todas as classes cujo nome comece com 'DAO', levando em consideração
apenas os métodos chamados 'ler' ignorando os parâmetros recebidos e
o tipo de retorno
*/

execution(* *.DAO*.ler(..))

/*
Todos os métodos do tipo void de qualquer classe do pacote casadocodigo
ignorando os parâmetros
*/

execution(void casadocodigo.*(..))

/*
Todos os métodos que recebam como parâmetro um objeto do tipo File
pertencentes ao pacote casadocodigo
*/

execution(* casadocodigo.*(java.io.File))

/*
Execução de qualquer método público de qualquer bean
*/
execution (public * *(..))
```

Outro gatilho interessante é o `@annotation`, que nos permite criar *join points* para todas as classes que possuam determinada anotação. O exemplo abaixo leva em consideração todas as classes que possuam a anotação `@Component` do Spring.

```
@annotation(org.springframework.stereotype.Component)
```

O Spring também estende o conjunto de gatilhos com o tipo `bean`, que nos permite referenciar um *bean* por um nome específico ou usando caracteres coringa como nos exemplos abaixo:

```
/* Aponta para ao bean daoPessoas */  
bean("daoPessoas")  
  
/* Aponta para qualquer bean cujo nome comece com 'dao' */  
bean("dao*")
```

É interessante mencionar que também podemos aplicar expressões lógicas na definição de nossos *point cuts*.

Escrevendo outros tipos de advices

Tirando proveito das anotações, podemos ver como implementar mais de um tipo de *advice* em uma mesma classe, como no código a seguir:

```
@Aspect  
@Component  
public class AdviceAroundProfiladorAnotacoes {  
  
    private List<Long> tempos = new ArrayList<Long>();  
  
    public List<Long> getTempos() {  
        return tempos;  
    }  
  
    // Nosso advice do tipo around  
    @Around("execution(* casadocodigo.DAOPessoas.*(..))")  
    public Object profilar(ProceedingJoinPoint joinPoint)  
        throws Throwable {  
        long momentoInicial = System.currentTimeMillis();  
        Object resultado = joinPoint.proceed();  
        long tempo = System.currentTimeMillis() - momentoInicial;  
    }  
}
```

```

        tempos.add(tempo);
        System.out.println("Tempo para executar = " + tempo + "ms");
        return resultado;
    }

    // Mais de um advice pode ser declarado na mesma classe
    // quando usamos anotações
}

```

Os parâmetros passados aos *advices* de tipo *before* ou *after throwing* são opcionais. Para *advices* do tipo *before*, como podemos ver a seguir, é possível passar informações a respeito do *join point* definindo um parâmetro de entrada do tipo `org.aspectj.lang.JoinPoint`, que nos permite, como pôde ser visto anteriormente, interceptar os parâmetros que serão passados ao método alvo.

```

@Aspect
@Component
public class AdviceAroundProfiladorAnotacoes {
    // acima está o nosso advice do tipo around

    // Advice do tipo before
    @Before("execution(* casadocodigo.DAOPessoas.persistir(..)")
    public void coletarEstatisticas(JoinPoint joinPoint) {
        File arquivo = (File) joinPoint.getArgs()[1];
        obterEstatisticasArquivo(arquivo)
    }

    // restante da classe abaixo
}

```

No caso de um *advice* do tipo *after throwing*, é importante uma nota adicional: caso seja passado um parâmetro para este que corresponda à exceção disparada, na anotação `@AfterThrowing` que vemos a seguir deve ser incluído o atributo adicional *throwing*, cujo valor corresponde ao nome do parâmetro declarado no método que implementa o *advice*.

```

@Aspect
@Component
public class AdviceAroundProfiladorAnotacoes {
    // os advices que vimos anteriormente estão acima
}

```

```
// para facilitar a leitura

// Advice do tipo after throwing
@AfterThrowing(
    pointcut="execution(* casadocodigo.DAOPessoas.*(..))",
    throwing="ex")
public void informar(IOException ex) {
    enviarEmail(ex)
}

// restante da classe ignorado para facilitar leitura
}
```

5.13 CONCLUINDO COM UM RESUMÃO CONCEITUAL

Vimos como funciona o mecanismo de AOP implementado pelo Spring. Começamos com a forma mais primitiva oferecida pelo *framework* que é o uso direto de sua API para, logo em seguida, passarmos ao uso dos recursos mais produtivos oferecidos pela ferramenta.

A grande dificuldade na introdução à AOP consiste em evitar o afogamento do leitor na quantidade significativa de conceitos que acompanha este paradigma de desenvolvimento. Sendo assim, um bom resumo dos principais conceitos envolvidos, podem ser elencados como:

- **Interesse transversal:** é a funcionalidade que encontramos em nosso sistema, espalhada como código repetitivo e que dificilmente conseguiríamos isolar de maneira elegante e consistente usando a orientação a objetos;
- **Aspecto:** é a implementação do interesse transversal. Importante lembrar que um aspecto não existe isolado, mas sempre na forma de um *advice*;
- **Advice:** é a forma que um aspecto é implementado. Corresponde ao momento em que este será disparado: é na execução do método? É antes, depois ou após dispararmos uma exceção?
- **Point cut:** é a expressão que identifica quais métodos deverão ser interceptados;
- **Join point:** é o método que queremos interceptar.

E agora vamos para a parte prática, na qual veremos como aplicar todos os conceitos que vimos nesta primeira seção do livro!

Parte II

Spring Framework na prática

Até o capítulo anterior, vimos praticamente toda a teoria necessária para o bom entendimento do Spring Framework. A partir de agora colocaremos mais nossa *mão na massa*. Usaremos como base uma aplicação real chamada *Spring Forum* escrita especialmente para este livro. Esta é um gerenciador de fórum bem parecido com o G.U.J. (<http://www.guj.com.br>), porém bem mais simplificado para que não prejudique a didática deste livro.

Bem vindo(a) ao Spring Forum Brasil!

Baseado no livro "Virando o Jogo com Spring Framework"
de Henrique Lobo Weissmann (Kico)

Assuntos

[AOP](#)[Container IoC](#)[ORM](#)[Spring Batch](#)[Spring MVC](#)[Spring WebFlow](#)

O que é isto?

O projeto Spring Forum é um gerenciador de comunidades virtuais (a la GUJ) baseado nos conceitos apresentados no livro *Virando o Jogo com Spring Framework* de *Henrique Lobo Weissmann*.

Seu código fonte é totalmente aberto e você é livre para explorá-lo da maneira que achar melhor.

Últimos membros

[Henrique Lobo](#)[Kico](#)[Josué](#)[Amadeus](#)[Carvalhinho](#)[John McCarty](#)

Figura 5.5: Página inicial do Spring Forum

CAPÍTULO 6

Colocando a mão na massa

Nosso modelo de domínio será simples, composto inicialmente por apenas quatro classes: `Usuario`, `Assunto`, `Topico` e `Resposta`.

Todo o código fonte é altamente comentado, de tal modo que possa ser usado como referência pelo leitor. Nos capítulos posteriores, sempre haverá menções a esta base para que o leitor possa fazer experimentações em seu ambiente de desenvolvimento local.

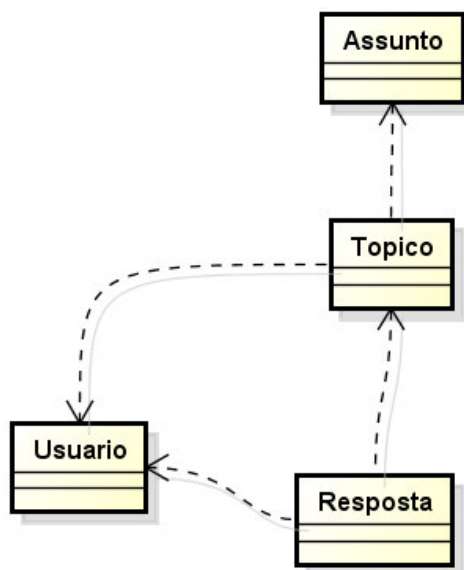


Figura 6.1: Modelo de domínio inicial

6.1 OBTENDO O CÓDIGO FONTE

Usamos o Git (<http://www.git-scm.com>), que é um sistema de controle de versões distribuído bastante popular atualmente. Como muitos projetos, nosso código fonte encontra-se hospedado no GitHub (<http://www.github.com>) no endereço <http://www.github.com/loboweissmann/spring-forum>. Há portanto duas maneiras de se obter o código fonte. A primeira delas consiste em baixá-lo no formato zip como pode ser visto na imagem abaixo:

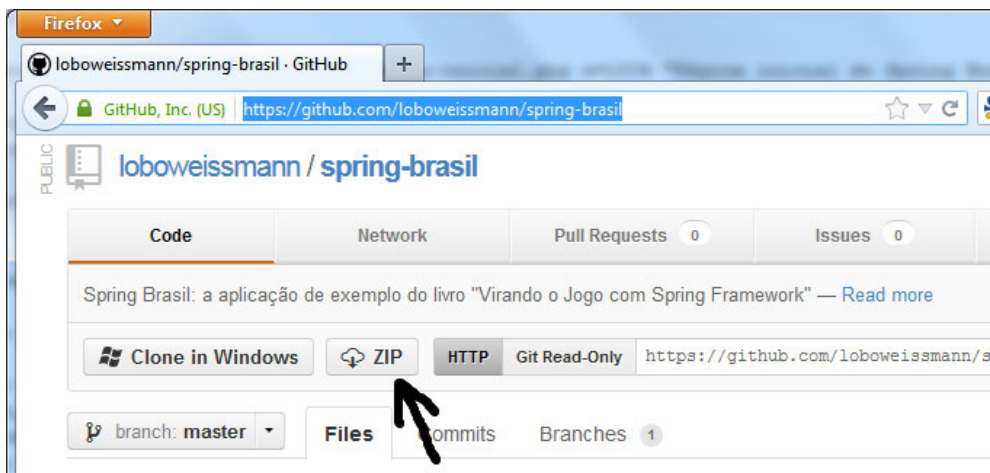


Figura 6.2: O link para baixar nosso código fonte no GitHub

A segunda opção a ser adotada por aqueles que possuem o git instalado em seus computadores podem clonar o repositório executando o comando abaixo:

```
git clone https://github.com/loboweissmann/spring-forum.git [diretório  
aonde será clonado o repositório]
```

Caso não seja fornecido o nome do diretório aonde será clonado o repositório, será gerado um com o nome padrão: spring-forum.

COMO INSTALO O GIT?

A instalação do Git pode ser diferente de acordo com o sistema operacional que você utilizará. Recomendo que você assista os screencasts gravados pelo pessoal da Caelum, disponíveis em: <http://blog.caelum.com.br/passos-a-passos-para-comecar-com-git-e-novo-curso-online/>

6.2 CONFIGURE SEU AMBIENTE DE DESENVOLVIMENTO

O projeto é baseado em Maven (<http://maven.apache.org>), sendo assim o leitor não precisa se preocupar em baixar manualmente todas as dependências usadas. Além de não precisarmos nos preocupar com este trabalho, também ganhamos a possibilidade de usarmos a IDE de nossa preferência, sem complicação nenhuma.

Versão do Java

Todo o código fonte desenvolvido funciona perfeitamente no Java 1.5 ou posterior.

Instalando o Maven

Caso não possua o Maven instalado em seu computador, o procedimento de instalação do mesmo é bastante simples e encontra-se sumarizado no passo a passo abaixo:

- 1) Baixe a última versão do Maven em <http://maven.apache.org> ;
- 2) O Maven é distribuído em arquivos compactados no formato .zip ou .tar.gz. Descompacte-o em um diretório de sua preferência;
- 3) Certifique-se de que exista uma variável de ambiente chamada JAVA_HOME em seu computador que aponte para a sua instalação do JDK;
- 4) Adicione o diretório bin, presente na sua instalação do MAVEN ao path do seu sistema.

Qual IDE usar

Atualmente praticamente todas as IDEs Java oferecem suporte ao Maven. Sendo assim, para executar suas experimentações em nosso código fonte basta que você abra o seu projeto Maven caso esteja usando Netbeans ou então importá-lo para o seu workspace se estiver trabalhando com o Eclipse.

Dado o fato de existir o *SpringSource Tool Suite* que é uma distribuição do Eclipse desenvolvida pela SpringSource, especificamente para o Spring Framework e seus subprojetos (incluindo Grails!), leva em consideração os recursos nesta presente para lidar com arquivos de configuração, anotações, beans e refatorações, portanto acredito que seja atualmente a melhor opção para aqueles que estão dando os primeiros passos no *framework*.

6.3 O QUE VÊM POR AÍ

No capítulo 7 mostraremos o Spring MVC, que como o próprio nome já diz, é o framework web MVC que acompanha o Spring. Veremos como funciona a orques-

tração de requisições e respostas e como integrá-lo com o motor de *templates* Apache Tiles 2.

Conhecendo bem o funcionamento do Spring MVC, podemos começar a nos aprofundar na camada de negócios do nosso projeto, conhecendo os recursos que o Spring nos oferece no capítulo 8, como por exemplo o suporte a Hibernate e JPA, configuração de fontes de dados e tudo o que precisamos para ter nosso ORM favorito funcionando por trás dos panos.

Sabendo como persistir nossas informações, iremos no capítulo 9 conhecer um recurso do Spring que atrai diversos desenvolvedores: o suporte a transações do Spring, assim daremos o controle transacional para nossa aplicação.

No capítulo 10 voltaremos ao Spring MVC explicando o projeto Spring Security, um framework de segurança desenvolvido pela SpringSource bastante flexível e fácil de usar.

Como hoje em dia é raro termos aplicações isoladas, aprenderemos também como integrar nosso projeto com outros sistemas no capítulo 11, aonde veremos o suporte que o Spring oferece a RMI, SOAP, APIs REST e muito mais.

Muitas atividades são executadas assincronamente em nossos projetos, como por exemplo tarefas administrativas: iremos aprimorar a performance do nosso fórum, agendando o seu indexador no capítulo 12.

No capítulo 13 iremos melhorar o modo como nos comunicamos com nossos usuários expondo o suporte que o Spring oferece ao envio de e-mails além da sua integração com o Twitter.

E a partir do capítulo 14 veremos alguns tópicos mais avançados do framework que não serão abordados em nossa aplicação de exemplo.

Preparado? Então, é mão na massa a partir de agora!

CAPÍTULO 7

Desenvolva aplicações web com Spring MVC

O objetivo inicial da primeira versão do Spring era fornecer “apenas” o container de IoC/DI e uma implementação da AOP para a plataforma Java, mas conforme o framework ia tomando forma a equipe de desenvolvimento percebeu que, estando com a faca e o queijo na mão, poderiam sem muito esforço criar uma alternativa ao framework web mais popular da época e que muito os incomodava: Struts. E assim, meio que por acidente foi incluído no projeto um componente que com o tempo se mostrou vital para a popularização do framework: o *Spring MVC*.

Como veremos neste capítulo, o *Spring MVC* é mais que um framework web: é também a aplicação de todas as boas práticas de projeto que vimos nos capítulos anteriores deste livro, o que o torna uma alternativa bastante viável na criação de novas aplicações web para a plataforma Java EE.

7.1 A BASE PARA TUDO: MVC

Atualmente o padrão de projeto mais popular adotado pelos frameworks de desenvolvimento de aplicações web, trata-se de uma estratégia que nos permite isolar completamente - ao menos em teoria - as camadas de negócio (Modelo) e Visualização através da inclusão de uma intermediária: o Controlador. A compreensão deste padrão se tornará mais clara conforme descrevemos estas camadas.

O modelo diz respeito à toda a parte do sistema responsável pela lógica de negócio e seus componentes auxiliares, como persistência, cacheamento, integração com outros sistemas etc.

A visualização, como o próprio nome já nos diz, é a parte visível ao usuário final. Quando nos deparamos com uma janela ou página HTML, estamos lidando diretamente com o resultado desta camada.

Finalmente, temos a camada menos óbvia que é o controlador, que orquestra a interação entre as duas camadas acima citadas. Quando o usuário clica em um *link*, o controlador é acionado. Este transforma os parâmetros de entrada para um formato que seja compatível com a interface disponibilizada pela camada de negócio, cujo resultado do processamento é recebido pelo controlador, que o modifica caso necessário e em seguida o envia à camada de visualização para que seja contemplado pelo usuário final.

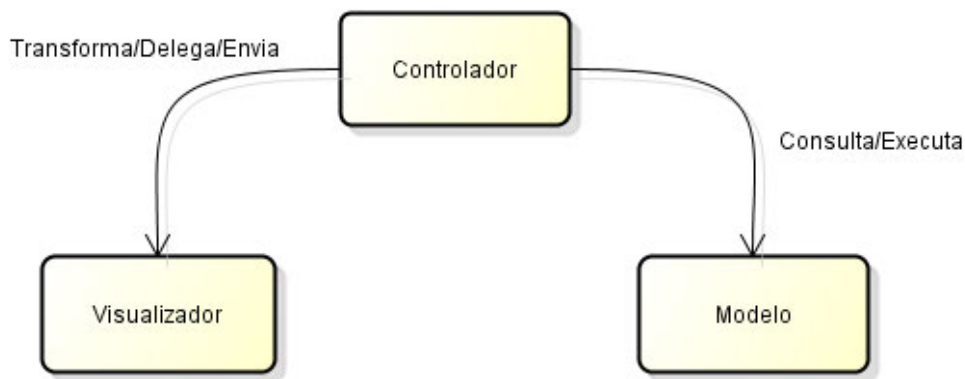


Figura 7.1: O padrão MVC

No final das contas, o que observamos no padrão MVC é mais uma solução ao problema do alto acoplamento. Como a camada de visualização não possui acesso

direto à camada de negócio e vice-versa, a substituição de uma ou outra se torna mais fácil. Além disto, vemos também ganhos imensos na manutenibilidade de nosso modelo. Dado que este **não sabe** da existência de uma camada de visualização, o desenvolvedor precisa lidar com menos dependências e, consequentemente acabará produzindo código de maior qualidade, mais facilmente testável e como objetivamos desde o primeiro capítulo, com acoplamento significativamente mais baixo.

7.2 DISPATCHER SERVLET: O MAESTRO POR TRÁS DO SPRING MVC

O componente responsável por orquestrar o funcionamento do *Spring MVC* é o *Dispatcher Servlet*. Este é na realidade a implementação do padrão *Front Controller*, muito usado na escrita de frameworks voltados para a criação de aplicações web. O objetivo deste padrão de projeto é fornecer um ponto de entrada central para todas as requisições direcionadas à nossa aplicação. É seu trabalho interpretar estes requisitos e decidir qual o componente responsável por seu processamento e eventual retorno para o usuário.

A figura a seguir, inspirada na documentação oficial do projeto, nos ajuda a ter uma visão global do modo como este padrão se encaixa dentro do contexto do Spring MVC.

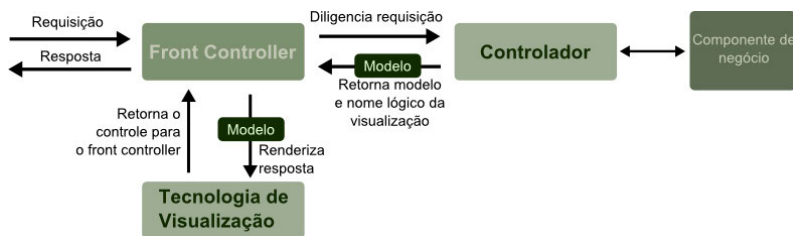


Figura 7.2: O padrão Front Controller no contexto do Spring MVC

Apesar de raríssimas vezes precisarmos interagir diretamente com este componente - que não foi projetado para isto - conhecê-lo é fundamental para trabalharmos com o framework pois interage diretamente com os principais componentes por trás do funcionamento do *Spring MVC*.

Toda chamada ao servidor inicia-se sob a forma de uma requisição e termina como uma resposta enviada ao cliente. Durante esse caminho ocorrem cinco eventos

esquematisados na figura a seguir, que expõem os componentes fundamentais por trás do Spring MVC. Em um primeiro momento o leitor pode se sentir assustado com este número, no entanto é importante mencionar que dos quatro elementos que serão descritos só precisaremos lidar com dois destes após finalizada a configuração da aplicação.

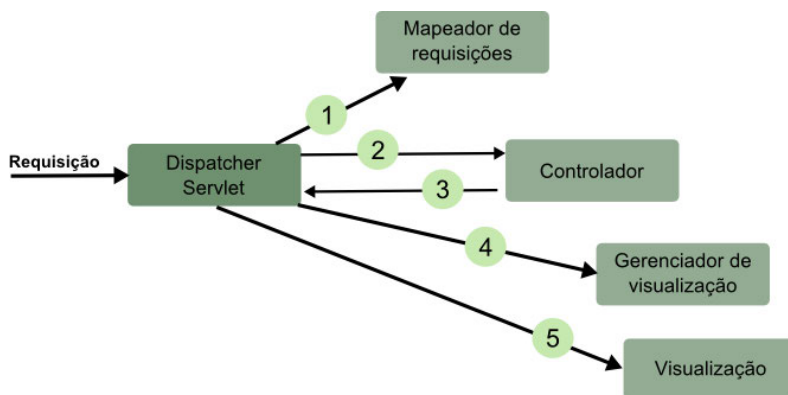


Figura 7.3: O ciclo de vida de uma requisição

Como mencionado, tudo começa no momento em que uma requisição chega ao Dispatcher Servlet. Esta terá sua assinatura analisada e enviada ao Mapeador de requisições (*Handler Mapping*), que é o componente responsável por descobrir qual controlador deve ser acionado.

Uma vez obtido o controlador alvo, este é executado e um nome lógico do template de visualização a ser renderizado como resposta ao usuário (*view*) é retornado ao Dispatcher Servlet e o conjunto de variáveis (*model*) que serão expostas nesta renderização.

Com o nome lógico da visualização e o modelo obtidos, entra em cena o Gerenciador de visualização (*View Resolver*), que possui a função de, com base no nome da view, retornar ao Servlet Dispatcher qual elemento de visualização - mais comumente uma página JSP - deverá ser renderizada de volta ao usuário que fez a requisição.

7.3 A PREPARAÇÃO DO PROJETO

Após toda esta teoria vamos à prática. Nosso primeiro passo será criar um projeto web no padrão Java EE (5 ou posterior). Uma das maneiras mais padronizadas e produtivas atualmente é usando o Maven, que criará a estrutura de diretórios básica e toda a estrutura que usaremos na declaração de nossas dependências. Além disto, projetos baseados em Maven também possuem a grande vantagem de serem agnósticos em relação à IDE, o que permitirá ao leitor trabalhar com a sua opção favorita.

Crie o projeto com Maven

Não é um exagero dizer que todas as IDEs atualmente oferecem suporte à criação de projetos web com Maven, mas o modo como cada uma implementa esta funcionalidade vai além do objetivo deste livro, razão pela qual será exposto apenas como criar um novo projeto a partir de um arquétipo padrão pela linha de comando. No caso, o comando usado para criar o projeto *Spring Fórum* com Maven é o seguinte:

```
mvn archetype:generate -DgroupId=br.com.itexto -DartifactId=spring-Forum  
-Dversion=1.0.0-SNAPSHOT -DarchetypeArtifactId=maven-archetype-webapp
```

Durante a execução do Maven, será perguntado qual o nome do pacote padrão usado pelo projeto. No nosso caso escolhemos `br.com.itexto.springForum`. E com isto criamos um projeto identificado pelo grupo `br.com.itexto` e identificado por `spring-Forum` com a versão `1.0.0-SNAPSHOT`.

Incluindo as dependências fundamentais

Com o projeto criado, nosso próximo passo será incluir as dependências fundamentais: neste primeiro momento só precisamos nos preocupar com uma que é o pacote `spring-webmvc` que deverá ser declarada na tag `<dependencies>` do arquivo `pom.xml`, como a seguir:

```
<dependency>  
  <groupId>org.springframework</groupId>  
  <artifactId>spring-webmvc</artifactId>  
  <version>3.1.2.RELEASE</version>  
</dependency>
```

Observe que não precisamos declarar outros componentes do Spring como por exemplo `spring-core`, `spring-context` e outros, pois todos são dependências tran-

sitivas do `spring-webmvc`. Para facilitar o uso da IDE, outra dependência importante a ser incluída é o JSTL 1.2, cuja dependência é exposta a seguir:

```
<dependency>
  <groupId>jstl</groupId>
  <artifactId>jstl</artifactId>
  <version>1.2</version>
  <scope>provided</scope>
</dependency>
```

Com esta dependência o recurso de preenchimento automático de código pela IDE se tornará mais fácil quando estivermos trabalhando com tags JSTL em nossos arquivos JSP.

Como template, o leitor pode usar o arquivo `pom.xml` presente na raiz do projeto Spring Fórum.

A configuração do Dispatcher Servlet

Se o núcleo do Spring MVC é o *Dispatcher Servlet* nada mais natural que nosso próximo passo seja sua configuração. Precisamos declará-lo no arquivo `web.xml`, presente no diretório `src/main/webapp/WEB-INF` do nosso projeto. O primeiro trecho de código que precisamos declarar é o do Servlet em si:

```
<servlet>
  <servlet-name>DispatcherServlet</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

A tag `<servlet>` é usada para declarar um servlet disponibilizado por nossa aplicação. Atenção especial deve ser dada ao nome que usaremos para identificar o servlet (`<servlet-name>`). Uma vez iniciado, o Servlet irá carregar um container do Spring configurado via XML por um arquivo chamado `WEB-INF/[nome do servlet]-servlet.xml`. Sendo assim, de acordo com o código anterior o Servlet buscará pelo arquivo `DispatcherServlet-servlet.xml`.

