



Programação funcional aplicada com Aspectos
Domine conceitos destes
paradigmas na teoria e na prática

Construa ambientes de alta disponibilidade
Aprimorando o desempenho
do processamento distribuído

Date and Time API
Conheça um dos destaques do Java 8

 DEV MEDIA

TUNING COM HIBERNATE

Aprimorando o
código em busca
de desempenho



Desenvolvendo código de qualidade
Saiba como otimizar
o seu código Java

Criando uma aplicação corporativa
Desenvolvendo a camada de
visão com JavaServer Faces

ISSN 1676836-1



MVP

R\$ 1.000.000,00
INVESTIDOS EM CONTEÚDO
NOS ÚLTIMOS 12 MESES.

APlique esse investimento
na sua carreira...

E mostre ao mercado
quanto você vale!

CONFIRA TODO O MATERIAL
QUE VOCÊ TERÁ ACESSO:

- + de **9.000** video-aulas
- + de **290** cursos online
- + de **13.000** artigos
- DEVMEDIA API's**
consumido + de **500.000** vezes



POR APENAS
R\$ 69,90* mensais

*Tempo mínimo de assinatura: 12 meses.



PRA QUEM QUER EXIGIR
MAIS DO MERCADO!

 **DEVMEDIA**



Edição 139 • 2015 • ISSN 1676-8361



MVP

Assine agora e tenha acesso a todo o conteúdo da DevMedia:
www.devmedia.com.br/mvp

EXPEDIENTE

Editor

Eduardo Spínola (eduspinola@gmail.com)

Consultor Técnico Diogo Souza (diogosouzac@gmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa Romulo Araujo

Diagramação Janete Feitosa

Distribuição

FC Comercial e Distribuidora S.A.

Rua Teodoro da Silva, 907, Grajaú - RJ

CEP 20563-900, (21) 3879-7766 - (21) 2577-6362

Atendimento ao leitor

A DevMedia possui uma Central de Atendimento on-line, onde você pode tirar suas dúvidas sobre serviços, enviar críticas e sugestões e falar com um de nossos atendentes. Através da nossa central também é possível alterar dados cadastrais, consultar o status de assinaturas e conferir a data de envio de suas revistas. Acesse www.devmedia.com.br/central, ou se preferir entre em contato conosco através do telefone 21 3382-5038.

Publicidade

publicidade@devmedia.com.br – 21 3382-5038

Anúncios – Anunciando nas publicações e nos sites do Grupo DevMedia, você divulga sua marca ou produto para mais de 100 mil desenvolvedores de todo o Brasil, em mais de 200 cidades. Solicite nossos Media Kits, com detalhes sobre preços e formatos de anúncios.

Java, o logotipo da xícara de café Java e todas as marcas e logotipos baseados em/ou referentes a Java são marcas comerciais ou marcas registradas da Sun Microsystems, Inc. nos Estados Unidos e em outros países.

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique à vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site Java Magazine, entre em contato com o editor, informando o título e mini-resumo do tema que você gostaria de publicar:



EDUARDO OLIVEIRA SPÍNOLA

eduspinola.wordpress.com

@eduspinola / @Java_Magazine

CURSOS ONLINE

A Revista Java Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.



CONHEÇA ALGUNS DOS CURSOS:

- **Curso de noSQL (Redis) com Java**
- **Curso Básico de JDBC**
- **Java Básico: Aplicações Desktop**
- **JSF com Primefaces**
- **Conhecendo o Apache Struts**

Para mais informações :

www.devmedia.com.br/curso/javamagazine

(21) 3382-5038



Sumário

Artigo no estilo Curso

06 – Criando uma aplicação corporativa em Java – Parte 3

[Bruno Rafael Sant'Ana]

Conteúdo sobre Novidades, Artigo no estilo Curso

18 – Construindo e otimizando um ambiente de alta disponibilidade com Hazelcast – Parte 2

[Cleber Muramoto]

Artigo do estilo Mentoring

34 – Hibernate: Evitando problemas de mapeamento

[Alessandro Jatoba]

Conteúdo sobre Novidades

44 – Date and Time API: Conheça um dos destaques do Java 8

[Leonardo Comelli]

Conteúdo sobre Boas Práticas

54 – Como desenvolver um código fonte de qualidade em Java

[Luis Cesar de Souza Moura]

Conteúdo sobre Novidades, Conteúdo sobre Boas Práticas

61 – Programação funcional aplicada com Aspectos

[Rômero Ricardo de Sousa Pereira]



Dê seu feedback sobre esta edição!

A Java Magazine tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/javamagazine/feedback



CRIANDO APLICAÇÕES INOVADORAS QUE VÃO MUDAR O MUNDO?

DESENVOLVA COM AS FERRAMENTAS QUE OS MELHORES DESENVOLVEDORES
USAM: GIT, REDMINE, JENKINS, SONAR, MAVEN E MUITO MAIS.

A suite ToolsCloud é um conjunto integrado das melhores ferramentas para desenvolvimento de software do mercado.

Um ambiente pronto para usar e com suporte, que os desenvolvedores curtem e os gerentes se surpreendem.

- Aumenta a produtividade;
- Facilita a colaboração e comunicação;
- Maior visibilidade e acompanhamento;
- Melhor planejamento e mais agilidade.

Na suite ToolsCloud você tem a sua disposição:

- As ferramentas mais usadas, integradas e de uso imediato;
- Repositório de código (git e svn), integração contínua (Jenkins/Hudson), Gerência de pendências (Redmine), e mais Maven, Nexus, Sonar;
- Infraestrutura robusta, em ambiente de nuvem, suporte e segurança;
- Facilita práticas de agilidade e lean startups;
- Ambiente isolado, customizável e flexível;
- Colaboração garantida: wiki, fóruns de discussão, chat (IRC).

Acompanhe a Toolscloud:



toolscloud@toolscloud.com



twitter.com/toolscloud



Criando uma aplicação corporativa em Java – Parte 3

Como criar a camada de visão utilizando o JavaServer Faces

ESTE ARTIGO FAZ PARTE DE UM CURSO

Ao longo desta série de artigos estamos desenvolvendo uma aplicação web que possibilita o gerenciamento de bibliotecas. Até o presente momento já criamos o projeto no Eclipse, configuramos o arquivo *persistence.xml*, realizamos o mapeamento de nossas entidades, implementamos o DAO e uma classe para gerenciar as transações. No entanto, o cliente ainda não consegue interagir com o sistema, pois ainda não criamos as páginas que irão conter a parte da interface gráfica.

Sendo assim, a partir de agora vamos desenvolver as principais páginas desta aplicação. São elas:

- *_template_simple.xhtml* e *_template.xhtml* – que representam os arquivos de template (veremos mais detalhes sobre eles adiante);
- *login.xhtml* – página para que o usuário do sistema possa se autenticar;
- *principal.xhtml* – página que contém os links que dão acesso às outras partes do sistema;
- *funcionario_biblioteca.xhtml* – página que permite que sejam cadastrados dados relacionados aos novos funcionários da biblioteca e um usuário e senha para cada um deles;
- *emprestimo.xhtml* – página que possibilita o empréstimo de livros;
- *devolucao.xhtml* – página para que se possa realizar a devolução dos livros emprestados.

Fique por dentro

No terceiro artigo da série será desenvolvida toda a interface gráfica do sistema para gerenciamento de bibliotecas. Para isso, aprenderemos a criar as páginas utilizando os principais componentes do JSF e a associá-los a propriedades de um managed bean. Explicaremos também o que é um managed bean, qual sua relação com a anotação `@Named`, do CDI, e porque usar `@Named` em vez de `@ManagedBean`.

Ainda nesse artigo, durante o desenvolvimento da aplicação, passaremos pela API do JSF mostrando de forma prática como utilizar em conjunto as classes **FacesContext** e **NavigationHandler** para navegar entre as views. Portanto, tal conteúdo é de grande valia para os leitores que desejam aprender mais sobre uma das principais tecnologias do Java: o JavaServer Faces.

Estas páginas irão conter os componentes visuais que compõem a interface gráfica da nossa aplicação. A camada de visão (que será composta por tais páginas) ficará separada das demais camadas pelo fato de o JSF ser um framework MVC. Ademais, como existe essa separação de responsabilidades, a lógica de negócios deve ficar dentro dos managed beans. Em suma, a ideia central é que as páginas trabalhem em conjunto com os managed beans – que contêm o código Java com a lógica de negócio.

Dito isso, daremos sequência ao desenvolvimento do nosso sistema, criando as páginas JSF, os managed beans e algumas outras classes relativas à navegação.

Criando as páginas JSF e os Managed Beans

Com as entidades e as classes DAO prontas, iremos iniciar o desenvolvimento das páginas JSF e dos managed beans.

Nossas páginas irão utilizar imagens e um arquivo .css para formatação. Estes arquivos (que podem ser baixados do projeto exemplo disponível no GitHub) devem ser colocados na pasta *resources*, que precisa ser criada dentro da pasta *WebContent* do projeto. Por estar fora do escopo deste artigo, não exibiremos o conteúdo do arquivo CSS.

Templates

Uma boa prática que devemos adotar durante o desenvolvimento de um sistema é evitar a duplicidade de código e isso também vale para nossas páginas. Uma maneira de evitarmos isso é criando arquivos de template com a biblioteca de tags Facelets. Um arquivo de template deve agrupar os códigos que são comuns a várias páginas. No caso da nossa aplicação, os templates definirão o cabeçalho e o rodapé, pois são coisas comuns a todas as páginas. Fazendo uma analogia, imagine um template como se fosse uma folha de papel com buracos, e esses buracos são locais onde podemos encaixar diversos conteúdos diferentes. Assim, cada cliente do template (*template client*), no caso a página que usa o template, poderia inserir um conteúdo diferente em cada um desses “buracos”.

Criaremos dentro de *WEB-INF* dois arquivos de template: um chamado *_template_simples.xhtml* e outro chamado *_template.xhtml*. Mantê-los dentro de *WEB-INF* é uma forma de evitar que sejam acessados diretamente pelo navegador através de uma URL.

A **Listagem 1** mostra o conteúdo da página *_template_simples.xhtml*. Para que as tags do Facelets funcionem nessa página, precisamos declarar o namespace *xmlns:ui="http://java.sun.com/jsf/facelets*, o que é feito logo no início do arquivo. Repare que em meio ao código dessa listagem encontramos a tag *ui:insert*, que faz parte da biblioteca Facelets. O prefixo *ui* é o que sinaliza que essa tag pertence a tal biblioteca. Note que *ui* é o mesmo valor utilizado na declaração do namespace. O propósito de *ui:insert* é demarcar as partes do template que podem ser substituídas pelas páginas clientes que irão usar esse template.

Seguindo com a análise do código, podemos observar que o corpo dessa página possui quatro divs: a primeira representa o cabeçalho e contém uma imagem; a segunda mostra o usuário logado e tem um botão para fazer logout; a terceira é reservada para comportar o conteúdo principal da página (é a que contém a tag *ui:insert*); e a quarta e última div representa o rodapé. Na div com id *usuarioLogado* o managed bean *usuarioLogadoBean* (veremos seu código adiante) é utilizado para exibir o nome do usuário logado caso algum já tenha se autenticado.

A outra página de template, *template.xhtml*, que veremos a seguir, tem basicamente a mesma estrutura, contendo apenas algumas tags a mais dentro da div do conteúdo. Como todas as outras páginas serão clientes ou da página *_template_simples.xhtml* ou da página *_template.xhtml*, terão por consequência essa estrutura também e irão apenas substituir o conteúdo principal.

Como mencionado anteriormente, o conteúdo do arquivo *_template.xhtml* é quase igual ao do arquivo *_template_simples.xhtml*. Devido a isso, na **Listagem 2** é mostrada somente a parte diferente

entre os dois para evitar listagens com código duplicado. O que muda entre os dois é o que está presente dentro da div **conteúdo**. Nesse local, no arquivo *_template.xhtml*, usamos a tag *h:messages* para mostrar ao usuário as mensagens que forem adicionadas através do método *addMessage()* da classe *FacesContext*. Logo abaixo é usada a tag *ui:insert*, que já foi explicada. Por último, através da tag *h:commandLink* é criado um link que aponta para o menu principal, que é apenas uma página que contém os links para as demais. Lembre-se que a tag *h:commandLink* precisa estar dentro de *h:form* para funcionar.

Listagem 1. Código do arquivo *_template_simples.xhtml*.

```
<%xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ui="http://java.sun.com/jsf/facelets">

<h:head>
  <title>Easy - Sistema Gerenciador de Biblioteca</title>
  <h:outputStylesheet library="css" name="style.css"/>
</h:head>

<h:body>
  <div id="cabecalho">
    <h:graphicImage library="imagens" name="logo-easy.png" id="logoCompany"/>
  </div>

  <div id="usuarioLogado">
    <h:form rendered="#{usuarioLogadoBean.logado}">
      Logado como: #{usuarioLogadoBean.usuario.usuario}
      <h:commandButton value="[Sair]" action="#{loginBean.efetuarLogout}" />
    </h:form>
  </div>

  <div id="conteudo">
    <ui:insert name="corpo" />
  </div>

  <div id="rodape">
    Copyright 2014.
    Todos os Direitos reservados a Easy Java Magazine
  </div>
</h:body>
</html>
```

Listagem 2. Código do arquivo *_template.xhtml*.

```
<!-- restante do código ... -->
<div id="conteudo">
  <br/>
  <h:form>
    <h:messages globalOnly="true" showDetail="true" />
  </h:form>
  <br/>
  <ui:insert name="corpo" />
  <h:commandLink value="Voltar ao Menu"
                 action="principal?faces-redirect=true" />
</h:form>

</div>
<!-- restante do código ... -->
```

Criando uma aplicação corporativa em Java – Parte 3

Autenticação

Antes de criarmos a página de login e o managed bean associado a ela para que o usuário possa se autenticar, precisamos criar outros managed beans que irão apoiar esse processo de autenticação.

O código do primeiro managed bean que apresentamos neste artigo se encontra na **Listagem 3, UsuarioLogadoBean**. Um managed bean pode ser definido como um objeto que é gerenciado pelo container. Se estivéssemos usando JSF sem CDI, poderíamos adotar a anotação `@ManagedBean(javax.faces.bean.ManagedBean)` para registrar a classe como um managed bean do JSF. Porém, a recomendação é que se use managed beans gerenciados pelo container CDI e não pelo JSF, e por isso é preferível usar a anotação `@Named` no lugar de `@ManagedBean`.

Listagem 3. Código do managed bean UsuarioLogadoBean.

```
package br.com.javamagazine.mb;

import java.io.Serializable;
import javax.enterprise.context.SessionScoped;
import javax.inject.Named;
import br.com.javamagazine.entidades.Usuario;

@Named
@SessionScoped
public class UsuarioLogadoBean implements Serializable{

    private static final long serialVersionUID = 1L;
    private Usuario usuario;

    public void logar(Usuario usuario){
        this.usuario = usuario;
    }

    public void deslogar(){
        this.usuario = null;
    }

    public boolean isLogado(){
        return usuario != null;
    }

    public Usuario getUsuario(){
        return usuario;
    }
}
```

Na própria especificação do JavaServer Faces 2.2 (JSR-344) é dito que as anotações do pacote `javax.faces.bean` serão depreciadas (*deprecated*) em futuras versões. Além disso, os managed beans do CDI são mais poderosos e flexíveis. Com eles podemos fazer uso de interceptors, escopo conversacional, eventos, injeção de dependências *type safe*, *decorators*, *stereotypes* e métodos produtores (que usam `@Produces`).

É importante ressaltar que a anotação `@Named` não torna a classe um bean. Ela apenas possibilita referenciar esse bean dentro de uma EL presente em alguma página JSF. O container CDI trata qualquer classe que satisfaça as condições a seguir como um managed bean:

- Que não seja uma classe interna não estática (*non-static inner class*);
- Que seja uma classe concreta ou “anotada” com `@Decorator`;
- Que não esteja “anotada” com qualquer anotação que a defina como um EJB ou declarada como um EJB no arquivo `ejb-jar.xml`;
- Que não implemente `javax.enterprise.inject.spi.Extension`;
- Que tenha um construtor apropriado, que pode ser um construtor default sem parâmetros ou um construtor marcado com `@Inject`.

A anotação `@Named` nos permite dar um nome a nosso bean, pois ela aceita uma `String` como argumento, e esse é o nome que deve ser usado dentro de qualquer EL que queira utilizar esse bean. Por exemplo, poderíamos colocar em cima da classe `UsuarioLogadoBean` a anotação `@Named("usuarioBean")`. Assim, na EL usariamos esse nome: `#{{usuarioBean.logado}}`. Também existe a opção de não passarmos nenhuma `String` para a anotação, escrevendo apenas `@Named`. Nesse caso, será usado o nome padrão, que é o nome não qualificado da classe com a primeira letra em minúsculo. Para a classe `UsuarioLogadoBean`, o nome padrão seria `usuarioLogadoBean`.

Na classe `UsuarioLogadoBean` foram colocadas as anotações `@Named` e `@SessionScoped`. Esta última indica que só existirá uma instância desse bean por sessão. Esse bean irá armazenar o objeto `usuario` que representa o usuário logado no sistema. É uma classe bem simples que contém apenas quatro métodos: o `logar()`, que seta o usuário logado na variável `usuario`; o `deslogar()`, que seta a variável `usuario` como nula; o `isLogado()`, que retorna um `boolean` para indicar se existe algum usuário logado; e o `getUsuario()`, que retorna o usuário logado.

O código do managed bean `FuncionarioEUsuarioVerificadorBean` é apresentado na **Listagem 4**. A função desse bean é basicamente verificar se já foi cadastrado algum funcionário da biblioteca e algum usuário e isso é feito por meio do único método existente na classe, chamado `existeFuncionarioEUsuarioCadastrado()`. Dentro desse método, a verificação é feita através dos DAOs que foram injetados por meio da anotação `@Inject`.

A **Listagem 5** exibe o código da página `login.xhtml`. Dentro dessa página informamos no atributo `template` da tag `ui:composition` qual o arquivo de template que será utilizado, ou seja, estamos declarando que a página `login.xhtml` é cliente do template `_template_simple.xhtml`. Também usamos a tag `ui:define` para definir o conteúdo que será substituído no template. Nesse caso, todo o conteúdo que estiver entre as tags `<ui:define name="corpo">` e `</ui:define>` na página `login.xhtml` entrarão no lugar da linha `<ui:insert name="corpo" />`, que foi colocada anteriormente, quando criamos o template.

Quando usamos JSF podemos utilizar EL para associar o valor de um componente (por exemplo, um `h:inputText`) a uma propriedade de um managed bean. Na página `login.xhtml` é feito esse tipo de associação através da EL `#{{loginBean.usuario.usuario}}`, estabelecendo a relação entre o valor do componente de texto onde será digitado o usuário (`h:inputText` com `id login`) e a propriedade

usuario do objeto **usuario**, que por sua vez é uma propriedade do managed bean **LoginBean**. Deste modo, quando mudarmos o valor do **h:inputText** com **id login**, a propriedade **usuario** do objeto **usuario** que está dentro de **LoginBean** também será alterada e vice versa. O mesmo conceito é aplicado para vincular o **h:inputSecret** de **id senha** com a propriedade **senha**, através da EL `#{loginBean.usuario.senha}`.

Listagem 4. Código do managed bean FuncionarioEUsuarioVerificadorBean.

```
package br.com.javamagazine.mb;

import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.inject.Named;

import br.com.javamagazine.dao.FuncionarioBibliotecaDao;
import br.com.javamagazine.dao.UsuarioDao;

@Named
@RequestScoped
public class FuncionarioEUsuarioVerificadorBean {

    @Inject
    private FuncionarioBibliotecaDao funcionarioBibliotecaDao;
    @Inject
    private UsuarioDao usuarioDao;

    public boolean existeFuncionarioEUsuarioCadastrado(){
        return (funcionarioBibliotecaDao.existeFuncionarioCadastrado() &&
        usuarioDao.existeUsuarioCadastrado());
    }
}
```

Listagem 5. Código da página login.xhtml.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets">
<ui:composition template="/WEB-INF/_template_simples.xhtml">
<ui:define name="corpo">
    <h2>Login no sistema</h2>
    <h:form>
        <h:panelGrid columns="2" styleClass="campos">
            <h:outputLabel value="Login:" for="login"/>
            <h:inputText id="login" value="#{loginBean.usuario.usuario}" />

            <h:outputLabel value="Senha:" for="senha"/>
            <h:inputSecret id="senha" value="#{loginBean.usuario.senha}" />
            <h:commandButton value="Efetuar Login"
                             action="#{loginBean.efetuarLogin}"/>
            <h:commandButton value="Criar usuário administrador"
                             action="funcionario_biblioteca?faces-redirect=true"
                             rendered="#{loginBean.opcaoCadastroUsuarioHabilitada}"/>
        </h:panelGrid>
    </h:form>
</ui:define>
</ui:composition>
</html>
```

Também podemos usar EL para referenciar métodos de managed beans dentro das tags JSF. Na página `login.xhtml` também foi utilizado esse conceito, pois ao clicarmos no primeiro **h:commandButton**, será invocado o método `efetuarLogin()` do managed bean **LoginBean**, conforme a configuração na EL `#{loginBean.efetuarLogin}`, colocada no atributo **action** do componente.

Na página de login ainda temos um outro **h:commandButton**, que só será mostrado caso a propriedade **opcaoCadastroUsuarioHabilitada** do **LoginBean** seja `true`. Esta propriedade será `true` somente enquanto não for cadastrado nenhum funcionário da biblioteca e um respectivo usuário para ele. Para ficar mais claro o motivo desse botão existir, vamos imaginar que acabamos de fazer o deploy desse sistema de biblioteca em produção e, portanto, o banco de dados está vazio. Ao acessarmos a tela de login teremos que informar o nome de usuário e a senha, mas o que fazer agora se ainda não foi cadastrado nenhum usuário e senha? Tudo bem, poderíamos fazer um `insert` direto no banco, mas note que a solução adotada é mais elegante.

Resolvemos esse problema exibindo o botão *Criar usuário administrador* que, conforme explicado anteriormente, só irá aparecer se não existir nenhum funcionário e nenhum usuário cadastrado, e quando clicado irá redirecionar para uma tela onde será possível cadastrar essas informações. Em nosso sistema, definimos que um usuário sempre deve estar vinculado a um funcionário, por isso ambos são cadastrados na mesma tela. O primeiro usuário que for cadastrado (através do botão *Criar usuário administrador*) será o administrador e terá permissão para cadastrar os demais usuários, que também poderão ser administradores ou não.

O código de **LoginBean**, managed bean que trabalhará em conjunto com a página JSF `login.xhtml`, é apresentado na **Listagem 6**. Neste código, o método `efetuarLogin()` primeiramente passa ao método `pesquisarUsuario()` um objeto **usuario** que armazena o usuário e a senha digitados na tela de login. Em seguida é checado se a invocação do método `pesquisarUsuario()` retornou algum objeto do tipo **Usuario** (o que significa que localizou um registro no banco correspondente ao usuário e senha informados) ou se retornou `null` (o que significa que não foi localizado nenhum registro no banco). Caso tenha sido retornado um objeto do tipo **Usuario**, ele passa a ser o usuário logado no sistema depois que o método `logar()` é invocado, e logo após é feito um redirect para a página `principal.xhtml`. Caso tenha sido retornado `null`, é feito um redirect para a página de login.

O método `efetuarLogout()` realiza logout do usuário que estava logado e também redireciona para a tela de login. Já o método `isOpcaoCadastroUsuarioHabilitada()` retorna `true` somente se ainda não tiverem sido cadastrados nenhum funcionário da biblioteca e nenhum usuário.

Criando a página principal do sistema

Quando o usuário se autenticar com sucesso no sistema, será redirecionado para a página principal, de nome `principal.xhtml`. Seu código pode ser visualizado na **Listagem 7**.

Criando uma aplicação corporativa em Java – Parte 3

Listagem 6. Código do managed bean LoginBean.

```
package br.com.javamagazine.mb;

import java.io.Serializable;
import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.inject.Named;
import br.com.javamagazine.dao.UsuarioDao;
import br.com.javamagazine.entidades.Usuario;

@Named
@RequestScoped
public class LoginBean implements Serializable {

    private static final long serialVersionUID = 1L;
    @Inject
    private UsuarioLogadoBean usuarioLogado;
    @Inject
    private FuncionarioEUsuarioVerificadorBean verificadorBean;
    private boolean opcaoCadastroUsuarioHabilitada;
    @Inject
    private UsuarioDao usuarioDao;
    private Usuario usuario = new Usuario();

    public String efetuarLogin(){

        Usuario usuarioEncontrado = usuarioDao.pesquisarUsuario(this.usuario);

        if(usuarioEncontrado != null){
            usuarioLogado.logar(usuarioEncontrado);
            return "principal?faces-redirect=true";
        }else{
            this.usuario = new Usuario();
            return "login?faces-redirect=true";
        }
    }

    public String efetuarLogout(){
        usuarioLogado.deslogar();
        this.usuario = new Usuario();
        return "login?faces-redirect=true";
    }

    public boolean isOpcaoCadastroUsuarioHabilitada() {
        opcaoCadastroUsuarioHabilitada = !verificadorBean.existeFuncionario
        EUusuarioCadastrado();
        return opcaoCadastroUsuarioHabilitada;
    }

    public Usuario getUsuario(){
        return this.usuario;
    }
}
```

Listagem 7. Código da página principal.xhtml.

```
</html>
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ui="http://java.sun.com/jsf/facelets">
<ui:composition template="/WEB-INF/_template_simples.xhtml">
    <ui:define name="corpo">
        <h:form>
            <h:commandLink value="Realizar Empréstimo de Livro"
                           action="emprestimo?faces-redirect=true" />
            <br/>
            <h:commandLink value="Realizar Devolução de Livro"
                           action="devolucao?faces-redirect=true" />
            <br/>
            <h:commandLink value="Cadastro de Funcionário e Usuário"
                           action="funcionario_biblioteca?faces-redirect=true"
                           rendered="#{usuarioLogadoBean.usuario.admin}" />
            <!-- restante dos links omitidos -->
        </h:form>
    </ui:define>
</ui:composition>
</html>
```

Ela é apenas uma página simples com links para as outras páginas. Estes links são gerados através da tag **h:commandLink**. Algo importante a se observar é que na última tag **h:commandLink** foi utilizado o atributo **rendered** para definir que o link para cadastrar funcionários e usuários só deve ser renderizado se o usuário logado for administrador (a EL usada para atingir esse resultado foi `#{usuarioLogadoBean.usuario.admin}`).

Qualquer usuário que não seja administrador não terá acesso para cadastrar outros usuários.

Criando as classes para navegação

Em nosso sistema, em alguns momentos precisaremos redirecionar o usuário para alguma página específica. Por exemplo, quando um usuário do sistema tenta acessar uma área restrita da aplicação sem se autenticar, deve ocorrer um redirecionamento para a página de login. Outro exemplo seria quando um usuário logado, mas que não é administrador, tenta acessar uma área do sistema reservada aos administradores. Nessa situação também deve ser feito um redirect, mas para a página principal.

A **Listagem 8** mostra o código da classe **Navegador**, que irá realizar o redirecionamento quando precisarmos, através do método **redirecionar()**, que recebe a URL de destino como argumento. Esse método usa duas classes da API do JSF (**NavigationHandler** e **FacesContext**) para poder direcionar o cliente para a URL que desejamos. Para que possam ser injetados um **NavigationHandler** e um **FacesContext** é necessário criarmos classes que produzam esses tipos de objetos.

A **Listagem 9** mostra a classe que produz objetos do tipo **NavigationHandler**. Essa classe tem um único método produtor, chamado **criarNavigationHandler()**, que a partir de um objeto **FacesContext** obtém um objeto **Application** e, através dele, retorna um objeto do tipo **NavigationHandler**.

Já a **Listagem 10** mostra a classe que produz objetos do tipo **FacesContext**. Para isso, essa classe apresenta um único método produtor, chamado **criarFacesContext()**, que apenas retorna o objeto **FacesContext** obtido pela invocação de **FacesContext.getCurrentInstance()**.

Listagem 8. Código da classe Navegador.

```
package br.com.javamagazine.util;

import javax.faces.application.NavigationHandler;
import javax.faces.context.FacesContext;
import javax.inject.Inject;

public class Navegador {

    @Inject
    private NavigationHandler navigationHandler;
    @Inject
    private FacesContext facesContext;

    public void redirecionar(String url) {
        navigationHandler.handleNavigation(facesContext, null, url +
            "?faces-redirect=true");
        facesContext.renderResponse();
    }

}
```

Listagem 9. Fábrica de NavigationHandler.

```
package br.com.javamagazine.fabricas;

import javax.enterprise.context.RequestScoped;
import javax.enterprise.inject.Produces;
import javax.faces.application.Application;
import javax.faces.application.NavigationHandler;
import javax.faces.context.FacesContext;
import javax.inject.Inject;

public class FabricaDeNavigationHandler {

    @Inject
    FacesContext facesContext;

    @Produces @RequestScoped
    public NavigationHandler criarNavigationHandler() {

        if (facesContext != null) {
            Application application = facesContext.getApplication();
            if (application != null) {
                return application.getNavigationHandler();
            }
        }
        return null;
    }
}
```

Listagem 10. Fábrica de FacesContext.

```
package br.com.javamagazine.fabricas;

import javax.enterprise.context.RequestScoped;
import javax.enterprise.inject.Produces;
import javax.faces.context.FacesContext;

public class FabricaDeFacesContext {

    public FabricaDeFacesContext() {
    }

    @Produces @RequestScoped
    public FacesContext criarFacesContext() {
        return FacesContext.getCurrentInstance();
    }
}
```

Criando o cadastro de funcionários e usuários

Para que possamos cadastrar os funcionários da biblioteca e os usuários, iremos criar a página *funcionário_biblioteca.xhtml*, que irá conter o formulário de cadastro. Por se tratar de uma página muito extensa, seu código será exibido e explicado em partes. As **Listagens 11, 12, 13 e 14** apresentam o código fragmentado.

Essa página é composta por duas grandes seções. A primeira consiste em um formulário com componentes para cadastrar um funcionário da biblioteca e suas informações pessoais, além de permitir que se cadastre também um usuário e uma senha, que serão vinculados ao funcionário. A segunda seção consiste em uma tabela que lista as informações de todos os funcionários salvos no banco (inclusive qual usuário foi vinculado a cada um deles), além de apresentar opções para exclusão e edição.

Esta tela será acessada para que seja feito o cadastramento de funcionários e usuários em duas situações distintas: a primeira ocorrerá quando não houver nenhum funcionário e nenhum usuário salvos no banco de dados, ou seja, o primeiro acesso ao sistema. Neste caso, quando aplicação for aberta, na página de login existirá um botão *Criar usuário administrador* que irá redirecionar para esta tela. A segunda situação ocorrerá quando for utilizado um usuário que é administrador para logar no sistema e a partir do menu principal for clicado no link que redireciona para essa página.

No primeiro cenário, após o cadastramento do primeiro funcionário e usuário, é desejável que haja um redirecionamento para a tela de login, para que o usuário recém-cadastrado seja usado para logar no sistema. Todo o mecanismo concebido para que o redirecionamento ocorra no momento certo, logo após o primeiro cadastro, gira em torno da tag *f:event*, presente na **Listagem 11**. No atributo *type* dessa tag, especificamos o tipo do evento, que nesse caso é *preRenderView*, um evento disparado antes da página JSF ser renderizada. Essa mesma tag também possui um atributo *listener*, através do qual informamos um método que deve ser invocado quando o evento ocorrer; nesse caso, o método *redirecionarParaLogin()* da classe *FuncionarioBibliotecaBean*.

Para que fique mais clara a utilização do componente *f:event*, realizaremos uma breve explicação das etapas a ele relacionadas. A primeira etapa ocorre quando o usuário do sistema o acessa pela primeira vez e clica no botão *Criar usuário administrador* na tela de login. Feito isso, é direcionado para a página *funcionário_biblioteca.xhtml*. Nessa página, irá cadastrar um funcionário e um usuário e clicar no botão *Salvar*. Este botão irá acionar o método *salvar()* do managed bean, que irá persistir todas as informações no banco de dados. Após a lógica dentro do bean ser executada, virá a fase **Render response** do ciclo de vida do processamento de uma requisição JSF (veremos mais sobre isso adiante) e a página *funcionário_biblioteca.xhtml* teria que ser renderizada. Mas antes que a renderização aconteça, o evento *preRenderView* é disparado e o método *redirecionarParaLogin()* do managed bean é acionado, realizando o redirecionamento para a página *login.xhtml*.

Criando uma aplicação corporativa em Java – Parte 3

Listagem 11. Código da página funcionario_biblioteca.xhtml.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ui="http://java.sun.com/jsf/facelets">
<ui:composition template="/WEB-INF/_template.xhtml">
<ui:define name="corpo">
<h:form>
  <h2>
    <h:outputText value="Novo Funcionário" rendered="#{empty funcionarioBibliotecaBean.funcionarioBiblioteca.id}" />
    <h:outputText value="Editar Funcionário" rendered="#{not empty funcionarioBibliotecaBean.funcionarioBiblioteca.id}" />
  </h2>

  <!-- conteúdo da Listagem 12 aqui - campos para preencher dados do funcionário -->
  <!-- conteúdo da Listagem 13 aqui - campos para preencher usuário e senha -->

</h:form>
<h:form>
  <!-- conteúdo da Listagem 14 aqui - listagem de funcionários e usuários -->

<f:metadata>
  <f:event type="preRenderView" listener="#{funcionarioBibliotecaBean.redirectarParaLogin}" />
</f:metadata>
</h:form>
</ui:define>
</ui:composition>
</html>
```

Listagem 12. Campos para preencher os dados do funcionário.

```
<fieldset>
  <legend>Dados do Funcionário</legend>
  <h:inputHidden value="#{funcionarioBibliotecaBean.funcionarioBiblioteca.id}" />

  <h:inputHidden value="#{funcionarioBibliotecaBean.telefoneFixo.id}" />
  <h:inputHidden value="#{funcionarioBibliotecaBean.telefoneCelular.id}" />

  <h:inputHidden value="#{funcionarioBibliotecaBean.funcionarioBiblioteca.endereco.id}" />
  <h:inputHidden value="#{funcionarioBibliotecaBean.funcionarioBiblioteca.usuario.id}" />
  <h:outputLabel value="Nome do Funcionário:" for="nome" />
  <h:inputText id="nome" value="#{funcionarioBibliotecaBean.funcionarioBiblioteca.nome}" />
  <h:outputLabel value="E-mail:" for="email" />
  <h:inputText id="email" value="#{funcionarioBibliotecaBean.funcionarioBiblioteca.email}" />
  <h:outputLabel value="CPF:" for="cpf" />
  <h:inputText id="cpf" value="#{funcionarioBibliotecaBean.funcionarioBiblioteca.cpf}" />
  <h:outputLabel value="Sexo:" for="sexo" />

  <h:selectOneRadio id="sexo" value="#{funcionarioBibliotecaBean.funcionarioBiblioteca.sexo}">
    <f:selectItem itemValue="M" itemLabel="Masculino" />
    <f:selectItem itemValue="F" itemLabel="Feminino" />
  </h:selectOneRadio>
```

Ainda analisando o código da página *funcionario_biblioteca.xhtml*, podemos perceber o uso extensivo de componentes JSF. Fica evidente, principalmente observando as **Listagens 12 e 13**, que vinculamos os valores desses componentes às propriedades de *FuncionarioBibliotecaBean*.

Através de um breve exemplo prático, vamos ressaltar os pontos que merecem atenção. Imagine que você precise desenvolver uma página que calcule o índice de massa corporal (IMC) de uma pessoa. Nesta página teríamos dois campos para se digitar (altura e peso) e um campo para mostrar o resultado do cálculo:

```
<h:inputText value="#{imcBean.altura}" />
<h:inputText value="#{imcBean.peso}" />
<h:outputText value="#{imcBean.valorImc}" />
```

Essa tela que contém os três componentes precisará trabalhar em conjunto com um managed bean que fará o cálculo do IMC, neste exemplo o *ImcBean*. Levando em conta este cenário, levantaremos duas questões, e suas respostas trarão à tona pontos importantes que devem ser assimilados pelos leitores:

1. Se criarmos três variáveis de instância privadas chamadas *altura*, *peso* e *valorImc* dentro de *ImcBean* e não criarmos seus respectivos getters e setters, nossa página não irá funcionar, uma vez que os getters e setters são necessários para que a relação entre componentes e o managed bean seja estabelecida com sucesso.
2. Dentro do managed bean, sou obrigado a criar uma variável de instância para cada componente? Também não. Quase sempre um componente tem uma variável de instância correspondente, embora isso não seja uma regra. Como já foi mencionado, os métodos

```
<h:outputLabel value="Telefone Fixo:" for="telefoneFixo" />
<h:inputText id="telefoneFixo" value="#{funcionarioBibliotecaBean.telefoneFixo.numero}" />
<h:outputLabel value="Telefone Celular:" for="telefoneCelular" />
<h:inputText id="telefoneCelular" value="#{funcionarioBibliotecaBean.telefoneCelular.numero}" />
<h:outputLabel value="Endereço:" for="endereco" />
<h:inputText id="endereco" value="#{funcionarioBibliotecaBean.funcionarioBiblioteca.endereco.endereco}" />
<h:outputLabel value="Complemento:" for="complemento" />
<h:inputText id="complemento" value="#{funcionarioBibliotecaBean.funcionarioBiblioteca.endereco.endereco.complemento}" />
<h:outputLabel value="Cep:" for="cep" />
<h:inputText id="cep" value="#{funcionarioBibliotecaBean.funcionarioBiblioteca.endereco.cep}" />
<h:outputLabel value="Bairro:" for="bairro" />
<h:inputText id="bairro" value="#{funcionarioBibliotecaBean.funcionarioBiblioteca.endereco.bairro}" />
<h:outputLabel value="Cidade:" for="cidade" />
<h:inputText id="cidade" value="#{funcionarioBibliotecaBean.funcionarioBiblioteca.endereco.cidade}" />
<h:outputLabel value="Estado:" for="estado" />
<h:inputText id="estado" value="#{funcionarioBibliotecaBean.funcionarioBiblioteca.endereco.estado}" />
</fieldset>
```

get e set têm papel fundamental, porém sua implementação fica a cargo do desenvolvedor. Neste exemplo, em nosso bean poderíamos abrir mão da variável privada **valorIMC** e implementar o método que retorna o valor do IMC da seguinte forma:

```
public double getValorIMC(){
    return peso / (altura * altura);
}
```

Passemos agora à análise do código da **Listagem 12**, onde podemos observar os componentes que possibilitam o preenchimento dos dados de um funcionário da biblioteca.