É possível customizar o nome do arquivo de configuração, para isto basta adicionar o parâmetro de inicialização `contextConfigLocation` à declaração do Servlet:

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

Declarado o Servlet, o próximo passo será o mapeamento das URLs a este direcionadas. Para tal usamos a tag `<servlet-mapping>`:

```
<servlet-mapping>
  <servlet-name>DispatcherServlet</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

Atenção especial deve ser dada à tag `<url-pattern>`. No caso do *Spring Fórum*, optamos pelo mapeamento `/`. Assim instruímos o Servlet Container a tratar nosso *Servlet Dispatcher* como o padrão a ser invocado para todas as requisições recebidas pela aplicação. Este não é o único mapeamento possível: na realidade qual opção a ser adotada irá variar de acordo com a situação e o gosto do desenvolvedor.

Podemos ter algumas outras opções interessantes de mapeamento que poderiam ser adotadas sem qualquer problema:

- `/**`: indica que qualquer requisição independente da URL será direcionada ao nosso servlet.
- `*.jsp`: toda URL terminada com `.jsp`. Exemplo: `http://www.springFórum.com.br/entrada.jsp`
- `/spring/**`: toda URL iniciada com `spring`. Exemplo: `http://www.springFórum.com.br/spring/entrada`

Evite chateações com JSP

A tecnologia de visualização mais adotada por programadores *Spring* é o JSP. É importante lembrar que nem sempre a *Expression Language* (EL) vêm habilitada por padrão, tal como ocorre nas versões anteriores à especificação Servlet 2.5, o que pode levar um desenvolvedor distraído ou pouco experiente a gastar horas do seu tempo tentando entender a razão pela qual estas simplesmente não funcionam.

Uma solução bastante ingênua seria incluir a diretiva de página `isELIgnored` no início de todo arquivo JSP do nosso projeto:

```
<%@ page isELIgnored="false" %>
<html>
...
</html>
```

Esta não é uma solução interessante, pois prejudica a produtividade do desenvolvedor, além de poder ser facilmente esquecida. Uma alternativa mais viável é ativar a *Expression Language* por padrão no arquivo `web.xml`:

```
<web-app
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  id="SpringForum"
  version="2.5">
  <display-name>Spring Fórum</display-name>

  <jsp-config>
    <jsp-property-group>
      <url-pattern>/**</url-pattern>
      <el-ignored>>false</el-ignored>
    </jsp-property-group>
  </jsp-config>
  <!-- Declaração dos servlets abaixo --!>
</web-app>
```

A tag `<jsp-config>` deve ser definida antes da declaração dos Servlets. Atenção especial deve ser dada à versão do `web.xml` declarada: 2.5 no mínimo. Outro ponto

de atenção é a tag `<url-pattern>`: esta define a qual grupo de arquivos JSP estamos nos referindo com este bloco de configurações. Neste caso, todos os arquivos receberão este tratamento.

Prepare o container do Spring

Com toda a estrutura fundamental pronta, nosso último passo na preparação do ambiente será a configuração do container IoC. No caso do *Spring Fórum*, o principal arquivo de configuração é o arquivo `spring-servlet.xml`, presente no diretório `src/main/webapp/WEB-INF/spring`.

Entra em cena um novo *namespace*, chamado `mvc`, que será usado para alimentarmos o *Dispatcher Servlet* com configurações adicionais. É neste ponto da aplicação que são incluídas as definições de componentes como o *Handler mapping* e o *View resolver* por exemplo. A seguir podemos ver a declaração deste namespace:

```
<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.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.1.xsd
    http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd">
  <!-- Conteúdo ignorado para facilitar leitura -->
</beans>
```

Apenas a criação do arquivo de configuração não é suficiente para que o Spring MVC seja ativado: é necessário incluirmos algumas instruções básicas neste:

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

<context:annotation-config />

<context:component-scan
  base-package="br.com.itexto.springForum.controladores"/>
</beans>
```

A única novidade é a tag `<mvc:annotation-driven/>`, que instrui o *Servlet Dispatcher* a buscar seus controladores a partir de anotações: a forma mais popular (e

produtiva) de se trabalhar com o Spring MVC. Sozinha, no entanto, esta não possui muita serventia, é necessário instruir o container de que este buscará a definição de nossos beans a partir de anotações - o controlador é um bean diferenciado pelo estereótipo `@Controller` - e quais os pacotes que devem ser varridos para tal: voltam as tags `<context:annotation-config/>` e `<context:component-scan>` que vimos no capítulo 4.

7.4 COMO TRABALHAR COM CONTEÚDO ESTÁTICO

Toda aplicação web precisa enviar a seus clientes conteúdo estático sob a forma de imagens, JavaScript, CSS etc. No caso do *Spring Fórum*, vamos optar por salvar todos estes recursos no diretório *webapp/recursos*. Em um primeiro momento poderíamos pensar em configurar um Servlet apenas para enviar estes recursos, talvez até mesmo implementando algum mecanismo de cacheamento HTTP. Isto é desnecessário, pois desde a versão 3.0.4 do Spring uma nova funcionalidade foi incluída no framework justamente para resolver essa situação. Trata-se da tag `<mvc:resources />`.

Usar esta tag é bastante simples:

```
<beans>
  <mvc:resources mapping="/recursos/**"
                 location="/recursos/"
                 cache-period="120"/>
</beans>
```

O parâmetro `mapping` identifica o padrão de URL usado quando desejamos obter um recurso estático. Em nosso caso optamos por um mapeamento no qual toda URL possua o prefixo `recursos`. Sendo assim, quando o browser acessa um endereço como por exemplo `http://localhost:8080/spring-forum/recursos/css/main.css`, será enviado para ele o arquivo `main.css`, presente no diretório *webapp/recursos/css*. Os dois asteriscos incluídos no final do valor indica qualquer subcaminho após o prefixo.

Para definir qual o local de armazenamento dos recursos estáticos usa-se o atributo `location`. Por padrão, este apontará sempre para algum diretório presente na raiz da aplicação. É interessante observar que também é possível definir como caminho alguma localização no *classpath* ou mesmo um diretório presente no sistema de arquivos:

```
<!-- A partir de um diretório -->
<mvc:resources mapping="/recursos/**"
```

```
        location="file:/arquivos"/>
<!-- A partir do classpath -->
<mvc:resources mapping="/recursos/**"
        location:"classpath:springForum.imagens"/>
```

Finalmente, temos o único parâmetro opcional desta tag - e nem por isto menos importante - que é o `cache-period`. Este recebe como valor o tempo em segundos de cacheamento do recurso estático enviado. Será incluído na resposta HTTP enviada ao browser um cabeçalho do tipo `Expires` cujo valor será a hora corrente acrescida do número de segundos que definimos no atributo.

CACHE HTTP COM O CABEÇALHO EXPIRES

Um dos requisitos não funcionais mais importantes de qualquer aplicação é performance. No caso de aplicações web, um gargalo bastante conhecido é a transmissão de conteúdo estático. Os navegadores tentam minimizar este problema cacheando o resultado das solicitações, evitando assim a ocorrência de solicitações repetidas a um mesmo recurso. É uma estratégia interessante e que pode ficar ainda melhor se fornecermos uma dica ao browser sobre quando aquele recurso estará desatualizado.

O protocolo HTTP nos permite fazer isto com o cabeçalho `Expires`. No caso, este fornece uma data de expiração para o recurso baixado, possibilitando ao navegador só repetir a consulta caso esta já se encontre no passado. A título de curiosidade, este cabeçalho segue a seguinte sintaxe:

```
"Expires" ":" HTTP-date
```

Um valor válido para a expiração deste livro seria, por exemplo:

```
Expires: Thu, 01 Dec 2024 16:00:00 GMT
```

Com isso, o navegador pediria essa informação novamente para o servidor apenas em 1 de Dezembro de 2024. Antes disso, usaria o que já estava cacheado e foi buscado na primeira requisição.

O uso desta tag também facilita muito a escrita de nossos arquivos JSP. A seguir podemos ver um pequeno trecho de um arquivo JSP, no qual declaramos o acesso a alguns arquivos CSS do nosso projeto.

```
<head>
  <link rel="stylesheet"
        href="recursos/stylesheets/foundation.min.css"/>
  <link rel="stylesheet" href="recursos/stylesheets/main.css"/>
</head>
```

7.5 NOSSO PRIMEIRO CONTROLADOR

Para este capítulo inicialmente iremos nos preocupar com apenas cinco beans, ilustrados com suas respectivas dependências na imagem a seguir:

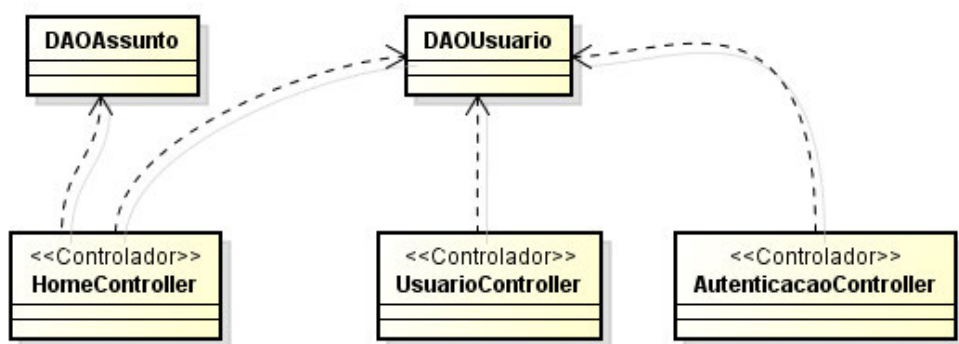


Figura 7.4: Os beans que tomarão nossa atenção

Três destes beans são controladores: HomeController, responsável pela página inicial e o formulário de cadastro do sistema, AutenticacaoController que usaremos temporariamente como nosso mecanismo de autenticação e UsuarioController, para listar o cadastro de usuários do sistema. Estes usam como dependências dois DAOs: DAOUsuario e DAOAssunto, usados respectivamente para obter e persistir instâncias de objetos do tipo Usuario e Assunto.

O primeiro controlador que veremos é também o mais simples: trata-se de HomeController, cujo código fonte responsável por processar a visualização da página inicial do site é listado a seguir:

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HomeController {

    @Autowired
    private DAOUsuario daoUsuario;
    @Autowired
    private DAOAssunto daoAssunto;

    @RequestMapping("/")
    public String index(Map<String, Object> model) {
        model.put("assuntos", getDaoAssunto().list());
        model.put("usuarios", getDaoUsuario().list());
        return "index";
    }

    // getters e setters
}
```

As linhas acima nos dizem bastante a respeito do modo como controladores são escritos para o Spring MVC. Nosso primeiro ponto de atenção diz respeito a anotação `@Controller`, que na realidade é uma especialização de `@Component` que vimos no capítulo 4.

Temos aqui um bean como outro qualquer, gerenciado pelo container - observe que possui algumas dependências - que será encontrado pela instrução `<component-scan>` incluída no arquivo `spring-servlet.xml` descrito no início do capítulo.

A anotação `@RequestMapping`, uma vez aplicada a uma função de nosso controlador o transforma em uma *action*, isto é, em um método responsável pelo processamento da URL mapeada. Observe que nossa action recebe como parâmetro um mapa. Este possui um nome especial dentro do Spring MVC: se chama modelo (*model*), e representa o conjunto de dados que são transportados do controlador para a camada de visualização.

Como pode ser observado, incluímos dois atributos no mapa: *assuntos* e *usuarios*, representando respectivamente a listagem de assuntos e últimos usuários cadastrados no sistema, tal como pode ser visto em destaque na imagem a seguir:



Figura 7.5: Aonde será exposto o conteúdo dos atributos `::assuntos::` e `::usuarios::` incluídos no modelo

O último ponto a ser observado é o valor de retorno do nosso método: a `String` `index` aponta para o nome lógico do *template* de visualização que será renderizado como resposta ao usuário que originou a requisição. Entenderemos com detalhes o mecanismo de renderização no transcorrer do capítulo.

7.6 A DEFINIÇÃO DA CAMADA DE VISUALIZAÇÃO

Assim que uma requisição é tratada e processada, uma resposta deve ser devolvida para o usuário. O responsável por essa tarefa é a camada de Visualização, onde temos diversas ferramentas que podem nos auxiliar, que são gerenciadas pelo *View Resolver*. Alguns dos *Resolvers* mais populares são:

- `InternalResourceViewResolver`: encontra o arquivo de visualização (normalmente JSP) no *classpath* da aplicação e o renderiza como resposta;
- `FreeMarkerViewResolver`: visualização baseada no projeto FreeMarker (<http://freemarker.sourceforge.net>), que é um motor de templates que se mostrou bastante popular no passado;
- `JasperReportsViewResolver`: visualização baseada no gerador de relatórios Jasper Reports (<http://jasperforge.org>);
- `TilesViewResolver`: uma das opções mais populares adotada em projetos Spring MVC. Permite a criação de interfaces mais elaboradas através de tem-

plates. Veremos mais a respeito do seu uso neste capítulo;

- `VelocityLayoutViewResolver`: usa o Apache Velocity (<http://velocity.apache.org>) como tecnologia de visualização;
- `XsltViewResolver`: uma opção bastante interessante para projetos que usem transformações XSLT na sua camada de visualização.

Estas são apenas algumas das opções que acompanham o Spring MVC e há muitas outras escritas por desenvolvedores independentes disponibilizadas para a comunidade. Caso seja de interesse do leitor, para implementar o seu próprio *View Resolver* basta escrever uma classe que implemente a interface `org.springframework.web.servlet.ViewResolver`.

Vamos tratar das opções mais populares entre os programadores.

InternalResourceViewResolver

A opção mais simples é o *InternalResourceViewResolver* e seu uso é bastante similar ao de outras implementações como *FreeMarkerViewResolver* e o *VelocityLayoutViewResolver*. Sua estratégia consiste em localizar o template de visualização a partir de um prefixo e sufixo definidos pelo desenvolvedor.

Para melhor entender seu funcionamento leve em consideração a opção de configuração abaixo:

```
<!--
    Não é preciso definir um nome para este bean, pois
    o Spring MVC busca beans do tipo ViewResolver pelo tipo
-->
<bean class=
    "org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass"
        value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/views"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

As propriedades que realmente nos interessam são *prefix* e *suffix*, que são usadas para definir a convenção adotada pela aplicação na localização dos templates de visualização. Levando em consideração o valor de retorno do nosso primeiro Controller, *index*, de acordo com a estratégia definida na configuração do nosso *View*

Resolver o template a ser renderizado seria `/views/index.jsp`. Se o nome lógico do componente de visualização fosse `usuario/show`, a view obtida seria o arquivo `/views/usuario/show.jsp`.

A última propriedade definida no bean acima se chama `::viewClass`, que usamos para definir qual o tipo de renderização adotada. No caso, `org.springframework.web.servlet.view.JstlView`, que tira proveito do JSTL padrão da plataforma Java EE, o que faz bastante sentido visto estarmos trabalhando com arquivos JSP.

Tiles 2 para interfaces mais elaboradas

Um dos requisitos do *Spring Fórum* é sua uniformidade visual. Todas as páginas do sistema devem possuir alguns elementos, como por exemplo formulário de autenticação, logotipo e rodapé. Ainda mais importante que isto, estes elementos devem ser implementados de tal modo que nossa produtividade seja garantida, ou seja, devem ser reaproveitáveis para que evitemos assim os riscos do copiar e colar. Na imagem a seguir podemos ver um exemplo de como estes blocos se repetem nas páginas do site:

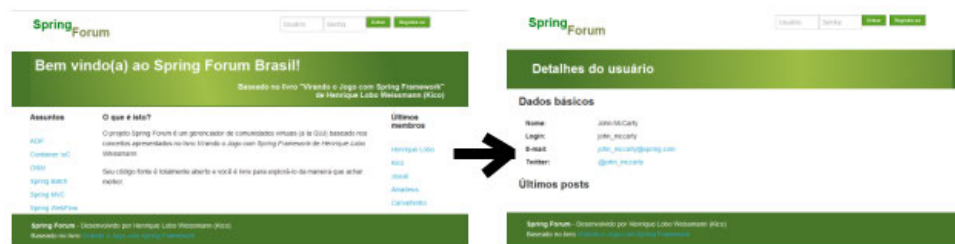


Figura 7.6: Uniformidade visual do Spring Fórum

Toda página do *Spring Fórum* possui a mesma estrutura organizada em três grandes blocos: *cabeçalho*, aonde se encontra o formulário de login e o logotipo do site, *conteúdo* aonde é exposto o tema de cada página visitada e, finalmente, *rodapé*, para informações não menos importantes, como créditos, links para contato etc. Na figura a seguir podemos ver a estrutura seguida pelas páginas:

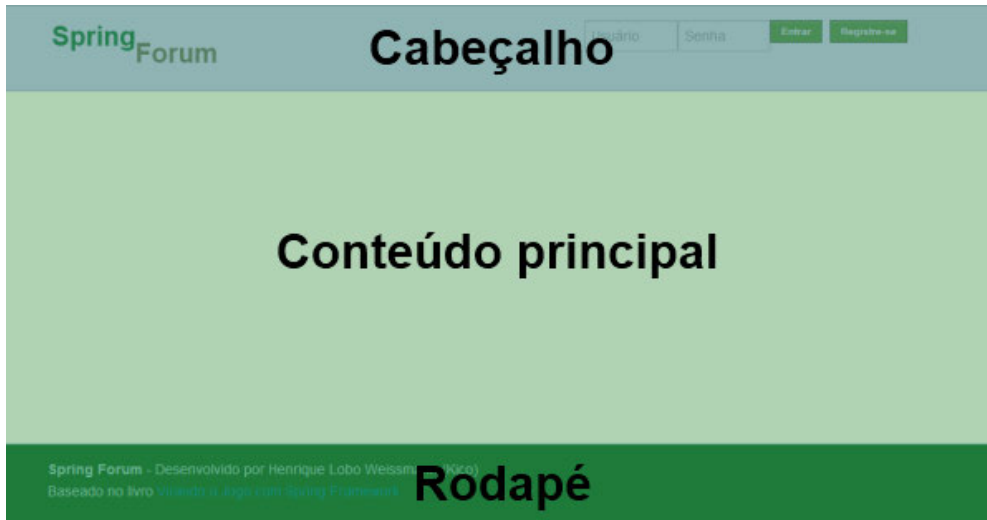


Figura 7.7: Esquema visual de nossas páginas

Para obter este resultado, implementamos as seções reaproveitáveis como trechos JSP, muitas vezes chamados de *templates* (ou *view helpers*) que serão inseridos no corpo principal da página conforme a ordem exposta na imagem acima. Usando JSP convencional obtemos este resultado com JSTL adotando uma estratégia similar à exposta a seguir:

```
<!DOCTYPE html>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
  <!-- Incluo o cabeçalho -->
  <c:import url="templates/cabecalho.jsp"/>
  <body>

    <!-- O conteúdo entraria aqui -->

    <c:import url="templates/rodape.jsp"/>
  </body>
</html>
```

Os arquivos `cabecalho.jsp` e `rodape.jsp` são os componentes a serem inseridos em nossa página. O problema é que precisaríamos repetir o esquema exposto no exemplo acima em cada página JSP do sistema que seguisse este layout, o que além

de não ser uma técnica produtiva, ainda vêm com o risco do esquecimento. Pior: se no futuro for necessário modificar o layout básico da aplicação, teríamos de alterar todos os arquivos aonde aplicamos a formatação original.

Entra em cena o *Apache Tiles* (<http://tiles.apache.org>), que é um motor de templates baseado em composição (*composite view*) [3]. Inicialmente desenvolvido como um componente do *Jakarta Struts* (<http://struts.apache.org>) por Cédric Dumoulin, com o tempo se tornou tão popular que acabou se transformando em um projeto independente adotado por diversos frameworks dentre os quais se encontra o *Spring MVC*.

Ao pensar em um “motor de templates baseado em composição”, lembre-se daqueles jogos de encaixe infantis. Assim como seu análogo físico, ele é composto por uma moldura e as peças que nesta encaixamos. No jargão do Tiles, a moldura se chama *definição* (::definition:), e as peças serão os atributos ou templates.

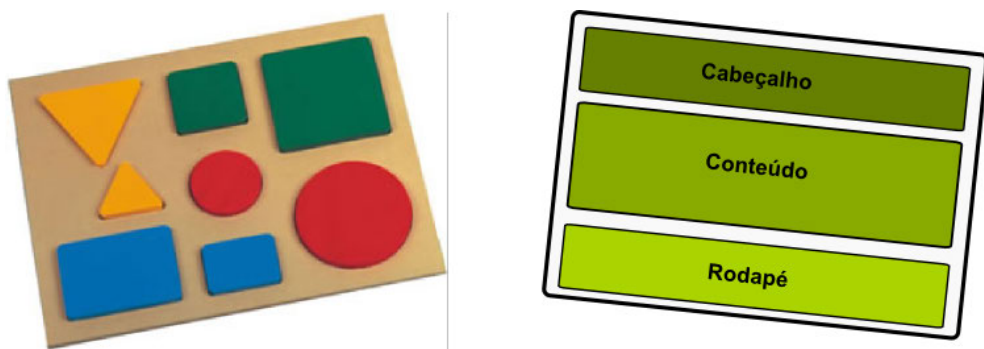


Figura 7.8: O Tiles é um jogo de encaixe, só que pra web

Configuramos o Tiles através de um arquivo XML aonde serão incluídas as definições e seus respectivos componentes. Podemos ver um exemplo desta configuração no código a seguir, que fica no arquivo `/src/main/webapp/tiles/tiles-config.xml` do Spring Fórum.

```
<!DOCTYPE tiles-definitions PUBLIC
    "-//Apache Software Foundation//DTD Tiles Configuration 2.0//EN"
    "http://tiles.apache.org/dtds/tiles-config_2_0.dtd">
<tiles-definitions>
    <!-- Definição básica -->
    <definition name="main" template="/views/templates/main.jsp">
```

```
<put-attribute name="titulo" value="Spring Fórum"/>
<put-attribute name="cabecalho"
    value="/views/templates/cabecalho.jsp"/>
<put-attribute name="rodape"
    value="/views/templates/rodape.jsp"/>
</definition>

<definition name="index" extends="main" >
    <put-attribute name="conteudo" value="/views/index.jsp"/>
</definition>

<definition name="signup" extends="main">
    <put-attribute name="conteudo" value="/views/signup.jsp"/>
    <put-attribute name="titulo" value="Bem vindo(a)!"/>
</definition>

<!-- restante omitido -->

</tiles-definitions>
```

A tag `<tiles-definition>` define o template e possui um único parâmetro obrigatório: `name`, que o identifica. A “moldura” usada para agrupar os componentes do template é definida no parâmetro `template`, cujo valor deve apontar para o arquivo JSP relacionado.

Como pode ser visto na configuração, os componentes não necessariamente apontam para arquivos JSP. Também podem apontar para `String`, como por exemplo o atributo `titulo`, usado para definir o título da página.

Nossa primeira definição, `main`, é a base para todas as outras do sistema. Tiles nos permite usar a herança que estamos acostumados a aplicar em nossas classes. Para tal, usamos o atributo `extends` da tag `<definition>` tendo como valor o nome da definição pai. A partir de então basta o programador definir quais atributos deseja sobrescrever em cada descendente (o que é opcional) ou mesmo adicionar novos caso ache necessário. O interessante do Tiles é que, conforme pode ser observado em nosso exemplo, a partir do momento em que escrevemos o template pai o desenvolvedor só precisará escrever o trecho JSP referente ao conteúdo de cada página.

Nosso template inicial lembra bastante o que escrevemos com JSTL anteriormente, a diferença está no uso da biblioteca de tags oferecida pelo Tiles:

```
<%@taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles" %>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

```
<html>
<head>

    <link rel="stylesheet"
        href="<c:url value="/recursos/stylesheets/estilo.css"/>" />

    <!-- Inclusão do título pelo Tiles -->
    <title>
        <tiles:getAsString name="titulo" />
    </title>
</head>
<body>
    <!-- Inclusão do cabeçalho -->
    <tiles:insertAttribute name="cabecalho"/>
    <!-- Inclusão do conteúdo -->
    <tiles:insertAttribute name="conteudo"/>
    <!-- Inclusão do rodapé -->
    <tiles:insertAttribute name="rodape"/>
</body>
</html>
```

As tags do Tiles são muito simples. Se quisermos incluir atributos que não sejam arquivos, basta incluir uma tag como por exemplo `<tiles:getAsString/>` e para incluir conteúdo dinâmico, use `<tiles:insertAttribute/>`. E isto termina nosso curso básico de Tiles, nosso próximo passo agora será configurá-lo em nossa aplicação e entender como seu *View Resolver* funciona.

Incluindo o suporte a Tiles em nosso projeto

Precisamos adicionar como dependência em nosso projeto o Apache Tiles. Até a versão corrente deste livro, o Spring ainda não oferece suporte nativo para a versão 3 do framework. Sendo assim, teremos de nos contentar por enquanto com a último *release* da família 2.0. Podemos modificar o `pom.xml` para adicionar essa nova dependência:

```
<dependency>
    <groupId>org.apache.tiles</groupId>
    <artifactId>tiles-core</artifactId>
    <version>2.2.2</version>
</dependency>
<dependency>
```

```
<groupId>org.apache.tiles</groupId>
<artifactId>tiles-jsp</artifactId>
<version>2.2.2</version>
</dependency>
```

Estas dependências correspondem ao core do Tiles e seu suporte a JSP. Claro, isto não é o suficiente, precisamos também incluir o *View Resolver* específico para o Tiles em nosso arquivo de configuração, tal como exposto a seguir:

```
<bean id="tilesViewResolver"
      class="org.springframework.web.servlet.view.UrlBasedViewResolver">
  <property name="viewClass"
    value="org.springframework.web.servlet.view.tiles2.TilesView"/>
</bean>
```

Como pode ser visto, estamos usando um view resolver do tipo *UrlBasedViewResolver*. O atributo *viewClass* define qual a classe responsável por renderizar nossas respostas de volta ao usuário final. Esta configuração não está completa: precisamos adicionar mais um bean em nosso sistema que instrua o Tiles a respeito de quais arquivos de configuração deverão ser levados em consideração durante sua execução. Entra em cena mais um bean: *TilesConfigurer*, cuja declaração é exposta a seguir:

```
<bean class=
  "org.springframework.web.servlet.view.tiles2.TilesConfigurer">
  <property name="definitions">
    <list>
      <value>/tiles/tiles-config.xml</value>
    </list>
  </property>
</bean>
```

Observe que não é necessário definir o nome do bean: nosso View Resolver configurado via Tiles sempre buscará outro bean do tipo *TilesConfigurer* para que funcione corretamente. Há um único atributo que precisamos definir neste bean: *definitions*, que recebe como valor uma lista de strings representando o caminho em nosso projeto aonde as configurações deverão ser encontradas.

Como o View Resolver do Tiles funciona

Todo nome lógico de view retornado por nossos controladores corresponderá a uma definição presente nos arquivos de configuração do Tiles. É importante mencionar que o nome lógico de uma visualização pode conter caminhos, como por exemplo `usuario/show`, o que é inclusive boa prática, pois possibilita a equipe de desenvolvimento no futuro optar por algum outro tipo de camada de visualização, como o `InternalResourceViewResolver`, que busca os templates a partir do caminho definido por seus prefixo e sufixo.

7.7 TRABALHE COM REDIRECIONAMENTOS E SESSÃO DO USUÁRIO

Faremos agora um controlador um pouco mais elaborado: aquele responsável pela autenticação de usuários no *Spring Fórum*, que neste primeiro momento ainda é excessivamente rudimentar, mas que já nos fornece a base para que possamos entender melhor o funcionamento da camada de controle do *Spring MVC*.

A marcação HTML usada em nosso formulário é a mais simples possível e é exposta a seguir. O único aspecto que talvez seja novidade é o uso da tag JSTL `<c:url/>` para garantir que a URL da tag *action* do formulário sempre aponte para o endereço correto, no caso, o endereço relativo da nossa URL de autenticação.

```
<form action="
```

Uma vez feita a submissão, nosso *Handler Mapping* irá identificar o controlador de autenticação pelo fato deste possuir a anotação `@RequestMapping` apontando para o mesmo endereço e em seguida executará o truque:

```
@Controller
public class AutenticacaoController {
    @RequestMapping(value="/login", method=RequestMethod.POST)
    public String login(@RequestParam("login") String login,
                       @RequestParam("senha") String senha,
```

```
        HttpSession sessao) {  
    Usuario usuario = daoUsuario.getUsuario(login, senha);  
    if (usuario == null) {  
        return "loginFalho";  
    } else {  
        usuario.setUltimoLogin(new Date());  
        daoUsuario.persistir(usuario);  
        sessao.setAttribute("usuario", usuario);  
        return "redirect:/";  
    }  
}  
}
```

7.8 A DEFINIÇÃO DO MÉTODO DE ACESSO

A primeira novidade que o leitor encontrará em nosso controlador é a inclusão do parâmetro `method` na anotação `@RequestMapping`. Este determina que nossa ação somente reconhecerá o acesso a URL de autenticação se o método HTTP for um POST. Ao tentar acessar o mesmo endereço pelo navegador, por exemplo, seremos saudados pelo erro HTTP 405 nos informando que o método não é suportado pelo *Spring Forum*, já que o navegador envia uma requisição GET, como pode ser visto na imagem a seguir.

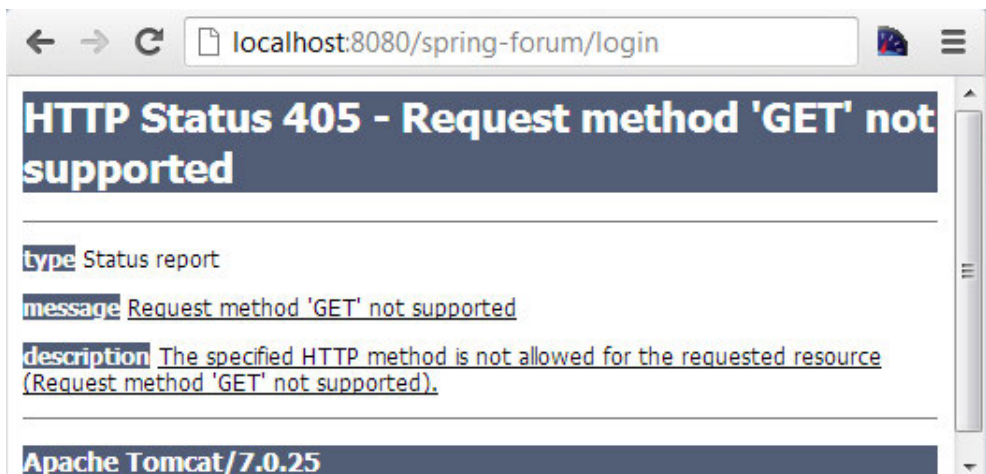


Figura 7.9: Tentar se autenticar no sistema pela URL não irá funcionar.

É possível também passar mais de um método de acesso ao parâmetro `method`. Basta incluí-los em uma matriz, tal como no exemplo a seguir, que possibilita agora autenticação também pelo método PUT.

```
@RequestMapping(value="/login",  
                 method={RequestMethod.POST, RequestMethod.PUT})
```

7.9 RECEBA PARÂMETROS DE ENTRADA

Um ponto interessante no Spring MVC é a total liberdade oferecida ao desenvolvedor no que diz respeito à assinatura de nossas ações:

```
@RequestMapping(value="/login", method=RequestMethod.POST)  
public String login(@RequestParam("login") String login,  
                   @RequestParam("senha") String senha,  
                   HttpSession sessao) {  
    ...  
}
```

Surge uma nova anotação: `@RequestParam`, que mapeia os parâmetros do formulário (ou URL) para os de nossa ação. O valor fornecido para a anotação corresponde ao nome do parâmetro de entrada definido pelo formulário ou URL. Seu uso é opcional porém recomendado. Caso omitida, o *Spring MVC* fará o mapeamento de parâmetros com base no nome dos mesmos definidos no método.

É importante observar que no *bytecode*, o nome dos parâmetros só se encontra presente quando o código é compilado com informações de depuração. Dado que nem sempre podemos ter esta garantia, adota-se a prática de anotar os parâmetros dos métodos como uma boa prática a ser seguida.

7.10 FAÇA REDIRECIONAMENTOS

Há um último ponto interessante em nossa classe `AutenticacaoController`. Quando é encontrado um usuário com o `login` e `senha` fornecidos, o valor retornado pela *action* é `redirect:/`. Esta instrução diz ao *Spring MVC* que deve ser feito um redirecionamento para a visualização cujo nome lógico é `/`.

Outra forma interessante de redirecionamento poderia ser definir como valor de retorno o resultado do processamento de outro método presente em um controlador. Hipoteticamente, poderíamos refatorar a função `login` que vimos acima para aplicar

esta técnica, tal como no exemplo a seguir, em que outra função, `loginFalho()` executaria alguma lógica de negócio e retornaria o nome lógico da View, representando uma autenticação mal sucedida.

```
@RequestMapping(value="/login", method=RequestMethod.POST)
public String login(@RequestParam("login") String login,
                   @RequestParam("senha") String senha,
                   HttpSession sessao) {
    Usuario usuario = daoUsuario.getUsuario(login, senha);
    if (usuario == null) { // Nenhum usuario encontrado
        return loginFalho();
    } else {
        // autentica usuario
    }
}
```

7.11 COMO LIDAR COM A SESSÃO DO USUÁRIO E A CLASSE `ModelAndView`

Vimos como acessar diretamente a sessão do usuário na action `login`, implementada na classe `AutenticacaoController`. Basta incluir como parâmetro um atributo do tipo `javax.servlet.http.HttpSession`. Também é possível acessar atributos de sessão de forma transparente, isto é, sem a necessidade de manipulação direta de objetos `HttpSession` de forma declarativa usando anotações.

Para exemplificar este recurso, nosso foco agora será o controlador `UsuarioController`, onde implementaremos uma ação responsável por retornar os detalhes do usuário atualmente autenticado no sistema. Relembrando, na action `login`, quando a autenticação é bem sucedida criamos um novo atributo de sessão chamado `usuario`. Este comportamento é o gancho para a action `infoAutenticado` cujo código fonte podemos ver a seguir:

```
@Controller("usuario")
@SessionAttributes("usuario")
public class UsuarioController {

    @RequestMapping("/usuario/autenticado")
    public ModelAndView infoAutenticado(
        @ModelAttribute("usuario") Usuario usuario) {
```

```
        ModelAndView mav = new ModelAndView("usuario/show");
        mav.getModel().put("usuario", usuario);
        return mav;
    }
}
```

A técnica contém dois passos: o primeiro consiste na inclusão da anotação `@SessionAttributes` na classe que implementa nosso controlador. Esta pode receber dois tipos de valores, uma `String`, tal como fizemos na classe `UsuarioController` ou no caso de precisarmos de mais de um atributo de sessão, um `array de String`. O valor destas `String` deve corresponder ao nome do atributo com o qual queremos trabalhar.

Usaremos os atributos de sessão no segundo passo: anotamos os parâmetros de nossas actions `@ModelAttribute`, que recebe como valor o nome da chave correspondente ao valor presente na sessão do usuário, tal como foi feito na action `infoAutenticado` exposta acima.

Por trás dos panos, o funcionamento do Spring MVC é simples. Após a requisição ser recebida pelo *Dispatcher Servlet*, este irá buscar o controlador responsável por seu processamento. Estando a anotação `@SessionAttributes` presente, este passará à action os atributos de sessão definidos pelas anotações `@ModelAttribute` e em seguida continuará sua execução normal.

É interessante observar que, ao contrário do que vimos até o momento, o valor de retorno da action `infoAutenticado` não é uma `String`, mas sim um objeto do tipo `org.springframework.web.servlet.ModelAndView`, que é uma classe utilitária do framework, usada para encapsular duas entidades que vimos sendo trabalhadas individualmente no decorrer deste capítulo: o nome lógico da visualização que será renderizada de volta ao usuário (*view*) e os dados embutidos nesta presentes no modelo (*model*).

Nas primeiras versões do *Spring MVC* este era o valor de retorno obrigatório de todas as actions, pois tornava a implementação do *Spring* mais fácil de ser feita. Passado algum tempo, com a introdução das anotações no Java 5 tornou-se mais fácil a obtenção das duas entidades encapsuladas por este objeto, o que fez com que esta convenção caísse por terra.

Apesar de não ser mais um tipo de retorno obrigatório, ainda é um objeto bastante útil e possui alguns métodos, como os listados abaixo, que facilitam bastante a vida do desenvolvedor:

- `String getViewName()`: retorna o nome da view;
- `void setViewName(String valor)`: define o nome da view a ser renderizada;
- `ModelAndView addObject(String nome, Object valor)`: adiciona um novo atributo ao modelo;
- `ModelMap getModelMap()`: retorna um objeto do tipo `ModelMap` representando o modelo. Esta classe por baixo dos panos é uma implementação da interface `Map<String, Object>` do Java;
- `void clear()`: limpa o modelo.

7.12 O CHATO E REPETITIVO TRABALHO DE CRIAR FORMULÁRIOS

Difícilmente encontramos uma aplicação web com interface gráfica que não precise lidar com algum tipo de formulário. Vimos como tratar formulários bastante simples no controlador `AutenticacaoController`, mas o exemplo visto não reflete a realidade, pois convenhamos, são muito raros formulários com tão poucos campos. Nesta sessão conheceremos o suporte oferecido pelo *Spring* na manipulação e validação de formulários.

Processando formulários

Faremos o formulário de cadastro de usuários do *Spring Fórum*, que usa como controlador a classe `HomeController`.

Spring Forum

Bem vindo(a) ao Spring Forum

Esperamos poder lhe ajudar a dominar o máximo possível todos os conceitos por trás do Spring Framework e seus projetos relacionados

Nome:

Email:

Nome do usuário (login):

Senha:

Faça parte!

Figura 7.10: O formulário de cadastro

Os campos presentes no formulário correspondem aos atributos definidos na classe de domínio `Usuario` que tem parte de sua implementação listada a seguir:

```
public class Usuario {  
    private String nome;  
    private String email;  
    private String login;  
    private String senha;  
  
    // getters e setters  
  
}
```

Parte do nosso objetivo no preenchimento do formulário é também o preenchimento dos atributos da classe `Usuario`. Para obter este resultado, entra em cena a biblioteca de tags do *Spring* para o processamento de formulários, cuja aplicação vemos na marcação do nosso formulário de registro a seguir:

```
<%@taglib prefix="sf" uri="http://www.springframework.org/tags/form"%>
<sf:form modelAttribute="usuario" action="executarRegistro">
    <label for="nome">Nome:</label>
    <sf:input path="nome"/>

    <label for="email">Email:</label>
    <sf:input path="email"/>

    <label for="login">Nome do usuário (login):</label>
    <sf:input path="login"/>

    <label for="senha">Senha:</label>
    <sf:password path="senha"/>

    <input type="submit" value="Faça parte!"/>
</sf:form>
```

O *Spring MVC* vem com uma biblioteca de tags especificamente desenvolvida para processamento de formulários. Esta é de fácil aprendizado para os desenvolvedores, pois possui semântica muito próxima ao subconjunto de tags presentes no HTML para o mesmo fim.

Como veremos, para praticamente todas as opções presentes no HTML há uma tag feita pela equipe de desenvolvimento do *Spring*, usada para fazer a ligação entre os objetos presentes no modelo fornecido à visualização e os campos definidos no formulário.

Como pode ser visto na listagem acima, para iniciarmos a implementação de nossos formulários na marcação começamos com a tag `<sf:form>` (você possui liberdade total para escolher o prefixo que quiser). Esta corresponde à tag de mesmo no HTML e possui como principal diferencial o atributo `modelAttribute`, onde o valor aponta para o objeto no modelo e cujas propriedades serão expostas ou ligadas aos campos do formulário.

No caso do formulário anterior, o objeto `usuario` não surge como mágica. Este foi incluído no modelo pela `action registro`, implementada no controlador `HomeController`:

```
@RequestMapping("/registro")
public String registro(Map<String, Object> model) {
    if (model.get("usuario") == null) {
        model.put("usuario", new Usuario());
    }
}
```

```
    }  
    return "registro";  
}
```

No formulário, podemos ver duas tags de ligação fornecidas pelo framework: `<sf:input>` e `<sf:password>`, usadas respectivamente para campos textuais e senha. O atributo mais importante presente nestas tags é `path`, cujo valor aponta para o nome da propriedade do objeto representado aonde será feita a ligação.

É importante mencionar que o valor do atributo `path` não precisa ser uma propriedade direta do objeto. Se hipoteticamente quiséssemos definir um atributo do tipo `Assunto` na classe `Usuario`, poderíamos ligá-lo ao formulário sem problema algum tal:

```
<label for="Assunto favorito">Assunto favorito:</label>  
  
<!-- Referenciando uma propriedade indireta -->  
<sf:input name="assunto" path="assunto.nome"/>
```

Além das tags listadas acima, também podemos contar com uma série de outras, cujas principais encontram-se listadas a seguir:

- `checkbox`: usada para renderizar campos do tipo caixa de seleção;
- `checkboxes`: usada para renderizar uma lista de caixas de seleção;
- `radiobutton`: para *radio* buttons;
- `select`: para caixas de seleção. E sim, há também uma `option` para que possamos definir com maior controle as opções presentes nesta;
- `textarea`: para caixas de texto de múltiplas linhas;
- `hidden`: usada para campos ocultos.

Submissão de formulários

Conhecidas as tags básicas de ligação, o próximo passo é a submissão. Muitos ao iniciarem o aprendizado do *Spring MVC* a partir da documentação oficial sofrem dificuldades em entender como proceder, justamente por ser um passo incrivelmente simples. A melhor maneira de entendê-lo é na prática, sendo assim, veja a implementação da action `executarRegistro`, implementada na classe `HomeController`:

```
@RequestMapping(value="/executarRegistro", method=RequestMethod.POST)
public String executarRegistro(Usuario usuario, HttpSession sessao) {
    getDaoUsuario().persistir(usuario);
    sessao.setAttribute("usuario", usuario);
    return "redirect:/";
}
```

O desenvolvedor precisa apenas incluir um parâmetro na assinatura do método alvo, cujo tipo corresponda ao objeto inicialmente presente no modelo e todo o processo de ligação é feito de forma transparente pelo framework.

No caso da página de registro, a entidade `Usuario` possui todas as suas propriedades diretamente relacionadas ao formulário. Nem sempre este será o caso. Para estas outras situações a prática mais comum é a criação de uma classe especial cujos atributos possam ser diretamente mapeados para o formulário em questão.

7.13 AINDA HÁ MAIS DE SPRINGMVC POR VIR

Vimos muito do funcionamento do SpringMVC e com o conhecimento que temos, já conseguimos criar qualquer aplicação web. No entanto, no mundo real, gostaríamos de facilitar muitas das tarefas que temos ao criar uma aplicação, por exemplo facilitar a validação dos dados, poder fazer upload de arquivos de forma descomplicada e outros recursos interessantes. No próximo capítulo, vamos ver como poder usar outras bibliotecas para aumentar o poder do SpringMVC e ver como ele pode nos ajudar nessa integração.

CAPÍTULO 8

Ações recorrentes com o SpringMVC

Há muito trabalho repetitivo a ser feito quando desenvolvemos aplicações web. Validação é um caso bem clássico. Qual desenvolvedor nunca se pegou escrevendo diversos `if` aninhados e repetidos? Upload de arquivos também pode ser considerado uma dessas tarefas chatas que temos que lidar quando desenvolvemos aplicação. Ficar manipulando bytes e arquivos é algo bastante trabalhoso. Será que não há uma maneira de deixar tudo isso mais fácil?

Nesse capítulo vamos estudar a integração do SpringMVC com diferentes bibliotecas e ver como elas podem atuar em conjunto para nos ajudar a criar uma aplicação web rica e de fácil desenvolvimento e manutenção.

8.1 FAÇA A VALIDAÇÃO DE FORMULÁRIOS

Um aspecto fundamental do desenvolvimento de qualquer aplicação diz respeito à validação dos dados de entrada. Se mal planejada, a validação de formulários pode se mostrar uma das tarefas mais tediosas e trabalhosas durante o processo de desenvolvimento.

Graças à equipe de desenvolvimento do Spring, não precisamos nos preocupar com isto, pois devido à própria natureza do framework de se integrar a outras tecnologias, este oferece suporte à JSR-303 (*Bean Validation*), que nos permite adicionar regras de validação aos nossos formulários ou classes de domínio através de anotações de uma forma bastante produtiva.

Configure as dependências necessárias

A implementação de referência para a JSR-303 é o projeto *Hibernate Validator*. Ao contrário do que possa parecer a alguns, trata-se de um projeto independente do ORM (*Object Relational Mapping*) de mesmo nome, podendo ser usado sem a necessidade de incluí-lo entre as dependências do projeto. Basta incluir o *Hibernate Validator* na lista de dependências no seu `pom.xml`:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>4.2.0.Final</version>
</dependency>
```

Definindo as validações

Usar o Hibernate Validator no contexto do *Spring MVC* é razoavelmente simples. Basta anotar os atributos da classe na qual queremos aplicar validações tal como fizemos com a classe *Usuario* listada a seguir:

```
public class Usuario {
    @Size(max=128) @NotNull @NotEmpty
    private String nome;

    @Email(message="Isto não é um e-mail válido") @NotNull @NotEmpty
    private String email;

    @NotNull
```

```
private Date dataCadastro = new Date();

@NotNull @NotEmpty
@Size(min=8, max=32, message="Login muito curto ou muito longo")
private String login;

// getters e setters
}
```

O nome das anotações já nos diz bastante a respeito do tipo de validação a ser aplicado em nossa classe. Das validações listadas acima, a única que não faz parte do conjunto padrão da JSR-303 é `@Email`, pertencente ao *Hibernate Validator*, usada para verificar se o valor corresponde a um endereço de e-mail válido.

É possível definir qual a mensagem de erro a ser exibida ao usuário no caso de uma validação falha, através do parâmetro `value`, presente em todas as anotações compatíveis com a JSR-303. A seguir como referência rápida segue uma listagem das anotações de validação mais comumente usadas pelos desenvolvedores:

- `@NotNull`: garante que o atributo anotado não possua valor nulo.
- `@NotEmpty`: aplicável a coleções ou strings, verifica se o texto digitado está em branco ou se a lista está vazia.
- `@Size`: usada para garantir que o tamanho de um texto esteja dentro dos parâmetros limitadores *min* ou *max*. Importante salientar que não é necessária a presença dos dois parâmetros, mas sim ao menos um.
- `@Email`: específica do *Hibernate MVC*, é usada para garantir que o valor do atributo seja um endereço de e-mail válido formalmente.
- `@Pattern`: recebe como parâmetro obrigatório *regexp* representando uma expressão regular contra a qual será validado o atributo anotado. Talvez seja a anotação mais poderosa da JSR-303 devido à grande flexibilidade oferecida pelas expressões regulares.

Validação do lado servidor

Para entender como a validação do lado servidor é feita, vamos modificar a ação `executarRegistro`. Sua lógica de funcionamento é simples: primeiro é verificado se há algum erro de validação. Caso existam, será feito o redirecionamento para o formulário de cadastro, aonde os mesmos serão expostos para o usuário:

```

@RequestMapping(value="/executarRegistro", method=RequestMethod.POST)
public String executarRegistro(@Valid Usuario usuario,
                               BindingResult bindingResult,
                               HttpSession sessao) {

    if (bindingResult.hasErrors()) {
        Map<String, Object> model = new HashMap<String, Object>();
        model.put("usuario", usuario);
        return registro(model);
    }

    getDaoUsuario().persistir(usuario);
    sessao.setAttribute("usuario", usuario);
    return "redirect:/";
}

```

Há detalhes importantes no corpo desta ação, a começar pela presença da anotação `@Valid` aplicada sobre o parâmetro `usuario`, que instrui o *Spring MVC* a executar os testes de validação sobre o objeto anotado no momento de execução da ligação dos parâmetros. Encontrando problemas, estes são armazenados em um objeto do tipo `org.springframework.validation.BindingResult`, que é a interface interna do framework responsável por representar o resultado da execução do processo de ligação (*binding*).

É importante mencionar que nem sempre a validação termina com a execução da função `hasErrors()` de `BindingResult`. Esta interface também nos permite iterar sobre os erros encontrados, de tal forma que possamos aplicar alguma regra de negócio em cima da validação. Por exemplo: imagine que tenhamos convencionado que, caso o *host* do e-mail não tenha sido fornecido o valor padrão seja `@gmail.com`. Poderíamos tirar proveito de `BindingResult`:

```

List<FieldError> erros_email = bindingResult.getFieldErrors("email");
for (FieldError erro : erros_email) {
    if (erro.getCode().equals("Email") &&
        contemEmail(usuario.getEmail())) {

        // aplico a convenção do sistema
        usuario.setEmail(usuario.getEmail() + "@gmail.com");
    }
}

```

A classe `FieldError` representa um erro encontrado durante a validação em algum campo da classe avaliada. Esta possui um atributo chamado `code`, cujo valor normalmente corresponde ao nome da anotação de validação. No código anterior, tudo o que fizemos foi iterar em cima da coleção de erros associados ao campo `email` em busca da validação `Email`. Encontrando-a, verificamos se o usuário havia digitado algum valor e em caso positivo, simplesmente adicionamos o sufixo `@gmail.com` ao valor até então incorreto.

Exiba os erros de validação

Encontrar erros de validação do lado servidor de nada vale se não fornecermos ao nosso usuário a possibilidade de corrigi-los. Para fazer essa exibição, podemos usar a mesma biblioteca de tags utilizada para gerar o formulário, e que já vimos anteriormente.

Para expor as mensagens de erro de volta ao usuário usamos a tag `<sf:errors/>`, obtendo resultado similar ao exposto na imagem a seguir:

A imagem mostra um formulário web com o título "Bem vindo(a) ao Spring Fórum". Abaixo do título, há um texto de boas-vindas: "Esperamos poder lhe ajudar a dominar o máximo possível todos os conceitos por trás do Spring Framework e seus projetos relacionados". O formulário contém os seguintes campos e mensagens de erro:

- Nome: **não pode estar vazio** (campo vazio)
- Email: **não é um endereço de e-mail** (campo contendo "asdf")
- Nome do usuário (login): **tamanho deve estar entre 8 e 32** (campo contendo "df")
- Senha: (campo vazio)

Abaixo dos campos, há um botão verde com o texto "Faça parte!".

Figura 8.1: Expondo mensagens de validação

O atributo `path` filtra os erros por campo. Já o atributo `cssClass` define o nome do seletor CSS que define a formatação a ser visualizada pelo usuário. Podemos ver um exemplo de seu uso no código a seguir, extraído do formulário de cadastro apresentado acima:

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

```
<sf:form modelAttribute="usuario" action="executarRegistro">
  <label for="nome">Nome:<sf:errors path="nome" cssClass="erro"/>
</label>
  <sf:input path="nome"/>
  <!-- Restante omitido, pois é basicamente o
        mesmo código aplicado a outros campos -->
</sf:form>
```

O único atributo obrigatório é `path`, que quando omitido, simplesmente não renderizará saída alguma para o usuário. Abaixo segue uma lista com os outros atributos que o desenvolvedor pode tirar proveito em seus projetos:

- `element`: define qual elemento HTML que irá envolver cada mensagem de erro;
- `cssStyle`: permite ao desenvolvedor incluir estilo CSS customizado para uma localização específica;
- `delimiter`: que texto será usado para separar as mensagens de erro expostas. Seu valor padrão é `
`.

8.2 ENVIE SEUS AVATARES PARA O SERVIDOR COM UPLOAD DE ARQUIVOS

Faremos agora a evolução final do nosso formulário de registro incluindo a possibilidade de fazer o upload de avatares, o que possibilitará aos membros do *Spring Fórum* se expressarem visualmente através destas imagens. Até o momento fizemos o *binding* apenas de classes simples como `Usuario`, porém ao lidar com upload de arquivos estamos trabalhando com uma criatura bastante diferente.

Configurando o projeto

Infelizmente o *Spring MVC* não vem pronto para lidar com upload de arquivos, porém a configuração do projeto para tal é bastante simples, composta por apenas dois passos: inclusão de novas dependências e a definição do bean responsável por lidar com este tipo de operação, o `MultipartResolver`.

Duas dependências precisam ser incluídas em nosso projeto: *Commons IO* e *Commons File Upload*, ambas pertencentes ao projeto *Apache Commons* (<http://commons.apache.org>). Com isso, podemos adicionar as duas no `pom.xml`:

```
<dependency>
  <groupId>commons-fileupload</groupId>
  <artifactId>commons-fileupload</artifactId>
  <version>1.2.2</version>
</dependency>
<dependency>
  <groupId>commons-io</groupId>
  <artifactId>commons-io</artifactId>
  <version>2.1</version>
</dependency>
```

O passo final para habilitarmos o upload de arquivos em nosso projeto é a declaração de um bean do tipo `org.springframework.web.multipart.commons.CommonsMultipartResolver`, responsável por interceptar requisições de formulário com `enctype` do tipo `multipart/form-data`. Atenção especial deve ser dada ao nome do bean: este obrigatoriamente deve ter o nome `multipartResolver`, pois o *Dispatcher Servlet* buscará um bean com este nome para lidar com upload de arquivos.

```
<bean id="multipartResolver" class=
  "org.springframework.web.multipart.commons.CommonsMultipartResolver">
  <property name="maxUploadSize" value="1048576"/>
</bean>
```

A única propriedade definida no bean é `maxUploadSize`, que recebe um número inteiro como valor representando o tamanho máximo em bytes que o projeto permitirá o upload. A definição desta propriedade do ponto de vista de segurança é altamente recomendada, pois por default seu valor é `-1`, indicando a possibilidade de recebermos arquivos de qualquer tamanho.