Listagem 13. Campos para preencher usuário e senha.

```
<fieldset>
<legend>Dados de Login</legend>
<h:outputLabel value="Usuário:" for="usuario" />
<h:inputText id="usuario" value="#{funcionarioBibliotecaBean.funcionarioBiblioteca.usuario.usuario}" />
<h:outputLabel value="Senha:" for="senha" />
<h:inputSecret id="senha" value="#{funcionarioBibliotecaBean.funcionarioBiblioteca.usuario.senha}" />
<h:outputLabel value="O usuário é administrador?" for="admin" />
<h:selectOneRadio id="admin" value="#{funcionarioBibliotecaBean.funcionarioBiblioteca.usuario.admin}" disabled="#{funcionarioBibliotecaBean.campoAdminDesabilitado}">
    <f:selectItem itemValue="true" itemLabel="Sim" />
    <f:selectItem itemValue="false" itemLabel="Não" />
</h:selectOneRadio>
<h:commandButton value="Salvar" action="#{funcionarioBibliotecaBean.salvar}" />
</fieldset>
```

Listagem 14. Listagem de funcionários da biblioteca e usuários.

```
<h2>Listagem de Funcionários da Biblioteca</h2>

<h: dataTable value="#{funcionarioBibliotecaBean.funcionariosBiblioteca}"
var="funcionarioBiblioteca" styleClass="dados" rowClasses="impar, par">
<h:column>
    <f:facet name="header">Nome do Funcionario</f:facet>
    #{funcionarioBiblioteca.nome}
</h:column>
<h:column>
    <f:facet name="header">Usuário</f:facet>
    #{funcionarioBiblioteca.usuario.usuario}, administrador:
    #{funcionarioBiblioteca.usuario.admin ? "Sim": "Não"}
</h:column>
<h:column>
    <f:facet name="header">E-mail</f:facet>
    #{funcionarioBiblioteca.email}
</h:column>
<h:column>
    <f:facet name="header">CPF</f:facet>
    #{funcionarioBiblioteca.cpf}
</h:column>

<h:column>
    <f:facet name="header">Sexo</f:facet>
    #{funcionarioBiblioteca.sexo}
</h:column>
<h:column>
    <f:facet name="header">Telefones</f:facet>
    #{funcionarioBiblioteca.telefones[0].numero},
```

Cada funcionário da biblioteca deve possuir um usuário e senha e para realizar o cadastramento dessas informações podemos utilizar a página *funcionario_biblioteca.xhtml*, que possui alguns componentes com essa finalidade (vide **Listagem 13**). Algo interessante a ser observado é que foi utilizado o atributo **disabled** de um dos componentes (**h:selectOneRadio** com **id admin**) para podermos habilitá-lo ou desabilitá-lo de acordo com a propriedade **campoAdminDesabilitado**, do managed bean **FuncionarioBibliotecaBean**. Esse componente foi utilizado para indicar se o usuário é ou não administrador, mas como temos uma regra de negócio que dita que o primeiro usuário a ser cadastrado deve ser administrador, esse componente é desabilitado no momento do primeiro cadastro.

Analisaremos agora a segunda seção presente na página *funcionario_biblioteca.xhtml*, a listagem de funcionários e usuários (vide **Listagem 14**), que é constituída basicamente por uma tabela, que é construída com o componente **h:dataTable**. Como podemos verificar, o valor desse componente é associado a uma propriedade do managed bean. Neste caso, o atributo **value** da tag **h:dataTable** recebe a EL **#{funcionarioBibliotecaBean.funcionariosBiblioteca}**, o que fará com que o componente invoque o método **getFuncionariosBiblioteca()** do bean para obter uma lista de objetos do tipo **FuncionarioBiblioteca**, que será usada para popular a tabela.

Como pudemos observar, os valores dos componentes presentes na página *funcionario_biblioteca.xhtml* foram associados com as propriedades do managed bean **FuncionarioBibliotecaBean**. Como o código desse bean é muito grande, ele foi dividido em várias partes, apresentadas da **Listagem 15 à 22**, e deste mesmo modo será explicado.

```
#{funcionarioBiblioteca.telefones[1].numero}
</h:column>
<h:column>
    <f:facet name="header">Endereço</f:facet>
    #{funcionarioBiblioteca.endereco.endereco}, #{funcionarioBiblioteca.endereco.complemento}, CEP: #{funcionarioBiblioteca.endereco.cep}
</h:column>
<h:column>
    <f:facet name="header">Bairro</f:facet>
    #{funcionarioBiblioteca.endereco.bairro}
</h:column>
<h:column>
    <f:facet name="header">Cidade</f:facet>
    #{funcionarioBiblioteca.endereco.cidade}
</h:column>
<h:column>
    <f:facet name="header">Estado</f:facet>
    #{funcionarioBiblioteca.endereco.estado}
</h:column>
<h:column>
    <f:facet name="header">Ações</f:facet>
    <h:commandLink action="#{funcionarioBibliotecaBean.remover(funcionarioBiblioteca)}" value="Remover" />
    &nbsp;
    <h:commandLink action="#{funcionarioBibliotecaBean.alterar(funcionarioBiblioteca)}" value="Alterar" />
</h:column>
</h: dataTable>
```

Criando uma aplicação corporativa em Java – Parte 3

Listagem 15. Código do managed bean FuncionarioBibliotecaBean.

```
package br.com.javamagazine.mb;

import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.inject.Named;
// restante dos imports omitidos

@Named
@RequestScoped
public class FuncionarioBibliotecaBean {

    @Inject
    private FuncionarioBibliotecaDao funcionarioBibliotecaDao;
    @Inject
    private FuncionarioUsuarioVerificadorBean verificadorBean;
    private FuncionarioBiblioteca funcionarioBiblioteca = new FuncionarioBiblioteca();
    private Telefone telefoneFixo = new Telefone();
    private Telefone telefoneCelular = new Telefone();
    private List<FuncionarioBiblioteca> funcionariosBiblioteca;
    @Inject
    private Navegador navegador;
    private boolean redirecionarParaLogin;
    private boolean campoAdminDesabilitado;

    FuncionarioBibliotecaBean(){
        funcionarioBiblioteca.setEndereco(new Endereco());
        Usuario usuario = new Usuario();
        usuario.setFuncionarioBiblioteca(funcionarioBiblioteca);
        funcionarioBiblioteca.setUsuario(usuario);
    }

    //Código do método isCampoAdminDesabilitado(), apresentado na
    //Listagem 16

    //Código do método salvar(), apresentado na Listagem 17
}
```

```
//Código do método redirecionarParaLogin(), apresentado na Listagem 18

//Código do método vincularTelefones(), apresentado na Listagem 19

//Código do método alterar(), apresentado na Listagem 20

//Código do método remover(), apresentado na Listagem 21

//Código do método getFuncionariosBiblioteca(), apresentado na Listagem 22

public FuncionarioBiblioteca getFuncionarioBiblioteca() {
    return funcionarioBiblioteca;
}

public void setFuncionarioBiblioteca(FuncionarioBiblioteca funcionarioBiblioteca) {
    this.funcionarioBiblioteca = funcionarioBiblioteca;
}

public Telefone getTelefoneFixo() {
    return telefoneFixo;
}

public void setTelefoneFixo(Telefone telefoneFixo) {
    this.telefoneFixo = telefoneFixo;
}

public Telefone getTelefoneCelular() {
    return telefoneCelular;
}

public void setTelefoneCelular(Telefone telefoneCelular) {
    this.telefoneCelular = telefoneCelular;
}
}
```

Na primeira listagem, temos a declaração das variáveis de instância, alguns métodos getters e setters (os demais métodos foram distribuídos em listagens separadas) e o construtor.

Nossa análise sobre o código desse bean será iniciada pelo construtor. Dentro dele são estabelecidas as relações entre as entidades, de forma que a entidade **FuncionarioBiblioteca** seja associada às outras duas: **Endereco** e **Usuario**. Fazemos essa “amarração” entre os objetos porque quando chegar a hora de os persistirmos no banco de dados, iremos passar para o método do DAO somente o objeto que representa o funcionário da biblioteca, porém os demais objetos vinculados a ele também devem ser persistidos. Mais do que persistir todos os dados, essas associações são essenciais para que sejam criados os relacionamentos dentro do banco. Outro ponto que vale a pena ser mencionado é que no momento em que o managed bean acaba de ser instanciado (pelo container), tanto o objeto que representa o funcionário da biblioteca quanto os outros dois vinculados a ele estão “vazios”, pois também acabaram de ser instanciados (através do operador **new**). Esses objetos serão “populados” com os valores dos componentes da página *funcionario_biblioteca.xhtml* e tais valores serão informados pelos usuários do sistema.

O parágrafo anterior encerra a explicação sobre o código apresentado na **Listagem 15**. No entanto, como podemos verificar, a implementação de alguns métodos foi omitida e apresentada em

listagens diferentes.

Respeitando a sequência indicada no código, iniciaremos as explicações pelo método **isCampoAdminDesabilitado()** – vide **Listagem 16**. Para que se entenda o motivo desse método existir, explicaremos duas situações em que o managed bean **FuncionarioBibliotecaBean** é usado. As duas situações são relacionadas ao cadastramento de funcionário e usuário:

1. Quando o usuário do sistema estiver na tela de login e não houver nenhum funcionário nem usuário cadastrados, irá aparecer para ele o botão *Criar usuário administrador*. Nessa situação, quando ele for cadastrar o primeiro funcionário e o primeiro usuário (ambos relacionados), esse usuário será administrador. Por isso o método **isCampoAdminDesabilitado()** deve retornar **true**. Consequentemente, o componente da tela que define se o usuário é admin ou não será desabilitado. Aproveitando o gancho, após o cadastramento do primeiro funcionário e usuário, o cliente do sistema será direcionado para a página de login novamente, para poder se autenticar (isso só acontecerá na primeira vez, ao realizar o segundo cadastro, por exemplo, não haverá essa mudança para a tela de login). Parte dessa lógica para fazer o redirecionamento para a página *login.xhtml* está no método **salvar()**, que veremos adiante;
2. Quando o usuário do sistema estiver logado e for administrador, poderá clicar no link para cadastramento de funcionários e usuá-

rios e, consequentemente, irá utilizar FuncionarioBibliotecaBean. Nessa situação o método **isCampoAdminDesabilitado()** deve retornar **false** e o usuário logado poderá escolher entre cadastrar um novo usuário administrador ou cadastrar um usuário comum (não administrador).

Listagem 16. Código do método **isCampoAdminDesabilitado()**.

```
public boolean isCampoAdminDesabilitado() {  
    campoAdminDesabilitado = !verificadorBean.existeFuncionarioEUusuario  
    Cadastrado();  
    return campoAdminDesabilitado;  
}
```

Agora passaremos à explicação do método **salvar()**, exibido na **Listagem 17**. Este método possui uma lógica para verificar se está sendo salvo o primeiro funcionário e seu respectivo usuário (a verificação é baseada no retorno do método **isCampoAdminDesabilitado()**, que foi explicado anteriormente). Caso seja o primeiro usuário, a variável **redirecionarParaLogin** é setada para **true** e por esse motivo o método **redirecionarParaLogin()**, que atua como listener, irá realizar o redirecionamento para *login.xhtml* (só executa o redirect se a variável for **true**).

Ainda no método **salvar()** há um condicional que verifica se o **id** do funcionário da biblioteca é nulo. Se o **id** for nulo, indica que está sendo inserido um funcionário novo e não editado um funcionário existente (pois caso o funcionário já existisse na base, ele teria um **id**). Caso seja um novo funcionário, passamos o objeto para o método **inserir()** do DAO. Caso seja a edição de um funcionário existente, passamos o objeto para o método **atualizar()** do DAO. No entanto, antes do funcionário ser salvo ou atualizado no banco de dados, vinculamos a ele os telefones, através do método **vincularTelefones()**.

Nas últimas linhas do método atribuímos objetos novos às variáveis membro, pois dessa forma, quando a execução desse método finalizar e chegar a hora da página *funcionario_biblioteca.xhtml* ser renderizada, seu formulário para cadastramento de funcionários e usuários estará em branco. Essa é uma forma de limpar o formulário na view, já que os valores de seus componentes estão ligados às propriedades de **FuncionarioBibliotecaBean**. Por último, atualizamos a lista de funcionários do managed bean com os dados que estão no banco, para que ela reflita o insert ou update que foi feito e para que isso também seja refletido no conteúdo do componente **h:dataTable**, da página *funcionario_biblioteca.xhtml*. Note que o método **salvar()** usa a anotação **@Transacional** que criamos, para que seja executado dentro de uma transação.

Em meio às explicações sobre o método **salvar()**, mencionamos o método **redirecionarParaLogin()**, e agora teremos a oportunidade de analisar seu código – vide **Listagem 18**. Este é um método bastante simples, que possui uma instrução **if** para verificar se o conteúdo da variável **redirecionarParaLogin** é **true**, e em caso afirmativo, invoca o método **redirecionar()** do objeto **navegador** passando como argumento a String “login”, o que ocasiona um redirecionamento para a página *login.xhtml*.

O método **vincularTelefones()** da **Listagem 19** basicamente adiciona os telefones digitados na página de cadastro à lista de telefones associada ao funcionário da biblioteca. Para um melhor entendimento do código, é válido citar que as variáveis membro **telefoneFixo** e **telefoneCelular** – mostradas nessa listagem – armazenam os telefones que são digitados pelo usuário do sistema.

Listagem 17. Código do método **salvar()**.

```
@Transacional  
public void salvar(){  
  
    if(isCampoAdminDesabilitado()){  
        redirecionarParaLogin = true;  
    }  
  
    if(funcionarioBiblioteca.getId() == null){  
        vincularTelefones();  
        funcionarioBibliotecaDao.inserir(funcionarioBiblioteca);  
    }else{  
        funcionarioBiblioteca.getTelefones().clear();  
        vincularTelefones();  
        funcionarioBibliotecaDao.atualizar(funcionarioBiblioteca);  
    }  
  
    telefoneFixo = new Telefone();  
    telefoneCelular = new Telefone();  
    funcionarioBiblioteca = new FuncionarioBiblioteca();  
    funcionariosBiblioteca = funcionarioBibliotecaDao.listarFuncionariosBiblioteca();  
}
```

Listagem 18. Código do método **redirecionarParaLogin()**.

```
public void redirecionarParaLogin(){  
    if(redirecionarParaLogin)  
        navegador.redirecionar("login");  
}
```

Listagem 19. Código do método **vincularTelefones()**.

```
public void vincularTelefones(){  
    telefoneFixo.setFixo(true);  
    funcionarioBiblioteca.getTelefones().add(telefoneFixo);  
    funcionarioBiblioteca.getTelefones().add(telefoneCelular);  
}
```

Já o método **alterar()** não irá modificar qualquer objeto no banco de dados, pois como já vimos, é o método **salvar()** que irá persistir as alterações na base. O papel deste método, apresentado na **Listagem 20**, é viabilizar que os dados do objeto a ser alterado sejam mostrados no formulário, permitindo sua edição. Para que esse método possa fazer seu trabalho, ele recebe o objeto funcionário que deve ser alterado como argumento e atribui esse objeto à variável de instância **funcionarioBiblioteca**. Além disso, atribui os telefones recuperados a partir do objeto que deve ser alterado às variáveis de instância **telefoneFixo** e **telefoneCelular**. Isso tudo é feito porque os componentes de formulário da página irão usar os getters dessas variáveis de instância para mostrar os dados na tela para edição.

O método **remover()**, cujo código se encontra na **Listagem 21**, recebe como argumento o funcionário da biblioteca a ser removido. Esse funcionário é repassado para o método **remover()** do DAO e

Criando uma aplicação corporativa em Java – Parte 3

logo em seguida a lista de funcionários do bean é atualizada com o conteúdo presente no banco de dados.

Listagem 20. Código do método alterar().

```
public void alterar(FuncionarioBiblioteca funcionarioBiblioteca){  
    this.funcionarioBiblioteca = funcionarioBiblioteca;  
    for(Telefone telefone : funcionarioBiblioteca.getTelefones()){  
        if(telefone.isFixo()){  
            this.telefoneFixo = telefone;  
        }else{  
            this.telefoneCelular = telefone;  
        }  
    }  
}
```

Listagem 21. Código do método remover().

```
@Transacional  
public void remover(FuncionarioBiblioteca funcionarioBiblioteca){  
    funcionarioBibliotecaDao.remover(funcionarioBiblioteca);  
    funcionariosBiblioteca = funcionarioBibliotecaDao.listarFuncionariosBiblioteca();  
}
```

Por fim, analisaremos o método `getFuncionariosBiblioteca()`, que pode ser visto na **Listagem 22**. O objetivo desse método é retornar uma lista de funcionários da biblioteca provinda da base de dados. Recuperada através do método `listarFuncionariosBiblioteca()` do DAO, essa lista é atribuída à variável de instância `funcionariosBiblioteca`, que é utilizada como retorno do método `getFuncionariosBiblioteca()`. Pelo fato do método `getFuncionariosBiblioteca()` ser chamado algumas vezes durante a renderização da página `funcionário_biblioteca.xhtml`, dentro dele foi colocado um teste condicional para evitar que sejam feitas múltiplas consultas ao banco de dados sem necessidade.

Listagem 22. Método getFuncionariosBiblioteca (complementa a Listagem 15).

```
@Transacional  
public List<FuncionarioBiblioteca> getFuncionariosBiblioteca(){  
    if(funcionariosBiblioteca == null){  
        funcionariosBiblioteca = funcionarioBibliotecaDao.listarFuncionariosBiblioteca();  
    }  
  
    return funcionariosBiblioteca;  
}
```

Ao longo dos artigos dessa série apresentamos bastante conteúdo, e nesse momento acreditamos que o leitor já esteja preparado para o desafio de implementar por si só o managed bean **Livro-Bean** e a página `livro.xhtml`, que compõem basicamente o CRUD de livros. O leitor terá que descobrir tudo que é necessário para que seu bean e sua página funcionem. Caso sinta dificuldades, na versão completa da aplicação, disponível no GitHub, você encontrará essa parte já implementada.

Autor



Bruno Rafael Sant'Ana

bruno.santana.ti@gmail.com

Graduado em Análise e Desenvolvimento de Sistemas pelo SENAC. Possui as certificações OCJP e OCWCD. Atualmente trabalha na Samsung com desenvolvimento Java e atua como CTO na startup Vester. Entusiasta de linguagens de programação e tecnologia.



Links:

Aplicação reduzida (igual a desenvolvida no artigo) no GitHub.

<https://github.com/brunosantanati/javamagazine-app-reduzida>

Aplicação completa no GitHub.

<https://github.com/brunosantanati/javamagazine-app-completa>

Endereço para download do JDK 8.

<http://www.oracle.com/technetwork/pt/java/javase/downloads/index.html>

Endereço para download do Eclipse Luna.

<https://www.eclipse.org/downloads/>

Endereço para download JBoss WildFly.

<http://wildfly.org/downloads/>

Endereço para download do instalador do MySQL.

<http://dev.mysql.com/downloads/installer/>

Endereço para download do driver do MySQL.

<http://dev.mysql.com/downloads/connector/j/>

Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!





DEVMEDIA

DÊ UM SALTO EM CONHECIMENTO!

Acesse o maior
portal para
desenvolvedores
da América
Latina!



20
mil
posts

430
mil
cadastrados

10
milhões de
page-views
por mês

Construindo e otimizando um ambiente de alta disponibilidade com Hazelcast – Parte 2

Como melhorar o desempenho do processamento distribuído em aplicações baseadas no Hazelcast

ESTE ARTIGO FAZ PARTE DE UM CURSO

Na primeira parte do artigo exploramos alguns conceitos como alta-disponibilidade, tolerância a falhas e *failover*, avaliando como podemos construir um ambiente capaz de prover estes requisitos de Qualidade de Serviço através de uma combinação do mecanismo de balanceamento de carga do Apache Web Server e servidores Tomcat, utilizando conectores específicos.

Mostramos que, no cenário de uma aplicação que não armazena estado na memória local dos servidores Tomcat (*stateless*), o *mod_jk* proporciona, além de Load Balancing, a feature de transparente *failover*, ou seja, a capacidade de continuar atendendo solicitações de usuários e/ou serviços caso um dos nós venha a falhar, sem a necessidade de intervenção por parte dos clientes.

Neste artigo vamos dar continuidade a este “exercício” de alta-disponibilidade, estendendo os conceitos para aplicações do tipo *stateful*, que podem armazenar temporariamente alguns dados na sessão HTTP de um cliente, ou seja, dados que podem ser perdidos no caso de falha de um servidor caso estejam simplesmente arquivados na memória local destes servidores.

Em seguida, faremos uma análise da solução de data grid que estamos utilizando, o Hazelcast, para que

Fique por dentro

O objetivo deste artigo é mostrar algumas técnicas que fazem a diferença em ambientes clusterizados, as quais permitem extrair um melhor desempenho do balanceamento de carga do Apache, assim como redução no uso da memória reservada para objetos armazenados em caches distribuídos, o que influencia de maneira positiva na escalabilidade e desempenho de soluções baseadas em Hazelcast, traduzindo-se em menores tempos de resposta para os usuários finais.

A longo do texto discutiremos conceitos como localidade de referência, afinidade de sessões e replicação de dados, que são essenciais para entender como minimizar o overhead intrínseco em soluções distribuídas.

possamos avaliar os custos associados à replicação de dados em um ambiente clusterizado e como podemos melhorar o overhead atrelado a operações sobre estruturas de dados distribuídas.

Avaliando alta disponibilidade em aplicações com estado

Neste tópico daremos continuidade à configuração da aplicação desenvolvida no artigo anterior, fazendo algumas alterações para que possamos testar o *failover* transparente no cenário em que a **HttpSession** é utilizada para armazenar dados de usuários “logados” no sistema.

Para testar a tolerância a falhas em aplicações *stateful*, a princípio bastaria colocar um ou mais atributos na sessão de um usuário e

observar o comportamento ao “desligar” um dos nós. Entretanto, uma das melhores maneiras de se testar alta-disponibilidade neste tipo de aplicação ocorre quando a mesma utiliza mecanismos de segurança, os quais “forçam” um usuário a se autenticar para obter permissão de acesso a determinados recursos, pois normalmente a autenticação está associada a features específicas de um servidor/framework que podem precisar de configurações adicionais para prover failover transparente, pois em geral a implementação padrão da sessão HTTP oferecida por *web containers* armazena os dados apenas na memória e/ou disco locais.

Essencialmente, o que “esperamos que aconteça” é que se um usuário já autenticado fizer uma requisição em um servidor em falha, o mesmo seja redirecionado para outro em funcionamento, tendo sua solicitação atendida normalmente, sem precisar se autenticar novamente.

Para testar este cenário, devemos escolher uma maneira pela qual os usuários possam se identificar. Em aplicações web Java, há quatro mecanismos de autenticação padrão:

- **Basic:** navegadores utilizam um painel modal “nativo” para que o usuário possa digitar e submeter suas credenciais. Geralmente os dados são codificados em Base64 e enviados como um *Http Header*;
- **Digest:** similar ao modo basic, porém os dados são criptografados antes de serem enviados;
- **Form:** neste mecanismo, a aplicação é responsável por prover uma página customizada para que usuários possam se identificar;
- **Client Cert:** é uma maneira sofisticada de realizar a autenticação de forma “automática” e normalmente é utilizada apenas para trocas de mensagens entre servidores ou por um número muito restrito de usuários, e.g., administradores do sistema, para obter acesso a determinados recursos da aplicação. Para um usuário poder utilizar o mecanismo de Client Cert em um navegador, um certificado deverá ser instalado no mesmo, o qual se encarregará de enviar as credenciais para o servidor quando houver necessidade de autenticação. O servidor deverá ser configurado para armazenar em sua *Keystore* (**BOX 1**) as respectivas credenciais. Esse método de autenticação não requer qualquer interação após ambos, clientes e servidores, estarem devidamente configurados e raramente é utilizado para aplicações que podem ser acessadas por muitos usuários, devido à complexidade do processo de emissão e gerenciamento de certificados.

BOX 1. Keystore

É um repositório de chaves assimétricas, também conhecidas como pares privado-público. Em Java, esses objetos podem ser criados programaticamente ou com o uso da ferramenta de linha de comando keytool. Em um processo de autenticação mútua, tanto servidor quanto cliente devem disponibilizar suas chaves públicas, também conhecidas como certificados, que serão “trocados” e validados no processo de autenticação. Para mais detalhes, consulte a seção [Links](#).

Destas quatro opções de autenticação, a mais comumente utilizada é a baseada em formulários (Form), na qual tipicamente apresentamos uma página com campos de login e senha para que

um usuário possa preenchê-los e solicitar sua autenticação. Este mecanismo, além de ser o mais utilizado por permitir a criação de uma interface customizada, é o mais adequado para se testar o failover, pois no caso de autenticação do tipo Basic ou Digest, os cabeçalhos de autenticação são submetidos a cada requisição e tipicamente são mantidos em cache após o preenchimento e envio dos dados. Já o método Client Cert é totalmente automatizado, ou seja, em caso de failover, não temos como saber de maneira simples se ocorreu novamente ou não uma autenticação de um usuário já autenticado.

Assim, para testar o failover, vamos configurar a autenticação via formulários em nossa aplicação e para nos auxiliar com esta tarefa utilizaremos o Spring Security.

Introduzindo autenticação com Spring Security

Antes de partirmos para a configuração do Spring Security, é interessante entender como funciona a autenticação baseada em formulários, que de certa forma está intimamente ligada à Sessão HTTP e também a uma *feature* do Apache, conhecida como Afinidade de Sessão, como veremos mais à frente. No mecanismo de autenticação do tipo Form, diferentemente de Basic, as credenciais são submetidas uma única vez e caso a tentativa de login seja bem sucedida o usuário receberá um Cookie chamado *jsessionId* que será utilizado para associar este usuário a uma *HttpSession* do lado do servidor, o que torna a aplicação *stateful*. Uma vez autenticado, a cada requisição este Cookie será enviado para que as credenciais do usuário possam ser recuperadas. Na API do Spring Security, a associação *jsessionId*↔credenciais é realizada no início do ciclo de vida de uma requisição para que, posteriormente, regras mais sofisticadas de autorização possam ser aplicadas baseadas em perfis do usuário, e.g., habilitar certos menus apenas para administradores, restringir a invocação de métodos da camada de serviço, etc.

O Spring Security permite a configuração de diversas regras de acesso, de acordo com um conjunto de papéis (roles), também conhecidos como perfis. Um perfil nada mais é do que um rótulo, que pode estar associado a diversos usuários. Para simplificar a configuração, vamos restringir o acesso a todas as páginas dentro da aplicação a um usuário que tenha o perfil de administrador, que será chamado de *ROLE_ADMIN*. As únicas exceções a esta regra, ou seja, recursos que terão acesso liberado para “qualquer um”, serão as páginas de login e a página que acusa falha no login. A configuração destas regras no Spring Security requer o uso do namespace *security* e apenas algumas linhas de XML, como mostrado na **Listagem 1**.

Esse trecho de configuração do Spring essencialmente diz que um usuário não autenticado, ao tentar acessar qualquer caminho da aplicação, e.g., */management/memory.html*, deverá ser redirecionado à página de *login.html*, apresentada na **Listagem 2**. Por simplicidade, na **Listagem 1** estamos utilizando um cadastro de usuários “hard-coded” definido no trecho `<sec:user>`, porém é possível adaptar esta configuração para usar um cadastro “real” de usuários.

Listagem 1. Configuração do Spring Security.

```
<beans xmlns="..." xmlns:sec="http://www.springframework.org/schema/security">

    <!--Acesso liberado em login.html e login-failed.htm-->
    <sec:http pattern="/login*" security="none"/>

    <sec:http
        auto-config="false"
        use-expressions="true">
        <sec:intercept-url
            pattern="/**"
            access="hasRole('ROLE_ADMIN')"/>
        <sec:form-login
            default-target-url="/login.html"
            login-page="/login.html"
            authentication-failure-url="/login-failed.html"
            username-parameter="username"
            password-parameter="password"/>
        <sec:logout logout-url="/login.html"/>
    </sec:http>

    <sec:authentication-manager>
        <sec:authentication-provider>
            <sec:user-service>
                <!--Usuário hard-coded-->
                <sec:user
                    name="admin"
                    password="admin"
                    authorities="ROLE_ADMIN"/>
            </sec:user-service>
        </sec:authentication-provider>
    </sec:authentication-manager>
</beans>
```

Listagem 2. Código da página de Login.

```
<html>
    <head>
        <meta http-equiv="Cache-Control" content="no-store,no-cache,must-revalidate">
        <meta http-equiv="Pragma" content="no-cache">
        <meta http-equiv="Expires" content="-1">
    </head>
    <body>
        <form method="POST" action="_spring_security_check">
            <table>
                <tr>
                    <td colspan="2">Login Page</td>
                </tr>
                <tr>
                    <td>user:</td>
                    <td><input type="text" name="username" /></td>
                </tr>
                <tr>
                    <td>pass:</td>
                    <td><input type="password" name="password" /></td>
                </tr>
                <tr>
                    <td colspan="2"><input type="submit" value="Go" /></td>
                </tr>
            </table>
        </form>
    </body>
</html>
```

Por fim, ainda temos que informar à aplicação para utilizar o Spring Security. Isto é feito com a adição de um filtro no arquivo `web.xml`, como mostrado na **Listagem 3**.

Listagem 3. Ativação do Spring Security no web.xml.

```
<web-app xmlns="...">

    <!-- Configuração Spring Security -->
    <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>

    <!-- Carrega a configuração de segurança -->
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/security.xml
        </param-value>
    </context-param>

    <!-- Configurações do RESTEasy -->
    ...
</web-app>
```

Testando o failover transparente e a afinidade de sessão

Com esta configuração ativa podemos testar uma *feature* oferecida pelo Apache: a afinidade de sessões. Por padrão, esta *feature* não é ativada ao inicializar o servidor, porém, se observarmos a configuração do balanceamento de carga na **Listagem 5** da parte 1, veremos que ativamos a afinidade de sessões ao definir a propriedade `sticky_session` como `true`.

Ao optar por trabalhar com *sticky sessions* estamos informando ao Apache para manter um usuário que tenha uma sessão “preso” no mesmo servidor Tomcat. E qual a vantagem disso? A vantagem é que a aplicação vai se beneficiar de um princípio conhecido como “Localidade de Referência”, que vamos explorar no tópico seguinte.

Se acessarmos a página de monitoramento de memória, após nos autenticarmos na página de login e verificarmos as estatísticas do Apache (`/jkstatus`), veremos que apenas o contador de um dos nós está aumentando, ou seja, após se autenticar, todas as requisições de um usuário são direcionadas para um único nó, enquanto que o outro não recebe nenhuma requisição.

Por outro lado, se “matarmos” o nó em que o usuário foi autenticado, veremos que, diferentemente do teste que fizemos quando a aplicação era *stateless*, não ocorreu o failover de forma transparente.

E por que não? Ao detectar a falha de um dos nós o Apache de fato redirecionou a requisição para o outro que estava disponível. Entretanto, este nó não tinha qualquer conhecimento da **HttpSession** que havia sido criada anteriormente no nó em falha e, portanto, irá solicitar ao usuário que se autentique novamente!

Assim, para implementar o *failover* transparente em uma aplicação *stateful*, somos forçados a replicar os dados da **HttpSession** de forma que qualquer nó do cluster possa reconhecer um usuário já autenticado e que “migrou” de um nó para outro devido à uma falha.

Configurando o Hazelcast

Configurar o Hazelcast para replicar a **HttpSession** é uma tarefa muito simples, porém o que efetivamente temos com que nos preocupar é a implicação disto.

Como verificamos anteriormente, o Apache é capaz de prover afinidade de sessão e afirmamos que isto em geral é benéfico devido ao princípio de “Localidade de Referência”. Este conceito prevê que uma parte dos dados em um ambiente distribuído poderá ser acessada diretamente da memória local, ou seja, parte da comunicação remota poderá ser eliminada se soubermos direcionar a requisição para o servidor em que se encontram esses dados.

Entretanto, o alicerce de replicação de sessão do Hazelcast é baseado na estrutura de dados *IMap*, que tem a característica de ser escalável por distribuir os dados que armazena de maneira praticamente homogênea entre todos os nós do cluster. Assim, não é possível controlar o destino dos atributos de uma **HttpSession** e corremos o “risco” de fazer com que o framework recorra a invocações remotas para obter determinado atributo de uma sessão, o que invalida o benefício da “Localidade de Referência”.

Para entender melhor esta situação, consideremos o cenário em que um usuário possui quatro atributos em sua **HttpSession**, denotados $\{[K_1, V_1], [K_2, V_2], [K_3, V_3], [K_4, V_4]\}$, sendo que os dois primeiros pares estão armazenados em uma instância e os dois últimos em outra. Se o usuário fizer uma solicitação que requer os valores associa-

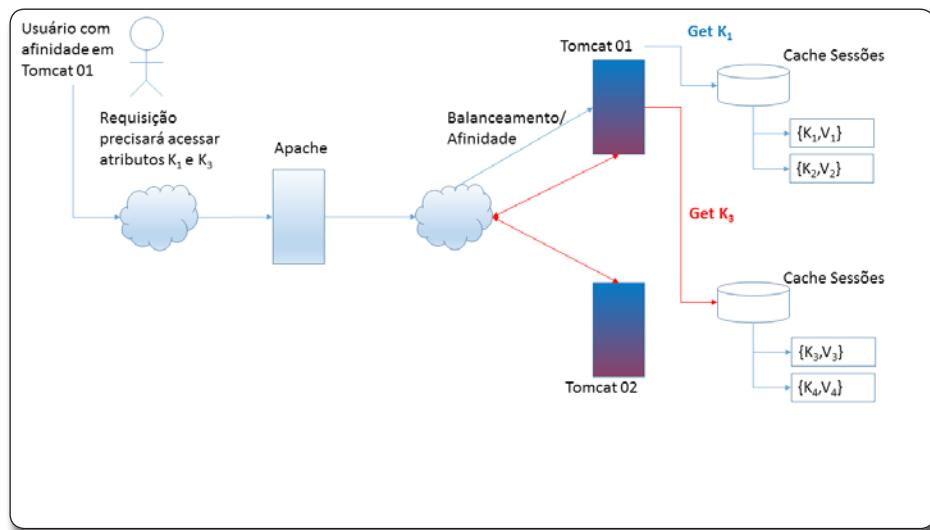


Figura 1. Perda de Localidade de Referência

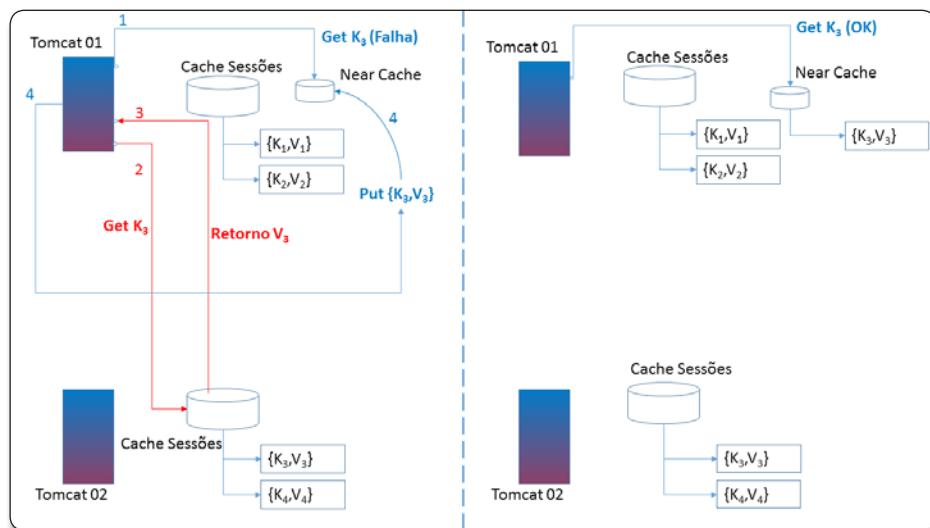


Figura 2. Recuperando a Localidade de Referência com Near Cache

dos à K_1 e K_3 , a operação para obter o valor mapeado por K_3 será realizada de forma remota, como mostrado na **Figura 1**.