A atualização do formulário

As modificações que precisamos fazer ao formulário são exatamente as mesmas que faríamos a um formulário HTML convencional: apenas inclusão do atributo `enctype` e de um campo do tipo `file`. Importante observar que, ao contrário dos demais tipos de campo de entrada que foram expostos neste capítulo, não é possível fazer o *binding* com campos do tipo `file`. A seguir podemos ver os únicos trechos alterados em sua marcação e o resultado final após estas modificações:

```
<sf:form modelAttribute="usuario"
  action="executarRegistro" enctype="multipart/form-data">
```



```

<!-- Restante oculto -->

<label for="avatar">Avatar:</label>
<input type="file" name="avatar"/>

<!-- Restante oculto -->
</sf:form>

```

The screenshot shows a web form with the following elements:

- Nome:** A text input field.
- Email:** A text input field.
- Nome do usuário (login):** A text input field.
- Senha:** A text input field.
- Avatar:** A section containing:
 - A button labeled "Escolher arquivo" (Choose file).
 - Text indicating "Nenhum arquivo selecionado" (No file selected).
 - A green button labeled "Faça parte!" (Join!).

Figura 8.2: A versão final do nosso formulário de cadastro

Configure o controlador

De nada adianta alterar as configurações do projeto e evoluir o formulário de cadastro se a action `executarRegistro`, responsável pelo processamento dos parâmetros submetidos pelo formulário não for melhorada também. A seguir podemos ver como ficou a versão final deste método:

```

@RequestMapping(value="/executarRegistro", method=RequestMethod.POST)
public String executarRegistro(@Valid Usuario usuario,
    BindingResult bindingResult, HttpSession sessao,
    @RequestParam(value="avatar", required=false) MultipartFile avatar){

```

```
if (bindingResult.hasErrors()) {
    Map<String, Object> model = new HashMap<String, Object>();
    model.put("usuario", usuario);
    return registro(model);
}
getDaoUsuario().persistir(usuario);

if (! avatar.isEmpty()) {
    processarAvatar(usuario, avatar);
}

sessao.setAttribute("usuario", usuario);
return "redirect:/";
}
```

A modificação mais importante é a inclusão do parâmetro *avatar*, do tipo `org.springframework.web.multipart.MultipartFile`, responsável por representar o arquivo recebido pelo controlador na requisição. Como podemos ver na implementação desta action, o único método desta classe que nos interessa é `isEmpty()`, que retorna `true` caso não tenha sido transmitido junto com a requisição um arquivo anexado.

Outro método foi incluído em nosso controlador apenas para lidar com o arquivo recebido. Trata-se de `processarAvatar`, cuja implementação resumida pode ser vista a seguir:

```
private void processarAvatar(Usuario usuario, MultipartFile avatar) {
    byte[] conteudo = avatar.getBytes();
    persistirConteudo(usuario, conteudo);
}
```

Para obter o conteúdo do arquivo recebido, executa-se o método `getBytes` do objeto `MultipartFile`, que retorna um `byte[]`, que é o conteúdo recebido a partir da submissão do formulário de cadastro.

8.3 DEFINA O CORPO DA RESPOSTA

Até este momento, nossas ações retornaram ou o nome da visualização a ser renderizada ou um objeto do tipo `ModelAndView`. Como uma action retornaria o conteúdo dos arquivos que o *Spring Fórum* agora está habilitado a receber?

Antes de responder a esta pergunta, nosso primeiro passo será evoluir a página de visualização de detalhes do usuário, para que fique similar à imagem a seguir:



Figura 8.3: Expondo o avatar dos usuários

Tudo o que precisamos fazer é definir o atributo `src` da tag `` apontando para a ação responsável por retornar um *array de bytes* representando o conteúdo do avatar, tal como fizemos no trecho a seguir:

```
<img class="avatar"
      src='<c:url value="/usuario/avatar/${usuario.login}"/>' />
```

Usamos a tag `<c:url>` do JSTL para garantir que a URL alvo esteja sempre de acordo com o contexto da aplicação. Cabe então o último ponto desta sessão que é a nossa action, que encontra-se implementada na classe `UsuarioController`, cujo código fonte podemos ver listado a seguir:

```
@RequestMapping("/usuario/avatar/{login}")
@ResponseBody
public byte[] avatar(@PathVariable("login") String login) {
    /*
        Obtém um array de bytes representando o
```

```
        avatar relacionado ao login
    */
    return obterConteudo(login);
}
```

A primeira novidade que podemos ver na implementação da action, diz respeito ao valor passado à anotação `@RequestMapping`. Trata-se da variável `login` na URL, o que nos permite receber mapear URLs como as abaixo para esta action:

```
http://localhost:8080/spring-forum/usuario/avatar/kicolobo
```

```
http://localhost:8080/spring-forum/usuario/avatar/mccarty
```

Outro ponto de destaque é a inclusão da anotação `@ResponseBody`. Repare que nossa action retorna um *array de bytes* e não uma `String` ou um objeto `ModelAndView` como fizemos anteriormente. A presença desta anotação instrui o *Spring MVC* a enviar o valor de retorno diretamente ao navegador do usuário.

8.4 FAÇA REQUISIÇÕES ASSÍNCRONAS COM AJAX

Atualmente presente na caixa de ferramentas de todo desenvolvedor web, é importante compreendermos como usar Ajax no *Spring MVC*.

Infelizmente, na versão atual do framework ainda não há uma biblioteca de tags que facilite a vida do desenvolvedor na escrita da camada de visualização, porém esta ausência não deve ser vista como uma grave limitação do framework dado a difusão de bibliotecas JavaScript como jQuery ou Dojo, que suprem a esmagadora maioria das necessidade dos programadores neste quesito.

Para os não iniciados, AJAX é um acrônimo para *Asynchronous Javascript and XML*. A técnica foi documentada pela primeira vez por Jesse James Garrett, no ano de 2005 em seu blog [6] e se consagrou ao ser aplicada na implementação do Google Maps. Não é exagero dizer que a técnica se tornou *língua franca* entre os desenvolvedores web.

Os fundamentos por trás da técnica vão além do objetivo deste livro, porém o leitor ainda não familiarizado com o conceito pode ter acesso a material de melhor qualidade sobre o assunto no post de Garrett [6] acima citado ou em livros inteiramente dedicados ao assunto como por exemplo *Use a Cabeça - AJAX*, de Brett McLaughlin [12].

Resumidamente, AJAX é usado no desenvolvimento de aplicações para dois objetivos: renderizar trechos específicos de nossas páginas ou na obtenção de informações que serão processadas do lado cliente, ou mesmo na mistura destes dois usos.

Renderização local

A técnica de renderização local (ou *in loco*) consiste na execução de uma consulta assíncrona ao servidor, cuja resposta será renderizada em algum local pré-definido da página, por exemplo dentro de uma tag `<div>` com identificador específico.

No *Spring Fórum* esta técnica foi aplicada na página que exibe os detalhes do usuário. Quando o link *Expor posts* é clicado, uma requisição assíncrona, disparada através do *jQuery*, é feita ao servidor. Este executa a action relacionada como se fosse uma requisição convencional e retorna ao navegador um trecho em HTML que simplesmente é exposto na div que anteriormente estava vazia, conforme pode ser visto na imagem a seguir que esquematiza seu funcionamento.

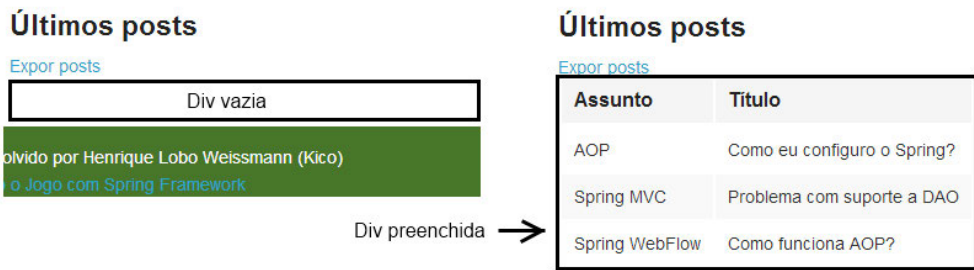


Figura 8.4: Renderização in loco

Do lado servidor, a action é exatamente como todas as outras que vimos anteriormente. Esta apenas busca todos os tópicos criados pelo usuário, cujo login é passado como parâmetro. Encontra-se implementada na classe `UsuarioController` conforme pode ser visto no código a seguir:

```
@RequestMapping("/usuario/posts/{login}")
public String topicosUsuario(@PathVariable("login") String login,
                             Map<String, Object> model) {
    model.put("topicos",
        getDaoTopico().
            getTopicosPorAutor(getDaoUsuario().getUsuario(login)));
    return "usuario/posts";
}
```

Se o leitor bem se lembra, no *Spring Fórum* é usado o Tiles como camada de visualização. Como nosso objetivo é apenas renderizar um trecho HTML, a definição

de template incluída no arquivo `tiles-config.xml` pode ser vista a seguir. Não há componentes a serem incluídos em nosso template, apenas este já é necessário para o que precisamos.

```
<definition name="usuario/posts"
            template="/views/usuario/postsUsuario.jsp"/>
```

E o trecho HTML inserido em nossa página também não poderia ser mais simples, conforme conferimos no trecho abaixo:

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<table>
    <thead>
        <th>Assunto</th>
        <th>Título</th>
    </thead>
    <tbody>
        <c:forEach items="${topicos}" var="post">
            <tr>
                <td>${post.assunto.nome}</td>
                <td>${post.titulo}</td>
            </tr>
        </c:forEach>
    </tbody>
</table>
```

Retorne o JSON

A outra forma de aplicarmos Ajax consiste em enviar conteúdo no formato JSON para o usuário, o que é assustadoramente simples com *Spring MVC*. Nativamente, o framework não oferece suporte a este formato, sendo assim nosso primeiro passo consistirá em incluir a biblioteca *Codehaus Jackson* ao classpath de nosso projeto. Basta incluir a configuração abaixo na lista de dependências do arquivo `pom.xml` do projeto.

```
<dependency>
    <groupId>org.codehaus.jackson</groupId>
    <artifactId>jackson-mapper-asl</artifactId>
    <version>1.7.1</version>
</dependency>
```

Para exemplificar como retornar JSON ao usuário, a mesma action exposta no exemplo anterior foi reescrita, conforme pode ser visto no código a seguir:

```
@RequestMapping("/usuario/postsJSON/{login}")
@ResponseBody
public List<Topico>
    topicosUsuarioJson(@PathVariable("login") String login) {
    Usuario usuario = getDaoUsuario().getUsuario(login);
    return getDaoTopico().getTopicosPorAutor(usuario);
}
```

Não há novidades no código acima. Incluindo a anotação `@ResponseBody`, uma vez que nossa action seja executada, o resultado será automaticamente convertido para JSON pelo *Jackson*. Ao acessar a url `http://localhost:8080/spring-forum/usuario/postsJSON/kicolobo` a seguinte saída obtida é exposta abaixo:

```
[{"titulo":"Spring MVC", "assunto":{"nome":"Web"}},
 {"titulo":"Como eu configuro o Spring?", "assunto":{"nome":"IoC"}}]
```

8.5 CONCLUINDO COM UMA SURPRESA

Ao longo deste capítulo, conhecemos a solução oferecida pela *SpringSource* para o desenvolvimento de aplicações web. Como pode ser visto, trata-se de uma solução incrivelmente flexível, que possibilita ao programador customizar cada aspecto do seu funcionamento. Finalizado este capítulo o leitor talvez se sinta assustado com a quantidade de configurações que precisamos alterar durante este capítulo conforme novos recursos eram apresentados.

Acontece que os usuários do *SpringSource Tool Suite* não precisam definir estas configurações manualmente. Na realidade, a IDE fornece a possibilidade de criar um projeto baseado em *Spring MVC* já pré-configurado. Para tal, basta acessar o menu *File > New > Spring Template Project*. Será exposta a janela a seguir, aonde tudo o que o usuário deve fazer é selecionar a opção *Spring MVC Project* e voilà, o projeto está pré-configurado e pronto para uso.

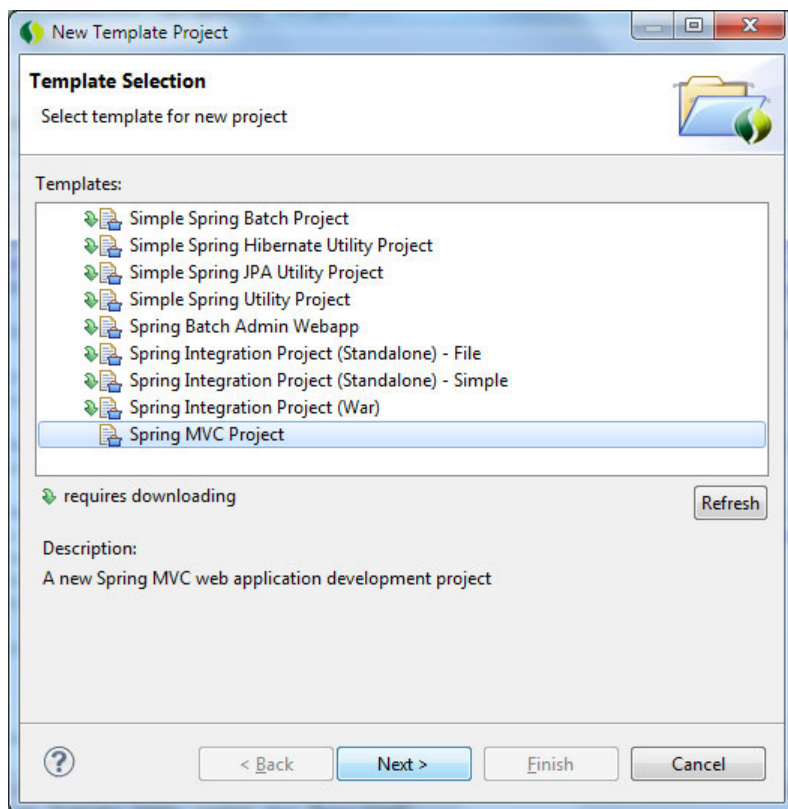


Figura 8.5: SpringSource Tool Suite

Um ponto importante a ser mencionado a respeito do aparente excesso de configurações é o fato destas atividades, mesmo com o auxílio da IDE, são executadas normalmente apenas no momento de criação do projeto. A partir deste momento, o fluxo básico consistirá basicamente apenas em implementar os controladores e templates de visualização relacionados, associados às técnicas apresentadas no capítulo 4 deste livro fornecem à equipe de desenvolvimento um framework de altíssima produtividade, com eficácia comprovada em inúmeros projetos pelo mundo [15].

A história do *Spring MVC* não termina por agora. Lacunas, como por exemplo a ausência de bibliotecas de tag que facilitem a vida do desenvolvedor que precise lidar com AJAX estão em desenvolvimento. Além disto, de forma bastante discreta o *MVC* vive uma vida dupla sob a forma do *Grails*, um framework para desenvolvimento de aplicações web baseado na linguagem Groovy tendo como base o *MVC*, que tem

se mostrado uma das formas mais produtivas de se criar aplicações web dentro da plataforma Java EE.

Mas já falamos demais sobre o *MVC*. Nos próximos capítulos nosso foco será nos ganhos que o framework oferece na camada de negócios.

CAPÍTULO 9

Acesso a dados

Difícilmente encontramos uma aplicação que não precise lidar com algum tipo de banco de dados. Apesar deste tipo de interação fazer parte do cotidiano do desenvolvedor, sua implementação possui uma série de dificuldades inerentes de sua própria natureza, que consomem uma boa parte das horas de desenvolvimento em um projeto. O Spring oferece um rico suporte nativo às tecnologias de acesso a dados mais populares atualmente como JDBC, Hibernate e JPA.

Como veremos, este suporte, assim como tudo no Spring, baseia-se na aplicação de boas práticas e padrões de projeto de eficácia comprovada. O mais interessante é o fato de que, apesar de lidarmos neste capítulo de tecnologias distintas como JDBC e Hibernate, o modo de trabalho é praticamente o mesmo, facilitando assim a vida do desenvolvedor caso precise executar uma migração tecnológica em algum momento.

Neste capítulo abordaremos as três tecnologias com suporte nativo do framework mais populares na escrita de aplicações: JDBC, Hibernate e JPA e, no decorrer destas, serão apresentadas as soluções oferecidas pelo Spring aos problemas que costumam assombrar os desenvolvedores na escrita e manutenção da camada

de persistência.

9.1 DAO: NO CENTRO DA MENTALIDADE SPRING

Ao optarmos pelo Spring estamos sempre de uma forma ou outra lidando com o problema do alto acoplamento. No que diz respeito à escrita da camada de persistência, vemos como conceito central o DAO - *Data Access Object* - que abordamos brevemente no primeiro capítulo e trataremos com maior riqueza de detalhes agora.

Este é um padrão de projeto cujo principal objetivo é isolar o máximo possível o código responsável por interagir com nossas bases de dados do restante do sistema. O DAO é um objeto que oferece às classes clientes uma interface abstrata aonde são expostas as operações básicas que possuem a finalidade de alterar ou pesquisar a base de dados.

A camada de persistência do projeto *Spring Fórum* adota o padrão DAO e na imagem a seguir podemos ver um exemplo de como o padrão se encontra implementado. `DAOAssunto` é a interface abstrata visível às classes clientes do sistema responsável por interagir com tudo em nosso sistema que diga respeito a objetos do tipo `Assunto`. À nossa classe cliente `HomeController` não interessa saber se usamos JDBC, Hibernate ou alguma base de dados *NoSQL*: tudo o que lhe interessa é que seja possível listar todos os assuntos possíveis através do método `list()`.

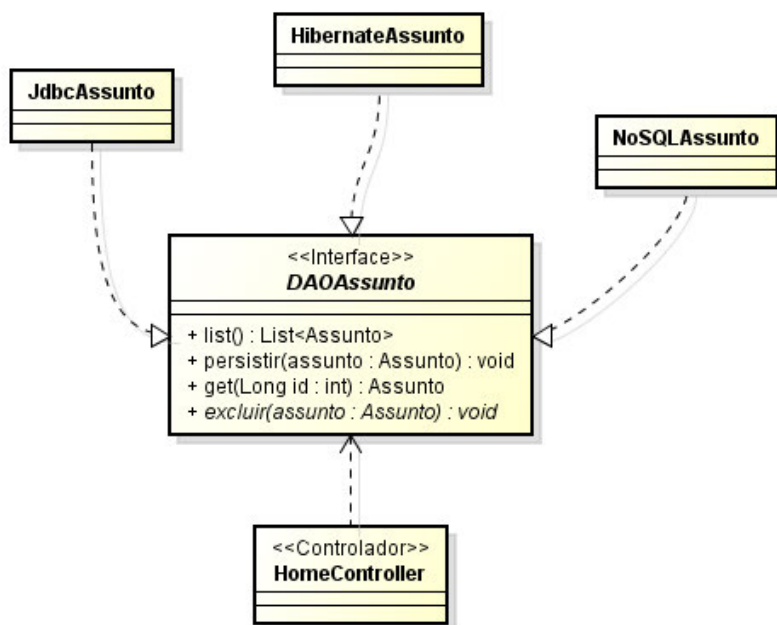


Figura 9.1: Parte da camada de persistência do Spring Fórum

O leitor já deve ter observado a repetição de um padrão: novamente vemos os ganhos vindos da aplicação da injeção de dependências/inversão de controle, que permitem ao desenvolvedor programar voltado para interfaces e não para implementações.

Guarde bem este padrão, pois todo o suporte oferecido pelo Spring às tecnologias de acesso a dados baseia-se neste. Mas antes de mergulharmos nas facilidades oferecidas pelo Spring na adoção desta prática, vamos descobrir como o Spring lida com um problema fundamental a todo DAO.

9.2 CONECTE-SE AO BANCO DE DADOS

Todo DAO precisa de uma fonte de dados (*data source*). As três tecnologias de acesso a dados abordadas neste capítulo têm como fonte de dados um objeto que implementa a interface `javax.sql.DataSource` que é responsável pela obtenção de objetos do tipo `java.sql.Connection` através da invocação do `getConnection`. No caso do Spring, tudo o que precisamos fazer é declarar um bean. A principal decisão a ser tomada pelo desenvolvedor diz respeito a qual implementação usar.

Há três formas de obter uma fonte de dados no Spring: usando JNDI, criando uma fonte de dados baseada em drivers ou através de um pool de conexões. Na preparação de nosso ambiente faz-se necessária a inclusão de uma nova dependência. No caso, estamos falando do módulo de suporte a JDBC do Spring, cuja dependência no formato Maven é listada a seguir:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>3.1.2.RELEASE</version>
</dependency>
```

JNDI

Talvez seja a forma mais interessante de se obter uma fonte de dados. Aos que não conhecem, JNDI (*Java Naming and Directory Interface*) é uma API presente na plataforma Java EE que permite ao desenvolvedor obter objetos ou recursos através de um nome. Caso sua aplicação seja executada em um servidor de aplicações, é grande a probabilidade deste oferecer suporte à criação de fontes de dados disponibilizadas às aplicações hospedadas usando esta API.

Há duas grandes vantagens no JNDI. A primeira vem do fato de se tratar de uma configuração externa à aplicação, o que possibilita ao desenvolvedor alterar seus parâmetros sem a necessidade de que esta seja recompilada ou instalada novamente no servidor. Dependendo do setup do servidor e da aplicação, muitas vezes o desenvolvedor sequer precisa reiniciar o sistema para que as alterações surtam efeito.

Há duas formas de se obter um data source desta maneira. O caminho mais fácil se dá através do namespace `jee`, que pode ser declarado tal como no trecho a seguir:

```
<beans default-autowire="byType"
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jee="http://www.springframework.org/schema/jee"
  xsi:schemaLocation="http://www.springframework.org/schema/jee
    http://www.springframework.org/schema/jee/spring-jee-3.1.xsd">
  <!-- Restante do arquivo oculto para facilitar a leitura -->
</beans>
```

A tag que nos interessa é `<jee:jndi-lookup/>`, que obtém um recurso qualquer pelo JNDI. Seu uso é muito simples como pode ser visto no código a seguir:

```
<!-- Obtém uma fonte de dados identificada como 'dataSource' -->  
<jee:jndi-lookupid="dataSource"  
    id="dataSource"  
    jndi-name="/jdbc/SpringForumDS"/>
```

Esta é usada para obter qualquer recurso JNDI. Seu único atributo obrigatório é `jndi-name`, que indica o nome do recurso a ser buscado. O segundo atributo, `resource-ref`, quando não fornecido, indica que o nome do recurso será buscado com base apenas no valor definido em `jndi-name`.

Outro atributo interessante nesta tag é `cache`, cujo valor default é `false`. Definindo-o como `true`, o recurso será cacheado evitando assim novas consultas ao diretório JNDI, o que pode propiciar um ganho de performance à aplicação. O único ponto negativo de definir este atributo como verdadeiro é o fato de que assim perdemos a possibilidade de alterarmos as configurações de nossas fontes de dados sem a necessidade de reiniciarmos a aplicação.

Como JNDI normalmente é usado em servidores de aplicação, outro parâmetro que deve ser levado em consideração é `resource-ref` que, caso omitido, tem seu valor padrão como `true`. Por consequência, o identificador JNDI receberá sempre o prefixo `java:comp/env/`, valor padrão usado por servidores de aplicação, minimizando assim a quantidade de configuração a ser digitada.

Cada servidor de aplicações/servlet possui modos distintos de configurar recursos JNDI, o que fugiria do escopo deste livro. É bom ficar atento à documentação oficial do ambiente que você estiver utilizando.

Fonte de dados baseada em driver

Se JNDI é discutivelmente a melhor opção de fonte de dados para produção, há também a pior que é a baseada em driver. Por padrão o Spring vem com uma implementação chamada `DriverManagerDataSource`, que é a mais simples possível, que sempre cria uma nova conexão com o banco de dados quando necessária. Não é recomendável para ser usada no ambiente de produção por ser um fato conhecido que a obtenção de conexões com o banco de dados é uma atividade bastante cara do ponto de vista computacional.

Se é uma opção tão ruim assim, por que incluí-la no Spring? Por que é uma solução bastante interessante para casos mais simples, como a escrita de testes integrados ou aplicações mais simples nas quais o preço a se pagar pela obtenção de uma conexão com o banco de dados compense. A definição deste bean dentro do arquivo de configuração é bastante simples, como podemos ver no exemplo listado a seguir:

```
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="username" value="springbrasil"/>
    <property name="password" value="springbrasil"/>
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url"
        value="jdbc:mysql://localhost:3306/springbrasil"/>
</bean>
```

O desenvolvedor precisa definir apenas quatro parâmetros: `username` e `password` para autenticar-se ao servidor, `driverClassName` definindo o driver responsável pela obtenção de conexões com o banco de dados e, finalmente, a URL de conexão, cujo valor irá variar de acordo com o sistema gerenciador de banco de dados escolhido (no nosso exemplo usamos o MySQL).

Há uma outra variante desta implementação que obtém uma única conexão com o banco de dados. É ainda menos recomendada que `DriverManagerDataSource` pelo fato de poder causar problemas em aplicações multi-threaded. Sua configuração é praticamente igual à da versão anterior, variando-se apenas o nome da classe a ser instanciada:

```
<bean id="dataSource"
    class=
    "org.springframework.jdbc.datasource.SingleConnectionDataSource">
    <property name="username" value="springbrasil"/>
    <property name="password" value="springbrasil"/>
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url"
        value="jdbc:mysql://localhost:3306/springbrasil"/>
</bean>
```

Definindo um pool de conexões

Se a configuração via JNDI estiver fora de mão e o leitor ouviu meu conselho de não usar uma fonte de dados baseada em driver, uma opção intermediária é adotar alguma implementação de pool de conexões. Um pool consiste em uma estratégia de programação que nos permite obter ganhos significativos de performance, minimizando nossa necessidade de instanciar objetos cuja construção seja cara.

A estratégia é simples: ao invés de a cada requisição criar um novo objeto, criar mais de uma instância de uma vez, liberando-as para os objetos clientes conforme estas sejam requisitadas. Quando a instância não for mais necessária, a classe cliente

a devolve para o pool aonde será reaproveitada por outra classe cliente, caso seja requisitada. Para a plataforma Java, há diversas implementações disponíveis, como o C3Po (<http://www.mchange.com/projects/c3po/>), DBCP, usado pelo Tomcat (<http://commons.apache.org/dbcp/>) e muitas outras.

Estas implementações possuem basicamente as mesmas configurações. Apenas para ilustrar, será exposto a seguir como usar o C3Po. O primeiro passo é incluí-lo no classpath do projeto. Basta adicionar sua dependência no arquivo `pom.xml` do projeto:

```
<dependency>
  <groupId>c3p0</groupId>
  <artifactId>c3p0</artifactId>
  <version>0.9.1.2</version>
</dependency>
```

Finalmente, basta declarar o bean tal como no código a seguir. Sua configuração é bastante similar à que precisamos fornecer para `DriverManagerDataSource`.

```
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
  destroy-method="close">
  <property name="driverClass" value="com.mysql.jdbc.Driver"/>
  <property name="user" value="sf"/>
  <property name="password" value="sf"/>
  <property name="jdbcUrl"
    value="jdbc:mysql://localhost:3306/spring_forum"/>

  <property name="maxPoolSize" value="10"/>
  <property name="acquireIncrement" value="1"/>
  <property name="maxIdleTime" value="120"/>
  <property name="acquireRetryAttempts" value="10"/>
  <property name="initialPoolSize" value="10"/>
</bean>
```

As primeiras quatro propriedades equivalem respectivamente à classe do driver do nosso banco de dados, dados de autenticação sob a forma de nome do usuário e senha e, finalmente, a URL de conexão. A novidade está por conta das cinco propriedades seguintes, cuja descrição detalhada podemos ver a seguir:

- `maxPoolSize`: define o número máximo de conexões a ser armazenado no pool.

- `acquireIncrement`: a cada requisição por uma conexão, define quantas novas conexões devem ser incluídas no pool caso este não possua mais conexões disponíveis até atingir o número máximo definido em `maxPoolSize`.
- `maxIdleTime`: define o tempo máximo em segundos que uma conexão não usada deve ser armazenada no pool.
- `acquireRetryAttempts`: define o número de tentativas na obtenção de uma nova conexão por parte do pool.
- `initialPoolSize`: define o número inicial de conexões que devem existir quando a primeira conexão for solicitada por alguma classe cliente.

Atenção especial deve ser dada ao atributo `destroy-method`, incluído na definição do bean. O método `close` fechará todas as conexões abertas pelo pool quando seu bean for finalizado pelo container.

9.3 DE VOLTA AO DAO: O PROBLEMA COM AS EXCEÇÕES

A exceção `SQLException` em um primeiro momento parece uma solução bastante elegante adotada pela equipe de desenvolvimento do JDBC. Infelizmente toda esta elegância cai do salto a partir do momento em que pensamos nas situações em que esta exceção é disparada:

- Quando não conseguimos obter uma conexão com o banco de dados;
- Comando SQL mal formado;
- Falha de conexão com o servidor;
- Violação de integridade;
- Problemas na liberação de recursos;
- Deadlocks;
- E qualquer outro problema decorrente do acesso a dados.

Como pode ser visto, é uma exceção aplicável a qualquer situação que fuja da execução normal de nosso programa. Se quisermos dar algum tratamento especial, temos de lidar com códigos de erro específicos do desenvolvedor do driver/banco de dados, tal como no exemplo a seguir:

```
try {  
    //faço algo que pode disparar uma SQLException  
} catch (SQLException ex) {  
    switch (ex.getErrorCode()) {  
        case 1022:  
            // chave duplicada no MySQL  
            break;  
        case 1037:  
            // falta de memória no MySQL  
            break;  
    }  
}
```

De repente `SQLException` se mostrou manca também, pois além de ser excessivamente genérica, ainda nos obriga a escrever código voltado para um fornecedor específico, jogando no lixo toda aquela história de portabilidade usada para vender o Java.

A situação melhora um pouco com o Hibernate, que possui uma hierarquia de exceções bem mais rica e nos possibilita tratar erros com maior granularidade. Há exceções como `QueryException`, `OptimisticLockException`, `ObjectNotFoundException` e muitas outras.

Em um primeiro momento temos a impressão de que nosso problema foi resolvido, principalmente porque não são exceções checadas e, portanto, não precisam ser incluídas nas declarações de métodos das nossas interfaces. Mas e se no futuro precisarmos por alguma razão voltarmos para o JDBC em um ponto ou outro da aplicação? Nossas classes cliente já estarão infectadas com referências a classes específicas do Hibernate. Pior ainda: e se estivermos implementando um componente reaproveitável?

Seria ótimo se houvesse uma hierarquia de exceções que fosse agnóstica em relação ao mecanismo de persistência, não é mesmo? Algo que vá além da deselegância de nossa amiga `SQLException` mas que também não nos amarrasse a algum ORM específico como Hibernate. Esta hierarquia existe e é fornecida pelo Spring.

No diagrama abaixo podemos ver quão abrangente e genérica esta é, abordando os principais problemas enfrentados pelos desenvolvedores. E sabe o que mais? Todas tem como base `DataAccessException`, que não é checada e portanto não precisa ser incluída na declaração `throws` de nossa interface.

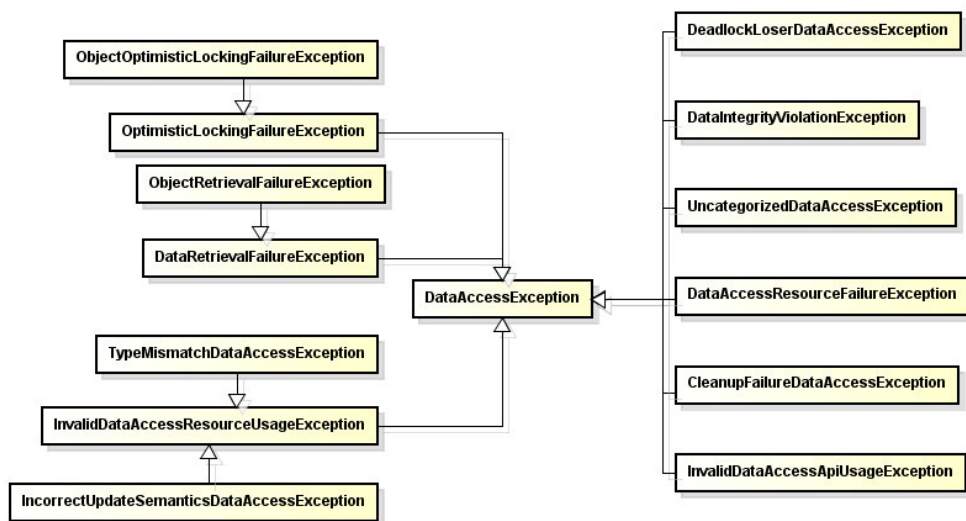


Figura 9.2: Hierarquia de exceções de acesso a dados do Spring

Internamente o Spring possui um mecanismo de tradução de exceções: há um para cada tecnologia suportada pelo Spring. Sendo assim, encontramos classes como `SQLExceptionTranslator`, `HibernateExceptionTranslator` e uma série de *[sua tecnologia de acesso favorita]*`ExceptionTranslators` à disposição do desenvolvedor. O mais interessante é que não precisamos lidar com estas classes diretamente: basta usarmos os templates fornecidos pelo Spring.

9.4 TEMPLATES: ACABE COM O CÓDIGO REPETIDO

Já reparou que em diversas situações repetimos sempre o mesmo script? Tome uma atualização de dados ou consulta JDBC por exemplo. Temos um algoritmo que sempre segue a mesma sequência de passos:

- 1) Obtenha uma conexão;
- 2) Inicie uma transação;
- 3) Execute uma consulta ou atualização de dados;
- 4) Comite a transação ou execute o rollback caso algo dê errado;

5) Libere recursos fechando a conexão.

Sempre precisamos executar na ordem acima os dois primeiros e últimos passos do algoritmo. Um template é um padrão de projeto que se popularizou com o livro *Padrões de Projeto* [4]. Seu objetivo é delegar para uma subclasse um callback a parte do algoritmo cuja execução seja customizada, mantendo assim intactos os passos do algoritmo que normalmente não sofrem modificações. Observando a imagem a seguir, fica claro que os templates a que me refiro não são muito diferentes do *Tiles* que vimos no capítulo 7: a diferença é que ao invés de lidarmos com marcação HTML/Javascript estaremos lidando com código Java.

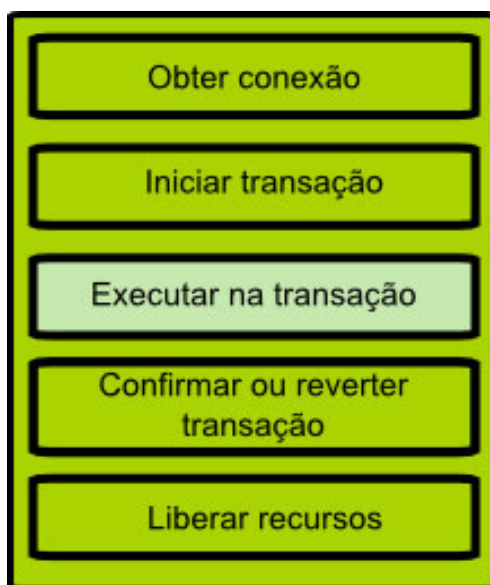


Figura 9.3: Exemplo de template

Templates desempenham um papel fundamental dentro do suporte a dados oferecido pelo Spring. Cada tecnologia suportada pelo framework possui seu próprio template, responsável por encapsular os passos imutáveis do algoritmo, executarem a transformação das exceções disparadas e reduzir a quantidade de código que o desenvolvedor precisa escrever.

Nossa compreensão deste padrão se tornará mais clara na prática, sendo assim convém tratarmos da API básica de persistência que o Spring suporta.

9.5 O TRABALHO COM JDBC

Talvez o JDBC seja o mecanismo de persistência que mais receba melhorias a partir do Spring. Talvez não seja exagero dizer que o framework revitaliza esta API com seus templates e classes de suporte de tal forma que o desenvolvedor possa sentir no futuro um forte mal estar ao usá-la sem o Spring. Com ferramentas de mapeamento tão avançadas como Hibernate, Toplink ou JPA muitos se questionam se realmente vale à pena usarmos JDBC diretamente para manipularmos nossas bases de dados relacionais. A verdade é que há diversos casos nos quais esta é a melhor opção:

- A equipe não precisa aprender outra API de acesso, apenas JDBC.
- Você precisa ter acesso rápido a funcionalidades específicas do seu gerenciador de banco de dados.
- É necessário tunar as consultas ou instruções de manipulação com granularidade máxima em seu projeto.
- Performance: apesar dos ORMs já otimizarem as consultas feitas ao banco de dados, sempre há um caso ou outro que a ferramenta não consegue alcançar. E nestes casos JDBC cai como uma luva.
- Você simplesmente prefere JDBC.

Quando lidamos com JDBC fica clara a quantidade de código duplicado que precisamos escrever. Como exemplo, veja o código abaixo responsável por listar todos os assuntos do projeto *Spring Fórum* usando a abordagem tradicional:

```
class JdbcAssunto implements DAOAssunto {
    public List<Assunto> listJDBC(int offset, int max) {
        List<Assunto> resultado = new ArrayList<Assunto>();
        Connection conexao = null;
        try {
            conexao = getDataSource().getConnection();
            PreparedStatement stmt =
                conexao.prepareStatement
                    ("select id, nome from assunto limit ? offset ?");
            stmt.setInt(1, max);
            stmt.setInt(2, offset);
            ResultSet rs = stmt.executeQuery();
            while (rs.next()) {
```

```
        resultado.add(
            new Assunto(rs.getLong(1), rs.getString(2)));
    }
} catch (SQLException ex) {
    // tratamento
} finally {
    if (conexao != null) {
        try {
            conexao.close();
        } catch (SQLException ex) {
            // trate o erro
        }
    }
}
return resultado;
}
}
```

E o código usado para persistir um assunto não é muito diferente, conforme podemos ver a seguir:

```
public void persistirJDBC(Assunto assunto) {
    Connection conexao = null;
    try {
        conexao = getConnection();
        conexao.setAutoCommit(false);
        PreparedStatement stmt =
            conexao.prepareStatement("insert into assunto (nome) values (?)");
        stmt.setString(1, assunto.getNome());
        stmt.executeUpdate();
        conexao.commit();
    } catch (SQLException ex) {
        try {
            conexao.rollback();
        } catch (SQLException ex2) {
            // não deu pra fazer rollback
        }
    } finally {
        if (conexao != null) {
            try {
                conexao.close();
            } catch (SQLException ex2) {

```

```
        // não consegui liberar o recurso
    }
}
}
```

Vemos nitidamente que há um código replicado que “emoldura” o que realmente interessa em cada método e que ocupa a maior parte do código que precisamos escrever. Entram em cena os templates oferecidos pelo Spring que facilitam nossa vida.

Templates JDBC

No desenvolver do Spring, três tipos de template surgiram para dar suporte ao JDBC. No centro encontra-se `JdbcTemplate`, que oferece todas as funcionalidades básicas oferecidas pelo framework. `NamedParameterJdbcTemplate` que nos oferece facilidades ao lidar com consultas que possuam parâmetros nomeados e `SimpleJdbcTemplate`, que por algum tempo foi visto como o caminho a ser seguido pelos desenvolvedores.

A história por trás do `SimpleJdbcTemplate` é curiosa. Esta classe aparece na versão 2.0 do Spring como uma conveniência que basicamente encapsula um objeto do tipo `JdbcTemplate` e provê uma API mais simples de ser usada por ser baseada na nova sintaxe oferecida pelo Java 5. A grosso modo, chamadas a esta API eram transformadas internamente e repassadas à instância `JdbcTemplate` para serem processadas.

Até a versão 3.0 do Spring era vista como a implementação que deveria ser adotada pelos desenvolvedores, visto que versões anteriores ao Java 5 não seriam mais suportadas pelo Spring no futuro e também por possuir suporte a parâmetros nomeados. Na terceira edição do livro *Spring em Ação* [16] o autor chega a dizer que as duas outras opções sequer precisam ser usadas, pelo fato de `SimpleJdbcTemplate` oferecer todos os seus recursos. Todo este prestígio acaba com a versão 3.1 do Spring, que passa a considerá-lo uma classe depreciada. Como?

Isto mesmo: olhando com maior atenção à classe `JdbcTemplate`, os desenvolvedores por trás do Spring perceberam que, pelo fato desta já possuir suporte ao Java 5 e estar no centro do suporte ao JDBC, não há razão em se manter três tipos de template.

O uso do JdbcTemplate

Com JdbcTemplate, o programador não precisa se preocupar com obtenção e liberação de recursos, pois estes já são implementados pelo template. Precisamos nos preocupar somente com o que realmente importa, que é implementação de nossos comandos SQL e extração dos resultados. Além disto, esta classe também possui a função de transformar exceções específicas do JDBC para a hierarquia oferecida pelo Spring que vimos no início deste capítulo.

Uma vantagem de JdbcTemplate é o fato desta classe ser *thread safe* após configurada, o que nos possibilita ter uma única instância desta compartilhada por todos os nossos beans que precisem de seus serviços. Há casos em que uma nova instância só é criada caso seja necessário lidar com fontes de dados distintas.

Por falar em instanciação, a sua é bastante simples: basta usar seu construtor que receba como parâmetro um objeto do tipo `javax.sql.DataSource` ou então simplesmente injetar uma fonte de dados por setter. Podemos ver no exemplo a seguir como declará-la em um arquivo de configuração XML:

```
<!-- por setter -->
<bean id="jdbcTemplatePorSetter"
      class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"/>
</bean>

<!-- por construtor -->
<bean id="jdbcTemplatePorConstrutor"
      class="org.springframework.jdbc.core.JdbcTemplate">
    <constructor-arg ref="dataSource"/>
</bean>

<!-- ou por auto wiring -->
<bean id="jdbcTemplateAutoWired"
      class="org.springframework.jdbc.core.JdbcTemplate"
      autowire="byType"/>
```

Normalmente, JdbcTemplate é injetado em nossos DAOs que o usam como base para todas as suas operações. Surgiu inclusive um padrão para este tipo de classe. Neste padrão, que ainda não possui um nome, uma nova instância do template é gerada no momento em que injetamos um objeto do tipo DataSource. Podemos ver um exemplo de implementação deste padrão no código a seguir:

```
public abstract class BaseJdbcDAO {
    private JdbcTemplate jdbcTemplate;
```



```
@Autowired
public void setDataSource(DataSource ds) {
    /*
        Não precisamos de uma instância real do DataSource
        se todas as nossas operações serão feitas usando o
        JdbcTemplate, certo?
    */
    jdbcTemplate = new JdbcTemplate(ds);
}
}
```

Este padrão é tão comum que no Spring já vêm uma implementação bastante similar ao exemplo exposto anteriormente. Trata-se da classe `org.springframework.jdbc.core.support.JdbcDaoSupport`. Não é necessário implementar o código descrito acima para encapsular seu `JdbcTemplate`, basta que suas classes estendam `JdbcDaoSupport`.

Consultando com JdbcTemplate

Como mencionado, ao lidarmos com os templates do Spring, só precisamos nos preocupar com a definição de nossas consultas e na extração de dados. `JdbcTemplate` nos oferece uma série de métodos incrivelmente úteis quando precisamos buscar apenas um campo de nossas consultas. Abaixo podemos ver um exemplo extraído do *Spring Fórum* que pode chocar o leitor acostumado ao modo de trabalho tradicional JDBC:

```
public class JdbcAssunto extends JdbcDaoSupport {
    public int getTotalAssuntos() {
        return getJdbcTemplate()
            .queryForInt("select count(*) from assunto");
    }
}
```

Uma linha de código! O ganho de produtividade se mostra mais evidente quando comparamos o exemplo acima com uma implementação mais tradicional como a seguir em que foi necessário escrever no mínimo 17:

```
public int getTotalAssuntosJDBC() {
    Connection conexao = null;
    ResultSet rs = null;
```

```
try {
    conexao = .getConnection();
    rs = conexao.createStatement()
        .executeQuery("select count(*) from assunto");
    rs.next();
    return rs.getInt(1);
} catch (SQLException ex) {
    // trato o erro de alguma forma
} finally {
    try {
        if (rs != null) rs.close();
        if (conexao != null) conexao.close();
    } catch (SQLException ex) {
        // trata o erro
    }
}
```

Para consultas nas quais pretendemos obter apenas um campo `JdbcTemplate`, o desenvolvedor não precisa se preocupar com a extração dos dados, visto que o próprio template já executa este trabalho para nós. Há uma série de funções para este fim implementadas em `JdbcTemplate`, cujas mais populares podemos ver na lista a seguir, que sempre possuem duas assinaturas: uma que contém apenas a consulta, e outra que contém, além da consulta, uma lista no formato *varargs* do Java 5. Dentre estas destacam-se `queryForInt`, `queryForLong`, `queryForString`, `queryForObject` e outras, cujo tipo retornado já se encontra descrito no próprio nome da função.

Buscar um único valor passando parâmetros é também igualmente simples, tal como pode ser visto no exemplo a seguir:

```
public long getIdUsuario(String login, String nome) {
    getJdbcTemplate()
        .queryForLong("select id from usuario where login = ? and nome = ?",
            login, nome);
}
```

Na prática sabemos que os dois exemplos acima são casos raros. É muito mais comum precisarmos que nossas consultas nos retornem entidades totalmente preenchidas (daí a popularidade das ferramentas ORM). Este processo de obter os dados a partir da consulta e em seguida transformá-los em objetos complexos é o que chamamos de extração. Para ilustrar o processo iremos mais uma vez recorrer ao projeto

Spring Fórum. O código a seguir retorna uma lista com os assuntos presentes no banco de dados usando paginação:

```
public List<Assunto> list(int offset, int max) {
    Integer[] parametros = {max, offset};
    return getJdbcTemplate().query(
        "select id, nome from assunto limit ? offset ?",
        parametros,
        new RowMapper<Assunto>(){

        public Assunto mapRow(ResultSet rs, int rowNum) throws SQLException {
            Assunto assunto = new Assunto();
            assunto.setId(rs.getLong(1));
            assunto.setNome(rs.getString(2));
            return assunto;
        }
    });
}
```

A função `query` de `JdbcTemplate` recebe três valores: nossa consulta, a lista de parâmetros que passaremos à nossa consulta sob a forma de uma matriz de objetos e uma implementação de `org.springframework.jdbc.core.RowMapper`, que é a interface de callback responsável por ler os campos de cada registro obtido por nossa consulta e retornar um objeto baseado nestes valores.

Observe que `RowMapper` usa *generics*. Com isto conseguimos escrever código bem limpo, como o exposto. Precisamos implementar um único método, `mapRow`, que possui por objetivo mapear os campos presentes no registro corrente de nossa consulta para o objeto de nossa escolha. Este método recebe dois parâmetros: um objeto do tipo `java.sql.ResultSet` e outro do tipo inteiro, usado para representar a posição do registro dentro do resultado obtido.

Importante: quem itera sobre os registros é a classe `JdbcTemplate`, e não nossa implementação de `RowMapper`, sendo assim, em hipótese alguma o desenvolvedor deve invocar o método `next()` do objeto `ResultSet` dentro desta classe. Este é um erro comum que pode ocasionar consequências imprevisíveis de acordo com o driver do banco de dados usado em sua aplicação.

Atualizando a base de dados

As mesmas facilidades que vimos para a obtenção de dados, também se aplica à sua atualização. Se para a obtenção de registros o `JdbcTemplate` nos oferece a

função `query`, a função básica que usaremos agora é `update`. Seu funcionamento é praticamente idêntico ao que já vimos anteriormente, o que pode ser confirmado no código a seguir que atualiza os dados da entidade `Assunto` passada como parâmetro:

```
public void update(Assunto assunto) {
    getJdbcTemplate().update("update assunto set nome = ? where id = ?",
        assunto.getNome(), assunto.getId());
}
```

Vemos exatamente a mesma assinatura do método `query`: uma string representando o comando a ser executado contra o banco de dados e os parâmetros passados em ordem usando a sintaxe *varargs* do Java 5. Na versão anterior do `JdbcTemplate` em que o suporte à nova sintaxe da linguagem Java não estava presente, seria passada uma matriz de objetos tal como no exemplo a seguir:

```
public void update(Assunto assunto) {
    Object[] parametros = {assunto.getNome(), assunto.getId()};
    getJdbcTemplate()
        .update("update assunto set nome = ? where id = ?", parametros);
}
```

Executar comandos DDL (*Data Definition Language*) contra a base de dados segue a mesma linha. A diferença é que usamos a função `execute`, que vemos em prática no exemplo a seguir:

```
String ddl = "create table assunto(" +
    "id long auto_increment not null, " +
    "nome varchar(255) not null, " +
    "primary key(id))";
getJdbcTemplate().execute(ddl);
```

O interessante desse método é que ele pode ser usado para executar qualquer tipo de instrução SQL, por exemplo, para executar registros de atualização de banco de dados em seus projetos, tal como no exemplo a seguir:

```
public void atualizarBD(File arquivo) {
    /*
        Obtém um java.io.BufferedReader a partir do arquivo
        Código omitido para facilitar a leitura do exemplo
    */
    BufferedReader reader = obterBufferedReader(arquivo);
```

```
String comando = null;
while ((comando = reader.readLine()) != null) {
    getJdbcTemplate().execute(comando);
}
// Restante omitido
}
```

Inserção facilitada: obtenha os valores gerados de forma automática

Na versão 2.5 do Spring uma nova classe foi adicionada ao arsenal JDBC do framework. Apesar de pouco divulgada e usada, é uma ferramenta muito interessante quando enfrentamos problemas ao lidar com a inserção de registros. Estou falando de `SimpleJdbcInsert`, uma classe que basicamente é um invólucro para o `JdbcTemplate`.

Apesar de parecer simples a inserção de registros é uma tarefa que carrega alguns desafios. Tome por exemplo a obtenção do valor de um campo gerado automaticamente como no caso de chaves primárias, que será o que veremos nesta seção. Assim como `JdbcTemplate`, `SimpleJdbcInsert` também é thread safe uma vez que tenha sido configurado, porém sua construção é mais interessante, pois baseia-se no padrão *builder*. No código a seguir podemos ver um exemplo de seu uso.

```
public void inserir(Assunto assunto) {
    HashMap<String, Object> parametros = new HashMap<String, Object>();
    parametros.put("nome", assunto.getNome());
    long id = new SimpleJdbcInsert(getDataSource())
        .withTableName("assunto")
        .usingGeneratedKeyColumns("id")
        .executeAndReturnKey(parametros).longValue();
    assunto.setId(id);
}
```

O exemplo insere um novo registro na tabela `assunto` e, logo em seguida, retorna o valor da chave primária da tabela, no caso, um campo auto incremental. A primeira parte do código cria um mapa aonde se encontrarão presentes os valores de todos os campos que pretendemos inserir em nossa tabela, levando como chave o nome do campo a ser preenchido.

A novidade está no modo como instanciamos nosso `SimpleJdbcInsert`. Seu construtor pode receber como valor tanto um objeto do tipo `JdbcTemplate` quanto um `DataSource`. Caso seja o segundo caso, será criado internamente um

novo objeto `JdbcTemplate` encapsulado. Em seguida são invocadas as funções `withTableName`, usada para definir o nome da tabela na qual será feita a inclusão e `usingGeneratedKeyColumns`, que irá instruir `SimpleJdbcInsert` a respeito de quais campos auto gerados devem ser monitorados.

Finalmente, a função `executeAndReturnKey`, que recebe como parâmetro o mapa criado anteriormente irá executar a inserção na tabela e retornará como resultado um objeto do tipo `java.lang.Number` representando o valor gerado para o campo `id` definido pelo método `usingGeneratedKeyColumns`.

Processamento em lote

Como mencionado, uma das possíveis razões pelas quais os desenvolvedores optam pelo JDBC é performance. Um dos maiores vilões nesta área é o tráfego na rede. Se formos executar N inserções ou edições em nossa base de dados em série e cada uma requerer uma conexão remota, com certeza pagaremos um alto preço neste requisito. Para minimizar este problema diversos fornecedores de bancos de dados incluem seus drivers JDBC a possibilidade de processamento em lote.