Para contornar esse tipo de situação, o Hazelcast introduziu uma *feature* conhecida como *Near Cache*. Quando habilitamos um *Near Cache* em um *IMap*, todas as operações de get são feitas primeiramente no *Near Cache* e em caso de falha a operação “degrada” para uma chamada remota. Ao completar esta invocação, o resultado é armazenado no *Near Cache* e enquanto nenhuma operação de put (ou remove) associada à chave armazenada nele ocorrer, todas as invocações posteriores serão locais. A **Figura 2** ilustra o fluxo da operação get em um mapa distribuído com o *Near*

Cache habilitado. Na primeira solicitação à K_3 , o Hazelcast fará uma invocação remota para obter o valor correspondente. Ao recuperar o valor do outro nó, o par $\{K_3, V_3\}$ será armazenado no *Near Cache*, de forma que possa ser recuperado localmente em solicitações futuras.

Para habilitar o *Near Cache*, basta declarar o mesmo na configuração do Hazelcast como um elemento aninhado do elemento `<map>`, como mostrado na **Listagem 4**.

Finalmente, devemos configurar a aplicação para utilizar o Hazelcast como mecanismo de armazenamento de sessões. Isto é feito com a declaração de um filtro no `web.xml`, conforme o trecho apresentado na **Listagem 5**.

Com esta configuração, caso haja falha em um dos nós do Tomcat, um usuário autenticado continuará a utilizar normalmente a aplicação. E por que isto acontece? Ao habilitar o filtro do Hazelcast, os dados de sessão serão armazenados em IMaps que, por padrão, armazenam *backups* dos dados de outro nó. Quando um nó falha, o Hazelcast inicializa um processo de reparticionamento, o qual consiste basicamente em redistribuir as informações deste nó, armazenadas na forma de backup, para as instâncias remanescentes.

Listagem 4. Configurando IMap com um near-cache.

```
<hazelcast xmlns="...">

<!-- Configurações Básicas, Rede, etc -->

<!-- Configuração do cache da sessão HTTP -->

<map name="http-sessions">
  <max-size>5000</max-size>
  <near-cache>
    <max-size>500</max-size>
  </near-cache>
</map>
</hazelcast>
```

Listagem 5. Configurando a replicação com um near-cache.

```
<web-app xmlns="...">

<filter>
  <filter-name>hazelcast-filter</filter-name>
  <filter-class>com.hazelcast.web.WebFilter</filter-class>
  <init-param>
    <param-name>map-name</param-name>
    <param-value>http-sessions</param-value>
  </init-param>
  <init-param>
    <param-name>config-location</param-name>
    <param-value>/WEB-INF/classes/hazelcast-config.xml</param-value>
  </init-param>
  <init-param>
    <param-name>instance-name</param-name>
    <param-value>jm-cluster</param-value>
  </init-param>
  <init-param>
    <param-name>sticky-session</param-name>
    <param-value>true</param-value>
  </init-param>
  <init-param>
    <param-name>deferred-write</param-name>
    <param-value>true</param-value>
  </init-param>
</filter>

<!--Deve ser o primeiro filtro da aplicação!-->
<filter-mapping>
  <filter-name>hazelcast-filter</filter-name>
  <url-pattern>/*</url-pattern>
  <dispatcher>FORWARD</dispatcher>
  <dispatcher>INCLUDE</dispatcher>
  <dispatcher>REQUEST</dispatcher>
</filter-mapping>

<!-- Configurações de Spring e RESTEasy -->

</web-app>
```

A configuração do Hazelcast ainda define alguns atributos que chamam a atenção, a saber:

- **map-name:** permite que escolhamos qual IMap será utilizado para armazenar os dados. Se não informado, o Hazelcast irá criar um IMap padrão. Como estamos usando Near Cache, devemos especificar qual configuração será aplicada;
- **sticky-sessions:** Deve ser habilitada quando estivermos utilizando um Load Balancer com afinidade de sessões. Com esta configuração, o Hazelcast elimina algumas chamadas remotas de controle interno do framework. Contudo, ela não elimina a necessidade de se habilitar o Near Cache para o mapa que armazenará as sessões HTTP;
- **deferred-write:** Informa se as modificações da sessão devem ser feitas imediatamente (desabilitado por padrão, **deferred-write=false**) ou apenas no final do ciclo de vida de uma requisição. Embora não esteja oficialmente documentado no Hazelcast, alguns frameworks podem não funcionar perfeitamente sem **deferred-write** habilitado.

Há ainda um último pequeno evento que merece atenção, o qual pode ocorrer quando uma sessão migra de um nó para outro, e para avaliar este cenário precisamos entender com um pouco mais de detalhes como “funciona” a afinidade de sessões.

Quando o Tomcat é configurado com uma *jvmRoute*, o id da sessão é concatenado com o valor deste parâmetro, e.g., se uma sessão com id XYZW for criada na instância que declara a *jvmRoute* como sendo node02, o valor do *jsessionId* será XYZW.node02. O Apache usa este sufixo para estabelecer a afinidade de sessão, porém, uma vez que node02 falhar, este sufixo não será modificado automaticamente e o Apache passará a enviar requisições para todos os nós do cluster, de maneira *round-robin*, pois não é capaz de se conectar ao node02.

Nota

No contexto de平衡amento de carga, round-robin é um termo utilizado para indicar o incremento sequencial e a repetição, também referidos como ‘circularidade’. Se temos quatro nós e o Load Balancer é configurado com round-robin, as requisições serão servidas na ordem 1->2->3->4->1->2->.... O Apache permite utilizar diferentes estratégias de balanceamento, e.g, baseadas em tráfego, onde o LoadBalancer tenta fazer com que todos os servidores de aplicação sirvam aproximadamente o mesmo número de bytes, ou ocupação, na qual o Apache tenta estabilizar o número de requisições concorrentes em cada servidor de aplicação. Quando utilizamos afinidade de sessões, o mecanismo de balanceamento de carga tende a se tornar menos relevante, pois os clientes acabam se fixando em um único nó.

Em um cenário com apenas dois nós isso não é importante, pois caso um falhe, o remanescente passará a responder por todas as requisições e conterá todos os dados em memória. Em outras palavras, com apenas dois nós temos somente alta-disponibilidade, não temos escalabilidade no contexto de armazenamento de dados.

Por sua vez, em um cenário com mais de dois nós, todos eventualmente receberão uma requisição oriunda de um usuário que migrou do nó em falha e isso implica que os Near Caches desses

nós serão populados com os dados da sessão dos usuários que estavam “conectados” ao nó em falha, ou seja, as HttpSession terão réplicas em todos os Near Caches, comprometendo a escalabilidade e o desempenho da aplicação.

Para entender melhor este “efeito”, considere que temos um cluster com três nós e um usuário tem três atributos em sua sessão, denotados pelos pares $\{[K_1, V_1], [K_2, V_2], [K_3, V_3]\}$. Imaginemos que inicialmente cada par está armazenado em um dos nós e que após falha do nó 1, o nó 2 ficará apenas com o par $\{[K_1, V_1]\}$ e o nó 3 ficará com $\{[K_2, V_2], [K_3, V_3]\}$.

No cenário em que um usuário esteja com afinidade no nó 1, esses pares ficarão armazenados no Near Cache do nó 1 apenas, pois o usuário não fará requisições aos nós 2 e 3 enquanto o nó 1 estiver respondendo. Assim que o nó 1 falhar, a próxima requisição do usuário será redirecionada para, e.g., o nó 2. Embora a requisição seja atendida “normalmente” pelo nó 2, a afinidade não é restabelecida, de forma que a próxima requisição poderá ser atendida pelo nó 3, e assim sucessivamente, ou seja, eventualmente os Near Caches de todos os nós irão conter todos os dados da sessão HTTP do usuário. Esquematicamente, esse efeito pode ser representado como na Figura 3.

Para contornar este problema, podemos criar um filtro de requisições que modifica o valor do Cookie *jsessionId* para atribuir uma nova *jvmRoute*, como mostra o código da Listagem 6.

O código avalia se o cookie *jsessionId* tem como sufixo a mesma *jvmRoute* do nó que está atendendo a requisição. Se não tiver, ele será invalidado e um novo cookie com a *jvmRoute* “correta” será criado. Desta forma, requisições posteriores passarão a ser atendidas apenas pelo nó que “restabeleceu” a afinidade da sessão e apenas o Near Cache deste será populado com os

dados da sessão do usuário, como mostra a Figura 4.

Nota:

Por padrão, o Hazelcast não armazena dados no Near Cache que já estão armazenados localmente no IMap, ou seja, na Figura 3 o Near Cache do nó 2 conteria apenas os pares $\{[K_2, V_2], [K_3, V_3]\}$ e o Near Cache do nó 3 apenas $\{[K_1, V_1]\}$. Entretanto, é possível e muitas vezes útil, habilitar uma opção chamada cache-local-entries, a qual provocará o armazenamento de cópias locais. Normalmente esta opção é utilizada quando o formato dos objetos guardados no IMap é diferente dos guardados no Near Cache.

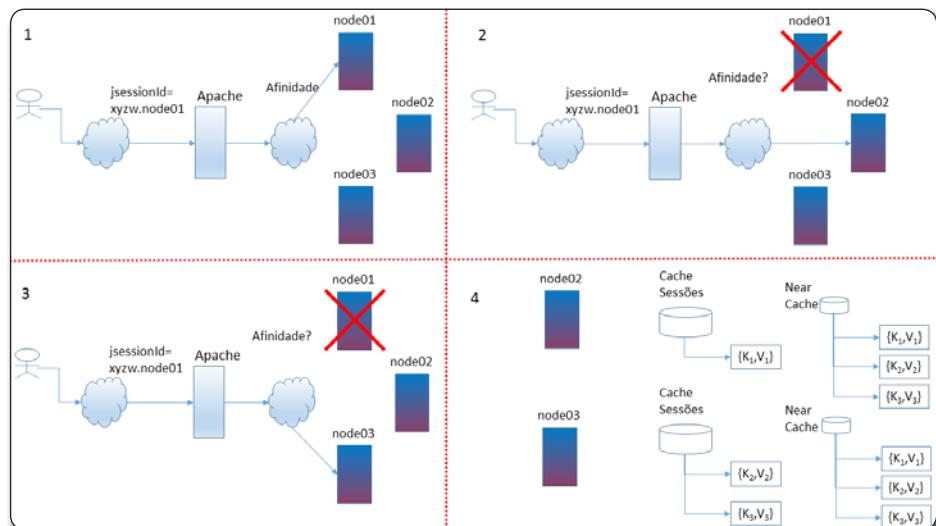


Figura 3. Perda de afinidade de sessão

Listagem 6. Reestabelecendo a afinidade de sessão

```
public class RouteAwareFilter implements Filter {
    // A jvmRoute foi definida como uma SystemProperty no arquivo catalina.properties
    static final String JVM_ROUTE = System.getProperty("tomcat.route");

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
        Cookie[] cookies;

        if (JVM_ROUTE != null &&
            request instanceof HttpServletRequest &&
            (cookies = ((HttpServletRequest) request).getCookies()) != null) {
            for (Cookie cookie : cookies) {
                if ("JSESSIONID".equals(cookie.getName())) {
                    String value;

                    if (cookie.getValue() != null && (value = cookie.getValue()).contains(JVM_ROUTE)) {
                        cookie.setValue(null);
                        ((HttpServletResponse) response).addCookie(cookie);

                        final int ix = value.lastIndexOf(":");
                        if (ix > 0) {
                            value = value.substring(0, ix + 1) + JVM_ROUTE;
                        } else {
                            value = value + ":" + JVM_ROUTE;
                        }
                        cookie = new Cookie("JSESSIONID", value);
                        cookie.setPath(request.getServletContext().getContextPath() + "/");
                        ((HttpServletResponse) response).addCookie(cookie);
                    }
                }
            }
        }
        chain.doFilter(request, response);
    }
}
```

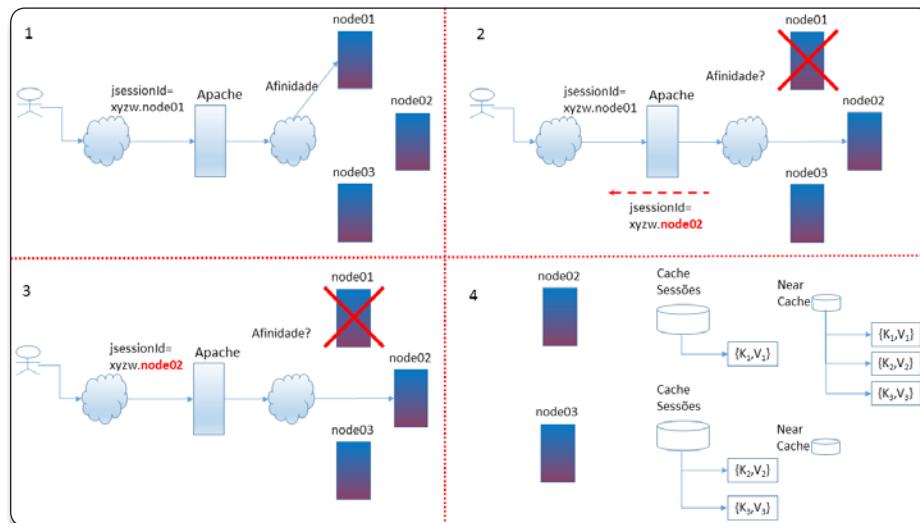


Figura 4. Restabelecendo de afinidade da sessão

Otimizações

Neste tópico faremos algumas considerações que têm por objetivo refinar certas configurações que podem auxiliar na melhoria do desempenho de aplicações que utilizam a combinação dos componentes Apache, Tomcat e Hazelcast como alicerce para oferecer escalabilidade e alta-disponibilidade.

Tomcat – Utilizando a biblioteca APR Nativa

Se examinarmos o log durante a inicialização de um servidor Tomcat, veremos a seguinte mensagem de warning: “*The APR based Apache Tomcat Native library which allows optimal performance in production environments was not found on the java.library.path: ...*”. Essa mensagem quer dizer que o Tomcat utilizará um conector padrão baseado em IO ou NIO, que podem ser muito bons em determinadas situações, porém o conector baseado no Apache Portable Runtime (APR) em geral é a melhor opção quando o Tomcat é utilizado em conjunto com o Apache, principalmente quando há necessidade de comunicação SSL (veja a seção **Links**) entre as duas pontas.

Infelizmente o Tomcat não disponibiliza as bibliotecas nativas já compiladas, porém criar os respectivos binários é uma tarefa relativamente simples de se fazer. Para isso:

- Crie um diretório chamado *apr*, dentro do diretório *tomcat-cluster* criado anterior-

mente (veja a seção “Tomcat Web Server” na Parte 1);

- Baixe o arquivo *apr-1.5.1.tar.gz* do site e descompacte-o dentro do diretório *apr*;
- Entre no diretório *apr/apr-1.5.1* e execute:
 - ./configure*
 - make*
 - make install*
- Após o passo três será criado o arquivo */usr/local/apr/bin/apr-1-config*, que será utilizado para compilar a biblioteca nativa do Tomcat

Os próximos passos consistem em compilar e ajustar o Tomcat para usar as bibliotecas nativas, conforme enumerado a seguir:

- Dentro de uma instância do Tomcat, e.g. *node01*, o mesmo disponibiliza os artefatos necessários para a compilação da biblioteca nativa em um arquivo chamado *tomcat-native<ver>.gz* (versão atual= 1.1.32). Copie o mesmo para o diretório *apr* e descompacte-o logo em seguida;
- Acesse o diretório *apr/tomcat-native-1.1.32-src/jni/native*
 - Execute o comando: *./configure --with-apr=/usr/local/apr/bin/apr-1-config --with-java-home=\$JAVA_HOME --with-ssl=no --prefix=<caminho_para_tomcat-cluster>/apr*
 - Execute o comando *make && make install*
- Ao executar o passo dois, os binários serão compilados e copiados para o diretório

definido pelo parâmetro *prefix*. Para que o Tomcat encontre estes binários, inicialize-o com a *system property* *java.library.path* apontando para este caminho;

- Ao inicializar o Tomcat e acessar o log, podemos constatar que a mensagem de warning anterior foi substituída por “*Laden APR based Apache Tomcat Native library 1.1.32 using APR version 1.5.1.*”.

Apache – Configurando o mod_worker

Algumas distribuições do Linux, e.g., Red Hat, por padrão não habilitam o chamado *mod_worker* do Apache durante a instalação do mesmo, e em seu lugar utilizam o módulo *prefork*. Estes módulos são responsáveis por controlar a alocação de conexões disponíveis no Apache e para aplicações que utilizam o *mod_jk* ou o *mod_proxy* para realizar o平衡amento de carga, há uma enorme diferença em termos de estabilidade, desempenho e escalabilidade, quando se utiliza o módulo *worker* em vez do *prefork*. De fato, o módulo *prefork* degrada rapidamente à medida que o número de conexões aumenta e hoje não deve ser sequer considerada a hipótese de habilitá-lo!

Para checar e habilitar o *mod_worker* após a instalação do Apache, nas distribuições Red Hat e CentOS, basta executar os seguintes passos:

- Checar o arquivo */etc/sysconfig/httpd*;
- O *mod_worker* estará desabilitado se a linha que referencia a variável *HTTPD* estiver comentada (precedida com o símbolo #):
 - *#HTTPD=/usr/sbin/httpd.worker*
- Para habilitar o *mod_worker* basta descomentar a linha e reiniciar o Apache.

Com o *mod_worker* habilitado, podemos parametrizar sua configuração de acordo com o hardware do servidor no qual está instalado o Apache. A sintaxe para alterar os valores padrão de inicialização do *mod_worker* está descrita na **Listagem 7**, a qual mostra um trecho de “xml” que deverá ser colocado no arquivo *httpd.conf* de modo a efetivar a parametrização.

É importante entender o significado destes parâmetros para que possamos ter um melhor controle sobre a quantidade de recursos que queremos disponibilizar no

servidor HTTP. Sendo assim, a seguir enumeramos as principais características de cada um:

- **ServerLimit**: Determina quantos sub-processos, no máximo, o Apache irá gerenciar. No *mod_worker*, cada sub-processo será responsável por gerenciar um certo número de Threads para atender as solicitações. O valor deste parâmetro não poderá ser maior que o número de processadores disponíveis no servidor, caso contrário o Apache não será inicializado;
- **StartServers**: Especifica quantos sub-processos serão alocados na inicialização do Apache;
- **ThreadsPerChild**: Determina o número máximo de Threads que poderão ser criadas por um sub-processo do Apache. Este valor não pode ser superior a 64;
- **MaxClients**: Determina o número máximo de clientes que poderão ser atendidos simultaneamente. Note que isso não significa que todos serão realmente servidos ao mesmo tempo, principalmente quando o Apache é usado como um LoadBalancer, e depende do tempo de resposta de outros servidores.

- O cálculo de *MaxClients* deve obedecer à seguinte restrição: *MaxClients* <= (*ServerLimit* x *ThreadsPerChild*). Por exemplo, se um servidor tem 10 processadores, como o número máximo que pode ser atribuído à *ThreadsPerChild* é 64, o valor de *MaxClients* deverá ser configurado com 640 ou menos;

- Para mais detalhes da relação do valor de *MaxClients* com os pools de conexão de servidores de aplicação, consulte a seção **Links**.

Hazelcast – Arquitetura física para clusters com muitos nós

Idealmente, a comunicação entre os clusters do Hazelcast deve ser feita em uma rede privada, ou seja, uma rede exclusiva para tráfego dos dados entre os nós do Hazelcast (vide Figura 5). Existem algumas maneiras de implementar topologias adequadas para comunicação intra-clusters. As mais comuns consistem na segmentação física das redes, também conhecidas como redes privadas, ou na criação de VLANs, no caso dos nós estarem fisicamente separados, e.g., em prédios distintos.

A vantagem de realizar esta segmentação é que todo fluxo de mensagens entre os servidores ficará isolado, ou seja, a rede de comunicação não receberá broadcasts e informações oriundas de outras redes, o que tipicamente ocorre em servidores que utilizam diversos serviços espalhados na rede como DNS, NTP, etc.

A desvantagem de isolar a comunicação é que podemos perder a capacidade de utilizar o *Hazelcast Client*, que permite criar aplicações que consultam e alteram diretamente as estruturas do Hazelcast, utilizando o protocolo “nativo” do framework, porém sem fazer parte do esquema de particionamento. Um exemplo interessante do uso do *Hazelcast Client* seria a construção de apli-

cações do tipo *chat*, em que muitos usuários se inscrevem num determinado tópico para enviar/receber mensagens.

Listagem 7. Parametrizando o mod_worker.

```
</IfModule worker.c>
ServerLimit 16
StartServers 4
ThreadsPerChild 64
MaxClients 1024
</IfModule>
```

Hazelcast – Elevando o desempenho da comunicação remota

Neste tópico vamos discutir como avaliar o comportamento de caches do Hazelcast e estabelecer algumas estratégias para extrair o melhor desempenho possível dos mesmos.

Um dos fatores críticos para o desempenho de qualquer aplicação que troca mensagens pela rede é a velocidade com que objetos podem ser transformados em streams de bytes e vice-versa. As primeiras versões do Hazelcast utilizavam a própria API de serialização do JDK, que é conhecida por oferecer boa portabilidade em troca de um desempenho questionável. Para contornar isto, os desenvolvedores do Hazelcast investiram consideravelmente em criar mecanismos que permitissem aos usuários do framework implementar suas próprias regras de serialização. Dentre estes mecanismos, o mais interessante foi a introdução da interface **DataSerializable**, a qual é muito similar à interface **Externalizable** do JDK, como mostra o código da Listagem 8.

As interfaces **ObjectData(Input/Output)** são derivadas das classes **Data(Input/Output)** do JDK, porém definem alguns métodos auxiliares a mais para escrita de arrays de tipos primitivos. Quando estamos lidando com objetos simples, normalmente é muito fácil implementar esta interface, bastando apenas tomar o cuidado de ler os dados na mesma ordem em que são escritos, como mostra o exemplo da Listagem 9.

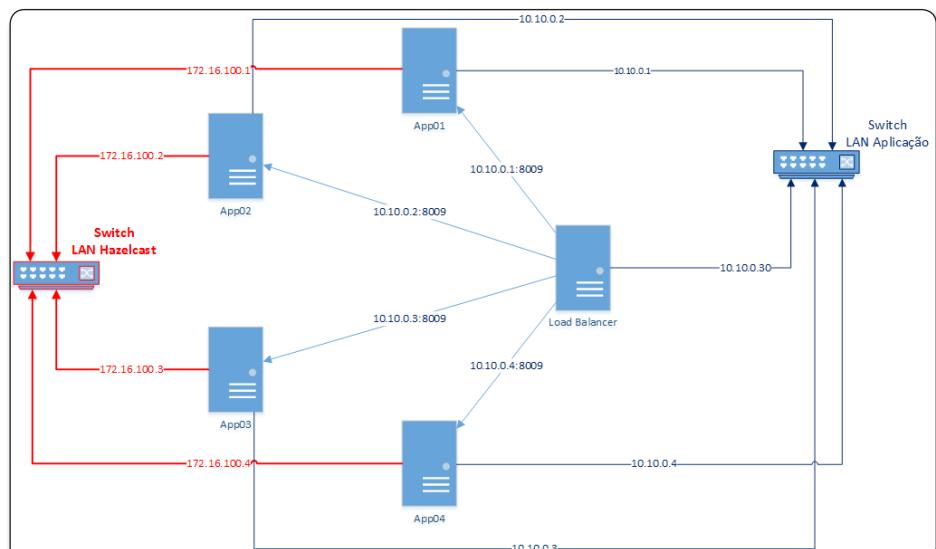


Figura 5. Esquema para rede privada com switches dedicados

Essencialmente, ao implementar a interface **DataSerializable** estamos informando ao Hazelcast para utilizar os métodos **readData()** e **writeData()**, em vez da API do JDK para serializar os objetos, ou seja, estamos evitando que *Type Descriptors* e reflexão

sejam utilizados no processo de serialização, o que em geral reduz consideravelmente o tamanho do objeto serializado e melhora o desempenho de leitura/escrita desses objetos (para mais detalhes, veja a série de artigos “Entendendo o Cache de Segundo Nível no Hibernate”).

Porém o Hazelcast não para por aí! O framework oferece um segundo nível de otimização através da interface **IdentifiedDataSerializable**, a qual deriva de **DataSerializable** e define dois métodos adicionais, como mostra a **Listagem 10**.

Basicamente, a diferença entre serializar objetos utilizando a API padrão do JDK e as interfaces do Hazelcast pode ser representada como mostrado na **Figura 6**.

Todos os esquemas de serialização utilizam um “cabeçalho” para indicar ao framework de serialização do Hazelcast como instanciar um objeto. No caso do JDK, este cabeçalho é muito maior que os correspondentes do Hazelcast, pois toda a informação de nomes e tipos dos campos de um objeto é escrita neste cabeçalho.

Ao utilizar a interface **IdentifiedDataSerializable** esta informação será “indexada” na memória local de cada servidor, o que reduz o tamanho do cabeçalho para apenas 8 bytes, independentemente da quantidade de campos ou do tamanho do nome de uma classe. Esta indexação, porém, não é automática, como ocorre em alguns frameworks como o *GridGain*, ou seja, devemos fazer algumas configurações adicionais para construir o mapeamento Classe \Leftrightarrow [int,int] [idFábrica,idClasse].

Como pode ser visto na **Listagem 10**, esta interface define dois métodos adicionais a serem implementados, além dos métodos **readData()** e **writeData()** da interface pai. O método **getFactoryId()** serve para identificar qual fábrica será usada para instanciar o objeto e o método **getId()** serve para associar uma classe a um número. Para implementar esta estratégia de serialização são necessários quatro passos:

1. Implementar os métodos **read()** e **write()** de **DataSerializable** para cada tipo que se deseje otimizar;
2. Implementar o método **getId()**, de modo que a associação id

\leftrightarrow Classe seja única, ou seja, um mesmo id não pode estar associado a classes diferentes;

3. Criar uma classe para instanciar os tipos a serem otimizados;

4. Alterar a configuração do Hazelcast para associar a classe criada no passo três com o número a ser retornado pelo método **getFactoryId()**.

Seguindo este esquema, se quisermos otimizar a serialização da classe **Message** da **Listagem 9**, primeiramente faríamos as modificações mostradas na **Listagem 11**, onde definimos o id da fábrica que instanciará este objeto, assim como o id da própria classe.

Em seguida é necessário criar uma classe para instanciar **Message** (ou outros objetos)

Listagem 8. A interface DataSerializable.

```
public interface DataSerializable {
    void writeData(ObjectDataOutput out) throws IOException;
    void readData(ObjectDataInput in) throws IOException;
}
```

Listagem 9. Serialização customizada com DataSerializable.

```
public class Message implements DataSerializable {
    int type;
    String payload;

    @Override
    public void readData(ObjectDataInput in) throws IOException {
        type = in.readInt();
        payload = in.readUTF();
    }

    @Override
    public void writeData(ObjectDataOutput out) throws IOException {
        out.writeInt(type);
        out.writeUTF(payload);
    }
}
```

Listagem 10. A interface IdentifiedDataSerializable.

```
public interface IdentifiedDataSerializable extends DataSerializable {
    int getFactoryId();
    int getId();
}
```

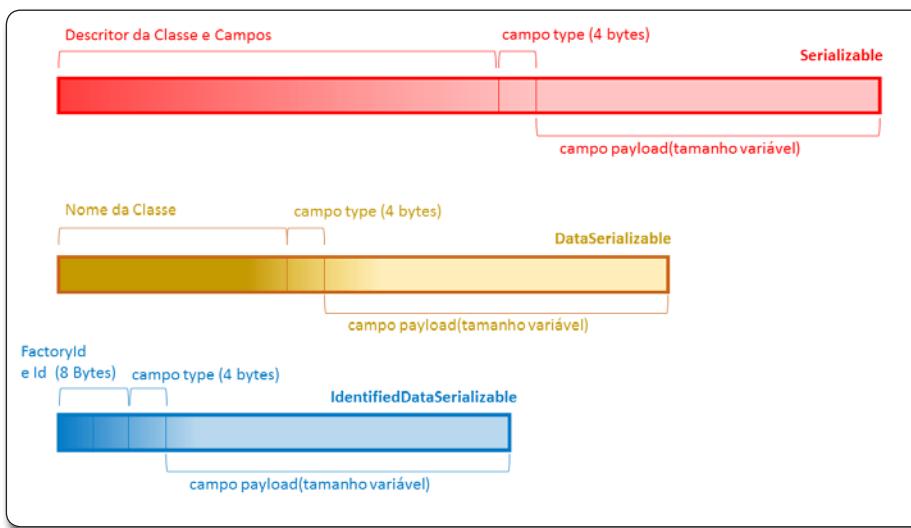


Figura 6. Comparação entre esquemas de serialização Hazelcast e JDK

a partir de identificadores numéricos. Esta classe deverá implementar a interface **DataSerializableFactory**, como mostrado na **Listagem 12**.

Listagem 11. Otimizando a serialização da classe Message.

```
public class Message implements IdentifiedDataSerializable {  
  
    int type;  
  
    String payload;  
  
    @Override  
    public int getFactoryId() {  
        return 1;  
    }  
  
    @Override  
    public int getId() {  
        return 0;  
    }  
  
    // read/write como da Listagem 9  
}
```

Listagem 12. Fábrica para instanciar IdentifiedDataSerializables.

```
public class ObjectFactory implements DataSerializableFactory {  
  
    @Override  
    public IdentifiedDataSerializable create(int typeId) {  
        switch (typeId) {  
            case 0:  
                return new Message();  
            default:  
                throw new IllegalArgumentException("Tipo com " + typeId +  
                    " não está mapeado!");  
        }  
    }  
}
```

Finalmente, é necessário vincular a fábrica da **Listagem 12** a um identificador numérico. Esta associação pode ser feita programaticamente ou declarativamente, como mostrado na **Listagem 13**.

Embora o processo para otimizar a serialização de objetos seja simples de implementar para objetos que contêm apenas atributos simples como primitivos e Strings, o mesmo não é verdade para objetos que contêm referências para outros objetos complexos, como listas e mapas, e se torna ainda mais complexo quando o objeto a ser serializado pode conter referências cíclicas.

Por exemplo, suponha que tenhamos que implementar uma espécie de trilha de auditoria das requisições efetuadas pelos usuários de um sistema, de modo que possamos fazer um “replay” da sequência de acessos. Esses dados serão salvos no banco, porém para melhorar o desempenho vamos mantê-los temporariamente em cache até que tenhamos um volume razoável para persistir. Para representar essa informação, poderíamos utilizar o modelo da **Listagem 14**, que armazena dados pertinentes a uma requisição.

Listagem 13. Configuração do Hazelcast com DataSerializableFactory.

```
<?xml version="1.0" encoding="UTF-8"?>  
<hazelcast xmlns="...">  
  
    <!-- Configuração de rede, caches, tópicos, etc... -->  
  
    <serialization>  
        <data-serializable-factories>  
            <!-- factory-id deve ser idêntico ao valor retornado por getFactoryId() -->  
  
            <data-serializable-factory factory-id="1">  
                br.jm.ObjectFactory  
            </data-serializable-factory>  
        </data-serializable-factories>  
    </serialization>  
</hazelcast>
```

Listagem 14. Modelo para armazenar as requisições feitas por um usuário.

```
package br.jm.model;  
  
public final class RequestNode implements Serializable {  
  
    private static final long serialVersionUID = 1L;  
  
    //URI solicitada  
    String uri;  
  
    //Host do Client  
    String host;  
  
    //Endereço IP do cliente  
    String addr;  
  
    //Próxima Requisição  
    RequestNode prev;  
  
    //Requisição anterior  
    RequestNode next;  
  
    //Cabeçalhos Http  
    Map<String, String> headers;  
  
    //Cookies  
    List<Cookie> cookies;  
  
    //Nível da requisição. Se for muito alto, e.g. 10, implica que devemos persistir os  
    //dados  
    int depth;  
  
    //gets e sets  
}
```

Note que neste modelo a referência para um objeto a ser serializado pode aparecer mais de uma vez, e.g., a instância de **RequestNode** com **depth=1** terá seu campo **prev** “apontando” para um **RequestNode** com **depth=0**, que, por sua vez, terá seu campo **next** “apontando” para o **RequestNode** com **depth=1**, e assim sucessivamente, como mostra a **Figura 7**.

Certamente, se quisermos otimizar a serialização desta estrutura, deveremos fazê-lo de forma a “marcar” objetos que já foram serializados para evitar que o sejam novamente, caso contrário poderemos entrar em um “loop infinito”.

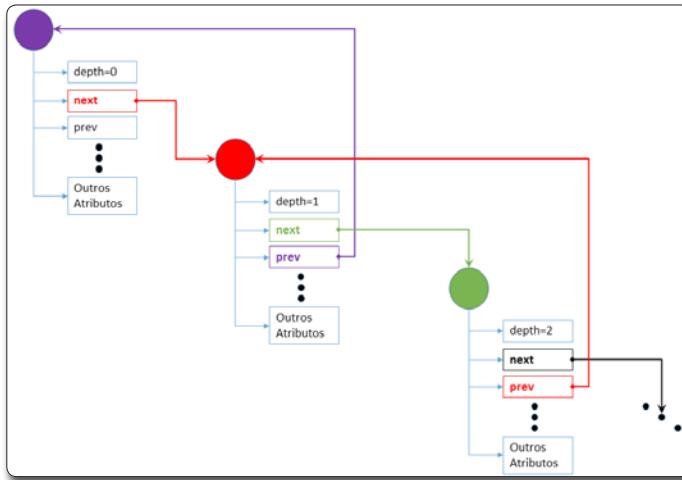


Figura 7. Representação de uma trilha de requisições

Para nos auxiliar nesta tarefa, o ideal é utilizar um framework de serialização já “pronto”, de modo que essas preocupações possam ser deixadas de lado. Existem dezenas de frameworks capazes de lidar com grafos contendo referências cíclicas como o da **Figura 7**, e para este artigo vamos adotar o *graph-serializers*, que provê um desempenho compatível com frameworks como *thrift* e *protocol buffers*, porém sem a necessidade de utilizar mecanismos de pré-compilação. Para mais detalhes de como utilizar o framework, o leitor pode consultar as referências na seção **Links**. No entanto, para o propósito do artigo, o que nos interessa é como integrar o framework com o Hazelcast. Se estivermos utilizando o Java 8, basta criarmos uma interface com métodos *default* implementados, conforme o código da **Listagem 15**. Em versões anteriores do Java, podemos utilizar a mesma estratégia através de herança ou classes utilitárias, caso não seja possível modificar a hierarquia de determinada classe.

Com a interface **IdentifiedGraph**, otimizar a serialização de objetos torna-se uma tarefa trivial, bastando que as classes alvo a implementem e sobrescrevam o método **getId()**. Internamente, o framework de serialização irá otimizar a escrita e leitura dos dados.

Avaliando os ganhos devidos a otimizações de serialização

O Hazelcast possui uma ferramenta chamada *Management Center* que permite visualizar as estatísticas de todas as suas estruturas de dados em tempo real. Infelizmente, essa ferramenta requer uma licença comercial quando se deseja monitorar um cluster com mais de dois nós, o que pode tornar inviável sua utilização. Contudo, toda informação exposta no *Management Center* está disponível em uma API pública de estatísticas do próprio Hazelcast e, portanto, podemos acessá-las e exibi-las da maneira que desejarmos.

Embora possamos coletar métricas de todos os serviços e estruturas disponibilizados pelo Hazelcast, o foco deste artigo serão as estatísticas de Mapas Distribuídos (**IMap**), porém os conceitos discutidos também se aplicam a outras estruturas de dados, visto que internamente todos os objetos a serem armazenados no cluster devem ser serializados.

Listagem 15. Integrando o framework de serialização com o Hazelcast.

```
package br.jm.support;

import java.io.IOException;
import java.io.Serializable;

import com.hazelcast.nio.ObjectDataInput;
import com.hazelcast.nio.ObjectDataOutput;
import com.hazelcast.nio.serialization.IdentifiedDataSerializable;
import com.nc.gs.core.Context;

public interface IdentifiedGraph extends IdentifiedDataSerializable {

    @Override
    public default int getFactoryId() {
        return 1;
    }

    @Override
    public default void readData(ObjectDataInput in) throws IOException {
        try (Context c = Context.reading()) {
            c.inflate(in, this);
        }
    }

    @Override
    public default void writeData(ObjectDataOutput out) throws IOException {
        try (Context c = Context.writing()) {
            c.write(out, this, false);
        }
    }
}
```

Listagem 16. RequestNode otimizado.

```
package br.jm.model;

public final class RequestNode implements IdentifiedGraph {

    //Campos, gets e sets

    //Único método a ser implementado
    @Override
    public int getId() {
        return 1;
    }
}
```

Para avaliar o desempenho da classe **IMap**, podemos partir da API de estatísticas, a qual expõe dois tipos de métricas, rotuladas **LocalMapStats** e **NearCacheStats**. A primeira refere-se a medidas relativas a operações e volume de dados de um cache, do ponto de vista local, isto é, as métricas não refletem a contagem de todos os nós do cluster e sim da instância de um **IMap** atrelada a um determinado nó. Os métodos mais interessantes dessa interface e seu respectivo significado estão enumerados a seguir:

1. **getGetOperationCount()**: retorna o total de operações do tipo “get” efetuadas por um cache para procurar um registro;
2. **getTotalGetLatency()**: retorna o tempo total gasto por um cache para realizar operações do tipo “get”. O tempo total inclui o tempo para realizar uma chamada remota somado ao tempo de deserializar o valor associado à chave utilizada na operação get;

3. `getPutOperationCount()`: similar a `getGetOperationCount()`, porém para inserções no cache;
4. `getTotalPutLatency()`: similar a `getTotalGetLatency()`, porém para operações de put;
5. `getOwnedEntryCount()`: retorna a quantidade de registros em cache;
6. `getOwnedEntryMemoryCost()`: retorna o volume de memória ocupado pelos registros em cache.