Por processamento em lote entenda a possibilidade de enviar grupos de comandos SQL em uma única transmissão ao banco de dados. Com isto o tráfego pela rede diminuirá significativamente e nossa performance sofrerá melhorias expressivas. Novamente aqui o *Spring* salva nossa pele com a interface de callback `org.springframework.jdbc.core.BatchPreparedStatementSetter`. Para ilustrar seu uso, vamos expor a implementação de uma classe que importa definições de usuários armazenadas em arquivo para o nosso banco de dados conforme veremos a seguir:

```
public void importarUsuarios(File arquivo) {
    // Obtém uma lista de usuários a partir do arquivo de entrada
    final List<Usuario> usuarios = obterUsuarios(arquivo);

    // Nosso BatchPreparedStatementSetter
    BatchPreparedStatementSetter batch =
        new BatchPreparedStatementSetter() {
            public void setValues(PreparedStatement ps, int i) {
                Usuario usr = usuarios.get(i);
                ps.setString(1, usr.getNome());
                ps.setString(2, usr.getLogin());
                ps.setString(3, usr.getSenha());
            }
        }
```

```
        public int getBatchSize() {
            return usuarios.size();
        }
    }

    // Executa o processamento em lote
    jdbcTemplate.
        batchUpdate(
            "insert into usuario (nome, login, senha) values (?, ?, ?)",
            batch);
}
```

A função `getBatchSize` retorna um valor inteiro representando o tamanho do lote. Em nosso caso, esta retorna como valor o número de instâncias da classe `Usuario` que a função `obterUsuarios` recebeu. Atenção especial deve ser dada ao método `setValues` que recebe dois parâmetros de entrada: um objeto do tipo `java.sql.PreparedStatement` e um inteiro indicando a posição do comando dentro do lote.

O programador deverá apenas definir os valores dos parâmetros dentro da consulta e mais nada. Lembre-se que o responsável por executar o lote de operações não é o `BatchPreparedStatementSetter` - repare no nome da classe - mas sim o nosso `JdbcTemplate`.

9.6 O TRABALHO COM O HIBERNATE

Como JDBC é a base de toda a infraestrutura, nada mais natural que seja também a opção mais primitiva, o que justifica o fato da maior parte deste capítulo ter sido voltada a esta tecnologia. Quando lidamos com um ORM, estamos encarando uma criatura bem mais evoluída: no caso do Hibernate, por exemplo, praticamente todas as facilidades referentes a manipulação e pesquisa de dados se encontram presentes na interface `org.hibernate.Session`.

Caso o leitor já tenha trabalhado com o Hibernate, já está familiarizado com o `SessionFactory`, que é o objeto responsável por criar, abrir e fechar sessões. E com as *sessões contextuais*, esta entidade consegue liberar o desenvolvedor da necessidade de implementar código responsável pela obtenção e fechamento de sessões.

Neste ponto é importante lembrarmos das três principais responsabilidades dos templates de acesso a dados do Spring:

- Facilitar a implementação de consultas e atualizações da base de dados;
- Gerenciar recursos de forma transparente ao programador. Por recurso, entenda a unidade básica responsável pela persistência. No caso do JDBC, estamos falando de conexões, para o Hibernate, sessões;
- Transformar exceções específicas de uma tecnologia para a hierarquia do Spring.

Levando em consideração que das três responsabilidades de um template de acesso a dados do Spring, as duas primeiras já foram resolvidas pelo próprio Hibernate em sua evolução, a existência de um template específico para esta tecnologia só serviria para atender a terceira necessidade.

Coloque-se no lugar de um dos responsáveis pela evolução do Spring: há um ganho real em manter um template específico para o Hibernate só para lidar com exceções específicas de plataforma? Não. Será que não estaríamos poluindo o código dos desenvolvedores que usam nosso framework só para suprir esta necessidade? Sim. Valeria à pena? Não. Há alguma solução alternativa para o problema das exceções? Sim!

EXISTE UM HIBERNATE TEMPLATE!

Nem sempre o Hibernate ofereceu o recurso de sessões contextuais. Nesta época a forma mais popular de lidar com o Spring era através da classe `HibernateTemplate`, disponível no pacote `org.springframework.orm.hibernate3`, que funcionava exatamente como `JdbcTemplate` que vimos na primeira parte deste capítulo.

Já no pacote `org.springframework.orm.hibernate4` esta classe não existe, e tentar usar `HibernateTemplate` com a versão 4 do Hibernate é altamente não recomendado, pois esta simplesmente não funcionará como esperado. Sendo assim, o melhor caminho a se seguir é simplesmente ignorar esta classe, que ainda é distribuída junto com o Spring apenas para fins de retro compatibilidade.

Preparando o ambiente

Para ativar o suporte ao Hibernate precisamos adicionar uma nova dependência ao nosso projeto. Trata-se do pacote `spring-orm`, que nos fornecerá suporte aos principais ORMs do mercado suportados pelo Spring: Hibernate (3, e 4), iBatis, JDO e JPA. Abaixo segue como esta dependência deve ser incluída no arquivo `pom.xml`.

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-orm</artifactId>
  <version>3.1.2.RELEASE</version>
</dependency>
```

Como lidar com exceções

Como mencionado, não precisamos de templates para lidarmos com as transformações de exceções específicas de plataforma para as da hierarquia uniforme do Spring. A solução é bastante simples: basta anotarmos nosso DAO com `@Repository` ao invés de `@Component` como fizemos até então. Esta é uma solução interessante que pode ser aplicada a qualquer DAO, inclusive JDBC. Esta anotação é um estereótipo que permite ao container do Spring aplicar funcionalidades mais interessantes a DAOs, pois os diferencia dos demais beans definidos em seu contexto.

A seguir podemos ver um exemplo de aplicação desta anotação, no caso, na versão *Hibernate* do nosso `DAOAssunto`.

```
import org.springframework.stereotype.Repository;

@Repository
public class HibernateAssunto {
    // conteúdo omitido
}
```

Apenas a presença da anotação não é suficiente: faz-se necessário também incluir a definição de um novo bean do tipo `PersistenceExceptionTranslationPostProcessor`. O que este bean faz é criar um aspecto do tipo `after throw` para todos os beans anotados com `@Repository`. A exceção será interceptada e automaticamente convertida para outra presente na hierarquia de exceções uniformizada do Spring. Abaixo podemos ver como a declaração deste bean - que sequer precisa de um nome - é trivial em XML:

```
<bean class="org.springframework.dao.annotation.  
    PersistenceExceptionTranslationPostProcessor"/>
```

E com isto os templates se tornam completamente desnecessários não só para o Hibernate mas também para qualquer outro ORM que seja nativamente suportado pelo Spring.

O trabalho com a SessionFactory

Assim como o primeiro passo ao lidarmos com JDBC é a configuração da fonte de dados, nosso segundo passo ao configurarmos o Hibernate é a definição da nossa implementação da SessionFactory. Há dois pacotes que devem ser levados em consideração no suporte ao Hibernate: `org.springframework.orm.hibernate3` e `org.springframework.orm.hibernate4`, que contém respectivamente as classes de suporte às versões 3 e 4 do ORM.

Qual implementação de SessionFactory escolher depende de duas condições: a versão do Hibernate usada em seu projeto e qual a opção de mapeamento adotada: XML ou anotações.

No caso de estarmos lidando com Hibernate 3, temos duas implementações a escolher: `LocalSessionFactoryBean`, caso tenhamos optado por configurações no formato XML ou `AnnotationSessionFactoryBean`, implementada no pacote `org.springframework.orm.hibernate3.annotation` para anotações.

Começaremos com `LocalSessionFactoryBean`, pois sua configuração básica será muito similar à de `AnnotationSessionFactoryBean`. Podemos ver abaixo um exemplo de sua declaração:

```
<bean  
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">  
<property name="dataSource" ref="dataSource"/>  
<property name="hibernateProperties">  
    <props>  
        <prop key="hibernate.dialect">  
            org.hibernate.dialect.MySQL5InnoDBDialect  
        </prop>  
        <prop key="hibernate.hbm2ddl.auto">update</prop>  
    </props>  
</property>  
</bean>
```

O trecho exposto é comum a todas as implementações de `SessionFactory`. Duas são as propriedades que sempre devem ser preenchidas: `dataSource`, apontando para a fonte de dados a ser usada e `hibernateProperties`, contendo todas as chaves de configuração que normalmente incluiríamos no arquivo `hibernate.cfg`.

O restante da configuração dirá respeito ao modo como mapeamos nossas classes. Caso nossa opção tenha sido por arquivos XML, podemos incluir os arquivos a serem levados em consideração um a um como no código a seguir:

```
<bean
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
<!-- restante oculto para facilitar leitura -->
<property name="mappingResources">
    <list>
        <value>springforum.entidades.Assunto.hbm.xml</value>
        <value>springforum.entidades.Topico.hbm.xml</value>
        <!-- os demais arquivos seguiriam a mesma regra -->
    </list>
</property>
</bean>
```

Mas o desenvolvedor também pode se preferir apontar quais os caminhos dentro do classpath ou sistema de arquivos aonde devem ser buscados os arquivos de configuração a serem lidos como no exemplo a seguir:

```
<bean class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
<!-- restante oculto para facilitar leitura -->
<property name="mappingLocations">
    <list>
        <value>classpath:br.com.itexto.springforum.entidades</value>
        <value>file:///home/kicolobo/mapeamentos</value>
    </list>
</property>
</bean>
```

Caso nosso objetivo seja lidar com anotações com Hibernate 3, conforme mencionado, `LocalSessionFactoryBean` não mais se encaixa. A partir deste momento passaremos a usar `AnnotationSessionFactoryBean`. A diferença entre esta implementação e a que vimos anteriormente é que não temos mais as propriedades `mappingResources` e `mappingLocations`, mas sim `annotatedClasses` aonde definimos individualmente cada classe anotada com a qual trabalharemos

e `annotatedPackages`, usada para definir quais os pacotes em nosso classpath que contém as entidades anotadas em nosso sistema.

A seguir podemos ver dois exemplos de definição deste bean:

```
<bean class="org.springframework.orm.hibernate3.annotation
    .AnnotationSessionFactoryBean">
    <!-- Definindo classes anotadas individualmente -->
    <property name="annotatedClasses">
        <list>
            <value>br.com.itexto.springforum.entidades.Assunto</value>
            <value>br.com.itexto.springforum.entidades.Topico</value>
        </list>
    </property>
    <!-- Ou definindo apenas os pacotes aonde se encontram nossas
        classes anotadas -->
    <property name="annotatedPackages">
        <list>
            <value>br.com.itexto.springforum</value>
        </list>
    </property>
</bean>
```

E sim, é possível misturar os dois tipos de caminho na mesma definição para obter máxima flexibilidade.

Quando formos trabalhar com Hibernate 4 nossa vida se torna mais simples, pois não temos mais duas implementações distintas de `SessionFactory`, mas apenas uma: `org.springframework.orm.hibernate4.LocalSessionFactoryBean`, que pode lidar tanto com configurações no formato XML quanto anotações. E ainda mais interessante, ainda nos possibilita misturar os dois formatos caso assim quiséssemos. Abaixo podemos ver um exemplo de definição deste bean:

```
<bean class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
    <!-- DataSource -->
    <property name="dataSource" ref="dataSource"/>
    <!-- As configurações específicas do Hibernate -->
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">
                org.hibernate.dialect.MySQL5InnoDBDialect
            </prop>
            <prop key="hibernate.hbm2ddl.auto">update</prop>
        </props>
    </property>
</bean>
```

```

        </props>
    </property>
    <!--
        As opções de mapeamento que são exatamente as
        mesmas que vimos para o Hibernate 3
    -->
    <property name="annotatedClasses">
        <list>
            <value>br.com.itexto.springforum.entidades.Assunto</value>
            <value>br.com.itexto.springforum.entidades.Topico</value>
        </list>
    </property>
    <property name="annotatedPackages">
        <list>
            <value>br.com.itexto.springforum</value>
        </list>
    </property>
    <property name="mappingResources">
        <list>
            <value>springforum.entidades.Assunto.hbm.xml</value>
            <value>springforum.entidades.Topico.hbm.xml</value>

        </list>
    </property>
</bean>

```

DAOs com Hibernate

Ao adotarmos uma ferramenta ORM como Hibernate a escrita da camada de persistência se torna muito mais simples porque é ordens de magnitude mais fácil escrever DAOs genéricos, ou seja, ao invés de escrevermos uma implementação do DAO para cada entidade, é possível escrever uma única que seja capaz de lidar com o maior número de situações possível e, para lidar com os casos em que situações peculiares ocorram, bastaria escrever uma subclasse desta.

Com essa classe genérica, poderíamos criar um método que encapsule a obtenção da *Session*, dessa forma, todos os outros DAOs que a herdarem terão essa funcionalidade.

```

public class GenericDAO<T> {
    @Autowired

```

```
private SessionFactory sessionFactory;

protected Session getSession() {
    return sessionFactory.getCurrentSession();
}
// restante oculto para facilitar a leitura
}
```

Por via de regra, sempre obtenha uma sessão de `SessionFactory` a partir da função `getCurrentSession`, pois assim temos a garantia de que estaremos sempre lidando dentro do contexto transacional correto - para tal, você deve ativar o suporte a transações do Spring como veremos no próximo capítulo, caso contrário uma exceção será disparada dizendo que nenhuma transação está ativa. Porém, uma vez configurado o desenvolvedor não precisa mais se preocupar com isto, tal como é exposto no exemplo a seguir:

```
public class GenericDAO<T> {
    public void persistir(T objeto) {
        getSession().saveOrUpdate(objeto);
    }
}
```

9.7 INTEGRE COM A JPA

Finalmente, temos o JPA. O modo como Spring lida com esta tecnologia é exatamente o mesmo que vimos com Hibernate. Enquanto na versão 3.0 do framework tínhamos a classe `JpaTemplate`, este perde sua função na versão 3.1 pelas mesmas razões que levaram à obsolescência de `HibernateTemplate`. Sendo assim, as mesmas práticas se aplicam:

- Anote seus DAOs com `@Repository` para que o Spring possa fazer a transição das exceções específicas do JPA para as padrão do Spring;
- Templates não são mais necessários porque `EntityManagerFactory` já cuida da criação e destruição das instâncias de `EntityManager`;
- As facilidades para pesquisa e persistência esperadas em um template já estão presentes em `EntityManager`.

Sendo assim, tudo o que precisamos saber para trabalhar com JPA é como instanciar o `EntityManagerFactory` correto. A mesma configuração de ambiente aplicada

no caso do Hibernate, a inclusão da dependência `spring-orm` aplica-se no caso do JPA.

EXISTE UM JpaTemplate!

Na realidade existe um `JpaTemplate`, porém desde a versão 3.1 do Spring seu uso é não recomendado e aplica-se apenas à primeira versão do JPA, sem planos de ser desenvolvido no futuro. As razões pelas quais esta classe foi marcada como depreciada pela equipe de desenvolvimento do framework são exatamente as mesmas que tornaram obsoleto o `HibernateTemplate`.

Obtendo o EntityManagerFactory

Sempre que iniciamos um projeto baseado em JPA a primeira escolha que precisamos fazer diz respeito ao tipo de `EntityManagerFactory` com o qual trabalharemos. Há dois tipos de acordo com quem o gerencia: a aplicação ou o servidor de aplicações. Quando gerenciado pela aplicação, esta é a responsável por controlar o contexto transacional e como serão criadas novas instâncias do `EntityManager`, já no caso em que o container o gerencia, a aplicação de forma alguma deveria interagir diretamente com esta entidade.

Para você que trabalha com Spring, não importará qual `EntityManagerFactory` será usado, pois o Spring será sempre o responsável por gerenciar o contexto transacional e o modo como novas instâncias do `EntityManager` serão instanciados. O único momento em que esta questão se apresentará será quando definirmos qual bean declararemos como nossa implementação de `EntityManagerFactory`.

Assim como no caso do suporte ao Hibernate 3, o Spring nos oferece duas implementações a escolha do desenvolvedor:

- `LocalEntityManagerFactoryBean`: para os casos em que a própria aplicação gerencia o `EntityManagerFactory`;
- `LocalContainerEntityManagerFactoryBean`: no caso de estarmos lidando com um servidor de aplicações.

Para começar, veremos como configurar o JPA quando este é gerenciado pela aplicação. Ao contrário do que ocorre no caso do Hibernate em que toda a configuração é movida para o container do Spring, quando lidamos com JPA é necessário

que as convenções deste mecanismo de persistência sejam respeitadas. Isto quer dizer que ainda se faz necessária a presença do arquivo `persistence.xml` dentro do diretório META-INF do seu projeto, o que nos traz como vantagem o fato de não precisarmos digitar estas configurações dentro do container do Spring. A seguir podemos ver um exemplo de como definir o bean `LocalEntityManagerFactoryBean`.

```
<bean class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="springForumPU" />
</bean>
```

Repare que só precisamos definir uma propriedade neste caso, que é `persistenceUnitName`. Seu valor deve corresponder à mesma unidade de persistência que definimos no arquivo `persistence.xml`. Esta é também a versão mais primitiva oferecida pelo Spring, pois não nos permite definir uma fonte de dados customizada como fizemos com Hibernate e JDBC, além de também não possuir suporte a transações globais. Deve ser usado apenas em ambientes de teste ou aplicações que sejam executadas stand alone, ou seja, fora de um servidor de aplicações.

Caso estejamos trabalhando com um servidor de aplicações, também podemos obter um `EntityManagerFactoryBean` via JNDI. Para tal, basta procedermos exatamente como fizemos para obter uma fonte de dados a partir desta API usando o namespace `jee` que vimos no início deste capítulo.

```
<jee:jndi-lookup id="emf"
    jndi-name="persistence/unidadePersistenciaSpringForum"/>
```

Neste caso o pré-requisito é de que toda a configuração JPA seja gerenciada unicamente pelo servidor de aplicações responsável pela detecção das unidades de persistência. No caso é importante lembrar que o nome JNDI deve corresponder ao nome da unidade de persistência específico de sua aplicação.

E finalmente temos o `LocalContainerEntityManagerFactoryBean`: como opção, que é a mais poderosa oferecida pelo Spring e nos permite configurar com um nível de granularidade bem maior o modo como nosso `EntityManagerFactory` deve se comportar. Este possui como principais vantagens a possibilidade de trabalhar com fontes de dados customizadas pelo Spring e o fato de estar habilitado para lidar com transações locais e globais, ao contrário de `LocalEntityManagerFactoryBean`.

Uma adição bastante útil é a presença da propriedade `packagesToScan`, que é muito útil quando encontramos mais de um arquivo `persistence.xml` no classpath de nossa aplicação.

```
<bean id="emf"
      class="org.springframework.orm.jpa.
          LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="dsSpringForum"/>
  <property name="packagesToScan">
    <list>
      <value>br.com.itexto.springforum.dao.jpa</value>
    </list>
  </property>
</bean>
```

Escrevendo DAOs

Basicamente a mesma estratégia que aplicamos ao Hibernate é repetida para o JPA. Basta que implementemos nossos DAOs de tal forma que nestes possa ser injetado nossa instância de *EntityManagerFactory*. Neste caso, podemos inclusive tirar proveito da anotação `@PersistenceUnit` do JPA, tal como pode ser visto na listagem a seguir:

```
@Repository
public class GenericDAO<T> {
    private EntityManagerFactory emf;
    @PersistenceUnit
    public void setEntityManagerFactory(EntityManagerFactory emf) {
        this.emf = emf;
    }

    protected EntityManager getEntityManager() {
        return emf.createEntityManager();
    }
    // restante do código omitido
}
```

E assim como fizemos no caso do Hibernate, o mesmo se aplica ao JPA. Trabalhamos com este motor de persistência exatamente como faríamos normalmente, tal como ilustrado no código a seguir:

```
public class GenericDAO<T> {  
    public void persistir(T objeto) {  
        getEntityManager().persist(objeto);  
    }  
}
```

9.8 CONCLUINDO

Neste capítulo vimos o suporte que o Spring oferece a três das mais populares opções de persistência adotadas por desenvolvedores Java: JDBC, Hibernate e JPA. Na primeira parte deste capítulo vimos como a API JDBC, apesar de tão sólida é limitada, fornecendo assim possibilidades de melhorias que são muito bem implementadas a partir dos templates do Spring.

É importante observar as mudanças que o framework sofreu do seu release anterior (3.0) para o atual (3.1) no que diz respeito ao seu suporte a ORMs, deixando de lado os templates e adotando como solução uma menor dependência entre nossas classes e as do framework.

Fica clara a tendência de redução do escopo do Spring nesta área conforme estas ferramentas evoluem. Esta tendência exibe uma característica muito positiva da equipe responsável pelo desenvolvimento do Spring, que é o seu pragmatismo. Não há razões para reinventar a roda quando a tecnologia subjacente já possui boas alternativas para os problemas apresentados.

CAPÍTULO 10

Gerenciando transações

Um dos requisitos fundamentais por trás de toda aplicação é garantir a consistência das nossas bases de dados. No capítulo anterior aprendemos como o Spring facilita a escrita da camada de persistência. Com isto resolvemos boa parte dos nossos problemas, porém ainda não garantimos a integridade dos nossos dados. Muitos sistemas atuais possuem um alto nível de concorrência e processamento paralelo, e não há como garantir esta necessidade sem o auxílio do contexto transacional que o Spring praticamente trivializa.

10.1 ENTENDENDO AS TRANSAÇÕES

A grosso modo, uma transação representa um conjunto de operações no qual ou todas são executadas com sucesso ou nenhuma é. Como exemplo vamos imaginar que no projeto *Spring Fórum* seja implementada uma loja. Sempre que um cliente comprar um produto, o seguinte conjunto de operações é executado:

- 1) O estoque é decrementado;

- 2) É feito o débito no cartão do cliente;
- 3) Um e-mail de confirmação de compra é enviado;
- 4) É enviada uma solicitação de envio para a transportadora.

Agora imagine que o segundo passo apresente algum problema e não seja feito o débito no cartão do cliente: teremos decrementado nosso estoque, enviado um e-mail de confirmação de compra e o produto talvez já esteja inclusive a caminho. Este conjunto de atividades possui um nome dentro do jargão transacional: *unidade de trabalho (unit of work)*.

A grosso modo, o que uma transação realmente faz é transformar um conjunto de tarefas em uma unidade de trabalho, unificando-as e assim garantindo que ou todas dão certo ou que, em caso de erro nossa base de dados volte a um estado consistente. Voltando ao exemplo anterior, caso um erro ocorresse no momento em que o débito no cartão do cliente fosse acontecer, o estoque voltaria para o seu estado anterior e os passos seguintes simplesmente não seriam executados, garantindo assim a integridade do nosso sistema.

Transações são ACIDAs

Toda transação deve satisfazer quatro condições que formam o acrônimo ACID:

- **Atomicidade:** o conjunto de operações que formam a unidade de trabalho devem ser vistas como uma única operação artificialmente indivisível. Ou todos os passos são executados com sucesso ou nada é feito;
- **Consistência:** se a transação for executada com êxito, o estado final da nossa base de dados deve sempre estar em um estado consistente. Aliás, esta é a razão pela qual transações existem, certo?
- **Isolamento:** dificilmente encontramos sistemas mono usuário atualmente. É incrementalmente comum - na realidade, a norma - a ocorrência de mais de uma transação simultânea sendo executada. É necessário que a execução de uma transação seja independente das demais, garantindo assim a execução de código concorrente e a integridade dos dados ao final do seu processamento.
- **Durabilidade:** finalizada a transação, os dados devem estar permanentemente persistidos na base de dados. Para melhor entender este conceito, é interessante observar que diversos mecanismos transacionais mantêm o estado

da base de dados durante a execução de uma transação em memória. Após a sua execução, estes obrigatoriamente devem estar salvos em um mecanismo de armazenagem durável, como por exemplo o disco rígido, fita etc.

É importante observar que quando falamos destes quatro atributos estamos nos referindo na maior parte das vezes a bases de dados relacionais. Com o surgimento de bancos de dados NoSQL, a situação tem se mostrado um pouco diferente, pois estes costumam oferecer suporte incompleto ao ACID. É fundamental que o leitor leve este aspecto em consideração caso esteja a trabalhar com este tipo de tecnologia.

10.2 AS POLÍTICAS TRANSACIONAIS

O Spring oferece suporte a dois tipos de transações: programáticas ou declarativas. No caso das transações programáticas, está nas mãos do desenvolvedor a responsabilidade de definir aonde começa e termina uma unidade de trabalho. Já no caso das transações declarativas, tenha em mente que estamos na realidade definindo regras que ditarão ao container quais os limites que devem ser aplicados a cada unidade de trabalho.

O suporte a transações declarativas do Spring tem como inspiração o *CMT* (Container Management Transactions) da plataforma Java EE. É interessante observar que até o aparecimento do Spring não havia como tirar proveito deste recurso sem a presença de um servidor pesado de aplicações. Como veremos, o Spring supera o CMT ao permitir uma personalização de maior granularidade através da possibilidade de podermos definir através das políticas transacionais qual o comportamento das unidades de trabalho.

Na implementação do Spring das transações declarativas há cinco políticas que podem ser manipuladas pelo programador.

Propagação

A política de propagação define os limites de uma transação. Pense como a resposta à seguinte pergunta: “dado que um método inicia uma transação e este evoca ou é chamado por outros, qual a propagação da transação por ele criada?”. O Spring nos oferece algumas respostas a esta pergunta sob a forma de constantes definidas na interface `org.springframework.transaction.TransactionDefinition` que podemos ver listadas a seguir:

- `PROPAGATION_MANDATORY`: o método só pode ser executado dentro de uma

transação. Se uma transação não estiver em execução, quando o método for invocado uma exceção será disparada;

- **PROPAGATION_NESTED**: caso o método seja executado a partir de uma transação, uma nova será criada para este método e seu commit ou rollback não afetará a que o encapsulou. Caso seja invocado sem que uma transação exista, uma nova será criada somente para si;
- **PROPAGATION_NEVER**: o método jamais deve ser executado dentro de uma transação. Caso isto ocorra, uma exceção será disparada;
- **PROPAGATION_NOT_SUPPORTED**: o método não deve ser executado dentro de uma transação. Caso seja chamado a partir de uma transação, esta será paralisada até que sua execução termine para em seguida voltar à sua execução normal;
- **PROPAGATION_REQUIRED**: o método só pode ser executado em uma transação. Caso seja invocado a partir de uma, então fará parte desta, ou seja, será parte da sua unidade de trabalho. Se não existir uma transação em execução, uma nova será criada;
- **PROPAGATION_REQUIRES_NEW**: o método só pode ser executado dentro da sua própria transação. Se for invocado a partir de uma transação, então uma nova será criada exclusivamente para seu processamento e a que o encapsulou será paralisada até que este finalize;
- **PROPAGATION_SUPPORTS**: tanto faz para o método se é ou não executado dentro de uma transação. Caso seja executado dentro de uma, fará parte da sua unidade de trabalho como um método qualquer.

Isolamento

Esta política define qual o nível de isolamento de nossa transação declarativa. A pergunta por trás desta política é *como minha transação é influenciada pela existência de outras sendo executadas paralelamente?* Há três problemas clássicos envolvendo esta questão. Caso esteja enfrentando uma situação familiar em seu projeto talvez esta seja a solução para a sua dificuldade.

O primeiro problema é chamado de *leitura suja* (*dirty reads*). Ocorre quando uma transação tem acesso a dados de outra que ainda não tenha finalizado sua exe-

cução. Voltando ao exemplo inicial deste capítulo, uma transação concorrente poderia ler o nosso estoque com a quantidade de um item igual a zero porque a transação responsável pela atualização daquele registro ainda não terminou sua execução.

Outro problema são as *leituras diferentes* (*non repeatable reads*), que é quando uma mesma consulta, executada com os mesmos parâmetros sempre retorna resultados diferentes quando isto não deveria ocorrer. Normalmente a causa do problema é alguma outra transação que naquele mesmo instante esteja alterando os mesmos registros da base de dados.

Similar ao problema das *leituras diferentes* é o das *leituras fantasmas* (*phantom reads*), em que nossa transação espera ler um número de registros e acaba lendo um número maior porque outra transação se encontra incluindo novos registros na base de dados.

Os problemas decorrentes de uma má política de isolamento são difíceis de serem resolvidos, mas facilmente evitáveis quando definimos o comportamento adequado. No caso do Spring, há quatro comportamentos distintos à disposição do programador que também se encontram representados como constantes na interface `org.springframework.transaction.TransactionDefinition`:

- `ISOLATION_DEFAULT`: a transação adotará o comportamento padrão da base de dados. É o que fornece maior portabilidade de código, porém requer que a equipe conheça bem o banco de dados com o qual está trabalhando, visto que o comportamento padrão varia bastante de fornecedor para fornecedor;
- `ISOLATION_READ_UNCOMMITTED`: trata-se da opção mais perigosa, pois permite à transação ler registros que ainda não tenham sido comitados por outras transações. Facilita bastante a ocorrência de erros do tipo *leitura suja*;
- `ISOLATION_READ_COMMITTED`: define que a transação só terá acesso a registros que já tenham sido comitados por outras transações. Evita a ocorrência de erros do tipo *leitura suja*, mas também acarretam um maior peso de performance, visto que podem gerar locks nas tabelas ou registros do sistema durante a sua execução;
- `ISOLATION_REPEATABLE_READ`: usado para evitar erros do tipo *leituras diferentes* e *fantasmas*. No caso, a transação só obterá resultados diferentes caso sejam registros alterados durante a sua execução;
- `ISOLATION_SERIALIZABLE`: é o nível de isolamento mais caro do ponto de vista

computacional, visto que garante a não ocorrência de todos os erros acima mencionados.

Muita atenção deve ser dada a qual política ser adotada, pois quanto mais branda, maior a possibilidade de erros, e quanto mais rígida maior será a penalidade sobre a performance do sistema como um todo devido à criação de novos locks sobre registros ou tabelas no banco de dados.

Somente leitura

Esta política é usada para definir se a transação não efetuará nenhuma escrita na base de dados subjacente, permitindo que esta execute algumas otimizações para a sua execução. O principal ganho desta política é o ganho de performance, pois se nenhum dado será alterado durante a execução de uma transação que implemente esta política, então o acesso aos dados é otimizado.

É importante mencionar que caso ocorra alguma tentativa de alteração de dados durante sua execução uma exceção será disparada pelo sistema.

Tempo de espera (timeout)

Muitas vezes, devido ao estado de um sistema, uma transação pode demorar muito para que seja executada ou volte do estado de paralisação, como por exemplo nos casos em que inicia acidentalmente uma nova transação ou está aguardando a finalização de outra. Nestes casos, para se evitar uma paralisação do sistema é interessante definirmos uma política de tempo de espera. Caso o tempo exceda o que foi definido, a instrução de rollback é executada, mantendo a base de dados em seu estado consistente.

Política de rollback

Por padrão uma transação declarativa executa a operação de rollback apenas se uma exceção de tempo de execução for disparada. A razão por trás desta convenção é o fato de que este tipo de exceção ao ser disparada não possibilita a implementação de estratégias de resgate. O Spring nos permite definir quais os tipos de erro que deverão ser usados para disparar o procedimento de rollback.

10.3 PREPARANDO O AMBIENTE DE DESENVOLVIMENTO

Para habilitar o suporte a transações do Spring é necessária a inclusão de uma nova dependência em nosso projeto que é o módulo `spring-tx`. Basta adicionar a dependência no arquivo `pom.xml` do seu projeto:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-tx</artifactId>
  <version>3.1.2.RELEASE</version>
</dependency>
```

Caso seu projeto já tenha incluído o módulo `spring-orm` ou `spring-jdbc`, esta nova dependência não precisa ser incluída, pois ambos já a tem como dependência transitiva.

10.4 COMO O SPRING IMPLEMENTA O SUPORTE A TRANSAÇÕES

Agora que toda a teoria foi exposta, podemos começar a colocar este conhecimento em prática, entendendo como o Spring oferece suporte ao gerenciamento transacional. No centro deste suporte temos o *Gerenciador de Transações* (*Transaction Manager*). Como o próprio nome já diz, este é o componente responsável por gerenciar as transações e não é necessariamente baseado na API padrão do Java para tal, que é a JTA (*Java Transaction API*), mas sim a tecnologia por trás do armazenamento, como por exemplo JDBC, Hibernate, JPA, JDO e outras.

Por padrão o Spring vêm com uma lista bem grande de gerenciadores de transação, e nós focaremos nos que abordamos no capítulo 9, ou seja, JDBC, Hibernate e JPA. A definição de qual implementação usar é fundamental, pois trata-se do primeiro passo a ser tomado pelo programador na configuração do suporte transacional do framework.

Transações com JDBC

Caso seu projeto use JDBC como mecanismo de persistência, o Spring oferece o `DataSourceTransactionManager`, que atua diretamente sobre a fonte de dados do seu sistema. Por trás dos panos, este é o bean responsável por chamar os métodos `setAutoCommit`, `commit` e `rollback` das conexões obtidas através da fonte de dados em caso de sucesso ou erro de suas transações.

A seguir podemos ver um exemplo de declaração deste bean. Para que possa atuar, a única propriedade a ser definida é `dataSource`, que deve apontar para o bean que representa a fonte de dados do seu sistema.

```
<bean id="transactionManager" class=
    "org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

Transações com Hibernate

Quando lidamos com Hibernate, a escolha por trás de qual gerenciador de transações varia de acordo com a versão do ORM, ou seja, caso esteja lidando com a versão 3, a classe do bean a ser declarado será `org.springframework.orm.hibernate3.HibernateTransactionManager`, e se for 4, `org.springframework.orm.hibernate4.HibernateTransactionManager`. A configuração dos dois beans é idêntica: ambos devem definir a dependência `sessionFactory` apontando para o `SessionFactory` do seu projeto, conforme pode ser visto no exemplo a seguir que expõe a configuração do gerenciador de transações para o Hibernate 4:

```
<bean id="transactionManager" class=
    "org.springframework.orm.hibernate4.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

JPA

O mesmo comportamento adotado nos dois gerenciadores de transação se aplica também ao caso do JPA. No caso, usamos a classe `org.springframework.orm.jpa.JpaTransactionManager`, a qual devemos injetar a propriedade `entityManagerFactory` apontando para o `Entity Manager Factory` default do nosso projeto, conforme pode ser visto no exemplo a seguir:

```
<bean id="transactionManager"
    class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="emf"/>
</bean>
```

10.5 TRANSAÇÕES PROGRAMÁTICAS

O controle transacional de maior granularidade é o programático, no qual o próprio desenvolvedor define manualmente a sua unidade de trabalho. Para que isto ocorra entra em cena mais um template: *org.springframework.transaction.support.TransactionTemplate*, que deve ser injetado nos beans aonde seja de nosso interesse aplicar a técnica. Lembrando do exemplo inicial deste capítulo, poderíamos injetá-lo dentro do nosso bean *Loja*, tal como exposto na configuração a seguir, que usa como base o gerenciador de transações do Hibernate 4:

```
<bean id="transactionManager" class=
    "org.springframework.orm.hibernate4.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>

<bean id="loja" class="br.com.itexto.springforum.Loja">
    <property name="transactionTemplate">
        <bean class=
            "org.springframework.transaction.support.TransactionTemplate">
            <property name="transactionManager"
                ref="transactionManager"/>
        </bean>
    </property>
</bean>
```

Dado que sempre que lidamos com transações o isolamento é fundamental, não é interessante ou seguro ter uma instância compartilhada por mais de um bean, razão pela qual declaramos nosso template como um bean embutido. Com a injeção pronta, tudo o que precisamos fazer é aumentar nosso template com alguma implementação da interface de callback *org.springframework.transaction.support.TransactionCallback* tal como podemos ver no exemplo a seguir:

```
public class Loja {

    private TransactionTemplate txTemplate;

    public void setTransactionTemplate(TransactionTemplate tx) {
        this.txTemplate = tx;
    }
}
```

```
}

public void executarVenda(Cliente cliente, Produto produto)
                                throws ErroSpringForum {
    txTemplate.execute(new TransactionCallback<Void>(){
        public void doInTransaction(TransactionStatus st) {
            try {
                decrementarEstoque(produto);
                efetuarDebito(cliente, produto.getValor());
                enviarEmailCompra(cliente, produto);
                enviarTransportadora(cliente, produto);
            } catch (ErroSpringForum erro) {
                st.setRollbackOnly();
                throw erro;
            }
            return null;
        }
    });
}

// restante da classe omitido
}
```

O que realmente nos interessa é o método `executarVenda`, aonde definimos nossa unidade de trabalho, composta pela execução em sequência dos métodos `decrementarEstoque`, `efetuarDebito`, `enviarEmailCompra` e `enviarTransportadora`, executados dentro do corpo do método `doInTransaction`, da nossa implementação da interface `TransactionCallback`.

O primeiro passo do gerenciamento programático de transações é a execução do método `execute` de `TransactionTemplate`. Este recebe como parâmetro uma implementação da interface `TransactionCallback`, que obriga o desenvolvedor a implementar um único método: `doInTransaction`, que define como parâmetro de entrada um objeto do tipo `TransactionStatus`.

`TransactionStatus` possui um único método que nos interessa: `setRollbackOnly`. Caso uma exceção do tipo `ErroSpringForum` seja disparada no corpo do método `doInTransaction`, este será invocado, informando à transação corrente que todas as operações anteriormente executadas deverão ser desfeitas.

10.6 DECLARANDO TRANSAÇÕES

O problema das transações programáticas é que se quisermos replicar o mesmo comportamento em mais de um ponto no sistema, o programador precisará suar bastante a camisa. Uma alternativa mais interessante para situações genéricas é a possibilidade oferecida pelo Spring de definirmos declarativamente o controle transacional.

Trata-se de um uso bastante interessante do conceito de AOP, pois o que realmente irá ocorrer por trás dos panos é a aplicação de um *advice* do tipo *around* em todos os beans sobre os quais desejemos adicionar este controle transacional.

Há dois caminhos a serem seguidos na definição declarativa de transações. O primeiro é menos intrusivo e consiste na definição dos pontos transacionais apenas via configuração, enquanto o segundo é a maneira mais produtiva existente atualmente através de anotações.

Declarando transações via configuração XML

Como mencionado, a opção menos intrusiva para adicionarmos o controle transacional declarativo é via configuração XML. Dado que o suporte a transações declarativas do Spring é baseado em AOP, faz-se necessária a inclusão do namespace `aop` em nosso arquivo de configuração em conjunto, com uma novidade que é o namespace `tx`, conforme pode ser visto no exemplo a seguir:

```
<beans default-autowire="byType"
  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:jee="http://www.springframework.org/schema/jee"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="http://www.springframework.org/schema/jee
    http://www.springframework.org/schema/jee/spring-jee-3.1.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-3.1.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.1.xsd">

</beans>
```

Dentro do namespace `tx` há uma tag chamada `<tx:advice>`, usada para definir

quais os métodos em que nosso controle transacional se aplica. Nosso primeiro passo será a sua declaração em nosso arquivo de configuração tal como pode ser visto no trecho a seguir:

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
  <tx:attributes>
    <tx:method name="executar*" propagation="REQUIRED"
              isolation="SERIALIZABLE"/>
    <tx:method name="*" propagation="SUPPORTS"/>
  </tx:attributes>
</tx:advice>
```

A tag `<tx:attributes>` é usada para definir todos os métodos que irão receber o nosso controle transacional. No caso, estamos definindo que todo método que comece com `executar` receberá a política de propagação `REQUIRED` com o mais alto nível de isolamento e todos os demais métodos com a política de propagação `SUPPORT`, ou seja, caso sejam executados dentro de uma transação, farão parte da sua unidade de trabalho.

O atributo `transaction-manager` define qual o gerenciador de transações a ser usado por este advice. Caso omitido, seria usado como padrão o bean cujo identificador fosse igual a `transactionManager`.

O próximo passo agora é definir quais os pointcuts aonde o advice será aplicado. Volta à cena a tag `<aop:config>` que vimos no capítulo 5 que fechará o nosso círculo tal como pode ser visto no trecho a seguir:

```
<aop:config>
<aop:advisor pointcut="execution(* *..Loja.*(..))"
             advice-ref="txAdvice"/>
</aop:config>
```

Com isto o aspecto é aplicado em todos os métodos do bean `loja` e nosso controle transacional está concluído usando XML.

Declarando transações com anotações

O problema com a configuração no formato XML é que esta tende a crescer bastante conforme o projeto evolui. A segunda alternativa que, como mencionei, é também a mais produtiva se dá através de anotações. Para habilitá-la só é necessário adicionar uma linha de configuração tal como no trecho a seguir:

```
<tx:annotation-driven transaction-manager="transactionManager" />
```

A tag `<tx:annotation-driven>` instrui o container a aplicar as regras de transacionalidade a todos os beans que possuam a anotação `@Transactional` inserida na definição da classe (aplicando a regra a todos os métodos da mesma) ou a métodos específicos.

O atributo `transaction-manager`, como no caso da tag `<tx:advice>`, é opcional: caso omitido será usado o primeiro bean encontrado cujo nome seja `transactionManager`.

A seguir podemos ver um exemplo de classe anotada com `@Transactional`:

```
@Component
public class Loja {
    @Transactional(propagation=Propagation.REQUIRES_NEW,
                    isolation=Isolation.SERIALIZABLE,
                    timeout=120)
    public void executarVenda(Produto produto, Cliente cliente) {
        //conteúdo oculto, agora sem o TransactionTemplate
    }
}
```

10.7 CONCLUINDO

Neste capítulo vimos como o Spring implementa seu suporte a transações. Ela é uma solução de altíssima produtividade devido à sua facilidade de uso, o que tornou este suporte uma das principais razões pelas quais desenvolvedores do mundo inteiro começaram a se interessar pelo framework. Justamente, o ponto mais importante é você entender a diferença entre as diferentes estratégias de transação e qual o impacto que elas podem ter em seu projeto.

Caso queira se aprofundar em mecanismos de transação em geral, recomendo a leitura do livro *Java Transaction Design Strategies*, de Mark Richards [13] que pode inclusive ser obtido gratuitamente no site da InfoQ - <http://www.infoq.com>.

CAPÍTULO 11

Protegendo nossa aplicação com Spring Security

O Spring Security é um framework de controle de acesso, que nos permite definir de forma declarativa *quem* acessará *o quê* em nossos sistemas. Muitos o descrevem como um *framework de segurança*, mas esta é uma definição muito imprecisa e que pode levar desenvolvedores à falsa impressão de que com ele conseguiríamos construir projetos invulneráveis a ataques, o que não é verdade.

O controle de acesso é parte fundamental na segurança de qualquer projeto, porém segurança não se limita a “apenas” este *aspecto*. Há outros problemas que também devem ser tratados pelo desenvolvedor que não são resolvidos pelo Spring Security, como por exemplo, ataques de *negação de serviço* (DOS: *Denial of Service*), entradas maliciosas, injeção de código e muitas outras ameaças que assolam sistemas pelo mundo.

O controle de acesso do Spring Security pode ser aplicado em dois níveis: nas requisições que chegam à nossa aplicação, caso estejamos lidando com um projeto web

e na invocação de métodos dos beans gerenciados pelo contexto do Spring. Como veremos, não é acidentalmente que usei a palavra *aspecto* no parágrafo anterior: controlar o acesso é um interesse transversal que é resolvido pelo framework através do suporte a AOP. Mas antes da prática é necessário conhecermos os dois conceitos principais por trás do seu funcionamento.

11.1 ENTENDENDO AUTENTICAÇÃO E AUTORIZAÇÃO

Todo controle de acesso possui dois procedimentos: *autenticação* e *autorização*. O primeiro tem por objetivo garantir que um usuário realmente é quem diz ser, enquanto o segundo verifica se este, uma vez autenticado, possui permissão para executar determinada tarefa no sistema.

Todo processo de autenticação se inicia com o usuário apresentando-se ao sistema, entregando suas *credenciais*, que são seus dados de identificação, como o login e senha ou dados biométricos. O sistema autenticador então gera uma assinatura única com estas credenciais e, com base nesta, verifica a existência ou não do usuário em suas bases de dados.

Existindo o usuário, não raro as credenciais estão associadas a dados cadastrais relativos a este, como por exemplo nome completo, data da última autenticação, endereço etc. A estas informações os frameworks de controle de acesso costumam dar o nome de *principal* (sujeito). Lembre-se: um usuário não necessariamente é um ser humano, pode ser outro sistema também.

Outro conjunto de informações associadas às credenciais são suas permissões de acesso, normalmente chamadas de *papéis* (*roles*) ou *autoridades* (*authorities*), que informam ao sistema quais os limites de atuação do indivíduo. O processo de verificação das permissões do usuário é a autorização, que ocorre sempre após a autenticação no sistema.

ACEGI E SPRING SECURITY SÃO A MESMA COISA?

Em 2003, um grupo de usuários entusiastas do Spring iniciaram um projeto chamado *Acegi Security System for Spring* baseado no suporte a AOP oferecido pelo framework. Com o passar do tempo, este foi se tornando cada vez mais popular até que sua primeira versão oficial foi lançada em 2006. No ano seguinte o projeto foi adotado como parte do portfólio do Spring e em sua segunda versão foi rebatizado como *Spring Security*.

A resposta à pergunta é que o Spring Security que temos hoje é na realidade a evolução do antigo projeto Acegi.

11.2 OS MÓDULOS DO SPRING SECURITY

Assim como o Spring, o framework é composto por uma série de componentes semi independentes que possibilitam ao programador usar apenas aqueles que serão úteis ao seu projeto. A versão 3.1 é composta pelos módulos a seguir:

- `spring-security-core`: o único módulo obrigatório por se tratar da essência do framework. Encontram-se aqui implementados os mecanismos de autorização e autenticação, além de todas as interfaces básicas que são reaproveitadas por todos os demais módulos e componentes terceirizados desenvolvidos para o framework;
- `spring-security-config`: a partir da versão 2.0, o framework passou a oferecer um namespace próprio que reduziu dramaticamente a quantidade de configuração que o desenvolvedor precisava digitar. Dado o ganho de produtividade deve ser considerado obrigatório por todos os não masoquistas;
- `spring-security-web`: toda a infraestrutura necessária para o desenvolvimento de projetos web. Neste módulo é que se encontra presente o código responsável por lidar com a interceptação de requisições a projetos web e portanto de uso obrigatório em projetos deste tipo;
- `spring-security-taglibs`: contém uma biblioteca de tags que nos permite definir quais áreas de nossas páginas serão acessíveis aos usuários do sistema com base em suas permissões. Assim como o módulo *spring-security-config*,

apesar de ser opcional acaba sendo usado sempre por todos aqueles que projetem suas páginas com alguma tecnologia baseada em JSP;

- `spring-security-remoting`: provê integração com o Spring Remoting;
- `spring-security-ldap`: suporte a autenticação a partir de servidores LDAP;
- `spring-security-cas`: suporte a autenticação/autorização por servidores de single sign on CAS;
- `spring-security-openid`: suporte a OpenID.

Assim como fizemos em toda a segunda parte deste livro, usaremos o projeto *Spring Fórum* como nosso guia. Nossa primeira atividade será a inclusão de quatro dependências no arquivo `pom.xml` tal como listadas a seguir:

```
<!-- Spring Security Core -->
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-core</artifactId>
  <version>3.1.3.RELEASE</version>
</dependency>
<!-- Spring Security Config -->
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-config</artifactId>
  <version>3.1.3.RELEASE</version>
</dependency>
<!-- Spring Security Web -->
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-web</artifactId>
  <version>3.1.3.RELEASE</version>
</dependency>
<!-- Spring Security Taglibs -->
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-taglibs</artifactId>
  <version>3.1.3.RELEASE</version>
</dependency>
```

11.3 CONFIGURANDO OS FILTROS DE ACESSO

Primeiro iremos tratar do modo como o Spring Security atua sobre o recebimento de requisições HTTP. Para ativar este suporte, teremos de declarar um filtro no arquivo `web.xml` do nosso projeto que será o responsável por lidar com todas as chamadas externas que nossa aplicação receberá e cuja configuração podemos ver a seguir:

```
<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>
```

Este filtro é o `DelegatingFilterProxy` e seu funcionamento consiste em encaminhar todas as requisições para um componente fundamental do Spring Security que é o `Filter Chain` (*filtro encadeador*), o real responsável pela configuração de acesso às URLs do sistema, e é executado como um bean gerenciado pelo contexto de aplicação do Spring.

Uma aplicação protegida pelo Spring Security pode ter mais de um filtro encadeador, sendo assim é necessário configurar o `DelegatingFilterProxy` de tal forma que este saiba qual o bean com o qual deve interagir. Fazemos isto através da definição do seu nome pela tag `<filter-name>`, que deve corresponder ao identificador deste bean.

Como veremos mais à frente neste capítulo, esta configuração é feita de forma automática pelo Spring Security minimizando o trabalho do programador. Por convenção do framework, o nome padrão deste bean é `springSecurityFilterChain`, o que torna uma boa prática sempre o adotarmos neste ponto da configuração do nosso projeto.

A partir deste ponto do livro faz-se necessária outra alteração em nosso projeto. No capítulo 7 definimos o arquivo básico de configuração do container Spring através da declaração do servlet `DispatcherServlet` tal como no trecho a seguir:

```
<servlet>
  <servlet-name>DispatcherServlet</servlet-name>
```

```
<servlet-class>
    org.springframework.web.servlet.DispatcherServlet
</servlet-class>

<init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/spring-servlet.xml</param-value>
</init-param>

<load-on-startup>1</load-on-startup>
</servlet>
```

O *Spring Fórum* era apenas o contexto deste servlet. A situação mudou agora pois precisamos configurar o contexto do Spring para que seja compartilhado pelo `DispatcherServlet` e nosso filtro. A solução para o problema é a declaração de um listener do tipo `ContextLoaderListener` no arquivo `web.xml` tal como exposto no trecho a seguir:

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

<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/spring/spring-servlet.xml
        /WEB-INF/spring/spring-data.xml
        /WEB-INF/spring/spring-security.xml
    </param-value>
</context-param>
```

Este listener é responsável por carregar um contexto do Spring para a nossa aplicação usando como arquivos de configuração o valor que definimos para o parâmetro de contexto `contextConfigLocation`. Não é necessário alterar a configuração do `DispatcherServlet` que vimos antes.

11.4 CONFIGURANDO O CONTEXTO DO SPRING

Daqui para frente só precisamos nos preocupar com as configurações padrão do Spring. Uma das grandes novidades que o Spring Security trouxe em relação ao Acegi foi o seu esquema XML que reduziu bastante a quantidade de configuração que o programador precisava escrever. Na época do Acegi, não eram raros os casos em que nos víamos dando manutenção em centenas de linhas de configuração. Graças à equipe da SpringSource esta é hoje apenas uma triste recordação.