A classe `NearCacheStats` define métodos similares à `LocalMapStats`, porém não define o conceito de latência, ou melhor, assume-se que a latência em um Near Cache é praticamente zero, o que não é necessariamente verdade, pois embora não haja um custo de execução de uma operação através da rede, há um custo de CPU associado à desserialização de objetos.

Para avaliar o comportamento de caches distribuídos, vamos adotar a seguinte metodologia:

- Cada nó criará N objetos similares à classe `RequestNode`, em versões padrão (serialização do JDK) e otimizada, de acordo com a medição que desejarmos executar;
- Em seguida, os nós preencherão um `IMap` com esses objetos, para posteriormente realizar operações do tipo `get`;
- Ao terminar as interações, vamos coletar as estatísticas e agregar os valores como *médias* dos nós, quando aplicável. Por exemplo, se executarmos o teste com três nós e o método `getTotalGetLatency()` reportar os valores [10,11,9], vamos utilizar o valor médio $10 = (10+11+9)/3$;
- Para completar nossa análise, vamos levar em conta o desvio padrão dos valores, que pode ser utilizado para identificar “anomalias” em nossas medições e para termos uma ideia de quanto flutuam os valores. Dado um conjunto de valores $\{X_1, \dots, X_N\}$, com média aritmética denotada \bar{X} , \bar{X} , o desvio padrão σ pode ser calculado pela fórmula apresentada na **Figura 8**:

$$\sigma = \sqrt{\frac{\sum_{i=1}^N (X_i - \bar{X})^2}{N - 1}}$$

Figura 8. Fórmula para cálculo do desvio padrão

- No exemplo acima, teríamos $\sigma = 1$. Este valor indica que, se conhecêssemos apenas a média (10), poderíamos dizer que todos os outros valores serão encontrados, aproximadamente, no intervalo $9 (= 10 - 1)$ à $11 (= 10 + 1)$. Graficamente, os desvios são representados como barras verticais, que se estendem para baixo e para cima de um valor médio.
- O código completo e as instruções de como executá-lo estão disponíveis no site da revista.

No “universo” Java, é muito simples abstrair e implementar funções de médias e desvios. Para obter dados estatísticos, partindo de determinado tipo genérico T que encapsula ou pode ser mapeado

para um valor inteiro, dada uma coleção de instâncias de T, o desvio desse valor pode ser expresso conforme a **Listagem 17**.

Analogamente, para o cálculo de desvios de outros tipos numéricos como `long` e `double`, basta utilizar a função de mapeamento correspondente do JDK, e.g. `ToLongFunction`.

Agora, como fazemos para um nó consolidar os dados de todos os outros membros? Simples, podemos utilizar a própria API de Executors do Hazelcast, como mostrado na **Listagem 18**.

Listagem 17. Cálculo de desvio padrão

```
public static <T> double stdDev(List<T> vals,ToIntFunction<T> extractor) {
    int sz = vals.size();
    double avg = vals.stream().mapToLong(e -> extractor.applyAsInt(e)).average()
        .getAsDouble();
    double sum = vals.stream().mapToDouble(e ->
        Math.pow(extractor.applyAsInt(e) - avg, 2)).sum();
    return Math.sqrt(sum / sz);
}
```

Listagem 18. Coletando estatísticas de caches de todos os nós.

```
public static List<LocalMapStats> getAllNodesStats(String instanceName,
String cache) {
    HazelcastInstance instance = HazelcastUtil.hazelcast();
    /*
     * Função que será executada em todos os membros. Note que não podemos
     * passar a variável
     * instance para a função, pois a mesma não é serializável, ou seja, cada nó
     * deverá resolver
     * a sua própria HazelcastInstance localmente!
     */
    Callable<LocalMapStats> lambda = (Callable<LocalMapStats> & Serializable) () -> HazelcastUtil.hazelcast().getMap(cache).getLocalMapStats();
    Map<Member, Future<LocalMapStats>> map = instance.
        getExecutorService("executor").submitToAllMembers(lambda);
    Collection<Future<LocalMapStats>> values = map.values();
    return values.stream().map(f -> {
        try {
            return f.get();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }).collect(Collectors.toList());
}
```

Com estes conceitos, mostraremos a seguir os resultados que obtivemos ao comparar operações em caches distribuídos utilizando os esquemas de serialização padrão e otimizados. Os dados a serem exibidos foram coletados utilizando quatro servidores Dell R720, com dois processadores Xeon octa-core modelo E5-2650 v2 em 2.6GHz.

A **Figura 9** exibe a latência acumulada, utilizando uma amostra de até 100.000 operações.

Em termos percentuais, este “ganho” em redução de latência flutua entre 18 e 20% para operações do tipo put e 23 a 26% para operações do tipo get, como mostra a **Figura 10**.

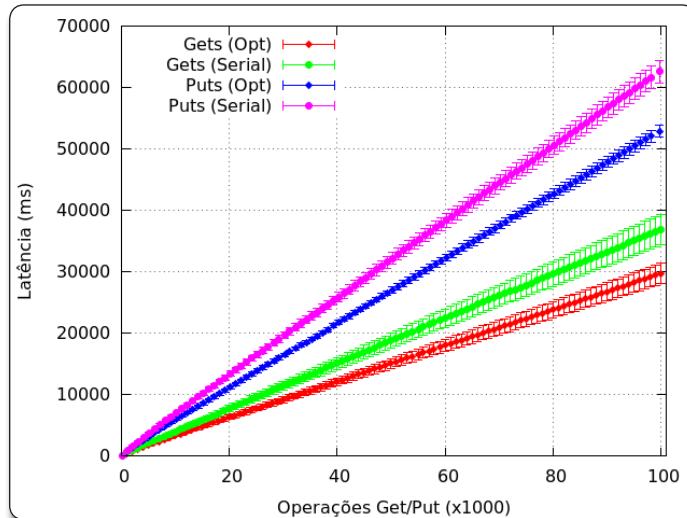


Figura 9. Latência das operações com esquemas de serialização do JDK e otimizado

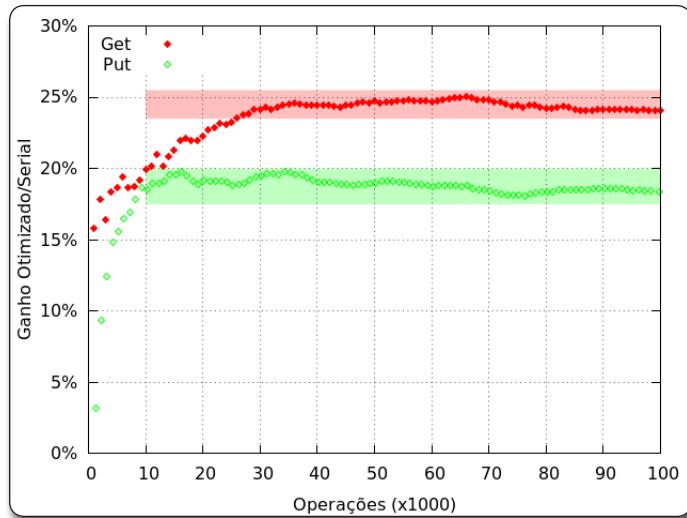


Figura 10. Ganhos percentuais

E o que acontece se colocarmos um Near Cache na história? No teste realizado, utilizamos um Near Cache de maneira que algumas entradas fossem periodicamente substituídas e consequentemente invalidadas nos IMaps dos servidores, possibilitando a verificação do comportamento do cache quando parte de seus dados é removido da memória local. Como mencionamos, quando um registro é obtido de um Near Cache, as estatísticas de latência não são incrementadas, o que implica que não estamos medindo por completo o efeito de otimizações associadas a melhorias na serialização.

A Figura 11 mostra a redução da latência de operações remotas na presença de um Near Cache juntamente com um processo que periodicamente invalida 20% do cache. Note que para ambas as métricas, padrão e otimizada, o gráfico se apresenta em duas linhas divergentes, o que reflete a oscilação no aumento da latência quando o Near Cache é invalidado e, coincidentemente, para o teste em questão, verificou-se que a linha de pior latência

da versão otimizada acompanha a linha de melhor latência da versão padrão.

A oscilação da latência pode ser melhor observada na Figura 12, que mostra apenas parte do conjunto de dados em uma escala ampliada.

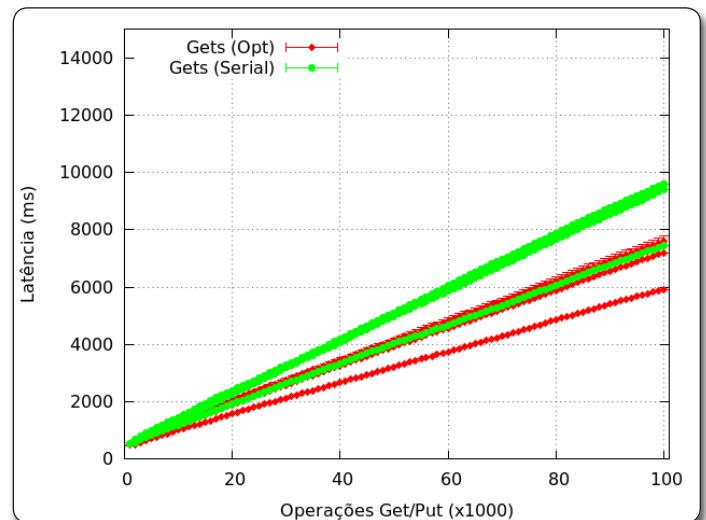


Figura 11. Efeito Near Cache e Invalidação

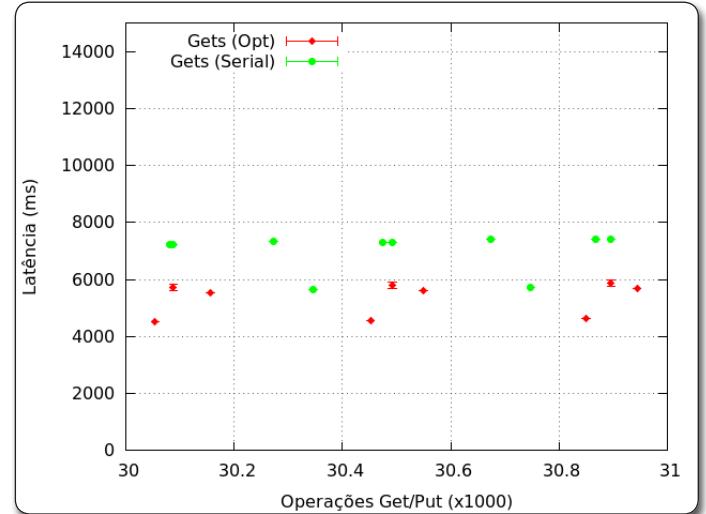


Figura 12. Efeito Near Cache e Invalidação (Zoom)

Se compararmos os gráficos das Figuras 9 e 11, perceberemos que a introdução de um Near Cache provoca uma queda substancial na latência das operações. Entretanto, para ter uma métrica mais “confiável” com relação ao ganho de desempenho, o mais correto seria medir o tempo total para executar operações em um cache distribuído.

Isso pode ser feito utilizando os métodos de processamento em lote do Hazelcast, que provoca a computação de todos os elementos de um IMap em todos os nós. No Hazelcast, uma das maneiras de se fazer isto consiste em utilizar a API de EntryProcessors, que abstrai a noção de iteração sobre todos os elementos de uma

estrutura de dados, popularmente conhecida como “for each”, para um ambiente de Data Grid, porém com a mesma simplicidade das outras operações distribuídas do framework como, por exemplo, as oferecidas pelo **ExecutorService**.

Quando utilizamos um **EntryProcessor**, o Hazelcast se encarrega que cada nó processe apenas os objetos que armazena localmente para, em seguida, transferir seu subconjunto para o nó que solicitou o processamento.

Para o teste em questão utilizamos a implementação de **EntryProcessor** mostrada na **Listagem 18**.

Listagem 18. EntryProcessor para medição de dados.

```
class Processor implements EntryProcessor<Object, Object> {

    private static final long serialVersionUID = 1L;

    @Override
    public EntryBackupProcessor<Object, Object> getBackupProcessor() {
        return null;
    }

    @Override
    public Object process(Entry<Object, Object> entry) {
        Object v = entry.getValue();
        if (v == null) {
            throw new IllegalStateException("null" + entry.getKey());
        }
        return v;
    }
}
```

Esta implementação garante que:

- Operações de desserialização irão ocorrer para todos os valores do **IMap**;
- Nenhum custo adicional de CPU, além da desserialização e mecanismos internos do Hazelcast, influenciará no resultado. Note que efetivamente não “processamos” o objeto, apenas fazemos uma verificação trivial;
- Ao retornar um valor não-nulo no método **process()**, estamos informando ao Hazelcast que queremos utilizar o resultado do processamento da entrada do Mapa passada como parâmetro, ou seja, o valor deverá ser transferido dos nós que executam o processamento para o nó que solicita o processamento;
- Neste caso, a latência será medida como o tempo de execução do método **executeOnEntries()** do **IMap**, como mostrado na **Listagem 19**.

Este trecho de código será executado diversas vezes, considerando um volume de dados no intervalo de [1.000, 100.000] para ser armazenado no **IMap**. Ao realizar estas medições, chegamos ao resultado mostrado na **Figura 13**.

Em termos percentuais, percebemos que utilizar um esquema de serialização otimizado provoca ganhos substanciais, na faixa de 350% a 450%, na maioria das vezes, como mostra a **Figura 14**.

Listagem 19. Código para medir tempo de execução do método **executeOnEntries()**.

```
long now = System.currentTimeMillis();
map.executeOnEntries(new Processor());
long elapsed = System.currentTimeMillis() - now;
```

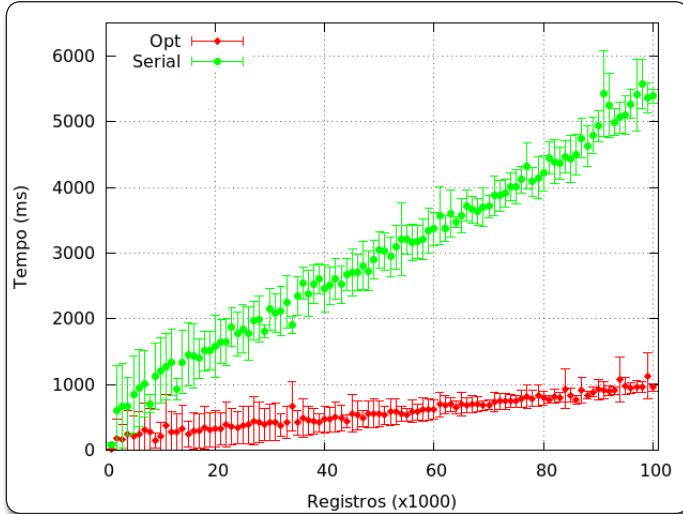


Figura 13. Tempo necessário para realizar o processamento de todos os registros de um **IMap**

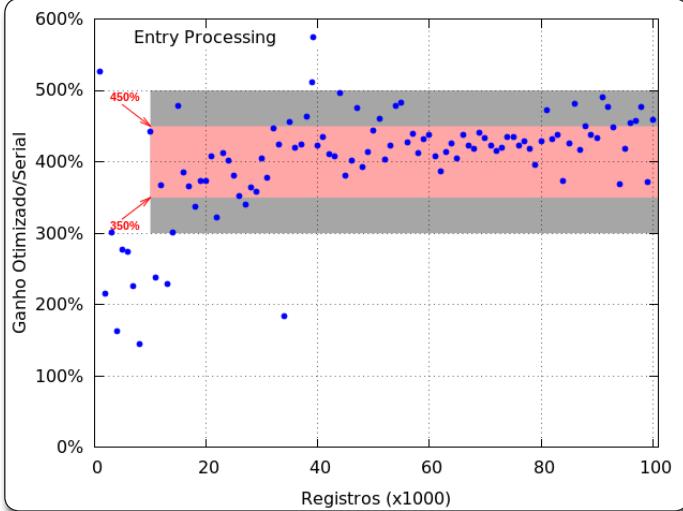


Figura 14. Entry Processing (Ganho Percentual)

Além do ganho em desempenho, também temos ganhos em escalabilidade. A quantidade de memória necessária para armazenar a mesma quantidade de registros em cache é aproximadamente 2.67 menor quando a serialização é otimizada, ou seja, do ponto de vista de escalabilidade vertical, precisamos de menos memória para conseguir armazenar determinado conjunto de dados e, do ponto de vista de escalabilidade horizontal, precisamos de menos nós. O gráfico da **Figura 15** mostra o comparativo entre as versões quando o objeto de estudo é a quantidade de memória alocada por nó, em média.

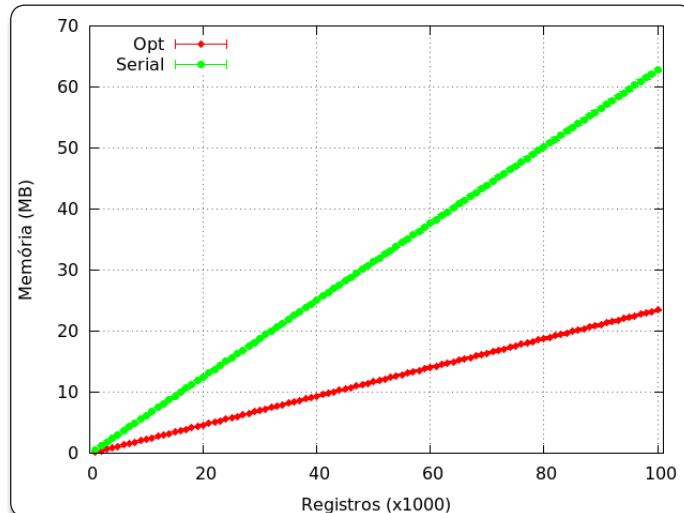


Figura 15. Valor médio de memória alocada por nó

Neste artigo vimos como disponibilizar uma aplicação *stateful* com alta disponibilidade e escalabilidade utilizando o Apache e o Hazelcast, e algumas peculiaridades deste *setup* que devem ser levadas em conta antes de publicarmos esta aplicação em produção.

O Hazelcast é uma excelente ferramenta para processamento distribuído e para que possamos extrair o máximo do poder de processamento, devemos sempre ter em mente o custo associado à serialização de objetos e transmissão de dados pela rede.

As otimizações de serialização discutidas servem para melhorar não somente o desempenho de caches, mas de todas as operações em estruturas de dados e serviços que envolvem a propagação de objetos de um nó para outro, como mensagens enviadas para tópicos e filas. Seguindo o mesmo raciocínio, seria possível também implementar trocas de mensagens mais eficientes em frameworks de invocação remota de métodos como o *Spring Remoting*, reduzindo o overhead de protocolos em serviços acessados por *stubs RMI* ou *HttpInvokers*.

Outro contexto interessante em que se espera obter ganhos substanciais com a aplicação deste tipo de otimização pode ser

explorado em frameworks de Map-Reduce baseados em serialização de objetos, como o próprio Hazelcast ou o GridGain, que realizam o processamento em Listas e Mapas distribuídos. Ao empregar técnicas de compressão do formato binário dos objetos, além de reduzirmos o consumo de memória também ganhamos na redução do número de pacotes a serem transmitidos pela rede.

Autor



Cleber Muramoto

Doutor em Física pela USP, é Especialista de Engenharia de Software na empresa Atech Negócios em Tecnologias-Grupo Embraer. Possui as certificações SCJP, SCBD, SCWCD, OCPJWSD e SCEA.



Links:

Conceitos de Autenticação Client-Cert.

<http://docs.oracle.com/cd/E19226-01/820-7627/bncbs/index.html>

Implementação de Autenticação Client-Cert no Servidor JBoss.

<http://www.manning.com/jamae/>

Framework de Serialização.

<https://github.com/cmuramoto/graph-serializers>

Worker vs Prefork.

<https://developer.jboss.org/wiki/OptimalModjik12Configuration>

APR vs NIO.

<http://tinyurl.com/apr-vs-nio>

Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!



CURSOS ONLINE



A Revista SQL Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

CONHEÇA OS CURSOS MAIS RECENTES:

- **Cursos: Curso de noSQL (Redis) com Java**
- **Desenvolvimento para SQL Server com .NET**
- **Curso PostgreSQL - Treinamento de banco de dados (Curso Básico)**



Para mais informações :

<http://www.devmedia.com.br/cursos/banco-de-dados>

Hibernate: Evitando problemas de mapeamento

Boas práticas de mapeamento para melhorar o desempenho de suas aplicações

ESTE ARTIGO É DO TIPO MENTORING

SAIBA MAIS: WWW.DEVMEDIA.COM.BR/MENTORING-SAIAMAIIS

A camada de persistência das aplicações, independentemente da plataforma ou arquitetura adotadas, sempre são alvos de suspeitas no caso de problemas de desempenho, travamento, escalabilidade, etc. Embora não necessariamente corretas, essas suspeitas são compreensíveis, dado o potencial destrutivo de bancos de dados mal configurados ou equivocadamente projetados e implementados. Por isso, não é absurdo afirmar que bancos de dados são vítimas históricas de más-práticas de projeto e implementação.

No entanto, a camada de persistência também tem sido objeto constante de pesquisa e desenvolvimento, o que vem resultando em diversas implementações interessantes. No histórico dessas implementações achamos interfaces e padrões bastante populares, como o ADO, DAO, ODBC, o próprio JDBC, entre outros, destinados a enfrentar problemas clássicos na camada de persistência de aplicações.

Esses padrões foram bastante úteis a partir da popularização dos bancos de dados relacionais, ocorrida nos anos de 1970 a 1980, que fizeram surgir inúmeros métodos e linguagens para projetar e desenvolver o acesso a dados. Foi nesse cenário que surgiu a linguagem SQL, já com o intuito de formar um padrão universalmente aceito e diminuir a proliferação exagerada de métodos de acesso aos sistemas gerenciadores de bancos de dados.

Sendo assim, uma vez que os problemas de isolamento da camada de persistência foram, de certa forma, assimilados, surgem na esteira da popularização crescente do desenvolvimento orientado a objetos os frameworks de mapeamento objeto-relacional, dentre os quais destacamos o Hibernate, que ocupa um papel de bastante importância no cenário mundial de desenvolvimento OO.

Cenário

Por meio de um exemplo que demonstra uma aplicação web desenvolvida com Hibernate e Spring, esse artigo expõe como algumas más práticas na implementação da camada de persistência passam despercebidas pela maioria dos desenvolvedores, causando problemas graves e erros catastróficos. A partir disso, serão demonstradas aqui formas de evitar esses erros comuns e boas práticas na utilização de recursos do Hibernate que aprimoram o desempenho de aplicações e mitigam falhas recorrentes no projeto de software.

Apesar de serem acusados de produzir modelos de dados pobres ou mal normalizados devido ao alto nível de abstração em que os objetos geralmente são projetados – quando comparados com as tabelas do banco de dados – é correto afirmar que as técnicas de mapeamento objeto-relacional reduzem a quantidade de código e simplificam a tarefa de programação. Com o Hibernate não é diferente, uma vez que ele fornece uma implementação que substitui o acesso direto às entidades da camada de persistência por operações de manipulação de objetos.

Com isso, o Hibernate – cujas primeiras versões foram lançadas no início dos anos 2000 – começa a ter seu uso massificado entre 2003 e 2005, período no qual ocorre a liberação do Hibernate 2 e a implementação da Java Persistence API 2 (JPA).

O Hibernate ORM é a ferramenta do projeto Hibernate que tem como objetivo fornecer uma implementação transparente para a correspondência entre os objetos da aplicação e a base de dados que irá persisti-los, simplificando a consulta ao banco de dados e livrando o desenvolvedor da tarefa de escrever a relação entre objetos da aplicação e as tabelas.

Com o Hibernate, os objetos do *front-end* da aplicação são construídos a partir de princípios da orientação a objetos, enquanto o *back-end* da aplicação seguirá conceitos e princípios de análise relacional e da normalização. Embora esses conceitos não sejam equivalentes, o mapeamento objeto-relacional implementado no Hibernate fornece os meios para a encontrar a compatibilidade necessária.

Apesar de toda a simplificação fornecida pelos recursos do Hibernate – e talvez causada por essa simplificação – não é raro se deparar com problemas causados por más-práticas de implementação ou erros na configuração do mesmo, principalmente quando seu desenvolvimento é realizado por desenvolvedores menos experientes ou sem fundamentos sólidos sobre os princípios de desenvolvimento OO e análise relacional.

Dessa forma, nesse artigo vamos demonstrar más-práticas bastante recorrentes no uso do Hibernate, que embora tenham soluções simples, ainda podem ser encontradas com bastante frequência, talvez devido a sua sutileza ou pelo fato de demandarem conhecimento sobre conceitos normalmente pouco aprofundados e específicos. Explicando suas consequências e como evitá-las, exploraremos más-práticas responsáveis por algumas falhas graves em aplicações, com custo de resolução alto, além do desgaste desnecessário acarretado pelo retrabalho.

Sendo assim, o cenário demonstrado nesse artigo é composto por uma aplicação que apresenta problemas intermitentes. Neste cenário a aplicação funciona normalmente por algum tempo, mas tem seu funcionamento paralisado subitamente, sem causa aparente. Além disso, exceções ocorrem de maneira caótica, sendo difícil relacionar os erros ocorridos com uma única causa. Complicando ainda mais a busca por uma solução, os problemas são resolvidos temporariamente ao reiniciar a aplicação, voltando a ocorrer após algum tempo de uso. Este cenário será descrito com mais detalhes no próximo tópico desse artigo.

Ainda vale destacar que os casos apresentados no cenário brevemente descrito anteriormente ocorrem por dois motivos bastante comuns – não somente entre desenvolvedores Java, mas nas comunidades de desenvolvedores de qualquer arquitetura: mau uso ou má implementação do pool de conexões com o banco de dados (vide **BOX 1**); e mau projeto e implementação de associações entre classes no Hibernate.

BOX 1. Connection Pooling

É uma técnica utilizada para permitir que vários clientes façam uso de um conjunto compartilhado de conexões reusáveis com o banco de dados, reduzindo assim o custo associado ao uso dessas conexões e aumentando a performance do acesso à camada de persistência.

Ambos os problemas podem ser evitados ou resolvidos com medidas simples, muitas vezes fornecidas pelos próprios fabricantes dos frameworks utilizados, e amplamente documentadas. Por isso, acreditamos que esse artigo é importante por fornecer uma descrição do caminho a seguir, exemplificando não só boas e simples práticas para evitar problemas, mas contribuindo para abrir a mente do desenvolvedor e estimular o raciocínio a respeito de como aprimorar a qualidade do projeto de software, mitigando problemas desgastantes e destacando meios de enfrentá-los de maneira mais confortável.

Cenário: uma aplicação web com Hibernate e Spring

Nesse artigo, vamos utilizar como exemplo uma aplicação web em que o usuário é capaz de fazer a manutenção do cadastro de docentes de uma universidade. Esse exemplo foi construído utilizando, além do Hibernate ORM, o framework Spring para o desenvolvimento seguindo a arquitetura MVC. Os detalhes da instalação e implementação do Spring não são objetos desse artigo e, portanto, não serão abordados, assim como a instalação e configuração do Hibernate. Para mais detalhes sobre o Spring MVC e o Hibernate ORM, visite os sites dos seus fabricantes, cujos endereços podem ser encontrados na seção **Links**.

No entanto, daremos foco na implementação do Hibernate e do padrão MVC com o Spring, além de aspectos de sua configuração que tenham relação direta com os problemas que queremos demonstrar nesse artigo ou que interfiram no funcionamento do nosso exemplo. Detalhes da arquitetura MVC, bem como da arquitetura orientada a serviços e do padrão DAO, conceitos usados no desenvolvimento do exemplo demonstrado aqui, também não serão explorados.

Sendo assim, considerando que foi utilizada a IDE Eclipse na implementação do projeto, crie o pacote base da aplicação. Com o botão direito sobre a pasta *Java Resources/src*, no *Package Explorer* do Eclipse, selecione *New > Package*. Conforme demonstra a **Figura 1**, digite “*br.com.devmedia.gestaoacademica*” no campo *Name* e clique em *Finish*.

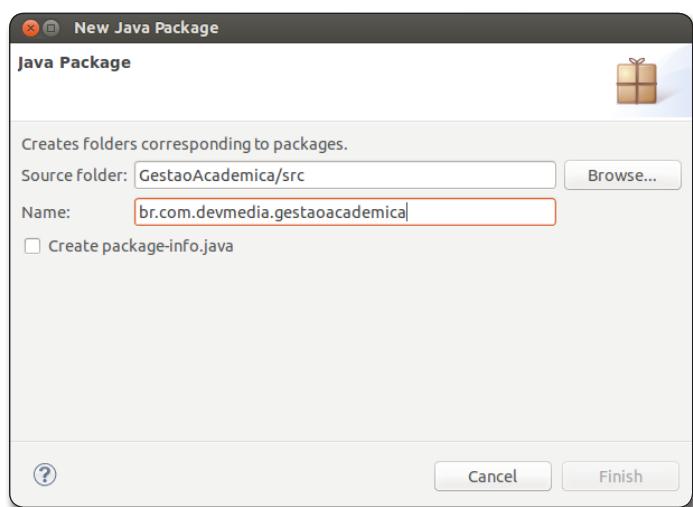


Figura 1. Criando o pacote base da aplicação

Para implementar a arquitetura MVC, crie dentro do pacote base os pacotes **control** e **model**. Crie também o pacote onde serão armazenadas as classes da camada de persistência da nossa aplicação (**dao**) e o pacote para a camada de serviço (**service**). Nossas páginas (a camada *view*) ficarão no diretório *WEB-INF/views*. Dessa forma, a estrutura do projeto deve ficar como mostrado na **Figura 2**.

Três arquivos serão utilizados para manter as configurações de nossa aplicação: */src/hibernate.cfg.xml* (onde manteremos as

Hibernate: Evitando problemas de mapeamento

configurações de mapeamento das classes da camada de modelo no Hibernate); /WEB-INF/hibernate.properties (onde ficarão as configurações de acesso ao banco de dados como senha e endereço); e /WEB-INF/spring-context.xml (onde são mantidas as configurações

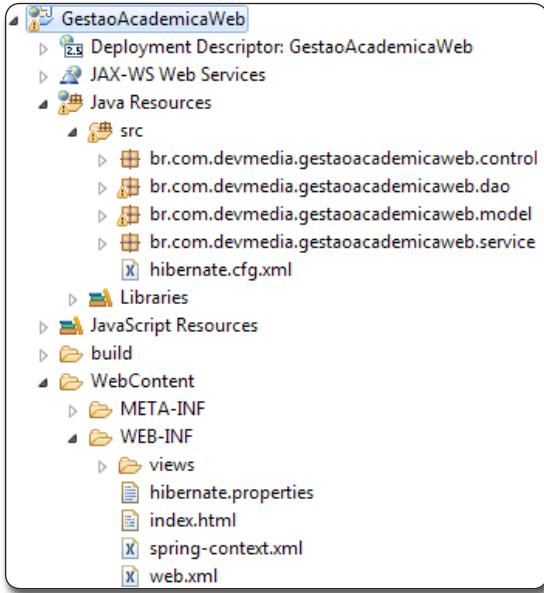


Figura 2. Estrutura do projeto

do Spring e sua integração com o Hibernate). Nesse primeiro momento, este último é o mais importante dos três arquivos e seu código-fonte pode ser visto na **Listagem 1**.

No arquivo *spring-context.xml* é possível ver as configurações que apontam para o arquivo *hibernate.properties* (onde lemos *bean id="propertyConfigurer"*). Nesse trecho é dito para o servidor de aplicação que configurações do Hibernate são encontradas no arquivo *hibernate.properties*. Vemos essas configurações sendo usadas, por exemplo, onde temos o código *p:password="\${hibernate.password}"*.

O exemplo que construiremos nesse artigo é constituído de dois beans na camada de modelo: **Docente** e **Projeto**. O diagrama de classes que ilustra a associação um-para-muitos entre essas duas classes pode ser visto na **Figura 3**.

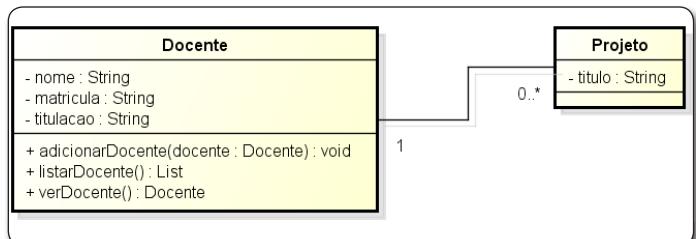


Figura 3. Diagrama de classes no domínio

Listagem 1. Código-fonte do arquivo *spring-context.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<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"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xmlns:lang="http://www.springframework.org/schema/lang"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:util="http://www.springframework.org/schema/util"
    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.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee.xsd
        http://www.springframework.org/schema/lang
        http://www.springframework.org/schema/lang/spring-lang.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util.xsd">

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

    <bean id="jspViewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass"
            value="org.springframework.web.servlet.view.JstlView"/>
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>
```

```
    <bean id="propertyConfigurer"
        class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"
        p:location="/WEB-INF/hibernate.properties"/>

    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="${hibernate.driverClassName}" />
        <property name="url" value="${hibernate.databaseUrl}" />
        <property name="username" value="${hibernate.username}" />
        <property name="password" value="${hibernate.password}" />
    </bean>
    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <property name="configLocation">
            <value>classpath:hibernate.cfg.xml</value>
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
                <prop key="hibernate.show_sql">true</prop>
                <prop key="hibernate.enable_lazy_load_no_trans">true</prop>
            </props>
        </property>
    </bean>
    <tx:annotation-driven />
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate4.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>
</beans>
```

Por pertencerem à camada de modelo, ambas as classes estão localizadas no pacote **br.com.devmedia.gestaoacademicaweb.model**. Em seus códigos-fontes, que podem ser vistos nas **Listagens 2 e 3**, vemos anotações do Hibernate que as associam com suas respectivas tabelas no banco de dados (**@Table**).

Cada atributo das classes também é associado a um campo em sua respectiva tabela por meio da anotação **@Column**.

Podemos ver ainda a implementação da associação entre as classes, expressa por meio da anotação **@OneToMany**, complementada pelo mapeamento correspondente (**mappedBy** na classe **Docente** e **@JoinColumn** na classe **Projeto**). As operações da camada de persistência consistem basicamente em inserir registros do tipo **Docente** no banco de dados (**adicionarDocente()**), bem como recuperá-los em uma operação de listagem (**listarDocentes()**).

Listagem 2. Código-fonte da classe Docente.

```
package br.com.devmedia.gestaoacademicaweb.model;
import java.util.Set;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
@Table(name="DOCENTES")
public class Docente {

    @Id
    @Column(name="ID")
    @GeneratedValue
    private Integer id;

    @Column(name="NOME")
    private String nome;

    @Column(name="MATRICULA")
    private String matricula;

    @Column(name="TITULACAO")
    private String titulacao;

    @OneToMany(mappedBy = "docente")
    private Set<Projeto> projetos;
}
```

```
public Integer getId() {
    return id;
}

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

public String getNome() {
    return nome;
}

public void setNome(String nome) {
    this.nome = nome;
}

public String getMatricula() {
    return matricula;
}

public void setMatricula(String matricula) {
    this.matricula = matricula;
}

public String getTitulacao() {
    return titulacao;
}

public void setTitulacao(String titulacao) {
    this.titulacao = titulacao;
}

public Set<Projeto> getProjetos() {
    return projetos;
}

public void setProjetos(Set<Projeto> projetos) {
    this.projetos = projetos;
}
```

Listagem 3. Código-fonte da classe Projeto.

```
package br.com.devmedia.gestaoacademicaweb.model;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

@Entity
@Table(name="PROJETOS")
public class Projeto {

    @Id
    @Column(name="ID")
    @GeneratedValue
    private Integer id;

    @Column(name="TITULO")
    private String titulo;

    @ManyToOne
    @JoinColumn(name="DOCENTE_ID")
    private Docente docente;
}
```

```
public Integer getId() {
    return id;
}

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

public String getTitulo() {
    return titulo;
}

public void setTitulo(String titulo) {
    this.titulo = titulo;
}

public Docente getDocente() {
    return docente;
}

public void setDocente(Docente docente) {
    this.docente = docente;
}
```

Hibernate: Evitando problemas de mapeamento

Criaremos também uma operação para carregar um objeto do tipo **Docente** (`verDocente()`), mas não implementaremos a inserção de objetos do tipo **Projeto** nesse artigo.

Sendo assim, como parte da camada de persistência em uma versão simplificada do padrão DAO (ou melhor, de sua variação MVC-DAO), criaremos duas classes no pacote **dao**, ou melhor, uma interface (**DocenteDAO**) e sua implementação (**DocenteDAOImpl**) contendo os métodos correspondentes às operações projetadas. Seus códigos-fontes podem ser vistos nas **Listagens 4 e 5**, respectivamente.

Listagem 4. Código-fonte da interface DocenteDAO.