Para garantir a modularidade de nosso sistema vamos criar um novo arquivo de configuração onde declararemos o namespace `security` usado pelo Spring Security. O corpo do nosso arquivo será similar ao listado a seguir:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:security="http://www.springframework.org/schema/security"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/security
                           http://www.springframework.org/schema/security/spring-security-3.1.xsd">
    <!-- Conteúdo oculto no momento -->
</beans>
```

Nossa primeira configuração é a inclusão da tag `<security:html>` que será a responsável por definir 50% do que o Spring Security precisa para funcionar. Podemos ver sua declaração inicial listada a seguir:

```
<security:http auto-config="true">
    <security:intercept-url
        pattern="/post/publicar/**" access="ROLE_MEMBRO"/>
    <security:intercept-url
        pattern="/usuario/editar/**" access="ROLE_ADMIN,ROLE_MEMBRO"/>
    <security:intercept-url pattern="/**" access="ROLE_ANONYMOUS"/>
</security:http>
```

É incrível a quantidade de tarefas executadas com tão pouco código: `<security:http>` com o atributo `auto-config` definido como `true` faz bastante coisa:

- Criará um formulário de autenticação de forma automática.
- Define as URLs de logout e autenticação a partir de convenções que veremos mais a frente.

- Configura o filtro encadeador (SpringSecurityFilterChain).

Isto sem mencionar que ainda aproveitamos para definir as regras de acesso para três padrões de URL!

Usamos `<security:intercept-url>` para definir as regras de acesso às URLs do sistema. A ordem em que são declaradas é importante, pois sempre que uma requisição chega ao *Filter Chain* esta é validada contra todas as regras de cima para baixo. O atributo `pattern` define o padrão da URL usando a sintaxe do Ant, enquanto `access` contém as regras de autorização. Neste primeiro momento usamos a notação mais simples possível que consiste na mera listagem de todas as permissões (roles) separadas por vírgula.

Atenção especial deve ser dada à terceira declaração de `<security:intercept-url>`. Observe o valor que definimos para o atributo `access: ROLE_ANONYMOUS`. Este é um valor especial para o Spring Security que indica a situação na qual o usuário tentando acessar nossa aplicação não se encontra autenticado. Sendo assim, imagine que invertêssemos a ordem em que declaramos as tags `<security:intercept-url>`. Neste caso, teríamos acesso liberado a qualquer um, pois o teste de autorização passaria para a url representada: `/* *`, que aponta para qualquer endereço que comece a partir da raiz do projeto.

Com `<security:intercept-url/>` também podemos habilitar suporte a HTTPS, que normalmente é aplicado em seções sensíveis do site, como por exemplo no checkout de compras em uma loja virtual, aonde dados confidenciais do usuário são enviados. Para tal, basta adicionar o atributo `requires-channel`, que atualmente recebe três possibilidades de valor: `https`, `http` e `any` (tanto faz se a URL for acessada por HTTP ou HTTPS). Podemos ver alguns exemplos de sua aplicação na listagem a seguir:

```
<!-- HTTPS aplicado -->
<intercept-url pattern="/checkout/**"
    access="ROLE_MEMBRO" requires-channel="https"/>
<!-- HTTP padrão -->
<intercept-url pattern="/**"
    access="ROLE_MEMBRO" requires-channel="http"/>
<!-- Tanto faz -->
<intercept-url pattern="/posts/**"
    access="ROLE_MEMBRO" requires-channel="any"/>
```

Caso o usuário acesse uma URL com o canal pré-definido com o protocolo errado - http ao invés de https por exemplo - o Spring Security fará o redirecionamento

de forma automática.

Como dito no início desta seção, a tag `<security>` resolve 50% do nosso problema ao definir as regras de *autorização* das requisições recebidas pela aplicação e configura o básico por trás do controle de acesso neste nível. Sabemos o *quê* está a ser acessado, resta saber *por quem*. Entram em cena dois componentes importantes do framework: o provedor e o gerenciador de autenticação (*authentication provider* e *authenciation manager*).

O Gerenciador e o provedor de autenticação

O gerenciador de autenticação é o responsável por receber as credenciais de um usuário e verificar sua autenticidade, o que é feito testando-as contra um ou mais provedores de autenticação a que tem acesso. Esta verificação é feita sequencialmente, ou seja, o primeiro provedor de autenticação que confirmar a existência de um usuário com estas credenciais valida-o como autêntico.

Pense no provedor de autenticação como uma fonte de dados feita especificamente para armazenar informações de usuários capaz de executar uma única consulta que recebe como parâmetro as credenciais retornando informações do usuário caso existam (incluindo suas permissões de acesso).

No contexto do controle de acesso a URLs o *Chain Filter* acessa apenas o gerenciador de autenticação, não precisando portanto se preocupar com a origem dos dados de autenticação do usuário tal como pode ser visto no esquema a seguir:



Figura 11.1: Chain Filter, Authentication Manager e seus Authentication Providers

Há inúmeros provedores de autenticação disponíveis para o Spring Se-

curity atualmente. Como exemplos podemos citar o suporte a LDAP, bancos de dados, registros em memória, além do suporte a redes sociais como Twitter, Facebook, LinkedIn, Google+ e outras. Por trás da cortina, um provedor de autenticação é apenas uma classe que implementa a interface `org.springframework.security.authentication.AuthenticationProvider`, sendo assim caso não exista um que atenda suas necessidades, facilmente podemos criar um, tal como faremos mais a frente neste capítulo.

Indo para a parte prática, é necessário que declaremos em nosso arquivo de configuração nosso gerenciador de autenticação. Para tal usamos `<security:authentication-manager>`, que receberá como filhos um ou mais elementos do tipo `<security:authentication-provider>`, representando os provedores de autenticação usados em nosso projeto tal como na listagem a seguir:

```
<security:authentication-manager>
  <security:authentication-provider>
    <security:user-service>
      <security:user name="bruce"
        password="lee" authorities="ROLE_MEMBRO"/>
      <security:user name="admin"
        password="admin" authorities="ROLE_MEMBRO,ROLE_ADMIN"/>
    </security:user-service>
  </security:authentication-provider>
</security:authentication-manager>
```

A tag `<security:user-service>` representa o provedor de autenticação mais simples disponibilizado pelo Spring Security: este armazena os dados dos usuário em memória com base na declaração de uma ou mais tags `<security:user>`, que definem o nome, senha e permissões dos usuários gerenciados.

Como podemos ver, este é declarado dentro da tag `<security:authentication-provider>` no interior de `<security:authentication-manager>`. Posteriormente iremos adicionar nossa própria implementação neste gerenciador de autenticação e assim finalmente veremos como estes atores interagem entre si por trás dos panos.

A interação entre nosso filtro e o gerenciador de autenticação se da em dois momentos:

- Usuário tenta se autenticar no sistema: há um mapeamento oculto na tag `<security:html>` que liga a URL do formulário de login por esta configurado e o nosso gerenciador de autenticação. Quando este recebe as credenciais

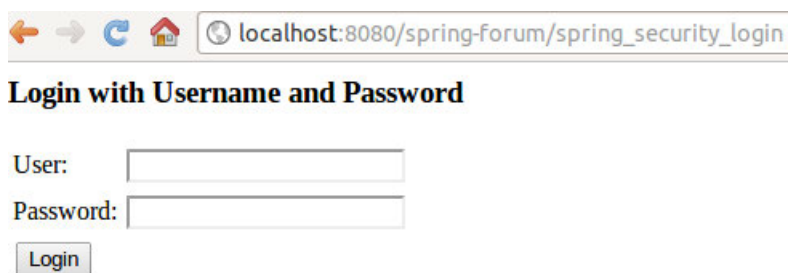
do usuário, verifica se existe um registro de usuário que corresponda a estas e o autentica no sistema, em seguida o retornando ao filtro.

- Usuário já está autenticado no sistema: o gerenciador de autenticação retornará as permissões do usuário, que serão verificadas contra as regras definidas pelo mapeamento da URL que corresponda ao padrão desejado pelo usuário.

Este é o nosso primeiro contato com estas entidades, para clarear um pouco as coisas, agora que estamos com nosso projeto inteiramente configurado, vamos tentar acessar a url de padrão `/post/publicar/*` * sem estarmos autenticados e ver o que acontece.

11.5 O FORMULÁRIO DE LOGIN

Iniciando o projeto e acessando a URL `http://localhost:8080/spring-forum/post/publicar` somos saudados pelo formulário de login gerado automaticamente pelo Spring Security:



The screenshot shows a web browser window with the address bar displaying `localhost:8080/spring-forum/spring_security_login`. The page content includes the heading **Login with Username and Password**. Below the heading, there are two text input fields: the first is labeled 'User:' and the second is labeled 'Password:'. At the bottom of the form is a button labeled 'Login'.

Figura 11.2: Formulário de login gerado pelo Spring Security

O filtro ao verificar que não existia um usuário autenticado e uma regra se aplicava à URL em questão faz um redirecionamento para o endereço padrão `/spring_security_login`, que aponta para o página que contém o formulário de autenticação gerado automaticamente pelo Spring Security. Caso o usuário forneça o login e senha corretos e o usuário encontrado possua as permissões necessárias para acessar o recurso, este se tornará disponível a este sem problema.

É interessante dar uma olhada no código fonte deste formulário listado no trecho a seguir pois nele veremos algumas convenções adotadas pelo framework:

```

<form name='f' action='/spring-forum/j_spring_security_check'
      method='POST'>
  <table>
    <tr>
      <td>User:</td>
      <td><input type='text' name='j_username' value=''></td>
    </tr>
    <tr>
      <td>Password:</td>
      <td><input type='password' name='j_password'></td>
    </tr>
    <tr>
      <td colspan='2'>
        <input name="submit" type="submit" value="Login"/>
      </td>
    </tr>
  </table>
</form>

```

A primeira destas convenções diz respeito à URL de autenticação: `j_spring_security_check`. Em seguida, observamos também o nome dos parâmetros incluídos no formulário: `j_username` e `j_password`, usados para representar o login e senha do usuário. Tudo o que o programador precisa fazer é copiar esta marcação, alterá-la para que fique de acordo com o layout do seu site e alterar algumas configurações básicas do formulário de autenticação no contexto do Spring tal como fizemos no caso do *Spring Fórum* listado a seguir:

```

<security:http auto-config="true" use-expressions="true">
  <security:form-login
    login-page="/"
    default-target-url="/"
    authentication-failure-url="/"/>
  <!-- Restante omitido -->
</security:http>

```

Adicionamos a nossa própria configuração do formulário de login usando `<security:form-login/>`, que antes era gerada de forma automática pelo framework. A primeira alteração feita diz respeito à URL aonde será exposto o formulário de login. Em nosso projeto este é exposto em todas as páginas, mas padronizamos o redirecionamento para a página inicial caso algum usuário não autenticado tente

acessar uma área restrita. Abaixo podemos ver como ficou o nosso formulário de autenticação em destaque:



Figura 11.3: Nosso formulário de login

Outro atributo importante que alteramos foi `default-target-url`, que define qual a URL a ser acessada caso a autenticação seja bem sucedida, e `authentication-failure-url`, que define qual a URL a ser exposta em caso de um login mal sucedido. No caso do *Spring Fórum*, ambas apontam para a página inicial. E isto é tudo o que precisamos saber neste momento a respeito do formulário de autenticação.

11.6 ESCRREVENDO UM PROVEDOR DE AUTENTICAÇÃO

De volta aos provedores de autenticação, a criação de um customizado nos ajudará a entender melhor o mecanismo por trás do Spring Security. Precisamos conhecer apenas três interfaces: `AuthenticationProvider`, `Authentication` e `GrantedAuthority`. O modo como estas se relacionam com o provedor de autenticação é ilustrado no diagrama de classes a seguir:

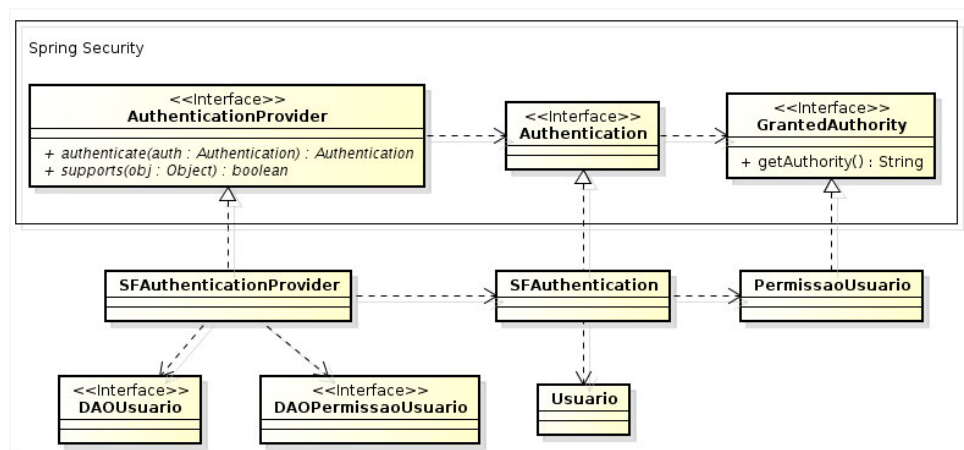


Figura 11.4: Nosso provedor de autenticação

A principal interface é `AuthenticationProvider`, que como o próprio nome já nos diz, representa um provedor de autenticação para o Spring Security. Como pode ser visto no diagrama, esta é bastante simples, forçando o programador a implementar apenas dois métodos. Começamos pelo mais simples que é a função `supports`, que recebe como parâmetro um valor do tipo `java.lang.Object` que representa um objeto de autenticação suportado pelo nosso provedor.

Um objeto de autenticação é o responsável por armazenar as credenciais do usuário. O Spring Security já vêm com alguns, sendo o mais popular a classe `UsernamePasswordAuthenticationToken`, que representa um objeto de autenticação do tipo login e senha: exatamente o usado por formulários de autenticação.

A seguir podemos ver como implementamos esta função na classe `SFAuthenticationProvider`:

```

public boolean supports(Class<?> authentication) {
    return (UsernamePasswordAuthenticationToken.
        class.isAssignableFrom(authentication));
}

```

A outra função a ser implementada é `authenticate`, que retorna um objeto do tipo `Authentication`, que é o responsável por armazenar os dados de autenticação do usuário, o que inclui sua lista de permissões. É interessante observar que este objeto é usado em dois momentos: no primeiro contém apenas as credenciais do

usuário que verificaremos contra nossa base de dados, e no segundo, caso o teste dê positivo, é usado internamente pelo framework no processo de autorização.

A seguir podemos ver a listagem da nossa implementação desta função:

```
public Authentication authenticate(Authentication auth)
    throws AuthenticationException {
    UsernamePasswordAuthenticationToken token =
        (UsernamePasswordAuthenticationToken) auth;
    String username = token.getName();
    String senha = token.getCredentials().toString();
    Usuario usuario = getDaoUsuario().getUsuario(username, senha);
    /*
        Não encontrei um usuário, retornar null indica
        uma autenticação falha.
    */
    if (usuario == null) return null;

    List<PermissaoUsuario> permissoes = getDaoPermissao().
        getPermissoesUsuario(usuario);
    SFAuthentication resultado =
        new SFAuthentication(usuario, permissoes);
    resultado.setAuthenticated(usuario != null);
    return resultado;
}
```

Em nosso caso, criamos nossa própria implementação da interface `Authentication`: `SFAuthentication`. Com ela apenas encapsulamos duas informações: a nossa instância da classe `Usuario` e a lista de permissões relativas a este, que são obtidas através do DAO `DAOPermissao`, que nos retorna uma lista de objetos do tipo `PermissaoUsuario`.

A classe `PermissaoUsuario` implementa a interface `GrantedAuthority`, que possui apenas uma função, chamada `getAuthority`, que retorna um valor do tipo `String`, que é o nome da permissão dentro do sistema.

Para finalizar o nosso provedor de autenticação, tudo o que precisamos fazer é declará-lo no arquivo de configurações tal como é feito na listagem a seguir:

```
<!-- O bean que representa nosso provedor -->
<bean id="sfAuthenticationProvider"
    autore="byType"
    class="br.com.itexto.springforum.security.SFAuthenticationProvider"/>
```



```

<security:authentication-manager>
  <!-- Nosso provedor -->
  <security:authentication-provider ref="sfAuthenticationProvider"/>
  <!-- Provedor baseado em memória -->
  <security:authentication-provider>
    <security:user-service>
      <security:user name="jimmy" password="hendrix"
        authorities="ROLE_MEMBRO"/>
    </security:user-service>
  </security:authentication-provider>
</security:authentication-manager>

```

Usando o atributo `ref` da tag `<security:authentication-provider>` basta referenciar o bean correspondente ao nosso provedor de autenticação e pronto, este se encontra na lista de provedores usadas pelo gerenciador de autenticação do sistema.

11.7 USANDO SPEL

Até este ponto do capítulo vimos apenas como definir estaticamente as permissões aos recursos da aplicação, o que é uma solução bastante limitada. Há outro modo caminho: assim como SpEL trouxe dinamismo para os nossos arquivos de configuração, este também se aplica ao Spring Security e o que é melhor: foi estendido para que nos ofereça máxima flexibilidade.

Podemos aplicar as expressões em todos os pontos do framework: nos arquivos de configuração, assim como nas anotações e taglibs que veremos mais adiante neste capítulo. Para habilitar seu uso no arquivo de configuração, tudo o que precisamos fazer é adicionar o atributo `use-expressions` com o valor `true` como foi feito no trecho a seguir:

```

<security:http auto-config="true" use-expressions="true">
  <!-- Restante omitido -->
  <security:intercept-url pattern="/post/publicar/**"
    access="hasRole('ROLE_MEMBRO')"/>
  <security:intercept-url pattern="/**"
    access="isAnonymous()"/>
</security:http>

```

A função mais comum é `hasRole`, que recebe uma string representando o nome

da permissão, e é excelente para quando queremos lidar com apenas uma permissão, mas a coisa fica meio complicada quando precisamos combinar mais de uma como no exemplo abaixo:

```
<security:intercept-url pattern="/membro/gerenciar"
    access="hasRole('ROLE_MEMBRO') or hasRole('ROLE_ADMIN')"/>
```

Neste caso, usamos `hasAnyRole`, que recebe uma lista de papéis separadas por vírgula como em:

```
<security:intercept-url pattern="/membro/gerenciar"
    access="hasAnyRole('ROLE_MEMBRO', 'ROLE_ADMIN')"/>
```

E claro, podemos combinar estas duas expressões também se quisermos, como no exemplo a seguir:

```
<spring:security-url pattern="/membro/gerenciar"
    access="hasAnyRole('ROLE_ADMIN', 'ROLE_MODERADOR')
        and ! hasRole('ROLE_MEMBRO')"/>
```

Além destas funções embutidas, há a listagem abaixo que o desenvolvedor pode usar de acordo com a sua necessidade:

- `principal`: aponta para o *principal* do usuário autenticado.
- `authentication`: nos fornece acesso ao objeto `Authentication` corrente.
- `permitAll`: sempre retorna `true`, permitindo que qualquer um acesse o recurso.
- `denyAll`: sempre retornará `false`, negando sempre o acesso.
- `isAnonymous()`: verifica se estamos lidando com um usuário não autenticado.
- `isAuthenticated()`: verifica se estamos lidando com um usuário autenticado.

11.8 PROTEGENDO INVOCÇÃO DE MÉTODOS EM BEANS

Como dito no início deste capítulo, o Spring Security se aplica a dois níveis: requisições e invocação de métodos. A maneira mais produtiva de tirar proveito deste recurso se dá através da anotação `@Secured`, definida no pacote `org.springframework.security.access.annotation`. Podemos ver um exemplo da sua aplicação no código a seguir:

```
public class UsuarioService {
    // Usando SpEL
    @Secured("hasRole('ROLE_ADMIN')")
    public void banir(Usuario usuario) {
        ...
    }
    // Passando uma lista de permissões
    @Secured({"ROLE_ADMIN", "ROLE_MODERADOR"})
    public void advertir(Usuario usuario, String mensagem) {
        ...
    }
}
```

Como já era de se imaginar, o Spring Security exerce o controle de acesso através do suporte a AOP. Sendo assim, faz-se necessário que adicionemos uma única linha em nosso arquivo de configuração, tal como exposto no código a seguir:

```
<security:global-method-security secured-annotations="enabled"/>
```

Difícil pensar em uma solução mais simples: agora todo bean gerenciado pelo container do Spring que possua a anotação `@Secured` em seus métodos ou na sua declaração de classe (o que irá aplicar a regra a todos os métodos do bean) terá seu acesso gerenciado pelo Spring Security.

O interessante do suporte a invocação de métodos é que o horizonte de aplicação do Spring Security se amplia: aplicações desktop poderão tirar proveito dos mesmos recursos.

11.9 TAGS

O Spring Security também atua na camada de visualização através da sua biblioteca de tags que podem ser aplicadas a páginas JSP ou tecnologias derivadas. Estas irão executar basicamente duas funções: liberar/negar acesso e expor informações sobre o usuário corrente.

Para tirar proveito destas tags, o primeiro passo a ser dado pelo programador é declará-la no topo do arquivo JSP exatamente como no trecho a seguir:

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

De longe a tag mais usada é `<sec:authorize>`, que libera a renderização do trecho JSP que envolve de acordo com as permissões definidas no atributo `access` tal como listado a seguir:

```
<sec:authorize access="isAnonymous()">
<!-- Exibe o formulário de login --%>
</sec:authorize>

<sec:authorize access="isAuthenticated()">
<!-- Exibe conteúdo para o usuário autenticado --%>
</sec:authorize>
```

Outra tag a nossa disposição é `<sec:authentication/>`, que exibirá na página o valor da propriedade do objeto `Authentication` do usuário autenticado. Na página inicial do Spring Fórum usamos isto para expor a mensagem de boas vindas e o link de logout, tal como no exemplo a seguir:

```
<sec:authorize access="isAuthenticated()">
  Bem vindo(a) <sec:authentication property="principal"/> -
  <a href="<c:url value="/j_spring_security_logout"/>">Sair</a>
</sec:authorize>
```

11.10 CONCLUSÃO

O Spring Security é um dos melhores exemplos práticos de aplicação da AOP. Como o leitor pode comprovar no transcorrer deste capítulo, em momento algum precisamos modificar nosso código fonte - com exceção da inclusão de uma anotação - para controlar seu acesso. É interessante observar também que sua configuração foi feita de uma forma bastante simples: ao final escrevemos algo em torno de 10 linhas em nosso arquivo de configuração e pronto: nosso controle de acesso estava pronto.

É muito importante que o leitor procure por componentes desenvolvidos por terceiros voltados ao Spring Security. Não seria exagero afirmar que há 90% de chance de alguém já ter escrito um provedor de autenticação que atenda às suas necessidades e caso este não exista, bom: já vimos que escrever o próprio não é uma tarefa difícil.

CAPÍTULO 12

E ai, gostou?

No transcorrer deste livro foi exposto o básico por trás do funcionamento do Spring Framework. Meu projeto inicial para sua escrita foi dividi-lo em duas partes: na primeira meu objetivo foi descrever os conceitos por trás da ferramenta - inversão de controle, injeção de dependências, AOP e outros - como eu gostaria de tê-los conhecido em um primeiro momento: expondo as razões que justificam sua existência. Coisas que gostaria muito de ter aprendido na faculdade.

Na segunda parte incluí os assuntos que gostaria que tivessem me sido apresentados no ambiente de trabalho. O suporte a dados do Spring é um bom exemplo: assim como diversos usuários do framework, fiquei anos sem tirar proveito deste suporte porque, apesar de ter ouvido falar de sua existência em livros, raríssimas vezes encontrei material que me apresentasse aquelas funcionalidades de forma clara.

Se tiver atingido os objetivos acima, dou-me por satisfeito, porém o seu caminho no aprendizado do Spring não termina nestas últimas páginas, mesmo porque as limitações deste livro me impediram de incluir todas as possibilidades existentes. Basicamente o que posso dizer ao leitor é: toda classe pode ser tratada como um

bean no Spring, o que torna o trabalho por trás da integração de qualquer tecnologia com o framework consiste apenas em sua declaração como tal.

A meta-prática: o código fonte do livro

Para facilitar o aprendizado do leitor, conforme ia escrevendo este livro desenvolvi um projeto paralelo chamado *Spring Fórum*, que ainda é bastante rudimentar no momento em que entreguei a versão final deste livro aos editores - a quem agradeço pela incrível paciência - mas que contém as aplicações referentes à toda a segunda parte do livro. Seu código fonte está disponível no GitHub em <https://github.com/loboweissmann/spring-forum>

Já com relação ao código fonte da primeira parte do livro, você encontra o código fonte do livro no github: <https://github.com/loboweissmann/spring-forum>

Futuro do Spring

Com o passar do tempo é inegável o fato de que a plataforma Java EE acabou se tornando a padronização da clássica dobradinha *Spring + Hibernate*. Muitos leitores se perguntam portanto se ainda vale à pena dedicar-se ao aprendizado do Spring. Sei que sou suspeito para falar a respeito, mas minha opinião é a de que sim, é um aprendizado válido e continuará a sê-lo por muitos anos.

É nos frameworks que surgem as inovações. Observando a evolução da plataforma Java EE podemos ver que a esmagadora maioria das “novidades” que nesta vão sendo incorporadas são na realidade o fruto da influência destes frameworks. E a grande diferença entre ser um inovador e ser um padrão é que o segundo normalmente se baseia no menor denominador comum, deixando os recursos mais específicos - e normalmente mais poderosos - presentes na fonte original.

Outra razão, e talvez a mais importante, seja o fato do Spring ser uma ferramenta disciplinadora. Com ele podemos ver nitidamente o poder de uma boa abstração através da implementação de interfaces, divisão de responsabilidades, tudo baseado no conceito de inversão de controle/injeção de dependências. Por si só, mesmo que o leitor jamais use o Spring na prática, este já justificaria seu aprendizado.

Obrigado

Pra finalizar, agradeço ao leitor que tenha chegado a esta última página do livro. Espero que com este trabalho tenha tornado mais claro o que está por trás do Spring (e talvez qualquer framework) e como tirar proveito desta ferramenta para tornar

seu trabalho mais fácil. Agradeço também ao Paulo Silveira e Adriano Almeida, editores deste trabalho pela sua paciência, confiança e dedicação na finalização deste trabalho, pois sem estes nada seria possível.

E até a próxima, pois não parei por aqui! :)

Referências Bibliográficas

- [1] Fred Brooks. *The Mythical Man-Month*. Addison-Wesley.
- [2] Fred Brooks. No silver bullet - essence and accidents of software engineering. <http://www.cs.nott.ac.uk/~cah/G51ISS/Documents/NoSilverBullet.html>.
- [3] Time de desenvolvimento do Apache Tiles. The composite view pattern. <http://tiles.apache.org/2.2/framework/tutorial/pattern.html>, ?
- [4] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1994.
- [5] Martin Fowler. Inversion of control containers and the dependency injection pattern. <http://martinfowler.com/articles/injection.html>, 2004.
- [6] Jesse James Garrett. Ajax: A new approach to web applications. <http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications>.
- [7] Rod Johnson. *Expert One-To-One J2EE Design and Development*. Wrox Press, 2002.
- [8] Rod Johnson. *Expert One-To-One J2EE Development Without EJB*. Wrox Press, 2004.
- [9] Martin Lippert. Springsource tool suite 2.9.1 installation instructions. http://download.springsource.com/release/STS/doc/STS-installation_instructions.pdf, 2012.
- [10] Robert C. Martin. The dependency inversion principle. <http://www.objectmentor.com/publications/dip.pdf>, 1996.

- [11] Robert C. Martin. Design principles and design patterns. http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf, 2009.
- [12] Brett McLaughlin. *Use a Cabeça - Ajax*. Alta Books.
- [13] Mark Richards. *Java Transaction Design Strategies*. InfoQ, 2006.
- [14] Spring Source. Spring reference documentation 3.1. <http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/pdf/spring-framework-reference.pdf>, 2012.
- [15] SpringSource. Springsource case studies. <http://www.springsource.org/case-studies>.
- [16] Craig Walls. *Spring in Action*. Manning, 2011.