```
package br.com.devmedia.gestaoacademicaweb.dao;  
  
import java.util.List;  
import br.com.devmedia.gestaoacademicaweb.model.Docente;  
  
public interface DocenteDAO {  
  
    public void adicionarDocente(Docente docente);  
    public List<Docente> listarDocentes();  
    public Docente verDocente(int id);  
}
```

Listagem 5. Código-fonte da classe DocenteDAOImpl.

```
package br.com.devmedia.gestaoacademicaweb.dao;  
  
import java.util.List;  
import br.com.devmedia.gestaoacademicaweb.model.Docente;  
import org.hibernate.SessionFactory;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Repository;  
  
@Repository  
public class DocenteDAOImpl implements DocenteDAO {  
  
    @Autowired  
    private SessionFactory sessionFactory;  
  
    public void adicionarDocente(Docente docente) {  
        sessionFactory.getCurrentSession().save(docente);  
    }  
  
    public List<Docente> listarDocentes() {  
        return sessionFactory.getCurrentSession().createCriteria(Docente.class).list();  
    }  
  
    public Docente verDocente(int id) {  
        return (Docente) sessionFactory.getCurrentSession().load(Docente.class, new Integer(id));  
    }  
}
```

Repare em **DocenteDAOImpl** o uso dos métodos do Hibernate `save()`, `list()` e `load()`, para adicionar, listar vários registros e carregar um único objeto do banco de dados, respectivamente.

Na camada de serviços, seguiremos um raciocínio semelhante. Criaremos no pacote **service** uma interface (**DocenteService**) e sua implementação (**DocenteServiceImpl**), cujos respectivos códigos-fontes podem ser vistos nas **Listagens 6 e 7**. Essa camada é responsável por estabelecer quais operações estarão disponíveis

para as camadas clientes e coordenar quais respostas a aplicação dará para cada operação, seguindo o padrão *Service Layer*, simplificado no nosso contexto de exemplo prático.

Listagem 6. Código-fonte da interface DocenteService.

```
package br.com.devmedia.gestaoacademicaweb.service;  
import java.util.List;  
import br.com.devmedia.gestaoacademicaweb.model.Docente;  
  
public interface DocenteService {  
    public void adicionarDocente(Docente docente);  
    public List<Docente> listarDocentes();  
    public Docente verDocente(int id);  
}
```

Listagem 7. Código-fonte da classe DocenteServiceImpl.

```
package br.com.devmedia.gestaoacademicaweb.service;  
  
import java.util.List;  
import br.com.devmedia.gestaoacademicaweb.dao.DocenteDAO;  
import br.com.devmedia.gestaoacademicaweb.model.Docente;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
import org.springframework.transaction.annotation.Transactional;  
  
@Service  
public class DocenteServiceImpl implements DocenteService {  
  
    @Autowired  
    private DocenteDAO docenteDAO;  
  
    @Transactional  
    public void adicionarDocente(Docente docente) {  
        docenteDAO.adicionarDocente(docente);  
    }  
  
    @Transactional  
    public List<Docente> listarDocentes() {  
        return docenteDAO.listarDocentes();  
    }  
  
    @Transactional  
    public Docente verDocente(int id) {  
        return docenteDAO.verDocente(id);  
    }  
}
```

Na camada de visão do nosso exemplo teremos três telas: uma para a listagem dos docentes cadastrados; outra para o cadastro de docentes; e mais uma para apresentar os detalhes do docente, como mostram as **Figuras 4, 5 e 6**. Enquanto a primeira exibe registros cadastrados no banco, a segunda adiciona os registros, quando o usuário clicar no botão *Salvar*, e a última tela simplesmente apresenta os detalhes de um único registro, exibidos quando o usuário clicar em *Ver detalhes* na listagem de docentes. Os códigos-fonte das páginas JSP podem ser vistos nas **Listagens 8, 9 e 10**.

Por fim, implementaremos a camada de controle (no pacote **control**), que recebe as requisições do usuário, de um cliente ou de outra camada e produz as respostas de saída. Vemos o seu código na **Listagem 11**. Conforme o nosso projeto, ela é responsável por gerenciar três tipos possíveis de requisições, que chamamos de “detalhe”, “listar” e “adicionar”.

Listagem 8. Código-fonte da listagem de docentes.

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

<html>
<head>
    <title>Listagem de Docentes</title>
</head>
<body>

<h3>Docentes Cadastrados</h3>

<h3><a href="form">+ Novo Docente</a></h3>
<c:if test="${!empty docenteList}">
<table border="1">
    <tr>
        <th>Nome</th>
        <th>Matrícula</th>
        <th>Titulação</th>
        <th>&nbsp;</th>
    </tr>
    <c:forEach items="${docenteList}" var="docente">
        <tr>
            <td>${docente.nome}</td>
            <td>${docente.matricula}</td>
            <td>${docente.titulacao}</td>
            <td><a href="detalhe/${docente.id}">Ver detalhes</a></td>
        </tr>
    </c:forEach>
    </table>
</c:if>
</body>
</html>
```

Listagem 9. Código-fonte do cadastro de docentes.

```
<%@taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Cadastro de Docentes</title>
</head>
<body>

<h3>Formulário de Cadastro de Docentes</h3>

<form:form method="post" action="adicionar.html" commandName="docente">

    <table>
        <tr>
            <td>Nome:</td>
            <td><form:input path="nome"/></td>
        </tr>
        <tr>
            <td>Matrícula:</td>
            <td><form:input path="matricula"/></td>
        </tr>
        <tr>
            <td>Titulação:</td>
            <td><form:input path="titulacao"/></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Salvar"/>
            </td>
        </tr>
    </table>
</form:form>

</body>
</html>
```

Listagem 10. Código-fonte da página de detalhe do docente.

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

<html>
<head>
    <title>Detalhe de Docentes</title>
</head>
<body>

<h3>Detalhe do Docente</h3>

<h3><a href="form">+ Novo Docente</a></h3>
<c:if test="${!empty docente}">
<table border="1">
    <tr>
        <td>Nome:</td><td>${docente.nome} </td>
    </tr>
    <tr>
        <td>Matrícula:</td><td>${docente.matricula} </td>
    </tr>
    <tr>
        <td>Titulação:</td><td>${docente.titulacao} </td>
    </tr>
<c:if test="${!empty docente.projetos}">
    <tr>
        <td colspan="2" align="center"><b>Projetos</b></td>
    </tr>
    <c:forEach items="${docente.projetos}" var="projeto">
        <tr>
            <td colspan="2">${projeto.titulo}</td>
        </tr>
    </c:forEach>
</c:if>
</table>
</c:if>
</body>
</html>
```

Figura 4. Listagem de docentes cadastrados

Docentes Cadastrados			
+ Novo Docente			
Nome	Matrícula	Titulação	
Catherine Burns	0545	Ph.D.	Ver detalhes
Mario Vidal	7890	Dr. Ing.	Ver detalhes
Paulo Carvalho	3423	D.Sc.	Ver detalhes

Figura 4. Listagem de docentes cadastrados

Formulário de Cadastro de Docentes

Nome:	<input type="text"/>
Matrícula:	<input type="text"/>
Titulação:	<input type="text"/>
<input type="button" value="Salvar"/>	

Figura 5. Formulário de cadastro de docentes

Detalhe do Docente

Nome:	Catherine Burns
Matrícula:	0545
Titulação:	Ph.D.
Projetos	
Avaliacao de risco em saude	
Analise da complexidade	

Figura 6. Detalhes de um docente

Hibernate: Evitando problemas de mapeamento

A página inicial de nossa aplicação será a listagem de docentes, que recebe o mapeamento “index”, implementado com a anotação do Spring `@RequestMapping`. Daremos ao formulário de cadastro de docentes o mapeamento “form”, enquanto a página de detalhes do docente receberá o mapeamento “detalhe”. Já a operação de adicionar não possui uma tela, mas receberá o mapeamento “adicionar”, para que possa funcionar como *action* do formulário de cadastro de docentes (vide **Listagem 10**). Com esses mapeamentos, os endereços das páginas da aplicação estão definidos e elas podem ser acessadas pelo navegador. A página inicial de nossa aplicação pode ser acessada através da URL <http://localhost:8080/GestaoAcademicaWeb/index>.

Listagem 11. Código-fonte da classe DocenteController.

```
package br.com.devmedia.gestaoacademicaweb.control;

import java.util.Map;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import br.com.devmedia.gestaoacademicaweb.model.Docente;
import br.com.devmedia.gestaoacademicaweb.service.DocenteService;

@Controller
public class DocenteController {

    @Autowired
    private DocenteService docenteService;

    @RequestMapping("/index")
    public String listarDocentes(Map<String, Object> map) {
        map.put("docente", new Docente());
        map.put("docenteList", docenteService.listarDocentes());
        return "listar_docentes";
    }

    @RequestMapping("/form")
    public String form(Map<String, Object> map) {
        map.put("docente", new Docente());
        return "inserir_docente_form";
    }

    @RequestMapping(value = "/adicionar", method = RequestMethod.POST)
    public String adicionarDocente(@ModelAttribute("docente") Docente docente,
                                   BindingResult result) {
        docenteService.adicionarDocente(docente);
        return "redirect:/index";
    }

    @RequestMapping("/detalhe/{id}")
    public String verDocente(@PathVariable("id") int id, Model model){
        model.addAttribute("docente", docenteService.verDocente(1));
        return "detalhe_docente";
    }
}
```

Como no nosso exemplo não é necessário implementar o cadastro de objetos do tipo **Projeto**, não criamos nenhuma camada além do modelo para esse tipo. Agora que a construção da nossa aplicação está concluída, podemos testar seu funcionamento em qualquer *container* Java; no nosso caso o Apache Tomcat 8. Rodando a aplicação em ambiente de desenvolvimento, vemos que seu funcionamento não apresenta nenhum problema aparente.

Nesse sentido é preciso reconhecer que diagnosticar más práticas não é uma tarefa trivial. Por vezes vemos casos em que o uso de padrões e a preocupação com boas práticas “pasteurizam” o desenvolvimento de tal forma que geram prejuízos maiores, tanto ao projeto quanto diretamente no código produzido.

No entanto, problemas – ou potenciais causas de problemas – existentes no simples exemplo que acabamos de implementar serão explicitados nos próximos tópicos. Assim, suas consequências serão abordadas e as formas pelas quais eles podem ser mitigados serão investigadas e discutidas.

Como o gerenciamento de conexões pode se tornar um problema

Em um primeiro olhar sobre a aplicação demonstrada, não é possível identificar nenhum problema aparente nem no projeto elaborado, nem no código escrito durante a implementação, bem como na instalação e configuração dos *frameworks* que deram suporte à arquitetura escolhida. Ainda assim, mesmo a aplicação que demonstramos aqui, simples, cujo modelo de domínio possui apenas duas classes, não está livre do risco de queda de desempenho.

Vale destacar que o gerenciamento das requisições e o controle da concorrência no acesso à camada de persistência são temas clássicos em Engenharia de Software, pois de forma geral qualquer aplicação que acesse bancos de dados precisa estar preparada para receber diversos acessos à sua camada de persistência.

De forma geral, o gerenciamento das conexões com o banco de dados é feito por um recurso chamado **pool de conexões**. Basicamente, o que um pool de conexões faz é abrir um determinado número de conexões, disponibilizando-as quando um cliente Java a requisita. Em outras palavras, quando uma requisição é feita, em vez de uma nova conexão com o banco ser aberta, simplesmente uma daquelas já disponíveis é alocada para o cliente. Quando o cliente não necessita mais dessa conexão, em vez de fechá-la, ele a devolve ao pool.

Consequentemente, quando o número de conexões requisitadas ultrapassa do contingente que o pool mantém aberto, uma nova conexão é aberta, a menos que o limite de conexões do pool – que, em geral, é configurável – tenha sido atingido. Nesses casos, normalmente uma exceção é lançada pelo interpretador.

A quantidade de conexões mantidas pelo pool é diretamente proporcional ao montante de recursos utilizados pela plataforma sobre a qual a aplicação está rodando, tanto em memória quanto em processamento. Isso significa que exagerar no consumo pode causar prejuízos sérios ao desempenho da aplicação, aumentando o risco de falhas catastróficas, travamentos e erros.

Nesse sentido, diversos *frameworks* e *containers* disponibilizaram APIs para lidar com o acesso ao banco de dados, algumas vezes fornecendo maneiras próprias de gerenciar o pool de conexões com recursos e propósitos bastante diversos. Tanto o Spring quanto o Hibernate possuem bibliotecas nativas para o gerenciamento de *data sources*, embora nem todas implementem o gerenciamento do pool de conexões – e aquelas que implementam não o fazem necessariamente da maneira mais adequada para o *deployment* de aplicações em ambientes de produção.

No nosso exemplo, deixamos o gerenciamento da conexão com o banco de dados a cargo do Spring, utilizando sua API para o controle de *data sources*, representada pela classe **DriverManagerDataSource** (vide **Listagem 1**). Essa API foi desenvolvida com o intuito de ser utilizada apenas em ambiente de desenvolvimento e, portanto, não possui uma implementação para o pool de conexões.

Dessa forma, o primeiro cuidado que o desenvolvedor precisaria ter ao colocar a nossa aplicação de exemplo em produção é buscar uma nova forma de gerenciar suas conexões com o banco. Não fazer isso é ter a certeza de que a aplicação apresentará problemas de uso.

Ainda assim, uma vez ciente da necessidade de utilizar uma API que implemente o pool de conexões, não seria estranho que o desenvolvedor se sentisse inclinado, então, a deixar o gerenciamento das conexões a cargo do Hibernate, que possui uma implementação nativa de um pool de conexões.

No entanto, isso pode gerar um problema ainda maior, pois o seu pool nativo foi desenvolvido de maneira bastante rudimentar, também visando facilitar o desenvolvimento, e não o *deployment* de aplicações em ambiente de produção. Embora isso esteja descrito de forma clara na documentação do Hibernate, o desenvolvedor deve ficar atento. Sendo assim, para ilustrar o problema e a respectiva solução, observe o código-fonte do arquivo *hibernate.properties* na **Listagem 12**.

Listagem 12. Código do arquivo *hibernate.properties*.

```
hibernate.driverClassName= com.mysql.jdbc.Driver  
hibernate.dialect= org.hibernate.dialect.MySQLDialect  
hibernate.databaseurl= jdbc:mysql://localhost:3306/gestaoacademica  
hibernate.username= gestaoacademica  
hibernate.password= gestaoacademica
```

Como podemos ver, tanto no arquivo *hibernate.properties* quanto em *spring-context.xml* (vide **Listagem 1**) não há configurações para o pool de conexões, dado que a classe **DriverManagerDataSource** não possui essa implementação.

De qualquer forma, o problema pode ser resolvido substituindo a API utilizada para gerenciar o *data source* no Spring por uma que suporte pools de conexões. As mais comuns e populares atualmente são a API DBCP (parte integrante do projeto Apache Commons, cujos pacotes acompanham o Spring Framework) e a API C3P0, cujos pacotes acompanham a distribuição mais recente do Hibernate.

Nesse artigo demonstraremos o uso das duas APIs, substituindo o código de configuração do *data source* no arquivo *spring-context.xml* pelos trechos de código exibidos nas **Listagens 13 e 14**, que correspondem às configurações do DBCP e C3P0, respectivamente.

Listagem 13. Alterações no código do arquivo *spring-context.xml* para utilizar a API DBCP.

```
...  
<bean id="dataSource"  
      class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close"  
      p:driverClassName="${hibernate.driverClassName}"  
      p:url="${hibernate.databaseurl}" p:username="${hibernate.username}"  
      p:password="${hibernate.password}"  
      p:initialSize="${hibernate.initialSize}"  
      p:maxActive="${hibernate.maxActive}"/>  
...
```

Listagem 14. Alterações no código do arquivo *spring-context.xml* para utilizar a API C3P0.

```
...  
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"  
      destroy-method="close">  
    <property name="driverClass" value="${hibernate.driverClassName}" />  
    <property name="jdbcUrl" value="${hibernate.databaseurl}" />  
    <property name="user" value="${hibernate.username}" />  
    <property name="password" value="${hibernate.password}" />  
    <property name="minPoolSize" value="${hibernate.initialSize}" />  
    <property name="maxPoolSize" value="${hibernate.maxActive}" />  
    <property name="maxIdleTime" value="10" />  
</bean>
```

Com isso, uma má-prática foi evitada e o pool de conexões pode ser adequadamente configurado de acordo com as características da aplicação e da arquitetura física em que ela será implantada. Para ilustrar isso, vemos novas propriedades do *data source* correspondentes ao gerenciamento do pool, como **initialSize** e **maxActive**, responsáveis pelo número de conexões abertas na inicialização da aplicação e o limite máximo de conexões ativas.

Implicações do mapeamento das associações no desempenho da aplicação

Não há dúvidas de que um bom projeto e uma boa modelagem ajudam a combater problemas de desempenho da aplicação, ou seja, alguns problemas na camada de persistência podem estar relacionados a tabelas sem índices adequados e consultas mal elaboradas, além, é claro, de acessos desnecessários ou mau gerenciamento das conexões, como vimos anteriormente.

Para ilustrar mais um problema de desempenho causado por uma má prática no uso dos inúmeros recursos disponíveis no Hibernate, observemos a forma pela qual a associação um-para-muitos entre **Docente** e **Projeto** foi implementada (vide **Listagens 2 e 3**). Nesse caso, vemos a implementação de um conceito essencial em bancos de dados relacionais: a junção entre entidades. Embora amplamente utilizados, devemos ter cuidado para que alguns fundamentos sobre os tipos possíveis de junções entre entidades não passem despercebidos.

Hibernate: Evitando problemas de mapeamento

Vemos a junção entre **Docente** e **Projeto** declarada por meio da anotação `@JoinColumn`. Essa anotação indica que a entidade que a possui é a mais forte da associação. No nosso exemplo, um docente possui muitos projetos. Podemos ver no atributo `mappedBy` a indicação da entidade responsável por unir as duas tabelas. Quando não for declarada, o Hibernate procurará por uma tabela associativa. No nosso caso, em que a entidade fraca é a tabela de projetos, declaramos o `mappedBy` indicando que o atributo **projetos** é mapeado pela entidade **Docente**.

Até o presente momento, não há nenhum problema na implementação das associações entre as classes, no entanto, ao declararmos uma associação entre duas classes, devemos definir também o tipo de busca (*fetch*) que será utilizado para recuperá-las através de uma consulta. Em outras palavras, o tipo de *fetch* determina como as associações de um objeto serão recuperadas quando este for carregado. Portanto, a escolha do tipo de *fetch* influencia diretamente na quantidade de idas ao banco de dados que são feitas quando carregamos um objeto e, consequentemente, no desempenho da aplicação.

Existem dois tipos de *fetch*: **EAGER** e **LAZY**. Quando escolhemos o tipo **EAGER**, todas as associações de um objeto são carregadas junto com ele. Por exemplo, toda vez que carregássemos um docente, seus projetos também seriam carregados. No tipo **LAZY**, pelo contrário, quando carregamos um objeto, suas associações não são carregadas. O tipo padrão de *fetch* no JPA é o **LAZY** e vemos na **Listagem 15** como deve ser feita a declaração explícita do tipo de *fetch* na classe **Docente**.

Listagem 15. Declarando de FetchType na classe Docente.

```
...
@OneToOne(mappedBy = "docente", fetch=FetchType.LAZY)
private Set<Projeto> projetos;
...
```

O primeiro problema que vemos nesse aspecto é que, uma vez mantido o tipo **LAZY**, a exibição de atributos do objeto que sejam resultantes de associações precisa ser feita explicitamente, ou seja, é necessário utilizar um método para buscá-los. Por outro lado, principalmente quando há muitas associações, os desenvolvedores tendem a utilizar o tipo **EAGER**, que simplifica a tarefa de programação ao custo do risco de aumento progressivo do tempo de carregamento de um objeto na medida em que a quantidade de registros que fazem parte das associações também aumenta.

É importante ressaltar que o uso de junções **LAZY** é considerado uma boa prática, mas deve ser avaliado de acordo com a aplicação. Se não é necessário carregar todas as associações de um objeto sempre que ele é carregado, utilize **LAZY**. Do contrário, utilize **EAGER**.

Para ilustrar essa situação, repare na **Listagem 16** a consulta gerada pelo Hibernate para responder a uma requisição para a página de listagem de docentes com o tipo de *fetch* declarado como **EAGER**. Podemos ver que o Hibernate realiza uma junção entre *docente* e *projeto*.

Quando declarado como **LAZY**, como mostra a **Listagem 17**, não é feita a junção com a tabela de projetos, que no caso da listagem de docentes é desnecessária.

Listagem 16. Consulta realizada pelo Hibernate com FetchType EAGER ao carregar a listagem de docentes.

```
Hibernate: select this_.ID as ID1_0_0_, this_.MATRICULA as MATRICUL2_0_1_, this_.NOME as NOME3_0_1_, this_.TITULACAO as TITULACA4_0_1_, projetos2_.DOCENTE_ID as DOCENTE_3_0_3_, projetos2_.ID as ID1_1_3_, projetos2_.ID as ID1_1_0_, projetos2_.DOCENTE_ID as DOCENTE_3_1_0_, projetos2_.TITULO as TITULO2_1_0_ from DOCENTES this_ left outer join PROJETOS projetos2_ on this_.ID=projetos2_.DOCENTE_ID
```

Listagem 17. Consulta realizada pelo Hibernate com FetchType LAZY ao carregar a listagem de docentes.

```
Hibernate: select this_.ID as ID1_0_0_, this_.MATRICULA as MATRICUL2_0_0_, this_.NOME as NOME3_0_0_, this_.TITULACAO as TITULACA4_0_0_ from DOCENTES this_
```

Novamente no que se refere à página de detalhe de docente, podemos ver que com o *fetch* do tipo **EAGER**, uma única consulta é realizada, trazendo por meio de uma junção o objeto do tipo **Docente** e seus respectivos projetos, como mostra a **Listagem 18**.

Por outro lado, com o *fetch* do tipo **LAZY**, vemos que o Hibernate precisa realizar mais consultas para carregar a página de docente, uma vez que a junção não foi realizada (vide **Listagem 19**).

Listagem 18. Consulta realizada pelo Hibernate com FetchType EAGER ao carregar a página de detalhe de docente.

```
Hibernate: select docente0_.ID as ID1_0_0_, docente0_.MATRICULA as MATRICUL2_0_0_, docente0_.NOME as NOME3_0_0_, docente0_.TITULACAO as TITULACA4_0_0_, projetos1_.DOCENTE_ID as DOCENTE_3_0_1_, projetos1_.ID as ID1_1_1_, projetos1_.ID as ID1_1_2_, projetos1_.DOCENTE_ID as DOCENTE_3_1_2_, projetos1_.TITULO as TITULO2_1_2_ from DOCENTES docente0_ left outer join PROJETOS projetos1_ on docente0_.ID=projetos1_.DOCENTE_ID where docente0_.ID=?
```

Listagem 19. Consultas realizadas pelo Hibernate com FetchType LAZY ao carregar a página de detalhe de docente.

```
Hibernate: select docente0_.ID as ID1_0_0_, docente0_.MATRICULA as MATRICUL2_0_0_, docente0_.NOME as NOME3_0_0_, docente0_.TITULACAO as TITULACA4_0_0_ from DOCENTES docente0_ where docente0_.ID=?
```

```
Hibernate: select projetos0_.DOCENTE_ID as DOCENTE_3_0_0_, projetos0_.ID as ID1_1_0_, projetos0_.ID as ID1_1_1_, projetos0_.DOCENTE_ID as DOCENTE_3_1_1_, projetos0_.TITULO as TITULO2_1_1_ from PROJETOS projetos0_ where projetos0_.DOCENTE_ID=?
```

```
Hibernate: select docente0_.ID as ID1_0_0_, docente0_.MATRICULA as MATRICUL2_0_0_, docente0_.NOME as NOME3_0_0_, docente0_.TITULACAO as TITULACA4_0_0_ from DOCENTES docente0_ where docente0_.ID=?
```

Dessa forma é bastante pertinente ressaltar que determinar qual tipo de *fetch* é uma boa ou má prática não é algo preciso. Depende das características do projeto e aspectos como escalabilidade e ciclo de vida da aplicação devem ser observados. No nosso caso, o objeto **docente** possui apenas uma associação. Logo, é difícil notar diferenças de desempenho entre o uso dos tipos **EAGER** e **LAZY**. Ainda assim, uma vez que a página inicial do nosso exemplo não

exibe informações de objetos do tipo **Projeto**, o uso do *fetch* de tipo **EAGER** realiza uma junção desnecessária.

Ao contrário do que muitos profissionais possam afirmar, salvo em casos muito claros e simples, não há consenso sobre a distinção entre boas e más práticas, especialmente no que diz respeito à camada de persistência. Historicamente, o trabalho com projetos de bancos de dados é uma categoria bastante específica dentro das ciências da computação, e possui conceitos muito bem fundamentados e estabelecidos. E se destacarmos o trabalho dos administradores de bancos de dados, há ainda mais especificidades.

Sendo assim, a caracterização daquilo que pode ser considerado boa ou má prática na implementação da camada de persistência passa pela percepção dos analistas, e não deve ser tratada como algo cuja discussão está encerrada.

Por isso, o principal objetivo desse artigo é demonstrar as consequências de algumas práticas adotadas por desenvolvedores Java e compará-las com formas diferentes de implementação. Mais do que apresentar soluções, nesse artigo pretendemos estimular a discussão sobre como abordar alguns aspectos da implementação da camada de persistência por meio do *framework* Hibernate, que fornece diversos mecanismos para tornar a vida do desenvolvedor mais fácil, mas que pode induzir o desenvolvedor ao erro, caso não seja dada a devida atenção.

Por exemplo, vemos que a não utilização de um pool de conexões traz riscos claros ao *deployment* de aplicações em ambiente de produção. Em complemento a isso, vemos que, na tentativa de simplificar o desenvolvimento, o Hibernate (e o Spring também) não deixa esse aspecto claro, podendo levar o desenvolvedor a erros catastróficos que, embora possuam soluções simples, podem ser muito difíceis de diagnosticar, principalmente para desenvolvedores inexperientes.

Na sequência, cabe ao desenvolvedor decidir qual é o melhor *fetch* pra as associações de seu modelo. Mesmo no exemplo demonstrando aqui, sem conhecer os requisitos de volume de acesso, escalabilidade e ciclo de vida da aplicação, essa avaliação não é precisa. De qualquer forma, o artigo explorou as características essenciais dos tipos disponíveis e compara os resultados. Nesse caso, cabe ao leitor explorar esses conceitos de acordo com os contextos de

suas aplicações, embora sempre adotando a prática de declarar o *fetch* explicitamente, ciente das implicações existentes.

Por fim, gostaríamos de deixar uma reflexão importante, baseada numa frase atribuída ao designer de automóveis Thomas Gale. Celebrizado por ter desenvolvido belos e funcionais carros para a Chrysler, Gale teria dito que “um bom projeto adiciona valor mais rápido do que adiciona custo” para justificar a forma como se dedicava ao projeto de seus carros. Essa frase pode até ser interpretada de diferentes formas, mas podemos, de fato, assumir que um bom projeto realmente adiciona valor ao produto final, reduzindo riscos, embora também traga novos custos. Ainda assim, os custos de corrigir problemas em produtos durante seu desenvolvimento são sempre menores do que os custos de correção de erros em produtos lançados e em pleno uso.

Autor



Alessandro Jatobá

jatoba@jatoba.org

Mestre em Ciência da Computação pela UFRJ, Doutorando em Engenharia também pela UFRJ com Doutorado-Sanduíche pelo Departamento de Engenharia e Projeto de Sistemas da Universidade de Waterloo, no Canadá. Trabalha com Java há mais de 15 anos, a maior parte deles liderando equipes de desenvolvedores em projetos Web. Também é professor universitário lecionando temas ligados ao desenvolvimento orientado a objetos.



Links:

Site oficial do Hibernate.

<http://www.hibernate.org>

Site oficial do Spring Framework.

<http://www.spring.io>

Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!



Date and Time API: Conheça um dos destaques do Java 8

Novidades da API para manipulação de data e hora introduzida no Java 8

A manipulação de datas e horas está presente na maioria dos softwares, independente do porte ou do negócio por trás do sistema. Infelizmente, devido às limitações dos recursos disponíveis nas versões que antecederam a versão 8, desenvolver um software que possui uma quantidade razoável de manipulação de datas e horas sempre foi algo muito custoso.

Na primeira versão do Java, as datas e horas eram representadas através da classe `java.util.Date`, que basicamente é uma tradução da biblioteca de datas e horas da linguagem C. Em virtude disso, a primeira classe para manipular datas e horas no Java não tinha suporte a internacionalização.

A partir do Java 1.1, com a chegada da classe `java.util.Calendar`, esse problema foi resolvido. Porém, os recursos para manipular datas e horas disponíveis desde a versão 1.1 possuem várias limitações como, por exemplo, a ausência de:

- **type-safe:** como a maioria dos métodos utilizam um inteiro (`int`) como parâmetro, não existem garantias que um determinado valor será válido para o método invocado. Por exemplo, em um método que possui um parâmetro do tipo `int` para representar um mês, é difícil saber qual o valor correto para representar o mês de Janeiro, se 0 ou 1.

- **thread-safe:** por não ser thread-safe, a classe `java.util.Calendar` permite que uma thread interfira e altere informações de uma data e/ou hora de outra thread (interferência esta que não foi prevista durante o desenvolvimento), o que pode gerar um grande efeito colateral no sistema;

Fique por dentro

Este artigo é útil por apresentar a implementação da JSR-310, disponível no Java 8, que reformulou totalmente os recursos para manipulação de datas e horas. Esta reformulação está centralizada em um novo pacote e distribuída em uma série de classes que serão abordadas neste artigo. Durante a apresentação destas classes serão utilizados exemplos práticos com o intuito de facilitar o entendimento e a comparação com a API existente nas versões anteriores.

Até então, trabalhar com data e hora no Java sempre foi algo custoso, devido às limitações da API. Com base nisso, a nova API busca simplificar a manipulação e a representação destas informações, permitindo assim uma maior produtividade e clareza no desenvolvimento de software.

- **flexibilidade:** as classes existentes desde a concepção do Java não possuem flexibilidade para criar novos sistemas de calendários, obrigando os desenvolvedores a utilizar um sistema de calendário pré-estabelecido, como o `GregorianCalendar`.

Com o intuito de contornar os problemas mencionados e proporcionar aos desenvolvedores maior facilidade e produtividade na manipulação de datas e horas, foram projetadas e construídas muitas bibliotecas, sendo a mais popular a Joda-Time.

Com o passar do tempo, a Joda-Time se tornou a API de data e hora padrão (apesar de não oficial) de grande parte dos sistemas desenvolvidos com versões anteriores ao Java 8. Esta é uma API fácil de utilizar, com suporte a diversos sistemas de calendários e com classes específicas para representar datas, horas, instantes, duração, períodos, entre outros.

Visando resolver grande parte dos problemas mencionados, em 2007 foi aprovada pelo JCP (*Java Community Process*) a proposta de uma nova API oficial de data e hora do Java, a JSR-310: Date and Time API.

Visão geral da API

A nova API de data e hora foi construída do zero, buscando corrigir todos os problemas existentes nas versões anteriores, utilizando, para isso, o Joda-Time como fonte de inspiração. Vale ressaltar que o líder da JSR-310 foi o mesmo que projetou a Joda-Time, Stephen Colebourne.

A implementação da JSR-310 foi planejada para ser lançada junto com a versão 8 do Java e possui as seguintes características:

- **Imutabilidade:** todas as classes são imutáveis, garantindo uma fácil utilização em ambientes multi-thread;
- **Separação de conceitos:** foram criadas duas categorias de tempo, uma para humanos (*human time*) e outra para máquinas (*continuous time*);
- **Clareza:** os métodos disponíveis nas novas classes possuem nomes condizentes com sua funcionalidade, além de uma documentação simples;
- **Extensível:** a nova API utiliza como padrão o sistema de calendário ISO-8601, porém permite a criação de outros sistemas de calendário de forma simples.

Com o intuito de manter as características citadas, novas classes foram criadas e são responsáveis por manipular datas, horas, fusos horários, instantes e duração. Estas novas classes estão divididas em cinco pacotes, que são apresentados na **Tabela 1**.

Pacote	Descrição
java.time	O principal pacote da API, contém classes para manipulação de datas, horas, instantes e durações.
java.time.chrono	Pacote para definição e manipulação de sistemas de calendários não compatíveis com a norma ISO-8601.
java.time.format	Este pacote contém as classes usadas para formatar ou analisar (parse) uma data e/ou hora.
java.time.temporal	É uma extensão do pacote principal, adicionando funcionalidades relacionadas a unidades (ano, mês, dia, hora), campos (mês do ano, dia da semana, hora do dia, minuto do dia, segundo do dia), ajustadores temporais personalizados, etc.
java.time.zone	Pacote com classes para suporte a “time zone” e suas respectivas regras.

Tabela 1. Pacotes da nova API de data e hora.

Datas e horas para computadores

A data para computadores é representada por um simples número, incrementado a cada instante, baseado no Unix Time, que é um sistema para descrever um determinado instante em uma linha do tempo. Esse instante é definido como um número que armazena a quantidade de segundos desde a meia-noite do dia 01 de janeiro de 1970 até a data e hora desejadas.

A **java.time.Instant** é a classe da nova API utilizada para representar um instante, que pode ser, por exemplo, o instante que um determinado processamento foi iniciado ou quando o mesmo terminou.

A **Listagem 1** contém um exemplo prático de utilização da classe **Instant**. Neste exemplo, é armazenado em uma variável o instante no qual o processamento foi iniciado e posteriormente, em outra variável, é armazenado o instante em que o mesmo processamento foi concluído.

Listagem 1. Utilizando a classe Instant para calcular o tempo de processamento.

```

01 Instant inicioDoProcessamento = Instant.now();
02 System.out.println(format("Processamento iniciado em {0}",
03 // Processamento iniciado em 2015-03-08T16:36:53.332Z
04
05 // código para validar as informações antes de gerar a nota fiscal
06
07 int numeroNotaFiscal = gerarNotaFiscal(dadosDaCompra);
08
09 // código para notificar outros subsistemas sobre a geração de uma nota fiscal
10
11 Instant finalDoProcessamento = Instant.now();
12 System.out.println(format("Processamento finalizado em {0}",
13 // Processamento iniciado em 2015-03-08T16:36:53.332Z
14
15 Duration tempoProcessamento = Duration.between(inicioDoProcessamento,
16 System.out.println(tempoProcessamento); // PT3.597S
17 System.out.println(tempoProcessamento.toMillis()); // 3597
18 System.out.println(tempoProcessamento.toNanos()); // 3597000000

```

Primeiramente é obtido o instante em que o processamento foi iniciado. Para isso, o método **now()** da classe **Instant** foi invocado. Logo em seguida, o método fictício responsável por gerar a nota fiscal é chamado e por fim, um novo instante é obtido para indicar o término do processamento.

Com os valores relacionados aos instantes de início e fim do processamento é possível calcular o tempo de processamento (ou a duração entre os instantes) utilizando a classe **java.time.Duration**. Esta classe possui o método estático chamado **between()**, que recebe como parâmetro dois instantes e o retorno é uma instância da mesma classe (**Duration**) contendo a duração entre os dois instantes informados como parâmetro.

Conforme exibido na linha 17 (“PT3.597S”), a representação textual de uma instância da classe **Duration** é baseada na ISO-8601. Explicando brevemente este formato com base no resultado do

Nota

A ISO-8601 é uma norma internacional emitida pela Organização Internacional de Padronização (International Organization for Standards - ISO) com o objetivo de eliminar o risco de dupla interpretação quando datas e horas são utilizadas entre sistemas ou além das fronteiras nacionais. Esta norma é baseada no Calendário Gregoriano Proléptico, que o torna compatível com o Sistema de Calendário usado na maioria dos países.

exemplo, a letra ‘P’ é adicionada no início do texto para indicar a representação de uma duração (anteriormente chamada de período). Além da representação textual obtida através do método `toString()`, é possível obter o tempo de duração do processamento em dias (`toDays()`), em horas (`toHours()`), em minutos (`toMinutes()`), em milissegundos (`toMillis()`) e em nanosegundos (`toNanos()`).

Conforme citado anteriormente, a classe `Instant` representa um instante em uma linha do tempo. Por este motivo, não é possível obter o valor de um determinado campo como, por exemplo, a hora do dia. Isso é demonstrado na **Listagem 2**, onde primeiramente é criado um instante através do método estático `parse()`, e depois, na linha 4, é invocado o método `get()`, que recebe como parâmetro um valor da enum `java.time.temporal.ChronoField`. Neste exemplo, é utilizado o `ChronoField.HOUR_OF_DAY`, com o intuito de obter a hora do dia, porém, quando este método é invocado, a exceção `UnsupportedTemporalTypeException` é lançada.

Por fim, é importante lembrar que o fato de não ser possível obter informações campo-a-campo de uma instância de `Instant` (por exemplo: ano, mês, dia do mês, hora do dia, minuto do dia, segundo do dia, etc.), não significa que essas informações não estão presentes no `Instant`, como podemos ver na saída da instrução presente na linha 2, que mostra uma representação textual da instância baseada na ISO-8601.

Listagem 2. Exceção ao tentar obter um campo específico em um Instant.

```
01 Instant instanteQualquer = Instant.parse("2015-12-25T16:36:53.332Z");
02 System.out.println(instanteQualquer); // 2015-12-25T16:36:53.332Z
03
04 instanteQualquer.get(ChronoField.HOUR_OF_DAY);
05 // Exception in thread "main" java.time.temporal.UnsupportedTemporalType
//Exception: Unsupported field: HourOfDay
```

Datas e horas para humanos

A categoria de data e hora para humanos busca apresentar estas informações em um formato mais condizente com o dia a dia. Por este motivo, o tempo é dividido em anos, meses, dias, horas, minutos e segundos. A partir disso, podemos, até mesmo, tratar variações como o horário de verão, fuso horário e até mesmo um sistema de calendário diferente dos compatíveis com a ISO-8601, como o Calendário Judaico.

Datas

Existem diferentes maneiras de representar uma data através da nova API. O primeiro exemplo, apresentado na **Listagem 3**, demonstra como instanciar uma data utilizando a classe `java.time.LocalDate`. Esta classe deve ser adotada em casos que não necessitam manipular hora e/ou fuso horário como, por exemplo: a data de vencimento de uma fatura.

O código exibido na **Listagem 3** diz respeito a uma lógica simples referente ao fechamento de uma fatura. Neste caso, foi considerado que a data de vencimento da fatura deve ser 10 dias após a data do seu fechamento.

Portanto, primeiramente é invocado o método estático `now()`, da classe `LocalDate`, para obter a data atual, que será considerada como a data do fechamento. Em seguida, através do método `plusDays()`, são adicionados 10 dias à data de fechamento.

Listagem 3. Calculando a data de vencimento de uma fatura com `java.time.LocalDate`.

```
01 LocalDate dataFechamento = LocalDate.now();
02 System.out.println(dataFechamento); // 2015-03-09
03
04 LocalDate dataVencimento = dataFechamento.plusDays(10);
05 System.out.println(dataVencimento); // 2015-03-19
06
07 LocalDate dataUltimaCompra = buscarDataDaUltimaCompra();
08 System.out.println(dataUltimaCompra); // 2015-01-02
09
10 Period periodoEntreCompras = Period.between(dataUltimaCompra,
dadosDaCompra.getData());
11 System.out.println(periodoEntreCompras); // P2M7D
12
13 if (periodoEntreCompras.getDays() >= 30) {
14    aplicarDesconto(dadosDaCompra);
15 }
16 // código de validação dos dados da compra antes de iniciar o processo
// para gerar a fatura
17 Fatura fatura = gerarFatura(dadosDaCompra, dataVencimento);
18 // código para notificar os demais subsistemas sobre a geração de uma fatura
```

Outra regra presente no exemplo é conceder determinado desconto se o cliente não realizou uma compra nos últimos 30 dias, de acordo com a data de compra atual. Para realizar esse cálculo é utilizada a classe `java.time.Period`, que possui o método `between()`, capaz de calcular a diferença entre duas instâncias de `LocalDate`. Assim como na classe `Instant`, o método `toString()` da classe `Period` retorna um texto baseado na ISO-8601. Considerando o exemplo anterior, o `Period` é representado pela String “P2M7D” (linha 11), sendo que a letra ‘P’ indica que se trata de um período, M sinaliza a quantidade de meses (no caso, 02) e D a quantidade de dias (07).

Também é importante notar que, além do texto retornado pelo método `toString()`, a classe `Period` disponibiliza os métodos `getDays()`, `getMonths()` e `getYears()` para obter os valores em dias, meses e anos, respectivamente. Na **Listagem 3** o método `getDays()` foi utilizado para verificar se a última compra foi realizada em até 30 dias, permitindo assim que determinado desconto seja concedido.

Uma das novidades da implementação da JSR-310 é o conceito de ajustadores temporais (*Temporal Adjuster*). Estes nada mais são do que uma maneira de modificar uma data, hora ou data/hora. Por padrão, a nova API disponibiliza uma série de ajustadores temporais como, por exemplo, o ajustador que retorna o último dia do mês.

Para verificarmos esse conceito na prática, a **Listagem 4** mantém o foco na funcionalidade descrita no exemplo anterior, porém, a forma de obter a data de vencimento da fatura será implementada utilizando os ajustadores temporais. Além de usar ajustadores, é possível observar que a data de fechamento foi obtida de uma maneira diferente. Neste caso, em vez de obter a data atual, uma data específica foi atribuída e, para isso, o método fábrica

`LocalDate.of()` foi utilizado, passando como parâmetro um dia do mês, o mês e o ano.

Em virtude da classe `LocalDate` ser imutável, ela possui todos os seus construtores privados. Portanto, para a criação de uma nova instância devemos chamar o método `of()`, que recebe como parâmetro o ano, o mês (representado pela enum `java.time.Month`) e o dia.

Listagem 4. Calculando a data de vencimento de uma fatura com `java.time.LocalDate` e `java.time.temporal.TemporalAdjuster`.

```
01 LocalDate dataFechamento = LocalDate.of(2015, Month.FEBRUARY, 1);
02 System.out.println(dataFechamento); // 2015-02-01
03
04 LocalDate dataVencimento = dataFechamento.with(TemporalAdjusters
    .lastDayOfMonth());
05 System.out.println(dataVencimento); // 2015-02-28
06
07 Fatura fatura = gerarFatura(dadosDaCompra, dataVencimento);
```

Nota

A enum `java.time.Month` foi adicionada na nova API para representar os 12 meses de um ano. Além da representação textual com o nome de cada mês, ela também possui um valor inteiro para cada mês respeitando o padrão da ISO-8601. Deste modo, temos do número inteiro 1 (Janeiro) até o número 12 (Dezembro). Apesar disso, mesmo existindo a possibilidade de utilizar os inteiros para representar os meses, é indicado usar a enum `Month` visando garantir maior clareza no código.

Voltando ao exemplo, logo após obter a data de fechamento, é necessário gerar uma data para o vencimento da fatura e para isso utilizamos um ajustador temporal (`TemporalAdjuster`). Os ajustadores temporais devem ser empregados como parâmetros do método `with()`, sendo que os mesmos podem ser os já implementados pela nova API ou algum personalizado.

No exemplo da **Listagem 4** é usado o ajustador temporal `lastDayOfMonth()`, que é responsável por retornar uma data com o último dia do mês, por exemplo: "2015-02-28". Além desse ajustador temporal, existem outros disponíveis como, por exemplo, o primeiro dia do mês, primeiro/último dia do próximo mês, primeiro/último dia do próximo ano, etc. Apesar dessa variedade, em alguns casos é necessário criar um ajustador personalizado, como no caso de precisarmos retornar o próximo dia útil baseado em uma data.

A **Listagem 5** mostra uma maneira de criar um ajustador temporal que retorna o próximo dia útil a partir de uma data. Para isso, basta criar uma classe que implemente a interface `java.time.temporal.TemporalAdjuster` e consequentemente o método `adjustInto()`, que recebe como parâmetro um objeto do tipo `Temporal`.

Na primeira linha do método `adjustInto()` é adicionado um dia no parâmetro recebido. Para realizar essa adição é utilizado o método `plus()`, que recebe como parâmetro a quantidade que será adicionada e a unidade da data, representada pela enum `ChronoUnit.DAYS` (o mesmo método pode ser utilizado para adicionar meses, anos, horas, minutos, etc.).

Listagem 5. Ajustador temporal personalizado para retornar o próximo dia útil.

```
01 public class ProximoDiaUtil implements TemporalAdjuster {
02     @Override
03     public Temporal adjustInto(Temporal temporal) {
04         Temporal diaSeguinte = temporal.plus(1, ChronoUnit.DAYS);
05
06         int diaDaSemana = diaSeguinte.get(ChronoField.DAY_OF_WEEK);
07         if (diaDaSemana == DayOfWeek.SATURDAY.getValue()) {
08             return diaSeguinte.plus(2, ChronoUnit.DAYS);
09         } else if (diaDaSemana == DayOfWeek.SUNDAY.getValue()) {
10             return diaSeguinte.plus(1, ChronoUnit.DAYS);
11         } else {
12             return diaSeguinte;
13         }
14     }
15 }
```

Após obter o próximo dia, devemos descobrir se o mesmo se trata de um dia útil ou não (sem considerar feriados). Para isso, é invocado o método `get()` com o parâmetro `ChronoField.DAY_OF_WEEK`, indicando que o retorno será um inteiro que representa o dia da semana.

Por fim, se o próximo dia fizer parte de um final de semana, então serão adicionados dois dias na data enviada como parâmetro se for sábado ou 1 dia se for domingo, garantindo assim que o ajustador sempre retorne o próximo dia útil.

A **Listagem 6** apresenta como deve ser utilizado um ajustador temporal personalizado. Neste exemplo, consideramos que a data de vencimento deve ser o próximo dia útil após a data de fechamento. Como verificado, todas as linhas dessa listagem já foram explicadas. O que é importante analisar aqui é a linha 5, onde ocorre a chamada ao método `with()`, que recebe como parâmetro uma instância do nosso ajustador. Como neste exemplo a data de fechamento utilizada é uma sexta-feira, 06/03/2015, ao chamarmos o ajustador temporal, é retornado para `dataVencimento` a data referente à próxima segunda-feira, 09/03/2015.

Listagem 6. Utilizando o ajustador temporal personalizado.

```
01 LocalDate dataFechamento = LocalDate.of(2015, Month.MARCH, 6);
02 System.out.println(dataFechamento); // 2015-03-06
03 System.out.println(dataFechamento.getDayOfWeek()); // FRIDAY
04
05 LocalDate dataVencimento = dataFechamento.with(new ProximoDiaUtil());
06 System.out.println(dataVencimento); // 2015-03-09
07
08 Fatura fatura = gerarFatura(dadosDaCompra, dataVencimento);
```

Horas

A classe `LocalTime` armazena apenas informações relacionadas a hora, minuto e segundo e assim como `LocalDate` não possui informações sobre fuso horário. Um exemplo básico de utilização dessa classe é o controle da hora de partida e chegada de uma linha de ônibus que acontece todos os dias, independente de uma data.

O exemplo apresentado na **Listagem 7** considera que é suficiente manipular apenas a hora de partida e chegada na linha de ônibus em questão. Além disso, note que definimos uma regra que, se a

compra da passagem for realizada nos cinco minutos que antecedem a partida e a quantidade de passageiros for maior que 40, a passagem deve ser vendida para o próximo ônibus, que sempre partirá após 30 minutos.

Listagem 7. Manipulando horas com LocalTime.

```
01 LocalTime horaDaPartida = LocalTime.of(19, 00);
02 System.out.println(horaDaPartida); // 19:00
03
04 LocalTime horaDaCompra = LocalTime.now();
05 System.out.println(horaDaCompra); // 18:56
06
07 long diferencaEmMinutos = horaDaCompra.until(horaDaPartida, ChronoUnit.
    MINUTES);
08 System.out.println(diferencaEmMinutos); // 4
09
10 if (diferencaEmMinutos <= 5 && getQuantidadeDePassageiros() > 40) {
11     LocalTime novaHoraDaPartida = horaDaPartida.plusMinutes(30);
12     System.out.println(novaHoraDaPartida); // 19:30
13 }
14
15 processarVenda();
```

Em primeiro lugar é criada uma instância de **LocalTime** através do método **of()** para representar a hora de partida do ônibus. Assim como a classe **LocalDate**, **LocalTime** não possui construtores públicos e a criação de novas instâncias deve ser a partir dos métodos fábrica, como o método **of()** que recebe como parâmetro a hora e o minuto.

Logo em seguida, uma nova instância de **LocalTime** é criada. Dessa vez utilizando o método **now()**, que retorna a hora atual que será utilizada como a hora da compra da passagem. Para contemplar uma das regras relacionadas à venda das passagens, é necessário obter a diferença entre a hora da compra e a hora da partida. Para isso, o método **until()** é invocado a partir da instância da hora da compra, com os parâmetros hora da partida e a unidade de tempo que será retornada; neste caso foi optado por receber a diferença em minutos, especificando então o valor **MINUTES** da enum **ChronoUnit**.

Por fim, uma estrutura condicional foi adicionada para aplicar a regra que obriga a venda da passagem para o próximo horário se a compra for realizada com menos de cinco minutos de antecedência (hora partida) e já existir mais de 40 passageiros vendidas. Visando manter a passagem dentro da regra estabelecida, é preciso alterar a hora de partida para o próximo horário disponível, e para isso, o método **plusMinute()** é invocado para adicionar os 30 minutos.

Para enriquecer nossa explicação, outro exemplo é demonstrado na **Listagem 8**. Neste caso, será calculado o tempo total de viagem de uma linha de ônibus e o tempo que o ônibus ficou transitando de fato. Para isso, primeiramente é criada uma instância de **LocalTime** usando o método **of()** para representar a hora de partida. Note que além da hora e do minuto, neste caso o campo **segundos** também foi especificado. Em seguida, é calculada a hora prevista de chegada, adicionando o tempo de duração da viagem (em horas) à hora de partida. Esse cálculo é feito através do método **plusHours()**.

Listagem 8. Calculando a duração entre duas instâncias de LocalTime.

```
01 LocalTime horaDaPartida = LocalTime.of(19, 01, 34);
02 System.out.println(horaDaPartida); // 19:01:34
03
04 int duracaoViagemEmHoras = 4;
05
06 LocalTime chegadaPrevista = horaDaPartida.plusHours(duracaoViagemEmHoras);
07 System.out.println(chegadaPrevista); // 23:01:34
08
09 LocalTime horaDaChegada = LocalTime.of(23, 05, 12);
10 System.out.println(horaDaChegada); // 23:05:12
11
12 Duration tempoDeViagem = Duration.between(horaDaPartida, horaDaChegada);
13 System.out.println(tempoDeViagem); // PT4H3M38S
14
15 Duration tempoRealDeViagem = tempoDeViagem.minusMinutes(30);
16 System.out.println(tempoRealDeViagem); // PT3H33M38S
```

Posteriormente é feito o cálculo da duração da viagem. Este cálculo considera a hora da partida e a hora da chegada (criada com o método **of()** na linha 09). A duração entre elas é obtida com a ajuda da classe **Duration**, a mesma adotada para calcular a duração entre instantes (**Instant**).

Nota

O método **between()** da classe **Duration** recebe como parâmetro dois objetos do tipo **Temporal**, permitindo assim que ele seja utilizado tanto com **Instant** quanto com **LocalTime**. Porém, se o método **between()** for invocado com parâmetros distintos como, por exemplo, o primeiro parâmetro do tipo **LocalTime** e o segundo do tipo **Instant**, antes de realizar o cálculo da duração o segundo parâmetro será convertido para o mesmo tipo do primeiro, garantindo assim a compatibilidade entre eles.

Continuando com o exemplo da **Listagem 8**, o retorno do método **between()**, que conterá o tempo de viagem, é uma instância da classe **Duration**, que tem sua representação textual definida pela norma ISO-8601, conforme citado anteriormente. Sendo assim, o texto de saída “PT4H3M38S” (vide linha 13) significa que a viagem tem uma duração de 4 horas, 3 minutos e 38 segundos.

Para finalizar, é calculado o tempo em que o ônibus de fato transitou pela rodovia. Isso é feito subtraindo o tempo de descanso previsto, estipulado em 30 minutos. Essa operação de subtração é feita com o método **minusMinutes()**, que retorna uma outra instância de **DateTime** com a duração de 3 horas, 33 minutos e 38 segundos (PT3H33M38S).

Datas e Horas

Note que até aqui analisamos apenas classes para manipular datas e horas separadamente, porém em alguns casos é preciso utilizar estas duas informações em conjunto como, por exemplo, para gerenciar a agenda de um consultório médico, que tem consultas em uma data e hora específica. Pensando nisso, foi criada a classe **java.time.LocalDateTime**. Esta classe usa internamente uma instância de **LocalDate** e outra de **LocalTime** para conseguir gerenciar a data e a hora em conjunto.

Listagem 9. Agendando uma consulta utilizando recursos de data e hora do Java 8.

```
01 LocalDateTime dataAgendamento = LocalDateTime.now();
02 System.out.println(dataAgendamento); // 2015-03-10T21:51:14.766
03
04 LocalDateTime dataUltimaConsulta = LocalDateTime.of(LocalDate.of(
05     2015, Month.FEBRUARY, 25),
06     LocalTime.of(9, 0));
07 System.out.println(dataUltimaConsulta); // 2015-02-25T09:00
08
09 LocalDateTime dataDesejavelDaProximaConsulta = dataUltimaConsulta.
10    plusWeeks(3);
11 System.out.println(dataDesejavelDaProximaConsulta); // 2015-03-18T09:00
12
13 LocalDateTime dataDisponivelProximaConsulta = LocalDateTime.of(
14     2015, Month.JUNE, 1, 10, 0);
15 System.out.println(dataDisponivelProximaConsulta); // 2015-06-01T10:00
16
17 if (dataDisponivelProximaConsulta.isAfter(dataDesejavelDaProximaConsulta)){
18     long semanasDiferenca = dataDesejavelDaProximaConsulta.until(
19         dataDisponivelProximaConsulta,
20         ChronoUnit.WEEKS);
21     System.out.println(semanasDiferenca); // 10
22
23     emitirAlerta(semanasDiferenca);
24 }
```

O exemplo exibido na **Listagem 9** simula uma etapa simples de um agendamento de uma consulta médica. Para isso, logo no início é obtida a data e hora atual, através do método `now()`, apenas para possuir a data em que foi iniciado o processo de agendamento. Depois, uma instância de `LocalDateTime` é criada com o auxílio do método `of()` para representar a data da última consulta. Este método, como verificado na linha 4, recebe como parâmetro uma instância de `LocalDate` e outra de `LocalTime`.

Em seguida, a data e hora desejável para a próxima consulta é criada adicionando três semanas na data da última consulta. Para isso, o método `plusWeeks()` foi invocado a partir da variável que contém a data da última consulta, com o parâmetro 3. Já na linha 11, a data disponível para a próxima consulta é criada também com o método `of()`; porém, neste caso, os dados de ano, mês, dia, hora e minuto são informados como parâmetros.

Por fim, se a data disponível para a próxima consulta estiver depois da data desejável, é necessário emitir um alerta com a quantidade de semanas de atraso da consulta. Para efetuar esse passo, o método `isAfter()`, que retorna se uma data/hora é posterior a outra, é invocado e logo depois a diferença em semanas que representa o atraso na consulta é calculado com o método `until()`, já explicado no subtópico “Horas”.

Dia/Mês/Ano e Ano: Os novos formatos

Além das classes que representam data, hora e data/hora citadas anteriormente, a nova API introduziu outras três classes, responsáveis por representar alguns tipos que não existiam nas versões anteriores ao Java 8, como:

- **MonthDay**: armazena apenas os valores de dia e mês. Pode ser utilizada para representar datas importantes que se repetem todo ano como, por exemplo, o Natal, 25 de Dezembro (vide **Listagem 10**, linha 2);

- **YearMonth**: armazena apenas os valores de mês e ano. Pode ser utilizada para representar datas em que não é necessário especificar o dia como, por exemplo, as Olimpíadas, Agosto de 2016 (linha 6);
- **Year**: armazena apenas o valor correspondente ao ano. Pode ser utilizada para representar apenas o ano, onde o dia e mês não são importantes como, por exemplo, o ano da implantação do plano Real, 2004 (linha 10).

Listagem 10. Novas classes para representar Dia/Mês, Ano/Mês e Ano.

```
01 /* java.time.MonthDay */
02 MonthDay.of(Month.DECEMBER, 25);
03 // -12-25
04
05 /* java.time.YearMonth */
06 YearMonth.of(2016, Month.AUGUST);
07 // 2016-08
08
09 /* java.time.Year */
10 Year.of(2004);
11 // 2004
```

Datas com Fuso Horário

As classes apresentadas anteriormente, que representam data e/ou hora, não possuem informações relacionadas a fuso horário. Essa informação é tratada apenas pela classe `java.time.ZonedDateTime`, que além do fuso horário, possui informações de data e hora.

O fuso horário se torna necessário quando uma determinada data/hora deve ser considerada em regiões ou países com fuso horário distintos. Um bom exemplo de uso de data/hora com fuso horário é a partida e a chegada de um voo internacional (ou até mesmo nacional).

A **Listagem 11** apresenta uma forma de calcular a duração de um voo internacional entre a cidade de São Paulo e Paris. Note nas primeiras linhas que uma instância de `LocalDateTime` foi criada com a data de partida e outra com a data de chegada. No entanto, ambas foram criadas sem as informações de fuso horário, que serão adicionadas posteriormente quando as instâncias de `ZonedDateTime` forem criadas.

Para armazenar as informações de fuso horário, uma instância da classe `ZonedDateTime` é criada a partir da invocação do método `of()`, que recebe como parâmetro uma instância de `LocalDateTime` e o fuso horário (instância de `ZoneId`). Já a definição do fuso horário é feita através da classe `ZoneId`, utilizando o método `of()` que recebe como parâmetro uma `String` responsável por identificar o fuso horário desejado. Neste exemplo serão utilizados os fusos horários de “America/Sao_Paulo” e “Europe/Paris”, conforme as linhas 9 e 18. A primeira instância de `ZonedDateTime` foi criada para representar a data e hora de partida com o fuso horário de São Paulo. Em seguida, são criadas duas instâncias de `ZonedDateTime` que representam o horário de chegada, sendo uma para o fuso horário de São Paulo e outra para o fuso horário de Paris.

Date and Time API: Conheça um dos destaques do Java 8

Listagem 11. Manipulando data/hora com fuso horário distintos.

```
01 LocalDateTime dataHoraPartidaLocal = LocalDateTime.of(LocalDate.of(2015, Month.MARCH, 3),  
02 LocalTime.of(17, 0));  
03 System.out.println(dataHoraPartidaLocal); // 2015-03-03T17:00  
04  
05 LocalDateTime dataHoraChegadaLocal = LocalDateTime.of(LocalDate.of(2015, Month.MARCH, 4),  
06 LocalTime.of(9, 0));  
07 System.out.println(dataHoraChegadaLocal); // 2015-03-04T09:00  
08  
09 ZoneId fusoHorarioSaoPaulo = ZoneId.of("America/Sao_Paulo");  
10 ZonedDateTime dataHoraPartidaSaoPaulo = ZonedDateTime.of(dataHoraPartidaLocal,  
11 fusoHorarioSaoPaulo);  
12 System.out.println(dataHoraPartidaSaoPaulo);  
// 2015-03-03T17:00-03:00[America/Sao_Paulo]  
13  
14 ZonedDateTime dataHoraChegadaSaoPaulo =  
ZonedDateTime.of(dataHoraChegadaLocal,  
15 fusoHorarioSaoPaulo);  
16 System.out.println(dataHoraChegadaSaoPaulo);  
// 2015-03-04T09:00-03:00[America/Sao_Paulo]  
17  
18 ZoneId fusoHorarioParis = ZoneId.of("Europe/Paris");  
19 ZonedDateTime dataHoraChegadaParis = ZonedDateTime.of(dataHoraChegadaLocal, fusoHorarioParis);  
20 System.out.println(dataHoraChegadaParis); //  
2015-03-04T09:00+01:00[Europe/Paris]  
21  
22 Duration duracaoFusoSP = Duration.between(dataHoraPartidaSaoPaulo,  
dataHoraChegadaSaoPaulo);  
23 System.out.println(duracaoFusoSP); // PT16H  
24  
25 Duration duracaoFusoParis = Duration.between(dataHoraPartidaSaoPaulo,  
dataHoraChegadaParis);  
26 System.out.println(duracaoFusoParis); // PT12H
```

Assim como nos demais exemplos, o método **between()** foi utilizado para calcular a duração do voo. Se calcularmos a duração considerando o fuso horário de São Paulo para a partida e chegada (conforme a linha 22), será retornado que o voo tem duração de 16 horas. No entanto, o cálculo de duração do voo foi realizado de maneira incorreta, pois o fuso horário de São Paulo está sendo utilizado tanto para a data e hora de partida quanto para a data e hora de chegada. Para calcular a duração do voo de forma correta, é necessário usar a data de partida com o fuso horário de São Paulo e a data de chegada com o fuso horário de Paris. Com estas informações corretas (linha 25), o tempo de voo é calculado conforme o esperado, retornando que a duração do voo é de 12 horas.

Integração entre tipos temporais

As principais classes da nova API de datas foram apresentadas separadamente, porém é possível que em determinados casos exista a necessidade de criar uma instância de uma classe a partir de outra, podendo essa classe estar na mesma categoria de dados temporais ou não.

Pensando nisso, a nova API fornece alguns métodos capazes de realizar essa operação de forma simples, como podemos verificar na **Listagem 12**. Esta listagem apresenta quatro exemplos

relacionados às classes que representam as datas para humanos (**LocalDate**, **LocalTime** e **LocalDateTime**).

No primeiro exemplo, uma instância de **LocalDate** é criada através do método **of()**. Em seguida, as informações de hora e minuto são adicionadas utilizando o método **atTime()**, que retorna uma instância de **LocalDateTime**. Logo depois, no segundo exemplo, uma instância de **LocalTime** é criada também com o método **of()**, e posteriormente, com o método **atDate()**, são adicionadas as informações de data baseadas em uma instância de **LocalDate**. Como podemos observar, o retorno do método **atDate()** é uma instância de **LocalDateTime** que possui tanto as informações de data quanto as informações de hora.

Listagem 12. Integração entre classes da mesma categoria – datas para humanos.

```
01 /* 1. Exemplo */  
02 LocalDate data = LocalDate.of(2015, Month.JANUARY, 1);  
03 LocalDateTime dataComHora = data.atTime(6, 30);  
04 System.out.println(dataComHora); // 2015-01-01T06:30  
05  
06 /* 2. Exemplo */  
07 LocalTime hora = LocalTime.of(13, 45, 16);  
08 LocalDateTime horaComData = hora.atDate(LocalDate.of(2015, Month.APRIL, 27));  
09 System.out.println(horaComData); // 2015-04-27T13:45:16  
10  
11 /* 3. Exemplo */  
12 LocalDateTime dataHora = LocalDateTime.now();  
13 LocalDate somenteData = dataHora.toLocalDate();  
14 System.out.println(somenteData); // 2015-03-07  
15  
16 /* 4. Exemplo */  
17 LocalTime somenteHora = dataHora.toLocalTime();  
18 System.out.println(somenteHora); // 10:32:28.890
```

Continuando a análise sobre o código da **Listagem 12**, o terceiro exemplo demonstra como criar uma instância de **LocalDate** a partir de uma **LocalDateTime**. Neste caso, o método **toLocalDate()** é invocado e retorna uma instância da classe **LocalDate** apenas com as informações de data. Por fim, o quarto exemplo utiliza o método **toLocalTime()** para criar uma instância de **LocalTime** com base na mesma instância de **LocalDateTime** usada no terceiro exemplo.

Nestes casos, apenas as classes da categoria de data para humanos foram utilizadas. Porém, em alguns casos, será necessário criar uma instância de uma classe que representa a data para humanos a partir de uma que representa a data para computadores. Felizmente esse tipo de funcionalidade também foi previsto pela nova API, conforme expõe a **Listagem 13**.

O primeiro exemplo dessa listagem demonstra como criar uma instância de **LocalDateTime** baseado em uma instância de **Instant**. Para isso, o método **ofInstant()** da classe **LocalDateTime** deve ser invocado, e como parâmetros, devem ser utilizadas a instância da classe **Instant** que será utilizada como base e uma instância da classe **ZoneId**.

É importante notar que nenhuma das classes em questão possui informações relacionadas ao fuso horário, porém ele é requerido pelo método **toInstant()**. Isso é necessário porque o instante criado

através do método `now()` é baseado no fuso horário UTC. Como a classe `LocalDateTime` armazena a data e hora local (sem fuso horário), o método `toInstant()` precisa saber qual o fuso horário local para converter as informações.

Listagem 13. Integração entre classes de categorias distintas – Datas para Humanos X Datas para Computadores

```
01 /* 1. Exemplo */
02 Instant instante = Instant.now();
03 System.out.println(instante); // 2015-03-11T18:38:55.642Z
04 LocalDateTime dataHoraAPartirInstante = LocalDateTime.ofInstant(instante,
05 ZoneId.of("America/Sao_Paulo"));
06 System.out.println(dataHoraAPartirInstante); // 2015-03-11T15:38:55.642Z
07
08 /* 2. Exemplo */
09 LocalDateTime dataHora = LocalDateTime.now();
10 Instant instanteAPartirDataHora = dataHora.toInstant(ZoneOffset.ofHours(-3));
11 System.out.println(instanteAPartirDataHora); // 2015-03-11T18:38:55.660Z
```

Ao analisar a saída da linha 3, identificamos que no final existe a letra 'Z', que caracteriza o horário em UTC. Além disso, também é possível identificar que a instância da classe `Instant` (linha 3) está três horas à frente da instância da classe `LocalDateTime` (linha 6), exatamente a diferença entre o fuso horário de São Paulo e o UTC (no período em que não existe horário de verão).

Nota

UTC, que em português significa Tempo Universal Coordenado, é o fuso horário de referência a partir do qual se calculam todos os demais fusos horários do mundo.

Além destas opções, em alguns casos pode ser necessário utilizar as classes da nova API em conjunto com as classes existentes nas versões anteriores ao Java 8. Um exemplo disso pode ser encontrado ao utilizar JPA, visto que esta API ainda não possui suporte à nova API de data e hora, o que deve ocorrer na plataforma Java EE 8. Deste modo, ao utilizar a nova API em conjunto com o JPA, será preciso converter as novas classes para as "antigas" e vice-versa.

Visando sanar essa necessidade, a versão 8 do Java além de adicionar novas classes, realizou algumas alterações nas já existentes (`Date` e `Calendar`), adicionando alguns métodos para viabilizar uma integração mais fácil.

A **Listagem 14** traz alguns exemplos de integração entre as versões das classes que representam data e hora. Primeiramente, é criada uma instância da classe `java.util.Date` através do método `from()` – adicionado na versão 8 – que recebe como parâmetro uma instância de `java.time.Instant`. Logo depois, é criada uma instância de `java.util.TimeZone` a partir de uma instância de `java.time.ZoneId`; dessa vez utilizando outro método recém-adicionado: `getTimeZone()`. Por fim, o novo método `from()` da classe `java.util.GregorianCalendar` é responsável por criar uma instância de `GregorianCalendar` a partir de uma instância de `java.time.ZonedDateTime`.

De modo semelhante, as integrações inversas também são possíveis, e para isso, como esperado, outros métodos foram incluídos nas classes "antigas" (`Date` e `Calendar`). Vejamos alguns exemplos na **Listagem 15**.

Listagem 14. Criando instâncias das classes "antigas" baseadas nas novas classes.

```
01 Date dataAPartirInstante = Date.from(Instant.now());
02
03 TimeZone fusoHorarioAPartirZoneld = TimeZone.getTimeZone(
04     ZoneId.of("America/Los_Angeles"));
05
06 GregorianCalendar dataHoraAPartirZonedDateTime = GregorianCalendar.
07     from(ZonedDateTime.now());
```

Listagem 15. Criando instâncias das classes novas baseadas nas "antigas".

```
01 Instant instanteAPartirDate = new Date().toInstant();
02 System.out.println(instanteAPartirDate); // 2015-03-11T13:59:02.434Z
03
04 Instant instanteAPartirCalendar = Calendar.getInstance().toInstant();
05 System.out.println(instanteAPartirCalendar); // 2015-03-11T13:59:02.440Z
06
07 ZoneId fusoHorarioAPartirTimeZone = TimeZone.getDefault().toZoneId();
08 System.out.println(fusoHorarioAPartirTimeZone); // America/Sao_Paulo
09
10 ZonedDateTime dataHoraComFuso =
11     new GregorianCalendar().toZonedDateTime();
12 System.out.println(dataHoraComFuso);
13 // 2015-03-11T10:59:02.462-03:00[America/Sao_Paulo]
```

Nos dois primeiros exemplos, uma instância de `Instant` é criada a partir do novo método `toInstant()` adicionado às classes `Date` e `Calendar`. O terceiro exemplo mostra como criar uma instância de `ZoneId` baseada em um `TimeZone`, o que é feito através do método `toZoneId()`. E no último exemplo, o método `toZonedDateTime()`, também adicionado na versão 8 do Java, é invocado a partir de uma instância de `GregorianCalendar` e retorna uma instância de `ZonedDateTime`.

Formatação e Parsing

Durante o desenvolvimento de um software é comum encontrar informações de datas em vários formatos e muitas são representadas por uma `String`, em vez de algum tipo específico de data e/ou hora. Além disso, também é comum ter que criar uma data e/ou hora a partir de um texto (parsing) ou mesmo fazer o processo inverso, gerar uma representação textual de uma data e/ou hora (formatação).

A principal opção de parsing e formatação nas versões anteriores à versão 8 do Java é utilizar a classe `java.text.SimpleDateFormat`, que permite a execução desses processos através dos métodos `format()` e `parse()`. No entanto, a classe `SimpleDateFormat` pode se tornar um sério problema ao ser utilizada em sistemas multi-thread, pois ela não é thread-safe. Em virtude disso, não é possível garantir que não haverá conflito entre threads concorrentes. Em outras palavras, um parse pode falhar porque outra thread alterou o formato esperado de uma data.

Com o intuito de melhorar os problemas existentes na classe `SimpleDateFormat`, a implementação da JSR-310 adicionou o

Date and Time API: Conheça um dos destaques do Java 8

pacote `java.time.format`, que tem como classe principal a `DateTimeFormatter`. Esta classe possui constantes com formatos pré-definidos para realizar a formatação ou *parsing* de uma data e/ou hora. Além disso, `DateTimeFormatter` disponibiliza o método `ofPattern()`, que recebe uma `String` que representa um formato de data e/ou hora específico, a ser utilizado durante o processo de formatação (linha 13).

Como exemplo desses recursos, na **Listagem 16** são apresentadas as duas maneiras de formatar uma data e/ou hora. A primeira utiliza o método `format()`, que recebe como parâmetro um formato pré-definido e identificado por uma constante da classe `DateTimeFormatter` (verifique as linhas 4, 7 e 10). Logo em seguida é exibida a segunda maneira, que utiliza o método `ofPattern()`. Este recebe como parâmetro um texto que representa um formato específico (por exemplo: `dd/MM/yyyy`) para formatação da data (verifique as linhas 13 e 16).

Listagem 16. Formatando datas e horas com auxílio da classe `DateTimeFormatter`.

```
01 LocalDateTime dataHora = LocalDateTime.of(2015, Month.JANUARY, 5, 20, 17);
02 // 2015-01-05T20:17
03
04 dataHora.format(DateTimeFormatter.BASIC_ISO_DATE);
05 // 20150105
06
07 dataHora.format(DateTimeFormatter.ISO_WEEK_DATE);
08 // 2015-W02-1
09
10 dataHora.format(DateTimeFormatter.ISO_DATE_TIME);
11 // 2015-01-05T20:17:00
12
13 dataHora.format(DateTimeFormatter.ofPattern("dd/MM/yyyy"));
14 // 05/01/2015
15
16 dataHora.format(DateTimeFormatter.ofPattern("d. MMMM yyyy", new
Locale("pt", "BR")));
17 // 5. Janeiro 2015
```

As constantes e o método `ofPattern()`, da classe `DateTimeFormatter`, também podem ser adotados no processo de *parsing*, que consiste basicamente em obter uma `String` em um formato específico e convertê-la em um objeto da nova API que representa uma data e/ou hora.

As formas de utilização da classe `DateTimeFormatter`, em conjunto com o método `parse()`, presente nas principais classes da nova API, são apresentadas na **Listagem 17**.

A tão aguardada API de data e hora introduziu um novo conceito e várias classes, permitindo a manipulação de datas e horas de forma simples e segura. Devido a essas novidades, é provável que as classes `java.util.Date` e `java.util.Calendar` sejam depreciadas nas próximas versões e posteriormente removidas.

As classes disponibilizadas pela nova API apresentam várias novidades/melhorias que foram descritas durante o artigo, porém, a adoção dela deve ocorrer de forma gradativa, tendo em vista que é necessário um certo tempo até que todos os frameworks estejam aptos a utilizá-la. O JPA, por exemplo, ainda não é compatível com os novos recursos, o que deve ocorrer oficialmente com o lançamento da plataforma Java EE 8.

Listagem 17. Parsing de datas e horas com auxílio da classe `DateTimeFormatter`.

```
01 // Cria uma instância de LocalDate a partir de um texto sem informar
// o padrão da data
02 LocalDate.parse("2015-01-06");
03 // 2015-01-06
04
05 // Cria uma instância de LocalDate a partir de um texto de acordo
// com o formato definido
06 // na constante ISO_WEEK_DATE
07 LocalDate.parse("2015-W10-1", DateTimeFormatter.ISO_WEEK_DATE);
08 // 2015-03-02
09
10 // Cria uma instância de LocalDate a partir de um texto que define o
// formato dd.MM.yyyy
11 LocalDate.parse("06.01.2015", DateTimeFormatter.ofPattern("dd.MM.yyyy"));
12 // 2015-01-06
13
14 // Cria uma instância de LocalDate a partir de um texto (que define o
// formato d. MMMM yyyy)
15 // e o Locale
16 LocalDate.parse("1. January 1970", DateTimeFormatter.ofPattern(
    "d. MMMM yyyy",
    17 new Locale("en", "US")));
18 // 1970-01-01
```

E como esta nova API está diretamente vinculada à versão 8 do Java, muitos desenvolvedores podem não ter um contato imediato com tais recursos por utilizarem versões anteriores em seus projetos. Para contornar essa limitação, o leitor pode adotar o projeto ThreeTen, uma alternativa que permite o uso das novas classes da Date and Time API no Java 7.

Autor



Leonardo Comelli

leonardo.comelli@gmail.com – <http://leocomelli.com.br>
Graduado em Ciências da Computação e Especialista em Banco de Dados, atualmente trabalha como Líder Técnico de Transição e Transformação na HP.



Links:

Site do projeto Joda-Time.

<http://www.joda.org/joda-time/>

Why JSR-310 isn't Joda-Time.

http://blog.joda.org/2009/11/why-jsr-310-isn-joda-time_4941.html

JSR 310: Date and Time API.

<https://jcp.org/en/jsr/detail?id=310>

Java SE 8 Date and Time

<http://www.oracle.com/technetwork/articles/java/jf14-date-time-2125367.html>

ISO-8601 - Data elements and interchange formats.

http://www.iso.org/iso/catalogue_detail?csnumber=26780

Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!



FÓRUM

DEVMEDIA

O lugar perfeito para você ficar por dentro de tudo
o que acontece nas tecnologias do mercado atual



No fórum da DevMedia você irá encontrar uma equipe disponível e altamente qualificada com consultores e colaboradores prontos para te ajudar a qualquer hora e sobre qualquer assunto. Temos as salas de Java, .NET, Delphi, Banco de Dados, Engenharia de Software, PHP, Java Script, Web Design, Automação comercial, Ruby on Rails e muito mais!

ACESSE AGORA
www.devmedia.com.br/forum



Como desenvolver um código fonte de qualidade em Java

Primeiros passos para aprimorar a qualidade do seu código fonte na linguagem Java

Qualidade de software é um tema muito discutido na atualidade, porém, normalmente leva-se em consideração apenas se o mesmo atende aos requisitos estabelecidos. Neste cenário, pouco se fala sobre a qualidade do código fonte, apesar deste ser o primeiro passo para um software atingir um nível de excelência.

Com base nisso, neste artigo serão abordados pontos importantes como o bom uso de padronizações, a relevância da tradicional dupla alta coesão e baixo acoplamento, a preocupação com performance e robustez, a importância de logging e documentação de código fonte, o valor dos testes unitários bem escritos e a utilização de validadores de código.

Qualidade de software é um assunto de extrema importância, pois um software que não atenda a todos os requisitos ou cheio de bugs não serve para nada. Entretanto, também deve-se levar em consideração a qualidade do código fonte do mesmo, ou seja, o quanto bem desenvolvido foi este software. Digamos que a qualidade do código fonte é a qualidade interna do software, aquela que não é vista pelo cliente, mas que, do mesmo modo, não deve ser desprezada.

Podemos fazer um comparativo do desenvolvimento de software com a fabricação de relógios. Um relógio pode ser muito bonito e perfeitamente funcional (mostra as horas, tem despertador, calendário, etc.), porém o que não vemos, e é de fundamental importância para a qualidade deste, são os componentes internos, como tambor, rodas e escapamento. Isso que faz a diferença entre um excelente relógio, que dura uma vida inteira,

Fique por dentro

Este artigo apresenta dicas, boas práticas e conceitos que devem ser aplicados durante todo o ciclo de desenvolvimento de software. A partir disso, é possível alcançar um código com boa legibilidade e de fácil compreensão, facilitando, principalmente, a manutenção. Portanto, este artigo é útil tanto para programadores, que desejam melhorar o seu nível profissional, como para gerentes ou líderes de equipes de desenvolvimento, que esperam não apenas um software que atenda aos requisitos do cliente, mas também um bom software, bem feito, com código fonte de qualidade.

e uma imitação qualquer, cujo conserto, se quebrado, seria mais alto que o valor pago na compra ou talvez nem seja possível de ser feito.

O que queremos mostrar com essa comparação é que, além de funcionar conforme a especificação, é fundamental que o software seja bom e bem feito, que dure, que a manutenção não seja tão custosa e que a evolução seja natural. Só atingiremos estes objetivos se tivermos código fonte de qualidade, e para isso temos que ter equipes de qualidade.

Em busca desses objetivos, foi criado em 2009 o Manifesto pelo Artesanato de Software (*Manifesto for Software Craftsmanship*), apresentado a seguir:

“Como aspirantes a Artesãos de Software elevamos o nível do desenvolvimento de software profissional ao praticar e auxiliar outros a aprender o ofício. Através deste trabalho passamos a valorizar:

- Não somente software funcionando, mas também software bem feito.

- Não somente responder a mudanças, mas também continuamente adicionar valor.
- Não somente indivíduos e interações, mas também uma comunidade de profissionais.
- Não somente colaboração com o cliente, mas também parcerias produtivas.

Ou seja, na busca pelos itens à esquerda descobrimos que os itens à direita são indispensáveis."

A ideia do manifesto é termos programadores melhores, mais interessados na qualidade do que estão fazendo, buscando sempre o aprimoramento, colaborando uns com os outros e com os clientes.

Quando falamos em código fonte de qualidade, nos referimos a um código padronizado (não apenas com os padrões da linguagem de programação) e que o padrão seja bem conhecido e seguido por todos na equipe; um código coeso e com baixo acoplamento; que seja performático e robusto; que apresente boa manutenibilidade e seja bem documentado; falamos também de um código bem testado (com testes unitários bem escritos e com boa cobertura, para garantir que todo nosso código de boa qualidade esteja atendendo à especificação).

Para nos auxiliar nesta difícil tarefa, um dos recursos que podemos lançar mão são os validadores de código, que podem ser manuais (feito por outro programador) ou automatizados (utilizando ferramentas de análise estática de código, como PMD, CheckStyle e FindBugs).

Nos tópicos a seguir iremos descrever melhor cada um dos itens supracitados, destacando sua importância para a qualidade do código fonte.

Padronização

É muito importante que todo o projeto de software seja bem padronizado, desde a formatação do código fonte, a nomenclatura de classes, métodos e arquivos, até o design propriamente dito. Para tirarmos proveito da padronização, ela deve ser bem conhecida e assimilada por toda a equipe.

Além disso, os padrões adotados no projeto podem ser evoluídos com o andamento do mesmo, desde que haja ampla divulgação dentro da equipe, a qual inclusive deve contribuir para esta evolução.

Formatação

Ao estabelecermos uma formatação padrão ao projeto, além do código ficar organizado e claro, com uma visão uniforme para todos, poderemos ter uma grande vantagem no momento de fazermos o merge de alterações concorrentes, pois as versões do arquivo em questão conterão apenas as diferenças reais, e não diferenças de formatação, que por muitas vezes dificultam esse trabalho.

Quanto maior o número de elementos de formatação que conseguirmos padronizar, melhor! Dentre estes elementos podemos citar endentação, tabulação, número de colunas do editor, se

demarcadores de início de bloco devem ficar na mesma linha do comando ou na linha de baixo, etc.

Em algumas IDEs, como o Eclipse, temos a possibilidade de criar um perfil de formatação. Neste caso, podemos criar um perfil com o padrão adotado no projeto e compartilhar com a equipe. O programador pode então escrever seu código da maneira que quiser e formatar automaticamente com base no perfil definido. Ainda sobre o Eclipse, por exemplo, faríamos isso com o atalho *Shift+Ctrl+F* ou acessando esta opção via menu.

Nomenclatura

A padronização de nomenclatura impacta diretamente na velocidade do desenvolvimento, na facilidade de manutenção e até na diminuição da curva de aprendizado de programadores novos no projeto.

Esta padronização deve ser iniciada escolhendo o idioma que será utilizado. Deve-se escolher o que for melhor para a equipe, e é importante que o mantenha, evitando o código no qual encontramos métodos com nomes escritos em mais de uma língua ou, até mesmo, métodos com uma parte do nome em uma língua e a outra parte em outra. A padronização dos nomes deve ser aplicada tanto para nomes de métodos como para nomes de classes e arquivos em geral.

Vejamos alguns exemplos de padronização da nomenclatura de métodos:

- Métodos sempre devem iniciar com um verbo que indique a funcionalidade do mesmo, como `calcularValorNotaFiscal()`, `persistirNotaFiscal()`, etc. Neste caso, podemos aumentar o nível de padronização, especificando um determinado verbo para uma determinada situação. Por exemplo:

- Sempre que o método incluir um novo registro no banco de dados, o nome deve iniciar com o verbo `incluir`, seguido do nome do elemento, como em `incluirNotaFiscal()`, a fim de evitar que alguns utilizem cadastrar, ou adicionar, ou salvar, etc.;

- Sempre que o método remover um elemento, o nome do mesmo deve iniciar com o verbo `remover`, seguido do nome do elemento, como em `removerNotaFiscal()`, a fim de evitar que alguns utilizem deletar, ou excluir, ou apagar, etc.

Para citar um exemplo de padronização da nomenclatura de classes, podemos definir:

- Todo DAO deve ter seu nome formado pelo nome da entidade em questão, seguido do sufixo `DAO`, por exemplo: `NotaFiscal-DAO`.

Para outros arquivos como, por exemplo, XHTML, podemos dizer que o nome deve:

- Ser composto pelo substantivo que define a funcionalidade, seguido do nome da entidade em questão, por exemplo: `pesquisa-NotaFiscal.xhtml`.

Note que o propósito aqui não é mostrar qual o padrão mais adequado, mas sim mostrar que deve-se ter um padrão de nomen-

Como desenvolver um código fonte de qualidade em Java

clatura. Tendo esses padrões em mente, fica muito fácil encontrar qualquer funcionalidade dentro do código fonte do sistema, aumentando a velocidade de desenvolvimento, além de facilitar a manutenção e o aprendizado.

Design Patterns

Este item é obrigatório! Todo programador deve conhecer os principais padrões de projeto e, principalmente, saber identificar como e em quais situações utilizar cada um. Não adianta saber de cor todos os padrões descritos no GOF e não saber como aplicá-los. Por isso é muito importante que se entenda os conceitos dos padrões.

Um design pattern é uma solução conhecida para um problema conhecido. Se bem compreendido, facilita no desenvolvimento do sistema, pois ao identificar o problema já se sabe a solução mais adequada a utilizar. Portanto, facilita o desenvolvimento, manutenção e a compreensão do código fonte.

Coesão e Acoplamento

É um assunto tão importante quanto esquecido pela maioria dos programadores. São princípios que andam juntos e devem sempre ser buscados: **alta coesão e baixo acoplamento**. Assim conseguimos um código mais claro, de fácil manutenção, menos sensível (diminui a possibilidade de surgir um erro causado por uma alteração em outra classe), mais flexível e, principalmente, mais fácil de ser testado unitariamente, como veremos adiante.

Coesão

Refere-se à responsabilidade da classe ou método. Se não for possível responder em poucas palavras a pergunta “qual a função desta classe?” ou “o que faz este método?”, é porque esta classe ou este método não são muito coesos. É a base do Princípio da Responsabilidade Única, descrita por Robert C. Martin em “The Single Responsibility Principle”, que diz que a classe deve ter como foco uma única responsabilidade. Por exemplo, um DAO deve ser focado em apenas realizar o acesso a dados. Da mesma forma, o método **persistir** deste DAO deve fazer apenas a persistência, sem nenhuma regra de negócio envolvida.

Acoplamento

Refere-se a quanto uma classe depende de outras, ou seja, o quanto a classe está acoplada em outras. Esta dependência ocorre quando uma classe referencia outra, através da execução de algum método seu. Quanto maior a dependência entre as classes, maior o acoplamento. É o princípio da Lei de Demeter, que diz que “Na classe C, para todos os métodos M definidos em C, todos os objetos com o qual M se comunica devem ser ou um argumento de M ou um membro de C”.

Performance e Robustez

Estes dois itens aparentemente só são buscados quando expressamente solicitados. No caso da performance, provavelmente por causa da evolução dos equipamentos como memória e processador-

res, foi, indevidamente, deixada de lado. Já a preocupação com a robustez deve ter algo a ver com a cada vez menor quantidade de profissionais sêniores nas equipes de desenvolvimento, substituídos por profissionais mais jovens, com grande conhecimento de tecnologias, mas com menos experiência. Times de desenvolvimento são como times de futebol: deve haver um equilíbrio nas equipes, mesclando profissionais mais jovens com outros mais experientes.

Performance

A questão da performance vai desde o banco de dados, com tabelas com índices corretos e bem utilizados, até a codificação propriamente dita.

É muito comum vermos consultas mal escritas, que apesar de funcionar (resultado final correto), poderiam ser muito mais rápidas. Em alguns casos, a tabela tem até o índice correto, mas a consulta foi feita de maneira que o banco de dados não consiga utilizar o mesmo (independente de se utilizar algum framework como o Hibernate).

Outro exemplo, também muito ruim, pode ser identificado quando uma consulta é feita de maneira que busque do banco de dados uma quantidade muito grande de registros, que então serão iterados para descartar ou não alguns elementos de acordo com algum dado. Obviamente este filtro feito pela iteração deveria ter sido incluído na consulta inicial.

Outra situação bem comum é a instanciação desnecessária de objetos. Por exemplo, um objeto que será utilizado apenas dentro de um trecho condicional deve ser instanciado apenas dentro do mesmo. Este caso se torna ainda mais grave se estiver dentro de uma iteração, onde essa instanciação desnecessária ocorrerá inúmeras vezes (se pensarmos em um sistema com um número grande de usuários simultâneos, poderemos ter um problema bem grave).

Robustez

É o quanto forte é nosso sistema. O ideal é que nos preocupemos sempre com os possíveis pontos fracos do sistema, os quais muitas vezes não são pegos nos testes, mas acabam aparecendo depois de algum tempo, já com o sistema em produção.

Normalmente estes erros aparecem na forma de exceções não checadas, ou seja, exceções lançadas em tempo de execução e que não obrigam que o programador implemente um tratamento para a mesma; são todas as exceções que herdam de `java.lang.RuntimeException`. Dentre elas, as mais comuns são a `java.lang.NullPointerException` e a `java.lang.IndexOutOfBoundsException` (e sua forma mais específica, `java.lang.ArrayIndexOutOfBoundsException`).

Em muitos casos, estes erros ocorrem por uma culpa dividida entre o analista de sistemas e o programador, onde um faz uma definição incompleta e o outro não o questiona. Vamos exemplificar com alguns trechos de especificações hipotéticas:

- ... execute a consulta C para obter o objeto O e então incremente o valor da propriedade P do objeto O...

- Aqui não foi dito o que fazer no caso de a consulta C não retornar o objeto O, retornando `null` em seu lugar. O programador deve questionar qual sua ação neste caso, a fim de evitar o possível `java.lang.NullPointerException`. É interessante atentar que não basta o programador decidir, por si só, fazer um teste para verificar se o objeto O não está nulo e apenas neste caso incrementar a propriedade P, pois podemos gerar um erro negocial. Deste modo, deveria ser lançada alguma espécie de exceção de negócio, informando que houve determinado erro.

- ... execute a consulta C para buscar os três maiores valores e então multiplique-os pela quantidade...

- Houve uma afirmação de que a consulta C irá retornar três registros, porém não houve a preocupação em especificar o que deve ser feito se a mesma retornar apenas dois registros, por exemplo. Assim como no item anterior, deve-se questionar o autor da regra, pois se for implementado da maneira que solicitado, podemos ter uma `java.lang.IndexOutOfBoundsException`. Da mesma forma, o programador não deve decidir implementar de maneira incremental, aplicando a regra para cada registro encontrado, a fim de evitar a exceção citada, pois pode ser um requisito que sejam encontrados os três registros, devendo então ser lançada uma exceção de negócio caso não os encontre.

Nestes dois exemplos, se a implementação for feita conforme especificado, sem se preocupar com os possíveis casos de exceção, ainda podemos ter o sistema com aparente qualidade, porém, com a possibilidade de que as consultas não retornem os resultados esperados, logo com pontos fracos comprometendo a robustez do nosso sistema.

Logging e documentação de código fonte

São muitas vezes deixados de lado por não afetarem o funcionamento do sistema. Porém, têm outras utilidades imprescindíveis, conforme veremos neste tópico.

Logging

Um bom log pode ser fundamental para encontrar a causa de um bug grave em produção, por exemplo. Devemos, como sempre, tomar cuidado também com os excessos, pois além de poder prejudicar a performance do sistema, um excesso de log pode dificultar inclusive a leitura do mesmo. Faça log, mas faça com qualidade. Faça log que possa ajudar em uma eventual necessidade.

Documentação do código fonte

Extremamente importante para o entendimento e manutenção do código fonte. Quanto tempo você perde para descrever a funcionalidade de determinado método? Talvez um minuto? Agora pense quanto tempo outras pessoas poderão poupar para entender a funcionalidade do mesmo se você já o descreveu. Isso não vale apenas para a definição do método (em forma de Javadoc, por exemplo), vale também para o auxílio no entendimento do

desenvolvimento do mesmo. Assim como dito no caso do logging, faça comentários construtivos.

Vejamos alguns exemplos:

```
valor = valor * 2; // multiplica o valor por 2
```

Este comentário é inútil, pois todos sabem que `valor = valor * 2` significa multiplicar o valor por 2. O interessante é explicar o motivo de tê-lo feito.

```
valor = valor * 2; // dobra o valor, pois para este tipo de produto o valor tem peso 2.
```

Aqui o comentário deixa claro o motivo da multiplicação, facilitando o entendimento.

Testes Unitários

Não vamos falar aqui sobre a fundamental importância dos testes unitários, mas sim de como um código bem feito nos dá todas as ferramentas para construir testes unitários tão bons quanto, com confiabilidade e cobertura maiores.

Teste unitário, como o nome já diz, testa uma unidade de processamento. Mas se o método faz tudo sozinho, como testar? Se o teste falhar, onde foi a falha? O que exatamente falhou?

Então, quanto mais coesos os métodos, mais fácil testarmos nosso sistema. Ou seja, quanto menor a responsabilidade de cada método, mais fácil testarmos e garantirmos o seu funcionamento. Isso acontece porque diminuímos a unidade a ser testada, podendo assim criar testes com mais qualidade e maior cobertura.

Vamos supor que precisemos testar o método `cancelarVenda()`. Se o método fizer todo o trabalho sozinho (como estornar a nota, gerar comprovante e imprimir ou mandar e-mail do mesmo, etc.) o teste além de muito difícil de ser implementado, certamente não seria tão confiável, pois desta maneira é muito difícil cobrir todas as possibilidades, e em caso de falha teríamos um esforço maior para descobrir o que gerou a mesma. Por outro lado, se o método dividir sua responsabilidade, por exemplo, com os métodos `estornarNotaFiscal()`, `gerarComprovanteEstorno()`, `imprimirComprovanteEstorno()` e `enviarEmailEstorno()`, poderemos testar separadamente cada um, garantindo seu funcionamento e, no método principal, `cancelarVenda()`, podemos utilizar um mock desses outros métodos, atribuindo o comportamento que desejarmos. Ou seja, todo o processamento dos métodos `estornarNotaFiscal()`, `gerarComprovanteEstorno()`, `imprimirComprovanteEstorno()` e `enviarEmailEstorno()` estaria de fora do escopo do teste do método `cancelarVenda()`. Testando estes métodos separadamente, podemos ter uma cobertura bem maior.

TDD – Test Driven Development

Amada por uns e odiada por outros, é uma metodologia de desenvolvimento muito interessante, apesar de bastante difícil de ser alcançada, onde desenvolvemos os testes antes da imple-

Como desenvolver um código fonte de qualidade em Java

mentação. Também não discutiremos aqui se você deve ou não utilizar na prática TTD, porém, pense como se estivesse utilizando. Ao implementar algum requisito, se você pensar “como vou testar isto?”, provavelmente terá um código muito mais coeso, com métodos com responsabilidades menores e mais fácil de serem testados, como já discutido.

Validadores de código

Seu uso é muito interessante, principalmente em equipes com programadores mais juniores, pois além de garantir um código melhor, também ajuda no aprimoramento profissional destes programadores. Os validadores podem ser manuais (feito por outro programador) ou automatizados (utilizando ferramentas de análise estática de código, como PMD, CheckStyle e FindBugs).

Estes validadores automatizados avaliam um conjunto de regras definidas e verificam se o código está aderente às mesmas, e em caso de violação de alguma destas regras, gera um aviso e, dependendo da prioridade, pode até falhar a geração do build do sistema.

Code review

Esta é uma prática muito importante, onde um programador mais experiente revisa o código gerado por outro programador, destacando pontos a serem melhorados, explicando os motivos e sugerindo soluções mais adequadas. Esta prática garante um código fonte de qualidade, além de aprimorar o nível tanto do profissional que está tendo seu código revisado como do revisor.

Definição de regras

É muito importante definir e divulgar para toda a equipe quais regras serão utilizadas nos validadores automatizados. Estes possuem várias regras pré-definidas, que podemos deixar ligadas ou desligadas, atribuir prioridades para as mesmas (como uma espécie de grau de gravidade) e definir a partir de qual prioridade a violação da regra deve falhar a geração do build do sistema ou apenas mostrar uma mensagem de aviso que a mesma foi violada.

Além das regras pré-definidas, podemos criar outras regras, o que é muito interessante, pois podemos criar regras de acordo com a necessidade do sistema em desenvolvimento. Com esse mesmo intuito, muitas das regras pré-definidas podem ser parametrizadas. Por exemplo, uma regra que define o número máximo de linhas ou o número máximo de métodos de uma classe tem um valor padrão, mas pode ser alterado de acordo com a necessidade.

Dificuldade de implantação

Quanto mais cedo implantadas e definidas as regras, menor é a dificuldade. Quando o sistema já está sendo codificado, a dificuldade aumenta muito, pois ao definirmos uma regra ela já pode ter várias violações, gerando um número de refactorings muito grande. Portanto, tente definir e implantar o mais cedo possível; quanto antes melhor.

No entanto, a principal dificuldade é a reação dos programadores, principalmente dos menos experientes, diante dos validadores, manuais ou automatizados. Por isso, o ideal é que cada regra seja mostrada e explicada a toda a equipe, deixando claro a todos os motivos das regras, quais os benefícios das mesmas.

É muito comum programadores criarem uma espécie de bloqueio de aceitação das regras, muito provavelmente por não ter recebido instruções suficientes sobre as mesmas, e acabam tentando burlar ao invés de se adaptar a elas. Um exemplo clássico disso é a tentativa de burlar a regra que diz para não termos literais em condicionais (**AvoidLiterals-InIfCondition**), como em `if(tipo == 7) {...}`. O objetivo da regra é, no caso de não termos um parâmetro, criarmos uma constante (**private static final**) com um nome significativo. Apresentamos exemplos dessa regra sendo violada, burlada e respeitada nas **Listagens 1, 2 e 3**, respectivamente.

Listagem 1. Exemplo de regra violada.

```
public class MinhaClasse {  
    public meuMetodo(int tipo){  
        if(tipo == 7){ // aqui a regra é violada  
            facaAlgo();  
        }  
    }  
}
```

Listagem 2. Exemplo de regra burlada.

```
public class MinhaClasse {  
    private static final int VALOR_7 = 7;  
    public meuMetodo(int tipo){  
        if(tipo == VALOR_7){ // aqui a regra não é violada, porém o nome da constante  
            facaAlgo(); // não é significativo, não nos diz do que se trata.  
        }  
    }  
}
```

Listagem 3. Exemplo de regra respeitada.

```
public class MinhaClasse {  
    private static final int TIPO_DEVOLUCAO = 7;  
    public meuMetodo(int tipo){  
        if(tipo == TIPO_DEVOLUCAO){ // deixa claro que o método facaAlgo() será  
            facaAlgo(); // executado quando o tipo for de devolução.  
        }  
    }  
}
```

Apesar da dificuldade inicial, a tendência é que com o tempo os programadores se acostumem com as regras e acabem codificando já pensando nas mesmas, evitando as violações e gerando código de mais qualidade.

Portanto, preocupe-se com a qualidade do código fonte, pois esta irá garantir boa parte da qualidade do software. Para isso, estude conceitos e padrões, além de praticar bastante.

**Conhecimento
faz diferença!**



Agilidade: Negociação de coisas

Agilidade: Acompanhamento de projetos ágeis distribuído através do Daily Meeting

Projeto
Diagrama de sequência na prática

Projeto
Como inserir padrões de projeto através de refatorações – Parte 2

SOA
Processo e levantamento de requisitos de negócios – Parte 2

Qualidade de Software
Definição, características e importância

Automação de Testes

Cuidados a serem tomados na implantação Processo e automação de testes de Software

Aulas desta edição:

- Atividades da Gerência de Projetos – Partes 10 a 14
- Estratégia de Teste Funcional baseada em Casos de Uso – Partes 5 a 9

ISSN 1983127-7

9781983127008 00028

+ de 290 vídeos
para assinantes

Faça já sua assinatura digital! | www.devmedia.com.br/es

Faça um upgrade em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional.

Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software.

Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**

 **DEV MEDIA**

Como desenvolver um código fonte de qualidade em Java

Neste momento vale ressaltar que para cada item abordado neste artigo existe ampla bibliografia disponível, sendo interessante que o leitor procure se aprofundar ainda mais no assunto, discutir a respeito, aplicar no dia a dia e até revisar códigos desenvolvidos anteriormente, verificando o que poderia e deveria ter sido feito de outra forma.

Assim, esperamos ter contribuído para o aprimoramento de programadores, para que tenhamos cada vez mais artesãos de software, gerando códigos fonte de qualidade e bem feitos, melhorando de maneira geral a qualidade dos nossos sistemas.

Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!



Autor



Luis Cesar de Souza Moura

luiscesarmoura@gmail.com

Bacharel em Ciência da Computação pela Universidade Federal do Rio Grande do Sul. Atuo na área de desenvolvimento de software desde 2004, atualmente como Arquiteto Java, posso as certificações SCJP, SCBCD, OCWCD e OCMJEA.



Links:

Manifesto for Software Craftsmanship.

<http://manifesto.softwarecraftsmanship.org>

The Single Responsibility Principle.

<http://www.objectmentor.com/resources/articles/srp.pdf>

CURSOS ONLINE

A Revista Clube Delphi oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.



CONHEÇA ALGUNS DOS CURSOS:

- **Curso de Multicamadas com Delphi e DataSnap**
- **Delphi para Iniciantes**
- **Criando componente Boleto em Delphi**
- **Loja Virtual em Delphi Prism**

Para mais informações :

www.devmedia.com.br/cursos/delphi

(21) 3382-5038



DEVMEDIA

Programação funcional aplicada com Aspectos

Como beneficiar-se de abstrações da programação funcional em um código orientado a aspectos

A partir do Java 8, o desenvolvedor ganhou novas ferramentas para trabalhar de forma diferente. Agora é possível desenvolver códigos orientados à programação funcional de forma mais fácil e tirar proveito do que há de melhor neste estilo: concisão, estética, paralelismo e facilidade de manutenção. Porém, não podemos esquecer que o desenvolvimento em Java pode (e deve) se beneficiar do fato de a linguagem não ser puramente funcional. A vantagem desta característica é a possibilidade de unir à programação orientada a objetos duas formas de pensar na criação de softwares: a programação funcional e a programação orientada a aspectos. Como veremos ao longo deste artigo, cada paradigma se apresenta como melhor solução dependendo da estrutura que se pretenda construir em função do modelo e da escalabilidade.

Hoje o desenvolvedor Java tem em mãos uma linguagem que permite, ao mesmo tempo, escrever códigos que se favoreçam tanto da elegância dos aspectos como forma de separar interesses ortogonais ao domínio da regra de negócio, quanto da eficiência da programação funcional, com a sua imposição como um estilo mais suínto e, ao mesmo, tempo eficaz. Entretanto, deve-se ter cautela ao mesclar estas duas técnicas no mesmo código, pois apesar de terem objetivos diferentes, a utilização de aspectos pode introduzir efeitos colaterais, ou melhor, *side-effects* (ver **BOX 1**) ao código, eliminando um dos maiores benefícios da programação funcional, que é exatamente a eliminação destes efeitos, para que o código torne-se mais facilmente modificável, paralelizável e livre de *bugs*.

Além disso, a utilização indevida de aspectos em códigos funcionais pode quebrar um dos preceitos

Fique por dentro

Os paradigmas de programação orientada a aspectos e de programação funcional já demonstraram ser a evolução no que diz respeito à estruturação de softwares modernos; ambos se apresentam como extensões do raciocínio orientado a objetos e se diferenciam nas suas utilidades, objetivos, benefícios e fraquezas. Devido à diferença de objetivos, em alguns casos, a coesão proporcionada pelos aspectos pode conduzir a problemas de side-effects. Então, como mesclar os dois modelos e tirar proveito do que há de melhor nos dois mundos?

Neste artigo entenderemos as principais características destes dois paradigmas, bem como suas principais diferenças e como utilizá-los em conjunto através de abstrações Monads, que são recursos já bem consolidadas no mundo funcional.

fundamentais da programação funcional, que consiste em funções que retornem sempre o mesmo resultado, uma vez utilizados os mesmos parâmetros independentemente de quantas vezes a função seja executada. Este problema pode ocorrer por causa dos *pointcuts* (pontos de corte) necessários ao processo de interceptação do código no fluxo principal para que outro algoritmo de interesse transversal seja executado. Neste caso, o código transversal pode alterar o valor de parâmetros ou do próprio retorno da função, criando toda a problemática aqui citada. Por outro lado, a não utilização dos aspectos pode resultar em códigos de difícil manutenção e com classes pouco coesivas, o que dificulta o reaproveitamento em outros módulos ou sistemas.

Neste artigo vamos revisitar os conceitos por trás da orientação a aspectos e compará-los com aqueles da programação funcional. Veremos como utilizar ambos os paradigmas para construir um programa ao mesmo tempo livre de *side-effects* e bem modularizado.

Programação funcional aplicada com Aspectos

BOX 1. Side Effect

Chamam-se Side Effects as características de uma função que além de prover um valor de retorno, também modifica o estado de algum atributo ou efetua alguma interação com um ambiente externo (um sistema de arquivos, por exemplo). Esta é uma característica muito comum em linguagens de programação imperativas devido à possibilidade de definir variáveis com escopos globais em relação ao método a ser executado.

Estes efeitos são fortemente criticados e evitados em linguagens de programação que pretendem ser funcionais devido às suas contradições com os preceitos deste paradigma, que prevê funções cujas execuções devem manter a consistência de resultados em função dos parâmetros informados. Neste caso, uma função que efetue operações em escopos externos como variáveis globais, sistemas de arquivos, acesso a serviços, entre outros, se tornará vulnerável a efeitos que não dependerão exclusivamente das variáveis passadas como parâmetros ou definidas em seu escopo, quebrando, portanto, a coerência dos resultados.

Além da coerência das funções, a inexistência de Side Effects em um programa garantirá o benefício de Thread Safety, que possibilitará a paralelização de qualquer ponto da aplicação com pouco ou nenhum esforço, uma vez que sem tais efeitos a concorrência de threads por algum recurso externo será inexistente.

Falando de Objetos

Para que possamos compreender a necessidade da semântica de aspectos sobre o raciocínio orientado a objetos, faz-se necessário revisitarmos os conceitos fundamentais da modelagem orientada a objetos. Suas premissas podem ser resumidas em quatro proposições básicas:

1. Um objeto não deve ter acesso a informações que não façam parte do seu escopo;
2. Todas as mudanças de estado em um objeto devem ocorrer somente através da sua interface;
3. As relações de herança entre as classes devem manter a coerência conceitual;
4. O entrelaçamento entre classes deve ser o menor possível.

Estes preceitos apresentam limitações ao tratar comportamentos que se espalham pelos objetos quando existem outros interesses

além da própria regra de negócio. O exemplo mais clássico sobre esta falha é o da auditoria através de log. Esse requisito está associado ao registro das atividades executadas no software para análises posteriores sobre erros, fraudes, tendências de execução, mineração de dados, entre outros, atravessando toda a regra de negócios de forma sincronizada com a execução para que se obtenha a fidelidade e coerência do histórico. Para exemplificar a falha de modelagem no que diz respeito a interesses transversais entre a regra de negócio e a atividade de log, na **Listagem 1** declaramos uma classe **CarrinhoDeCompra** que possui um método simples de adicionar itens e outro para a configuração de um objeto que será utilizado para o registro de log.

Listagem 1. Declaração da classe CarrinhoDeCompra.

```
01. package br.com.jm.faop.business;
02.
03. import java.util.ArrayList;
04. import java.util.List;
05. import java.util.logging.Logger;
06. import br.com.jm.faop.business.Item;
07.
08. public class CarrinhoDeCompra {
09.
10.     private List<Item> items = new ArrayList<>();
11.     private Logger logger;
12.
13.     public void addItem(Item item) {
14.         this.items.add(item);
15.         this.logger.info(String.format("Item [%s] adicionado ao carrinho de compra.", item.getName()));
16.     }
17.
18.     public void setLogger(Logger logger) {
19.         this.logger = logger;
20.     }
21.
22.     public List<Item> getItems() {
23.         return items;
24.     }
25. }
```



Podemos ver neste código a declaração do método `setLogger(Logger logger)`, que é utilizado para atribuir um `Logger` ao carrinho de compras. Este `Logger` é utilizado no método `addItem(Item item)` para informar no log que o `Item` (**Listagem 2**) foi adicionado com sucesso. Apesar da importância dos logs em aplicações mais complexas, neste exemplo percebemos de forma clara que conceitualmente um carrinho de compras não tem relação alguma com um `Logger`, mas também sabemos que sistematicamente os logs são interesses quase sempre presentes e necessários como requisitos não funcionais. Portanto, neste caso estamos quebrando a coesão (**BOX 2**) da classe `CarrinhoDeCompra`, que agora possui mais responsabilidades do que as relacionadas às compras. Outrossim, a classe `Logger` agora está diretamente acoplada à classe `CarrinhoDeCompra`, maximizando o entrelaçamento entre elas. Na classe `CarrinhoDeCompraTest`, apresentada na **Listagem 3**, podemos ver o esforço de criação e configuração do `Logger` no `CarrinhoDeCompra` para que o teste funcione corretamente.

BOX 2. Coesão

É a medida de distribuição da lógica de um sistema entre seus componentes. Quanto mais especializados e logicamente relacionados os componentes estiverem, mais coeso o sistema será. Os objetos devem ser coesos pelas seguintes razões: quanto mais baixa a coesão, mais eles serão suscetíveis a alterações (manutenção) e a bugs pela quantidade de responsabilidades que assumem. Além disso, objetos com baixa coesão são raramente fáceis de serem reutilizados.

Listagem 2. Declaração da classe Item.

```

01. package br.com.jm.faop.business;
02.
03. public class Item {
04.     private String name;
05.
06.     public Item(String name) {
07.         this.name = name;
08.     }
09.
10.    public String getName() {
11.        return this.name;
12.    }
13. }
```

Listagem 3. Declaração da classe CarrinhoDeCompraTest.

```

01. package br.com.jm.faop.business;
02.
03. import java.util.logging.Logger;
04. import org.junit.Assert;
05. import org.junit.Test;
06. import br.com.jm.faop.business.CarrinhoDeCompra;
07.
08. public class CarrinhoDeCompraTest {
09.
10.    public @Test void testAddItem() {
11.        CarrinhoDeCompra carrinho = new CarrinhoDeCompra();
12.        carrinho.setLogger(Logger.getLogger(carrinho.getClass().getName()));
13.        Item item = new Item("Spaghetti");
14.        carrinho.addItem(item);
15.        Assert.assertTrue("O item \"Spaghetti\" deve estar contido no carrinho de compras", carrinho.getItems().contains(item));
16.    }
17. }
```

Na linha 12 o `Logger` deve ser configurado para que não haja uma exceção (`NullPointerException`) durante a execução do método `addItem(Item item)`. Nota-se que a execução do log não é algo importante para o teste, mas que é algo necessário para que não haja a exceção.

Outro problema da modelagem apresentada aqui é o que chamamos de código espalhado (*scattering* – **BOX 3**). Isso ocorre em regras mal definidas que se espalham pelos objetos do modelo, dificultando a manutenção de tais regras e a reutilização em outros sistemas. Geralmente, regras espalhadas conduzem facilmente a outro efeito negativo, que é o emaranhamento (*tangling* – **BOX 4**), mais conhecido como código espaguete, resultado da implementação de interesses ortogonais em um mesmo objeto. Neste exemplo do `CarrinhoDeCompra`, o interesse de auditoria está se entrelaçando com o interesse do negócio principal.

BOX 3. Código espalhado (scattering)

Ocorre quando um interesse adicional ao domínio precisa ser implementado em diversos trechos entre várias classes. Desta forma o código de interesse adicional precisa ser espalhado nas diversas implementações. A Orientação a Aspectos resolve esse problema através da criação de pontos de corte (*pointcuts*) que permitem a inserção do código sem espalhá-lo pelas implementações.

BOX 4. Código emaranhado (tangled)

Ocorre quando um ou mais interesses sistemáticos precisam ser coordenados juntos à implementação do domínio no mesmo módulo, quebrando a ideia de coesão da orientação a objetos. A orientação a aspectos resolve esse problema através da criação de pontos de corte para a inserção do código de interesse transversal dentro do contexto de domínio da solução.

É importante entendermos que a programação funcional, tal como possibilitada pelo Java, não resolve estas limitações de interesses transversais presentes na orientação a objetos, por se tratarem de regras dependentes do mesmo fluxo de execução, apesar de semanticamente paralelas. Até a versão 8, o Java não possuía uma forma nativa para paralelizar interesses diferentes no mesmo fluxo e eliminar a escrita de código espaguete. Em linguagens puramente funcionais, como Haskell, por exemplo, é possível imitar tal comportamento através dos *Monads*, que são implementados nativamente na linguagem. Porém, ainda assim, nessas linguagens este recurso se mostra limitado quando há necessidade de mesclar muitos interesses em um único fluxo, como log e tratamento de erros além da própria regra de negócio.

Em Java, podemos alcançar esta separação de interesses sistemáticos através da criação de aspectos que permitam introduzir tais regras de forma paralela no mesmo fluxo, eliminando o acoplamento (**BOX 5**) indevido e evitando o espalhamento de código, mantendo a coesão dos objetos envolvidos. Estes benefícios podem ser conferidos na **Listagem 4**, em que temos a declaração do `CarrinhoDeCompra` visto no exemplo anterior. Note que a classe está totalmente livre de implementações transversais, tornando-se coesa com o modelo de negócio que implementa.

Programação funcional aplicada com Aspectos

BOX 5. Acoplamento

É a relação de força das conexões entre os objetos de uma solução. Quanto mais um objeto tem dependência do outro, mais forte se torna o acoplamento entre eles. O acoplamento deve ser evitado ao máximo a fim de minimizar o impacto da alteração de uma classe nas outras durante a manutenção.

Listagem 4. Nova declaração da classe CarrinhoDeCompra.

```
01. package br.com.jm.faop.business;
02.
03. import java.util.ArrayList;
04. import java.util.List;
05. import br.com.jm.faop.business.Item;
06.
07. public class CarrinhoDeCompra {
08.
09.     private List<Item> items = new ArrayList<>();
10.
11.    public void addItem(Item item) {
12.        this.items.add(item);
13.    }
14.
15.    public List<Item> getItems() {
16.        return items;
17.    }
18.}
```

Já na **Listagem 5** criamos uma nova classe **AuditoriaCarrinhoDeCompraAspect**, a ser utilizada para representar o aspecto que realizará o log através da interceptação do método de adição de um item ao carrinho de compra.

Neste exemplo, o aspecto é criado com o auxílio da API AspectJ. Através desta API temos um novo conjunto de palavras reservadas e de notações para que possamos declarar os aspectos que mudarão ou adicionarão comportamentos novos ao sistema.

Listagem 5. Declaração do aspecto AuditoriaCarrinhoDeCompraAspect.

```
01. package br.com.jm.faop.business;
02.
03. import java.util.logging.Logger;
04. import br.com.jm.faop.business.Item;
05.
06. public aspect AuditoriaCarrinhoDeCompraAspect {
07.
08.     after(Item itm) returning(): execution(public * br.com.jm.faop.business.CarrinhoDeCompra.addItem(Item)) && args(itm) {
09.         this.logger.info(String.format("Item [%s] adicionado ao carrinho de compra.", itm.getName()));
10.    }
11.
12.    private Logger logger;
13.
14.    {
15.        setLogger(Logger.getLogger(AuditoriaCarrinhoDeCompraAspect.class.getName()));
16.    }
17.
18.    public void setLogger(Logger logger) {
19.        this.logger = logger;
20.    }
21.}
```

Listagem 6. Declaração plugin aspectj-maven-plugin.

```
01. <project xmlns="http://maven.apache.org/POM/4.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                               http://maven.apache.org/xsd/maven-4.0.0.xsd">
02. ...
03. <build>
04. ...
05. <plugins>
06. ...
07. <plugin>
08.     <groupId>org.codehaus.mojo</groupId>
09.     <artifactId>aspectj-maven-plugin</artifactId>
10.     <version>1.7</version>
11.     <configuration>
12.         <testAspectDirectory>src/main/aspect</testAspectDirectory>
13.         <aspectDirectory>src/main/aspect</aspectDirectory>
14.         <sources>
15.             <source>
16.                 <basedir>src/main/aspect</basedir>
17.             </source>
18.         </sources>
19.         <complianceLevel>1.8</complianceLevel>
20.         <source>1.8</source>
21.         <target>1.8</target>
22.     </configuration>
23.     <executions>
24.         <execution>
25.             <id>weave-jar-dependencies</id>
26.             <goals>
27.                 <goal>compile</goal>
28.                 <goal>test-compile</goal>
29.             </goals>
30.         </execution>
31.     </executions>
32.     </plugin>
33. ...
34. </plugins>
35. ...
36. </build>
37. ...
38.</project>
```

A execução do aspecto ocorrerá após o retorno do método **addItem(Item item)**. A API também fornece o plugin Maven declarado na **Listagem 6** para a realização do *weaving* (ver **BOX 6**) necessário à interpretação do código nativo e à união do aspecto ao fluxo de execução em que ele será inserido. A dependência da API é declarada na **Listagem 7**.

Ao declarar o plugin estamos configurando o próprio projeto do exemplo (aqui chamado de **functional-aspects**) como uma dependência para o processo de *weaving*.

Com esta abordagem, o teste se torna muito mais prático e objetivo, livre dos efeitos visíveis causados pela intrusão do **Logger**, que deixa de ser uma preocupação diretamente relacionada ao **CarrinhoDeCompra**. Esta diferença pode ser verificada na **Listagem 8**.

O dilema entre coesão e side-effects

No exemplo das **Listagens 4** e **5** resolvemos o problema de interesses ortogonais e mantivemos a coesão tanto do **CarrinhoDeCompra** quanto da tarefa de *logging*, que fica isolada em um

aspecto específico para a finalidade de auditoria; assim, ambos, classe e aspecto, mantêm-se coerentes com suas responsabilidades. Porém, neste exemplo estamos viabilizando o problema dos *side-effects*. O que aconteceria se durante a execução da linha 9 do aspecto **AuditoriaCarrinhoDeCompraAspect** o disco em que o **Logger** escreve ficasse cheio e tivéssemos uma exceção? Esta exceção causaria a quebra no fluxo de execução como um todo, inclusive na execução do método **addItem(Item item)** da classe **CarrinhoDeCompra**, já que ele é o código interceptado pelo interesse transversal.

BOX 6. Weaver de Aspectos

Weavers são ferramentas de meta-programação utilizados por linguagens orientadas a aspectos para inserir instruções especificadas pelos códigos de aspectos a fim de gerar uma implementação final que une tarefas transversais em um mesmo fluxo de execução.

Os Weavers utilizam-se de pointcuts e joinpoints para decidir quais métodos devem ser interceptados pelo código de aspecto conhecido como advice. Sua implementação então decide em que momento as instruções devem ser executadas; esses momentos podem ser: antes, depois ou ao redor do código interceptado. No Java ainda existem momentos adicionais como: depois do retorno e depois de uma exceção.

Como o Java não é uma linguagem nativamente orientada a aspectos, este processo de weaving é feito através de APIs como AspectJ ou Spring AOP, que realizam a tarefa do weaver diretamente nos bytecode de forma estática, durante a compilação da classe ou de forma dinâmica, durante a execução da classe.

Sabemos que o desenvolvedor do aspecto pode tratar o erro e não deixá-lo propagar-se no fluxo da lógica principal, mas esta possibilidade não é a ideal porque depende do desenvolvedor lembrar de tais tratamentos. Além disso, nada impede que durante a escrita do aspecto o desenvolvedor decida alterar o valor de algum parâmetro ou o retorno do método interceptado, quebrando outro fundamento da programação funcional, que consiste na fidelidade das funções; elas devem manter o mesmo retorno para todas as execuções realizadas com os mesmos parâmetros.

Listagem 7. Declaração da dependência aspectjrt.

```
01. <project xmlns="http://maven.apache.org/POM/4.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
           http://maven.apache.org/xsd/maven-4.0.0.xsd">
02. ...
03. <dependencies>
04. ...
05.   <dependency>
06.     <groupId>org.aspectj</groupId>
07.     <artifactId>aspectjrt</artifactId>
08.     <version>1.8.2</version>
09.   </dependency>
10. </dependencies>
11. ...
12. </project>
```

Listagem 8. Nova declaração da classe CarrinhoDeCompraTest.

```
01. package br.com.jm.faop.business;
02.
03. import org.junit.Test;
04. import br.com.jm.faop.business.CarrinhoDeCompra;
05.
06. public class CarrinhoDeCompraTest {
07.
08.   public @Test void testAddItem() {
09.     CarrinhoDeCompra carrinho = new CarrinhoDeCompra();
10.     Item item = new Item("Kindergarten");
11.     carrinho.addItem(item);
12.     Assert.assertTrue("O item \"Kindergarten\" deve estar contido no carrinho de compras", carrinho.getItems().contains(item));
13.   }
14. }
```

Com a alteração de um parâmetro ou de um valor de retorno em um aspecto, esta característica é perdida. Tal dilema é o que torna claro o conflito entre os paradigmas de orientação a aspectos e programação funcional no Java. Mas existirá uma forma de inserir instruções transversais ao fluxo principal de forma não



Programação funcional aplicada com Aspectos

intrusiva e, principalmente, que possibilite a construção de forma mais sistemática em vez de algo aleatório como um critério do desenvolvedor? A resposta para esta pergunta é sim! E a solução para isso são os *Monads*.

Entendendo Monads

Monads são encadeamentos de funções que servem como container de valores a serem processados sequencialmente, permitindo-os – tanto os valores quanto as funções – serem computados de forma isolada. Em outras palavras, Monads são armazenadores de dados que permitem que funções tenham suas execuções encadeadas em torno destes dados. Além disso, os Monads se acoplam uns aos outros a fim de desencadearem execuções independentes de tipo, de forma sequencial e isolada. Podemos identificar algumas características chaves nesta definição: contenção, encadeamento e isolamento, cuja implementação torna os Monads um recurso muito poderoso, como será demonstrado em seguida.

A criação de um Monad contempla dois métodos simples, geralmente chamados de **unit()** (ou **return**) e **flatMap()** (ou **bind**) em linguagens funcionais. Estes métodos devem possuir algumas propriedades algébricas que serão explicadas mais à frente e são estes métodos os responsáveis por garantir duas das características citadas: contenção e encadeamento. O método **unit()** é responsável por fornecer a propriedade de contenção, por ser um método que recebe um tipo **A** e retorna um tipo **M<A>**, que atuará como container do valor parametrizado. Um exemplo mais concreto para o método **unit()** pode ser visto na declaração a seguir:

```
1. Monad<Integer> monadOfInteger = Monad.unit(2);
```

O método **unit()** por si não representa um grande benefício além de criar um container para um valor, pois é o método **flatMap()** que traz a propriedade de encadeamento ao Monad. Este método deve receber uma variável de tipo **B** (eventualmente **B** e **A** podem ser o mesmo tipo) e retornar um tipo **M**, porém, diferentemente, o método **flatMap()** realiza essa transformação não diretamente no tipo **B**, mas sim através da aplicação de uma função que saiba transformar um tipo **A** em um tipo **M** (que por sua vez é quem sabe transformar **B** em **M**). Esta ideia fica mais clara através do exemplo na **Listagem 9**.

Neste código estamos encapsulando o **Monad<Integer>** do exemplo anterior em um **Monad<Double>**, cuja função é obter a raiz quadrada do valor de tipo **Integer** que será parametrizado em seu **flatMap()** no momento da computação de todo o encadeamento de Monads. O resultado de sua execução será o valor **5.00**, raiz quadrada do valor **25** recebido no **flatMap()** durante a transformação dos Monads. Agora temos uma visão mais clara dos métodos com as propriedades de contenção e encadeamento. Suas implementações podem ser conferidas na classe **Monad<A>**, declarada na **Listagem 10**.

Note que nesta classe utilizamos a interface funcional **java.util.function.Function<T,R>**, presente na API do Java 8.

O método **get()** declarado será utilizado na obtenção do valor no final da computação.

Neste código estamos criando uma perspectiva totalmente nova em termos de estrutura e reuso: uma vez que esta abstração nunca muda, a sua utilização permite que a lógica de negócio seja expressa em termos de funções cujo fluxo sequencial é realizado através da execução sucessiva de cada Monad mapeado no encadeamento. Logo veremos o benefício desta abordagem para eliminarmos o nosso dilema entre coesão e side-effects, em outro exemplo.

Listagem 9. Encadeamento do **Monad<Integer>** no **Monad<Double>**.

```
01. Monad<Integer> monadOfInteger = Monad.unit(25);
02. Monad<Double> monadOfDouble = monadOfInteger.flatMap(value ->
    Monad.unit(Math.sqrt(value)));
03. double result = monadOfDouble.get();
04. System.out.println(result);
```

Listagem 10. Declaração da classe **Monad<A>**.

```
01. package br.com.jm.faop.monad;
02.
03. import java.util.function.Function;
04.
05. public class Monad<A> {
06.     private A value;
07.
08.     private Monad(A value) {
09.         this.value = value;
10.     }
11.
12.     public static final <A> Monad<A> unit(A value) {
13.         return new Monad<A>(value);
14.     }
15.
16.     public final <B> Monad<B> flatMap(Function<A, Monad<B>> function) {
17.         return function.apply(value);
18.     }
19.
20.     public A get() {
21.         return value;
22.     }
23. }
```

Até agora falamos sobre duas das palavras-chave para a compreensão dos Monads: contenção e encadeamento, mas ainda está faltando a ideia essencial para garantir a consistência de um encadeamento: o isolamento. Na verdade, isolamento é uma tradução livre para o inglês *confinement*, cujos estudos acadêmicos estão voltados para o cálculo efetivo de isolamento dos *side-effects* em execuções de interesses transversais. Não está no escopo deste artigo o aprofundamento da matemática por trás do isolamento, mas, intuitivamente, o leitor deve ter percebido que as suas aplicações têm muito em comum com o dilema coesão x *side-effects* que estamos tratando aqui. Portanto, a sua compreensão em um nível mais relacionado à aplicação dos Monads em Java torna-se indispensável.

Podemos definir isolamento como as regras necessárias para que o encadeamento entre os Monads fluia continuamente até o final do processamento, garantindo que a execução não seja interrompida durante a computação das funções encapsuladas. O comportamento de continuidade é garantido devido às propriedades algébricas previstas pelas regras que podemos elencar como:

- Identidade à esquerda (*left identity*): Ao aplicarmos um `unit()` a um valor `A` e em seguida um `flatMap()` em uma função `F<A>` que retorne uma função `M`, o seu resultado deve ser igual à aplicação do valor `A` na própria função `F<A>`, de acordo com as regras de igualdade do Java. Exemplo: `Monad.unit(2).flatMap(val -> Monad.unit(Math.sqrt(val))).equals(Monad.unit(Math.sqrt(2)))` deve retornar `true`;
- Identidade à direita (*right identity*): Se encapsularmos um valor em um `Monad` e aplicarmos a ele um `flatMap()` cuja função seja apenas retornar o valor encapsulado, então o retorno do `flatMap()` deve ser o próprio `Monad` com o valor encapsulado, de acordo com as regras de igualdade do Java. Exemplo: `Monad.unit(2).flatMap(val -> Monad.unit(2)).equals(Monad.unit(2))` deve retornar `true`;
- Associatividade (*associativity*): Em um encadeamento de funções `Monad`, a aplicação sucessiva de `flatMap()` não deve sofrer alterações independentemente de como o encadeamento seja composto. Exemplo: `Monad.unit(2).flatMap(val -> Monad.unit(val*3)).flatMap(val -> Monad.unit(val*5)).equals(Monad.unit(2).flatMap(val -> Monad.unit(val*3).flatMap(otherVal -> Monad.unit(otherVal*5))))` deve retornar `true`.

A partir destas regras, podemos então alterar alguns detalhes da nossa classe `Monad<A>` para que ela esteja de acordo com as regras aqui mencionadas. Estas alterações podem ser conferidas na [Listagem 11](#).

Nessa listagem, a única alteração necessária é a implementação dos métodos `equals(Object obj)` e `hashCode()`. Eles são responsáveis por garantir a igualdade de Monads que estejam aloados em

endereços de memória diferentes, mas que sejam semanticamente iguais por conterem o mesmo objeto encapsulado.

Agora que entendemos as regras essenciais para Monads implementados de forma pura, ou seja, seguindo os três preceitos algébricos que regulam a sua implementação, é preciso dizer que tais regras não preveem um problema que ocorre no contexto da programação por se tratar de um problema inexistente no campo matemático: as exceções.

Imagine uma cadeia de funções para executar um cálculo como no exemplo: `Monad.unit(2).flatMap(i -> Monad.unit(i-2).flatMap(j -> Monad.unit(10/j))).flatMap(i -> Monad.unit(Integer.class.cast(i) * 10)).get()`. Sabemos que o trecho `Monad.unit(10/j)` resultará em uma exceção `java.lang.ArithmetricException` por causa da divisão por `j`, que no momento da execução do `flatMap()` terá o valor `0` e não realizará a computação do restante do encadeamento. Neste caso é preferível que seja assim, caso contrário o resultado do cálculo ficará incorreto. Mas qual seria o fluxo esperado em um exemplo como: `Monad.unit(new ClientInfo("cliente@email.com", "(99) 99999-9999", "Seu saldo é insuficiente, deseja recarregar?")).flatMap(EmailSender::sendMail).flatMap(SMSender::sendSMS)`? Aqui estamos criando a hipótese de um fluxo em que o saldo de uma conta (de celular, por exemplo) se acaba e o fornecedor do serviço deseja comunicar ao cliente. Para isso, é iniciado um encadeamento em que o objeto do tipo `EmailSender` receberá a instância de `ClientInfo` e terá a oportunidade de enviar um e-mail ao cliente através do método estático `sendMail(ClientInfo info)`. Em seguida, o fluxo passa ao objeto do tipo `SMSender`, que por sua vez terá a oportunidade de enviar uma mensagem de SMS ao cliente através do método estático `sendSMS(ClientInfo info)`, utilizando as mesmas informações do cliente. Neste caso, se a tentativa de enviar e-mail falhar por causa de uma exceção, o que deverá acontecer depois? O fluxo deve parar na exceção ou o `SMSender` deveria ter a oportunidade de enviar o SMS mesmo assim?



Programação funcional aplicada com Aspectos

Listagem 11. Declaração da classe `Monad<A>` com as regras de isolamento aplicadas.

```
01. package br.com.jm.faop.monad;
02.
03. import java.util.Objects;
04. import java.util.function.Function;
05.
06. public class Monad<A> {
07.     private A value;
08.
09.     private Monad(A value) {
10.         this.value = value;
11.     }
12.
13.     public static final <A> Monad<A> unit(A value) {
14.         return new Monad<A>(value);
15.     }
16.
17.     public final <B> Monad<B> flatMap(Function<A, Monad<B>> function) {
18.         return function.apply(value);
19.     }
20.
21.     public A get() {
```



```
22.         return value;
23.     }
24.
25.     @Override
26.     public int hashCode() {
27.         return Objects.hashCode(value);
28.     }
29.
30.     @Override
31.     public boolean equals(Object obj) {
32.         if (this == obj)
33.             return true;
34.         if (obj == null)
35.             return false;
36.         if (getClass() != obj.getClass())
37.             return false;
38.         Monad<?> other = (Monad<?>) obj;
39.         return Objects.equals(value, other.value);
40.     }
41. }
```

Fica claro que esta decisão cabe à regra de negócio que o sistema descreve, mas o que vale a pena observarmos é que no caso em que o melhor seja seguir o fluxo e permitir a computação das operações posteriores, estaremos quebrando a regra de identidade à esquerda do Monad, pois se a aplicação de uma função $F<A, M>$ a um valor “a” resulta em uma exceção, então pela regra da identidade à esquerda, a aplicação da mesma função em um Monad $M<A>$ deve ter o mesmo retorno, ou seja, a exceção. Por isso, falam-se sobre Monads puros – que implementam fielmente as três regras básicas – e Monads impuros – que são implementações cujos interesses de negócio entram em conflito com uma das regras. Veremos, em seguida, esta implementação de Monads impuros sendo aplicada para solucionar o conflito coesão x side-effects que estamos abordando.

Aplicação de Monads impuros em aspectos

É possível tirar proveito dos Monads através de uma implementação impura que possibilite à função encadeada especificar se suas exceções devem ser desconsideradas durante a execução do Monad. No exemplo do `CarrinhoDeCompra`, será preciso implementar tal comportamento para que o aspecto não interfira no fluxo de execução principal, porém, não queremos que o comportamento do método `addItem(Item item)` seja alterado. Pelo contrário, queremos que as exceções e quaisquer outros possíveis problemas de execução continuem sendo considerados para garantirmos que nada se altere em seu comportamento. Portanto, podemos criar um novo método `inpureFlatMap(SilentFunction<A, Monad> function)` na classe `Monad` que forneça uma execução livre de falhas. Assim, a sua utilização no aspecto garantirá que o código continue ao



RENOVE JÁ!

Sua assinatura pode estar acabando



Renovando a assinatura
de sua revista favorita
você ganha brindes
e descontos exclusivos.

Faça a renovação de sua assinatura agora mesmo.



www.devmedia.com.br/renovacao ou (21)3382-5038

 DEV MEDIA

Programação funcional aplicada com Aspectos

mesmo tempo coeso e orientado ao paradigma funcional, sem a necessidade de alterar o método `flatMap()`. Na **Listagem 12** podemos verificar a classe `Monad<A>` já com a alteração aqui mencionada.

No método `inpureFlatMap(SilentFunction<T, R> function)` percebemos a nova interface `SilentFunction<T, R>` (**Listagem 13**) sendo parametrizada. Esta interface estende a interface `Function<T, R>` a fim de fornecer um novo método *defender* (**BOX 7**) `handleException(Throwable t)` para que o Monad

BOX 7. Métodos defenders

No Java 8 foi introduzido para a criação de interfaces. Estes métodos podem ser criados com o comportamento padrão a ser adotado pelo método caso ele não seja declarado pela classe concreta que implemente a interface. Para todos os efeitos, os métodos declarados como defenders fazem parte do contrato a ser seguido pela classe que os implementa como qualquer outro método declarado na interface, porém a declaração do comportamento padrão pode garantir que uma interface com mais de um método ainda seja considerada funcional, caso apenas um dos métodos não seja defender.

Listagem 12. Declaração da classe `Monad<A>` impura.

```
1. package br.com.jm.faop.monad;
2.
3. import java.util.Objects;
4. import java.util.function.Function;
5. import br.com.jm.faop.function.SilentFunction;
6.
7. public class Monad<A> {
8.     private A value;
9.
10.    private Monad(A value) {
11.        this.value = value;
12.    }
13.
14.    public static <A> Monad<A> unit(A value) {
15.        return new Monad<A>(value);
16.    }
17.
18.    public <B> Monad<B> flatMap(Function<A, Monad<B>> function) {
19.        return function.apply(value);
20.    }
21.
22.    public <B> Monad<A> inpureFlatMap(SilentFunction<A, Monad<B>> function) {
23.        try {
24.            return function.apply(value);
25.        } catch (Throwable e) {
26.            try {
27.                function.handleException(e);
28.            } catch(Throwable t) {
29.                t.printStackTrace();
30.            }
31.            return this;
32.        }
33.    }
34.
35.    public A get() {
36.        return value;
37.    }
38.
39.    @Override
40.    public int hashCode() {
41.        return Objects.hashCode(value);
42.    }
43.
44.    @Override
45.    public boolean equals(Object obj) {
46.        if (this == obj)
47.            return true;
48.        if (obj == null)
49.            return false;
50.        if (getClass() != obj.getClass())
51.            return false;
52.        Monad<?> other = (Monad<?>) obj;
53.        return Objects.equals(value, other.value);
54.    }
55.}
```



tenha a oportunidade de redirecionar o tratamento do erro à interface antes de, finalmente, imprimir o erro na saída do programa, através da chamada `t.printStackTrace()`. O `defender handleException(Throwable t)` faz basicamente o mesmo trabalho de imprimir a pilha de exceções, porém, o usuário da interface terá a oportunidade de implementar o seu método e tomar outras decisões baseadas no tipo da exceção que for parametrizada. É claro que, nos casos em que a execução do método `handleException(Throwable t)` retorne outro erro, o bloco `catch` também evitará que o erro se propague pelo fluxo principal, de fora do aspecto.

Listagem 13. Declaração da interface `SilentFunction<T, R>`.

```
01. package br.com.jm.faop.function;
02.
03. import java.util.function.Function;
04.
05. public interface SilentFunction<T, R> extends Function<T, R> {
06.
07.     default public void handleException(Throwable t) {
08.         t.printStackTrace();
09.     }
10. }
```

Outro detalhe muito importante sobre o novo método `inpureFlatMap()` é que diferentemente do método `flatMap()`, a função recebida deve transformar um tipo A em um tipo `Monad<A>` e retorná-lo como resultado da função, diferentemente da transformação em `flatMap()`, que dá a possibilidade para o retorno de um `Monad` de outro tipo. Com esta modificação temos a garantia de que o aspecto que interceptará a execução não terá a possibilidade de modificar o valor de retorno e tampouco o parâmetro de entrada na execução principal, a não ser que o desenvolvedor intencionalmente force a modificação.

Portanto, com esta nova implementação é possível alterar o código de teste para que ele se torne mais funcional e permita a

utilização de `Monads` para flexibilizar a introdução do *advice* que realizará a tarefa de log. A **Listagem 14** define a implementação final da classe de teste `CarrinhoDeCompraTest`.

Listagem 14. Declaração final da classe `CarrinhoDeCompraTest`.

```
01. package br.com.jm.faop.business;
02.
03. import org.junit.Assert;
04. import org.junit.Test;
05. import br.com.jm.faop.business.CarrinhoDeCompra;
06. import br.com.jm.faop.monad.Monad;
07.
08. public class CarrinhoDeCompraTest {
09.
10.     public @Test void testAddItem() {
11.         CarrinhoDeCompra carrinho = new CarrinhoDeCompra();
12.         Item item = new Item("Suco de Laranja");
13.         Assert.assertEquals(item,
14.             Monad.unit(item)
15.                 .flatMap(itm -> {
16.                     carrinho.addItem(itm);
17.                     return Monad.unit(itm);
18.                 })
19.                 .get()
20.         );
21.     }
22. }
```

A partir desta nova implementação podemos modificar o *pointcut* do aspecto para que ele intercepte não mais a execução de `carrinho.addItem(itm)`, mas sim a de `.flatMap(itm -> {...})`. Esta mudança já representa um passo importante na tarefa de manter-se dentro do paradigma funcional sem perder a coesão dos aspectos, pois a partir de agora interceptaremos não mais a tarefa de adicionar um item ao carrinho especificamente, mas sim o ponto logo após a sua execução no encadeamento de `Monads`. Portanto, o único objeto ao qual teremos acesso será um `Monad<Item>`, que



Programação funcional aplicada com Aspectos

nos permitirá efetuar mais um `flatMap()` (desta vez o impuro) para inserir o comportamento transversal. Podemos conferir a definição final da classe `AuditoriaCarrinhoDeCompraAspect` na [Listagem 15](#).

Neste código os *pointcuts* exercem papel importante ao determinar quando o método `flatMap()` deve ser interceptado – apenas quando o seu argumento for uma `Function<Item>`, `Monad<Item>>` – e em que ponto o código deve ser executado – após o retorno. O *pointcut* `!withinThis()` evitara qualquer possibilidade de recursividade que possa conduzir a loops infinitos.

Agora temos uma implementação que considera tanto as características da programação funcional quanto as características da programação orientada a aspectos, promovendo coesão entre as classes envolvidas, separando os interesses sistêmicos, livrando-se de side-effects e mantendo-se conciso. Repare que, após a nova implementação na [Listagem 15](#), os fluxos estão separados de tal forma que o resultado da chamada ao método `inpureFlatMap()` nem mesmo será considerado no retorno do método interceptado, mas se este mesmo código fosse realizado em um *advice around()*, o retorno do tipo `SilentFunction<A, Monad<A>>` garantiria a continuidade do fluxo sem problemas de conflitos de tipos e o bloco `try/catch` do método evitaria a propagação do erro pelo fluxo principal.

Monads nativos no Java 8

Até este ponto do artigo, temos criado nossos próprios Monads para demonstrar a sua força junto aos aspectos. Porém, nem sempre será necessário criar uma implementação nova de Monads em todos os projetos; no Java 8 os Monads estão abstraídos em classes como `java.util.Optional<T>` e `java.util.stream.Stream<T>`, cujo funcionamento é muito parecido com o da nossa classe `Monad<A>` implementada neste artigo. O uso destes Monads é extremamente encorajado não somente

Listagem 15. Declaração final do aspecto `AuditoriaCarrinhoDeCompraAspect`.

```
01. package br.com.jm.faop.business;
02.
03. import java.util.function.Function;
04. import java.util.logging.Logger;
05. import br.com.jm.faop.business.Item;
06. import br.com.jm.faop.monad.Monad;
07.
08. public aspect AuditoriaCarrinhoDeCompraAspect {
09.
10.     protected pointcut withinThis(): cflow(within(AuditoriaCarrinhoDeCompra
11.         Aspect));
12.
13.     protected pointcut executionOfflatMap(): execution(public * br.com.jm.faop.
14.         monad.Monad.flatMap(*));
15.     after() returning(Monad<?> result) : executionOfflatMap() && !withinThis(){
16.         if(result.getClass().isAssignableFrom(Item.class)) {
17.             @SuppressWarnings("unchecked") <Item> mItem = (Monad<Item>)
18.                 result;
19.             mItem.inpureFlatMap(item -> {
20.                 this.logger.info(String.format("Item [%s] adicionado ao carrinho de
21.                     compra.", item.getName()));
22.                 return Monad.unit(item);
23.             });
24.         }
25.     }
26.     private Logger logger;
27.     {
28.         setLogger(Logger.getLogger(AuditoriaCarrinhoDeCompraAspect.class.
29.             getName()));
30.     }
31.     public void setLogger(Logger logger) {
32.         this.logger = logger;
33.     }
34. }
```



no contexto de utilização em aspectos, mas aonde for possível. Com a sua utilização, o código tenderá a se tornar muito mais funcional e livre de problemas como exceções de `NullPointerException`, dificuldades de paralelização e side-effects. A manutenção de códigos que utilizam estas implementações tende a ser mais fácil e menos intrusiva pela flexibilidade de encadeamento dos métodos `flatMap()`.

Por se tratarem de Monads puros, estas implementações não poderiam ser aplicadas da forma idealizada neste artigo para a utilização em aspectos, por isso é importante que o desenvolvedor entenda a abstração e esteja preparado para criar suas próprias implementações conforme elas se tornem necessárias em casos em que os Monads nativos do Java 8 não atendam às necessidades regidas pelos requisitos do sistema a ser desenvolvido.

Objetivo de cada paradigma

É importante que tenhamos a compreensão de que o paradigma de orientação a aspectos é uma solução para a problemática de abstração dos interesses da lógica de negócios e da lógica de suporte à aplicação que a orientação a objetos não consegue resolver de forma consistente, enquanto o paradigma da programação funcional atua principalmente como uma solução para minimizar o problema de fluxo de algoritmos, inerente às características herdadas da arquitetura de computadores desde a concepção dos micro controladores até a implementação das linguagens de programação (ver **BOX 8**).

Fica claro que a abordagem da programação funcional agrupa a eficiência em termos de fluxo de execução ao criar uma nova forma de estruturar os programas, adicionando os objetos a consistência do acoplamento de funções como forma de criar um *pipe* único que garanta a realização de uma tarefa do começo ao fim, assegurando a inexistência

de side-effects. Desta forma, vale a pena notar que a utilização de aspectos em uma solução não elimina as implicações oriundas de uma linguagem imperativa, com tipos fortes como o Java, ao passo que a programação funcional também não resolve a necessidade de modelos que consigam separar os interesses sistêmicos em aspectos individuais. O grande desafio no desenvolvimento em Java é unir os benefícios dos dois paradigmas para que os novos softwares se beneficiem tanto da performance em fluxo de execução quanto das melhores abstrações em termos de modelagem.

BOX 8.0 gargalo de von Neumann (von Neumann bottleneck)

O gargalo de von Neumann é uma limitação no sistema de I/O das CPUs baseadas na arquitetura de von Neumann. Esta limitação ocorre por causa do modo de processamento das CPUs, que realizam apenas um cálculo por vez (também conhecido como word-at-a-time) e, portanto, necessitam de muitas interações com a memória através do barramento para realizar tarefas mais elaboradas.

O gargalo ocorre porque o ciclo de acesso à memória é bem mais lento que o ciclo de execução da CPU, o que obriga esta a entrar em estado de espera durante o acesso àquela. Além disso, a leitura dos dados em memória é antecedida por interações para a obtenção dos endereços a serem acessados, o que agrava o problema, já que cada acesso a um dado em memória implica em pelo menos mais uma interação para a resolução do endereço a ser acessado.

Este problema foi apresentado por John Backus em sua palestra no evento Turing Award de 1977. Segundo Backus, o gargalo é a causa primária da ineficiência das linguagens de programação imperativas, já que elas carregam em suas semânticas a mesma dinâmica de processamento através de statements, expressões, transições de estados e incapacidade de representar propriedades matemáticas mais sofisticadas.

Ainda em sua palestra, Backus destaca a programação funcional como uma forma de minimizar o problema daquelas que ele chama de linguagens de von Neumann, tendo como princípio a ideia de que na programação funcional as expressões se arranjam em outras maiores, o que, no final, exige apenas um ciclo de computação para calcular um estado, minimizando o acesso à memória e ao mesmo tempo adicionando uma forma diferente de estruturar os programas de computadores.



Programação funcional aplicada com Aspectos

Como podemos ver, a união dos paradigmas de orientação a aspectos e de programação funcional pode ser alcançada através de Monads. Com esta abstração será possível alcançar melhores resultados na separação de interesses da aplicação e na manutenção do código, que se mostrará mais conciso. Contudo, não podemos nos limitar à utilização de Monads apenas em aspectos. No Java 8, é possível utilizar suas implementações (`java.util.Optional<T>` e `java.util.stream.Stream<T>`) para a criação de códigos totalmente funcionais que garantam a facilidade de manutenção, eliminação de pontos nulos e um fluxo livre de side-effects, entre outros benefícios.

A partir deste conhecimento, o desenvolvedor terá condições não somente de utilizar de forma consciente estes Monads, como também de criar as suas próprias implementações quando elas se fizerem necessárias. Vale a pena manter em mente que a utilização dos Monads nativos do Java 8 deve ser priorizada antes de se pensar em criar novas abstrações. Certamente será possível utilizá-los em grande parte das situações em que um Monad seja necessário.

Os Monads fornecidos nativamente pela API do Java 8 são abstrações puras. Portanto, nem sempre a sua utilização será possível como solução em toda implementação. Por isso, muitas vezes implementações impuras serão a solução mais viável para garantir um fluxo mais funcional e eficaz.

Atualmente já existem implementações de Monads bem conhecidas, como: List Monads, IO Monads e Maybe Monads.

Autor



Rômero Ricardo de Sousa Pereira

javeiro@uninove.edu.br e rpereira@atex.com



É formado em Ciência da Computação pela Universidade Nove de Julho, desde 2009, aonde atuou como professor em cursos extensivos de Java durante dois anos. Possui oito anos de experiência como desenvolvedor de sistemas, sendo quase sete atuando como desenvolvedor Java, passando por experiências desde o desenvolvimento de aplicações Standalones com Java SE, até soluções Web e Enterprise de alta demanda com Java EE. Atualmente é consultor Java na Atex Digital Media do Brasil. Possui as certificações OCJP e OCWCD e está cursando MBA em Desenvolvimento de Aplicações Java – SOA e Internet das Coisas (IoT) na FIAP.

Em linguagens puramente funcionais, como Haskell, essas implementações são facilitadas pelo suporte nativo. O contato e familiaridade com estas implementações será importante para que o leitor esteja apto a criar suas próprias abstrações em Java quando se fizerem necessárias.

Um bom exercício a partir deste novo conhecimento é a tentativa de utilizar o Monad criado neste artigo em um aspecto de advice `around()`, para que o desenvolvedor tenha a percepção de como a interceptação de um código pode conduzir a side-effects caso um `flatMap()` impuro não seja utilizado.

Links:

Orientação a Objetos.

<http://www.ipipan.gda.pl/~marek/objects/TOA/oobasics/oobasics.html>

Programação Orientada a Aspectos: AspectJ.

Programação Orientada a Aspectos com Java - Diogo Vinícius Winck | Vicente Goetten Junior [NOVATEC] - 2006

Can Programming be Liberated from the von Neumann style? A functional style and its Algebra of Programming.

<http://web.stanford.edu/class/cs242/readings/backus.pdf>

Monads for functional programming.

<http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf>

On Weaving Aspects.

<http://trese.cs.utwente.nl/aop-ecoop99/papers/boellert.pdf>

Monads, your App as a Function.

<http://mttkay.github.io/blog/2014/01/25/your-app-as-a-function/>

The Confinement Problem in the Presence of Faults.

<http://people.cs.missouri.edu/~harrisonwl/papers/icfem12.pdf>

Você gostou deste artigo?

Dê seu voto em www.devmedia.com.br/javamagazine/feedback

Ajude-nos a manter a qualidade da revista!



Somos tão apaixonados por tecnologia que o nome da empresa diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia. Somos uma equipe composta de gente que entende e gosta do que faz, assim como você.



Estrutura

100% NACIONAL.
Servidores de primeira linha, links de alta capacidade.



Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.



Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.



1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única. Conheça!



Porta 80

WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce |
Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486

DESENVOLVA PARA CORPORAÇÕES OU PARA O DIA A DIA COM JAVA

FORMAÇÃO DESENVOLVEDOR JAVA

DOMINE O DESENVOLVIMENTO DE APLICAÇÕES ORIENTADAS A OBJETO COM **UML** E APRENDA TÓPICOS AVANÇADOS DE PROGRAMAÇÃO COMO O USO DE **SERVLETS, JSP, FRAMEWORKS WEB** DENTRO DO PADRÃO **MVC**.

CERTIFICAÇÕES:

➤ Oracle Certified Professional, Java SE 6 Programmer

➤ Oracle Certified Professional, Java EE 5 Web Component Developer

